Roger S. Pressman

Sexta edición

INGENIERÍA DEL SOFTWARE

Un enfoque práctico

CONTENIDO BREVE

CONTENIDO

CAPÍTULO

1

SOFTWARE E INGENIERÍA DEL SOFTWARE

CONCEPTOS CLAVE

Es común darse cuenta que la invención de una tecnología puede tener efectos profundos e inesperados en otras tecnologías con las que en apariencia no tiene ninguna relación, como en empresas comerciales, en personas y aun en la cultura en su conjunto. Este fenómeno a menudo se denomina "la ley de las consecuencias imprevistas".

En la actualidad, el software de computadora es la tecnología individual más importante en el ámbito mundial. También es uno de los ejemplos principales de la ley de las consecuencias imprevistas. Nadie en la década de 1950 podría haber predicho que el software se convertiría en una tecnología indispensable en los negocios, la ciencia y la ingeniería; tampoco que el software permitiría la creación de tecnologías nuevas (por ejemplo, la ingeniería genética), la expansión de tecnologías existentes (como las telecomunicaciones), el fin de tecnologías antiguas (como la industria de la impresión); que el software sería la fuerza conductora detrás de la revolución de las computadoras personales; que los productos empaquetados de software se podrían comprar en los centros comerciales; que una compañía de software se volvería muy grande y más influyente que la mayoría de las compañías de la era industrial; que una gran red construida con software llamada Internet cubriría y cambiaría todo, desde la investigación bibliográfica hasta las compras de los consumidores y los hábitos diarios de los jóvenes (y no tan jóvenes).

Nadie podría haber previsto que el software estaría relacionado con sistemas de todo tipo: de transporte, médicos, de telecomunicaciones, militares, industriales, de entretenimiento, máquinas para oficina (la lista parece no tener fin).

UN VISTAZO RÁPIDO

¿Qué es? El software de computadora es el producto que los ingenieros de software construyen y después mantienen en el largo plazo. Incluye los programas que se ejecutan dentro de una computadora de cualquier tamaño y arquitectura, el contenido que se presenta conforme los programas se ejecutan y los documentos, tanto físicos como virtuales, que engloban todas las formas de medios electrónicos.

¿Quién lo hace? Los ingenieros de software lo construyen y lo mantienen, y casi todos en el mundo industrializado lo usan de manera directa o indirecta.

¿Por qué es importante? Porque afecta de forma muy cercana todos los aspectos de nuestras vidas y se ha vuelto omnipresente en el comercio, la cultura y las actividades cotidianas.

¿Cuáles son los pasos? El software de computadora se construye de la misma forma que cualquier producto de éxito: mediante la aplicación de un proceso que conduzca a un resultado de alta calidad que satisfaga las necesidades de la gente que usará el producto. Se aplica un enfoque de ingeniería del software.

¿Cuál es el producto obtenido? Desde el punto de vista del ingeniero de software, el producto obtenido lo forman los programas, el contenido (datos) y los documentos que constituyen el software. Pero desde el enfoque del usuario, el producto obtenido es la información resultante que de alguna manera mejora el mundo del usuario.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Una manera es leer el resto de este texto, seleccionar las ideas aplicables a un software específico y aplicarlas.

Y si se toma en cuenta la ley de las consecuencias imprevistas, hay muchos efectos que todavía es imposible predecir en el trabajo diario.

Por último, nadie podría haber predicho que millones de programas de computadora tendrían que corregirse, adaptarse y mejorarse conforme pasara el tiempo y que la labor de desarrollar estas actividades de "mantenimiento" absorbería más gente y recursos que todo el trabajo aplicado para la creación del software nuevo.

> "[illegible] y los descubrimientos tecnológicos son los elementos conductores del cr[illegible]"
>
> [illegible]

A medida que la importancia del software ha crecido, la comunidad del software ha intentado de manera continua desarrollar tecnologías que hagan más fácil, más rápida y menos cara la construcción y el mantenimiento de programas de computadora de alta calidad. Algunas de estas tecnologías se limitan al dominio de una aplicación específica (por ejemplo, al diseño y la implementación de sitios Web); otras se enfocan al dominio de una tecnología (como la programación orientada a objetos y la programación orientada a aspectos); y existen otras con base general (por ejemplo, sistemas operativos como LINUX). Sin embargo, aún no se desarrolla una tecnología de software que lo haga todo, y la probabilidad de que ésta surja en el futuro es pequeña. Aun así, las personas dejan sus trabajos, su seguridad y hasta sus vidas en manos del software de computadoras. Más vale que éste sea bueno.

Este texto presenta un marco para quienes construyen software de computadora: las personas que deben hacer buen software. El marco, que incluye un proceso, un conjunto de métodos y una serie de herramientas se llama *ingeniería del software*

> "[illegible] el papel de la ingeniería es [illegible] tuales de la vida humana, para que así la vida sea más fácil, segura y placentera."
>
> **Richard Fairley y Mary [illegible]**

1.1 EL PAPEL EVOLUTIVO DEL SOFTWARE

PUNTO CLAVE
El software es tanto un producto como el vehículo para su entrega.

En la actualidad, el software tiene un papel dual. Es, a la vez, un producto y un vehículo mediante el cual se entrega un producto. Como producto, ofrece la potencia de cómputo presentada como hardware de una computadora o, de manera más amplia, por una red de computadoras accesible mediante hardware local. Sin importar el lugar en que resida el software, ya sea en un celular o dentro de una computadora central, éste es un transformador de información; realiza la producción, el manejo, la adquisición, la modificación, el despliegue o la transmisión de la información que puede ser tan simple como un solo bit o tan compleja como una presentación multimedia. En su papel de vehículo para la entrega de un producto, el software actúa como la base para el control de la computadora (sistemas operativos), la comunicación de información (redes), y la creación y el control de otros programas (utilerías de software y ambientes).

El software entrega el producto más importante de nuestro tiempo: información. Transforma los datos personales (por ejemplo, las transacciones financieras de un individuo) de forma que los datos sean más útiles en un contexto local; maneja información de negocios para mejorar la competitividad, proporciona una vía para las redes de información alrededor del mundo (Internet) y proporciona los medios para adquirir información en todas sus formas

El papel del software de computadora ha experimentado un cambio significativo en un periodo un poco mayor a 50 años. Las mejorías sustanciales en el desempeño del hardware, los cambios profundos en las arquitecturas de cómputo, los enormes incrementos en las capacidades de memoria y almacenamiento, y la amplia variedad de opciones de salida y de entrada han propiciado el surgimiento de sistemas más elaborados y complejos basados en computadoras.

Los libros populares publicados durante las décadas de 1970 y 1980 ofrecen una amplia visión histórica de la cambiante percepción de las computadoras y del software y su impacto en la cultura. Osborne [OSB79] describió una "nueva Revolución Industrial". Toffler [TOF80] llamó al surgimiento de la microelectrónica parte de "la tercera ola del cambio" en la historia de la humanidad, y Naisbitt [NAI182] predijo la transformación de una sociedad industrial en una "sociedad de la información" Feigenbaum y McCorduck [FEI83] sugirieron que la información y el conocimiento (controlados por computadoras) serían el punto de enfoque para el poder en el siglo XXI, y Stoll [STO89] argumentó que la "comunidad electrónica" creada por redes y software era la clave del intercambio de conocimiento alrededor del mundo. Todos estos escritores tenían razón.

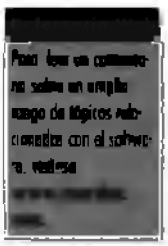
Si se tiene tiempo, revisar uno o más de estos libros clásicos. Pongan atención en las predicciones erróneas que estos expertos hicieron en lo referente a eventos y tecnologías. Consérvese la humildad: nadie sabe en realidad el futuro de los sistemas que se construyen.

Al comienzo de la década de 1990, Toffler [TOF90] describió un "cambio de poder" en el que todas las viejas estructuras (gubernamentales, educativas, industriales, económicas y militares) se desintegrarían a medida que las computadoras y el software condujeran a una "democratización del conocimiento". Yourdon [YOU92] se preocupaba de que las compañías estadounidenses pudieran perder su margen competitivo en negocios relacionados con el software y predijo "la declinación y caída del programador estadounidense". Hammer y Champy [HAM93] argumentaban que las tecnologías de información representarían un papel primordial en la "reingeniería de la corporación". A mediados de la década de 1990 la penetración de las computadoras y del software provocó el surgimiento de una serie de libros de "neoluditas" (como *Resisting the Virtual Life*, editado por James Brook y Iain Boal, y *The Future Does Not Compute*, de Stephen Talbot). Estos autores satanizaban a la computadora al enfatizar inquietudes legítimas, pero ignorando los grandes beneficios que ya se habían obtenido [LEV95].

> "Las computadoras facilitan la realización de muchas cosas, pero la mayoría de las cosas que facilitan no necesitan hacerse."
>
> **Andy Rooney**

A finales de la década de 1990, Yourdon [YOU96] evaluó de nuevo a los candidatos a profesionales del software y sugirió el "surgimiento y resurrección" del programador estadounidense. A medida que Internet cobraba mayor importancia, el giro que había dado Yourdon parecía ser el correcto. Al finalizar el siglo xx, el enfoque cambió nuevamente, esta vez con el impacto del Y2K, "bomba de tiempo" (por ejemplo [YOU98a], [KAR99]). Aunque las fatales predicciones de aquellos que vislumbraban una catástrofe respecto al Y2K fueron falsas, sus populares escritos acarrearon la permanencia del software en la vida de los seres humanos.

Ya iniciado el nuevo siglo, Johnson [JOH01] explicó el poder del "surgimiento" como un fenómeno que explica lo que sucede cuando interconexiones presentes en entidades relativamente simples resultan en un sistema que "se autoorganiza para formar un comportamiento más adaptable e inteligente". Yourdon [YOUR02] retomó los trágicos sucesos ocurridos el 11 de septiembre de 2001 en Nueva York para explicar el impacto continuo del terrorismo global en la comunidad informática. Wolfram [WOL02] presentó un tratado sobre "un nuevo tipo de ciencia" en donde expone una teoría unificadora basada sobre todo en elaboradas simulaciones de software. Daconta y sus colegas [DAC03] explicaron la evolución de la "red semántica", y cómo esto cambiará el modo en que la gente interactúa a través de las redes globales



En la actualidad una enorme industria del software se ha convertido en un factor dominante en la economía del mundo industrializado. El programador solitario de la era inicial ha sido sustituido por equipos de especialistas en software, en los que cada uno se enfoca en una parte de la tecnología requerida para desarrollar una aplicación compleja. Hasta ahora, las preguntas formuladas al programador solitario son las mismas que se hacen cuando se construyen los sistemas basados en computadoras modernas:[1]

- ¿Por qué tarda tanto la obtención del software terminado?
- ¿Por qué son tan altos los costos de desarrollo del software?
- ¿Por qué es imposible encontrar todos los errores en el software antes de entregarlo a los clientes?

1 En un excelente libro de ensayos sobre el negocio del software, Tom DeMarco [DEM95] expone la idea contraria. Explica: "En vez de cuestionarse por qué cuesta tanto el software, uno debe preguntarse qué se ha hecho para hacer que hoy el software cueste tan poco. La respuesta a esa pregunta ayudará a continuar con el extraordinario nivel de logros que siempre ha distinguido a la industria del software".

- ¿Por qué se gastan tanto tiempo y esfuerzo en el mantenimiento de los programas existentes?
- ¿Por qué es difícil medir el progreso al desarrollar y darle mantenimiento al software?

Éstas y muchas otras preguntas demuestran la preocupación de la industria por el software y por la manera en que éste se desarrolla; una preocupación que ha conducido a la adopción de la práctica de la ingeniería del software.

1.2 SOFTWARE

En 1970, menos del uno por ciento de las personas podrían haber definido lo que significaba "software de computadora". En la actualidad, la mayoría de los profesionales y muchos miembros del público creen que entienden el software. Pero, ¿en realidad lo hacen?

¿Cómo debe definirse el software?

Una definición de software en un libro de texto puede tener la siguiente forma: *el software se forma con 1) las instrucciones (programas de computadora) que al ejecutarse proporcionan las características, funciones y el grado de desempeño deseados; 2) las estructuras de datos que permiten que los programas manipulen información de manera adecuada; y 3) los documentos que describen la operación y el uso de los programas.* No existe duda de que se pueden encontrar definiciones más completas. Pero se requiere más que una definición formal.

Para entender el software (y la ingeniería del software), es importante examinar las características que lo hacen diferente de otras cosas que construye el ser humano. El software es un elemento lógico, en lugar de físico, de un sistema. Por lo tanto, el software tiene características muy diferentes a las del hardware.

PUNTO CLAVE

El software se desarrolla, no se manufactura.

1. *El software se desarrolla o construye; no se manufactura en el sentido clásico.*

   A pesar de que existen similitudes entre el desarrollo del software y la manufactura del hardware, las dos actividades son diferentes en lo fundamental. En ambas, la alta calidad se alcanza por medio del buen diseño, pero la fase de manufactura del hardware puede incluir problemas de calidad inexistentes (o que son fáciles de corregir) en el software. Ambas actividades dependen de las personas, pero la relación entre la gente utilizada y el trabajo realizado es diferente por completo (véase el capítulo 24). Ambas actividades requieren la construcción de un "producto", pero los enfoques son diferentes. Los costos del software se concentran en la ingeniería. Esto significa que los proyectos de software no se pueden manejar como si fueran proyectos de manufactura.

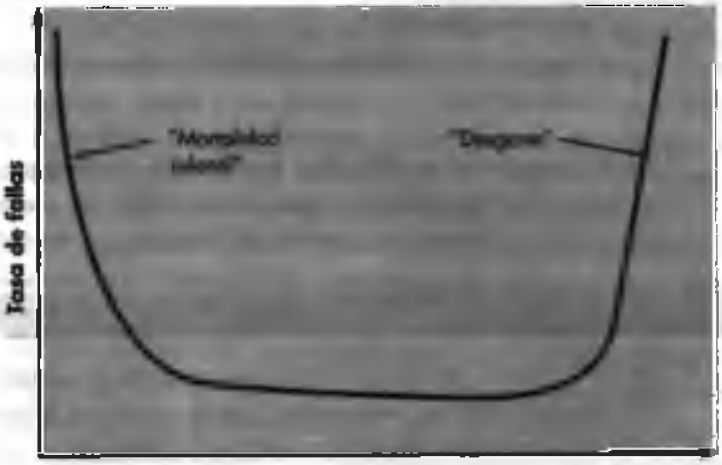
PUNTO CLAVE

El software no se desgasta, pero se deteriora.

2. *El software no se "desgasta".*

   En la figura 1.1 se muestra, para el hardware, la tasa de fallas como una función del tiempo. La relación, llamada a menudo "curva de la bañera", indica

FIGURA 1.1
Curva de fallas para el hardware.



que el hardware tiene un número considerablemente alto de fallas al inicio de su vida (a menudo éstas se atribuyen a defectos de diseño o manufactura). Después, los defectos se corrigen y la tasa de fallas baja hasta un nivel estable (se desea que éste sea muy bajo) por algún periodo. Sin embargo, conforme pasa el tiempo, la tasa de fallas se eleva de nuevo conforme los componentes del hardware sufren los efectos acumulativos del polvo, la vibración, el abuso, las temperaturas extremas y muchos otros males ambientales. Expresado en forma más simple, el hardware comienza a *desgastarse*.

CONSEJO

Si se desea reducir el deterioro del software, es necesario realizar un mejor diseño (capítulos 9-12).

El software es inmune a los males ambientales que desgastan el hardware Por lo tanto, la curva de la tasa de fallas para el software debería tener la forma de la "curva idealizada" que se muestra en la figura 1.2. Los defectos sin descubrir causan tasas de falla altas en las primeras etapas de vida de un programa. Sin embargo, los errores se corrigen (en el mejor de los casos sin agregar otros errores) y la curva se aplana como se muestra en la figura 1.2 La curva idealizada es una simplificación burda del modelo de fallas real para el software (para más información véase el capítulo 26). Sin embargo, la implicación es clara: el software no se desgasta, pero sí se *deteriora*.

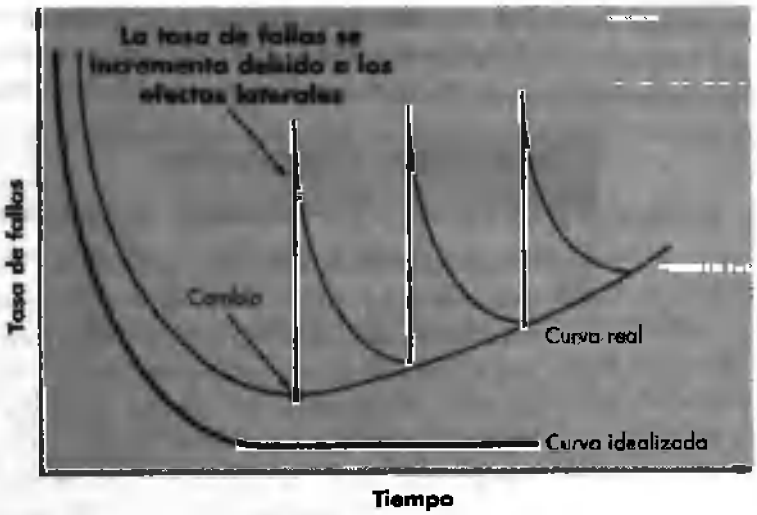
PUNTO CLAVE

Los métodos de la ingeniería del software pretenden reducir la magnitud de los picos y la pendiente de la curva real que se muestra en la figura 1.2.

Esta contradicción aparente se puede explicar de mejor manera si se considera la "curva real" de la figura 1.2. Durante su vida,[2] el software experimenta cambios. Conforme éstos ocurren se presenta la posibilidad de introducir errores, lo que ocasiona que la curva de fallas tenga un pico, como se muestra en la figura 1.2 Antes de que la curva pueda regresar a su estado original con una tasa de fallas estable, se requiere otro cambio, lo que ocasiona que la

2 De hecho, desde el momento en que comienza el desarrollo, y mucho antes de que se entregue la primera versión, el cliente puede solicitar cambios

FIGURA 1.2
Curvas de falla para el software.



PUNTO CLAVE

La mayor parte del software aún se construye a la medida del cliente.

curva tenga otro pico. De esta manera, el nivel de fallas mínimo se comienza a elevar; el software se deteriora debido a los cambios.

Otro aspecto del desgaste ilustra la diferencia entre el hardware y el software. Cuando un componente del hardware se desgasta se sustituye con un repuesto. Pero en el software no existen repuestos. Cualquier falla del software implica un error en el diseño o el proceso mediante el cual se pasó del diseño al código máquina ejecutable. Por lo tanto, el mantenimiento del software implica de manera considerable una complejidad mayor que el del hardware.

3. *A pesar de que la industria tiene una tendencia hacia la construcción por componentes, la mayoría del software aún se construye a la medida*

Considérese la forma en que se diseña y construye un hardware de control para un producto de cómputo. El ingeniero de diseño dibuja un esquema simple del sistema de circuitos digital, realiza algunos análisis fundamentales para asegurarse de que el diseño realizará las funciones apropiadas y después busca en los catálogos de componentes digitales cada circuito integrado de acuerdo con un número de parte, una función definida y validada, una interfaz bien definida y un conjunto estandarizado de directrices de integración. Una vez seleccionado cada componente, puede solicitársele para después ensamblarlo.

Cuando una disciplina de ingeniería evoluciona se crea una colección de diseños estándar de componentes. Los tornillos y los circuitos integrados son sólo dos ejemplos de los miles de componentes estándar que utilizan los ingenieros mecánicos y eléctricos al diseñar sistemas nuevos. Los componentes reutilizables se han creado para que el ingeniero se pueda concentrar en los

elementos que en realidad son innovadores en el diseño; es decir, en las partes que representan algo nuevo. En el mundo del hardware, la reutilización de componentes es una parte natural del proceso de ingeniería. En el ámbito del software, dicha actividad apenas se ha comenzado a extender.

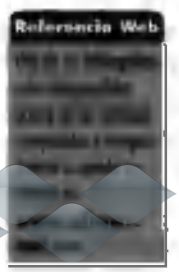
> "Las ideas son los bloques de construcción de las ideas."
>
> **Jason Zabalaoy**

Un componente de software se debe diseñar e implementar de forma que pueda utilizarse en muchos programas diferentes. Los componentes reutilizables modernos encapsulan tanto los datos como el proceso que se aplica a éstos, lo que permite al ingeniero de software crear aplicaciones nuevas a partir de partes reutilizables.[3] Por ejemplo, las interfaces actuales con el usuario se construyen con componentes reutilizables que permiten la creación de ventanas gráficas, menús desplegables y una amplia variedad de mecanismos de interacción. Las estructuras de datos y los detalles de procesamiento requeridos para construir la interfaz están contenidos en una librería de componentes reutilizables para la construcción de la interfaz

1.3 LA NATURALEZA CAMBIANTE DEL SOFTWARE

En la actualidad existen siete grandes categorías del software de computadora que presentan retos continuos para los ingenieros de software.

Software de sistemas. El software de sistemas es una colección de programas escritos para servir a otros programas. Algunos programas de sistemas (como los compiladores, editores y utilerías para la administración de archivos) procesan estructuras de información complejas pero determinadas.[4] Otras aplicaciones de sistemas (por ejemplo, componentes del sistema operativo, controladores, software de red, procesadores para telecomunicaciones) procesan datos indeterminados. En cada caso, el área de software de sistemas se caracteriza por una interacción muy intensa con el hardware de la computadora; utilización por múltiples usuarios; operación concurrente que requiere la gestión de itinerarios, de compartición de recursos, y de procesos sofisticados; estructuras de datos complejas y múltiples interfaces externas.



Software de aplicación. El software de aplicación consiste en programas independientes que resuelven una necesidad de negocios específica. Las aplicaciones en

3 La ingeniería del software basado en componentes se presenta en el capítulo 30

4 El software es *determinado* si el orden y el ritmo de las entradas, el procesamiento y las salidas son predecibles. El software es *indeterminado* si el orden y el ritmo de las entradas, el procesamiento y las salidas no se pueden predecir.

esta área procesan datos empresariales o técnicos de forma que facilitan las operaciones de negocios o la toma de decisiones técnicas o de gestión. Además del procesamiento de datos convencional, el software de aplicación se utiliza para controlar las funciones de negocios en tiempo real (por ejemplo, el procesamiento de transacciones en los puntos de venta y el control de procesos de manufactura en tiempo real.)

Software científico y de ingeniería. El software científico y de ingeniería, que se caracterizaba por algoritmos "devoradores de números", abarca desde la astronomía hasta la vulcanología, desde el análisis de la tensión automotriz hasta la dinámica orbital de los transbordadores espaciales, y desde la biología molecular hasta la manufactura automatizada Sin embargo, las aplicaciones modernas dentro del área científica y de ingeniería se alejan en la actualidad de los algoritmos numéricos convencionales. El diseño asistido por computadora, la simulación de sistemas y otras aplicaciones interactivas han comenzado a tomar características de software en tiempo real e incluso de software de sistemas

Software emportado. El software emportado reside dentro de la memoria de sólo lectura del sistema y con él se implementan y controlan características y funciones para el usuario final y el sistema mismo. El software incrustado puede desempeñar funciones limitadas y curiosas (como el control del teclado de un horno de microondas) o proporcionar capacidades de control y funcionamiento significativas (por ejemplo, las funciones digitales de un automóvil, como el control de combustible, el despliegue de datos en el tablero, los sistemas de frenado, etcétera)

Software de línea de productos. El software de línea de productos, diseñado para proporcionar una capacidad específica y la utilización de muchos clientes diferentes, se puede enfocar en un nicho de mercado limitado (como en los productos para el control de inventarios) o dirigirse hacia los mercados masivos (por ejemplo, aplicaciones de procesadores de palabras, hojas de cálculo, gráficas por computadora, multimedia, entretenimiento, manejo de bases de datos, administración de personal y finanzas en los negocios).

Aplicaciones basadas en Web. Las "WebApps" engloban un espectro amplio de aplicaciones En su forma más simple, las WebApps son apenas un poco más que un conjunto de archivos de hipertexto ligados que presenta información mediante texto y algunas gráficas. Sin embargo, a medida que el comercio electrónico y las aplicaciones B2B adquieren mayor importancia, las WebApps evolucionan hacia ambientes computacionales sofisticados que no sólo proporcionan características, funciones de cómputo y contenidos independientes al usuario final, sino que están integradas con bases de datos corporativas y aplicaciones de negocios.

Software de inteligencia artificial. Este software utiliza algoritmos no numéricos en la resolución de problemas complejos que es imposible abordar por medio de un análisis directo. Las aplicaciones dentro de esta área incluyen la robótica, los sis-

temas expertos, el reconocimiento de patrones (imagen y voz), las redes neuronales artificiales, la comprobación de teoremas y los juegos en computadora.

> "No existe una computadora que tenga sentido común."
>
> **Marvin Minsky**

Existen millones de ingenieros de software que trabajan duro en una o más de estas categorías. En algunos casos se construyen sistemas nuevos, pero en otros las aplicaciones existentes se corrigen, adaptan y mejoran. Es común ver a un joven ingeniero de software que trabaja en programas más viejos que él mismo. Las generaciones pasadas de creadores de software han dejado un legado en cada una de las categorías que se han definido párrafos atrás. Se espera que el legado de la generación actual facilite la tarea de los ingenieros de software del futuro. No obstante, en el horizonte han aparecido retos nuevos:

Computación ubicua. El crecimiento rápido de las redes inalámbricas podría conducir pronto a la verdadera computación distribuida. El reto para los ingenieros de software será desarrollar software de sistema y de aplicación que permita que dispositivos pequeños, computadoras personales y sistemas de empresa se comuniquen a través de grandes redes.

Alimentación de la red. La World Wide Web se convierte con rapidez en un dispositivo computacional, así como en un proveedor de contenido. El reto para los ingenieros de software es crear aplicaciones simples (por ejemplo, planeación de las finanzas personales) y complejas que beneficien a mercados de usuarios finales específicos alrededor del mundo.

> "No siempre es posible predecir, pero siempre es posible prepararse."
>
> **Anónimo**

Fuente abierta. Existe una tendencia creciente que impulsa la distribución del código fuente para aplicaciones de sistemas (como sistemas operativos, bases de datos y ambientes de desarrollo) de forma que los clientes hagan modificaciones locales. El reto para los ingenieros de software es construir un código fuente que sea descriptivo en sí mismo, pero, aún más importante, desarrollar técnicas que permitan tanto a los clientes como a los diseñadores conocer los cambios realizados y la forma en que se manifiestan dentro del software.

La "nueva economía". La locura del punto-com que se afianzó en los mercados financieros hacia finales de la década de 1990 y la subsiguiente ruptura en los primeros años del siglo XXI ha llevado a mucha gente de negocios a creer que la nueva economía está muerta. La nueva economía está viva y saludable, pero evolucionará con lentitud; la caracterizará la comunicación y la distribución masiva. Andy Lippman [LIP02] puntualiza esta situación cuando escribe:

> Estamos entrando en una era caracterizada por las comunicaciones entre las máquinas distribuidas y la gente dispersa, en lugar de la que define una conexión entre dos individuos o entre un individuo y una máquina. El antiguo enfoque de la telefonía se refiere a "conexiones con"; la siguiente ola se refiere a "conexiones entre". Por mencionar algunos ejemplos, se tiene Napster, la mensajería instantánea, los sistemas de mensaje cortos y las BlackBerries.

El reto para los ingenieros de software es construir aplicaciones que faciliten la comunicación y la distribución de productos en masa mediante productos apenas en formación.

Cada uno de estos "nuevos retos" obedecerá sin duda la ley de las consecuencias imprevistas y tendrán efectos (para la gente de negocios, los ingenieros de software y los usuarios finales) que no pueden predecirse en la actualidad. Sin embargo, los ingenieros de software se pueden preparar al iniciar un proceso que tenga la suficiente agilidad y adaptabilidad como para acoplarse a los cambios drásticos en la tecnología y las reglas de negocios que con seguridad se presentarán en la década siguiente.

> "[La] computadora por sí misma hará una transición histórica de algo que se usa para tareas analíticas... a algo que puede provocar emociones."
>
> **David Vaskevitch**

3.4 SOFTWARE HEREDADO

¿Qué es el software heredado?

Existen cientos de miles de programas de computadora y todos pertenecen a uno de los siete grandes dominios de aplicación —software de sistemas, software de aplicación, software científico y de ingeniería, software empotrado, software de producto, WebApps y aplicaciones IA— que se expusieron en la sección 1.3. Algunos de estos programas son de vanguardia —sólo divulgados entre ciertas personas, industrias y gobiernos—, pero otros son más viejos, y en algunos casos *mucho* más viejos.

Estos programas viejos —con frecuencia referidos como *software heredado*— han sido el foco de atención y preocupación continua desde la década de 1960. Dayani-Fard y sus colegas [DAY99] describen el software heredado de la siguiente forma:

> Los sistemas de software heredado ... fueron desarrollados hace décadas y han sido modificados en forma continua para cumplir los requerimientos de los cambios en los negocios y en las plataformas de cómputo. La proliferación de dichos sistemas ha causado dolores de cabeza a las grandes organizaciones, las cuales los perciben como costosos en su mantenimiento y riesgosos en su evolución.

Liu y sus colegas [LIU98] extendieron esta descripción al escribir que "muchos sistemas heredados persisten como el soporte de las funciones centrales de negocios y son indispensables para las empresas". Por lo tanto, al software heredado lo caracterizan su longevidad y el ser crítico para los negocios.

1.4.1 Calidad del software heredado

¿Qué se debe hacer si se tiene un software heredado con poca calidad?

Por desgracia, existe una característica adicional que tal vez esté presente en el software heredado: *poca calidad*.[5] Algunas veces, los sistemas heredados tienen diseños imposibles de extender, código complicado, documentación escasa o inexistente, casos de prueba y resultados que nunca fueron archivados, un historial de cambio manejado con pobreza, etcétera; la lista podría seguir hasta tener una longitud considerable. No obstante, estos sistemas son el soporte de "las funciones centrales de negocios y son indispensables para las empresas" [LIU98]. ¿Qué se puede hacer?

La única respuesta razonable podría ser no hacer nada, al menos hasta que el sistema heredado experimente algún cambio significativo. Pero si satisface las necesidades de sus usuarios y funciona de manera confiable, el sistema no está roto y no requiere arreglos. Sin embargo, conforme pasa el tiempo, los sistemas heredados evolucionan por una o más de las razones siguientes:

¿Cuáles son los tipos de cambios que se realizan sobre el software heredado?

- El software debe adaptarse para satisfacer las necesidades de los nuevos ambientes o las nuevas tecnologías de cómputo.
- El software debe mejorarse para implementar los nuevos requerimientos de los negocios.
- El software debe extenderse para hacerlo operable con sistemas y bases de datos más modernos.
- El software debe rediseñarse para hacerlo viable dentro de un ambiente de red.

Cuando suceden estas formas de evolución en un software heredado, éste debe someterse a una reingeniería (capítulo 31) de modo que conserve su viabilidad en el futuro. La meta de la ingeniería de software moderna es "imaginar metodologías que se basen en la noción de la evolución"; esto es, la noción de que "los sistemas de software cambian de manera continua, los nuevos sistemas de software se construyen a partir de los viejos, y... todos deben interactuar y cooperar con los demás" [DAY99].

CONSEJO

Cualquier ingeniero de software debe reconocer que el cambio es natural. No debe intentar combatirlo.

1.4.2 Evolución del software

El software de computadora evoluciona a través del tiempo, sin importar su dominio de aplicación, tamaño o complejidad. El cambio (que con frecuencia es llamado *mantenimiento del software*) conduce este proceso, y se presenta cuando se corrigen errores, cuando el software se adapta a un nuevo ambiente, cuando el cliente solicita características o funciones nuevas, y cuando la aplicación experimenta una reingeniería para proporcionar beneficios en un contexto moderno. Sam Williams [WIL02] refiere esta situación cuando escribe:

5 En este caso, la calidad se juzga con base en el pensamiento moderno de la ingeniería del software, que en cierto modo es un criterio injusto, puesto que algunos conceptos y principios modernos de la ingeniería del software aún no habían sido bien entendidos cuando se desarrolló el software heredado.

> Debido a que los programas a gran escala como Windows y Solaris se expanden bien en el intervalo de 30 a 50 millones de líneas de código, los administradores de proyecto exitosos han aprendido a dedicar tanto tiempo a combinar los enredos de nuestro código heredado como a agregar código nuevo. Para decirlo de manera más simple, en una década en la que el desempeño promedio del microchip de PC se incrementó cien veces, la incapacidad de escalar el software incluso a tasas lineales ha pasado de un pequeño secreto a una enorme alteración en toda la industria.

En los últimos 30 años, Manny Lehman [LEH97a] y sus colegas han analizado en forma detallada la industria del software y los sistemas en un esfuerzo dirigido a desarrollar una *teoría unificada para la evolución del software*. Los detalles de dicho trabajo superan el enfoque del presente texto,[6] pero las leyes subyacentes derivadas de su estudio son dignas de destacarse [LEH97b]:

La ley del cambio continuo (1974). Los sistemas de tipo electrónico[7] deben adaptarse en forma continua, de lo contrario se volverán menos satisfactorios a través del tiempo.

La ley de la complejidad creciente (1974). Cuando un sistema de tipo electrónico está en evolución, su complejidad se incrementa a menos que se realice el trabajo necesario para mantenerla o reducirla.

La ley de la autorregulación (1974). El proceso de evolución de un sistema de tipo electrónico se autorregula con la distribución del producto y las mediciones del proceso cercanas a la normal.

¿Por qué los sistemas heredados evolucionan con el paso del tiempo?

La ley de la conservación de la estabilidad organizacional (1980). La tasa de actividad global efectiva promedio en un sistema de tipo electrónico en evolución no varía a lo largo del periodo de vida del producto.

La ley de la conservación de la familiaridad (1980). Cuando un sistema de tipo electrónico está en evolución y se quiere tener un desarrollo satisfactorio, todos los involucrados con el sistema, como los desarrolladores, el personal de ventas y los usuarios, deben mantener el dominio sobre su contenido y comportamiento. El crecimiento excesivo disminuye ese dominio. Por tanto, el crecimiento promedio permanece sin cambio durante la evolución del sistema.

La ley del crecimiento continuo (1980). El contenido funcional de los sistemas de tipo electrónico debe incrementarse en forma continua para mantener la satisfacción del usuario a lo largo del periodo de vida del sistema.

La ley de la calidad decreciente (1996). La calidad de los sistemas de tipo electrónico parecerá declinar a menos que éstos se mantengan y adapten en forma rigurosa de acuerdo con los cambios en su ambiente operacional.

6 Para una clara explicación de la evolución del software, el lector interesado puede revisar [LEH97a].

7 Los *sistemas de tipo electrónico* son programas de software que han sido implementados en un contexto computacional del mundo real y que, por tanto, evolucionarán a través del tiempo.

La ley del sistema de retroalimentación (1996). Los procesos de evolución de los sistemas de tipo electrónico constituyen sistemas de retroalimentación con niveles, ciclos y agentes múltiples, y deben tratarse de forma que se obtengan mejorías significativas sobre cualquier base razonable.

Las leyes que Lehman y sus colegas han definido son una parte inherente de la realidad de un ingeniero de software. En lo sucesivo, en este texto se discutirán modelos para el proceso del software, métodos de ingeniería de software y técnicas de gestión que pretenden mantener la calidad del software mientras éste se encuentra en evolución.

1.5 MITOS DEL SOFTWARE

Los mitos del software —creencias acerca del software y de los procesos empleados para construirlo— se pueden rastrear hasta los primeros días de la computación. Los mitos tienen ciertos atributos que los convierten en insidiosos. Por ejemplo, los mitos parecen una relación de hechos razonables (algunas veces contienen elementos verdaderos), se observan de manera intuitiva, y con frecuencia los promulgan practicantes experimentados, quienes "conocen el terreno".

> "En ausencia de normas significativas, una industria nueva como el software suele depender de las costumbres."
>
> Tom DeMarco

En la actualidad, la mayoría de los profesionales reconocidos en la ingeniería del software identifican los mitos en su real dimensión: actitudes equivocadas que han causado problemas serios a los administradores y al personal técnico por igual. Sin embargo, las antiguas actitudes y viejos hábitos son difíciles de modificar, por lo que aún subsisten creencias falsas sobre el software.

Referencia Web
La red de administradores de proyectos de software puede ayudar a enfrentar con éstos y otros mitos. Dicha red se puede encontrar en www.spmn.com.

Mitos de la administración. Los administradores con responsabilidades sobre el software, al igual que sus pares en la mayoría de las disciplinas, a menudo están bajo presión por mantener los presupuestos, evitar que los itinerarios se extiendan y mejorar la calidad. De la misma forma que una persona a punto de ahogarse se aferra a un tronco, con frecuencia el administrador del software se aferra a un mito si siente que esta creencia reducirá la presión (aun en forma temporal).

Mito: *Ya se tiene un libro lleno de estándares y procedimientos para la construcción de software. ¿Esto proporcionará a mi gente todo el conocimiento necesario?*

Realidad: Tal vez sea verdad que el libro de estándares existe, pero ¿se usa? ¿Los encargados de la construcción del software saben de su existencia? ¿El libro refleja la práctica moderna de la ingeniería del software? ¿Está completo? ¿Es adaptable? ¿Está dirigido al mejoramiento del tiem-

po de entrega sin dejar de enfocarse en la calidad? En muchos casos la respuesta a todas estas preguntas es no.

Mito: *Si se está atrasado en el itinerario es posible contratar más programadores para así terminar a tiempo (algunas veces llamado el concepto de la horda mongola)*

Realidad: El desarrollo de software no es un proceso mecánico como la manufactura. En palabras de Brooks [BRO75]: "Agregar gente a un proyecto de software atrasado lo atrasa más" De inicio, este enunciado podría parecer contrario a la intuición. Sin embargo, cuando se agregan nuevos integrantes a un equipo la gente que ya estaba trabajando debe invertir tiempo en la enseñanza a los recién llegados, lo cual reduce el tiempo dedicado al esfuerzo para el desarrollo productivo. Se puede agregar gente, pero sólo de una manera planeada y bien coordinada

Mito: *Si decido subcontratar el proyecto de software a un tercero, puedo relajarme y dejar que esa compañía lo construya*

Realidad: Si una organización no entiende cómo administrar y controlar internamente los proyectos de software, de manera invariable entrará en conflicto al subcontratar este tipo de proyectos

CONSEJO

Es necesario trabajar duro para entender qué se debe hacer antes de comenzar. En ocasiones no es posible desarrollar todos los detalles, pero entre más se sepa, menor es el riesgo que se corre.

Mitos del cliente. El cliente que solicita un software de computadora puede ser la persona del escritorio de al lado, un grupo técnico en el piso de abajo, el departamento de ventas o de mercadotecnia, o una compañía externa que ha solicitado el software bajo contrato. En muchos casos, el cliente cree en mitos acerca del software porque los profesionales y administradores del software hacen muy poco para corregir la desinformación. Los mitos conducen a expectativas falsas (del cliente) y en definitiva a insatisfacción con el desarrollador.

Mito: *Un enunciado general de los objetivos es suficiente para comenzar a escribir programas; los detalles se pueden afinar después.*

Realidad: A pesar de que no siempre es factible que el enunciado de los requerimientos sea comprensible y estable, un enunciado ambiguo de los objetivos es la receta perfecta para el desastre. Los requerimientos precisos (los cuales se derivan usualmente en forma iterativa) se desarrollan sólo mediante la comunicación continua y efectiva entre el cliente y el desarrollador.

Mito: *Los requerimientos del proyecto cambian de manera continua, pero el cambio puede ajustarse con facilidad porque el software es flexible.*

Realidad: Es verdad que los requerimientos del software cambian, pero el impacto del cambio varía de acuerdo con el momento en que éste se introduce. Cuando los cambios en los requerimientos se solicitan en

etapas tempranas (antes de iniciar con el diseño o el código), el impacto en el costo es relativamente pequeño.[8] Sin embargo, conforme pasa el tiempo, el impacto en el costo crece con rapidez —se han distribuido los recursos, se ha establecido un marco general para el diseño— y el cambio puede provocar una convulsión que requiera recursos adicionales y una modificación significativa en el diseño.

Mitos del desarrollador. Los mitos que aún subsisten entre los desarrolladores del software han permanecido a través de 50 años de cultura de programación. Durante los primeros años del software, la programación era vista como una forma de arte; por ello, las viejas formas y actitudes son difíciles de eliminar.

Mito: *Una vez que el programa ha sido escrito y puesto a funcionar, el trabajo está terminado*

Realidad: Alguien dijo alguna vez que entre más rápido se comience a escribir código, más tiempo pasará para que el programa esté terminado. Los datos de la industria indican que entre 60 y 80 por ciento de todo el esfuerzo aplicado en el software se realizará después de que el sistema haya sido entregado al cliente por primera vez.

CONSEJO

Siempre que se piense que no hay tiempo para la ingeniería del software, se debe considerar si habrá tiempo para hacerlo todo de nuevo.

Mito: *Mientras el programa no se esté ejecutando, no existe forma de evaluar su calidad.*

Realidad: Uno de los mecanismos más efectivos para el aseguramiento de la calidad del software se puede aplicar desde el inicio de un proyecto· la revisión técnica formal. Las revisiones al software (descritas en el capítulo 26) son un "filtro de calidad" que han probado ser más efectivas que las pruebas para encontrar ciertas clases de errores en el software.

Mito: *El único producto del trabajo que puede entregarse para tener un proyecto exitoso es el programa en funcionamiento.*

Realidad: Un programa en funcionamiento es sólo una parte de la configuración del software que incluye muchos elementos. La documentación proporciona un fundamento para la ingeniería exitosa y, aún más importante, representa una guía para el mantenimiento del software.

Mito: *La ingeniería del software obligará a emprender la creación de una documentación voluminosa e innecesaria y de manera invariable tornará más lento el proceso.*

Realidad: La ingeniería del software no se refiere a la elaboración de documentos. Está relacionada con la creación de calidad. Una mejor calidad

8 Muchos ingenieros de software han adoptado un enfoque "ágil" que adapta los cambios en forma incremental, con lo que se controla su impacto y costo. Los métodos ágiles se exponen en el capítulo 4.

conduce a la reducción de los trabajos redundantes. Y una menor cantidad de trabajos redundantes resulta en menores tiempos de entrega.

Muchos profesionales de los sistemas reconocen la falacia de los mitos del software. Por el contrario, las actitudes y los métodos habituales conducen a adoptar malas prácticas administrativas y técnicas, a pesar de que la realidad exige un mejor enfoque. El reconocimiento de las realidades del software es el primer paso hacia la formulación de soluciones prácticas para la ingeniería del software.

1.6 CÓMO INICIA TODO

Cualquier proyecto de software se inicia por alguna necesidad de negocios: la necesidad de corregir un defecto en una aplicación existente; el imperativo de adaptar un sistema heredado a un ambiente de negocios cambiante; el requerimiento de extender las funciones y características de una aplicación existente, o la necesidad de crear un producto, servicio o sistema nuevos.

Con frecuencia, en el inicio de un proyecto de ingeniería del software la necesidad de negocios se expresa de manera informal durante una simple conversación. En el recuadro que está abajo se presenta una conversación típica.

Con excepción de una referencia pasajera, el software no se mencionó durante la conversación. Aun así, el software hará la diferencia en el futuro de la línea de productos *HogarSeguro*. El mercado aceptará el producto sólo si el software incrustado en él satisface de manera apropiada las necesidades del cliente (que aún no ha sido definido). En los capítulos subsecuentes se dará seguimiento a la ingeniería del software en *HogarSeguro*.

HOGARSEGURO[9]

Cómo se inicia un proyecto

La escena: Sala de juntas en CPI [illegible], una compañía (ficticia) que fabrica productos de [illegible] para uso comercial y doméstico.

Los actores: Mal Golden, gerente general, desarrollo de productos; Lisa Pérez, gerente de mercadotecnia; Lee Warren, gerente de ingeniería; Joe Camalleri, vicepresidente ejecutivo, desarrollo de negocios.

La conversación:

Joe: Dime Lee, ¿de qué se trata ese asunto que escuché? ¿Tu equipo está desarrollando un qué? ¿Una caja inalámbrica genérica universal?

Lee: Es genial, como del tamaño de una caja de cerillos, se puede conectar a sensores de todos los tipos, una cámara digital, casi a cualquier cosa. Usando el protocolo 802.11b inalámbrico

9 El proyecto *HogarSeguro* se usará a lo largo de este texto para ilustrar los trabajos internos de un equipo de proyecto, mientras éste construye un producto de software. La compañía, el proyecto y las personas son ficticios, pero las situaciones y los problemas son reales.

Nos permite tener acceso a la salida del dispositivo sin usar cables. Pensamos que nos llevará a una nueva generación de productos.

Joe: ¿Estás de acuerdo, Mal?

Mal: Sí. De hecho, por las ventas tan irregulares que ha habido este año, necesitamos algo nuevo. Lisa y yo hemos estado haciendo una pequeña investigación de mercado, y pensamos que tenemos una línea de productos que podría llegar a ser grande.

Joe: ¿Qué tan grande? ¿Como para ser una línea básica?

Mal (evitando un compromiso directo): Dile de nuestra idea, Lisa.

Lisa: Es toda una nueva generación de lo que llamamos "productos para la administración doméstica". Los llamamos *HogarSeguro*. Utilizan la nueva interfase inalámbrica, proporcionan a los usuarios domésticos o dueños de negocios pequeños un sistema que se controla con su PC: seguridad en el hogar, vigilancia de la casa, control de aparatos e instrumentos. Tú sabes, apagar el aire acondicionado de tu casa mientras estás manejando, y ese tipo de cosas.

Lee (interrumpiendo): Ingeniería hizo un estudio de factibilidad de esta idea, Joe. Se puede realizar a un bajo costo de manufactura. La mayor parte del hardware lo tenemos en existencia. El software es un asunto por resolver, pero nada que no podamos hacer.

Joe: Interesante. Ahora, pregunté sobre la línea básica.

Mal: Las PC han penetrado el 60 por ciento de los hogares en Estados Unidos. Si logramos ponerle el precio adecuado a esta cosa, podría ser una aplicación demoledora. Nadie más tiene nuestra caja inalámbrica; nos pertenece. Tendremos una ventaja de dos años sobre la competencia. ¿Ganancias? Podrían ser entre 30 y 40 millones de dólares en el segundo año.

Joe (sonriendo): Vamos a llevar esto al siguiente nivel. Estoy interesado.

1.7 Resumen

El software se ha convertido en el elemento clave de la evolución de los sistemas y productos basados en computadoras, así como en una de las tecnologías más importantes en el ámbito mundial. En los pasados 50 años, el software ha evolucionado desde ser una herramienta para la solución de problemas especializados y el análisis de información, hasta convertirse en una industria por sí mismo. Todavía se tienen problemas al desarrollar software de alta calidad a tiempo y dentro del presupuesto. El software —programas, datos y documentos— se dirige a un amplio espectro de tecnologías y áreas de aplicación. En la actualidad el software evoluciona de acuerdo con un conjunto de leyes que han permanecido inalteradas a lo largo de 30 años. La intención de la ingeniería del software es proporcionar un marco general para construir software con una calidad mucho mayor.

Referencias

[BRO75] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1995.
[DAC03] Daconta, M., L. Obrst y K. Smith, *The Semantic Web*, Wiley, 2003.
[DAY99] Dayani-Fard, H. *et al*., "Legacy Software Systems: Issues, Progress, and Challenges", IBM Technical Report: TR-74.165-k, abril de 1999, disponible en http://www.cas.ibm.com/toronto/publications/TR-74.165/k/legacy.html.
[DEM95] DeMarco, T., *Why Does Software Cost So Much?*, Dorset House, 1995.
[FEI83] Feigenbaum, E. A. y P. McCorduck, *The Fifth Generation*, Addison-Wesley, 1983.
[HAM93] Hammer, M. y J. Champy, *Reengineering the Corporation*, HarperCollins Publishers, 1993.

[JOH01] Johnson, S., *Emergence: The Connected Lives of Ants, Brains, Cities and Software*, Scribner, 2001

[KAR99] Karlson, E. y J. Kolber, *A Basic Introduction to Y2K. How the year 2000 Computer Crisis Affects YOU*, Next Era Publications, Inc, 1999

[LEH97a] Lehman, M y L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1997.

[LEH97b] Lehman, M. *et al* , "Metrics and Laws of Software Evolution—The Nineties View", en *Proceedings of the 4th International Software Metrics Symposium (METRICS '97)*, IEEE, 1997, puede descargarse de http://www ece.utexas edu/~perry/work/papers/feast1.pdf

[LEV95] Levy, S , "The Luddites Are Back", en *Newsweek*, 12 de julio de 1995, p. 55.

[LIP02] Lippman, A., "Round 2.0", en *Context Magazine*, agosto de 2002, http://www.context-mag com/

[LIU98] Liu, K *et al* , "Report on the First SEBPC Workshop on Legacy Systems", Durham University, febrero de 1998, disponible en http://www dur.ac.uk/CSM/SABA/legacy-wksp1/report html.

[OSB79] Osborne, A , *Running Wild—The Next Industrial Revolution*, Osborne/McGraw-Hill, 1979.

[NAI82] Naisbitt, J., *Megatrends*, Warner Books, 1982.

[STO89] Stoll, C., *The Cuckoo's Egg*, Doubleday, 1989.

[TOF80]Toffler, A., *The Third Wave*, Morrow Publishers, 1980.

[TOF90] Toffler, A *Powershift*, Bantam Publishers, 1990

[WIL02] Williams, S., "A Unified Theory of Software Evolution", en salon com, 2002. http://www salon.com/tech/feature/2002/04/08/lehman/index.html.

[WOL02] Wolfram, S., *A New Kind of Science*, Wolfram Media, Inc , 2002.

[YOU92] Yourdon, E., *The Decline and Fall of the American Programmer*, Yourdon Press, 1992.

[YOU96] Yourdon, E., *The Rise and Resurrection of the American Programmer*, Yourdon Press, 1996.

[YOU98a] Yourdon, E. y J. Yourdon, *Time Bomb 2000*, Prentice-Hall, 1998.

[YOU98b] Yourdon, E., *Death March Projects*, Prentice-Hall, 1999.

[YOU02] Yourdon, E , *Byte Wars*, Prentice-Hall, 2002.

Problemas y puntos a considerar

1.1. Encontrar al menos cinco ejemplos adicionales de la manera en que la ley de las consecuencias imprevistas se aplica al software de computadora.

1.2. Encontrar algunos ejemplos (positivos y negativos) que indiquen el impacto del software en la sociedad actual Revisar una de las referencias anteriores a 1990 en la sección 1.1, e indicar las predicciones del autor que resultaron correctas, así como las que fueron erróneas.

1.3. Desarrollar sus propias respuestas a las preguntas formuladas en la sección 1.1. Debátanse con los compañeros de clase

1.4. ¿La definición de software que se presenta en la sección 1.2 se aplica a los sitios Web? Si la respuesta es afirmativa, indicar la sutil diferencia entre un sitio Web y el software convencional

1.5. Muchas aplicaciones modernas cambian frecuentemente (antes de presentarlas al usuario final y después de que se empieza a utilizar la primera versión). Sugiéranse algunas formas de construir software para detener el deterioro debido al cambio.

1.6. Considérense las siete categorías presentadas en la sección 1.3. ¿Es posible aplicar el mismo enfoque de la ingeniería del software a cada una de ellas? Explicar la respuesta.

1.7. Seleccionar alguno de los nuevos retos mencionados en la sección 1.3 (o algún desafío aún más nuevo que pudiera haber surgido desde la impresión de este texto) y escribir un documento de una cuartilla que describa la tecnología y los retos que representa para los ingenieros de software.

1.8. Describir con palabras propias *la ley de la conservación de la estabilidad organizacional* (sección 1.4.2)

1.9. Describir con palabras propias *la ley de la conservación de la familiaridad* (sección 1.4.2.).

1.10 Describir con palabras propias *la ley de la calidad decreciente* (sección 1.4.2.).

1.11. A medida que la presencia del software se vuelve más generalizada, los riesgos al público (debido a las fallas en los programas) representan una preocupación significativa y creciente. Desarrollar un escenario catastrófico realista en el que la falla de un programa de computadora podría producir un gran daño (ya sea económico o humano).

1.12. Examinar con atención al grupo de noticias de Internet comp.risk y preparar un resumen de los riesgos al público que se han discutido recientemente. Fuente alternativa: *Software Engineering Note* publicada por la ACM

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN[10]

Existen miles de libros que tratan sobre el software de computadora. La inmensa mayoría discute los lenguajes de programación o las aplicaciones del software, pero muy pocos tratan del software en sí mismo. Pressman y Herron (*Software Shock*, Dorset House, 1991) presentan uno de los primeros debates (dirigidos al público en general) del software y de la forma en que los profesionales lo construyen. El libro más vendido de Negroponte (*Being Digital*, Alfred A. Knopf, Inc., 1995) ofrece una visión de la computación y su impacto global en el siglo xxi. DeMarco [DEM95] ha escrito una colección de ensayos divertidos y profundos acerca del software y del proceso a través del cual éste se desarrolla. Los libros de Norman (*The Invisible Computer*, MIT Press, 1998) y Bergman (*Information Appliances and Beyond*, Academic Press/Morgan Kaufmann, 2000) sugieren que el impacto extendido de las PC disminuirá conforme los instrumentos de información y la computación omnipresente conecten a todos en el mundo industrializado y casi cualquier "aparato" que se posea esté conectado a una nueva infraestructura de Internet.

Minasi (*The Software Conspiracy: Why Software Companies Put Out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) argumentaba que la "plaga moderna" de las impurezas del software se puede eliminar y sugiere formas de lograrlo. Compaine (*Digital Divide: Facing a Crisis or Creating a Myth*, MIT Press, 2001) escribe que la "brecha" entre aquellos que tienen acceso a los recursos de información (como la Web) y los que no lo tienen se está reduciendo conforme avanza la primera década del presente siglo.

En Internet existe una amplia variedad de fuentes de información sobre tópicos relacionados con el software y su administración. Asimismo, en nuestro sitio web se puede encontrar una lista actualizada de recursos en la red que son relevantes para el estudio del software: **http://www.mhhe.com/pressman**.

10 La sección *Otras lecturas y fuentes de información* que se presenta al final de cada capítulo ofrece un breve panorama de las fuentes impresas que pueden ayudarle a aumentar su comprensión de los temas principales presentados en este capítulo. Hemos creado un sitio de Internet muy extenso para apoyar *Ingeniería del software. un enfoque práctico* en http://www.mhhe com/pressman. Entre los muchos tópicos incluidos se encuentran referencias capítulo por capítulo sobre ingeniería del software existentes en la red que complementan del material presentado en cada capítulo. Con estas referencias se proporciona un enlace con Amazon.com para localizar los libros que se mencionan en cada sección.

PARTE

Uno

EL PROCESO DEL SOFTWARE

En esta parte de *Ingeniería del software: un enfoque práctico* se estudiará el proceso que proporciona un marco de trabajo para la práctica de la ingeniería del software. En los capítulos siguientes se responden estas preguntas:

- ¿Qué es un proceso de software?
- ¿Cuáles son las actividades del marco general presentes en todos los procesos del software?
- ¿Cómo se modelan los procesos y cuáles son los patrones del proceso?
- ¿Cuáles son los modelos de proceso prescriptivo y cuáles son sus fortalezas y debilidades?
- ¿Cuáles son las características de los modelos incrementales que los hacen idóneos para los proyectos modernos de software?
- ¿Qué es el proceso unificado?
- ¿Por qué la "agilidad" es un lema en el trabajo de la ingeniería moderna del software?
- ¿Qué es el desarrollo ágil del software y cómo difiere de los modelos de proceso más tradicionales?

Cuando se respondan estas preguntas se estará mejor preparado para entender el contexto en el cual se aplica la práctica de la ingeniería del software.

CAPÍTULO

2 El proceso: una visión general

Conceptos clave

En un fascinante libro que ofrece la visión de un economista sobre el software y la ingeniería del software, Howard Baetjer, Jr. [BAE98] comenta sobre el proceso del software:

> Debido a que el software, como cualquier capital, es conocimiento materializado, y dado que el conocimiento en un inicio es disperso, tácito, latente y en gran medida incompleto, el desarrollo del software es un proceso de aprendizaje social. El proceso es un diálogo en el cual el conocimiento que el software debe convertir se conjunta y se materializa en este último. El proceso proporciona interacción entre los usuarios y las herramientas en evolución, y entre los diseñadores y sus herramientas [tecnología]. Es un proceso iterativo en el que la herramienta en evolución sirve como un medio para la comunicación, en el cual cada nueva etapa del diálogo logra obtener más conocimiento útil de las personas implicadas.

De hecho, la construcción del software de computadora es un proceso iterativo de aprendizaje, y el resultado, algo que Baetjer llamaría "el capital del software", es una materialización del conocimiento recolectado, depurado y organizado conforme el proceso estuvo en ejecución.

Un vistazo rápido

¿Qué es? Cuando se trabaja para construir un producto o sistema es importante seguir una serie de pasos predecibles: una especie de mapa de carreteras que ayude a crear un resultado de alta calidad y a tiempo. El mapa de carreteras que debe seguirse se llama proceso de software.

¿Quién lo hace? Los ingenieros de software y sus jefes adaptan el proceso a sus necesidades y después lo siguen. Además, la gente que ha solicitado el software tiene una función qué desempeñar en el proceso de definirlo, construirlo y probarlo.

¿Por qué es importante? Porque ofrece estabilidad, control y organización a una actividad que puede volverse caótica si no se controla. Sin embargo, un enfoque de ingeniería del software moderno debe ser "ágil". Debe requerir sólo aquellas actividades, controles y documentaciones apropiados para el equipo del proyecto y el producto que ha de producirse.

¿Cuáles son los pasos? En detalle, el proceso que se adopte depende del software que se está construyendo. Un proceso puede ser apropiado para crear un software para un sistema de aeronáutica, mientras que un proceso distinto por completo sería el indicado para la creación de un sitio Web.

¿Cuál es el producto obtenido? Desde el punto de vista del ingeniero de software, los productos obtenidos son los programas, documentos y datos que se producen como consecuencia de las actividades y tareas definidas por el proceso.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Existen muchos mecanismos de evaluación del proceso de software que permiten a las organizaciones determinar la "madurez" del proceso de software. No obstante, la calidad, el tiempo requerido, la viabilidad a largo plazo del producto que se construye son los mejores indicadores de la eficacia del proceso que se utiliza.

Pero, ¿qué es con exactitud un proceso de software desde un punto de vista técnico? Dentro del contexto de este libro, un *proceso de software* se define como un marco de trabajo para las tareas que se requieren en la construcción de software de alta calidad. ¿El proceso es un sinónimo de ingeniería del software? La respuesta es sí y no. Un proceso de software define el enfoque que se adopta mientras el software está en desarrollo. Pero la ingeniería del software también abarca las tecnologías que requiere el proceso (métodos técnicos y herramientas automatizadas).

Aún más importante es que la ingeniería del software la realizan personas creativas y con conocimiento que deben trabajar en un proceso de software maduro que sea apropiado para el producto que construyen y para las demandas de sus mercados.

2.1 INGENIERÍA DEL SOFTWARE: UNA TECNOLOGÍA ESTRATIFICADA

A pesar de que cientos de autores han definido en forma individual la *ingeniería del software*, la definición que propuso Fritz Bauer [NAU69] en una conferencia fundamental sobre la materia aún se puede utilizar como base para el debate:

> [La ingeniería del software es] el establecimiento y uso de principios sólidos de la ingeniería para obtener económicamente un software confiable y que funcione de modo eficiente en máquinas reales.

Casi cualquier lector se sentirá tentado a sumar otras ideas a esta definición. Dice poco sobre los aspectos técnicos de la calidad del software; no se refiere de manera directa a la necesidad de satisfacer al cliente o al tiempo de entrega de un producto; omite mencionar la importancia de la medición y la métrica; no establece la importancia de un proceso efectivo. No obstante, la definición de Bauer ofrece una idea básica. ¿Cuáles son "los principios sólidos de la ingeniería" que pueden aplicarse en el desarrollo del software de computadora? ¿De qué manera se construye "económicamente" un software "confiable"? ¿Qué se requiere para crear programas de computadora que funcionen "de manera eficiente" no sólo en una, sino en varias "máquinas reales" diferentes? Estas interrogantes continúan siendo un reto para los ingenieros de software.

> "Más que una disciplina o un cuerpo de conocimiento, la ingeniería es un verbo, una palabra de acción, una manera de abordar un problema."
>
> **Scott Whitmire**

El IEEE [IEE93] ha elaborado una definición más comprensible al establecer:

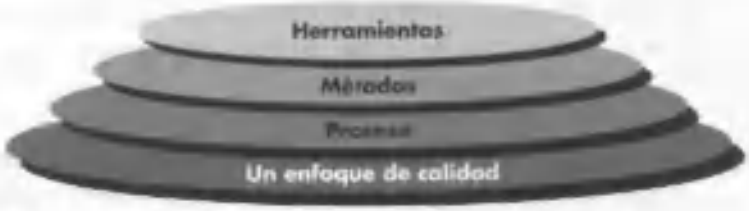
¿Cómo se define la ingeniería del software?

> Ingeniería del software: 1) La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software; es decir, la aplicación de la ingeniería al software. 2) El estudio de enfoques como en 1).

Y aun así, lo que es "sistemático, disciplinado" y "cuantificable" para un equipo de software, puede ser gravoso para otro. Se requiere de disciplina, pero también de adaptabilidad y agilidad.

FIGURA 2.1

Estratos de la ingeniería de software.



La ingeniería del software es una tecnología estratificada. Como se muestra en la figura 2.1, cualquier enfoque de la ingeniería (incluido el de la ingeniería del software) debe estar sustentado en un compromiso con la calidad. La Gestión de la Calidad Total, Sigma Seis y enfoques similares fomentan una cultura de mejora continua del proceso, y es esta cultura la que al final conduce al desarrollo de enfoques muy efectivos para la ingeniería del software. La base que soporta la ingeniería del software es un *enfoque en la calidad*.

PUNTO CLAVE

La ingeniería del software abarca un proceso, métodos y herramientas

La base de la ingeniería del software es el estrato del *proceso*. El proceso de la ingeniería del software es el elemento que mantiene juntos los estratos de la tecnología y que permite el desarrollo racional y a tiempo del software de computadora. El proceso define un marco de trabajo [PAU93] que debe establecerse para la entrega efectiva de la tecnología de la ingeniería del software. El proceso del software forma la base para el control de la gestión de los proyectos de software y establece el contexto en el cual se aplican los métodos técnicos, se generan los productos del trabajo (modelos, documentos, datos, reportes, formatos, etcétera), se establecen los fundamentos, se asegura la calidad, y el cambio se maneja de manera apropiada.

Los *métodos* de la ingeniería del software proporcionan los "cómo" técnicos para construir software. Los métodos abarcan un amplio espectro de tareas que incluyen la comunicación, el análisis de requisitos, el modelado del diseño, la construcción del programa, la realización de pruebas y el soporte. Los métodos de la ingeniería del software se basan en un conjunto de principios básicos que gobiernan cada área de la tecnología e incluye actividades de modelado y otras técnicas descriptivas.

Referencia Web

Cross Talk es una [illegible] sobre [illegible] sobre el proceso, los métodos y las herramientas. Puede encontrarse en **www.stsc.hill.af.mil.**

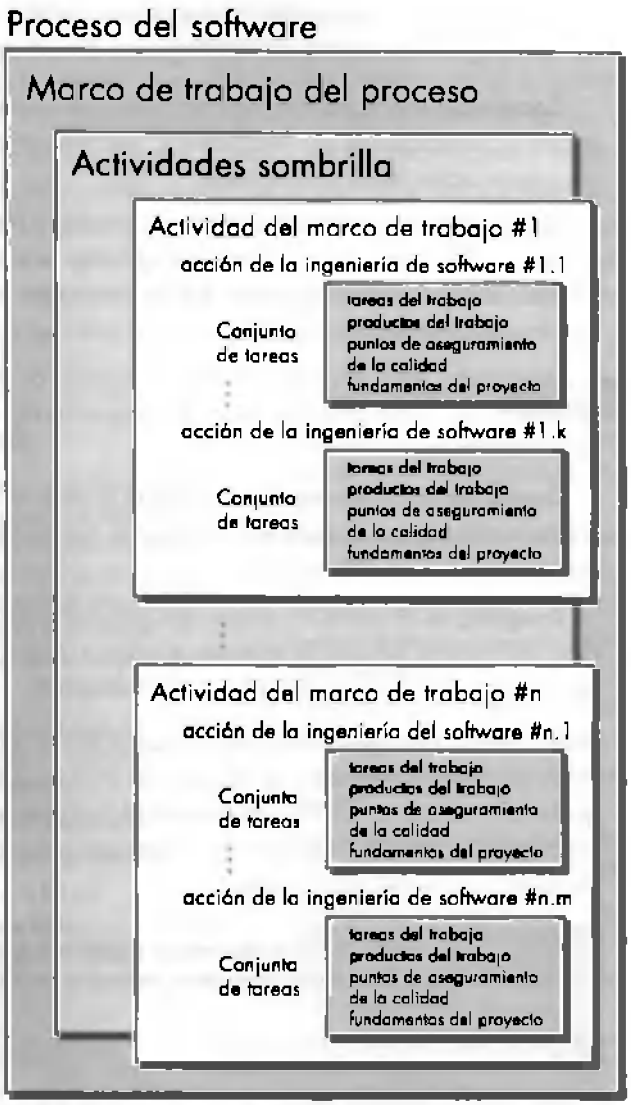
Las *herramientas* de la ingeniería del software proporcionan el soporte automatizado o semiautomatizado para el proceso y los métodos. Cuando las herramientas se integran de forma que la información que cree una de ellas pueda usarla otra, se dice que se ha establecido un sistema para el soporte del desarrollo del software, que con frecuencia se denomina ingeniería del *software asistida por computadora*.

2.2 MARCO DE TRABAJO PARA EL PROCESO

Un *marco de trabajo* establece la base para un proceso de software completo al identificar un número pequeño de *actividades del marco de trabajo* aplicables a todos los proyectos de software, sin importar su tamaño o complejidad. Además, el marco de trabajo del proceso abarca un conjunto de *actividades sombrilla* aplicables a lo largo del proceso del software.

FIGURA 2.2

Un marco de trabajo del proceso de software.



Como se muestra en la figura 2.2, cada actividad dentro del marco contiene un conjunto de *acciones de ingeniería del software*; es decir, una serie de tareas relacionadas que produce un producto del trabajo en la *ingeniería del software* (por ejemplo, el diseño es una acción de la ingeniería del software). Cada acción la forman *tareas de trabajo* individuales que completan alguna parte del trabajo implicado por la acción.

> "Un proceso define quién está haciendo qué, cuándo y cómo lograr cierta meta."
>
> **Ivar Jacobson, Grady Booch y James Rumbaugh**

El siguiente *marco de trabajo genérico del proceso* (utilizado como base para la descripción de los modelos de proceso en los capítulos subsecuentes) se puede aplicar en la inmensa mayoría de los proyectos de software:

¿Cuáles son las cinco actividades del marco de trabajo del proceso general?

Comunicación. Esta actividad del marco de trabajo implica una intensa colaboración y comunicación con los clientes;[1] además, abarca la investigación de requisitos y otras actividades relacionadas.

Planeación. Esta actividad establece un plan para el trabajo de la ingeniería del software. Describe las tareas técnicas que deben realizarse, los riesgos probables, los recursos que serán requeridos, los productos del trabajo que han de producirse y un programa de trabajo

Modelado. Esta actividad abarca la creación de modelos que permiten al desarrollador y al cliente entender mejor los requisitos del software y el diseño que logrará satisfacerlos

Construcción. Esta actividad combina la generación del código (ya sea manual o automatizado) y la realización de pruebas necesarias para descubrir errores en el código.

Despliegue. El software (como una entidad completa o un incremento completado de manera parcial) se entrega al cliente, quien evalúa el producto recibido y proporciona información basada en su evaluación.

Estas cinco actividades genéricas del marco de trabajo son útiles durante el desarrollo de programas pequeños, la creación de grandes aplicaciones en la red, y en la ingeniería de sistemas basados en computadoras grandes y complejas. Los detalles del proceso del software serán muy diferentes en cada caso, pero las actividades dentro del marco permanecerán iguales.

> "Einstein argumentaba que debía existir una explicación simplificada de la naturaleza porque Dios no es caprichoso ni arbitrario. Tal fe no conforta al ingeniero del software. Mucha de la complejidad que debe maniobrar es de carácter arbitrario."
>
> **Fred Brooks**

Si se usa un ejemplo derivado del marco de trabajo genérico del proceso, la actividad de *elaboración del modelo* la componen dos acciones de la ingeniería del software: *análisis* y *diseño*. El análisis[2] abarca un conjunto de tareas de trabajo (por ejemplo, la investigación, elaboración, negociación, especificación y validación de requisitos) que conducen a la creación del modelo de análisis (o a la especificación de re-

1. Un *cliente* es cualquier persona que tiene un interés en el éxito del resultado del proyecto: gerentes de negocios, usuarios finales, gente de apoyo, etcétera. Rob Thomset bromea diciendo que "un cliente (en inglés *stakeholder*) es una persona que sostiene una estaca (*stake*) grande y afilada... si no cuidas a tus clientes, ya sabes dónde terminará la estaca"
2. El análisis se explicará con mayor detenimiento en los capítulos 7 y 8

PUNTO CLAVE

Distintos proyectos demandan diferentes conjuntos de tareas. El equipo de software elige el conjunto de tareas con base en el problema y en las características del proyecto.

quisitos). El diseño abarca tareas de trabajo (diseño de datos, diseño arquitectónico, diseño de interfaz y diseño al nivel de componentes) que crean un modelo de diseño (una especificación de diseño).[3]

Como también se aprecia en la figura 2.2, cada acción de la ingeniería del software la representa un gran número de diferentes *conjuntos de tareas*: una serie de tareas de trabajo, productos relacionados, puntos para el aseguramiento de la calidad y fundamentos de proyecto dentro de la ingeniería del software. El conjunto de tareas que mejor se ajuste a las necesidades del proyecto y a las características del equipo es el que se escoge al final. Esto implica que una acción de la ingeniería del software (como el diseño) se puede adaptar a las necesidades específicas del proyecto de software y a las características del equipo de proyecto.

INFORMACIÓN

Conjunto de tareas

Un *conjunto de tareas* define el trabajo real que debe realizarse para cumplir los objetivos de una acción de ingeniería del software. Por ejemplo, la "recopilación de requisitos" es una acción importante de la ingeniería del software que ocurre durante la actividad de comunicación. La meta de la reunión de requisitos es entender qué desean los distintos clientes del software que se va a construir.

Para un proyecto pequeño, al parecer simple, el conjunto de tareas para la recopilación de requisitos puede ser como se enumera a continuación:

1. Hacer una lista de los clientes para el proyecto
2. Invitar a todos los clientes a una reunión informal
3. Pedir a cada cliente que haga una lista de características y funciones requeridas.
4. Establecer un debate sobre los requisitos y elaborar una lista final
5. Priorizar los requisitos.
6. Advertir las áreas de incertidumbre.

Para un proyecto de software mayor y más complejo, se requeriría un conjunto diferente de tareas. Este puede incluir la siguiente lista:

1. Hacer una lista de los clientes para el proyecto.
2. Entrevistar a cada uno de los clientes, por separado, para determinar de manera general sus deseos y necesidades.
3. Elaborar una lista preliminar de las funciones y características basadas en la información que ofrezcan los clientes
4. Hacer un programa de reuniones para recopilar los requisitos.
5. Conducir las reuniones
6. Producir escenarios informales de los usuarios como parte de cada reunión.
7. Refinar escenarios de los usuarios con base en el intercambio de información con los clientes.
8. Elaborar una lista revisada de los requisitos de los clientes.
9. Utilizar técnicas de despliegue de funciones de calidad para jerarquizar los requisitos.
10. Empaquetar los requisitos para que puedan entregarse de manera incremental.
11. Observar las restricciones que serán puestas en el sistema
12. Debatir métodos para validar el sistema

Ambos conjuntos de tareas consiguen la recopilación de requisitos, pero son muy diferentes en cuanto a profundidad y formalidad. El equipo de software elige el conjunto de tareas que permitirá alcanzar la meta de cada actividad del proceso y acción de ingeniería del software que mantenga la calidad y agilidad.

3 Cabe aclarar que "la elaboración del modelo" debe interpretarse de un modo diferente cuando se realiza el mantenimiento de un software existente. En algunos casos ocurre el modelado del diseño y el análisis, pero en otras situaciones de mantenimiento se le utiliza para ayudar a entender el software heredado, al igual que para representar adiciones o modificaciones en éste.

El marco de trabajo descrito en la visión general de la ingeniería de software lo completa una serie de *actividades sombrilla*. Las actividades típicas en esta categoría incluyen:

¡PUNTO CLAVE

Las actividades sombrilla ocurren a lo largo del proceso de software y se enfocan de modo principal en la gestión, el rastreo y el control del proyecto.

Seguimiento y control del proyecto de software: permite que el equipo de software evalúe el progreso comparándolo con el plan del proyecto y así tomar las acciones necesarias para mantener el programa.

Gestión del riesgo: evalúa los riesgos que pudieran afectar los resultados del proyecto o la calidad del producto.

Aseguramiento de la calidad del software: define y conduce las actividades requeridas para asegurar la calidad del software.

Revisiones técnicas formales: evalúa los productos del trabajo de la ingeniería del software en un esfuerzo encaminado a descubrir y eliminar los errores antes de que éstos se propaguen hacia la siguiente acción o actividad.

Medición: define y recolecta mediciones del proceso, el proyecto y el producto para ayudar al equipo a entregar software que satisfaga las necesidades del cliente; se puede usar en conjunto con todas las otras actividades del marco de trabajo o actividades sombrilla.

¡PUNTO CLAVE

La adaptación del proceso de software es esencial para el éxito del proyecto.

Gestión de la configuración del software: maneja los efectos del cambio a través del proceso del software

Gestión de la reutilización: define los criterios para la reutilización de productos del trabajo (se incluyen componentes del software) y establece mecanismos para la creación de componentes reutilizables.

Preparación y producción del producto de trabajo: abarca las actividades requeridas para crear productos del trabajo como modelos, documentos, registros, formatos y listas.

Las actividades sombrilla se aplican durante el proceso del software y se tratan con detalle en capítulos posteriores de este texto.

Todos los modelos de proceso se caracterizan dentro del marco del proceso mostrado en la figura 2.2. La aplicación inteligente de cualquier modelo de proceso del software debe reconocer que la adaptación (al problema, proyecto, equipo y a la cultura organizacional) es esencial para el éxito de esta actividad. Pero los modelos de proceso difieren de manera fundamental en:

- El flujo global de actividades y tareas, y las interdependencias entre las actividades y las tareas.
- El grado en el cual las tareas de trabajo están definidas dentro de cada actividad del marco de trabajo.
- El grado en el cual se identifican y se solicitan los productos de trabajo
- La forma en la que se aplican las actividades de aseguramiento de la calidad.
- La manera en la que se aplican las actividades de seguimiento y control.

? ¿De qué manera difieren los modelos del proceso entre sí?

- El grado general de detalle y el rigor con el que se describe el proceso.
- El grado en el que los clientes están comprometidos con el proyecto.
- El grado de autonomía otorgado al equipo de proyecto de software.
- El grado en el cual están definidos la organización y las responsabilidades en el equipo.

> "Siento que una receta es sólo un tema con el que un cocinero inteligente puede jugar cada vez de una manera distinta."
> **Madame Benoit**

Los modelos de proceso que enfatizan la definición, la identificación y la aplicación detallada de las actividades del proceso han sido aplicados dentro de la comunidad de la ingeniería del software en los últimos 30 años. La aplicación de estos *modelos prescriptivos* intenta mejorar la calidad del sistema, hacer que los proyectos sean más manejables, que las fechas de entrega y los costos sean más predecibles, y guiar a los equipos de ingenieros de software mientras realizan el trabajo que requiere construir un sistema. Por desgracia, ha habido ocasiones en que estos objetivos no se han alcanzado. Si los modelos prescriptivos se aplican en forma dogmática y sin ninguna adaptación, éstos pueden incrementar el grado de burocracia asociada con la construcción de sistemas basados en computadoras, y de manera inadvertida crear dificultades para los desarrolladores y los clientes.

¿Qué caracteriza un proceso "ágil"?

En años recientes se han propuesto modelos de proceso que subrayan la agilidad del proyecto y siguen un conjunto de principios,[4] los cuales conducen a un enfoque más informal para el proceso del software (dicho enfoque no es menos efectivo, según argumentan quienes lo propusieron). Estos *modelos ágiles del proceso* resaltan la manejabilidad y adaptabilidad. Son apropiados para muchos tipos de proyectos y son útiles de manera particular cuando se desarrollan aplicaciones en la red.

¿Cuál de los enfoques para el proceso del software es el mejor? Esta pregunta ha ocasionado un debate emocional entre los ingenieros de software y se abordará en el capítulo 4. Por ahora, es importante darse cuenta de que estos dos enfoques de proceso tienen una meta común: crear software de alta calidad que satisfaga las necesidades del cliente, pero tienen perspectivas diferentes.

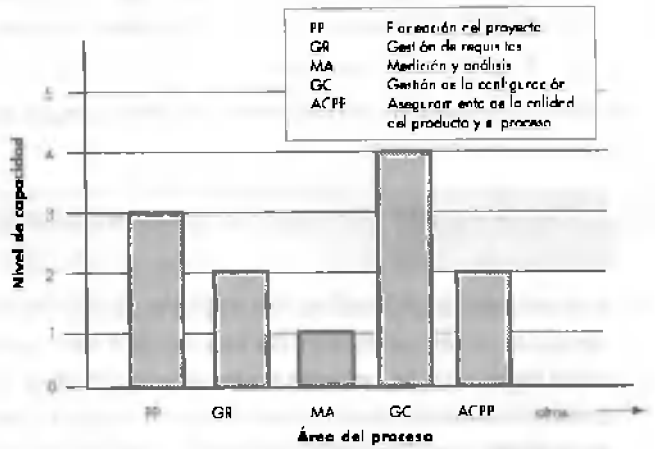
2.3 INTEGRACIÓN DEL MODELO DE CAPACIDAD DE MADUREZ (IMCM)

El Instituto de Ingeniería del Software (SEI, por sus siglas en inglés) ha desarrollado un modelo completo de un amplio proceso basado en un conjunto de capacidades de software y de sistemas que deben estar presentes conforme las organizaciones alcanzan diferentes grados de capacidad y madurez del proceso. El SEI sostiene que para lograr estas capacidades una organización debe crear un modelo de proceso (figura 2.2) que se ajuste a las directrices establecidas por la *integración del modelo de capacidad de madurez* (IMCM) [CMM02].

4 Los modelos ágiles y los principios que los guían se explican en el capítulo 4.

FIGURA 2.3

Perfil de capacidad del área del proceso de la IMCM [PHI02].



La IMCM representa un modelo completo de proceso en dos formas diferentes: 1) como un modelo continuo y 2)como un modelo discreto. El modelo continuo IMCM describe un proceso en dos dimensiones, como se ilustra en la figura 2.3 Cada área del proceso (por ejemplo, la planeación del proyecto o la gestión de los requisitos) se evalúa de manera formal contra las metas y prácticas específicas y se clasifica de acuerdo con los siguientes niveles de capacidad:

Referencia Web

Una información completa acerca de la IMCM puede obtenerse en http://www.sei.cmu.edu/cmmi/.

Nivel 0: Incompleto. El área del proceso (por ejemplo, la gestión de requisitos) aún no se realiza o todavía no alcanza todas las metas y objetivos definidos para el nivel 1 de capacidad.

Nivel 1: Realizado. Todas las metas específicas del área del proceso (como las definió la IMCM) han sido satisfechas. Las tareas de trabajo requeridas para producir el producto específico han sido realizadas.

Nivel 2: Administrado. Todos los criterios del nivel 1 han sido satisfechos. Además, todo el trabajo asociado con el área de proceso se ajusta a una política organizacional definida; toda la gente que ejecuta el trabajo tiene acceso a recursos adecuados para realizar su labor; los clientes están implicados de manera activa en el área de proceso, cuando esto se requiere; todas las tareas de trabajo y productos están "monitoreados, controlados y revisados; y son evaluados en apego a la descripción del proceso" [CMM02].

Nivel 3: Definido. Todos los criterios del nivel 2 se han cumplido. Además, el proceso está "adaptado al conjunto de procesos estándar de la organización, de acuerdo con las políticas de adaptación de esta misma, y contribuye a la información de los productos del trabajo, mediciones y otras mejorías del proceso para los activos del proceso organizacional" [CMM02].

Nivel 4: Administrado en forma cuantitativa. Todos los criterios del nivel 3 han sido cumplidos. Además, el área del proceso se controla y mejora mediante mediciones y evaluación cuantitativa. "Los objetivos cuantitativos para la calidad y el

desempeño del proceso están establecidos y se utilizan como un criterio para administrar el proceso" [CMM02].

Nivel 5: Mejorado. Todos los criterios del nivel 4 han sido satisfechos. Además, el área del proceso "se adapta y mejora mediante el uso de medios cuantitativos (estadísticos) para conocer las necesidades cambiantes del cliente y mejorar de manera continua la eficacia del área del proceso que se está considerando" [CMM02].

> "Gran parte de la crisis del software es autoinfligida, como cuando CIO dice: "Prefiero que esté mal a que esté tarde. Siempre podemos repararlo después."
>
> **Mark Paulk**

Una organización debe intentar apegarse a la IMCM. Sin embargo, la implementación de cada aspecto del modelo puede ser excesiva en algunas situaciones.

La IMCM define cada área del proceso en función de "metas específicas" y de las "prácticas específicas" requeridas para alcanzar dichas metas. Las *metas específicas* establecen las características que deben existir para que las actividades implicadas por un área de proceso sean efectivas. Las *prácticas específicas* convierten una meta en un conjunto de actividades relacionadas con el proceso.

Por ejemplo, la **planeación del proyecto** es una de las ocho áreas del proceso definidas por la IMCM para la categoría de "gestión del proyecto".[5] Las metas específicas (ME) y las prácticas específicas asociadas (PE) que se han definido para la planeación del proyecto son [CMM02]:

ME 1 Establecer estimaciones

PE 1.1-1 Estimar el alcance del proyecto

PE 1.2-1 Establecer estimaciones para los atributos del producto y las tareas del trabajo.

PE 1.3-1 Definir el ciclo de vida del proyecto.

PE 1.4-1 Determinar estimaciones de esfuerzo y costo.

ME 2 Desarrollar un plan de proyecto

PE 2.1-1 Establecer el presupuesto y el programa.

PE 2.2-1 Identificar los riesgos del proyecto.

PE 2.3-1 Planear la gestión de los datos.

PE 2.4-1 Planear los recursos del proyecto.

PE 2.5-1 Planear los conocimientos y habilidades que se requieren

PE 2.6-1 Planear la participación del cliente

PE 2.7-1 Establecer el plan de proyecto.

5 Otras áreas del proceso definidas como "gestión del proyecto" incluyen: monitoreo y control del proyecto, gestión de acuerdos con proveedores, gestión integrada del proyecto para IPPD, gestión del riesgo, integración del equipo, gestión de integración del proveedor y gestión cuantitativa del proyecto.

ME 3 Comprometerse con la planeación

PE 3.1-1 Revisar los planes que afectan el proyecto.

PE 3.2-1 Conciliar el trabajo y los niveles de recursos.

PE 3.3-1 Comprometerse con la planeación

Además de las metas y prácticas específicas, la IMCM también define una serie de cinco metas genéricas y prácticas relacionadas con cada área del proceso. Cada una de las metas genéricas corresponde a uno de los cinco niveles de capacidad. Por lo tanto, para lograr un nivel de capacidad particular se debe alcanzar la meta genérica para ese nivel y las prácticas genéricas que corresponden a esa meta. Para ilustrar lo anterior, a continuación se enumeran las metas genéricas (MG) y las prácticas genéricas (PG) para el área del proceso de planeación del proyecto [CMM02]:

MG 1 Alcanzar las metas específicas

PG 1.1 Realizar prácticas base.

MG 2 Institucionalizar un proceso de gestión

PG 2.1 Establecer una política organizacional.

PG 2.2 Planear el proceso.

PG 2.3 Proporcionar recursos.

PG 2.4 Asignar responsabilidades.

PG 2.5 Capacitar gente

PG 2.6 Manejar configuraciones.

PG 2.7 Identificar y hacer participar a clientes.

PG 2.8 Monitorear y controlar el proceso.

PG 2.9 Evaluar la adherencia de un modo objetivo.

PG 2.10 Revisar el estatus con un alto grado de gestión

MG 3 Institucionalizar un proceso definido

PG 3.1 Establecer un proceso definido.

PG 3.2 Recolectar información de la mejoría.

MG 4 Institucionalizar un proceso manejado en forma cuantitativa

PG 4.1 Establecer objetivos cuantitativos para el proceso.

PG 4.2 Estabilizar el desempeño del subproceso

MG 5 Institucionalizar un proceso de mejoramiento.

PG 5.1 Asegurar la mejora continua del proceso.

PG 5.2 Corregir las causas de los problemas desde la raíz

Referencia Web

Una información completa, así como una versión disponible de la IMCM puede obtenerse en www.sei.cmu.edu/cmmi/.

El modelo discreto de la IMCM define las mismas áreas, metas y prácticas del proceso que el modelo continuo. La principal diferencia es que el modelo discreto establece cinco niveles de madurez, en vez de cinco niveles de capacidad. Para lograr un nivel de madurez se deben conseguir metas y prácticas específicas relacionadas con un conjunto de áreas del proceso. La relación entre los niveles de madurez y las áreas del proceso se muestran en la figura 2.4.

INFORMACIÓN

La IMCM: ¿se debe o no hacer?

La IMCM es un modelo total del proceso. Define (en alrededor de 700 páginas) las características del proceso que deben existir si una organización desea establecer un proceso de software completo. La pregunta que se ha debatido durante una década es ¿la IMCM es excesiva? Como la mayor parte de las cosas en la vida (y en el software) la respuesta no es un simple sí o no.

Siempre debe adoptarse el espíritu de la IMCM. Frente al riesgo de la simplificación excesiva, se argumenta que el desarrollo del software debe tomarse con seriedad; debe planearse; debe controlarse de manera uniforme; debe administrarse con precisión; debe conducirse de manera profesional. Debe centrarse en las necesidades de los clientes del proyecto, las habilidades de los ingenieros de software y la calidad del producto terminado. Nadie debe poner en duda estas ideas.

Los requisitos detallados de la IMCM deben tomarse en cuenta con seriedad si una organización construye sistemas grandes y complejos que impliquen docenas o cientos de personas por varios meses o años. Es posible que la IMCM sea correcta en ciertas situaciones, si la cultura organizacional es flexible frente a modelos de procesos estándares y se realiza una gestión para lograr que sea un éxito.

No obstante, en otras situaciones es posible que la IMCM sea demasiado para que una organización la asimile de manera exitosa. ¿Esto significa que la IMCM es mala o demasiado burocrática o que está pasada de moda? No. Tan sólo significa que lo correcto para la cultura de una compañía puede no serlo para otra.

La IMCM es un logro significativo para la ingeniería del software. Proporciona una exposición integral de las actividades y acciones que deben estar presentes cuando una organización construye un software de computadora. Aun si una organización de software elige no adoptar sus detalles, todo equipo de software debe retomar su espíritu y aprender de su exposición del proceso y la práctica de la ingeniería del software.

FIGURA 2.4

Áreas del proceso requeridas para alcanzar un nivel de madurez.

Nivel	Enfoque	Áreas del proceso
De optimización	*Mejora continua del proceso*	Innovación organizacional y despliegue Análisis causal y resolución
Gestionado de modo cuantitativo	*Gestión cuantitativa*	Ejecución del proceso organizacional Gestión cuantitativa del proyecto
Definido	*Estandarización del proceso*	Desarrollo de requisitos Solución técnica Integración del producto Verificación Validación Enfoque del proceso organizacional Definición del proceso organizacional Capacitación organizacional Gestión integrada del proyecto Gestión integrada del proveedor Gestión del riesgo Análisis y resolución de la decisión Ambiente organizacional para la integración Equipo integrado
Gestionado	*Gestión básica del proyecto*	Gestión de requisitos Planeación del proyecto Monitoreo y control del proyecto Gestión de acuerdos del proveedor Medición y análisis Aseguramiento de la calidad del producto y del proceso Gestión de la configuración
Ejecutado		

2.4 PATRONES DEL PROCESO

¿Qué es un patrón del proceso?

El proceso de software puede definirse como una colección de patrones que definen un conjunto de actividades, acciones, tareas de trabajo o comportamientos relacionados [AMB98] que requiere el desarrollo de un software de computadora. Dicho en términos generales, *un patrón de proceso* ofrece una plantilla: un método consistente para describir una característica importante del proceso de software. Mediante la combinación de patrones, un equipo de software puede construir un proceso que satisfaga lo mejor posible las necesidades de un proyecto.

> "La repetición de patrones es muy distinto a la repetición de partes. Además, las partes diferentes serán únicas porque los patrones son únicos."
>
> **Christopher Alexander**

Los patrones pueden definirse en cualquier grado de abstracción.[6] En algunos casos se puede utilizar un patrón para describir un proceso completo (por ejemplo, un prototipo). En otras situaciones se utilizan los patrones para describir una actividad del marco de trabajo importante (como la planeación) o una tarea dentro de una actividad del marco de trabajo (por ejemplo, la estimación de un proyecto).

Ambler [AMB98] propuso la siguiente plantilla para describir un patrón de proceso:

PUNTO CLAVE

Una plantilla del patrón ofrece un medio consistente para describir un patrón

Nombre del patrón. Al patrón se le asigna un nombre significativo que describa su función dentro del software (como **comunicación con el cliente**).

Propósito. Se describe con brevedad el objetivo del patrón. Por ejemplo, el objetivo de la **comunicación con el cliente** es "establecer una relación de colaboración con el cliente en un esfuerzo encaminado a definir el alcance del proyecto, los requisitos del negocio y otras condiciones del proyecto". El propósito puede expandirse con textos explicatorios adicionales y diagramas apropiados, si se requieren.

Tipo. Se especifica el tipo de patrón. Ambler [AMB98] sugiere tres tipos:

- Los *patrones de tarea* definen una acción de la ingeniería del software o una tarea de trabajo que es parte del proceso y relevante para una práctica exitosa de la ingeniería del software (por ejemplo, la **recopilación de requisitos** es un patrón de tarea).
- Los *patrones de escenario* definen una actividad del marco de trabajo para el proceso. Debido a que una actividad del marco de trabajo reúne múltiples tareas de trabajo, un patrón de escenario incorpora múltiples patrones de tarea relevantes para el escenario (actividad del marco de trabajo). Un ejemplo del patrón de escenario es la **comunicación**. Este patrón incorporaría el patrón de tarea **reunión de requisitos** y otros.

6 Los patrones se aplican a muchas actividades de ingeniería del software. El análisis, el diseño y los patrones de prueba se explican en los capítulos 7, 9, 10, 12 y 14. Los patrones y "antipatrones" para las actividades de gestión de proyectos se explican en la parte 4 de este libro.

- Los *patrones de fase* definen la secuencia de actividades del marco de trabajo que ocurre junto con el proceso, aun cuando el flujo general de actividades es iterativo por naturaleza. Un ejemplo de un patrón de fase puede ser un **modelo en espiral o de construcción de prototipos**.[7]

Contexto inicial. Se describen las condiciones en las cuales se aplica el patrón. Antes de iniciar éste se debe cuestionar 1) qué actividades organizacionales o relativas al equipo han ocurrido, 2) cuál es el estado de entrada para el proceso, y 3) qué información de ingeniería del software o información del proyecto ya existe.

Por ejemplo, el patrón de **planeación** (un patrón discreto) requiere que 1) los clientes y los ingenieros de software hayan establecido una colaboración en cuanto a comunicación; 2) se haya completado con éxito un gran número de patrones de tarea (especificados) para el patrón de **comunicación con el cliente**; y 3) se conozcan los alcances del proyecto, los requisitos básicos del negocio y las restricciones del proyecto

Problema. Se describe el problema que debe resolver el patrón. Por ejemplo, el problema que debe resolver **la comunicación con el cliente** puede describirse de la siguiente manera: *La comunicación entre el desarrollador y el cliente muchas veces es inadecuada porque no se establece un formato efectivo para obtener información, no se crea un mecanismo útil para registrarla, y no se realiza una revisión significativa*

Solución. Se describe la implementación del patrón. En esta sección se discute cómo el estado inicial del proceso (existente antes de que se haya implementado el patrón) se modifica como consecuencia del inicio del patrón. También se describe cómo la información de la ingeniería del software o la información del proyecto, disponible antes de iniciado el patrón, se transforma como consecuencia de la ejecución exitosa del patrón.

Contexto resultante. Se describen las condiciones que habrá una vez que el patrón haya sido implementado con éxito. Para completar el patrón deben realizarse las siguientes preguntas: 1) ¿qué actividades organizacionales o relacionadas con el equipo debieron haber ocurrido?, 2) ¿cuál es el estado de salida para el proceso?, y 3) ¿qué información de la ingeniería del software o información del proyecto ha sido desarrollada?

Patrones relacionados. Se proporciona una lista de todos los patrones de proceso directamente relacionados con éste, en forma jerárquica o de alguna otra forma. Por ejemplo, el patrón estacionario de **comunicación** abarca los patrones de tarea **reunión del equipo para el proyecto**, **definición de una línea de colaboración**, **aislamiento de alcances**, **reunión de requisitos**, **descripción de restricciones** y una **creación de un modelo mini-spec**.

7 Estos patrones de fase se exponen en el capítulo 3.

Usos conocidos/Ejemplos. Se indican los ejemplos específicos en los cuales el patrón es aplicable Por ejemplo, la **comunicación** es obligatoria al principio de cada proyecto de software; se recomienda por medio del proceso de software, y es obligatoria una vez que la actividad de **despliegue** esté realizada

Referencia Web

Se pueden encontrar fuentes externas sobre los patrones del proceso en **www.ambysoft.com/processPatternsPage.html.**

Los patrones de proceso proporcionan un mecanismo efectivo para describir cualquier proceso de software. Los patrones permiten una organización de ingeniería del software para desarrollar un descripción del proceso jerárquico que comience en un alto grado de abstracción (un patrón de fase). La descripción se refina hasta un conjunto de patrones estacionarios que describen actividades del marco de trabajo, y más tarde se refina de un modo jerárquico en patrones de tareas más detallados para cada patrón estacionario. Después de que se han desarrollado los patrones de proceso, pueden reutilizarse para la definición de variantes de proceso; es decir, un equipo de software puede definir un modelo de proceso personalizado usando patrones como bloques de construcción para el modelo de proceso.

INFORMACIÓN

Ejemplo de un patrón del proceso

El siguiente patrón de proceso abreviado describe un enfoque aplicable cuando los clientes tienen una idea general de lo que debe hacerse, pero no están seguros de los requisitos específicos del software

Nombre del patrón. Prototipo.

Propósito. El objetivo del patrón es construir un modelo (un prototipo) que los clientes evalúen de modo iterativo en un esfuerzo encaminado a identificar los requisitos del software.

Tipo. Patrón de fase.

Contexto inicial. Deben cumplirse las siguientes condiciones antes de iniciar este patrón: 1) los clientes han sido identificados; 2) se ha establecido un modo de comunicación entre los clientes y el equipo de trabajo; 3) los clientes han identificado el problema que ha de resolverse; 4) se ha desarrollado un entendimiento inicial del alcance del proyecto, los requisitos básicos del negocio y las restricciones del proyecto.

Problema. Los requisitos son vagos o no existen. No obstante, se reconoce con claridad la existencia de un problema, y éste debe ir acompañado de una solución de software Los clientes no están seguros de lo que desean; es decir, no pueden describir ningún detalle de los requisitos del software

Solución. Aquí se presenta una descripción del proceso de prototipo Para más detalles, véase el capítulo 3

Contexto resultante. Los clientes aprueban un prototipo de software que identifica requisitos básicos (por ejemplo, modelos de interacción, rasgos computacionales, funciones de procesamiento). Después 1) el prototipo puede evolucionar recorriendo una serie de incrementos para convertirse en el software de producción, o 2) el prototipo se descarta y el software de producción se construye con otros patrones de proceso.

Patrones relacionados. Los siguientes patrones están relacionados con este patrón: **comunicación con el cliente; diseño iterativo; desarrollo iterativo; evaluación del cliente; extracción de requisitos.**

Usos conocidos/ejemplos. El prototipo se recomienda cuando los requisitos son inseguros

2.6 EVALUACIÓN DEL PROCESO

La existencia de un proceso de software no es garantía de que éste será entregado a tiempo, de que satisfará las necesidades del cliente, o de que mostrará las características técnicas que conducirán a características de calidad a largo plazo (capítulo 26) Los patrones de proceso deben ir acompañados de una práctica sólida de la

FIGURA 2.5

PUNTO CLAVE

En la evaluación se pretende comprender el estado actual del proceso de software y su interés mejorarlo.

¿De qué técnicas manuales se dispone para evaluar el proceso de software?

ingeniería del software (parte 2 de este libro). Además, el proceso mismo debe evaluarse para asegurarse de que cumpla una serie de criterios básicos del proceso que han demostrado ser esenciales para una ingeniería de software exitosa [8]. La relación entre el proceso de software y los métodos aplicados para la evaluación y el mejoramiento se muestra en la figura 2.5. Se han propuesto varios enfoques para la *evaluación del proceso de software* en las décadas pasadas.

El **método de evaluación de la IMCM estándar para el mejoramiento del proceso (MEIEMP)** ofrece un modelo de cinco pasos para la evaluación del proceso que incluye la iniciación, el diagnóstico, el establecimiento, la acción y el aprendizaje. El método MEIEMP utiliza el SEI IMCM (sección 2.3) como base para la evaluación [SEI00].

La apreciación basada en el CMM para el mejoramiento del proceso interno (ABC MPI) ofrece una técnica de diagnóstico para evaluar la madurez relativa de una organización de software mediante la ABC MPI (un precursor de la IMCM, el cual se explicó en la sección 2.3) como base para la evaluación [DUN01].

El estándar **SPICE (ISO/IEC15504)** define un conjunto de requisitos para la evaluación del proceso de software, lo que pretende es ayudar a las organizaciones en el desarrollo de una evaluación objetiva de la eficacia de cualquier proceso de software definido [SPI99].

El **ISO 9001:2000 para software** es un estándar genérico que se aplica a cualquier organización que desee mejorar la calidad general de los productos, sistemas

8 La IMCM del SEI [CMM02] describe, en detalle y con amplitud, las características de un proceso de software y los criterios para un proceso exitoso.

o servicios que provee. Por lo tanto, el estándar se aplica de modo directo a compañías y organizaciones de software.

Debido a que el ISO 9001:2000 se usa de manera amplia en el ámbito internacional, se examinará con brevedad en los párrafos que siguen.

> "Las organizaciones de software han mostrado deficiencias significativas en su habilidad para capitalizar las experiencias ganadas en proyectos completos."
>
> **NASA**

La Organización Internacional de Estandarización (ISO, por sus siglas en inglés) ha establecido el estándar ISO 9001:2000 [ISO00] para definir los requisitos de un sistema de gestión de calidad (capítulo 26) que sirva para elaborar productos de más alta calidad y así mejorar la satisfacción del cliente.[9]

La estrategia fundamental que sugiere el ISO 9001:2000 se describe de la siguiente manera:

> El ISO 9001:2000 subraya la importancia que tiene para una organización identificar, implementar, gestionar y mejorar de manera continua la efectividad de los procesos necesarios para el sistema de administración de la calidad, y gestionar las interacciones de estos procesos para conseguir los objetivos de la organización.

Referencia Web

Un excelente resumen del ISO 9001:2000 puede encontrarse en http://praxiom.com/iso-9001.htm.

El ISO 9001:2000 ha adoptado un ciclo de "planear-hacer-revisar-actuar" que se aplica a los elementos de gestión de calidad de un proyecto de software. Dentro de un contexto de software, "planear" establece los objetivos, las actividades y tareas del proceso necesarios para conseguir un software de alta calidad y una satisfacción del cliente; "hacer" implementa el proceso de software (incluidas las actividades del marco de trabajo y las actividades sombrilla); "revisar" monitorea y mide el proceso para asegurarse de que todos los requisitos establecidos para la gestión de calidad hayan sido cumplidos; "actuar" inicia las actividades de mejoramiento del proceso de software, el cual tiene una continuidad de trabajo para mejorar el proceso.

Para un tratamiento más detallado del ISO 9001:2000 los lectores interesados en el tema deben consultar los estándares ISO o [CIA01], [KET01] o [MON01]

2.6 Modelos de proceso personales y en equipo

El mejor proceso de software es el que está cerca de la gente que realizará el trabajo. Si un modelo de proceso de software ha sido desarrollado en un ámbito corporativo u organizacional, puede ser efectivo sólo si es en gran medida adaptable para satisfacer las necesidades del equipo del proyecto, que es el que en realidad lleva a cabo el trabajo de ingeniería del software. En un escenario ideal, cada ingeniero de software crearía un proceso que llene lo mejor posible sus propias necesidades, y al mis-

9 El aseguramiento de la calidad del software (ACS), un elemento importante de la gestión de calidad, ha sido definido como una actividad sombrilla que se aplica a través de todo el marco de trabajo del proceso. Se expone en detalle en el capítulo 26.

mo tiempo satisfaga las amplias necesidades del equipo y la organización. De modo alternativo, el equipo mismo crearía su propio proceso, y al mismo tiempo cubriría las necesidades más reducidas de los individuos y las necesidades amplias de la organización. Watts Humphrey ([HUM97] y [HUM00]) argumenta que es posible crear un "proceso de software personal" o un "proceso de software en equipo". Ambos requieren de un arduo trabajo, capacitación y coordinación, pero ambos se pueden conseguir [10]

> "Una persona que tiene éxito sólo se ha formado el hábito de hacer las cosas que la gente sin éxito no hace."
>
> **Dexter Yager**

Referencia Web

Una gran cantidad de fuentes para el PSP puede encontrarse en www.lpd.edu.de/PSP/.

2.6.1 Proceso de software personal (PSP)

Cada desarrollador utiliza algún proceso para construir un software de computadora. El proceso puede ser fortuito o *ad hoc*, puede cambiar a diario, puede no ser eficiente, efectivo o hasta exitoso, pero existe un proceso. Watts Humphrey [HUM97] sugiere que para cambiar un proceso personal inefectivo, un individuo debe pasar por cuatro fases, en las cuales se requiere capacitación e instrumentación cuidadosa. El *proceso de software personal* (PSP) resalta la medida personal del producto de trabajo que se produce y la calidad resultante del producto de trabajo. Además, el PSP responsabiliza al profesional encargado de la planeación del proyecto (por ejemplo, la estimación y la planificación) y le confiere el poder de controlar la calidad de todos los productos de trabajo del software que se desarrollan.

El modelo PSP define cinco actividades del marco de trabajo: planeación, diseño de alto nivel, revisión del diseño de alto nivel, desarrollo y análisis de resultados.

¿Qué actividades del marco de trabajo se aplican durante el PSP?

Planeación. Esta actividad selecciona requisitos y, con base en éstos, desarrolla el tamaño y la estimación de los recursos. Además, se estiman los defectos (el número de defectos proyectado en el trabajo). Todas las mediciones se registran en hojas de trabajo o en plantillas. Al final, se identifican las tareas de desarrollo y se crea un programa del proyecto.

Diseño de alto nivel. Se elaboran las especificaciones externas para que cada componente sea construido y se crea un diseño del componente. Se construyen prototipos cuando existe incertidumbre. Todos los elementos se registran y rastrean.

Revisión del diseño de alto nivel. Los métodos formales de verificación (capítulo 26) se aplican a errores descubiertos en el diseño. Las mediciones se mantienen para todas las tareas importantes y los resultados de trabajo.

Desarrollo. El diseño al nivel de componente se refina y revisa. Se genera, revisa, compila y prueba el código. Las mediciones se mantienen para todas las tareas importantes y los resultados de trabajo.

10 Vale la pena mencionar que los proponentes del desarrollo ágil del software (capítulo 4) también argumentan que el proceso debe permanecer cerca del equipo. Ellos proponen un método alternativo para lograrlo.

Análisis de resultados. Mediante las mediciones y medidas recolectadas (una cantidad sustancial de datos debe analizarse de manera estadística) se determina la efectividad del proceso. Las mediciones y medidas deben ofrecer una guía para modificar el proceso y así mejorar su efectividad.

El PSP destaca la necesidad que tiene cada ingeniero de software de identificar los errores desde el principio y la importancia de entender los tipos de errores que suele cometer. Esto se lleva a cabo mediante una actividad de evaluación rigurosa aplicada en todos los productos de trabajo que genera el ingeniero de software.

PUNTO CLAVE

El PSP destaca la necesidad de registrar y analizar los tipos de errores que se cometen para desarrollar estrategias encaminadas a eliminarlos.

El PSP representa un enfoque disciplinado, basado en mediciones, de la ingeniería de software que puede conducir a un choque de culturas a muchos profesionales. Sin embargo, cuando el PSP se presenta de un modo adecuado a los ingenieros de software [HUM96], la mejoría resultante en la productividad de la ingeniería del software y la calidad de éste son significativas [FER97]. No obstante, la industria no ha adoptado con amplitud el PSP. Las razones, tristemente, tienen más relación con la naturaleza humana y la inercia organizacional que con las fuerzas y debilidades del enfoque del PSP. Este último es un reto intelectual y demanda un grado de compromiso (por parte de los profesionales y sus jefes) que no siempre es posible obtener. La capacitación es relativamente larga y sus costos son altos. En lo cultural, el grado requerido de medición es difícil para mucha gente involucrada con el software.

Una interrogante es si el PSP puede usarse como un proceso de software efectivo a un nivel personal. La respuesta es, sin duda, sí. Pero aun si el PSP no es adoptado en su totalidad, vale la pena estudiar muchos de los conceptos de mejora del proceso que éste presenta.

2.6.2 Proceso de software en equipo (PSE)

Referencia Web

Más información sobre la construcción de equipos de alto desempeño empleando el PSE y el PSP puede obtenerse en **www.sei.cmu.edu/tsp/.**

Debido a que muchos proyectos de software en el ámbito industrial los dirige un equipo de profesionales, Watts Humphrey extendió las lecciones aprendidas para la introducción del PSP y propuso un *proceso de software en equipo* (PSE). La meta del PSE es construir un equipo de proyecto "autodirigido" que se organice para producir un software de alta calidad. Humphrey [HUM98] define los siguientes objetivos del PSE:

- Construir equipos autodirigidos que planeen y tengan un seguimiento de su trabajo, establezcan metas y posean sus procesos y planes. Estos grupos pueden ser equipos de software puros o equipos de producto integrado (EPI) de 3 a 20 ingenieros.
- Mostrar a los jefes cómo preparar y motivar a sus equipos y cómo ayudarlos a sostener un alto desempeño.
- Acelerar el mejoramiento del proceso de software al realizar, con el comportamiento normal y esperado, el nivel 5 del MCM.
- Ofrecer una guía de mejoramiento a organizaciones de alta madurez.

CONSEJO

- Facilitar la enseñanza universitaria de habilidades de equipo industrial de calidad.

Un equipo autodirigido entiende en forma consistente sus metas y objetivos generales. Define funciones y responsabilidades para cada uno de sus miembros; registra datos cuantitativos del proyecto (como productividad y calidad); identifica un proceso de equipo apropiado para el proyecto y una estrategia para implementar el proceso; define estándares locales aplicables al trabajo de ingeniería de software del equipo, evalúa en cada momento riesgos y reacciones; y registra, gestiona y reporta el estatus del proyecto.

> "Encontrar buenos jugadores es fácil. Hacer que jueguen en equipo es otra historia."
>
> **Casey Stengel**

PUNTO CLAVE

El PSE define las siguientes actividades del marco de trabajo: lanzamiento, diseño de alto nivel, implementación, integración y prueba, y análisis de resultados. Al igual que sus contrapartes en el PSP (nótese que la terminología es diferente), estas actividades permiten al equipo planear, diseñar y construir un software de una manera disciplinada al mismo tiempo que miden de modo cuantitativo el proceso y el producto. Los análisis de resultados muestran el escenario para el mejoramiento del proceso.

El PSE utiliza una amplia variedad de escritos, formas y estándares que sirven para guiar a los miembros del equipo en su trabajo. Los *escritos* definen actividades específicas del proceso (por ejemplo, lanzamiento, diseño, implementación, integración y prueba, y análisis de resultados del proyecto) y otras funciones más detalladas del trabajo (como planeación del desarrollo, desarrollo de requisitos, gestión de la configuración de software y prueba de unidad) que son parte del proceso del equipo. Con fines ilustrativos, es útil tomar en cuenta la actividad inicial del proceso: el *lanzamiento del proyecto*.

Cada proyecto es "lanzado" con una secuencia de tareas (definida como un escrito) que permite al equipo establecer una base sólida para iniciar el proyecto. Se recomienda el siguiente *escrito de lanzamiento* (sólo de manera general) [HUM00]:

- Revisar los objetivos del proyecto con la gestión y acordar y documentar las metas del equipo.
- Establecer las funciones del equipo.
- Definir el proceso de desarrollo del equipo.
- Elaborar un plan de calidad y plantear los objetivos de calidad.
- Preparar un plan para las necesidades de soporte necesarias.
- Producir una estrategia de desarrollo general.
- Elaborar un plan de desarrollo para el proyecto en su totalidad.
- Hacer planes detallados para cada ingeniero en la siguiente fase.
- Adaptar los planes individuales a un plan de equipo.

- Hacer un balance de la cantidad de trabajo del equipo para obtener un programa mínimo en términos generales.
- Valorar los riesgos del proyecto y asignar responsabilidad de rastreo para cada riesgo clave.

Es importante señalar que la actividad de lanzamiento puede aplicarse antes de cada actividad del marco de trabajo del PSE, el cual se explicó párrafos atrás. Esto se ajusta a la naturaleza iterativa de muchos proyectos y permite que el equipo se adapte a las necesidades cambiantes del cliente y a las lecciones aprendidas de actividades previas.

El PSE reconoce que los mejores equipos de software son autodirigidos. Los miembros del equipo plantean los objetivos del proyecto, adaptan el proceso para cubrir sus necesidades, controlan el programa y la medición y el análisis de las medidas recolectadas; además, trabajan de manera continua para mejorar el enfoque del equipo respecto de la ingeniería del software.

Al igual que el PSP, el PSE es un enfoque riguroso para la ingeniería del software que ofrece beneficios distintos y cuantificables en productividad y calidad. El equipo debe comprometerse con el proceso y debe recibir capacitación para asegurarse de que el enfoque se aplique de manera apropiada.

2.7 Tecnología del proceso

Los modelos genéricos de proceso tratados en las secciones precedentes deben adaptarse para que los utilice un equipo de proyecto de software. Para lograrlo se han desarrollado *herramientas de tecnología del proceso* destinadas a ayudar a las organizaciones de software a analizar sus procesos actuales, organizar sus tareas, controlar y monitorear su progreso, y administrar su calidad técnica [NEG99].

Las herramientas de tecnología del proceso permiten que una organización de software construya un modelo automatizado del marco de trabajo común del proceso, de los conjuntos de tareas y las actividades sombrilla explicadas en la sección 2.2. El modelo, que a menudo se representa como una red, entonces puede analizarse para determinar el flujo de trabajo típico y examinar las estructuras de proceso alternativas que podrían conducir a la reducción del tiempo o costo del desarrollo.

Una vez creado un proceso aceptable es posible utilizar otras herramientas de tecnología del proceso para localizar, monitorear e incluso controlar todas las tareas de ingeniería del software definidas como una parte del modelo del proceso. Cada miembro del equipo de software puede emplear dichas herramientas en la elaboración de una lista de verificación de las tareas de trabajo que se deben desarrollar, los productos del trabajo que es imperativo producir, y las actividades para el aseguramiento de la calidad que deben realizarse. La herramienta de tecnología del proceso también se puede aprovechar para coordinar el uso de otras herramientas de la ingeniería del software asistida por computadora que sean apropiadas para una tarea de trabajo particular.

HERRAMIENTAS DE SOFTWARE

Herramientas de modelado del proceso

Objetivo: Si una organización trabaja en el mejoramiento de un proceso de un negocio (o de un software), el primer objetivo es entenderlo. Las herramientas de modelado del proceso (también llamadas *tecnología del proceso* o *herramientas de gestión del proceso*) se utilizan para representar los elementos clave de un proceso para que éste pueda entenderse con mayor claridad. Tales herramientas también ofrecen vínculos con descripciones del proceso que ayudan a quienes se interesen en el proceso a entender las acciones y las tareas de trabajo necesarias para desarrollarlo. Las herramientas de modelación del proceso proporcionan vínculos con otras herramientas que ofrecen soporte a actividades definidas del proceso.

Mecánica: Las herramientas de esta categoría permiten al equipo definir los elementos de un modelo del proceso único (acciones, tareas, productos de trabajo), ofrecen una guía detallada del contenido o la descripción de cada elemento del proceso, y después gestionan el proceso mientras se conduce. En algunos casos las herramientas de tecnología del proceso incorporan tareas de gestión del proyecto estándar, como estimación, itinerario, rastreo y control.

Herramientas representativas:[11]

Igrafx Process Tools, distribuidas por Corel Corporation (www.igrafx.com/products/process), es una serie de herramientas que permiten al equipo organizar, medir y modelar el proceso de software.

Objexis Team Portal, desarrollado por Objexis Corporation (www.objexis.com), proporciona la definición y el control completos del flujo de trabajo del proceso.

2.8 PRODUCTO Y PROCESO

Si el proceso es débil, sin duda el producto final sufrirá las consecuencias. Asimismo, una confianza excesiva en el proceso es peligrosa. En un breve ensayo Margaret Davis [DAV95] comenta sobre la dualidad del producto y el proceso:

> Alrededor de cada diez años, a veces cada cinco, la comunidad del software redefine "el problema" cambiando su enfoque de los aspectos del producto a los asuntos del proceso. Entonces se han obtenido lenguajes de programación estructurados (producto) seguidos por métodos de análisis estructurado (proceso) y encapsulación de datos (producto), así como el énfasis actual en el Modelo de Madurez de Capacidad para el Desarrollo de Software del Instituto de Ingeniería del Software (proceso) [seguidos por los métodos orientados a objetos y el desarrollo ágil de software].
>
> Mientras la tendencia natural de un péndulo es llegar al reposo en el punto medio entre dos extremos, el objetivo de la comunidad del software cambia en forma constante porque cada vez que una oscilación termina se aplica una nueva fuerza. Estas oscilaciones son nocivas en sí mismas porque confunden al profesional promedio del software con los cambios radicales en el significado de hacer el trabajo o dejar que éste se desarrolle solo. Las oscilaciones tampoco resuelven "el problema", puesto que están destinadas a fallar mientras el producto y el proceso sean tratados como si formaran una dicotomía en lugar de una dualidad.

11 Las herramientas mencionadas aquí representan una muestra de esta categoría. En la mayoría de los casos los nombres son marcas registradas de sus respectivos desarrolladores.

Existen precedentes en la comunidad científica hacia las nociones de dualidad, cuando las contradicciones en las observaciones no se pueden explicar por completo con una u otra teoría competidora. La naturaleza dual de la luz, la cual parece ser en forma simultánea una partícula y una onda, ha sido aceptada desde la década de 1920, cuando Louis de Broglie la propuso. Creo que las observaciones posibles sobre los artefactos del software y su desarrollo demuestran una dualidad fundamental entre el producto y el proceso. No se puede llegar a entender el artefacto completo, su contexto, uso, significado y valor si sólo se ve como un proceso o únicamente como un producto. .

Todas las actividades humanas pueden verse como un proceso, pero cada ser humano tiene un sentido de autovaloración de aquellas actividades que generan una representación que puede emplear o apreciar más de una persona, emplear una y otra vez, o aprovechar en algún otro contexto. Es decir, el ser humano encuentra placer en reutilizar sus productos y en que otros los reutilicen.

Por lo tanto, mientras la rápida asimilación de reutilizar metas en el desarrollo de software aumenta de manera potencial la satisfacción que experimentan los profesionales de su trabajo, también aumenta la urgencia de aceptación de la dualidad del producto y el proceso. Considerar un artefacto reutilizable como sólo un producto o sólo como un proceso oscurece el contexto y las maneras de utilizarlo, u oscurece el hecho de que cada uso resulta en un producto que, a su tiempo, se aprovechará como entrada a alguna otra actividad de desarrollo de software. Poner un punto de vista sobre el otro reduce en forma sustancial las oportunidades de reutilizar y, por lo tanto, pierde la oportunidad de incrementar la satisfacción del trabajo

> "Sin duda, el sistema ideal, si se pudiera obtener, sería un código tan flexible y diminuto como para proveer por anticipado en cada situación concebible una regla exacta. Pero la vida es demasiado compleja para obtener esta idea incluso con todo el poder humano."
>
> **Benjamin Cardozo**

La gente obtiene tanta (o más) satisfacción del proceso creativo que del producto final. Un pintor disfruta los trazos del pincel tanto como el resultado del cuadro. Un escritor disfruta la búsqueda de una metáfora apropiada tanto como el libro terminado. Un profesional del software creativo debe sentir tanta satisfacción del proceso como del producto terminado.

El trabajo que realiza la gente de software cambiará en los años que siguen. La dualidad del producto y el proceso es un elemento importante para mantener a la gente creativa comprometida mientras finaliza la transición desde la programación hasta la ingeniería del software

2.9 Resumen

La ingeniería del software es una disciplina que integra al proceso, los métodos y las herramientas para el desarrollo del software de computadora. Se ha propuesto un gran número de modelos de proceso para la ingeniería del software, pero todos definen un conjunto de actividades del marco de trabajo, una colección de tareas con-

ducidas para realizar cada actividad, productos de trabajo generados como consecuencia de las tareas y un conjunto de actividades sombrilla que acompañan el proceso entero. Los patrones de proceso pueden aprovecharse para definir las características del mismo.

La integración del modelo de capacidad de madurez (IMCM) es un modelo total del proceso, que describe las metas, prácticas y capacidades específicas con que debe contar un proceso de software maduro El SPICE y otros estándares definen los requisitos para guiar una evaluación del proceso de software, y el estándar ISO 9001:2000 examina la gestión de la calidad dentro de un proceso.

Se han propuesto los modelos personal y en equipo para el proceso de software. Ambos destacan la medición, la planeación y la autodirección como ingredientes clave para un proceso de software exitoso.

Los principios, conceptos y métodos que permiten realizar el proceso llamado *ingeniería del software* serán considerados en el resto de este libro.

Referencias

[AMB98] Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press/SIGS Books, 1998.

[BAE98] Baetjer, Jr., H., *Software as Capital*, IEEE Computer Society Press, 1998, p. 85

[CIA01] Cianfrani, C. *et al.*, *ISO 9001:2000 Explained*, American Society of Quality, 2001

[CMM02] *Capability, Maturity Model Integration (CMMI)*, Versión 1.1, Software Engineering Institute, marzo de 2002, disponible en http://www.sei.cmu.edu/cmmi/.

[DAV95] Davis, M., "Process and Product: Dichotomy or Duality", en *Software Engineering Notes*, ACM Press, vol. 20, núm. 2, abril de 1995, pp 17-18

[DUN01] Dunaway, D. y S Masters, *CMM-Based Appraisal for Internal Process Improvement (CBA IPI Version 1.2 Method Description*, Software Engineering Institute, 2001, puede descargarse de http://www.sei.cmu.edu/publications/documents/01.reports/01tr033.html

[ELE98] El Emam, K., J. Drouin y W. Melo (eds.), *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE Computer Society Press, 1998

[FER97] Ferguson, P. *et al.*, "Results of applying the personal software process", en *IEEE Computer*, vol. 30, núm. 5, mayo de 1997, pp. 24-31.

[HUM96] Humphrey, W., "Using a Defined and Measured Personal Software Process", en *IEEE Software*, vol. 13, núm. 3, mayo-junio de 1996, pp 77-88.

[HUM97] Humphrey, W., *Introduction to the Personal Software Process*, Addison-Wesley, 1997.

[HUM98] Humphrey, W., "The Three Dimensions of Process Improvement, Part III: The Team Process", en *Crosstalk*, abril de 1998. Disponible en http://www.stsc.hill.af.mil/crosstalk/1998/apr/dimensions.asp

[HUM00] Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley, 2000

[IEE93] *IEE Standards Collection: Software Engineering*, IEE Standard 610.12-1990, IEEE, 1993.

[ISO00] *ISO 9001:2000 Document Set*, International Organization for Standards, 2000, http://www.iso.ch/iso/en/iso9000-14000/iso9000/9000isoindex.html

[ISO01] "Guidance on the Process Approach to Quality Management Systems", Document ISO/TC 176/SC2/N544R, International Organization for Standards, mayo de 2001

[KET01] Ketola, J. y K. Roberts, *ISO 9001:2000 in a Nutshell*, 2a. ed., Paton Press, 2001

[MON01] Monnich, H., Jr. y H. Monnich, ISO 9001:2000 for Small-and Medium-Sized Businesses, American Society of Quality, 2001.

[NAU69] Naur, P. y B. Randall (eds.), *Software Engineering. A Report on a Conference Sponsored by the NATO Science Comittee*, NATO, 1969

[NEG99] Negele, H., "Modeling of Integrated Product Development processes", *Proc. 9th Annual Symposium of INCOSE*, Reino Unido, 1999

[PAU93] Paulk, M. *et al.*, *Capability Maturity Model for Software*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[PHI02] Phillips, M., "CMMI V1.1 Tutorial", abril de 2002, disponible en http://www.sei.cmu.edu/cmmi/.

[SEI00] *SCAMPI, V1.0 Standard CMMI ® Assessment Method for Process Improvement: Method Description*, Software Engineering Institute, Technical Report CMU/SEI-2000-TR-009, disponible en http//www.sei.cmu.edu/publications/documents/00.reports/00tr009.html

[SPI99] "SPICE: Software Process Assessment, Part 1: Concepts and Introduction", Versión 1.0, ISO/IEC JTC1, 1999

Problemas y puntos a considerar

2.1. En la introducción a este capítulo, Baetjer puntualiza: "El proceso ofrece una interacción entre usuarios y diseñadores, entre usuarios y herramientas en evolución, entre diseñadores y herramientas en evolución [tecnología]". Háganse cinco preguntas respecto a *a*) lo que los diseñadores deben preguntar a los usuarios; *b*) los usuarios deben preguntar a los diseñadores; *c*) lo que los usuarios deben preguntarse a sí mismos sobre el producto de software que se construirá; y *d*) lo que los diseñadores deben preguntarse a sí mismos sobre el producto de software que se construirá y el proceso que se utilizará para hacerlo.

2.2. En la figura 2.1 se colocan los tres estratos de ingeniería del software arriba de un estrato titulado "un enfoque en la calidad". Esto implica un programa de calidad de una organización amplia como gestión de la calidad total. Realizar una pequeña investigación y desarrollar una guía de los principios clave de un programa de gestión de calidad total.

2.3. ¿Existe la posibilidad de que las actividades genéricas del proceso de ingeniería del software no se apliquen? Si es así, descríbase.

2.4. Las actividades sombrilla ocurren a lo largo de todo el proceso del software. ¿Se aplican de modo uniforme a través del proceso o algunas están concentradas en una o más actividades del marco de trabajo?

2.5. Descríbase un marco de trabajo del proceso con palabras propias. Cuando se dice que las actividades del marco de trabajo son aplicables a todos los proyectos, ¿esto significa que las mismas tareas de trabajo se aplican a todos los proyectos, sin importar el tamaño y complejidad? Explíquese la respuesta.

2.6. Intente establecer un conjunto de tareas para la actividad de *comunicación*.

2.7. Investigar un poco más acerca de la IMCM y discutir las ventajas y desventajas de los modelos de la IMCM continuo y discreto.

2.8. Desplegar la documentación de la IMCM del sitio de la red del SEI y seleccionar un área del proceso que no sea la planeación del proyecto. Hacer una lista de las metas específicas (ME) y de las prácticas específicas (PE) asociadas que se definan mediante el área que se haya elegido.

2.9. Considerar la actividad de *comunicación* dentro del marco de trabajo. Desarrollar un patrón completo del proceso (podría ser un patrón discreto) aprovechando los principios descritos en la sección 2.4.

2.10. ¿Cuál es el propósito de la evaluación del proceso? ¿Por qué el SPICE ha sido desarrollado como un estándar para la evaluación del proceso?

2.11. Investigar más sobre el PSP y preparar una breve presentación que indique los beneficios cuantitativos del proceso.

2.12. La utilización de "escritos" (un mecanismo requerido en el PSE) no goza de gran aceptación entre la comunidad del software. Hacer una lista de las ventajas y desventajas mientras se toman en cuenta los escritos y sugerir al menos dos situaciones en que serían útiles y otras dos situaciones en donde no tendrían tantos beneficios.

Otras lecturas y fuentes de información

El estado actual de la ingeniería del software y el proceso de software lo determinan bien publicaciones mensuales como *IEEE Software*, *Computer*, y *IEEE Transactions on Software Engineering*. Publicaciones periódicas como *Application Development Trends* y *Cutter IT Journal* a menudo contienen artículos sobre temas de ingeniería del software. La disciplina se "resume" cada año en la *Proceeding of the International Conference on Software Engineering*, patrocinado por el IEEE y ACM, y se discute a profundidad en publicaciones como *ACM Transactions on Software Engineering and Methodology*, *ACM Software Engineering Notes* y *Annals of Software Engineering*. Miles de páginas de la red están dedicadas a la ingeniería del software y al proceso de software.

En los años recientes se han publicado muchos libros referentes al proceso de software y a la ingeniería del software. Algunos presentan un panorama del proceso en su totalidad, mientras otras centran su atención en unos cuantos temas importantes y excluyen otros. Entre las propuestas más populares se encuentran:

Abran, A. y J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.

Ahern, D, *et al.*, *CMMI Distilled*, Addison-Wesley, 2001.

Chrisis, B *et al.*, *CMMI: Guidelines for Process Integration and Product Improvement*, Addison-Wesley, 2003.

Christensen, M. y R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.

Glass, R., *Fact and Fallacies of Software Engineering*, Addison-Wesley, 2002

Hunter, R. y R. Thayer (eds.), *Software Process Improvement*, IEEE-CS Press (Wiley), 2001

Persse, J., *Implementing the Capability Maturity Model*, Wiley, 2001.

Pfleeger, S., *Software Engineering: Theory and Practice*, 2a. ed., Prentice-Hall, 2001.

Potter, N. y M. Sakry, *Making Process Improvement Work*, Addison-Wesley, 2002

Sommerville, I., *Software Engineering*, 6a ed., Addison-Wesley, 2000

En lo que respecta a lecturas más ligeras, un libro de Robert Glass (*Software Conflict*, Yourdon Press, 1991) presenta ensayos sorprendentes y controversiales sobre el software y el proceso de ingeniería del software. Yourdon (*Death March Projects*, Prentice-Hall, 1997) expone lo que sale mal cuando fallan grandes proyectos de software y cómo evitar esos errores.

Garmus (*Measuring the Software Process*, *Prentice-Hall, 1995*) y Florac y Carlton (*Measuring the Software Process*, Addison-Wesley, 1999) explican cómo evaluar de modo estadístico la eficacia de cualquier proceso de software.

Se ha publicado una gran variedad de procedimientos y estándares de la ingeniería del software desde la década pasada. El *IEEE Software Engineering Standards* contiene muchos estándares diferentes que cubren casi cada uno de los aspectos importantes de la tecnología. El conjunto de documentos ISO 9001:2000 proporciona una guía a las organizaciones de software que deseen mejorar sus actividades de gestión de calidad. Otros estándares de ingeniería del software se pueden obtener del Departamento de Defensa, la FAA y otras agencias gubernamentales y no lucrativas de Estados Unidos de América. Fairclough (*Software Engineering Guides*, Prentice-Hall, 1996) ofrece una referencia detallada de estándares de ingeniería del software producida por la Agencia Espacial Europea (ESA, por sus siglas en inglés).

En Internet está disponible una gran variedad de fuentes de información sobre ingeniería del software y el proceso de software. Una lista actualizada de referencias de la World Wide Web relevantes para el proceso de software puede encontrarse en el sitio **http://www.mhhe.com/pressman.**

CAPÍTULO

3 MODELOS PRESCRIPTIVOS DE PROCESO

CONCEPTOS CLAVE

Los modelos prescriptivos de proceso se propusieron originalmente para ordenar el caos del desarrollo de software. La historia ha indicado que estos modelos convencionales han traído consigo cierta cantidad de estructuras útiles para el trabajo en la ingeniería del software, y han proporcionado un camino a seguir razonablemente efectivo para los equipos de software. Sin embargo, el trabajo de la ingeniería del software y el producto resultante aún permanecen "al borde del caos" [NOG00].

En un documento intrigante sobre la extraña relación entre el orden y el caos en el mundo del software, Nogueira y sus colegas establecen:

> El borde del caos se define como "un estado natural entre el orden y el caos, una relación estrecha entre la estructura y la sorpresa" [KAU95]. El borde del caos se puede visualizar como un estado inestable, estructurado en forma parcial. . . es inestable porque es atraído de manera constante hacia el caos o hacia el orden absoluto.
>
> Se tiende a pensar que el orden es el estado ideal de la naturaleza. Esto podría ser un error. La investigación... apoya la teoría de que la operación lejos del equilibrio genera creatividad, procesos organizados por sí mismos y retroalimentación creciente [ROO96]. El orden absoluto significa la ausencia de variabilidad, lo cual sería una ventaja en ambientes impredecibles. El cambio ocurre cuando existe alguna estructura para que pueda organizarse, dicha estructura no debe ser tan rígida como para que evite el cambio. Por otro lado, demasiado caos puede imposibilitar la coordinación y la coherencia. La falta de estructura no siempre significa desorden.

UN VISTAZO RÁPIDO

¿Qué es? Los modelos prescriptivos de proceso definen un conjunto distinto de actividades, acciones, tareas, fundamentos y productos de trabajo que se requieren para desarrollar software de alta calidad. Estos modelos de proceso no son perfectos, pero proporcionan una guía útil para el trabajo de la ingeniería del software.

¿Quién lo hace? Los ingenieros de software y sus gerentes adaptan un modelo prescriptivo de proceso a sus necesidades y después lo siguen. Además, la gente que ha solicitado el software tiene un papel por desempeñar conforme se ejecuta el modelo de software.

¿Por qué es importante? Porque proporciona estabilidad, control y organización a una actividad que, si no se controla puede volverse caótica. Algunas veces los modelos de proceso prescriptivo se han referido como "modelos rigurosos de proceso", ya que a menudo incluyen las capacidades sugeridas por la IMCM (capítulo 2). Sin embargo, todos los modelos de proceso se pueden adaptar para usarlos de forma efectiva y en un proyecto de software específico.

¿Cuáles son los pasos? El proceso conduce a un equipo de software a través de un conjunto de actividades del marco de trabajo que se organizan en un flujo de proceso, el cual puede ser lineal, incremental o evolutivo. La terminología y los detalles de cada modelo de proceso difieren, pero las actividades genéricas del marco de trabajo permanecen razonablemente consistentes.

¿Cuál es el producto obtenido? Desde el punto de vista de un ingeniero de software, los productos de trabajo son los programas, documentos y datos que se producen como consecuencia de las actividades y tareas que define el proceso

¿Cómo puedo estar seguro de que lo he hecho correctamente? Existe cierta cantidad de mecanismos para la evaluación del proceso de software que permite a las organizaciones determinar la "madurez" de sus respectivos procesos. Sin embargo, los mejores indicadores de la eficacia del proceso que se utiliza son la calidad, el tiempo de entrega y la viabilidad a largo plazo del producto que se construye

Las implicaciones filosóficas de este argumento son significativas respecto de la ingeniería del software. Si los modelos prescriptivos del proceso[1] buscan estructura y orden, ¿éstos resultan inapropiados para un mundo del software que se basa en el cambio? Además, si se rechazan los modelos convencionales del proceso (y el orden que éstos implican) y se reemplazan con algo menos estructurado, ¿se imposibilita alcanzar la coordinación y la coherencia en el trabajo del software?

No existen respuestas fáciles, pero existen opciones disponibles para los ingenieros de software. En este capítulo se examinará el enfoque del proceso prescriptivo, en el cual el orden y la consistencia del proyecto son los aspectos dominantes. En el capítulo 4 se examinará el enfoque del proceso ágil en el cual la organización propia, la colaboración, la comunicación y la adaptabilidad dominan la filosofía del proceso.

3.1 Modelos prescriptivos

CLAVE

Cualquier organización de ingeniería del software debe describir un conjunto único de actividades dentro del marco de trabajo (capítulo 2) para el (los) proceso(s) de software que adopte. También debe llenar cada actividad del marco de trabajo con un conjunto de acciones de ingeniería del software, y definir cada acción en cuanto a un conjunto de tareas que identifique el trabajo (y los productos del trabajo) que deben completarse para alcanzar las metas de desarrollo. Después, la organización debe adaptar el modelo de proceso resultante y ajustarlo a la naturaleza específica de cada proyecto, a las personas que lo realizarán, y el ambiente en el que se ejecutará el trabajo. Sin importar el modelo del proceso seleccionado, los ingenieros de software han elegido de manera tradicional un marco de trabajo genérico para el proceso, el cual incluye las siguientes actividades dentro del marco: comunicación, planeación, modelado, construcción y desarrollo

"Existen muchas formas de ir hacia delante, pero sólo una de permanecer."

Franklin D. Roosevelt

1 Los modelos prescriptivos de proceso a menudo se denominan modelos "convencionales" de proceso

CONSEJO

Aunque un proceso sea prescriptivo, no se debe asumir que éste es estático. Los modelos prescriptivos se deben adaptar a las personas, al problema y al proyecto.

En las secciones siguientes se examinarán varios de los modelos prescriptivos del proceso de software. Se les llama "prescriptivos" porque prescriben un conjunto de elementos del proceso· actividades del marco de trabajo, acciones de ingeniería del software, tareas, productos del trabajo, aseguramiento de la calidad, y mecanismos de control del cambio para cada proyecto. Cada modelo de proceso prescribe también un *flujo de trabajo;* esto es, la forma en la cual los elementos del proceso se interrelacionan entre sí.

Todos los modelos de proceso del software se ajustan a las actividades genéricas del marco de trabajo descritas en el capítulo 2, pero cada uno aplica una importancia diferente a estas actividades y define un flujo de trabajo que invoca cada actividad del marco de trabajo (así como acciones y tareas de la ingeniería del software) de una manera diferente.

3.2 EL MODELO EN CASCADA

Existen ocasiones en que los requisitos de un problema se entienden de una manera razonable· cuando el trabajo fluye desde la comunicación a través del despliegue de una manera casi lineal. Esta situación se encuentra a veces cuando es necesario hacer adaptaciones o mejorías bien definidas a un sistema existente (por ejemplo, una adaptación a un software contable debido a los cambios en las regulaciones del gobierno). Esto puede ocurrir también en un número limitado de proyectos de nuevos desarrollos, pero sólo cuando los requerimientos están bien definidos y son estables en forma razonable.

El *modelo en cascada*, algunas veces llamado el *ciclo de vida clásico*, sugiere un enfoque sistemático, secuencial[2] hacia el desarrollo del software, que se inicia con la especificación de requerimientos del cliente y que continúa con la planeación, el modelado, la construcción y el despliegue para culminar en el soporte del software terminado.

El modelo en cascada es el paradigma más antiguo para la ingeniería del software. Sin embargo, en las décadas pasadas, las críticas a este modelo de proceso han

FIGURA 3.1 El modelo en cascada.

2 A pesar de que el modelo en cascada original, que propuso Winston Royce [ROY70], prevé "ciclos de retroalimentación", la inmensa mayoría de las organizaciones que aplica este modelo de proceso lo trata como si fuera estrictamente lineal

ocasionado que aun sus más fervientes practicantes hayan cuestionado su eficacia [HAN95]. Entre los problemas que algunas veces se encuentran al aplicar el modelo en cascada están:

¿Por qué algunos fallan el modelo en cascada?

1. Es muy raro que los proyectos reales sigan el flujo secuencial que propone el modelo. A pesar de que el modelo lineal incluye iteraciones, lo hace de manera indirecta. Como resultado, los cambios confunden mientras el equipo de proyecto actúa.
2. Con frecuencia es difícil para el cliente establecer todos los requisitos de manera explícita. El modelo en cascada lo requiere y se enfrentan dificultades al incorporar la incertidumbre natural presente en el inicio de muchos proyectos.
3. El cliente debe tener paciencia. Una versión que funcione de los programas estará disponible cuando el proyecto esté muy avanzado. Un error grave será desastroso si no se detecta antes de la revisión del programa.

En un análisis interesante de proyectos reales, Bradac [BRA94] concluyó que la naturaleza lineal del modelo en cascada conduce a "estados de bloqueo" en los cuales algunos miembros del equipo del proyecto deben esperar a otros para terminar tareas dependientes. De hecho, el tiempo de espera puede superar el que se aplica en el trabajo productivo. El estado de bloqueo tiende a ser más común al principio y al final del proceso secuencial.

En la actualidad, el trabajo del software está acelerado y sujeto a una cadena infinita de cambios (de características, funciones y contenido de la información). Con frecuencia, el modelo en cascada no es apropiado para dicho trabajo. Sin embargo, puede servir como un modelo de proceso útil en situaciones donde los requerimientos están fijos y donde el trabajo se realiza, hasta su conclusión, de una manera lineal.

3.3 MODELOS DE PROCESO INCREMENTALES

En muchas situaciones los requisitos iniciales del software están bien definidos en forma razonable, pero el enfoque global del esfuerzo de desarrollo excluye un proceso puramente lineal. Además, quizá haya una necesidad imperiosa de proporcionar de manera rápida un conjunto limitado de funcionalidad para el usuario y después refinarla y expandirla en las entregas posteriores del software. En estos casos se elige un modelo de proceso diseñado para producir el software en forma incremental.

> "Con demasiada frecuencia, el trabajo en el software sigue la primera ley del ciclismo: no importa hacia dónde vayas, es montaña arriba y contra el viento."
>
> Anónimo

3.3.1 El modelo incremental

PUNTO CLAVE

El modelo incremental entrega una serie de lanzamientos, llamados *incrementos*, que proporcionan en forma progresiva más funcionalidad para los clientes a medida que se entrega cada uno de los incrementos.

El *modelo incremental* combina elementos del modelo en cascada aplicado en forma iterativa. Como se muestra en la figura 3.2, el modelo incremental aplica secuencias lineales de manera escalonada conforme avanza el tiempo en el calendario. Cada secuencia lineal produce "incrementos" del software [MCD93]. Por ejemplo, el software procesador de texto, desarrollado con el paradigma incremental en su primer incremento, podría realizar funciones básicas de administración de archivos, edición y producción de documentos; en el segundo incremento, ediciones más sofisticadas, y tendría funciones más complejas de producción de documentos; en el tercer incremento, funciones de corrección ortográfica y gramatical; y en el cuarto, capacidades avanzadas de configuración de página. Se debe tener en cuenta que el flujo del proceso de cualquier incremento puede incorporar el paradigma de construcción de prototipos que se expone en la sección 3.4.1.

CONSEJO

Si el cliente demanda la entrega en una fecha imposible de cumplir, sugiérase entregar uno o más incrementos para esa fecha y el resto del software (incrementos adicionales) después

A menudo, al utilizar un modelo incremental el primer incremento es un *producto esencial*. Es decir, se incorporan los requisitos básicos, pero muchas características suplementarias (algunas conocidas, otras no) no se incorporan. El producto esencial queda en manos del cliente (o se somete a una evaluación detallada). Como resultado de la evaluación se desarrolla un plan para el incremento siguiente. El plan afronta la modificación del producto esencial con el fin de satisfacer de mejor manera las necesidades del cliente y la entrega de características y funcionalidad adicionales. Este proceso se repite después de la entrega de cada incremento mientras no se haya elaborado el producto completo.

El modelo de proceso incremental, al igual que la construcción de prototipos y otros enfoques evolutivos, es iterativo por naturaleza. Pero a diferencia de la construcción de prototipos, el modelo incremental se enfoca en la entrega de un producto operacional con cada incremento. Los primeros incrementos son versiones

FIGURA 3.2

El modelo incremental.

"incompletas" del producto final, pero proporcionan al usuario la funcionalidad que necesita y una plataforma para evaluarlo.[3]

El desarrollo incremental es útil sobre todo cuando el personal necesario para una implementación completa no está disponible. Los primeros incrementos se pueden implementar con menos gente. Si el producto esencial es bien recibido se agrega (si se requiere) más personal para implementar el incremento siguiente. Además, los incrementos se pueden planear para manejar los riesgos técnicos. Por ejemplo, un sistema grande podría requerir la disponibilidad de un hardware nuevo que está en desarrollo y cuya fecha de entrega es incierta. Sería posible planear los primeros incrementos de forma que se evite el uso de este hardware, lo que permitiría la entrega de funcionalidad parcial a los usuarios finales sin retrasos desordenados.

3.3.2 El modelo DRA

El *desarrollo rápido de aplicaciones* (DRA) es un modelo de proceso de software incremental que resalta un ciclo de desarrollo corto. El modelo DRA es una adaptación a "alta velocidad" del modelo en cascada en el que se logra el desarrollo rápido mediante un enfoque de construcción basado en componentes. Si se entienden bien los requisitos y se limita el ámbito del proyecto,[4] el proceso DRA permite que un equipo de desarrollo cree un "sistema completamente funcional" dentro de un periodo muy corto (por ejemplo, de 60 a 90 días) [MAR91].

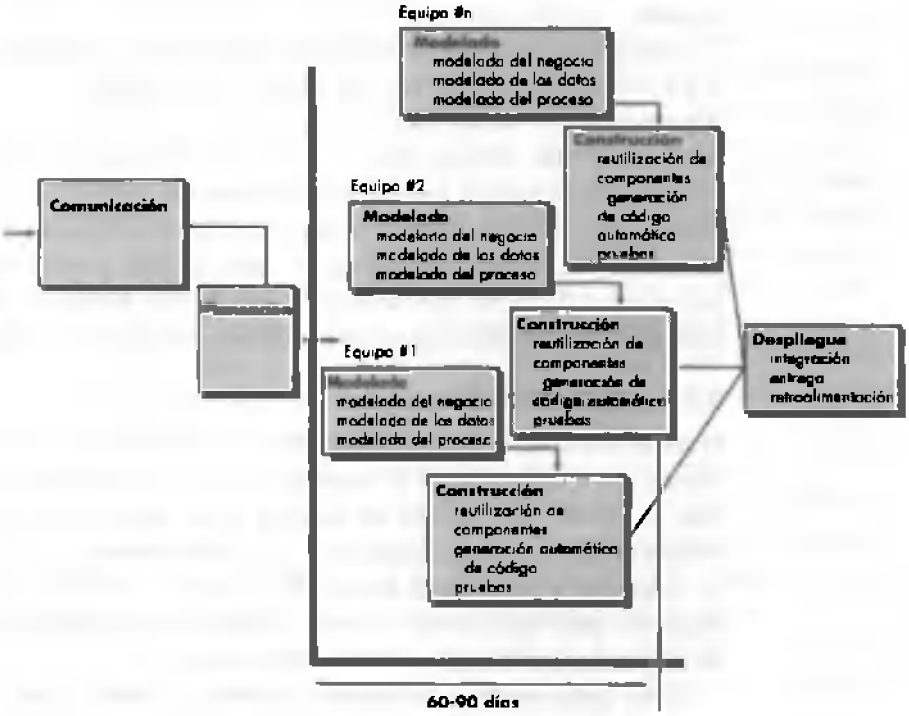
Como otros modelos de proceso, el enfoque DRA cumple con las actividades genéricas del marco de trabajo que ya se han presentado. La *comunicación* trabaja para entender el problema de negocios y las características de información que debe incluir el software. La *planeación* es esencial porque varios equipos de software trabajan en paralelo sobre diferentes funciones del sistema. El *modelado* incluye tres grandes fases —modelado de negocios, modelado de datos y modelado del proceso— y establece representaciones del diseño que sirven como base para la actividad de construcción del modelo DRA. La *construcción* resalta el empleo de componentes de software existente y la aplicación de la generación automática de código. Por último, el *despliegue* establece una base para las iteraciones subsecuentes, si éstas son necesarias [KER94].

El modelo de proceso DRA se ilustra en la figura 3.3. Es obvio que las restricciones de tiempo impuestas sobre un proyecto DRA exigen un "ámbito de escalas" [KER94]. Si una aplicación de negocios se puede modular de forma que cada gran función pueda completarse en menos de tres meses (mediante la aplicación del enfoque ya

3 Es importante observar que para todos los modelos de proceso ágiles que se tratan en el capítulo 4 también se utiliza una filosofía incremental.

4 Estas condiciones no se pueden garantizar por ningún medio. De hecho, muchos proyectos de software tienen los requisitos muy pobremente definidos al principio. En tales casos los enfoques de construcción de prototipos o evolutivos (sección 3.4) son mejores opciones de proceso. Véase [RE195].

FIGURA 3.3
El modelo DRA.



descrito), ésta es una candidata para el DRA. Cada gran función se puede abordar mediante un equipo de DRA por separado, para después integrarlas y formar un todo.

¿Cuáles son las desventajas del modelo DRA?

Como todos los modelos de proceso, el enfoque DRA tiene inconvenientes [BUT94]: 1) para proyectos grandes, pero escalables, el DRA necesita suficientes recursos humanos para crear el número correcto de equipos DRA; 2) si los desarrolladores y clientes no se comprometen con las actividades rápidas necesarias para completar el sistema en un marco de tiempo muy breve, los proyectos de DRA fallarán; 3) si un sistema no se puede modular en forma apropiada, la construcción de los componentes necesarios para el DRA será problemática; 4) si el alto rendimiento es un aspecto importante, y se alcanzará al convertir interfases en componentes del sistema, el enfoque DRA podría no funcionar; y 5) el DRA sería inapropiado cuando los riesgos técnicos son altos (por ejemplo, cuando una aplicación nueva aplica muchas nuevas tecnologías).

3.4 MODELOS DE PROCESO EVOLUTIVOS

El software, como todos los sistemas complejos, evoluciona con el tiempo [GIL88]. Los requisitos de los negocios y productos a menudo cambian conforme se realiza el desarrollo; por lo tanto, la ruta lineal que conduce a un producto final no será real; las estrictas fechas tope del mercado imposibilitan la conclusión de un producto

CLAVE

completo, por lo que se debe presentar una versión limitada para liberar la presión competitiva y de negocios; se tiene claro un conjunto de requisitos del producto o sistema esencial, pero todavía se deben definir los detalles de las extensiones del producto o sistemas. En éstas y otras situaciones similares, los ingenieros de software necesitan un modelo de proceso que haya sido diseñado de manera explícita para incluir un producto que evolucione con el tiempo.

Los *modelos evolutivos* son iterativos; los caracteriza la forma en que permiten que los ingenieros de software desarrollen versiones cada vez más completas del software

3.4.1 Construcción de prototipos

A menudo un cliente define un conjunto de objetivos generales para el software, pero no identifica los requisitos detallados de entrada, procesamiento o salida. En otros casos, el responsable del desarrollo del software está inseguro de la eficacia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma que debería tomar la interacción humano-máquina. En éstas, y en muchas otras situaciones, un *paradigma de construcción de prototipos* puede ofrecer el mejor enfoque.

CONSEJO

A pesar de que la construcción de prototipos se puede utilizar como un modelo de proceso independiente, se emplea más comúnmente como una técnica susceptible de implementarse dentro del contexto de cualquiera de los modelos de proceso expuestos en este capítulo. Sin importar la forma en que éste se aplique, el paradigma de construcción de prototipos ayuda al ingeniero de sistemas y al cliente a entender de mejor manera cuál será el resultado de la construcción cuando los requisitos estén satisfechos.

El paradigma de construcción de prototipos (figura 3.4) se inicia con la comunicación. El ingeniero de software y el cliente encuentran y definen los objetivos glo-

Figura 3.4 El modelo de construcción de prototipos.

bales para el software, identifican los requisitos conocidos y las áreas del esquema en donde es necesaria más definición. Entonces se plantea con rapidez una iteración de construcción de prototipos y se presenta el modelado (en la forma de un diseño rápido). El diseño rápido se centra en una representación de aquellos aspectos del software que serán visibles para el cliente o el usuario final (por ejemplo, la configuración de la interfaz con el usuario y el formato de los despliegues de salida). El diseño rápido conduce a la construcción de un prototipo. Después, el prototipo lo evalúa el cliente/usuario y con la retroalimentación se refinan los requisitos del software que se desarrollará. La iteración ocurre cuando el prototipo se ajusta para satisfacer las necesidades del cliente. Esto permite que al mismo tiempo el desarrollador entienda mejor lo que se debe hacer.

De manera ideal, el prototipo debería servir como un mecanismo para identificar los requisitos del software. Si se construye un prototipo de trabajo, el desarrollador intenta emplear los fragmentos del programa ya existentes o aplica herramientas (como generadores de informes, administradores de ventanas, etcétera) que permiten producir programas de trabajo con rapidez.

Pero ¿qué debe hacerse con el prototipo una vez cumplido el propósito descrito? Brooks [BRO75] ofrece una respuesta.

> En la mayoría de los proyectos, el primer sistema construido apenas se utiliza. Tal vez sea demasiado lento, muy grande o torpe en su uso, o reúna las tres características a la vez. No existe otra opción que comenzar de nuevo, aunque sea doloroso, es lo mejor, y construir una revisión rediseñada en la que se resuelvan estos problemas .. Cuando se aplica un concepto nuevo de sistema o una tecnología nueva, se tiene que construir un sistema inservible y que sea necesario desechar, porque incluso la mejor planeación no es omnisciente como para que el sistema esté perfecto desde la primera vez. Por lo tanto, la pregunta de la administración no es si debe construirse un sistema piloto y desecharlo. Esto tendrá que hacerse. La única pregunta es si se planea la construcción de un desechable o se promete entregárselo a los clientes

El prototipo puede servir como "primer sistema", el que Brooks recomienda desechar. Pero ésta tal vez sea una visión idealizada. Es verdad que a los clientes y los desarrolladores les gusta el paradigma de construcción de prototipos. A los usuarios les gusta el sistema real y a los desarrolladores les gusta construir algo de inmediato. Sin embargo, la construcción de prototipos también se torna problemática por las siguientes razones:

1. El cliente ve lo que parece una versión en funcionamiento del software, sin saber que el prototipo está unido "con chicle y cable para embalaje", que por la prisa de hacerlo funcionar no se ha considerado la calidad del software global o la facilidad de mantenimiento a largo plazo. Cuando se informa que el producto debe construirse otra vez para mantener los altos niveles de calidad, el cliente no lo entiende y pide la aplicación de "unos pequeños ajustes" para que se pueda elaborar un producto final a partir del prototipo. Es muy frecuente que la gestión del desarrollo de software sea muy lenta.

Resístase a la presión por convertir un prototipo burdo en un producto. Como resultado, casi siempre la calidad se reduce.

2. A menudo, el desarrollador establece compromisos de implementación para lograr que el prototipo funcione con rapidez. Tal vez se utilice un sistema operativo o lenguaje de programación inadecuado sólo porque está disponible y es conocido; se puede implementar un algoritmo ineficiente sólo para mostrar capacidad. Después de un tiempo, el desarrollador quizá se familiarice con estas selecciones y olvide las razones por las que son inapropiadas. La selección menos ideal ahora se convierte en una parte integral del sistema

A pesar de que tal vez surjan problemas, la construcción de prototipos puede ser un paradigma efectivo para la ingeniería del software. La clave es definir las reglas del juego desde el principio; es decir, el cliente y el desarrollador se deben poner de acuerdo en que el prototipo se construya y sirva como un mecanismo para la definición de requisitos, en que se descarte, al menos en parte, y en que después se desarrolle el software real con un enfoque hacia la calidad.

HogarSeguro

Selección de un modelo de proceso. Primera parte

El escenario: Sala de reuniones para el grupo de ingeniería del software en CPI Corporation, una compañía (ficticia) que fabrica productos de sensores para uso doméstico y comercial.

Los actores: Lee Warren, gerente de ingeniería; Doug Miller, gerente de ingeniería del software; Jamie Lazar, miembro del equipo de software; Vinod Raman, miembro del equipo de software; y Ed Robbins, miembro del equipo de software.

La conversación:

Lee: Entonces recapitulemos. He pasado algún tiempo analizando la línea de productos *HogarSeguro* como la vemos en el momento. No hay duda, tenemos mucho trabajo que hacer sólo para definir esta cosa, pero me gustan las que comenzaron a pensar acerca de cómo enfocar la parte del software de este proyecto.

Doug: Parece que en el pasado hemos estado muy desorganizados en nuestro enfoque hacia el software.

Ed: No lo sé, Doug. Siempre sacamos el producto.

Doug: Es cierto, pero con muchos sacrificios, y este proyecto parece mayor y más complejo que cualquier cosa que hayamos hecho antes.

Jamie: No parece ser tan duro, pero estoy de acuerdo... nuestro enfoque *ad hoc* en los proyectos pasados no funcionará aquí, en particular si tenemos un tiempo de entrega muy restringido.

Doug (sonriendo): Quiero que nuestro enfoque sea un poco más profesional. Asistí a un curso corto la semana pasada y aprendí mucho acerca de la ingeniería del software. Creo que es una buena idea. Necesitamos un proceso aquí.

Jamie (frunciendo el ceño): Mi trabajo es construir programas de computadora, no andar moviendo papeles.

Doug: Dale una oportunidad antes de decirme que no. Me refiero a esto. [Doug describe el marco de trabajo del proceso expuesto en el capítulo 2 y los modelos prescriptivos del proceso que se han presentado hasta este punto.]

Doug: Entonces, me parece que un modelo lineal no es para nosotros. . . supone que tenemos todos los requisitos frente a nosotros y, conociendo este lugar, eso no es probable.

Vinod: Sí, y ese modelo DRA suena muy orientado a la IT .. Tal vez sea bueno para construir un sistema de control de inventarios o algo así, pero no es correcto para *HogarSeguro*.

Doug: Estoy de acuerdo.

Ed: Ese enfoque de construcción de prototipos me parece bueno. De cualquier manera se parece mucha a lo que hacemos aquí.

Vinod: Ése es el problema. Me preocupa que no nos proporcione la suficiente estructura.

Doug: No hay de qué preocuparse. Tenemos muchas otras opciones, y quiero que seleccionen lo mejor para el equipo y el proyecto.

3.4.2 El modelo en espiral

El *modelo en espiral*, que Boehm [BOE88] propuso originalmente, es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de la construcción de prototipos con los aspectos controlados y sistemáticos del modelo en cascada. Proporciona el material para el desarrollo rápido de versiones incrementales del software. Boehm [BOE01] describe este modelo de la siguiente manera:

> El modelo de desarrollo en espiral es un generador del modelo de proceso guiado por el riesgo que se emplea para conducir sistemas intensivos de ingeniería del software concurrente y con múltiples usuarios. Tiene dos características distintivas principales. Una de ellas es un enfoque cíclico para el crecimiento incremental del grado de definición e implementación de un sistema, mientras disminuye su grado de riesgo. La otra es un conjunto de puntos de fijación para asegurar el compromiso del usuario con soluciones de sistema que sean factibles y mutuamente satisfactorias.

¡PUNTO CLAVE

El modelo en espiral se puede adoptar y aplicarlo a través del ciclo de vida completo de una aplicación, desde el desarrollo del concepto hasta el mantenimiento.

Cuando se aplica el modelo en espiral, el software se desarrolla en una serie de entregas evolutivas. Durante las primeras iteraciones, la entrega tal vez sea un documento del modelo o un prototipo. Durante las últimas iteraciones se producen versiones cada vez más completas del sistema desarrollado.

Un proceso en espiral se divide en un conjunto de actividades del marco de trabajo que define el equipo de ingeniería del software. Para propósitos ilustrativos se aprovechan las actividades genéricas del marco de trabajo expuestas páginas atrás.[5] Cada una de las actividades del marco de trabajo representa un segmento de la ruta en espiral que se presenta en la figura 3.5. Cuando comienza este proceso evolutivo el equipo de software realiza actividades implicadas en un circuito alrededor de la

FIGURA 3.5
Un modelo en espiral típico.

5 El modelo en espiral expuesto en esta sección es una variación del modelo que propuso Boehm. Para más información sobre el modelo en espiral original, véase [BOE88]. En [BOE98] se puede encontrar una exposición más reciente del modelo en espiral de Boehm.

espiral que tiene una dirección en el sentido del movimiento de las manecillas del reloj, y que se inicia desde el centro. El riesgo (capítulo 25) es un factor considerado en cada revolución. Los *puntos de fijación* —una combinación de productos de trabajo y condiciones incluidas a lo largo de la ruta de la espiral— se consideran para cada paso evolutivo.

El primer circuito alrededor de la espiral quizá genere el desarrollo de una especificación del producto; los pasos subsecuentes alrededor de la espiral se pueden aprovechar para desarrollar un prototipo y después, en forma progresiva, versiones más elaboradas del software. Cada paso a través de la región de planeación resulta en ajustes al plan del proyecto. Los costos y el itinerario se ajustan con base en la retroalimentación derivada de la relación con el cliente después de la entrega. Además, el administrador del proyecto ajusta el número de iteraciones planeado que se requiere para completar el software.

A diferencia de otros modelos de proceso que terminan cuando se entrega el software, el modelo en espiral puede adaptarse y aplicarse a lo largo de la vida del software de computadora. Por lo tanto, el primer circuito alrededor de la espiral podría representar un "proyecto de desarrollo del concepto", el cual se inicia en el centro de la espiral y continúa por múltiples iteraciones[6] hasta que el desarrollo del concepto esté completo. Si el concepto se desarrollará en un producto real, el proceso continúa en la siguiente fase de la espiral y comienza un "proyecto de desarrollo de un producto nuevo". El nuevo producto evolucionará a través de un número de iteraciones alrededor de la espiral. Después, un circuito alrededor de la espiral se podría emplear para representar un "proyecto de mejoramiento del producto". En esencia, la espiral, cuando se caracteriza de esta forma, permanece operativa hasta que el software se retira. En ocasiones el proceso está inactivo, pero siempre que se inicie un cambio el proceso comienza en el punto de entrada aprobado (por ejemplo, mejoramiento del producto).

El modelo en espiral es un enfoque realista para el desarrollo de software y de sistemas a gran escala. Como el software evoluciona conforme avanza el proceso, el desarrollador y el cliente entienden y reaccionan de mejor manera ante los riesgos en cada etapa evolutiva. El modelo en espiral emplea la construcción de prototipos como un mecanismo encaminado a reducir riesgos pero, en forma más importante, permite al desarrollador la aplicación del enfoque de la construcción de prototipos en cualquier etapa evolutiva del producto. Mantiene el enfoque sistemático de los pasos que sugiere el ciclo de vida clásico, pero lo incorpora al marco de trabajo iterativo que refleja de forma más verídica el mundo real. El modelo en espiral exige *una consideración directa de los riesgos técnicos en todas las etapas del proyecto y, si se aplica en forma apropiada, debe reducir los riesgos antes de que se vuelvan* problemáticos.

---

[6] *Las flechas que apuntan hacia dentro a lo largo del eje que separa la región de despliegue de la de comunicación indican una posibilidad de iteración local a lo largo de la misma ruta de la espiral.*

Así como otros paradigmas, el modelo en espiral no es una panacea. Es difícil convencer a los clientes (en particular en situaciones bajo contrato) de que el enfoque evolutivo es controlable, ya que se requiere una habilidad considerable para evaluar el riesgo y se confía en dicha habilidad para obtener el éxito. Si un riesgo importante no se descubre y administra, sin duda surgirán problemas.

3.4.3 El modelo de desarrollo concurrente

El *modelo de desarrollo concurrente*, llamado algunas veces *ingeniería concurrente*, se representa en forma esquemática como una serie de actividades del marco de trabajo, acciones y tareas de la ingeniería del software y sus estados asociados. Por ejemplo, la actividad de *modelado*, definida para el modelo en espiral, se lleva a cabo al invocar las acciones siguientes: construcción de prototipos o modelado y especificación del análisis y diseño.[7]

A menudo, el modelo concurrente es más apropiado para proyectos de ingeniería de sistemas (capítulo 6), donde están implicados diferentes equipos de ingeniería.

En la figura 3.6 se proporciona un esquema de una tarea de ingeniería de software relacionada con la actividad de modelado para el modelo de proceso concurrente. La actividad —modelado— puede estar en alguno de los estados[8] destacados mencionados antes en cualquier momento dado. De forma similar, otras actividades o tareas (por ejemplo, la comunicación y la construcción) se representan de una manera análoga. Todas las actividades existen de forma concurrente, pero se encuentran en diferentes estados. Por ejemplo, al principio del proyecto la actividad de *comunicación* (no se muestra en la figura) ha completado su primera iteración y existe en el estado de **en espera de cambios**. La actividad de *modelado* que existió en el estado **ninguno** mientras se realizaba la comunicación inicial, ahora realiza una transición al estado **en desarrollo**. Sin embargo, si el cliente indica cambios en los requisitos, la actividad de *modelado* se mueve del estado **en desarrollo** al estado de **en espera de cambios.**

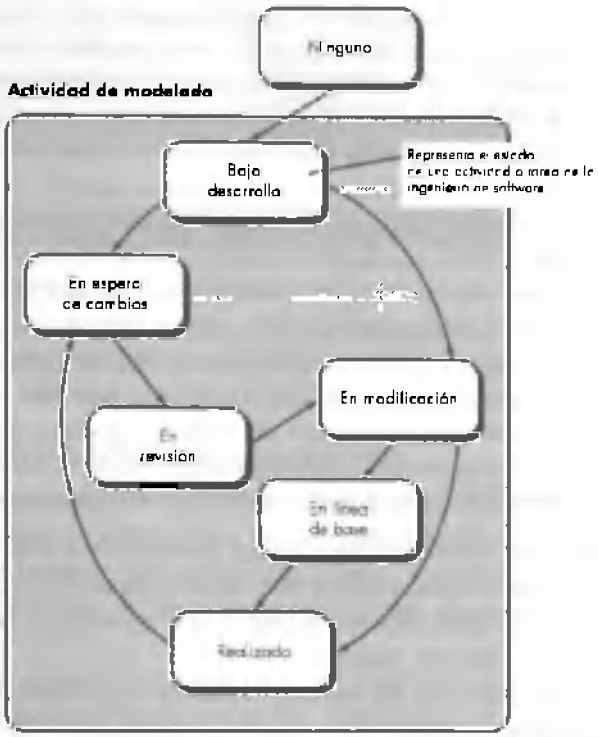
El modelo de proceso concurrente define una serie de eventos que dispararán transiciones de estado a estado para cada una de las actividades, acciones o tareas de la ingeniería del software. Por ejemplo, durante los primeros estados del diseño (una acción de la ingeniería del software que ocurre en el curso de la actividad de modelación) no se detecta una inconsistencia en el modelo del análisis. Esto genera el evento *corrección del análisis del modelo*, el cual disparará la creación del análisis desde el estado **realizado** hasta el estado de **en espera de cambios**.

El modelo de proceso concurrente se aplica a todos los tipos de desarrollo de software y proporciona una visión exacta del estado actual de un proyecto. En lugar de confinar las actividades, acciones y tareas de la ingeniería del software a una secuencia de eventos, define una red de actividades. Cada actividad, acción o tarea en la red existe de manera simultánea con otras actividades, acciones o tareas. Los

7 Se debe notar que el análisis y el diseño son acciones complejas que requieren un debate sustancial. La parte 2 de este libro considera estos tópicos en forma detallada.

8 Un *estado* es alguna forma de comportamiento observable desde el exterior.

Figura 3.6 El elemento del modelo concurrente de proceso.



eventos generados en un punto de la red del proceso disparan transiciones entre los estados

3.4.4 Un comentario final sobre los procesos evolutivos

Ya se ha detectado que al software de computadoras moderno lo caracteriza el cambio continuo, los tiempos de entrega muy reducidos, y una necesidad intensa de satisfacer al cliente/usuario. En muchos casos, el tiempo de llegada al mercado es el requisito de gestión más importante. Si se pierde una ventana del mercado, el mismo proyecto de software puede perder su significado.[9]

> "Llego hasta aquí, y sólo el mañana guía mi camino."
>
> **Dave Matthews Band**

Los modelos evolutivos de proceso se concibieron para abocarse a estos aspectos y, además, como modelos de proceso de clase general. Estos modelos también tienen debilidades, las cuales resumen Nogueira y sus colegas [NOG00]:

9 Sin embargo, es importante notar que llegar en primer lugar a un mercado no garantiza el éxito. De hecho, muchos productos de software muy exitosos han sido los segundos o hasta los terceros en llegar al mercado (al aprender errores de los antecesores).

A pesar de los incuestionables beneficios de los procesos evolutivos de software, se tienen algunas dificultades con este tipo de procesos. La primera dificultad es que la construcción de prototipos [y otros procesos evolutivos más elaborados] implican un problema para la planeación del proyecto debido al número incierto de ciclos requeridos para construir el producto. La mayoría de las técnicas de gestión y estimación de proyectos se basa en configuraciones lineales de las actividades, por lo que éstas no se ajustan por completo.

La segunda dificultad es que los procesos evolutivos de software no establecen la velocidad máxima de la evolución. Si las evoluciones suceden demasiado rápido, sin un periodo de relajación, existe la certidumbre de que el proceso caerá en el caos. Por otro lado, si la velocidad es demasiado lenta, se podría afectar la productividad.

Una tercera dificultad es que los procesos de software se deben enfocar en la flexibilidad y extensibilidad en vez de en la alta calidad. Esta afirmación suena atemorizante. Sin embargo, se debe priorizar la velocidad del desarrollo sobre los cero defectos. Si el desarrollo se extiende para alcanzar una alta calidad, se produciría un retraso en la entrega del producto, la cual se haría cuando el nicho de oportunidad ya haya desaparecido. Este cambio de paradigma lo impone la competencia en el borde del caos.

En efecto, un proceso de software que se enfoca en la flexibilidad y la velocidad del desarrollo por encima de la alta calidad suena atemorizante. Aun así, esta idea la ha propuesto cierto número de respetados expertos en la ingeniería del software (por ejemplo, [YOU95], [BAC97]).

El propósito de los modelos evolutivos es desarrollar software de alta calidad[10] de una manera iterativa o incremental. Sin embargo, es posible aplicar un proceso evolutivo para destacar la flexibilidad, extensibilidad y velocidad del desarrollo. El reto para los equipos de software y sus dirigentes es establecer un balance apropiado entre estos parámetros críticos del proyecto y el producto, y el producto y la satisfacción del cliente (el árbitro final de la calidad del software).

HogarSeguro

Selección de un modelo de proceso. Segunda parte

El escenario: Sala de reuniones para el grupo de ingeniería del software en CPI Corporation, una compañía que fabrica productos de consumo para uso doméstico y comercial.

Los actores: Lee Warren, gerente de ingeniería; Doug Miller, gerente de ingeniería del software; Ed y Vinod, miembros del equipo de ingeniería del software.

La conversación:

(Doug describe las opciones de procesos evolutivos.)

Ed: Ahora veo algo que me gusta. Un enfoque incremental tiene sentido y en realidad me gusta el flujo de ese modelo en espiral. Eso lo hace realista.

Vinod: Estoy de acuerdo. Entregamos un incremento, aprendemos de la retroalimentación del cliente, replaneamos, y después entregamos otro incremento. También se ajusta a la naturaleza del producto. Podemos tener algo en el mercado muy rápido y después agregar funcionalidad con cada versión. Es decir, con cada incremento.

10 En este contexto, la calidad del software se define con mucha amplitud para abarcar no sólo la satisfacción del cliente, sino también una variedad de criterios técnicos tratados en el capítulo 26.

...minuto, Doug, ¿dijeron que ... con cada vuelta alrededor de la ... muy bueno, necesitamos un plan, un ... apegamos a él.

Doug: Lee, ese es un pensamiento de la vieja escuela. Como dijo Ed, debemos mantenerlo realista. Digo que es mejor mover el plan a medida que aprendemos más y conforme soliciten los cambios.

3.5 MODELOS ESPECIALIZADOS DE PROCESO

Los modelos especializados de proceso adoptan muchas de las características de uno o más de los modelos convencionales presentados en las secciones anteriores. Sin embargo, los modelos especializados tienden a aplicarse cuando se ha elegido un enfoque de ingeniería del software definido de una manera muy estrecha.[11]

3.5.1 Desarrollo basado en componentes

Los nuevos componentes de software comerciales (NCSC), desarrollados por vendedores que los ofrecen como productos, se pueden emplear cuando el software está en proceso de construcción. Estos componentes proporcionan funcionalidad dirigida con interfaces bien definidas que permiten que el componente se integre en el software.

El modelo de *desarrollo basado en componentes* (DBC) (capítulo 30) incorpora muchas de las características del modelo en espiral. Es evolutivo por naturaleza [NIE92] y exige un enfoque iterativo para la creación del software. Sin embargo, el modelo configura aplicaciones a partir de componentes de software empaquetados en forma previa.

Las actividades de modelado y construcción comienzan con la identificación de los componentes candidatos. Estos componentes se pueden diseñar como módulos de software convencionales o como clases o paquetes de clases orientados a objetos.[12] Sin importar la tecnología que se aplique en la creación de los componentes, el modelo de desarrollo basado en componentes incorpora los siguientes pasos (implementados mediante un enfoque evolutivo):

- Los productos basados en componentes disponibles se investigan y evalúan para el dominio de aplicación en cuestión.
- Se consideran los aspectos de integración de componentes.
- Se diseña una arquitectura de software (capítulo 10) para adaptar los componentes.

11 En algunos casos estos modelos especializados de proceso se pueden describir de mejor manera como una colección de técnicas o una metodología para alcanzar una meta específica del desarrollo del software. Sin embargo, éstas implican un proceso.

12 La tecnología orientada a objetos se trata a lo largo de la parte 2 de este libro. *En este contexto, una clase encapsula una serie de datos y los procedimientos que procesan dichos datos. Un paquete de clases* es una colección de clases relacionadas que trabajan juntas para alcanzar algún resultado final.

- Los componentes (capítulo 11) se integran en la arquitectura.
- Se realizan pruebas detalladas (capítulo 11) para asegurar una funcionalidad apropiada.

El modelo de desarrollo basado en componentes conduce a la reutilización del software, la cual proporciona beneficios a los ingenieros de software. Con base en estudios de reutilización, QSM Associates, Inc. informa que el ensamblaje de componentes conduce a una reducción de 70 por ciento del ciclo de desarrollo; 84 por ciento del costo del proyecto y un índice de productividad de 26.2, comparado con una norma de la industria de 16.9 [YOU94]. A pesar de que estos resultados están en función de la robustez de la biblioteca de componentes, no hay duda de que el modelo de desarrollo basado en componentes proporciona ventajas significativas para los ingenieros de software.

3.5.2 El modelo de métodos formales

El modelo de *métodos formales* (capítulo 28) comprende un conjunto de actividades que conducen a la especificación matemática del software de computadora. Los métodos formales permiten que un ingeniero de software especifique, desarrolle y verifique un sistema basado en computadora al aplicar una notación matemática rigurosa. Algunas organizaciones de desarrollo del software aplican en la actualidad una variación de este enfoque, llamado *ingeniería del software de sala limpia* [MIL87, DYE92]. Este modelo se explica en el capítulo 29.

> "Es más fácil escribir un programa incorrecto que entender uno correcto."
>
> **Alan Perlis**

Cuando se utilizan métodos formales durante el desarrollo, éstos proporcionan un mecanismo para eliminar muchos de los problemas difíciles de superar con otros paradigmas de la ingeniería del software. La ambigüedad, el estado incompleto y la inconsistencia se descubren y corrigen con mayor facilidad —no mediante una revisión *ad hoc*, sino a través de la aplicación de un análisis matemático—. Cuando los métodos formales se utilizan durante el diseño sirven como base para la verificación de programas y, por consiguiente, permiten que el ingeniero de software descubra y corrija errores que de otra manera podrían no haberse detectado.

A pesar de que aún no existe un enfoque establecido, los modelos de métodos formales ofrecen la promesa de un software libre de defectos. Sin embargo, se ha mencionado una gran preocupación acerca de su aplicabilidad en su entorno de gestión:

¿Si los métodos formales pueden demostrar la corrección del software, por qué no se utiliza en forma extensa?

- En la actualidad, el desarrollo de modelos formales es muy caro y consume mucho tiempo.
- Se requiere una capacitación detallada, debido a que pocos responsables del desarrollo de software cuentan con los antecedentes necesarios para aplicar métodos formales.

- Es difícil la utilización de estos modelos como un mecanismo de comunicación con clientes que no tienen muchos conocimientos técnicos.

No obstante, tal vez el enfoque a través de métodos formales haya ganado adeptos entre los desarrolladores de software que deben construir sistemas de alta seguridad (por ejemplo, en los campos de la aeronáutica y de los dispositivos médicos), y entre los desarrolladores que padecen severas penurias económicas cuando aparecen errores en el software.

3.5.3 Desarrollo del software orientado a aspectos

Sin importar el proceso de software que se elija, los constructores de software complejo implementan de manera invariable un conjunto específico de características, funciones y contenido de información. Estas características específicas del software se modelan como componentes (por ejemplo, clases orientadas a objetos) y después se construyen dentro del contexto de una arquitectura de sistema. Conforme los sistemas basados en computadora se vuelven más elaborados (y complejos), ciertos "intereses" —propiedades requeridas para el cliente o áreas de interés técnico— abarcan toda la arquitectura. Algunos intereses son propiedades de alto nivel de un sistema (por ejemplo, seguridad, falta de tolerancia). Otros intereses afectan las funciones (como la aplicación de reglas de negocios), mientras que otros son sistémicos (como la sincronización de tareas o la gestión de memoria).

Cuando los intereses se relacionan con múltiples funciones, características e información del sistema, con frecuencia se denominan *intereses generales*. Los *requerimientos de aspectos* definen estos intereses generales que ejercen un impacto a través de la arquitectura del software. El *desarrollo de software orientado a aspectos* (DSOA), referido con frecuencia como *programación orientada a aspectos* (POA), es un paradigma de la ingeniería del software relativamente nuevo que proporciona un proceso y enfoque metodológico para definir, especificar, diseñar y construir *aspectos* —"mecanismos más allá de subrutinas y legados para localizar la expresión de un interés general" [ELR01].

Grundy [GRU02] explica con mayor profundidad los aspectos en el contexto de lo que él llama *ingeniería de componentes orientada a aspectos* [ICOA]:

> La ICOA utiliza un concepto de capas horizontales a través de componentes de software descompuestos en forma vertical, llamados "aspectos", para caracterizar propiedades generales de los componentes, los cuales pueden ser funcionales y no funcionales. Por lo general, los aspectos sistémicos incluyen interfases con el usuario, trabajo en colaboración, distribución, persistencia, gestión de la memoria, procesamiento de transiciones, seguridad, integridad y otros. Los componentes pueden proporcionar o requerir uno o más "detalles de aspecto" relacionados con un aspecto particular, como un mecanismo de visión, acceso extensible y tipo de interfase (aspectos de la interfase con el usuario); generación, transportación y recepción de eventos (aspectos de distribución); almacenamiento/recuperación e indización de datos (aspectos de persistencia); autentificación, codificación y derechos de acceso (aspectos de seguridad); atomicidad de la transacción,

CLAVE

los intereses
los cuales
muchos
del

control de concurrencia y control del transporte (aspectos de transacción), y así sucesivamente. Cada detalle de aspecto tiene una serie de propiedades en relación con características personales y no funcionales del detalle.

Hasta ahora no se ha concretado un proceso orientado a aspectos diferente. Sin embargo, es probable que tal proceso adopte características de los modelos de proceso en espiral y concurrente (secciones 3.4.2 y 3.4.3). La naturaleza evolutiva del modelo en espiral es apropiada cuando se identifican y construyen los aspectos La naturaleza paralela del desarrollo concurrente es esencial porque los aspectos se desarrollan de manera independiente de los componentes del software localizados y, aun así, los aspectos tienen un impacto directo sobre estos componentes. Por lo tanto, resulta esencial implementar una comunicación asincrónica entre las actividades del proceso de software aplicadas al desarrollo y la construcción de aspectos y componentes.

Si se desea conocer una exposición detallada del desarrollo del software orientado a aspectos, se recomienda remitirse a libros dedicados a esta materia. El lector interesado puede consultar [GRA03], [KIS02] o [ELR01].

HERRAMIENTAS DE SOFTWARE

Gestión del proceso

Objetivo: Ayudar en la definición, ejecución y gestión de los modelos prescriptivos del proceso

Mecánica: Las herramientas de gestión del proceso permiten que una organización o equipo de software defina un modelo completo de proceso del software (actividades del marco de trabajo, tareas de aseguramiento de la calidad, puntos de verificación, fundamentos y productos de trabajo). Además, las herramientas proporcionan una guía mientras los ingenieros de software hacen el trabajo técnico y una plantilla para los gerentes, que deben rastrear y controlar el proceso de software.

Herramientas representativas[13]

La GDPA, una serie de herramientas para la definición del proceso de investigación, desarrollada en la Universidad de Bremen en Alemania (www.informatik.uni-bremen.de/uniform/gdpa/home.htm),proporciona un amplio espectro de funciones para el modelado y la gestión del proceso.

SpeeDev, desarrollado por Spee Dev Corporation (www speedev.com), incluye una serie de herramientas para la definición del proceso, gestión de los requisitos, resolución de características, planeación del proyecto y seguimiento del mismo.

Step Gate Process, desarrollado por Objexis (www.objexis.com), incluye muchas herramientas que ayudan en la automatización del flujo de trabajo.

En el sitio de Internet http://205.252.62.38/English/D-ProcessNotation.htm es posible encontrar una valiosa exposición sobre los métodos y la notación que se puede usar para definir y describir un modelo completo del proceso.

13 Las herramientas mencionadas aquí representan un muestreo de esta categoría. En la mayoría de los casos los nombres son marcas registradas de sus respectivos desarrolladores.

3.6 El proceso unificado

En su libro fundamental sobre el proceso unificado, Ivar Jacobson, Grady Booch y James Rumbaugh [JAC99] exponen la necesidad de un proceso de software "guiado por los casos de uso, de arquitectura céntrica, iterativo e incremental". Estos autores establecen:

> En la actualidad la tendencia en el software se encamina a sistemas mayores y complejos. Eso se debe en parte al hecho de que las computadoras se volvían más poderosas cada año, lo que alentaba que los usuarios esperaran más de ellas. Esta tendencia también la impulsó el uso expandido de Internet para el intercambio de todo tipo de información. Nuestro apetito por un software cada vez más complejo crece en la medida en la que aprendemos cómo puede mejorarse un producto desde que sale uno hasta que llega el otro. Necesitamos un software que se adapte mejor a nuestras necesidades, pero que, a su vez, haga el software más complejo. En resumen, queremos más.

De alguna manera, el proceso unificado (PU) es un intento encaminado a reunir los mejores rasgos y características de modelos de procesos de software, pero los caracteriza de manera que implementa muchos de los mejores principios del desarrollo ágil de software (capítulo 4). El proceso unificado reconoce la importancia de la comunicación con el cliente y los métodos encaminados a describir el punto de vista del cliente con respecto a un sistema (por ejemplo, el caso de uso[14]). El PU enfatiza el importante papel de la arquitectura de software, y "ayuda al arquitecto a enfocarse en las metas correctas, como el entendimiento, el ajuste a los cambios futuros y la reutilización" [JAC99]. Sugiere un flujo de proceso iterativo e incremental y que proporciona el sentido evolutivo esencial en el desarrollo del software moderno.

En esta sección se presenta una visión general de los elementos clave del proceso unificado. En la parte 2 de este texto se analizan los métodos que pueblan el proceso y las técnicas y notaciones complementarias del UML,[15] las cuales se requieren al aplicar el proceso unificado en el trabajo real de la ingeniería del software.

3.6.1 Una breve historia

Durante la década de 1980 y al principio de la década siguiente, los métodos y lenguajes de programación orientados a objetos (OO)[16] obtuvieron una amplia difusión entre la comunidad de la ingeniería del software. Durante el mismo periodo se pro-

14 Un caso de uso (capítulos 7 y 8) es un contexto narrativo o plantilla que describe una característica o función del sistema desde el punto de vista del usuario. El caso de uso lo escribe el usuario y sirve como una base para la creación de un modelo de análisis más completo.

15 El UML (*Unified Modeling Language*) se ha convertido en la notación más utilizada para el modelado del análisis y el diseño. Representa una unión entre tres importantes notaciones orientadas a objetos.

16 Si el lector no se encuentra familiarizado con los métodos orientados a objetos, en los capítulos 8 y 9 se presenta una breve revisión general de ellos. Para una presentación más detallada véase [REE02] [STI01] o [FOW99].

puso una amplia variedad de análisis y diseño orientados a objetos (AOO y DOO) y se introdujo un modelo de proceso orientado a objetos de propósito general (similar a los modelos evolutivos presentados en este capítulo). Al igual que la mayoría de los paradigmas "nuevos" para la ingeniería del software, los seguidores de cada uno de los métodos de AOO y DOO argumentaban acerca de cuál de ellos era el mejor, pero ningún método o lenguaje dominó la escena de la ingeniería del software

Al principio de la década de 1990, James Rumbaugh [RUM91], Grady Booch [BOO94] e Ivar Jacobson [JAC92] comenzaron a trabajar en un "método unificado" que combinaría las mejores características de cada uno de sus métodos individuales y adoptaría características adicionales que propusieran otros expertos (por ejemplo, [WIR90]) en el campo OO. El resultado fue el lenguaje de modelado unificado (UML, por sus siglas en inglés) —que contiene una notación robusta para el modelado y el desarrollo de sistemas OO—. Para 1997, el UML se convirtió en un estándar de la industria para el desarrollo de software orientado a objetos. Al mismo tiempo, Rational Corporation y otras firmas desarrollaron herramientas automáticas para apoyar los métodos del UML.

El UML proporciona la tecnología necesaria para apoyar la práctica de la ingeniería del software orientada a objetos, pero no provee el marco de trabajo del proceso que guíe a los equipos en la aplicación de la tecnología. En los años siguientes, Jacobson, Rumbaugh y Booch desarrollaron el *proceso unificado*, un marco de trabajo para la ingeniería del software orientada a objetos, mediante la utilización del UML. En la actualidad, el proceso unificado y el UML se emplean de forma amplia en proyectos OO de todos los tipos. El modelo iterativo e incremental que propone el PU puede y debe adaptarse para satisfacer necesidades de proyecto específicas.

Como consecuencia de la aplicación del UML se puede producir un arreglo de productos de trabajo (por ejemplo, modelos y documentos). Sin embargo, éstos los reducen los ingenieros de software para lograr que el desarrollo sea más ágil y reactivo ante el cambio.

3.6.2 Fases del proceso unificado[17]

Referencia Web

En www.rational.com/products/rup/whitepapers.jsp se pueden encontrar documentos útiles sobre el PU

Se han analizado cinco actividades genéricas del marco de trabajo y se ha explicado que éstas se pueden aplicar para describir cualquier modelo de proceso del software. El proceso unificado no es la excepción. En la figura 3.7 se muestran las "fases" del proceso unificado (PU) y se relacionan con las actividades genéricas que se trataron en el capítulo 2.

La fase de *inicio* del PU abarca la comunicación con el cliente y las actividades de planeación. Al colaborar con los clientes y usuarios finales se identifican los requisitos de negocios para el software, se propone una arquitectura aproximada para el

17 Algunas veces el proceso unificado se llama proceso unificado racional (PUR) después de que lo respaldara la Rational Corporation, un contribuyente importante en el desarrollo y refinamiento del proceso y un constructor de ambientes completos (herramientas y tecnología).

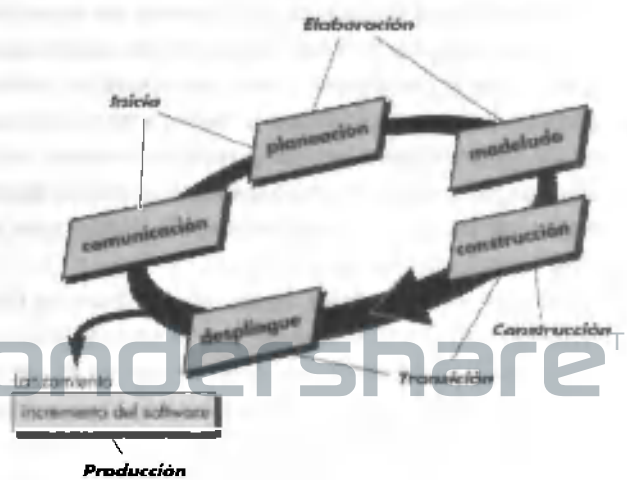
CLAVE

sistema, y se desarrolla un plan para la naturaleza iterativa e incremental del sistema subsiguiente. Los requisitos fundamentales de negocios se describen a través de un conjunto preliminar de casos de uso que describen cuáles características y funciones son deseables para cada clase importante de usuarios. En general, un caso de uso describe una secuencia de acciones que desarrolla un *actor* (por ejemplo, una persona, una máquina, otro sistema) cuando éste interactúa con el software. Los casos de uso ayudan a identificar el ámbito del proyecto y proporcionan una base para la planeación de éste.

En este punto, la arquitectura no es más que un esquema tentativo de los subsistemas más importantes y de las funciones y características que los forman. Después, la arquitectura se refinará y expandirá en un conjunto de modelos que representarán visiones diferentes del sistema. La planeación identifica recursos, evalúa los riesgos importantes, define un itinerario y establece una base para las fases que se aplicarán conforme se desarrolle el incremento del software.

La fase de *elaboración* abarca la comunicación con el cliente y las actividades de modelado del modelo genérico del proceso (figura 3.7). La elaboración refina y expande los casos de uso preliminares que se desarrollaron como una parte de la fase de inicio; además, expande la representación arquitectónica para incluir cinco visiones diferentes del software: el modelo de caso de uso, el modelo de análisis, el modelo de diseño, el modelo de implementación y el modelo de despliegue. En algunos casos, la elaboración crea una "línea de base arquitectónica ejecutable" [ARL02] que representa un sistema ejecutable en su "primer corte".[18] La línea de base arquitectónica demuestra la viabilidad de la arquitectura, pero no proporciona todas las

Figura 3.7 El proceso unificado



18 Es importante destacar que la directriz arquitectónica no es un prototipo que se deseche (sección 3.4.1). En lugar de ello la directriz se aprovecha durante la siguiente fase del PU.

características y funciones necesarias para utilizar el sistema. Además, el plan se revisa de manera cuidadosa al término de la fase de elaboración para asegurar que el ámbito, los riesgos y los datos entregados aún son razonables. Las modificaciones al plan se deben hacer en este momento.

La fase de *construcción* del PU es idéntica a la actividad de construcción definida para el proceso genérico del software. Si se utiliza el modelo arquitectónico como entrada, la fase de construcción desarrolla o adquiere los componentes del software que harán que cada caso de uso sea operativo para los usuarios finales. Lograr esto requiere que los modelos de análisis y diseño iniciados durante la fase de elaboración se completen para reflejar la versión final del incremento del software. Todas las características y funciones necesarias y requeridas del incremento del software (por ejemplo, la entrega) se implementan en código fuente. Conforme los componentes están en proceso de implementación, se diseñan y ejecutan pruebas de unidad para cada uno de ellos. Además, se realizan las actividades de integración (ensamblaje de componentes y pruebas de integración). Los casos de uso se utilizan para derivar un conjunto de pruebas de aceptación que se ejecutan antes del inicio de la siguiente fase del PU.

La fase de *transición* del PU abarca las últimas etapas de la actividad genérica de construcción y la primera parte de la actividad genérica de despliegue. El software se entrega a los usuarios finales para realizar pruebas beta,[19] y la retroalimentación del usuario reporta tanto defectos como cambios necesarios. Además, el equipo de software crea la información de soporte necesaria (por ejemplo, manuales del usuario, guías de resolución de problemas, procedimientos de instalación) para el lanzamiento. Al final de la fase de transición, el incremento de software se convierte en un lanzamiento de software utilizable.

Referencia Web

En www.

La fase de *producción* del PU coincide con la actividad de despliegue del proceso genérico. Durante esta fase se monitorea el uso subsiguiente del software, se proporciona el soporte para el ambiente operativo (infraestructura), y se reciben y evalúan los informes de defectos y los requerimientos de cambios.

Es probable que mientras se realizan las fases de construcción, transición y producción ya se hayan iniciado los trabajos para el siguiente incremento del software. Esto significa que las cinco fases del PU no suceden en una secuencia, sino en una concurrencia por etapas.

A lo largo de las fases del PU se distribuye un flujo de trabajo de ingeniería del software. En el contexto del PU, un *flujo de trabajo* es análogo a un conjunto de tareas (definido en el capítulo 2). Esto es, un flujo de trabajo identifica las tareas necesarias para completar una acción importante de ingeniería del software y los productos de

19 En la *prueba beta*, una acción de prueba controlada (capítulo 13), el software lo utilizan usuarios finales reales, con la intención de descubrir defectos y deficiencias. Se establece un esquema de informe de defectos y deficiencias de manera formal, y el equipo de software evalúa la retroalimentación.

trabajo que se producen como consecuencia de la realización exitosa de tareas. Se debe destacar que no todas las tareas identificadas para un flujo de trabajo del PU se realizan para cualquier proyecto de software. El equipo debe adaptar el proceso (acciones, tareas, subtareas y productos de trabajo) para satisfacer sus necesidades

3.6.3 Productos de trabajo del proceso unificado

En la figura 3.8 se ilustran los *productos de trabajo* clave que se produjeron como consecuencia de las cuatro fases técnicas del PU. Durante la fase de inicio, el propósito es establecer una "visión" general para el proyecto, identificar un conjunto de requisitos de negocios, formar un caso de negocios para el software y definir los riesgos del proyecto y del negocio que pudieran representar un obstáculo para el éxito. Desde el punto de vista del ingeniero de software, el producto de trabajo más importante generado durante la etapa de inicio es el *modelo de caso de uso:* una colección de casos de uso que describen la forma en que actores externos ("usuarios" humanos y no humanos del software) interactúan con el sistema y obtienen valor a partir de éste. En esencia, el modelo de casos de uso es una colección de escenarios de uso con plantillas estandarizadas que implican características y funciones del software mediante la descripción de una serie de condiciones previas, un flujo de eventos o un escenario, y un conjunto de condiciones exteriores para la interacción representada. En un inicio, los casos de uso describen requisitos al nivel del dominio de negocios (por ejemplo, el grado de abstracción es alto). Sin embargo, el modelo de casos de uso se refina y elabora conforme cada fase del PU se ejecuta y sirve como una entrada importante para la creación de productos de trabajo subsecuentes. Durante la fase de inicio sólo se completa entre el 10 y 20 por ciento de los casos de uso. Después de la elaboración, se ha creado entre un 80 y 90 por ciento del modelo.

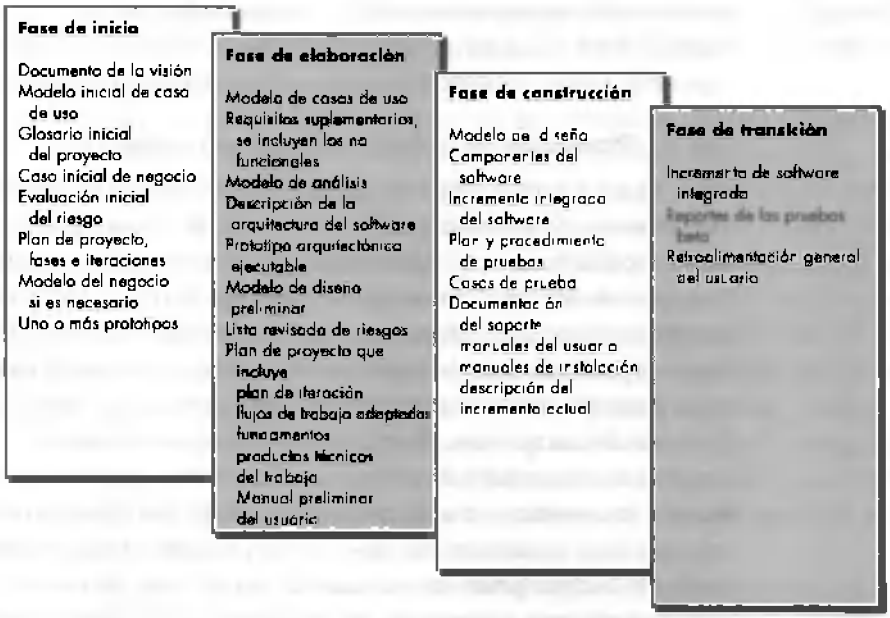
La fase de elaboración produce un conjunto de productos de trabajo que elabora requisitos (incluso requisitos no funcionales[20]), así como una descripción arquitectónica y un diseño preliminar. Cuando el ingeniero de software inicia con el análisis orientado a objetos, el objetivo primordial es definir un conjunto de clases de análisis que describan en forma adecuada el comportamiento del sistema. El *modelo de análisis* del PU es un producto de trabajo que se desarrolla como consecuencia de esta actividad. Los paquetes de clases y análisis (colecciones de clases) definidos como una parte del modelo de análisis se refinan después en un *modelo de diseño*, el cual identifica clases de diseño, subsistemas y las interfases entre los subsistemas. Los modelos de análisis y diseño expanden y refinan una representación evolutiva de la arquitectura del software. Además, en la fase de elaboración se revisan los riesgos y el plan de proyecto para asegurar que cada uno de ellos conserve su validez.

La fase de construcción produce un modelo de *implementación* que traduce las clases de diseño en componentes de software que se construirán para ejecutar el sis-

20 Requisitos que no se pueden deducir del modelo de caso de uso

FIGURA 3.8

Principales productos de trabajo producidos para cada fase del PU.



tema, y un modelo de *despliegue* convierte los componentes en el ambiente físico de computación. Por último, un modelo de *prueba* describe las pruebas empleadas para asegurar que los casos de uso se reflejen de manera apropiada en el software que se ha construido.

La fase de transición entrega el incremento del software y evalúa los productos de trabajo elaborados durante la etapa en que los usuarios finales trabajan con el software. En esta etapa se produce la retroalimentación proveniente de las pruebas beta y los requerimientos cualitativos de cambio

3.7 RESUMEN

Los modelos prescriptivos del proceso de software se han aplicado durante muchos años en un esfuerzo encaminado a ordenar y estructurar el desarrollo del software. Cada uno de estos modelos convencionales sugiere un flujo de proceso que de alguna forma es diferente, pero todos realizan el mismo conjunto de actividades genéricas del marco de trabajo: comunicación, planeación, modelado, construcción y despliegue.

El modelo en cascada sugiere una progresión lineal de actividades del marco de trabajo que a menudo resulta inconsistente con la realidad moderna en el mundo del software (por ejemplo, con el cambio continuo, los sistemas en evolución, las fechas de entrega restringidas). Sin embargo, este modelo se puede aplicar en situaciones en las cuales los requisitos están bien definidos y son estables.

Los modelos incrementales del proceso de software producen software como una serie de entregas de incrementos. El modelo DRA está diseñado para proyectos grandes que se deben entregar en marcos de tiempo muy reducidos.

Los modelos de proceso evolutivos reconocen la naturaleza evolutiva de la mayoría de los proyectos de ingeniería del software y están diseñados para ajustarse al cambio. Los modelos evolutivos, como el de construcción de prototipos y el modelo en espiral, generan productos de trabajo incrementales (o versiones del software en funcionamiento) con rapidez. Estos modelos se pueden adaptar para aplicarlos a través de todas las actividades de la ingeniería del software: desde el desarrollo de conceptos hasta el mantenimiento del sistema a largo plazo.

El modelo basado en componentes destaca la reutilización y ensambladura de componentes. Los modelos de métodos formales conducen a la utilización de un enfoque basado en las matemáticas para el desarrollo y la verificación del software. El modelo orientado a aspectos incluye los intereses generales que cubren la arquitectura total del sistema.

El proceso unificado es un proceso de software "guiado por los casos de uso, de arquitectura céntrica, iterativo e incremental" diseñado como un marco para los métodos y herramientas del UML. El proceso unificado es un modelo incremental en el que se definen cinco fases: 1) una fase de *inicio* que abarca la comunicación con el cliente y las actividades de planeación, y destaca el desarrollo y el refinamiento de casos de uso como un modelo primario; 2) una fase de *elaboración* que abarca la comunicación con el cliente y las actividades de modelado con un enfoque en la creación de modelos de análisis y diseño, con énfasis en las definiciones de clase y representaciones arquitectónicas; 3) una fase de *construcción* que refina y después traduce el modelo de diseño en componentes de software implementados; 4) una fase de *transición* que transfiere el software del desarrollador al usuario final para realizar las pruebas beta y obtener la aceptación; y 5) una fase de *producción* en la cual se realiza el monitoreo continuo y el soporte.

REFERENCIAS

[AMB02] Ambler, S. y L. Constantine, *The Unified Process Inception Phase*, CMP Books, 2002.

[ARL02] Arlow, J. e I. Neustandt, *UML and the Unified Process*, Addison-Wesley, 2002.

[BAC97] Bach, J., "Good Enough Quality: Beyond the Buzzword", en *IEEE Computer*, vol.30, núm. 8, agosto de 1997, pp. 96-98.

[BOE88] Boehm, B., "A Spiral Model for Software Development Enhancement", en *Computer*, vol. 21, núm. 5, mayo de 1988, pp. 61-72.

[BOE98] Boehm, B., "Using the WINWIN Spiral Model: A Case Study", en *Computer*, vol 3, núm 7, julio de 1998, pp 33-34.

[BOE01] Bohem, B., "The Spiral Model as a Tool for Evolutionary Software Acquisition", en *CrossTalk*, mayo de 2001, disponible en http://www.stsc.hill.af.mil/crosstalk/2001/05/boehm.html.

[BOO94] Booch, G., *Object-Oriented Analysis and Design*, 2a. ed., Benjamin Cummings, 1994.

[BRA94] Bradac, M., D. Perry y L. Votta, "Prototyping a Process Monitoring Experiment", en *IEEE Trans. Software Engineering*, vol. 20, núm. 10, octubre de 1994, pp. 774-784.

[BRO75] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.

[BUT94] Butler, J., "Rapid Application Development in Action", en *Managing System Development*, Applied Computer Research, vol. 14, núm.5, mayo de 1994, pp. 6-8.
[DYE 92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992
[ELR01] Elrad, T., R Filman y A Bader (eds.), "Aspect-Oriented Programming", en *Comm. ACM*, vol. 44, núm. 10, octubre de 2001, edición especial
[FOW99] Fowler, M y K. Scott, *UML Distilled*, 2a. ed., Addison-Wesley, 1999
[GIL88] Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, 1998
[GRA03] Gradecki, J y N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley, 2003.
[GRU02] Grundy, J., "Aspect-Oriented Component Engineering", 2002, http://www.cs. auckland. ac.nz/~john-g/aspects.html.
[HAN95] Hanna, M., "Farewell to Waterfalls", en *Software Magazine*, mayo de 1995, pp. 38-46.
[HES96] Hesse, W., "Theory and Practice of the Software Process—A Field Study and its Implications for Project Management", *Software Process Technology, 5th European Workshop*, EWSPT 96, Springer LNCS 1149, 1996, pp. 241-256
[HES01] Hesse, W., "Dinosaur Meets Archaeopteryx? Seven Theses on Rational's Unified Process (RUP)", en *Proc. 8th Intl Workshop on Evaluation of Modeling Methods in System Analysis and Design*, Ch. VII Interlaken, 2001
[JAC92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992
[JAC99] Jacobson, I., Booch, G. y J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
[KAU95] Kauffman, S., *At Home in the Universe*, Oxford, 1995
[KER94] Kerr, J. y R. Hunter, *Inside RAD*, McGraw-Hill, 1991.
[MAR91] Martin, J., *Rapid Application Development*, Prentice-Hall, 1991
[McDE93] McDermid J y P. Rook, "Software Development Process Models", en *Software Engineer's Reference Book*, CRC Press, 1993, pp. 15/26-15/28
[MIL87] Mills, H.D., M Dyer y R. Linger, "Cleanroom Software Engineering", en *IEEE Software*, septiembre de 1987, pp. 19-25
[NIE92] Nierstrasz, O., S. Gibbs y D Tsichritzis, "Component-Oriented Software Development", en *CACM*, vol. 35, núm. 9, septiembre de 1992, pp 160-165.
[NOG00] Nogueira, J., C Jones y Luqi, "Surfing the Edge of Chaos. Applications to Software Engineering", Command and Control Research Technology Symposium, Naval Post Graduate School, Monterey CA, junio de 2000, disponible en http//www dodccrp org/2000CCRTS/cd/html/pdf_papers/Track_4/075 pdf.
[REE02] Reed, P., *Developing Applications with Java and UML*, Addison-Wesley, 2002
[REI95] Reilly, J. P., "Does RAD Live Up th the Hype", en *IEEE Software*, septiembre de 1995, pp. 24-26
[ROO96] Roos, J., "The Poised Organization: Navigating Effectively on Knowledge Landscapes", 1996, disponible en http://www imd.ch/fac/roos/paper_po.html.
[ROY70] Royce, W W., "Managing the Development of Large Software Systems: Concepts and Techniques" *en, Proc WESCON*, agosto de 1970.
[RUM91] Rumbaugh, J *et al.*, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
[STI01] Stiller, E. y C. LeBlanc, *Project-Based Software Engineering. An Object-Oriented Approach*, Addison-Wesley, 2001.
[WIR90] Wirfs-Brock, R., B. Wilkerson y L. Weiner, *Designing Object-Oriented Software*, Prentice-Hall, 1990
[YOU94] Yourdon, E., "Software Reuse", en *Application Development Strategies*, vol. 6, núm 12, diciembre de 1994, pp. 1-16.
[YOU95] Yourdon, E., "When Goood Enough Is Best", en *IEEE Software*, vol 12, núm. 3, mayo de 1995, pp. 79-81.

PROBLEMAS Y PUNTOS A CONSIDERAR

3.1. Leer [NOG00] y escribir un documento de dos o tres páginas que trate sobre el impacto del "caos" en la ingeniería del software.

3.2. Dar tres ejemplos de proyectos de software que pudieran adaptarse al modelo en cascada. Ser específico.

3.3. Proporcionar tres ejemplos de proyectos de software que pudieran adaptarse al modelo de construcción de prototipos. Ser específico.

3.4. ¿Cuáles adaptaciones se requieren en el proceso si el prototipo evolucionará hacia un sistema o producto que puede entregarse?

3.5. Para lograr un desarrollo rápido el modelo DRA asume la existencia de una cosa. ¿Cuál es y por qué dicha suposición no siempre es verdadera?

3.6. Proporcionar tres ejemplos de proyectos de software que pudieran adaptarse al modelo incremental. Ser específico.

3.7. ¿Qué se puede decir acerca del software que está en desarrollo o en mantenimiento mientras se avanza hacia fuera del flujo de proceso en espiral?

3.8. ¿Es posible combinar modelos de proceso? Si la respuesta es afirmativa, menciónese un ejemplo.

3.9. El modelo concurrente del proceso define un conjunto de "estados". Describir con palabras propias lo que representan estos estados, y después indicar cómo entran en juego dentro del modelo concurrente del proceso.

3.10. ¿Cuáles son las ventajas y desventajas de desarrollar software para el cual la calidad es "lo suficientemente buena"? Esto es, ¿qué pasa cuando se resalta la velocidad del desarrollo sobre la calidad del proyecto?

3.11. Proporcionar tres ejemplos de proyectos de software que pudieran adaptarse al modelo basado en componentes. Ser específico.

3.12. Es posible probar que un componente de software o incluso un programa completo está correcto. Entonces, ¿por qué no todos lo hacen?

3.13. Exponer con argumentos propios el significado de "intereses generales". La iterativa sobre el ADP se expande con rapidez. Investigar y escribir un documento breve sobre la situación actual.

3.14. ¿El proceso unificado y el UML son la misma cosa? Explicar la respuesta.

3.15. ¿Cuál es la diferencia entre una fase del PU y un flujo de trabajo del PU?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La mayor parte de los textos sobre ingeniería del software consideran los modelos prescriptivos de proceso con algún detalle. Los libros de Sommerville (*Software Engineering*, sexta edición, Addison-Wesley, 2000), Pfleeger (*Software Engineering: Theory and Practice*, Prentice-Hall, 2001), y Schach (*Object Oriented and Classical Software Engineering*, McGraw-Hill, 2001) consideran los paradigmas convencionales y analizan sus fortalezas y debilidades. A pesar de que no se dedica en forma específica al proceso, Brooks (*The Mythical Man-Month*, segunda edición, Addison-Wesley, 1995) presenta la experiencia ganada en proyectos antiguos que tienen una gran relación con el proceso. Firesmith y Henderson-Sellers (*The OPEN Process Framework: An Introduction*, Addison-Wesley, 2001), presenta una plantilla general para crear "procesos de software flexible, pero, aún así, indisciplinados" y analiza los atributos y objetivos del proceso.

Sharpe y McDermott (Workflow Modeling: Tools for Process Improvement and Application Development, Artech House, 2001) presentan herramientas para el modelado de procesos de software y negocios. Jacobson Griss y Jonsson (*Software Reuse*, Addison-Wesley, 1997) y McClure (*Software Reuse Techniques*, Prentice-Hall, 1997) presentan mucha información útil

sobre el desarrollo basado en componentes. Heineman y Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) describen el proceso requerido para implementar sistemas basados en componentes. Kenett y Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) consideran la forma en que la gestión de calidad y el diseño de proceso están conectados en forma íntima entre sí.

Ambriola (*Software Process Technology*, Springer-Verlag, 2001), Derniame y sus colegas (*Software Process: Principles, Methodology, and Technology*, Springer-Verlag, 1999) presentan conferencias editadas que incluyen muchos aspectos teóricos y de investigación y que son relevantes para el proceso de software

Jacobson, Booch y Rumbaugh han escrito el libro fundamental sobre el proceso unificado [JAC99]. Sin embargo, los libros de Arlow y Neustadt [ARL02] y una serie de tres volúmenes de Ambler y Constantine [AMB02] ofrecen una excelente información complementaria. Krutchen (*The Rational Unified Process*, segunda edición, Addison-Wesley, 2000) ha escrito una valiosa introducción al PU. La gestión de un proyecto dentro del contexto del PU está escrita en detalle por Royce (*Software Project Management: A Unified Framework*, Addison-Wesley, 1998). La descripción definitiva del PU la ha desarrollado la Rational Corporation y está disponible en línea en www rational com

En Internet existe una amplia variedad de fuentes de información sobre la ingeniería y el proceso del software En el sitio web de SEPA se puede encontrar una lista actualizada de referencias en la red mundial que son relevantes para el proceso de software: **http://www.mhhe. com/pressman.**

CAPÍTULO

4

DESARROLLO ÁGIL

En 2001, Kent Beck y otros 16 notables desarrolladores, escritores y consultores [BEC01] (conocidos como la "Alianza Ágil") firmaron el "Manifiesto para el desarrollo ágil de software", el cual establecía:

> Hemos descubierto mejores formas de desarrollar software al construirlo por nuestra cuenta y ayudar a otros a hacerlo. Por medio de este trabajo hemos llegado a valorar:
>
> A los *individuos y sus interacciones* sobre los procesos y las herramientas
>
> Al *software en funcionamiento* sobre la documentación extensa.
>
> A la *colaboración del cliente* sobre la negociación del contrato.
>
> A la *respuesta al cambio* sobre el seguimiento de un plan.
>
> Esto es, aunque los términos a la derecha tienen valor, nosotros valoramos más los aspectos de la izquierda.

Un manifiesto se asocia por lo general con un movimiento político emergente, aquel que ataca a la vieja vanguardia y sugiere un cambio revolucionario (en el mejor de los casos para mejorar). De alguna forma, esto es con exactitud de lo que se trata el desarrollo ágil.

A pesar de que las ideas subyacentes que conducen al desarrollo ágil han estado presentes por muchos años, no ha sido sino hasta la década pasada que estas ideas han cristalizado en un "movimiento". En esencia, los métodos ágiles[1] se desarrollaron en un intento por superar las debilidades advertidas y reales en la ingeniería del software convencional. El desarrollo ágil proporciona beneficios importantes, pero es imposible aplicarlo en todos los proyectos, productos, personas y situaciones.

UN VISTAZO RÁPIDO

¿Qué es? La ingeniería de software ágil combina una filosofía y un conjunto de directrices de desarrollo. La filosofía busca la satisfacción del cliente y la entrega temprana de software incremental; equipos de proyecto pequeños y con alta motivación; métodos informales; un mínimo de productos de trabajo de la ingeniería del software; y una simplicidad general del desarrollo. Las directrices de desarrollo resaltan la entrega sobre el análisis y el diseño (aunque estas actividades no se descartan), y la comunicación activa y continua entre los desarrolladores y los clientes.

¿Quién lo hace? Los ingenieros de software y otros participantes del proyecto (gerentes, clientes y usuarios finales) trabajan juntos en un equipo ágil: un equipo con organización propia y que controla su propio destino. Un equipo ágil fomenta la comunicación y la colaboración entre todos los que trabajan en él.

¿Por qué es importante? El ambiente moderno de los negocios ocasiona que los sistemas basados en computadoras y los productos de

1 A los métodos ágiles algunas veces se les llama métodos *ligeros* o *livianos*.

software estén acelerados y en cambio continuo. La ingeniería del software ágil representa una opción razonable a la ingeniería convencional para ciertas clases de software y ciertos tipos de proyectos de software. Ha demostrado su utilidad al entregar sistemas exitosos con rapidez.

¿Cuáles son los pasos? El desarrollo ágil podría llamarse con mayor precisión "ingeniería del software ligera". Las actividades básicas del marco de trabajo –comunicación con el cliente, planeación, modelado, construcción, entrega y evolución– se conservan, pero éstas se conforman como un conjunto mínimo de tareas que empuja al equipo de proyecto hacia la construcción y la entrega (habrá quienes argumenten que esto se hace a costa del análisis del problema y del diseño de la solución).

¿Cuál es el producto obtenido? Los clientes e ingenieros de software que han adoptado la filosofía ágil tienen la misma visión: el único producto de trabajo realmente importante es un "incremento de software" en funcionamiento, el cual se entrega al cliente en una fecha prometida.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Si el equipo de software está de acuerdo en que el proceso funciona y dicho equipo produce incrementos de software entregables que satisfacen al cliente, entonces el trabajo está bien hecho.

No es la antítesis de la práctica sólida de la ingeniería del software y es posible aplicarla como una filosofía predominante para cualquier trabajo de software.

En la economía moderna, a menudo resulta difícil o imposible predecir cómo evolucionarán con el tiempo los sistemas basados en computadoras (por ejemplo, las aplicaciones Web). Las condiciones del mercado cambian con rapidez, las necesidades de los usuarios finales evolucionan, y las nuevas amenazas competitivas emergen sin previo aviso. En muchas situaciones ya es imposible definir por completo los requisitos antes de que comience el proyecto. Los ingenieros de software deben ser tan ágiles como para responder a un ambiente de negocios fluido.

¿Lo anterior significa que el reconocimiento de estas realidades modernas ocasiona que se descarten los valiosos principios, conceptos, métodos y herramientas de la ingeniería del software? No, ¡en lo absoluto¡ Como todas las disciplinas de ingeniería, la ingeniería del software continúa en evolución. Se puede adaptar con facilidad para enfrentar los retos que implica una exigencia de agilidad.

> "Agilidad: 1, todo lo demás: 0."
>
> **Tom DeMarco**

En un libro que invita a la reflexión y trata sobre el desarrollo ágil de software, Alistair Cockburn [COC02a] argumenta que los modelos prescriptivos de proceso presentados en el capítulo 3 tienen una falla importante: *olvidan las fragilidades de las personas que construyen el software de computadora*. Los ingenieros de software no son robots. Ellos muestran una gran variedad en los estilos de trabajo y diferencias significativas en su grado de habilidad, creatividad, orden, consistencia y espontaneidad. Algunos se comunican muy bien en forma escrita, otros no. Cockburn argumenta que los modelos de proceso pueden "enfrentar las debilidades comunes de la gente con disciplina o tolerancia [alguna de las dos]" [COC02a], y que los modelos

de proceso más prescriptivos eligen la disciplina Él establece: "Como la consistencia en la acción es una debilidad humana, las metodologías que exigen un alto grado de disciplina son frágiles" [COC02a].

Los modelos de proceso funcionarán si proporcionan un mecanismo realista que fomente la disciplina necesaria, o deben estar caracterizados de manera que muestren "tolerancia" por la gente que realiza el trabajo de la ingeniería del software De manera invariable, la gente de software adopta y sostiene más fácilmente las prácticas tolerantes, pero (como Cockburn lo admite) tal vez sea menos productiva Como la mayoría de las cosas en la vida, se deben considerar los intercambios

4.1 ¿Qué es la AGILIDAD?

¿Qué es la agilidad en el contexto del trabajo de la ingeniería del software? Ivar Jacobson [JAC02] proporciona una definición útil:

> *Agilidad* se ha convertido actualmente en la palabra de moda en cuanto se describe un moderno proceso de software Cualquiera es ágil. Un equipo ágil es un equipo rápido que responde de manera apropiada a los cambios. Éstos son, en gran parte, la materia del desarrollo de software. Cambios en el software que se va a construir, cambios entre los miembros del equipo, cambios debidos a las nuevas tecnologías, cambios de todo tipo que pueden incidir en el producto que se construye o en el proyecto que crea el producto. En cualquier actividad de software se debe incluir un soporte para los cambios, esto es algo que adoptamos porque es el alma y el corazón del software. Un equipo ágil reconoce que el software lo desarrollan individuos que trabajan en equipo y que las aptitudes de esta gente, y su capacidad para colaborar, son esenciales para el éxito del proyecto.

De acuerdo con la visión de Jacobson, la insistencia en el cambio es el conductor primordial hacia la agilidad Los ingenieros de software deben tener pies veloces si quieren ajustarse a los cambios rápidos que describe Jacobson.

> "La agilidad es dinámico, con contenido específico, ajustable al cambio de manera dinámica y orientada al crecimiento "
>
> **Steven Goldman *et al.***

Pero la agilidad es más que una respuesta efectiva al cambio. También incluye la filosofía del manifiesto enunciado al principio de este capítulo. Estimula las estructuras y actitudes de los equipos para que la comunicación (entre los miembros del equipo, entre los técnicos y la gente de negocios, entre los ingenieros de software y sus gerentes) sea más fácil. Resalta la entrega rápida del software operativo y le resta importancia a los productos de trabajo intermedio (lo cual no siempre es bueno); adopta al cliente como una parte del equipo de desarrollo y trabaja para eliminar la actitud del tipo "nosotros y ustedes" que aún perjudica a muchos proyectos de software; reconoce que la planeación tiene sus límites en un mundo incierto y que el plan de proyecto debe ser flexible.

CONSEJO

No cometer el error de suponer que el proceso proporciona soluciones, un sustituto para el pensamiento y la disciplina

La Alianza Ágil [AGI03] define 12 principios para quienes quieren alcanzar la agilidad.

1. Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software valioso.
2. Bienvenidos los requisitos cambiantes, incluso en fases tardías del desarrollo La estructura de los procesos ágiles cambia para la ventaja competitiva del cliente.
3. Entregar con frecuencia software en funcionamiento, desde un par de semanas hasta un par de meses, con una preferencia por la escala de tiempo más corta.
4. La gente de negocios y los desarrolladores deben trabajar juntos a diario a lo largo del proyecto.
5. Construir proyectos alrededor de individuos motivados. Darles el ambiente y el soporte que necesitan, y confiar en ellos para obtener el trabajo realizado.
6. El método más eficiente y efectivo de transmitir información hacia y dentro de un equipo de desarrollo es la conversación cara a cara.
7. El software en funcionamiento es la medida primaria de progreso
8. Los procesos ágiles promueven el desarrollo sustentable. Los patrocinadores, desarrolladores y usuarios deben ser capaces de mantener un paso constante de manera indefinida
9. La atención continua a la excelencia técnica y al buen diseño mejora la agilidad
10. La simplicidad —el arte de maximizar la cantidad de trabajo no realizado— es esencial.
11. Las mejores arquitecturas, los mejores requisitos y los mejores diseños emergen de equipos autoorganizados.
12. A intervalos regulares el equipo refleja la forma en que se puede volver más efectivo; entonces su comportamiento se ajusta y adecua en concordancia

La agilidad se puede aplicar en cualquier proceso de software. Sin embargo, para lograrlo es esencial que el proceso sea diseñado en una forma que permita al equipo del proyecto adaptar y coordinar las tareas, conducir la planeación en una forma que entienda la fluidez de un enfoque de desarrollo ágil, eliminar todo pero no los productos de trabajo esenciales y mantenerlos controlados, y enfatizar una estrategia de entrega incremental que proporcione software en funcionamiento al cliente tan rápido como sea factible para el tipo de producto y el ambiente operativo.

4.2 ¿QUÉ ES UN PROCESO ÁGIL?

Cualquier *proceso ágil de software* se caracteriza de una manera que refiere tres suposiciones clave [FOW02] acerca de la mayoría de los proyectos de software:

1. Resulta difícil predecir cuáles requisitos del software persistirán y cuáles cambiarán. De igual forma, es difícil presagiar cómo cambiarán las prioridades del cliente mientras se ejecuta un proyecto
2. Para muchos tipos de software, el diseño y la construcción están intercalados Esto es, ambas actividades se deben realizar de manera conjunta, de modo que los modelos de diseño sean probados conforme se crean. Resulta difícil predecir cuánto diseño se necesita antes de que la construcción se utilice para probar el diseño.
3. El análisis, el diseño y la construcción no son predecibles (desde el punto de vista de la planeación), lo que sería deseable.

Dados los puntos anteriores, surge una pregunta importante: ¿cómo se crea un proceso susceptible de manipular en forma impredecible? La respuesta, como ya se ha puntualizado antes, reside en la adaptabilidad del proceso (a un proyecto y a condiciones técnicas que cambian con rapidez). Por lo tanto, un proceso ágil debe ser *adaptable*.

Pero una adaptación continua sin un progreso logra muy poco. Por lo tanto, un proceso ágil de software debe adaptarse en forma *incremental*. Para llevar a cabo una adaptación incremental, un equipo ágil requiere de la retroalimentación con el cliente (para que sea factible realizar las adaptaciones apropiadas). Un catalizador efectivo para la retroalimentación del cliente es un prototipo operacional o una porción de un sistema operacional. Por lo tanto, debe instituirse una *estrategia incremental de desarrollo*. Los *incrementos de software* (prototipos ejecutables o una porción de un sistema operacional) deben entregarse en cortos periodos para que la adaptación mantenga un buen ritmo con el cambio (imprevisibilidad) Este enfoque iterativo le permite al cliente evaluar el incremento del software de manera regular, proporcionar la retroalimentación necesaria al equipo de software, e influir sobre las adaptaciones del proceso que se realizan para adecuar la retroalimentación.

> "No existe un sustituto para la retroalimentación rápida, ni en el proceso de desarrollo ni en el producto mismo."
> **Martin Fowler**

4.2.1 Las políticas del desarrollo ágil

Existe un debate considerable (a veces estridente) sobre los beneficios y la aplicabilidad del desarrollo ágil del software como alternativa a procesos de ingeniería del software más convencionales. Jim Highsmith [HIG02a] (a manera de broma) analiza los extremos cuando caracteriza el sentimiento del campo proagilidad ("agilitado-

res"). "Los metodólogos tradicionales son un conjunto de tipos que se arrastran en el lodo y que prefieren producir documentación que no fluye, en vez de un sistema de trabajo que cubra las necesidades del negocio." Como contraparte, establece la posición del campo de la ingeniería del software (de nuevo, en forma de broma): "Los metodólogos 'ligeros' y, eh, 'ágiles' son un conjunto de intrusos informáticos que van a estar ahí para dar una maldita sorpresa cuando intenten elevar sus juguetes al nivel de software de la empresa".

Al igual que todos los argumentos sobre la tecnología del software, este debate sobre la metodología corre el riesgo de degenerar en una guerra religiosa. Si estalla la guerra, desaparece el pensamiento racional, y las creencias, en vez de los hechos, guían la toma de decisiones.

Nadie está en contra de la agilidad. La pregunta real es: ¿cuál es la mejor manera de lograrla? Igual de importante es la pregunta: ¿cómo se construye un software que satisfaga hoy las necesidades de los clientes y muestre las características de calidad que le permitan extenderse y escalar para cubrir a largo plazo las necesidades del cliente?

CONSEJO

No es necesario elegir entre agilidad e ingeniería del software. En lugar de ello, se puede definir un enfoque de ingeniería de software que sea ágil

No existen respuestas absolutas para ninguna de estas preguntas. Aun dentro de la escuela ágil se han propuesto varios modelos de proceso (sección 4.3), cada uno con un enfoque sutilmente diferente para el problema de la agilidad. Dentro de cada modelo hay un conjunto de "ideas" (que los agilitadores suelen llamar "tareas de trabajo") que representan una separación significativa de la ingeniería del software convencional. Y aun así, muchos conceptos de agilidad son tan sólo adaptaciones de buenos conceptos de la ingeniería del software. Como punto final, hay mucho que ganar si se considera lo mejor de ambas escuelas, y nada que ganar si se denigra alguno de los dos enfoques.

El lector interesado puede consultar [HIG01], [HIG02a] y [DEM02] para un animado resumen de los aspectos técnicos y políticos importantes.

4.2.2 Factores humanos

Los defensores del desarrollo ágil del software resaltan la importancia de los "factores de las personas" en un desarrollo ágil exitoso. Como establecen Cockburn y Highsmith [COC01]: "El desarrollo ágil se centra en los talentos y las habilidades de los individuos, puesto que el proceso se ajusta a personas y equipos específicos". El punto clave en esta afirmación es que el proceso *se ajusta a las necesidades de las personas y del equipo*, y no al revés.[2]

> "Aquello apenas suficiente para un equipo es excesivo o insuficiente para otro."
>
> **Alistair Cockburn**

2 La mayoría de las organizaciones de software exitosas reconocen esta realidad sin importar el modelo de proceso que elijan.

Si los miembros del equipo de software van a manejar las características del proceso que se aplica para construirlo, debe existir un gran número de rasgos clave entre la gente de un equipo ágil y el equipo mismo:

Competencia. En el contexto de un desarrollo ágil (al igual que en la ingeniería del software convencional), la "competencia" abarca un talento innato, habilidades específicas relacionadas con el software, y un conocimiento general del proceso que el equipo haya elegido aplicar. La habilidad y el conocimiento del proceso pueden y deben enseñarse a toda la gente que funge como miembro de un equipo ágil.

Enfoque común. Aunque los miembros del equipo ágil desempeñen tareas diferentes y aporten distintas habilidades al proyecto, todos deben enfocarse en una meta: entregar al cliente un incremento de trabajo de software dentro del tiempo establecido. Alcanzar esta meta requiere que el equipo también se centre en adaptaciones continuas (pequeñas y grandes) mediante las cuales el proceso satisfará las necesidades del equipo.

Colaboración. La ingeniería del software (sin considerar el proceso) incluye evaluar, analizar y usar información que se comunica al equipo de software; crear información que ayudará al cliente y a otros a entender el trabajo del equipo; y construir información (software de computadora y bases de datos relevantes) que ofrezca un valor comercial para el cliente. Estas tareas se cumplirán si los miembros del equipo colaboran, entre ellos, con el cliente y con sus gerentes.

Habilidad para la toma de decisiones. Todo buen equipo de software (incluidos los equipos ágiles) debe permitirse la libertad de controlar su propio destino. Esto implica que al equipo se le dé autonomía, es decir, autoridad para tomar decisiones en cuanto a cuestiones técnicas y del proyecto.

Capacidad de resolución de problemas confusos. Los gestores de software deben reconocer que el equipo ágil enfrentará ambigüedades y sufrirá golpes de manera continua debido al cambio. En algunos casos, el equipo debe aceptar que el problema que está resolviendo hoy tal vez no sea el problema que debe resolverse mañana. Sin embargo, las lecciones aprendidas en cualquier actividad para la resolución de problemas (incluidas aquellas que sirven para solucionar el problema erróneo) pueden beneficiar al equipo en fases posteriores del proyecto.

Confianza y respeto mutuo. El equipo ágil se debe convertir en lo que De Marco y Lister [DEM98] llaman un equipo "cuajado" (véase el capítulo 21). Un equipo cuajado muestra la confianza y el respeto necesarios para que "se unan con tanta fuerza, que el todo sea mayor que la suma de las partes" [DEM98].

Organización propia. En el contexto del desarrollo ágil, la *organización propia* implica tres factores: 1) el equipo ágil se organiza a sí mismo para el trabajo que debe hacerse; 2) el equipo organiza el proceso que mejor se ajusta a su ambiente local; 3) el equipo organiza el programa de trabajo con el que se alcance de mejor

manera la entrega del incremento del software. La organización propia tiene varios beneficios técnicos, pero lo más importante es que mejora la colaboración y eleva la moral del equipo. En esencia, el equipo sirve como su propia gestoría. Ken Schwaber [SCH02] puntualiza estos aspectos cuando escribe: "El equipo selecciona la cantidad de trabajo que cree que es capaz de hacer dentro de la iteración, y el equipo se compromete con el trabajo. Nada desalienta más a un equipo que alguien más se comprometa por él. Nada motiva más a un equipo que aceptar la responsabilidad de cumplir los compromisos que él mismo hizo".

4.3 Modelos ágiles de proceso

La historia de la ingeniería del software está llena de decenas de descripciones y metodologías, métodos de modelado y notaciones, herramientas y tecnologías obsoletas. Cada elemento surgió con notoriedad y después lo eclipsó algo nuevo y (supuestamente) mejor. Con la introducción de un amplio espectro de modelos ágiles de proceso —cada uno en busca de su aceptación dentro de la comunidad del desarrollo de software— el movimiento ágil está en la misma ruta histórica [3]

> "Nuestra profesión va tras las metodologías como un adolescente va tras la ropa."
>
> **Stephen Hawrysh y Jim Ruprecht**

En las siguientes secciones se presenta una visión general de cierto número de diferentes *modelos ágiles de proceso*. Existen muchas similitudes (en filosofía y práctica) entre estos enfoques. La intención es subrayar aquellas características de cada método que lo hacen único. Es importante señalar que *todos* los modelos ágiles se ajustan (en mayor o menor grado) al *Manifiesto para el desarrollo de software* y a los principios enunciados en la sección 4.1

4.3.1 Programación extrema (PE)

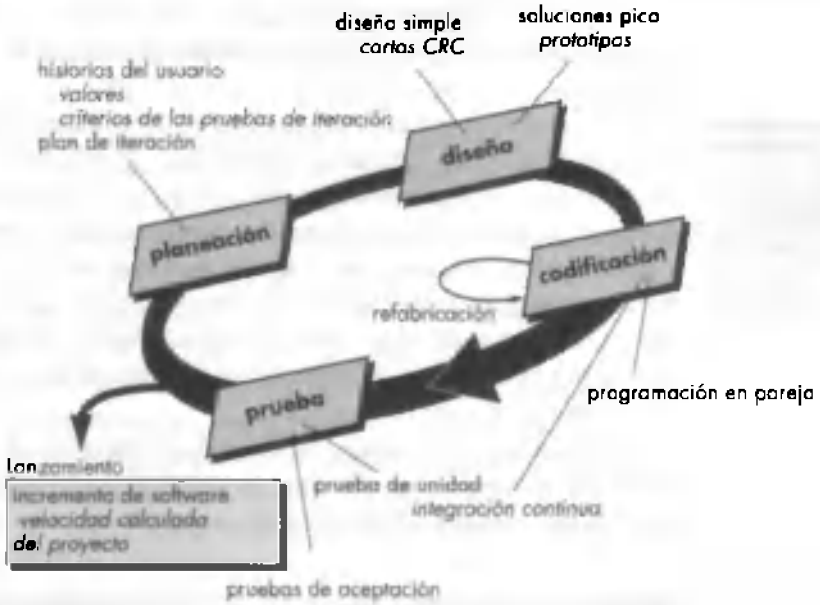
A pesar de que los primeros trabajos sobre las ideas y los métodos asociados con la *programación extrema (PE)* se realizaron a finales de la década de 1980, el trabajo fundamental sobre la materia, escrito por Kent Beck [BEC99], se publicó en 1999. Los libros subsiguientes de Jeffries *et al.* [JEF01] sobre los detalles técnicos de la PE, y el trabajo adicional de Beck y Fowler [BEC01b] sobre la planeación de la PE expusieron los detalles del método.

Referencia Web

En el sitio www.extremepro... encontrar una excelente revisión de los "reglas" para la PE.

La PE utiliza un enfoque orientado a objetos (parte 2 de este libro) como su paradigma de desarrollo preferido. La PE abarca un conjunto de reglas y prácticas que ocurren en el contexto de cuatro actividades del marco de trabajo: planeación, dise-

3 Esto no es algo malo. Antes de que uno o más modelos o métodos sean aceptados como un estándar de facto, todos deben competir por los corazones y las mentes de los ingenieros de software. Los "ganadores" evolucionan con la mejoría que proporciona la práctica, mientras que los "perdedores" desaparecen o se unen a los modelos "ganadores".



ño, codificación y pruebas. En la figura 4.1 se ilustra el proceso de la PE y se observan algunas de las ideas y tareas clave asociadas con cada actividad del marco de trabajo. En los siguientes párrafos se resumen las actividades clave de la PE.

Planeación. La actividad de planeación comienza creando una serie de *historias* (también llamadas *historias del usuario*) que describen las características y la funcionalidad *requeridas para el software que se construirá. Cada historia (similar a los casos de uso descritos en los capítulos 7 y 8) la escribe el cliente y se coloca en una* carta índice. El cliente le asigna un *valor* (es decir, una prioridad) a la historia basándose en los valores generales del negocio respecto de la característica o la función.[4] Los miembros del equipo de la PE evalúan entonces cada historia y le asignan un *costo*, el cual se mide en semanas de desarrollo. Si la historia requiere más de tres semanas de desarrollo, se le pide al cliente que la divida en historias menores, y se realiza de nuevo la asignación del valor y el costo. Es importante destacar que las historias nuevas pueden escribirse en cualquier momento.

Los clientes y el equipo de PE trabajan juntos para decidir cómo agrupar las historias hacia el próximo lanzamiento (el siguiente incremento de software) para que el equipo de la PE las desarrolle. Una vez establecido el *compromiso* básico (el acuerdo de las historias que se incluirán, la fecha de entrega y otras cuestiones del proyecto) para un lanzamiento, el equipo de la PE ordena las historias que se desarrollarán de una de las siguientes tres maneras: 1) todas las historias serán imple-

4 El valor de una historia puede depender también de la presencia de otra historia.

mentadas de un modo inmediato (dentro de pocas semanas); 2) las historias con valor más alto se moverán en el programa y se implementarán al principio; o 3) las historias más riesgosas se moverán dentro del programa y se implementarán al principio.

Referencia Web
En el sitio **c2.com/cgi/wiki?planningGame** se puede encontrar un valioso "juego de planeación" para la PE.

Después de que se ha entregado el primer lanzamiento del proyecto (también llamado incremento de software), el equipo de la PE calcula la velocidad del proyecto. Dicho de un modo más simple, *la velocidad del proyecto* es el número de historias de los clientes implementado durante el primer lanzamiento. Entonces, la velocidad del proyecto puede utilizarse para 1) ayudar a estimar fechas de entrega y el programa para lanzamientos subsecuentes, y 2) determinar si se ha hecho un compromiso excesivo en algunas de las historias de todo el proyecto de desarrollo. Si se presenta un compromiso excesivo, el contenido de los lanzamientos se modifica o se cambian las fechas de las entregas finales.

Conforme avanza el trabajo de desarrollo, el cliente puede agregar historias, cambiar el valor de la historia existente, dividir historias o eliminarlas. Entonces el equipo de la PE considera de nuevo los lanzamientos restantes y modifica sus planes de acuerdo con ello.

> "La programación extrema es una disciplina de desarrollo de software que se basa en valores de simplicidad, comunicación, retroalimentación y audacia."
>
> **Ron Jeffries**

Diseño. El diseño de la PE sigue de manera rigurosa el principio MS (mantenerlo simple). Siempre se prefiere un diseño simple respecto de una presentación más compleja. Además, el diseño ofrece una guía de implementación para una historia como está escrita, ni más ni menos. Se desaprueba el diseño de funcionalidad extra (porque el desarrollador supone que se requerirá más tarde).

La PE apoya el uso de tarjetas CRC (capítulo 8) como un mecanismo efectivo para pensar en el software en un contexto orientado a objetos. Las tarjetas CRC (colaborador-responsabilidad-clase) identifican y organizan las clases orientadas al objeto[6] que son relevantes para el incremento del software actual. El equipo PE conduce el ejercicio del diseño por medio de un proceso similar al descrito en el capítulo 8 (sección 8.7.4.). Las tarjetas CRC son el único producto de trabajo realizado como parte del proceso de la PE.

Si se encuentra un problema difícil de diseño como parte del diseño de la historia, la PE recomienda la creación inmediata de un prototipo operacional de esa porción del diseño. El prototipo del diseño, llamado la *solución pico*, se implementa y evalúa. El propósito es reducir el riesgo cuando comience la verdadera implementa-

5 Estas directrices de diseño se deberían seguir en todos los métodos de ingeniería del software, aunque a veces las notaciones y terminologías sofisticadas que se utilizan en el diseño pueden emplearse de una manera más simple.

6 En el capítulo 8, y a lo largo de la parte 2 del libro, se estudian las clases orientadas a objetos.

ción y validar las estimaciones originales en la historia que contiene el problema del diseño

La PE apoya la *refabricación*, una técnica de construcción que también lo es de diseño Fowler [FOW00] describe la refabricación de la siguiente manera:

> Refabricación es el proceso de cambiar un sistema de software de tal manera que no altere el comportamiento externo del código y que mejore la estructura interna. Es una manera disciplinada de limpiar el código [y modificar/simplificar el diseño interno], lo que minimiza las oportunidades de introducir errores En esencia, al refabricar, se mejora el diseño del código después de que se ha escrito.

Debido a que el diseño de la PE virtualmente no utiliza la notación y produce, si acaso, muy pocos productos de trabajo, distintos a las tarjetas de CRC y soluciones pico, el diseño se considera como un artefacto que puede y debe modificarse de manera continua a medida que prosigue la construcción. El propósito de refabricar es controlar estas modificaciones al sugerir pequeños cambios del diseño que "pueden mejorar de manera radical el diseño" [FOW00]. Sin embargo, debe notarse que el esfuerzo requerido para refabricar puede aumentar en forma drástica a medida que crece el tamaño de la aplicación.

Una noción central en la PE es que el diseño ocurre tanto antes como *después* del comienzo de la codificación. Refabricar significa que el diseño ocurre de manera continua a medida que se construye el sistema. De hecho, la actividad de construcción misma le proporcionará al equipo de PE una guía sobre cómo mejorar el diseño.

Codificación. La PE recomienda que después de diseñar las historias y realizar el trabajo de diseño preeliminar el equipo no debe moverse hacia la codificación, sino que debe desarrollar una serie de pruebas de unidad que ejerciten cada una de las historias que vayan a incluirse en el lanzamiento actual (incremento de software).[7] Una vez creada la prueba de unidad, el desarrollador es más capaz de centrarse en lo que debe implementarse para pasar la prueba de unidad. No se agrega nada extraño (MS) Una vez que el código está completo, la unidad puede probarse de inmediato, y así proporcionar una retroalimentación instantánea a los desarrolladores.

Un concepto clave durante la actividad de codificación (y uno de los aspectos de la PE de los que más se ha hablado) es la *programación en pareja*. La PE recomienda que dos personas trabajen juntas en una estación de trabajo de computadora para crear el código de una historia Esto proporciona un mecanismo para la resolución de problemas en tiempo real (dos cabezas piensan mejor que una) y el aseguramiento de la calidad en las mismas condiciones. También alienta que los desarrolladores se mantengan centrados en el problema que se tiene a la mano. En la práctica, cada persona tiene un papel sutilmente diferente Por ejemplo, una persona puede pen-

7 Este enfoque es análogo a conocer las preguntas del examen antes de comenzar a estudiar Esto facilita mucho más el estudio al enfocar la atención sólo sobre las preguntas que serán formuladas

sar en los detalles de codificación de una porción particular del diseño, mientras que la otra se asegura de que se sigan los estándares de codificación (una parte requerida de la PE) y que el código que se genera "coincida" con el diseño más amplio de la historia.

Cuando los programadores completan su trabajo el código que desarrollaron se integra con el trabajo de otros. En algunos casos esto lo lleva a cabo diariamente el equipo de integración. En otros casos, la pareja de programadores es la responsable de la integración. Esta estrategia de "integración continua" ayuda a evitar problemas de compatibilidad e interfaz y proporciona un ambiente de "prueba de humo" (capítulo 13) que ayuda a descubrir los errores desde el principio.

Pruebas. Ya se ha hecho notar que la creación de una prueba de unidad[8] antes de comenzar la codificación es un elemento clave para el enfoque de la PE. Las pruebas de unidad que se crean deben implementarse con un marco de trabajo que permita automatizarlas (por lo tanto, pueden ejecutarse de manera fácil y repetida). Esto apoya una estrategia de regresión de prueba (capítulo 13) cuando el código se modifica (al cual a menudo se le confiere la filosofía de la PE de refabricar).

Cuando las unidades individuales de prueba se organizan en un "conjunto universal de pruebas" [WEL99], las pruebas de integración y validación del sistema pueden realizarse a diario. Esto proporciona al equipo de PE una indicación continua del progreso y también puede encender luces de emergencia previas si las cosas salen mal. Wells [WEL99] establece: "Arreglar problemas pequeños cada pocas horas toma menos tiempo que arreglar problemas enormes justo antes de la fecha límite".

PUNTO CLAVE

Las pruebas de aceptación de la PE se derivan de las historias del usuario.

Las *pruebas de aceptación* de la PE, también llamadas *pruebas del cliente*, las especifica el cliente y se enfocan en las características generales y la funcionalidad del sistema, elementos visibles y revisables por el cliente. Las pruebas de aceptación se derivan de las historias del usuario que se han implementado como parte de un lanzamiento de software.

HogarSeguro

Tomar en cuenta el desarrollo de software ágil

El escenario: Oficina de Doug Miller

Los actores: Doug Miller, gerente de ingeniería de software; Jamie Lazar, miembro del equipo de software, Vinod Raman, miembro del equipo de software.

La conversación:

(llaman a la puerta)

Jamie: Doug, ¿tienes un minuto?

Doug: Seguro Jamie, ¿qué pasa?

Jamie: Hemos estado pensando en nuestra conversación de ayer acerca del proceso — tú sabes, de cuál sería el proceso que elegiríamos para este nuevo proyecto de *HogarSeguro*

Doug: ¿Y?

8 Las pruebas de unidad, que se tratan con detalle en el capítulo 13, se enfocan sobre un componente individual del software, al ejercitar la interfaz de los componentes, las estructuras de datos y la funcionalidad en un esfuerzo por descubrir los errores locales en el componente

hablando con un amigo de otro
me comentó sobre la programación
modelo de proceso ágil, ¿has oído algo

buenas y malas.

a nosotros nos suena muy bien. Te
el software muy rápido, utiliza algo
ación en pareja para hacer
de calidad en tiempo real... pienso que es

uchas ideas realmente buenas. Por
el concepto de la programación en
de que todos los participantes del
arte del equipo de desarrollo

¿Eso significa que mercadotecnia
el equipo con nosotros?

): Ellos son participantes, ¿no?

Dios! . Van a estar pidiendo cambios

ecesariamente. Mi amigo me dijo que hay
r" los cambios durante el proyecto de

¿ustedes creen que debamos usar PE?

Es algo que definitivamente deberíamos de

Doug: Estoy de acuerdo. Y aun si elegimos como nuestro enfoque un modelo incremental, no existe ninguna razón por la que no podamos incorporar mucho de lo que la PE tiene que ofrecer.

Vinod: Doug, antes dijiste "escuché cosas buenas y malas". ¿Cuáles fueron las "malas"?

Doug: Lo que no me gusta es la forma en que la PE menosprecia el análisis y el diseño... es como decir que la acción real sólo está en la escritura del código

(los miembros del equipo se miran entre sí y sonríen)

Doug: Entonces, ¿están de acuerdo con el enfoque de la PE?

Jamie (hablando por ambos): ¡Jefe, lo que nosotros hacemos es escribir código!

Doug (riendo): Es cierto, pero me gustaría verlos dedicar un poco menos de tiempo a la codificación y a la recodificación y un poco más de tiempo a analizar lo que se tiene que hacer y a diseñar una solución que funcione.

Vinod: Puede ser que adoptemos ambas formas, agilidad con un poco de disciplina.

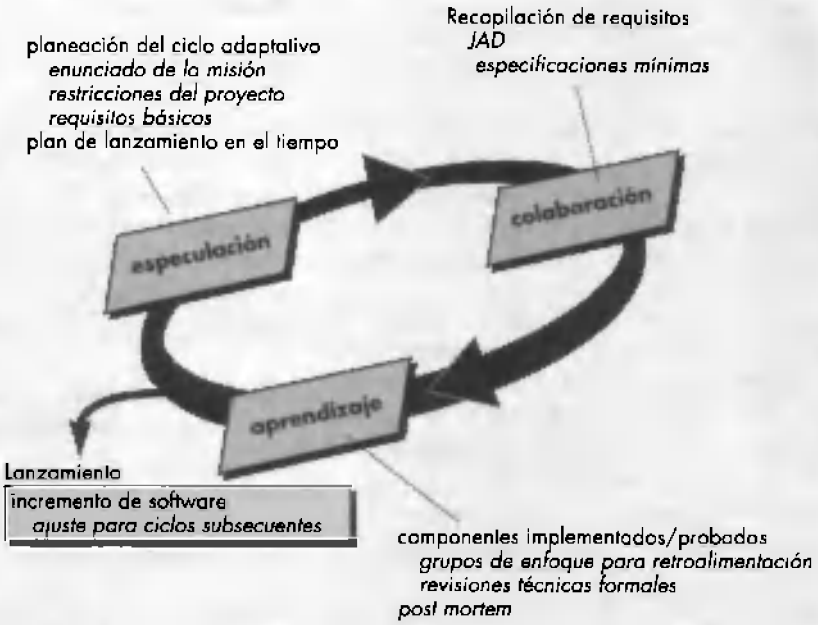
Doug: Vinod, creo que podemos hacerlo. De hecho, estoy seguro de ello.

4.3.2 Desarrollo adaptativo de software (DAS)

El *desarrollo adaptativo de software* (DAS) lo propuso Jim Highsmith [HIG00] como una técnica para construir software y sistemas complejos. Los apoyos filosóficos del DAS se enfocan en la colaboración humana y la organización propia del equipo. Highsmith [HIG98] expone lo anterior cuando escribe:

> La organización propia es una propiedad de los sistemas adaptativos complejos, similar a un "ajá" colectivo; es en el momento de energía creativa cuando surge la solución a algún problema persistente. La organización propia emerge cuando los individuos, los agentes independientes (células en un cuerpo, especies en un ecosistema, desarrolladores en un equipo de software) cooperan [colaboran] para crear salidas emergentes. Una salida emergente es una propiedad más allá de la capacidad de cualquier agente individual. Por ejemplo, las neuronas individuales del cerebro no poseen conciencia, pero en forma colectiva generan la propiedad de la conciencia. Tendemos a ver este fenómeno del surgimiento colectivo como un accidente, o al menos como independiente y sin reglas. El estudio de la organización propia demuestra que dicha visión es errónea.

FIGURA 4.2
Desarrollo adaptativo de software.



Referencia Web
En el sitio **www.adaptivesd.com** se pueden encontrar ligas para el DAS.

¿Cuáles son las características de los ciclos adaptativos del DAS?

Highsmith argumenta que un enfoque de desarrollo ágil y adaptativo basado en la colaboración es "tanto como una fuente de *orden* en las complejas interacciones entre disciplina e ingeniería" Él define un "ciclo de vida" del DAS (figura 4.2), el cual incorpora tres fases: especulación, colaboración y aprendizaje.

Especulación. En esta fase se inicia el proyecto y se conduce el *ciclo adaptativo de planeación*. Este último utiliza información de inicio del proyecto —el enunciado de la misión del cliente, restricciones de proyecto (por ejemplo, fechas de entrega o descripciones del cliente) y los requisitos básicos— para definir el conjunto de ciclos de lanzamiento (incrementos del software) que se requerirán para el proyecto.[9]

Colaboración. La gente motivada trabaja junta de una forma que multiplica su talento y sus salidas creativas más allá de sus números absolutos. Este enfoque de colaboración es un tema recurrente en todos los métodos ágiles, pero la cooperación no es fácil. No es sólo comunicación, aunque la comunicación es parte de ella. No es sólo un asunto de trabajo en equipo, aunque un equipo "cuajado" (capítulo 21) es esencial para la presencia de la colaboración real. No es un rechazo al individualismo, ya que la creatividad individual representa un papel importante en el pensamiento de colaboración. Esto es, por encima de todo, una cuestión de confianza. Las personas que trabajan juntas deben confiar entre sí para 1) criticar sin animosidad;

CONSEJO
La colaboración efectiva con el cliente ocurrirá sólo si se eliminan todas las actitudes del tipo "yo y ustedes".

9 Obsérvese que el plan del ciclo adaptativo puede adaptarse, y con probabilidad lo hará, al proyecto cambiante y a las condiciones del negocio

2) ayudar sin resentimientos; 3) trabajar tan duro o más duro de lo que ya lo hacen; 4) tener el conjunto de aptitudes para contribuir al trabajo en curso; y 5) comunicar los problemas o preocupaciones en una forma que conduzca a la acción efectiva.

> "Me gusta escuchar. He aprendido mucho al escuchar a las personas. La mayoría de la gente nunca escucha."
>
> **Ernest Hemingway**

Aprendizaje. Como miembros de un equipo de DAS se comienzan a desarrollar los componentes integrantes de un ciclo adaptativo, la importancia radica en el aprendizaje y en el progreso a través de un ciclo completo. De hecho, Highsmith [HIGG00] argumenta que los desarrolladores de software a menudo sobreestiman su comprensión (de la tecnología, el proceso y el proyecto), y que el aprendizaje les podría ayudar a mejorar su grado de entendimiento real. Los equipos del DAS aprenden de tres maneras:

1. **Grupos enfocados.** El cliente o los usuarios finales proporcionan retroalimentación sobre los incrementos de software que se entregan. Esto indica en forma directa la satisfacción o la insatisfacción de las necesidades del negocio
2. **Revisiones técnicas formales.** Los miembros del equipo del DAS revisan los componentes del software desarrollado mientras mejoran su calidad y su aprendizaje.
3. ***Post mortem*.** El equipo de DAS se vuelve introspectivo al vigilar su propio desempeño y proceso (con el propósito de aprender acerca de su enfoque y después mejorarlo).

Es importante destacar que la filosofía del DAS es meritoria sin importar el modelo de proceso empleado. El acento general en la dinámica de la organización propia en los equipos, la colaboración interpersonal y el aprendizaje individual y por equipo conducen grupos de proyectos de software con una mayor posibilidad de éxito.

4.3.3 Método de desarrollo de sistemas dinámicos (MDSD)

El *método de desarrollo de sistemas dinámicos* [STA97] es un enfoque de desarrollo de software ágil que "proporciona un marco de trabajo para construir y mantener sistemas con restricciones de tiempo muy estrechas mediante el empleo de la construcción de prototipos incrementales en un ambiente de proyecto controlado" [CCS02]. Similar a algunos aspectos del proceso DRA descrito en el capítulo 3, el MDSD sugiere una filosofía tomada de una modificación del principio de Pareto. En este caso, 80 por ciento de la aplicación se puede entregar en 20% del tiempo que tomaría entregar 100 por ciento de la aplicación (sistema completo).

Al igual que la PE y el DSA, el MDSD sugiere un proceso iterativo de software. Sin embargo, el enfoque del MDSD en cada iteración sigue la regla del 80 por ciento. Esto es, sólo se necesita el trabajo suficiente para cada incremento y para facilitar el

movimiento hacia el nuevo incremento. Los detalles restantes se pueden completar posteriormente cuando se conozcan más los requisitos de negocios o cuando los cambios hayan sido solicitados o ajustados.

En la red mundial hay una organización (DSDM Consortium, www.dsdm.org) que de manera colectiva asume el papel de "conservador" del método. Esta organización ha definido un modelo ágil de proceso, llamado el ciclo de vida del MDSD. Este método define tres ciclos iterativos diferentes, a los cuales preceden dos actividades del ciclo de vida adicionales:

Referencia Web
En **www.cs3inc.com/DSDM.htm** se puede encontrar una revisión útil del MDSD

Estudio de factibilidad: establece los requisitos básicos de negocio y las restricciones asociadas con la aplicación del método y para evaluar si la aplicación es una candidata viable para el proceso del MDSD.

Estudio de negocios: establece los requisitos funcionales y de información que permitirán que la aplicación proporcione un valor al negocio; también define la arquitectura básica de la aplicación.

Iteración de modelo funcional: produce una serie de prototipos incrementales que demuestran la funcionalidad para el cliente (nota: todos los prototipos del MDSD están diseñados para evolucionar hacia la aplicación entregable). El propósito durante este ciclo iterativo es recopilar requisitos adicionales mediante la retroalimentación de lo que obtiene el usuario, mientras éste trabaja con el prototipo.

Iteración de construcción y diseño: revisa la construcción de prototipos durante la iteración del modelo funcional para asegurar que cada uno de ellos ha sido desarrollado de una manera que le permitirá proporcionar un valor operativo de negocios para los usuarios finales. En algunos casos, la iteración del modelo funcional y el diseño y la iteración de construcción suceden en forma concurrente.

Implementación: coloca el incremento de software más reciente (un prototipo "operacionalizado") en el ambiente operativo. Se debe destacar que 1) el incremento puede no estar 100 por ciento completo o 2) se pueden requerir cambios cuando el incremento se coloca en el sitio. En cualquier caso, el trabajo de desarrollo del MDSD continúa al regresar a la actividad de iteración del modelo de función.

El MDSD se puede combinar con la PE para obtener un enfoque conjunto que define un modelo sólido de proceso (el ciclo de vida del MDSD) con los aspectos prácticos (PE) necesarios para construir incrementos de software. Además, los conceptos del DAS de colaboración y equipos autoorganizados se pueden adaptar a un modelo de proceso combinado.

4.3.4 Melé

Melé (término derivado de una jugada de rugby[10]) es un modelo ágil de proceso que desarrollaron Jefff Sutherland y su equipo a principios de la década de 1990. En años

10 Un grupo de jugadores se alinea alrededor del balón y los compañeros de equipo trabajan juntos (algunas veces de manera violenta) para desplazar el balón hacia el lado contrario del campo de juego

recientes, Schwaber y Beedle [SCH01] han presentado el desarrollo posterior de los métodos de melé. Los principios de la melé [ADM96] son consistentes con el manifiesto ágil.

- Los equipos de trabajo pequeños están organizados para "maximizar la comunicación, minimizar los gastos generales y maximizar el hecho de compartir conocimiento tácito e informal".
- El proceso debe adaptarse a los cambios técnicos y de negocios "para asegurar que se produzca el mejor producto posible"
- El proceso produce incrementos frecuentes de software "los cuales se pueden inspeccionar, ajustar, probar, documentar y construir".
- El trabajo de desarrollo y la gente que lo realiza están divididos en "particiones o paquetes de bajo acoplamiento".
- Conforme se construye el producto se realizan pruebas y documentación constantes.
- Los procesos de melé proporcionan la "capacidad de declarar un producto como 'realizado' siempre que esto se requiera (porque la competencia acaba de hacer un lanzamiento, porque la compañía necesita el dinero, porque el usuario/cliente necesita las funciones, porque ya se está en el momento en que se prometió .." [ADM96].

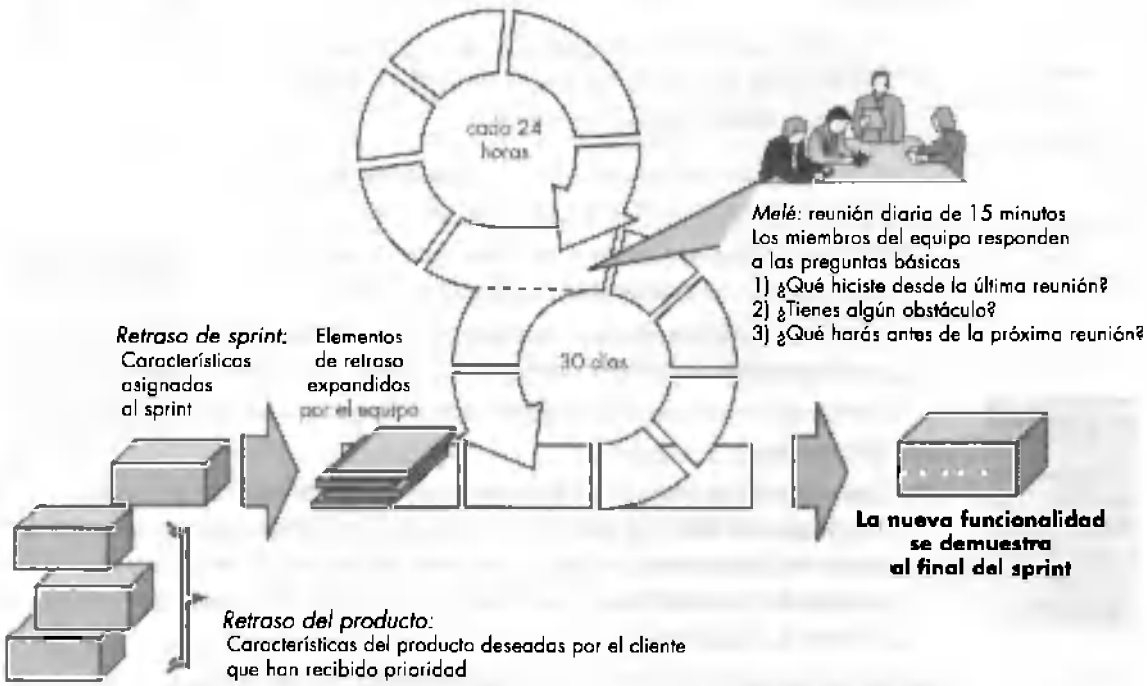
Con los principios de la melé se guían las actividades dentro de un proceso que incorpora las siguientes actividades del marco de trabajo: requisitos, análisis, diseño, evolución y entrega. En cada actividad del marco de trabajo las tareas de trabajo suceden dentro del patrón de proceso (tratado en el párrafo siguiente) llamado *sprint*. El trabajo realizado dentro de un sprint (el número requerido de sprints variará de acuerdo con el tamaño y la complejidad del producto) se adapta al problema y con frecuencia lo modifica en tiempo real el equipo de melé. En la figura 4.3 se ilustra el flujo general del proceso de melé

> *"Las melés nos permiten construir un software más suave."*
>
> **Mike Beedle et al.**

La melé subraya el uso de un conjunto de "patrones de proceso de software" [NOY02] que ha probado su efectividad en proyectos con tiempos de entrega muy reducidos, requisitos cambiantes y condiciones críticas en los negocios. Cada uno de estos patrones de proceso define un conjunto de actividades de desarrollo:

Retrasos: son una lista que considera la prioridad de los requisitos o características de proyecto que proporcionan un valor comercial para el cliente. En cualquier momento se pueden agregar elementos a los retrasos (así se introducen los cambios). El gerente de producto evalúa los retrasos y actualiza las prioridades de acuerdo con lo requerido.

FIGURA 4.3 Flujo de proceso de la melé.



PUNTO CLAVE

La melé incorpora un conjunto de patrones de proceso que resalta las prioridades del proyecto, la división del trabajo, las unidades de trabajo, la comunicación y la retroalimentación frecuente del cliente.

Sprint: consiste en unidades de trabajo que se requieren para satisfacer un requisito definido en los retrasos en un periodo predefinido (el lapso usual es de 30 días). En esta etapa, los elementos de los retrasos a los que se dirigen las unidades de trabajo del sprint están congelados (es decir, durante el sprint no se introducen cambios). Por lo tanto, el sprint permite a los miembros del equipo trabajar en un ambiente enfocado al corto plazo, pero estable

Reuniones de melé: son reuniones cortas (por lo general de 15 minutos) y las realiza a diario el equipo de melé Existen tres preguntas que plantean y responden todos los miembros del equipo

- ¿Qué hiciste desde la última reunión?
- ¿Cuáles obstáculos encontraste?
- ¿Qué esperas lograr para la siguiente reunión del equipo?

Un líder de equipo, llamado "maestro de la melé", preside la reunión y evalúa las respuestas de cada persona. Cada reunión de melé ayuda al equipo a descubrir problemas potenciales tan pronto como sea posible. Estas reuniones diarias también conducen a la "socialización del conocimiento" [BEE99] y, por ende, a promover una estructura de equipo con organización propia.

Demostración: se entrega el incremento de software al cliente de forma que éste demuestre y evalúe la funcionalidad implementada. Es importante señalar que la demostración quizá no contenga toda la funcionalidad planeada, sino aquellas funciones susceptibles de entregarse dentro del periodo establecido.

Beedle y sus colegas [BEE99] presentan un análisis completo de estos patrones y establecen: "La MELÉ supone la existencia del caos...". El patrón de proceso de la melé permite que un equipo de desarrollo de software trabaje de manera exitosa en un mundo donde la eliminación de la incertidumbre es imposible.

4.3.5 Cristal

Alistair Cockburn [COC02a] y Jim Highsmith [HIG02b] crearon la *familia cristal de los métodos ágiles*[11] con el fin de lograr un enfoque de desarrollo de software que coloca un premio en la "manejabilidad" durante lo que Cockburn caracteriza como "un juego cooperativo de inventiva y comunicación con recursos limitados, con una meta primaria consistente en la entrega de software útil y en funcionamiento y una meta secundaria de prepararse para el juego siguiente" [COC02b].

Para alcanzar la manejabilidad, Cockburn y Highsmith definieron un conjunto de metodologías, cada una de ellas con elementos esenciales comunes a todas, y funciones, patrones de proceso, productos de trabajo y prácticas únicas en cada una de ellas. En realidad, la familia cristal es un conjunto de procesos ágiles, los cuales han probado su efectividad en diferentes tipos de proyectos. El objetivo es permitir que los equipos ágiles seleccionen el miembro de la familia cristal más apropiado para su proyecto y ambiente.

4.3.6 Desarrollo conducido por características (DCC)

El *desarrollo conducido por características* (DCC) lo concibieron originalmente Peter Coad y sus colegas [COA99] como un modelo de proceso práctico para la ingeniería del software orientada a objetos. Stephen Palmer y John Felsing [PAL02] han extendido y mejorado el trabajo de Coad, al describir un proceso adaptativo y ágil que puede aplicarse en proyectos de software de tamaño moderado y grande.

En el contexto del DCC una *característica* "es una función valuada por el cliente que puede implementarse en dos semanas o menos" [COA99]. La importancia que se le concede a la definición de características proporciona los siguientes beneficios.

- Como las características son bloques pequeños de funcionalidad entregable, los usuarios las describen con mayor facilidad, pueden entender cómo éstas se relacionan con otras con mayor rapidez, y pueden revisarlas de mejor manera en búsqueda de ambigüedades, errores u omisiones.
- Las características se pueden organizar en un agrupamiento jerárquico relacionado con el negocio

11 El nombre "cristal" se deriva de las características de los cristales geológicos, cada uno con su propio color, forma y dureza

- Como una característica es el incremento de software entregable, el equipo desarrolla características operativas cada dos semanas.
- Debido a que las características son pequeñas, sus diseños y representaciones de código son más fáciles de inspeccionar de manera efectiva.
- La planeación del proyecto, la elaboración de su programa y su rastreo los guía la jerarquía de la característica, en lugar de hacerlo un conjunto de tareas de ingeniería del software adaptado en forma arbitraria.

Coad y sus colegas [COA99] sugieren la siguiente plantilla para definir una característica:

<acción> el **<resultado>** **<porlparaldela>** un **<objeto>**

donde un **<objeto>** es "una persona, sitio o cosa (incluyendo papeles, momentos en el tiempo o intervalos de tiempo, o descripciones del tipo de catálogo de entrada)". Los ejemplos de las características para una aplicación de comercio electrónico podrían ser:

Agregar el producto a un carrito de supermercado.
Desplegar las especificaciones técnicas de un producto.
Almacenar la información de navegación para un cliente.

Un conjunto de características agrupa características relacionadas en categorías relacionadas con el negocio y se define como [COA99]:

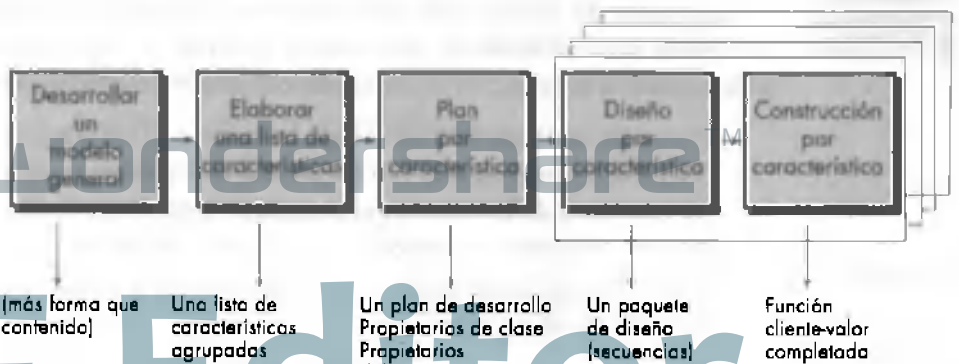
<acción><-ar, -er, -ir> un **<objeto>**

Por ejemplo: *hacer la venta del producto* es un conjunto de características que podría abarcar las características relacionadas con anterioridad y otras.

El enfoque del DCC define cinco actividades de "colaboración" dentro del marco de trabajo (en el DCC éstas se llaman "procesos") como se muestra en la figura 4.4.

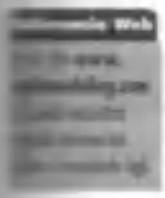
FIGURA 4.4

Desarrollo conducido por características [COA99] (usado con autorización)



El DCC concede una mayor importancia a las directrices y técnicas de la gestión del proyecto que muchos otros métodos ágiles. Cuando los proyectos crecen en tamaño y complejidad, con frecuencia la gestión *ad hoc* del proyecto es inadecuada Resulta esencial para los desarrolladores, sus gerentes y el cliente entender el estatus del proyecto. cuáles logros se han tenido y cuáles programas se han encontrado Si la presión de la fecha límite es significativa, resulta crítico determinar si los incrementos de software (características) están programados de manera apropiada Para lograrlo el DCC define seis puntos de fijación durante el diseño y la implementación de una característica: "ensayo del diseño, diseño, inspección del diseño, código, inspección del código, promoción de la construcción" [COA99].

4.3.7 Modelado ágil (MA)



En muchas situaciones los ingenieros de software deben construir sistemas grandes y críticos para los negocios. El ámbito y la complejidad de dichos sistemas se deben modelar de forma que 1) todas las circunscripciones entiendan mejor lo que se debe lograr; 2) el problema se divida de manera efectiva entre la gente que lo debe resolver; y 3) la calidad se evalúe en cada paso conforme el sistema se desarrolla y construye.

En los últimos 30 años se ha propuesto una amplia variedad de métodos y notación para el modelado de ingeniería del software en el análisis y diseño (tanto arquitectónico como al nivel de componentes). Estos métodos tienen un mérito significativo, pero se ha comprobado que su aplicación enfrenta dificultades y es desafiante poderlos sostener (sobre muchos proyectos). Parte del problema es el "peso" de estos métodos de modelado. Con esto se hace referencia al volumen de notación requerida, el grado de formalismo sugerido, el tamaño de los modelos para proyectos grandes, y la dificultad para el mantenimiento del modelo conforme ocurren los cambios. Aun así, el modelado del análisis y el diseño tiene un beneficio sustancial para los proyectos grandes: por ninguna otra razón que hacer que estos proyectos sean manejables en el sentido intelectual. ¿Existe un enfoque ágil para el modelado de la ingeniería del software que pudiera proporcionar una alternativa?

En el "Sitio oficial del modelado ágil", Scott Ambler [AMB02] describe el *modelado ágil* (MA) de la siguiente manera:

> El modelado ágil (MA) es una tecnología basada en la práctica para el modelado efectivo de los sistemas basados en software Dicho de una forma más simple, el modelado ágil es una colección de valores, principios y prácticas para el modelado de software que puede aplicarse en un proyecto de desarrollo de software de una manera efectiva y ligera Los modelos ágiles son más efectivos que los tradicionales porque son sólo lo suficientemente buenos, no tienen que ser perfectos [AMB02].

Además de los valores consistentes con el manifiesto ágil, Ambler sugiere *valor* y *humildad*. Un equipo ágil debe tener el valor para tomar decisiones que ocasionarán el rechazo y la refabricación de un diseño. Debe tener la humildad de reconocer que quienes manejan la tecnología no tienen todas las respuestas, y que los expertos en negocios y otros participantes de la empresa son dignos de respeto y consideración.

A pesar de que el MA sugiere un arreglo amplio de principios de modelado "esenciales" y "suplementarios", los responsables de que el MA sea único son los siguientes [AMB02]:

Modelar con un propósito. Un desarrollador que use el MA debe tener una meta específica en mente (por ejemplo, comunicar información al cliente o ayudarle a entender mejor algún aspecto del software) antes de crear el modelo. Una vez identificada la meta para el modelo, el tipo de notación que se usará y el grado de detalle requerido serán más obvios.

Usar múltiples modelos. Existen muchos modelos y notaciones diferentes con los cuales describir el software. Sólo un pequeño subconjunto es esencial para la mayoría de los proyectos. El MA sugiere que para proporcionar la visión necesaria cada modelo debe presentar un aspecto diferente del sistema, y sólo aquellos modelos que proporcionen un valor para la audiencia a la que están dirigidos deben usarse.

"Viajar ligero" es un enfoque apropiado para todo el trabajo de ingeniería del software. Construir sólo aquellos modelos que proporcionan valor ¡ni más, ni menos

Viajar ligero. La realización de trabajo de la ingeniería del software requiere conservar sólo los modelos que proporcionarán valor a largo plazo y descartar el resto. Cada producto de trabajo que se conserve debe recibir mantenimiento conforme se presentan cambios. Esto representa un trabajo que reduce la velocidad del equipo. Ambler [AMB02] observa que "cada vez que se decide conservar un modelo se intercambia la agilidad por la conveniencia de tener la información disponible para el equipo de una forma abstracta (por ende, existe una posibilidad de mejorar la comunicación dentro del equipo, así como con los propietarios del proyecto)".

El contenido es más importante que la representación. El modelado debe comunicar información a la audiencia a la que está dirigido. Un modelo sintácticamente perfecto que comunique sólo un poco del contenido útil no tiene tanto valor como un modelo con una notación defectuosa que, sin embargo, comunique un contenido valioso para su audiencia.

Conocer los modelos y las herramientas con que se crean. Es necesario entender las fortalezas y debilidades de cada modelo y las herramientas con los que se creó.

Adaptar en forma local. El enfoque del modelado se debe adaptar a las necesidades del equipo ágil.

HERRAMIENTAS DE SOFTWARE

Desarrollo ágil

Objetivo: El objetivo de las herramientas del desarrollo ágil es ayudar en uno o más aspectos del desarrollo ágil con énfasis en la facilitación de la generación rápida de software operativo. Estas herramientas también se pueden utilizar cuando se aplican los modelos prescriptivos de proceso (capítulo 3).

Mecánica: La mecánica de las herramientas varía. En general, los conjuntos de herramientas técnicas incluyen apoyo automatizado para la planeación del proyecto, el desarrollo

...de uso y la recopilación de requisitos, el diseño ... generación de código y la realización de pruebas.

...as representativas:[12]

... Como el desarrollo ágil es un tópico actual, la ...oría de los vendedores de herramientas de ...are pretenden vender herramientas que apoyen el ...oque ágil. Las herramientas que se presentan a ...nuación tienen características que las hacen útiles ... ...nera particular para los proyectos ágiles.

...treme, desarrollado por Microtool ... microtool.com), proporciona soporte para la gestión de un proceso ágil en varias actividades técnicas dentro del proceso.

Ideogramic UML, desarrollado por Ideogramic (www.ideogramic.com) es un conjunto de herramientas para el UML creado en forma específica para usarlo dentro de un proceso ágil.

Together Tool Set, distribuido por Borland (www.borland.com o www.togethersoft.com), proporciona un paquete de herramientas que apoya muchas actividades técnicas dentro de la PE y otros procesos ágiles.

4.4 RESUMEN

Una filosofía ágil para la ingeniería del software se relaciona con cuatro aspectos clave: la importancia de la organización propia de los equipos, los cuales controlan el trabajo que realizan; comunicación y colaboración entre los miembros del equipo y entre los profesionales y sus clientes; un reconocimiento de que el cambio representa una oportunidad; y un especial cuidado en la entrega rápida del software que satisfaga al cliente. Los modelos de proceso ágil se diseñaron para cumplir con cada uno de estos aspectos.

La programación extrema (PE) es el proceso ágil que más se utiliza. Organizada como cuatro actividades del marco de trabajo —planeación, diseño, codificación y pruebas—, la PE sugiere algunas técnicas innovadoras y poderosas que permiten a un equipo ágil crear frecuentes lanzamientos de software al entregar características y funcionalidad que describe y después prioriza el cliente.

El desarrollo de software adaptativo (DSA) destaca la colaboración humana y la organización propia del equipo. Organizado con tres actividades del marco de trabajo —especulación, colaboración y aprendizaje—, el DSA utiliza un proceso iterativo que incorpora la planeación del ciclo adaptativo, métodos de recopilación de requisitos relativamente rigurosos y un ciclo iterativo de desarrollo que incorpora grupos enfocados en el cliente y revisiones técnicas formales como mecanismos de retroalimentación en tiempo real. El método de desarrollo de sistemas dinámicos (MDSD) define tres diferentes ciclos iterativos —iteración funcional del modelo, iteración de diseño y construcción e implementación— precedidos por dos actividades del ciclo de vida adicionales: estudio de factibilidad y estudio de negocios. El MDSD abo-

12 Las herramientas mencionadas aquí son sólo una muestra de esta categoría. En casi todos los casos los nombres son marcas registradas de sus respectivos desarrolladores.

ga por el uso de programas y sugiere que sólo se requiere el trabajo suficiente para cada incremento de software y así facilitar el movimiento hacia el incremento próximo

La melé subraya el uso de un conjunto de patrones de proceso de software que han probado su efectividad en proyectos con límites de tiempo muy ajustados, requisitos cambiantes y que son críticos para el negocio. Cada patrón de proceso define un conjunto de tareas de desarrollo y permite al equipo de melé construir un proceso que se adapte a las necesidades del proyecto.

Cristal es una familia de modelos ágiles de proceso que pueden adaptarse a las características específicas de un proyecto. Como otros enfoques ágiles, cristal adopta una estrategia iterativa, pero se ajusta al rigor del proceso para incluir proyectos de tamaños y complejidades diferentes.

El desarrollo conducido por características (DCC) es algo más "formal" que otros métodos ágiles, pero aun así mantiene la agilidad al enfocar al equipo de proyecto en el desarrollo de características, funciones que evalúa el cliente y que se pueden implementar en dos semanas o menos. El DCC concede una mayor importancia al proyecto y a su gestión que otros enfoques ágiles. El modelado ágil (MA) sugiere que el modelado es esencial para todos los sistemas, pero que la complejidad, tipo y tamaño del modelo debe ajustarse al software que será construido. Mediante la proposición de una serie de principios de modelado esenciales y complementarios, el MA proporciona una guía útil para los profesionales durante las tareas de análisis y diseño.

Referencias

[ADM96] Advanced Development Methods, Inc., "Origins of Scrum", 1996, http://www.controlchaos.com/.

[AGI03] The Agile Alliance Home Page, http.//www.agilealliance org/home

[AMB02] Ambler, S., "What is Agile Modeling (AM)?", 2002, http://www agilemodeling.com/index htm

[BEC99] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999

[BEC01a] Beck, K. *et al* , "Manifesto for Agile Software Development", http://www.agilemanifesto org/

[BEC01b] Beck, K. y M Fowler, *Planning Extreme Programming*, Addison-Wesley, 2001.

[BEE99] Beedle, M. *et al.*, "SCRUM· An extension pattern language for hyperproductive software development", incluido en *Pattern Languages of Program Design 4*, Addison Wesley, Longman, Reading, MA, 1999. Obtenido de http://jeffsutherland.com/scrum/scrum-plop.pdf

[BUS00] Buschmann, F. *et al* , *Pattern-Oriented Software Architecture*, 2 volúmenes, Wiley, 1996, 2000.

[COA99] Coad, P., E. Lefebvre y J. DeLuca, *Java Modeling in Color with UML*, Prentice-Hall, 1999

[COC01] Cockburn, A. y J. Highsmith, "Agile Software Development: The People Factor", *IEEE Computer*, vol 34, núm. 11, noviembre de 2001, pp. 131-133.

[COC02a] Cockburn, A., *Agile Software Development*, Addison-Wesley, 2002

[COC02b] Cockburn, A., "What is Agile and What Does It Imply?", presentado en el Agile Development Summit en Westminster College en Salt Lake City, marzo de 2002, http://crystalmethodologies.org/

[CCS02] CS3 Consulting Services, 2002, http:/www.cs3inc.com/DSDM.htm

[DEM98] DeMarco, T. y T. Lister, *Peopleware*, 2a ed , Dorset House, 1998

[DEM02] DeMarco, T. y B. Boehm, "The Agile Methods Fray", en *IEEE Computer*, vol. 35, núm 6, junio de 2002, pp 90-92.

[FOW00] Fowler, M. *et al.*, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
[FOW01] Fowler M. y J. Highsmith, "The Agile Manifesto", *Software Development Magazine*, agosto de 2001, http://www.sdmagazine.com/documents/s=844/sdm0108a.htm.
[FOW02] Fowler, M., "The New Methodology", junio de 2002, http://www.martinfowler.com/articles/newMethodology.html#N8B.
[HIG98] Highsmith, J., "Life—The Artificial and the Real", *Software Development*, 1998, en http://www.adaptivesd.com/articles/order.html
[HIG00] Highsmith, J., *Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems*, Dorset House Publishing, 1998.
[HIG01] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 1", *Cutter IT Journal*, vol. 14 núm. 12, diciembre de 2001.
[HIG02a] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 2", *Cutter IT Journal*, vol. 15, núm. 1, enero de 2002
[HIG02b] Highsmith, J., *Agile Software Development Ecosystems*, Addison-Wesley, 2002.
[JAC02] Jacobson, I, "A Resounding 'Yes' to Agile Processes—But Also More", *Cutter IT Journal*, vol. 15, núm. 1, enero de 2002, pp. 18-24.
[JEF01] Jeffries, R, *et al.*, *Extreme Programming Installed*, Addison-Wesley, 2001
[NOY02] Noyes, B., "Rugby, Anyone?", *Managing Development* (una publicación en línea de Fawcette Technical Publications), junio de 2002, http://www.fawcette.com/resources/managingdev/methodologies/scrum/
[PAL02] Palmer, S. y J. Felsing, *A Practical Guide to Feature Driven Development*, Prentice-Hall, 2002.
[SCH01] Schwaber, K. y M. Beedle, *Agile Software Development with SCRUM*, Prentice-Hall, 2001
[SCH02] Schwaber, K, "Agile Processes and Self Organization", Agile Alliance, 2002, http://www.aanpo.org/articles/index.
[STA97] Stapleton, J., *DSDM—Dynamic System Development Method: The Method in Practice*, Addison Wesley, 1997.
[WEL99] Wells, D., "XP—Unit Tests", 1999, http://www.extremeprogramming.org/rules/unittests.html

Problemas y puntos a considerar

4.1. Léase de nuevo el "Manifiesto para el desarrollo ágil de software" al principio de este capítulo. ¿El lector puede pensar en una situación en la que uno o más de los cuatro "valores" pueda meter a un equipo de software en problemas?

4.2. Describase agilidad (para proyectos de software) con palabras propias.

4.3. ¿Por qué un proceso iterativo facilita más manejar el cambio? ¿Todos los procesos ágiles tratados en este capítulo son iterativos? ¿Es posible concluir un proyecto en sólo una iteración y aún así seguir siendo ágil? Expliquense las respuestas.

4.4. ¿Podría cada uno de los procesos ágiles describirse recurriendo a las actividades genéricas del marco de trabajo mencionadas en el capítulo 2? Construyase una tabla que coloque las actividades genéricas dentro de las actividades definidas para cada proceso ágil.

4.5. Trátese de idear un "principio de agilidad" adicional que pudiera ayudar a una equipo de ingeniería del software a volverse aún más manejable.

4.6. Selecciónese un principio de agilidad de los enunciados de la sección 4.1 y trátese de determinar si cada uno de los modelos de proceso presentados en este capítulo muestran el principio.

4.7. ¿Por qué cambian tanto los requisitos? Después de todo, ¿la gente no sabe lo que quiere?

4.8. Considérense los siete rasgos enunciados en la sección 4.2.2. Ordénense los rasgos con base en su percepción desde lo que es más importante hasta lo que tiene menor importancia.

4.9. La mayoría de los procesos ágiles recomiendan la comunicación cara a cara. Aun en la actualidad, los miembros de un equipo de software y sus clientes pueden estar geográficamen-

te separados entre sí. ¿Esto implica la necesidad de evitar la separación geográfica? ¿Es posible pensar en formas de contrarrestar este problema?

4.10 Escríbase una historia del usuario para PE que describa los "sitios favoritos" o la característica de "favoritos" disponible en la mayoría de los exploradores Web

4.11. ¿Qué es una solución pico en PE?

4.12. Describanse los conceptos de PE *refabricación* y *programación en pareja* con palabras propias.

4.13. Utilícese la plantilla del patrón de proceso presentada en el capítulo 2 y desarróllese un patrón de proceso para cualquiera de los patrones de melé presentados en la sección 4 3 4

4.14. ¿Por qué se dice que cristal es una *familia de métodos ágiles*?

4.15. Utilícese la plantilla de característica para el DCC descrito en la sección 4.3.6 y defínase un conjunto de características de un explorador Web. Ahora desarróllese un conjunto de características para el conjunto mencionado antes.

4.16. Visítese el sitio oficial del modelado ágil y hágase una lista completa de todos los principios de MA esenciales y complementarios.

Otras lecturas y fuentes de información

La filosofía total y los principios subyacentes del desarrollo ágil de software se consideran a profundidad en los libros de Ambler (*Agile Modeling*, Wiley, 2002), Beck [BEC99], Cockburn [COC02] y Highsmith [HIG02b]

Los libros de Beck [BEC99], Jeffries y sus colegas (*Extreme Programming Installed*, Addison-Wesley, 2000), Succi y Marchesi (*Extreme Programming Examined*, Addison Wesley, 2001), Newkirk y Martin (*Extreme Programming in Practice*, Addison Wesley, 2001) y Aver y sus colegas (*Extreme Programming Applied Play to Win*, Addison-Wesley, 2001) ofrecen una exposición de los pros y contras de la PE junto con una guía de la mejor forma de aplicarla. McBreen (*Questioning Extreme Programming*, Addison Wesley, 2003) asume una posición crítica con respecto a la PE, al definir cuándo y dónde ésta es apropiada. Por otro lado, McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003) presenta una consideración profunda de la programación en pareja

Fowler y sus colegas (*Refactoring. Improving the Design of Existing Code*, Addison-Wesley, 1999) se enfoca en un nivel muy detallado en el importante concepto de la refabricación dentro de la PE. McBreen (*Software Craftsmanship: The New Imperative*, Addison-Wesley, 2001) analiza el arte del software y aboga a favor de las alternativas ágiles y la ingeniería de software tradicional.

El DSA lo aborda a profundidad Highsmith [HIG00]. Stapelton realizó un tratamiento valioso del MDSD (*DSDM The Method in Practice*, Addison-Wesley, 1997) Palmer y Felsing [PAL02] presentan un tratamiento detallado del DCC Carmichael y Haywood (*Better Software Faster*, Prentice-Hall, 2002) presentan otro útil tratamiento del DCC que incluye un recuento paso a paso por la mecánica del proceso. Schwaber y sus colegas (*Agile Software Development with SCRUM*, Prentice-Hall, 2001) presentan un detallado tratamiento de la melé.

Martin (*Agile Software Development*, Prentice-Hall, 2003) analiza los principios, patrones y prácticas ágiles poniendo especial cuidado en la PE. Poppendieck y Poppendieck (*Lean Development, An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) proporciona las directrices para manejar y controlar los proyectos ágiles. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) presenta una valiosa visión de principios, procesos y prácticas ágiles.

En Internet se dispone de una amplia variedad de fuentes de información sobre el desarrollo ágil de software. En el sitio web de SEPA se puede encontrar una lista actualizada de referencias en la red mundial, las cuales son relevantes para el proceso ágil: **http://www.mhhe.com/pressman.**

PARTE

Dos

PRÁCTICA DE LA INGENIERÍA DEL SOFTWARE

En esta parte de *Ingeniería del software: un enfoque práctico* se aprenderá acerca de los principios, conceptos y métodos que comprende la práctica de la ingeniería del software. En los capítulos siguientes se abordarán las siguientes preguntas:

- ¿Qué conceptos y principios guían la práctica de la ingeniería del software?
- ¿De qué manera la ingeniería de sistemas conduce a una ingeniería del software efectiva?
- ¿Qué es la ingeniería de requisitos, y cuáles son los conceptos relacionados que conducen a un buen análisis de requisitos?
- ¿Cómo se crea el modelo de análisis y cuáles son sus elementos?
- ¿Qué es ingeniería de diseño y cuáles son los conceptos relacionados que conducen a un buen diseño?
- ¿Qué conceptos, modelos y métodos se utilizan para crear diseños arquitectónicos, de interfaz y al nivel de componentes?
- ¿Qué estrategias son aplicables a la realización de pruebas de software?
- ¿Qué métodos se utilizan para diseñar casos de prueba efectivos?
- ¿Qué mediciones y métricas se usan para establecer la calidad del análisis y los modelos de diseño, código fuente y casos de prueba?

Una vez que estas preguntas hayan sido respondidas, habrá una mejor preparación para la práctica de la ingeniería del software

CAPÍTULO

5 La práctica: una visión genérica

Conceptos clave

En un libro que explora las vidas y los pensamientos de los ingenieros de software, Ellen Ullman [ULL97] representa un fragmento de vida al relatar los pensamientos de un practicante bajo presión:

> No tengo idea de qué hora es. No hay ventanas en esta oficina, tampoco reloj, sólo el LED parpadeante en rojo de un horno de microondas, el cual parpadea 12:00, 12:00, 12 00, 12:00. Joel y yo hemos estado programando por días Tenemos un "bicho", el necio demonio de un error Por eso la pulsación roja sin tiempo se siente bien, como una lectura de nuestros cerebros, los cuales se han sincronizado de alguna manera a la misma velocidad del parpadeo
>
> ¿En qué estamos trabajando?.. Los detalles se me escapan ahora. Podríamos estar ayudando a gente pobre y enferma o ajustando una serie de rutinas de bajo nivel para verificar bits en un protocolo de una base de datos distribuida, no me importa. Me debería importar; en otra parte de mí ser —más tarde, quizá cuando salgamos de este cuarto lleno de computadoras— me importará mucho por qué, para quién y con qué propósito estoy escribiendo software Pero ahora no. He pasado a través de una membrana donde el mundo real y sus usos ya no importan Soy un ingeniero de software.

Sin duda, una imagen oscura de la práctica de la ingeniería del software, pero con la que, después de reflexionar, muchos de los lectores de este libro serán capaces de identificarse.

Un vistazo rápido

¿Qué es? La práctica es un amplio arreglo de conceptos, principios, métodos y herramientas que deben considerarse cuando se planea y desarrolla el software. Representa los detalles –las consideraciones técnicas y los cómos– que están bajo la superficie del proceso de software: las cosas que se necesitarán para realmente construir software de computadora de alta calidad.

¿Quién lo hace? La práctica de la ingeniería del software la aplican los ingenieros de software y sus gerentes.

¿Por qué es importante? El proceso del software proporciona a todos los involucrados en la creación de un sistema o producto basado en computadora un mapa del camino para llegar de manera exitosa a su destino. La práctica proporciona los detalles que se necesitan para transitar a lo largo del camino. Indica dónde están ubicados los puentes, los bloqueos del camino y los obstáculos. Ayuda a entender los conceptos y principios que se deben comprender y seguir para conducir de manera segura y rápida. Enseña cómo conducir, dónde reducir y dónde aumentar la velocidad. En el contexto de la ingeniería del software, la práctica es lo que se realiza a diario mientras el software evoluciona desde una idea hasta una realidad.

¿Cuáles son los pasos? Existen tres elementos de la práctica que se aplican sin importar el modelo de proceso que se escoja Éstos son los

conceptos, los principios y los métodos. Un cuarto elemento de la práctica –las herramientas– apoya la aplicación de los métodos.

¿Cuál es el producto obtenido? La práctica incluye las actividades técnicas que producen algunos productos de trabajo definidos por el modelo de proceso del software que se ha elegido.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Primero se debe comprender con firmeza los conceptos y principios aplicables al trabajo que se realiza en el momento (por ejemplo, el diseño). Después es preciso asegurarse que se ha seleccionado un método apropiado para el trabajo; se debe tener la certeza que se ha entendido la forma de aplicar el método y el uso de las herramientas automatizadas cuando éstas son apropiadas para la tarea, y se debe ser firme en la necesidad de usar técnicas para asegurar la calidad de los productos de trabajo que se produzcan.

Las personas que crean software de computadora practican el arte, la maestría o la disciplina[1] llamada ingeniería del software. Pero, ¿qué es la "práctica" de la ingeniería del software? En un sentido genérico, la *práctica* es una colección de conceptos, principios, métodos y herramientas a las que un ingeniero de software recurre a diario. La práctica permite a los gerentes coordinar proyectos de software e ingenieros de la especialidad para construir programas de computadora. La práctica multiplica un modelo de proceso de software con los cómos técnicos y de gestión necesarios para realizar el trabajo. La práctica transforma un enfoque fortuito en algo más organizado, más efectivo y con más probabilidades de alcanzar el éxito.

5.1 LA PRÁCTICA DE LA INGENIERÍA DEL SOFTWARE

Información Web

En el capítulo 2 se introdujo un modelo de proceso de software genérico compuesto de una serie de actividades que establecen un marco de trabajo para la práctica de la ingeniería del software. Las actividades genéricas del marco de trabajo —comunicación, planeación, modelado, construcción y despliegue— y las actividades sombrilla establecen la arquitectura de una esqueleto para el trabajo de ingeniería del software. Todos los modelos de proceso de software presentados en los capítulos 3 y 4 pueden organizarse en este esqueleto arquitectónico. ¿Pero qué cabida tiene ahí la práctica de la *ingeniería del software*? En las secciones siguientes se considerarán los conceptos y principios genéricos que se aplican a las actividades del marco de trabajo.[2]

1 Algunos escritores utilizan uno de estos términos y excluyen los otros. En realidad, la ingeniería del software es las tres cosas a la vez.

2 Se exhorta al lector para que revise las secciones relevantes dentro de este capítulo, conforme se discutan los métodos específicos de la ingeniería del software y las actividades sombrilla en los capítulos posteriores del libro.

5.1.1 La esencia de la práctica

Se podría argumentar que el enfoque de Polya consiste en simple sentido común. Es verdad. Pero es sorprendente la frecuencia con la que el sentido común no es común en el mundo del software

En un libro clásico, *How to Solve It*, escrito antes de que existieran las computadoras modernas, George Polya [POL45] puntualizó la esencia de la resolución de problemas y, en consecuencia, la esencia de la práctica de la ingeniería del software:

1. *Entender el problema* (comunicación y análisis).
2. *Planear una solución* (modelado y diseño de software).
3. *Llevar a cabo el plan* (generación de código).
4. *Examinar el resultado para probar la precisión* (realización de pruebas y aseguramiento de la calidad).

En el contexto de la ingeniería del software estos pasos de sentido común conducen a una serie de preguntas esenciales [adaptadas de POL45]:

Entender el problema.

- *¿A quién le interesa la solución del problema?* Es decir, ¿quiénes son los clientes?
- *¿Cuáles aspectos se desconocen?* ¿Qué datos, funciones, características y comportamientos se requieren para resolver de manera apropiada el problema?
- *¿El problema puede dividirse en categorías?* ¿Es posible representar problemas menores que puedan entenderse con mayor facilidad?
- *¿El problema puede representarse de manera gráfica?* ¿Se puede crear un modelo de análisis?

Planear la solución.

- *¿Se habían visto problemas similares antes?* ¿Existen patrones reconocibles en una solución potencial? ¿Hay un software existente que implemente los datos, las funciones, las características y los comportamientos que se requieren?
- *¿Se ha resuelto un problema similar?* Si es así, ¿los elementos de la solución pueden reutilizarse?
- *¿Se pueden definir los subproblemas?* Si es así, ¿las soluciones para los subproblemas parecen fáciles?
- *¿Se puede representar una solución de modo que conduzca a una implementación efectiva?* ¿Se puede crear un modelo de diseño?

Llevar a cabo el plan.

- *¿La solución marcha conforme al plan?* ¿El código fuente se puede seguir conforme al modelo de diseño?
- *¿Es probable que cada parte de la solución del componente sea correcta?* ¿Se ha revisado el diseño y el código, o mejor aún, se han aplicado al algoritmo pruebas de corrección?

Examinar el resultado.

- *¿Es posible probar cada parte de la solución del componente?* ¿Se ha implementado una estrategia de prueba razonable?
- *¿La solución produce resultados acordes con los datos, funciones, rasgos y comportamientos que se requieren?* ¿El software ha sido validado contra todos los requisitos de los clientes?

> "En la solución de cada problema existe un grano de descubrimiento."
>
> **George Polya**

5.1.2 Principios esenciales

El diccionario define la palabra *principio* como "una ley o supuesto importante que se requiere en un sistema de pensamiento". A través de este libro se examinan principios en muchos grados diferentes de abstracción. Algunos se enfocan en la ingeniería del software como un todo, otros consideran una actividad genérica y específica del marco de trabajo (por ejemplo, comunicación con el cliente), y otros se centran en acciones de la ingeniería del software (como diseño arquitectónico) o tareas técnicas (escribir un escenario de uso). Sin importar qué tan específicos son, los principios ayudan a establecer un conjunto sólido de práctica de ingeniería del software. Por esa razón son importantes.

David Hooker [HOO96] ha propuesto siete principios esenciales, los cuales se enfocan en la práctica de la ingeniería del software como un todo, que se reproducen enseguida [3]

El primer principio: *la razón por la que todo existe*

CONSEJO

...e comienza un ...e software, ...e estar seguro ...e un ...o de negocios ...s usuarios ...r en él

Un sistema de software existe por una razón: *para ofrecer un valor a sus usuarios.* Todas las decisiones deben tomarse con esto en mente. Antes de especificar un requisito de un sistema, antes de señalar una pieza de funcionalidad del sistema, antes de determinar las plataformas del hardware o los procesos de desarrollo, uno debe hacerse preguntas como: ¿esto agrega un valor real al sistema? Si la respuesta es negativa no se debe hacer Todos los demás principios están apoyados en éste

El segundo principio: *MS (mantenerlo simple)*

El diseño de software no es un proceso fortuito Existen muchos factores que deben considerarse en cualquier esfuerzo de diseño. *Todo el diseño debe ser tan simple como sea posible, pero no más simple.* Esto facilita un sistema de más fácil comprensión y entendimiento. Esto no quiere decir que las características, hasta las internas, deban descartarse en nombre de la simplicidad. Además, los diseños más elegantes por lo general son los más simples. Simple tampoco significa "rápido y malo". De hecho, se

3 Reproducido con permiso del autor [HOO96] Hooker define patrones para estos principios en http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment

requiere de mucha reflexión y trabajo sobre múltiples iteraciones para simplificar. El resultado buscado es un software que se mantenga y sea menos propenso al error.

> "Existe cierta majestuosidad en la simplicidad, la cual está muy por encima de la curiosidad del ingenio."
> **Alexander Pope (1688-1744)**

El tercer principio: *mantener la visión*

Una visión clara es esencial para el éxito de un proyecto de software. Sin la visión clara el proyecto podría terminar con "dos [o más] significados" en uno. Sin una integridad conceptual un sistema amenaza con tornarse en una masa confusa de diseños incompatibles, unida por un tipo inadecuado de tornillos...

Arriesgar la visión arquitectónica de un sistema de software debilita y al final rompe hasta un sistema bien diseñado. Tener a un arquitecto capaz de mantener la visión y reforzar lo acordado ayuda al aseguramiento de que un proyecto de software sea exitoso.

El cuarto principio: *lo que uno produzca, otros lo consumirán*

PUNTO CLAVE

Si el software tiene un valor, éste cambiará a lo largo de su vida útil. Por esa razón, el software debe construirse de forma que se le pueda dar mantenimiento.

En muy pocas ocasiones un sistema de software con fuerza industrial se construye y utiliza de manera aislada. De alguna u otra forma, alguien más utilizará, mantendrá, documentará o dependerá de su capacidad de entender el sistema. Por lo tanto, siempre debe especificarse, diseñarse e implementarse con la idea de que alguien más tendrá que entender lo que se realice. La audiencia de cualquier producto de software es potencialmente grande, por lo que se debe especificar tomando en cuenta a los usuarios. Se debe diseñar teniendo en mente a quienes lo implementen, así como codificar considerando a aquellos que deben mantener y extender el sistema. Tal vez alguien tenga que depurar el código escrito y eso lo convertirá en un usuario del código. El hecho de facilitar el trabajo a otro agrega valor al sistema.

El quinto principio: *estar abierto al futuro*

Un sistema con una larga vida tiene más valor. En los ambientes computacionales de la actualidad, en los que las especificaciones cambian a cada momento y las plataformas de hardware son obsoletas después de algunos meses, la vida del software se mide, de modo típico, en meses en lugar de años. No obstante, los verdaderos sistemas de software "con fuerza industrial" deben durar más tiempo. Estos sistemas tendrán éxito si están listos para adaptarse a éstos y otros cambios. Los sistemas que logran el éxito son aquellos que han sido diseñados de esta manera desde el principio. *Nunca se debe diseñar para llegar a una esquina.* Uno siempre se debe preguntar: "¿qué tal si?", y prepararse para todas las respuestas posibles, al crear sistemas que resuelvan el problema general, no un problema específico.[4] Es muy probable que esto conduzca a la reutilización de un sistema entero.

4 Nota del autor: esta recomendación puede ser peligrosa si se lleva hasta el extremo. El diseño para el "problema general" algunas veces requiere compromisos de desempeño y puede implicar un mayor esfuerzo para el proyecto.

El sexto principio: *planear para la reutilización*

La reutilización ahorra tiempo y esfuerzo.[5] Al alcanzar un alto grado de reutilización se logra una de las metas más difíciles en el desarrollo de un sistema de software La reutilización de código y diseños ha sido proclamada como un beneficio importante del uso de tecnologías orientadas a objetos. Sin embargo, la recuperación de la inversión no es automática. Las posibilidades de reutilización que proporciona la programación orientada a objetos (o convencional) se podrán considerar si se tiene una visión a futuro y una planeación. Existen muchas técnicas para llevar a cabo la reutilización en cada etapa del proceso de desarrollo del sistema; las relativas al diseño detallado y al nivel de código son muy conocidas y están bien documentadas La nueva bibliografía se está enfocando en la reutilización del diseño en la forma de patrones de software Sin embargo, esto es sólo una parte de la batalla.

La comunicación de oportunidades para la reutilización a otros integrantes de la organización es primordial. ¿Cómo se puede reutilizar algo cuya existencia se ignora? *La planeación adelantada para la reutilización reduce el costo e incrementa el valor de los componentes reutilizables y los sistemas en que dichos componentes se incorporan.*

El séptimo principio: *pensar*

Este último principio tal vez sea el que más se pasa por alto. *Casi siempre, cuando se tiene un pensamiento claro y completo antes de la acción, se producen los mejores resultados* Cuando se reflexiona acerca de algo existe una mayor probabilidad de hacerlo bien Siempre se obtiene conocimiento de la manera de hacerlo bien de nuevo Si se piensa en algo y aun así se hace mal, esto se convierte en una experiencia valiosa. Un efecto colateral del pensamiento es aprender a reconocer, cuando alguien no sabe algo, en qué punto se puede investigar la respuesta Cuando el pensamiento claro se ha introducido en el sistema es cuando surge su valor real. La aplicación de los primeros seis principios requiere una reflexión intensa, por lo que las recompensas potenciales son enormes

Si todos los ingenieros de software y todos los equipos de desarrollo tan sólo siguieran los siete principios de Hooker, muchas de las dificultades que se han experimentado durante la construcción de sistemas complejos basados en computadora se podrían eliminar.

5.2 Prácticas de comunicación

Antes de que los requisitos del cliente puedan analizarse, modelarse o especificarse, éstos deben recopilarse por medio de una actividad de *comunicación* (también llamada *obtención de requisitos*) Un cliente tiene un problema que se puede ajustar a

5 Nota del autor aunque esto es cierto para quienes reutilizan el software en proyectos futuros, la reutilización puede resultar cara para quienes deben diseñar y construir componentes reutilizables Algunos estudios indican que el diseño y la construcción de componentes reutilizables pueden costar entre 25 y 200 por ciento más que el software solicitado. En algunos casos, la diferencia de costo no se puede justificar

una solución basada en computadora. Un desarrollador responde a la solicitud de ayuda del cliente. La comunicación ha comenzado. Pero el camino desde la comunicación hasta el entendimiento suele estar lleno de baches.

La comunicación efectiva (entre pares técnicos, con el cliente u otros participantes del software y con los gerentes de proyecto) está entre las actividades más desafiantes que enfrenta un ingeniero de software. En este contexto se examinan los principios o conceptos de comunicación de acuerdo con la manera en que se aplican en la comunicación con el cliente. Sin embargo, muchos de los principios se aplican del mismo modo a las formas de comunicación que ocurren dentro de un proyecto de software.

Principio #1: *Escuchar.* Se debe centrar la atención en las palabras de quien habla, en vez de formular la respuesta a dichas palabras. Es necesario pedir una explicación si algo no está claro, pero deben evitarse las interrupciones constantes. Nunca se debe ser conflictivo con palabras o actitudes (por ejemplo, dirigir la mirada a los lados o sacudir la cabeza) cuando una persona habla

CONSEJO

Antes de comunicar se debe estar seguro de entender el punto de vista de la otra parte, saber un poco de sus necesidades, y entonces opinar

Principio #2: *Prepararse antes de comunicar.* Se debe invertir algún tiempo en entender el problema antes de reunirse con otros. Si es necesario, se puede realizar una investigación para entender el negocio y dominar la jerga. Si se tiene la responsabilidad de conducir la reunión, es recomendable preparar una agenda del día antes de la junta

Principio #3: *Alguien debe facilitar la actividad.* Cualquier reunión de comunicación debe tener un líder o mediador 1) para mantener una conversación dinámica y en una dirección productiva; 2) para mediar en cualquier conflicto que ocurra; 3) para asegurar que se sigan los otros principios.

Principio #4: *La comunicación cara a cara es lo mejor.* Pero, por lo general funciona mejor cuando está presente otra representación de la información relevante. Por ejemplo, un participante podría crear un esquema o un documento que sirva como foco de la discusión

> "Las preguntas y las respuestas llanas forman el camino más corto hacia la mayoría de las perplejidades."
>
> Mark Twain

Principio #5: *Tomar notas y documentar las decisiones.* Las cosas suelen caer en malentendidos. Alguien que participe en la comunicación debe servir como "grabadora" y escribir todos los puntos y decisiones importantes.

Principio #6: *Buscar la colaboración.* La colaboración y el consenso se presentan cuando el conocimiento colectivo de los miembros del equipo se combina para describir las funciones o características del producto o sistema. Cada pequeña colaboración sirve para construir confianza entre los miembros del equipo y crear una meta común para dicho equipo

Principio #7: *Conservar el enfoque, examinar un módulo a la vez.* Entre más gente esté implicada en una comunicación, más posibilidades existen de que la

discusión salte de un tópico al siguiente. El mediador debe mantener la conversación centrada en un módulo sin dejar un tema hasta que éste haya sido resuelto (sin embargo, véase el principio #9)

INFORMACIÓN

La diferencia entre clientes y usuarios finales

Los ingenieros de software se comunican con muchos participantes diferentes, pero los clientes y los usuarios finales tienen el impacto más significativo sobre el trabajo técnico que sigue. En algunos casos el cliente y el usuario son uno mismo, pero en muchos proyectos el cliente y el usuario final son personas diferentes, que trabajan para diferentes administradores en diferentes organizaciones de negocios.

El *cliente* es la persona o grupo que: 1) en un inicio solicita el software que se va a construir; 2) define los objetivos generales de negocios para el software; 3) proporciona los requisitos básicos del producto; 4) coordina los recursos económicos para el proyecto. En un negocio de productos o sistemas, con frecuencia el cliente es el departamento de mercadotecnia. En un ambiente de TI el cliente puede ser un componente o departamento del negocio.

Un *usuario final* es la persona o grupo que: 1) en realidad usará el software que se construye para alcanzar algún propósito de negocios, y 2) definirá los detalles operativos del software de forma que el propósito del negocio pueda alcanzarse.

Principio #8: ***Si algo no está claro, se hace un dibujo.*** La comunicación verbal sólo llega hasta cierto punto. Con frecuencia un esquema o figura puede proporcionar claridad cuando las palabras fallan al realizar su trabajo.

Principio #9: ***a) Una vez que se llega a un acuerdo sobre algo, se debe continuar; b) si no se puede llegar a un acuerdo, se debe continuar; c) si una característica o función no está clara y no se puede clarificar en el momento, se debe continuar.*** La comunicación, como cualquier actividad de ingeniería del software, requiere de tiempo. En lugar de que se itere sin fin, los participantes deben reconocer que hay muchos tópicos que requieren análisis (véase el principio #2) y que "continuar" algunas veces es la mejor forma agilitar la comunicación.

¿Qué pasa si no se puede llegar a un arreglo con el cliente en algún asunto relacionado con el proyecto?

Principio #10: ***La negociación no es un concurso o un juego. Funciona mejor cuando ambas partes ganan.*** En muchas ocasiones los ingenieros de software y el cliente deben negociar funciones y características, prioridades y fechas de entrega. Si el equipo ha colaborado de buena forma, todas las partes tienen una meta común. Por lo tanto, la negociación demandará el compromiso de todas las partes.

HOGARSEGURO

Errores de comunicación

El escenario: Lugar de trabajo del equipo de ingeniería del software.

Los actores: Jamie Lazar, miembro del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, miembro del equipo de software.

La conversación:

Ed: ¿Qué han oído acerca de este proyecto de *HogarSeguro*?

Vinod: La reunión de inicio está programada para la próxima semana.

Jamie: Yo ya he investigado un poco, pero no me fue bien.

Ed: ¿Qué quieres decir?

Jamie: Bueno, le hice una llamada a Lisa Pérez, la responsable de mercadotecnia en este asunto.

Vinod: ¿Y ...?

Jamie: Yo quería que me dijera algo sobre las características y funciones de *HogarSeguro*... ese tipo de cosas. En lugar de eso, comenzó a hacerme preguntas sobre seguridad de sistemas, vigilancia de sistemas ... no soy una experta.

Vinod: ¿Qué le dice eso?

[Jamie se encoge de hombros.]

Vinod: Que mercadotecnia nos necesitaría para que actuemos como consultores y que será mejor que hagamos algo de tarea en esta área del producto antes de la reunión de inicio. Doug dijo que quería que "colaboráramos" con nuestro cliente, así que será mejor que aprendamos cómo hacerlo.

Ed: Probablemente sería mejor que fuéramos a su oficina. Las llamadas telefónicas no funcionan para este tipo de asuntos.

Jamie: Ambos están en lo correcto. Debemos actuar juntos o nuestras primeras comunicaciones serán un lío.

Vinod: Vi que Doug leía un libro sobre "ingeniería de requisitos". Podría apostar que ahí se enlistan algunos principios para la buena comunicación. Se lo voy a pedir prestado mañana.

Jamie: Buena idea ... después nos puedes enseñar.

Vinod (sonriendo): Sí, claro.

CONJUNTO DE TAREAS

Conjunto de tareas genéricas para la comunicación

1. Identificar al cliente primario y otros participantes (sección 7.3.1).
2. Reunirse con el cliente primario para las "preguntas libres de contexto" (sección 7.3.4), las cuales definen:
   - Las necesidades y valores del negocio.
   - Las características y necesidades del usuario final.
   - Las salidas visibles que se hayan requerido para el usuario.
   - Las restricciones del negocio.
3. Desarrollar un enunciado escrito de una página sobre el ámbito del proyecto, el cual está sujeto a revisión.
4. Revisar el enunciado del ámbito con los interesados en el software y ajustarlo según lo requerido.
5. Colaborar con el cliente y el usuario final para definir:
   - Escenarios de uso visibles para el cliente con el uso del formato estándar[6] (sección 7.5).
   - Salidas y entradas resultantes.
   - Características, funciones y comportamientos importantes del software.
   - Riesgos de negocios definidos por el cliente (sección 25.3).
6. Desarrollar una breve descripción escrita (por ejemplo, una serie de listas) de escenarios, salidas/entradas, características/funciones y riesgos.
7. Iterar con el cliente para refinar los escenarios, salidas/entradas, características/funciones y riesgos.
8. Asignar prioridades definidas por el cliente a cada escenario del usuario, característica, función y comportamiento (sección 7.4.2).
9. Revisar toda la información recopilada durante la actividad de comunicación con el cliente y otros participantes, y ajustarla de la forma que se requiera.
10. Prepararse para la actividad de planeación (capítulos 23 y 24).

6 Los formatos para escenarios de uso se discuten en el capítulo 8.

5.3 PRÁCTICAS DE LA PLANEACIÓN

La actividad de comunicación ayuda al equipo de software a definir sus metas y objetivos generales (por supuesto, sujeto al cambio conforme pasa el tiempo). Sin embargo, entender estas metas y objetivos no es lo mismo que definir un plan para llegar a ellos. La actividad de *planeación* abarca un conjunto de prácticas técnicas y de gestión que permiten al equipo de software definir un mapa del camino mientras se viaja a través de su meta estratégica y objetivos tácticos

> "En la preparación para la batalla siempre he encontrado que los planes son inútiles, pero que la planeación es indispensable."
>
> **Dwight D. Eisenhower**

Existen muchos enfoques diferentes para la planeación. Algunas personas son "minimalistas", y argumentan que el cambio con frecuencia obvia la necesidad de un plan detallado. Otros son tradicionalistas, y dicen que el plan proporciona un mapa efectivo del camino, y mientras más detallado sea éste, menor probabilidad habrá de que el equipo pierda el rumbo. Además, otros son "agilistas" y argumentan que tal vez un "juego de planeación" rápido sea necesario, pero que el mapa del camino surgirá cuando comience el "trabajo real" sobre el software.

¿Qué hacer? En muchos proyectos la sobreplaneación consume tiempo y no produce frutos (demasiadas cosas cambian), pero la planeación insuficiente es una receta para el caos. Como la mayoría de las cosas en la vida, la planeación se debe producir con moderación, lo suficiente para proporcionar una guía útil para el equipo —no más, no menos.

Sin importar el rigor con el que se conduzca la planeación, los siguientes principios son válidos en todo momento.

Principio #1: ***Entender los alcances del proyecto.*** Es imposible utilizar un mapa de carreteras si no se sabe el sitio a donde se quiere ir. El hecho de entender los alcances proporciona al equipo de software un destino.

Principio #2: ***Involucrar al cliente en la actividad de planeación.*** El cliente define prioridades y establece las restricciones del proyecto. El ajuste de estas realidades a menudo requiere que los ingenieros de software negocien las órdenes de entrega, los límites de tiempo y otros asuntos relacionados con el proyecto.

Principio #3: ***Reconocer que la planeación es iterativa.*** El plan de un proyecto nunca se graba en una piedra. En cuanto comience el trabajo es muy probable que las cosas cambien. En consecuencia, debe ajustarse el plan para adaptarlo a los cambios. Además, los modelos iterativos e incrementales del proceso dictan la replaneación (después de la entrega de cada incremento de software) basada en la retroalimentación recibida de los usuarios

Principio #4: ***Estimar con base en el conocimiento disponible.*** La finalidad de la estimación es proporcionar un indicio del esfuerzo, costo y duración de las tareas, con base en el conocimiento que el equipo tiene del trabajo que se va a realizar. Si la información es vaga o poco confiable, las estimaciones tendrán, de igual forma, poca confiabilidad.

Principio #5: ***Considerar el riesgo cuando se define el plan.*** Si el equipo ha definido riesgos que tienen un alto impacto y una alta probabilidad, es necesario un plan de contingencia. Además, el plan de proyecto (que incluye el calendario) se debe ajustar para incluir la posibilidad de que uno o más de estos riesgos se torne un problema real.

Principio #6: ***Ser realista.*** Las personas no trabajan el 100 por ciento de cada día. El ruido siempre entra en cualquier comunicación humana. Las omisiones y la ambigüedad son hechos de la vida. Los cambios ocurrirán. Hasta los mejores ingenieros de software cometen errores. Éstas y otras realidades deben considerarse mientras se establece el plan del proyecto.

> "El éxito está más en función del sentido común consistente que del genio."
>
> **An Wang**

PUNTO CLAVE

El término granularidad se refiere al detalle con el que algunos elementos de la planeación se representan o dirigen.

Principio #7: ***Ajustar la granularidad mientras se define el plan.*** La *granularidad* se refiere al grado de detalle que se introduce conforme se desarrolla el plan. Una "granularidad fina" en el plan proporciona detalles significativos de las tareas de trabajo, los cuales se planean en incrementos relativamente cortos de tiempo (de forma que el ajuste y el control se den con frecuencia). Un plan de "granularidad gruesa" proporciona tareas de trabajo más amplias, las cuales se planean en periodos más largos. Por lo general, la granularidad cambia de fina a gruesa conforme el tiempo límite del proyecto está más lejos de la fecha actual. En las siguientes semanas o meses el proyecto se puede planear con detalle significativo. Las actividades que no se realizarán por muchos meses no requieren una granularidad fina (hay demasiadas cosas que pueden cambiar).

Principio #8: ***Definir cómo se intentará asegurar la calidad.*** El plan debe identificar la forma en que el equipo de software pretende asegurar la calidad. Si habrá revisiones técnicas formales,[7] éstas se deben calendarizar. En caso de que se utilice programación en pareja (capítulo 4) durante la construcción ésta debe estar definida de manera explícita en el plan.

Principio #9: ***Describir cómo se pretende incluir el cambio.*** Incluso a la mejor planeación puede superarla el cambio incontrolable. El equipo de software debe identificar la forma en que se incluirán los cambios conforme se realiza el trabajo de ingeniería del software. Por ejemplo, ¿el cliente puede solicitar un cambio en

7 Las revisiones técnicas formales se estudian en el capítulo 26.

cualquier momento? ¿Si se presenta una solicitud de cambio el equipo está obligado a implementarlo de inmediato? ¿Cómo se evalúan el impacto y el costo del cambio?

Principio #10: ***Adaptar el plan a menudo y hacer los ajustes cuando éstos se requieran.*** Día tras día los proyectos de software van por detrás del calendario establecido. Por ello, es de mucha ayuda adaptar el progreso a diario. Se deben buscar áreas problemáticas y situaciones en las que el trabajo calendarizado no vaya de acuerdo con el trabajo que se ejecuta en realidad. Cuando se encuentran desfases, el plan se ajusta en concordancia con ello.

En la búsqueda de mayor efectividad, todos los integrantes del equipo de software deben participar en la actividad de planeación. Sólo entonces son miembros del equipo "comprometidos" con el plan.

En un excelente documento sobre procesos y proyectos de software, Barry Boehm [BOE96] establece: "Se necesita un principio de organización que se reduzca para proporcionar planes [de proyecto] simples para proyectos simples." Boehm sugiere un enfoque dirigido a los objetivos, fundamentos y calendarios del proyecto, a las responsabilidades, enfoques técnicos y de gestión y a los recursos requeridos. Él lo llamó *principio W*^{5}*HH* (why, what, when, who, where, how, how), debido a una serie de preguntas que conducen a una definición de características clave del proyecto y el plan de proyecto resultante:

¿Cuáles son las preguntas que se deben hacer y responder para desarrollar un plan de proyecto exitoso?

¿Por qué está en desarrollo este sistema? Todas las partes deben evaluar la validez de las razones del negocio para el trabajo en el software. Dicho de otra manera, ¿el propósito del negocio justifica el gasto de personal, tiempo y dinero?

¿Qué se hará? Se debe identificar la funcionalidad que se construirá y, por ende, las tareas requeridas para realizar el trabajo.

¿Cuándo se terminará? Es necesario establecer un flujo de trabajo y un tiempo límite para las tareas clave del proyecto, así como identificar los fundamentos requeridos por el cliente.

¿Quién es el responsable de una función? Se deben definir el papel y la responsabilidad de cada miembro del equipo de software.

¿En dónde se ubican dentro de la organización? No todos los papeles y responsabilidades residen dentro del mismo equipo de software. El cliente, los usuarios y otros participantes también tienen responsabilidades.

¿Cómo se realizará el trabajo en los sentidos técnico y de gestión? Una vez que se establece el ámbito del producto, es necesario definir una estrategia técnica y de gestión para el proyecto.

¿Cuánto se necesita de cada recurso? La respuesta a esta pregunta se obtiene al desarrollar estimaciones (capítulo 23) con base en las respuestas a las preguntas anteriores.

Las respuestas a las preguntas W^{5}HH de Boehm son importantes independientemente del tamaño o la complejidad de un proyecto de software. Pero, ¿cómo se inicia el proceso de planeación?

> "Pensamos que los desarrolladores de software están perdiendo una verdad vital: la mayoría de las organizaciones no saben lo que hacen. Ellos piensan que lo saben, pero no es así"
>
> **Tom DeMarco**

CONJUNTO DE TAREAS

Conjunto de tareas genéricas para la planeación

1. Reevaluar el ámbito del proyecto (secciones 7.4 y 21.3).
2. Evaluar los riesgos (sección 25.4).
3. Desarrollar o refinar los escenarios del usuario (secciones 7.5 y 8.5).
4. Extraer funciones y características a partir de los escenarios (sección 8.5).
5. Definir las funciones y características técnicas que forman la infraestructura del software.
6. Agrupar las funciones y características (escenarios) de acuerdo con la prioridad del cliente.
7. Crear un plan de proyecto con una granularidad gruesa (capítulos 23 y 24).
   - Definir el número proyectado de incrementos de software.
   - Establecer un calendario general del proyecto (capítulo 24).
   - Establecer las fechas de entrega proyectadas para cada incremento
8. Crear un plan con granularidad fina para la iteración actual (capítulos 23 y 24).
   - Definir tareas de trabajo para cada función y característica (sección 23.6).
   - Estimar el esfuerzo para cada tarea de trabajo (sección 23.6).
   - Asignar responsabilidad para cada tarea de trabajo (sección 23.4).
   - Definir los productos de trabajo que serán producidos.
   - Identificar los métodos para el aseguramiento de la calidad que se usarán (capítulo 26).
   - Describir los métodos para el cambio en la gestión (capítulo 27).
9. Rastrear el progreso de manera regular (sección 24.5.2).
   - Observar las áreas problemáticas (por ejemplo, el desfase del calendario).
   - Hacer los ajustes que se requieran.

5.4 PRÁCTICA DEL MODELADO

Los modelos se crean para obtener un mejor entendimiento de la entidad real que se construirá. Cuando la entidad es un objeto físico (por ejemplo, un edificio, un avión, una máquina), se puede construir un modelo idéntico en forma y tamaño, pero en menor escala. Sin embargo, cuando la entidad es software, el modelo debe tomar una forma diferente. Debe ser capaz de representar la información que el software transforma, la arquitectura y las funciones que permiten que ocurra la transformación, las características que desean los usuarios, y el comportamiento del sistema conforme se realiza la transformación. Los modelos deben cumplir estos objetivos en diferentes grados de abstracción (primero al presentar el software desde el punto de vista del cliente y después al representar el software en un nivel más técnico).

PUNTO CLAVE

s de
s representan los
s del cliente.
modelos de diseño
un
n concreto
construcción

En el trabajo de la ingeniería del software se crean dos clases de modelos: modelos de análisis y modelos de diseño. Los *modelos de análisis* representan los requisitos del cliente al presentar el software en tres dominios diferentes: el dominio de la información, el dominio funcional y el dominio del comportamiento. Los *modelos de diseño* representan características del software que ayudan a los profesionales a construirlo de manera efectiva. la arquitectura (capítulo 10), la interfaz del usuario (capítulo 12), y el detalle al nivel de componentes (capítulo 11).

En las secciones siguientes se presentan los principios y conceptos básicos que son relevantes para el modelado del análisis y el diseño. Los métodos técnicos y la notación que permiten que los ingenieros de software creen modelos de análisis y diseño se presentan en los capítulos posteriores.

> "El primer problema del ingeniero en cualquier situación de diseño es descubrir cuál es realmente el problema."
>
> Anónimo

5.4.1 Principios del modelado del análisis

En las pasadas tres décadas se ha desarrollado un gran número de métodos de modelado del análisis. Los investigadores han identificado los problemas del análisis y sus causas y han desarrollado una variedad de notaciones de modelado y los conceptos heurísticos correspondientes para manejarlos. Cada método de análisis tiene un punto de vista único. Sin embargo, todos los métodos de análisis están relacionados por medio de una serie de principios operativos:

Principio #1: ***El dominio de información de un problema debe representarse y entenderse.*** El *dominio de información* lo forman los datos que fluyen hacia el sistema (a partir de los usuarios finales, otros sistemas o dispositivos externos), los datos que fluyen desde el sistema (a través de la interfaz del usuario, interfases de red, reportes, gráficas y otros medios) y los almacenamientos de datos que se recopilan y reorganizan los objetos consistentes de información (es decir, los datos que se mantienen en forma permanente).

PUNTO CLAVE

o del
su enfoque en
s del
la
o que se
; la
n se debe
y el
o que

Principio #2: ***Se deben definir las funciones que ejecuta el software.*** Las funciones del software proporcionan un beneficio directo a los usuarios finales y también aporta soporte interno a aquellas características visibles para el usuario. Algunas funciones transforman los datos que fluyen hacia el sistema. En otros casos, las funciones efectúan algún grado de control sobre el procesamiento interno del software o elementos externos del sistema. Las funciones se pueden describir en muchos grados diferentes de abstracción, que van desde un enunciado general del propósito hasta una descripción detallada de los elementos del procesamiento que deben utilizarse.

Principio #3: ***Se debe representar el comportamiento del software (como una consecuencia de eventos externos).*** Al comportamiento del software de computadora lo guia su interacción con el ambiente externo. La entrada que proporcionan

los usuarios finales, los datos de control que aporta un sistema externo o los datos de monitoreo que se recolectan a través de una red ocasionan que el software se comporte de una manera específica.

Principio #4: ***Los modelos que presentan información, función y comportamiento deben partirse de forma que descubran el detalle de una manera estratificada (o jerárquica).*** El modelado del análisis es el primer paso en la resolución de problemas en la ingeniería del software. Esto permite al profesional entender mejor el problema y establecer una base para la solución (diseño). Los problemas complejos son difíciles de resolver por completo. Por esta razón, se utiliza una estrategia de "divide y ganarás". Un problema grande y complejo se divide en sub problemas hasta que cada subproblema es relativamente fácil de entender. Este concepto se llama *partición*, y es una estrategia clave en el modelado del análisis.

Principio #5: ***La tarea del análisis debe moverse de la información esencial hacia el detalle de implementación.*** El modelado del análisis comienza con la descripción del problema desde la perspectiva del usuario final. La "esencia" del problema se describe sin ninguna consideración de la forma en la que se implementará la solución. Por ejemplo, un videojuego requiere que el jugador "instruya" al protagonista en qué dirección seguir cuando éste se mueve dentro de un laberinto peligroso. Esa es la esencia del problema. El detalle de implementación (descrita en forma normal como una parte del modelo del diseño) indica cómo se implementará la esencia. Respecto del videojuego se podría aplicar la entrada de voz. De manera alternativa, se podría digitar un comando del teclado, o se podría apuntar un *joystick* (o un *mouse*) en una dirección específica.

CONJUNTO DE TAREAS

Conjunto de tareas genéricas para el modelado del análisis

1. Revisar los requisitos del negocio, las características/necesidades del usuario, las salidas visibles para el usuario, las restricciones del negocio, y otros requisitos técnicos que se hayan determinado durante las actividades de comunicación con el cliente y de planeación
2. Expandir y refinar los escenarios del usuario (sección 8.5).
   - Definir a todos los actores
   - Representar la forma en que los actores interactúan con el software.
   - Extraer funciones y características a partir de los escenarios del usuario.
   - Revisar los escenarios del usuario para verificar que estén completos y su exactitud (sección 26.4)
3. Modelar el dominio de la información (sección 8.3).
   - Representar todos los objetos importantes de información.
   - Definir los atributos para cada objeto de información.
   - Representar las relaciones entre los objetos de información.
4. Modelar el dominio funcional (sección 8.6).
   - Mostrar la forma en que las funciones modifican los objetos de datos.
   - Refinar las funciones para proporcionar los detalles de la elaboración
   - Escribir una narración del procesamiento que describa cada función y subfunción.
   - Revisar los modelos funcionales (sección 26.4).

5. Modelar el dominio del comportamiento (sección 8.8).
   - Identificar los eventos externos que ocasionan cambios en el comportamiento dentro del sistema
   - Identificar los estados que representan cada forma de comportamiento observable desde el exterior.
   - Presentar el modo en el que un evento lleva al sistema a cambiar de un estado a otro.
   - Revisar los modelos de comportamiento (sección 26.4).
6. Analizar y modelar la interfase del usuario (capítulo 12).
   - Dirigir el análisis de tareas
   - Crear prototipos de la imagen en pantalla,
7. Revisar todos los modelos en cuanto a que estén completos, su consistencia y las omisiones.

5.4.2 Principios de modelado del diseño

El modelo de diseño del software es el equivalente al plano de una casa para un arquitecto. Comienza con la representación de la totalidad del objeto que será construido (por ejemplo, una reproducción tridimensional de la casa) y con lentitud lo refina para proporcionar una guía para construir cada detalle (por ejemplo, el diseño de la plomería). En forma similar, el modelo del diseño que se crea para el software proporciona una variedad de diferentes vistas del sistema.

> "Primero ve que el diseño sea sabio y justo: averiguado esto, persíguelo resueltamente; no por un rechazo dejes ir el propósito que te has resuelto efectuar."
>
> William Shakespeare

No hay pocos métodos para derivar los diferentes elementos de un diseño de software. Algunos métodos se guían mediante los datos al permitir a la estructura de datos dictar la arquitectura del programa y los componentes de procesamiento resultantes. A otros los conduce el patrón al utilizar información acerca del dominio del problema (el modelo de análisis) para desarrollar estilos arquitectónicos y patrones de procesamiento. Incluso otros están orientados a objetos, al usar los objetos del dominio del problema como los conductores para la creación de estructuras de datos y los métodos para manipularlos. Aún así, todos ellos siguen un conjunto de principios de diseño que se pueden aplicar sin importar el método que se utilice:

Principio #1: ***El diseño debe ser rastreable hasta el modelo de análisis.*** El modelo de análisis describe el dominio de la información del problema, las funciones visibles para el usuario, el comportamiento del sistema y un conjunto de clases de análisis que empaqueta los objetos del negocio con los métodos que les sirven. El modelo de diseño traduce esta información a una arquitectura: un conjunto de subsistemas que implementan las funciones más importantes y un conjunto de diseños al nivel de componentes que son la realización de las clases de análisis. Excepto el modelo asociado con la infraestructura de software, los elementos del modelo de diseño deben ser rastreables hasta el modelo de análisis.

Referencia Web
En cs.wwwc.edu/~cahyun/Design/ se pueden encontrar comentarios profundos sobre el proceso de diseño, junto con una exposición de la estética del diseño.

Principio #2: ***Siempre se debe considerar la arquitectura del sistema que se va a construir.*** La arquitectura del software (capítulo 10) es el esqueleto del sistema que se va a construir. Éste afecta las interfases, las estructuras de datos, el flujo y el comportamiento del control del programa, la manera en que se pueden realizar las pruebas, la facilidad de mantenimiento del sistema resultante, y mucho más. Por todas estas razones, el diseño debe iniciarse con las consideraciones del diseño arquitectónico. Sólo después de que se ha establecido la arquitectura, es posible considerar los aspectos al nivel de componentes.

Principio #3: ***El diseño de datos es tan importante como el diseño de funciones de procesamiento.*** El diseño de datos es un elemento esencial del diseño arquitectónico. La manera en que se realizan los objetos de los datos dentro del diseño no puede dejarse a la suerte. Un diseño de datos bien estructurado ayuda a simplificar el flujo del programa, facilita el diseño y la implementación de los componentes del software, y confiere más eficiencia al procesamiento en general.

Principio #4: ***Las interfaces (internas y externas) deben diseñarse con cuidado.*** La manera en que fluyen los datos entre los componentes de un sistema tiene mucho que ver con la eficiencia del procesamiento, la propagación del error y la simplicidad del diseño. Una interfaz bien diseñada facilita la integración y ayuda a quien realiza la prueba a validar funciones de componentes.

Principio #5: ***El diseño de interfaz del usuario debe ajustarse a las necesidades del usuario final.*** *Sin embargo, en cada caso, debe resaltarse la facilidad del uso.* La interfaz del usuario es la manifestación visible del software. Sin importar qué tan sofisticadas sean sus funciones internas, sin importar qué tan comprensibles sean las estructuras de datos, no importa qué tan bien diseñada esté su arquitectura, un diseño de interfaz pobre siempre conduce a la percepción de que el software está "mal" hecho.

Principio #6: ***El diseño al nivel de componentes debe ser independiente del modo funcional.*** La independencia funcional es una medida del "significado sencillo" de un componente de software. La funcionalidad que entrega un componente debe ser *cohesiva*; es decir, debe centrarse en una y sólo una función o subfunción.[8]

Principio #7: ***Los componentes deben estar apareados entre sí en forma mínima y vinculados con el ambiente externo.*** El apareamiento se consigue de muchas maneras: vía interfaz de componente, por mensajes, a través de datos globales. A medida que aumenta el nivel de apareamiento, la probabilidad de propagación del error también aumenta y la facilidad de mantenimiento general del software disminuye. Por lo tanto, el apareamiento de componentes debe mantenerse tan bajo como sea posible.

8 En el capítulo 9 se puede encontrar una exposición adicional acerca de la cohesión.

Principio #8: ***Las representaciones del diseño (modelos) deben ser fácilmente comprensibles.*** El propósito del diseño es comunicar información a los profesionales que generarán códigos, a aquellos que probarán el software, y a quienes tal vez mantengan el software en lo futuro. Si el diseño es difícil de entender, no servirá como un medio efectivo de comunicación.

Principio #9: ***El diseño debe desarrollarse de manera iterativa. En cada iteración el diseñador debe buscar la mayor simplicidad.*** Como casi todas las actividades creativas, el diseño ocurre de modo iterativo. Las primeras iteraciones sirven para refinar el diseño y corregir errores, pero las iteraciones posteriores deben buscar que el diseño sea tan simple como sea posible

Cuando se aplican estos principios de manera apropiada, el ingeniero de software crea un diseño que muestra los factores internos y externos de calidad. *Los factores de calidad externos* son aquellas propiedades del software que los usuarios pueden observar fácilmente (como velocidad, confiabilidad, corrección, facilidad de uso). Los *factores de calidad internos* son importantes para los ingenieros de software, ya que conducen hacia un diseño de alta calidad desde una perspectiva técnica. Lograr factores de calidad internos requiere que el diseñador entienda conceptos básicos de diseño (capítulo 9).

INFORMACIÓN

Modelado ágil

En su libro sobre modelado ágil, Scott Ambler [AMB02] define una serie de principios[9] cuando el análisis y el diseño se conducen dentro del contexto de la filosofía del desarrollo ágil de software (capítulo 4):

Principio #1: La meta primaria del equipo de software es construir software, no crear modelos.

Principio # 2: Viajar ligero; es decir, no deben crearse más modelos de los necesarios

Principio # 3: Intentar producir el modelo más simple que describirá el problema o el software.

Principio # 4: Construir modelos de forma que éstos sean ajustables al cambio.

Principio # 5: Ser capaz de enunciar un propósito explícito para cada modelo que se cree

Principio # 6: Adaptar los modelos desarrollados al sistema que se tiene en mano

Principio # 7: Tratar de construir modelos útiles, pero olvidarse de construir modelos perfectos.

Principio # 8: No volverse dogmático acerca de la sintaxis del modelo. Si éste comunica su contenido de manera exitosa, la representación es secundaria

Principio # 9: Si el instinto indica que un modelo no es el correcto aunque éste luzca bien en el papel, probablemente existe una razón para estar preocupados.

Principio # 10: Obtener retroalimentación tan pronto como sea posible.

Sin importar el modelo de proceso que se elija o las prácticas específicas de la ingeniería del software que se apliquen, todos los equipos de software quieren ser ágiles. Por lo tanto, estos principios se deben aplicar sin importar el modelo de proceso del software que se haya seleccionado.

9 Los principios mencionados en esta sección se han abreviado y adaptado para los propósitos de este libro

CONJUNTO DE TAREAS

Conjunto de tareas genéricas para el diseño

1. Utilizar el modelo de análisis, seleccionar un estilo arquitectónico (patrón) apropiado para el software (capítulo 10).
2. Dividir el modelo de análisis en subsistemas de diseño y ubicar éstos dentro de la arquitectura (capítulo 10).
   - Tener la certeza de que cada subsistema es coherente en el sentido funcional.
   - Diseñar interfases de subsistema.
   - Ubicar las clases o funciones de análisis para cada subsistema.
   - Mediante la utilización del modelo del dominio de la información, diseñar estructuras de datos apropiadas.
3. Diseñar la interfaz del usuario (capítulo 12).
   - Revisar los resultados del análisis de tareas.
   - Especificar la secuencia de acción con base en los escenarios del usuario.
   - Crear un modelo de comportamiento de la interfaz.
   - Definir los objetos de la interfaz y mecanismos de control.
   - Revisar el diseño de la interfaz y ajustarlo como sea necesario (sección 26.4).
4. Conducir el diseño al nivel de componente.
   - Especificar todos los algoritmos en un grado relativamente bajo de abstracción.
   - Refinar la interfaz de cada componente.
   - Definir las estructuras de datos en el nivel de componente (sección 26.4).
   - Revisar el diseño en el nivel de componentes (sección 26.4).
5. Desarrollar un modelo de despliegue (sección 9.4.5).

5.5 PRÁCTICA DE LA CONSTRUCCIÓN

La *actividad de construcción* abarca una serie de tareas de codificación y realización de pruebas que conducen al software operativo que está listo para entregarlo al cliente o usuario final. En el trabajo de la ingeniería del software moderna la codificación puede ser: 1) la creación directa de código fuente de un lenguaje de programación, 2) la generación automática de código fuente al utilizar una representación intermedia del diseño del componente que será construido; 3) la generación automática de código mediante un lenguaje de programación de cuarta generación (por ejemplo, Visual C++).

> "Durante gran parte de mi vida he sido un lisgón del software, asomándome furtivamente en el código sucio de otras personas. Ocasionalmente, encuentro una joya real, un programa bien estructurado escrito con un estilo consistente, sin enredos, desarrollado de forma que cada componente es simple y organizado, y diseñado para que el producto pueda cambiar con facilidad."
>
> **David Parnas**

El enfoque inicial de las pruebas está al nivel de componentes, con frecuencia llamadas *pruebas de unidad*. Los otros niveles de prueba incluyen: 1) *pruebas de integración* (realizadas mientras el sistema está en construcción) 2) *pruebas de validación*, las cuales evalúan si los requisitos han sido satisfechos para el sistema completo (o para el incremento de software); y 3) *pruebas de aceptación*, que realiza el cliente en un esfuerzo encaminado a ejercitar las características y funciones.

Existe una serie de principios y conceptos aplicables a la codificación y las pruebas. Éstos se presentan en las secciones siguientes.

5.5.1 Principios y conceptos de codificación

Los principios y conceptos que guían la tarea de codificación están alineados de manera muy cercana al estilo de la programación, los lenguajes de la programación y los métodos de programación. Sin embargo, existe un conjunto de principios fundamentales que pueden establecerse:

Principios de preparación: *Antes de escribir una línea de código se debe estar seguro de:*

1. Entender el problema que se intenta resolver.
2. Entender los principios y conceptos básicos del diseño.
3. Escoger un lenguaje de programación que satisfaga las necesidades del software que se va a construir y el ambiente en el que éste va a operar
4. Seleccionar un ambiente de programación que proporcione herramientas que faciliten el trabajo
5. Crear un conjunto de pruebas de unidad que serán aplicadas una vez que se complete el componente que se va a codificar.

Principios de codificación: *Cuando se comience a escribir el código se debe estar seguro de:*

1. Restringir los algoritmos al seguir la práctica de la programación estructurada [BOH00]
2. Seleccionar las estructuras de datos que satisfarán las necesidades del diseño
3. Entender la arquitectura del software y crear interfases que sean consistentes con ella.
4. Mantener la lógica condicional tan simple como sea posible
5. Crear ciclos anidados en una forma que los haga fáciles de probar.
6. Seleccionar nombres de variables significativas y seguir otros estándares locales de codificación
7. Escribir código que tenga documentación propia.
8. Crear una configuración lineal (por ejemplo, sangrías y líneas en blanco que ayuden a la comprensión del código).

Principios de validación: *Después de haber completado los primeros pases de codificación, se debe estar seguro de:*

1. Conducir un ensayo de código cuando sea apropiado
2. Realizar pruebas de unidad y corregir los errores que se hayan descubierto.
3. Refabricar el código

Los libros sobre codificación y los principios que la guían incluyen trabajos recientes sobre el estilo de programación [KER78], construcción práctica de software [MCC93], perlas de programación [BEN99], el arte de la programación [KNU99], aspectos de la programación pragmática [HUN99], y muchos otros.

CONJUNTO DE TAREAS

Conjunto de tareas genéricas para la construcción

1. Construir la infraestructura arquitectónica (capítulo 10).
   - Revisar el diseño arquitectónico.
   - Codificar y probar los componentes que forman la infraestructura arquitectónica
   - Adquirir patrones arquitectónicos reutilizables
   - Probar la infraestructura para asegurar la integridad de la interfaz
2. Construir un componente del software (capítulo 11)
   - Revisar el diseño al nivel de componente.
   - Crear una serie de pruebas de unidad para el componente (secciones 13.3.1 y 14.7).
   - Codificar las estructuras de datos y la interfase del componente
   - Codificar los algoritmos internos y las funciones de procesamiento relacionadas.
   - Revisar el código conforme éste se escribe (sección 26.4).
   - Buscar la exactitud.
   - Asegurarse de que se han mantenido los estándares de codificación
   - Asegurarse de que el código se documenta a sí mismo
3. Realizar pruebas de unidad al componente.
   - Dirigir todas las pruebas de unidad.
   - Corregir los errores descubiertos.
   - Aplicar de nuevo las pruebas de unidad.
4. Integrar el componente terminado a la infraestructura arquitectónica

5.5.2 Principios de las pruebas

En un libro clásico sobre las *pruebas* realizadas al software, Glen Myers [MYE79] establece una serie de reglas que bien pueden servir como objetivos de las pruebas:

¿Cuáles son los objetivos de las pruebas al software?

- Las pruebas consisten en un proceso en el que se ejecuta un programa con la intención de encontrar un error que aún no se descubre.
- Un buen caso de prueba es aquel en el que hay una gran probabilidad de encontrar un error que aún no se descubre.
- Una prueba exitosa es aquella que encuentra un error que aún no se descubría.

Estos objetivos implican un cambio radical desde el punto de vista de algunos desarrolladores de software. Éstos se oponen a la visión inusual de que la prueba exitosa es aquella en la que no se encuentran errores. El objetivo aquí es diseñar pruebas que de manera sistemática descubran diferentes clases de errores y que lo hagan con un gasto mínimo de tiempo y esfuerzo.

Davis [DAV95] sugiere un conjunto de principios para las pruebas,[10] el cual se ha adaptado para aprovecharlo en este libro:

10 Aquí se presenta sólo un pequeño subconjunto de los principios de Davis para las pruebas. Para obtener más información véase [DAV95]

Principio #1: ***Todas las pruebas deben ser rastreables hasta los requisitos del cliente.*** El objetivo de las pruebas realizadas al software es descubrir errores. De aquí se desprende que los defectos más severos (desde el punto de vista del cliente) son aquellos que hacen fallar el programa al tratar de satisfacer sus requisitos.

Principio #2: ***Las pruebas se deben planear mucho antes de que comience el proceso de prueba.*** La planeación de las pruebas (capítulo 13) puede comenzar tan pronto como el modelo de análisis esté completo. La definición detallada de los casos de prueba puede comenzar en cuanto el modelo de diseño haya sido solidificado. Por tanto, todas las pruebas se pueden planear y diseñar antes de que se haya generado cualquier código.

Principio #3: ***El principio de Pareto es aplicable para las pruebas de software.*** Para establecerlo de manera simple, el principio de Pareto implica que 80 por ciento de los errores descubiertos durante las pruebas con probabilidad serán rastreables hasta 20 por ciento de todos los programas. El problema, por supuesto, es aislar estos componentes sospechosos y después probarlos.

Principio #4: ***Las pruebas deben comenzar "en lo pequeño" y progresar hacia "lo grande".*** Las primeras pruebas que se planean y ejecutan, por lo general, se enfocan en los componentes individuales. Conforme progresan las pruebas, el enfoque cambia en un intento de encontrar errores en grupos integrados de componentes y, por último, en el sistema completo.

Principio #5: ***Las pruebas exhaustivas no son posibles.*** El número de permutaciones entre las rutas, incluso de un programa con un tamaño moderado, es excepcionalmente grande. Por esta razón es imposible ejecutar cada combinación de rutas para las pruebas. Sin embargo, se puede cubrir de manera adecuada la lógica del programa para asegurar que se hayan ejercitado todas las condiciones al nivel de componentes (capítulo 14).

CONSEJO

...amplio ...de ...debe ...que se ...'en lo grande', ...en la ...del ...y se terminó ...o pequeño', al ...en los ... Para las ...sólo es ...y el ...y probar ...

CONJUNTO DE TAREAS

Conjunto de tareas genéricas para las pruebas

1. Diseñar pruebas de unidad para cada componente del software (sección 13.3.1)
   - Revisar cada prueba de unidad para asegurar una cobertura apropiada
   - Dirigir la prueba de unidad.
   - Corregir los errores descubiertos.
   - Aplicar de nueva las pruebas de unidad
2. Desarrollar una estrategia de integración (sección 13.3.2).
   - Establecer el orden y la estrategia que se usará para la integración
   - Definir las "construcciones" y las pruebas requeridas para ejercitarlas

11 Este principio se refiere a las pruebas *funcionales*, es decir, a las que se enfocan en los requisitos. Las pruebas *estructurales* (que se enfocan en los detalles arquitectónicos o lógicos) pueden no referirse en forma directa a los requisitos específicos.

Dirigir pruebas de humo a diario

Dirigir pruebas de regresión cada vez que sea necesario.

3. Desarrollar una estrategia de validación (sección 13.5)

Establecer los criterios de validación

Definir las pruebas requeridas para validar el software.

4. Dirigir las pruebas de integración y validación.

Corregir los errores descubiertos

Aplicar de nuevo las pruebas cada vez que sea necesario.

5 Dirigir las pruebas con mucho orden.

Realizar las pruebas de recuperación (sección 13.6.1).

Realizar las pruebas de seguridad (sección 13.6.2).

Realizar las pruebas de tensión (sección 13.6.3).

Realizar las pruebas de desempeño (sección 13.6.4).

6. Coordinar con el cliente las pruebas de aceptación (sección 13.5.3).

5.6 DESPLIEGUE

Como se mencionó en el capítulo 2, la actividad de despliegue abarca tres acciones entrega, soporte y retroalimentación Como el software moderno es evolutivo por naturaleza, el despliegue no se presenta una sola vez, sino varias veces conforme el software avanza hacia su terminación. Cada ciclo de entrega le proporciona al cliente y a los usuarios finales un incremento de software operativo que provee funciones y características útiles. Cada ciclo de soporte proporciona documentación y asistencia humana para todas las funciones y características introducidas durante todos los ciclos de despliegue que se han presentado hasta la fecha. Cada ciclo de retroalimentación ofrece al equipo de software una guía importante que conduce a modificaciones en las funciones, características y el enfoque que se toma para el siguiente incremento.

La entrega de un incremento de software representa un fundamento importante para cualquier proyecto de software. Cuando el equipo se prepara para crear un incremento, se debe seguir una serie de principios clave:

Principio #1: ***Se deben administrar las expectativas que el cliente tiene del software.*** Con demasiada frecuencia, el cliente espera más de lo que el equipo ha prometido entregar y de inmediato se presenta un desacuerdo. Esto genera una retroalimentación improductiva que arruina la moral del equipo. En su libro sobre la administración de expectativas, Naomi Kartun [KAR94] establece: "El punto inicial para administrar las expectativas es volverse más consciente acerca de lo que se comunica y de la forma en que se hace". Sugiere que un ingeniero de software debe ser cuidadoso de no enviar al cliente mensajes conflictivos (como prometer más de lo que se puede entregar de manera razonable en el marco de tiempo con el que se cuenta, o entregar más de lo que se promete para un incremento de software y después menos de lo prometido para el siguiente).

CONSEJO

Se debe tener la seguridad de que el cliente sabe qué esperar antes de que se entregue el incremento de software. De otra manera, el cliente esperará más de lo que se le puede entregar.

Principio #2: ***Se debe ensamblar y probar un paquete de entrega completo.*** Se debe ensamblar un CD-ROM u otro medio que contenga todo el software ejecutable, archivos con los datos de soporte, documentos de soporte y otra información relevante para que después lo prueben los usuarios reales. Todos los protocolos de instalación y otras características operativas se deben ejercitar posteriormente en todas las configuraciones de cómputo posibles (por ejemplo, hardware, sistemas operativos, dispositivos periféricos, arreglos de red).

Principio #3: ***Se debe establecer un régimen de soporte antes de entregar el software.*** Un usuario final espera responsabilidad e información exacta cuando surja una pregunta o problema. Si el soporte es *ad hoc* o, aún peor, inexistente, el cliente se siente insatisfecho de inmediato. El soporte debe ser planeado, el material de soporte se debe preparar y se deben establecer mecanismos para mantener un registro apropiado con que el equipo de software pueda realizar una evaluación categórica de los tipos de soporte solicitados.

Principio #4: ***Se debe proporcionar material instructivo apropiado a los usuarios finales.*** El equipo de software entrega más que el software en sí; en caso de ser necesario, se debe desarrollar un entrenamiento apropiado, se deben proporcionar directrices para la resolución de problemas, y se deben publicar descripciones acerca de "cuál es la diferencia con este incremento de software".[12]

Principio #5: ***El software con errores se debe arreglar primero y entregarse después.*** Ante la presión del tiempo, algunas organizaciones de software entregan incrementos de baja calidad con una advertencia para el cliente de que los errores "se eliminarán en la próxima versión". Esto es un error. En el negocio del software se dice: "Los clientes olvidarán que se les entregó un producto de alta calidad unos pocos días después, pero nunca olvidarán los problemas que les causó un producto de baja calidad. El software se los recuerda todos los días".

El software entregado proporciona un beneficio para el usuario final, pero también provee una *retroalimentación* útil para el equipo de software. Al utilizar el incremento, los usuarios finales deben ser motivados a comentar sobre las características y funciones, facilidad de uso, confiabilidad y cualesquiera otras características apropiadas. La retroalimentación debe recopilarla y registrarla el equipo de software y aprovecharla para 1) hacer modificaciones inmediatas al incremento entregado (si es necesario); 2) definir los cambios que serán incorporados en el próximo incremento planeado; 3) realizar las modificaciones necesarias al diseño para ajustarlo a los cambios; y 4) revisar el plan (incluyendo el calendario de entrega) del próximo incremento para reflejar los cambios.

12 Durante la actividad de comunicación el equipo de software debe determinar los tipos de materiales de ayuda que quieren los usuarios.

CONJUNTO DE TAREAS

Conjunto de tareas genéricas para el despliegue

1 Crear medios de entrega
- Ensamblar y probar todos los archivos ejecutables
- Ensamblar y probar todos los archivos de datos
- Crear y probar toda la documentación del usuario
- Implementar las versiones electrónicas (por ejemplo, pdf).
- Implementar archivos de "ayuda" con hipertexto.
- Implementar una guía para la resolución de problemas
- Probar los medios de entrega con un grupo pequeño de usuarios representativos.

2. Establecer la persona o grupo encargado del soporte humano
- Crear la documentación o las herramientas de soporte por computadora
- Establecer mecanismos de contacto (por ejemplo, un sitio web, teléfono, correo electrónico)
- Establecer mecanismos para la localización de problemas.
- Establecer mecanismos para el reporte de problemas.
- Establecer una base de datos para el reporte de problemas/errores.

3. Establecer mecanismos de retroalimentación del usuario.
- Definir el proceso de retroalimentación.
- Definir las formas de retroalimentación (en papel o electrónica)
- Establecer la base de datos de retroalimentación.
- Definir el proceso de evaluación de la retroalimentación.

4. Diseminar los medios de entrega a todos los usuarios.
5. Dirigir las funciones de soporte continuas.
- Proporcionar asistencia en la instalación y el inicio.
- Proporcionar asistencia continua y de resolución de problemas.

6. Recopilar la retroalimentación del usuario
- Registrar la retroalimentación.
- Evaluar la retroalimentación.
- Comunicarse con los usuarios sobre la retroalimentación

5.7 RESUMEN

La práctica de la ingeniería del software incluye conceptos, principios, métodos y herramientas que aplican los ingenieros de software durante el proceso de software. Cada proyecto de ingeniería del software es diferente, aun así existe un conjunto de principios y tareas aplicables para cada actividad del marco de trabajo del proceso, sin importar el proyecto o producto.

Si se pretende dirigir una buena práctica de la ingeniería del software, es necesario un conjunto de puntos esenciales técnicos y de gestión. Los puntos técnicos incluyen la necesidad de entender los requisitos y las áreas de incertidumbre del prototipo, así como la necesidad de definir de manera explícita la arquitectura del software y el plan de integración de componentes. Los puntos esenciales de gestión incluyen la necesidad de definir prioridades y especificar un calendario realista que las refleje, la necesidad de precisar medidas de control del proyecto apropiadas para la calidad y el cambio.

Los principios de comunicación con el cliente se enfocan en la necesidad de reducir el ruido y mejorar el canal de comunicación conforme progresa la conversación entre el desarrollador y el cliente. Ambas partes deben colaborar para que se establezca la mejor comunicación.

Los principios de planeación se enfocan en las directrices encaminadas a construir el mejor mapa para realizar el trabajo que conducirá a terminar un sistema o producto. El plan se puede diseñar sólo para un incremento de software, o se puede definir respecto del proyecto completo. Independientemente de ello, el plan debe indicar qué se hará, quién lo hará y cuándo se completará el trabajo.

El modelado incluye tanto el análisis como el diseño, al describir representaciones del software que se vuelven más detalladas de manera progresiva. La finalidad de los modelos es solidificar la comprensión del trabajo que se realizará y proporcionar una guía técnica para quienes implementarán el software.

La construcción incorpora un ciclo de codificación y pruebas en el cual primero se genera el código fuente y después éste se prueba para detectar errores. La integración combina los componentes individuales e involucra una serie de pruebas que se enfocan en los aspectos del funcionamiento general y de las interfases locales. Los principios de codificación definen las acciones genéricas que deben ocurrir antes de que se escriba el código, mientras éste se crea y después de que se haya completado. Aunque existen muchos principios para las pruebas, sólo uno es el dominante: las pruebas se forman con un proceso en el que un programa se ejecuta con el objetivo de encontrar un error.

Durante el desarrollo del software evolutivo se presenta el desarrollo para cada incremento de software que se presenta al cliente. Los principios clave para la entrega consideran administrar las expectativas del cliente y dotar al usuario con información de soporte para el software. El soporte necesita una preparación previa. La retroalimentación permite al cliente sugerir cambios que tienen un valor de negocios y proporcionan al desarrollador una entrada para el próximo ciclo iterativo de la ingeniería del software.

Referencias

[AMB02] Ambler, S. y R. Jeffries, *Agile Modeling*, Wiley, 2002.

[BEN99] Bentley, J., *Programming Pearls*, 2a. ed., Addison-Wesley, 1999.

[BOE 96] Boehm, B., "Anchoring the Software Process", en *IEEE Software*, vol. 13, núm. 4, julio de 1996, pp. 73-82.

[BOH00] Bohl, M. y M. Rin, *Tools for Structured Design: An Introduction to Programming Logic*, 5a. ed., Prentice-Hall, 2000.

[DAV95] Davis, A., *201 Principles of Software Development*, McGraw-Hill, 1995.

[FOW99] Fowler, M. *et al.*, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[GAR95] Garlan, D. y M. Shaw, "An Introduction to Software Architecture", en *Advances in Software Engineering and Knowledge Engineering*, vol. I (V. Ambriola y G. Tortora, eds.), World Scientific Publishing Company, 1995.

[HIG00] Highsmith, J., *Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems*, Dorset House Publishing, 2000.

[HOO96] Hooker, D., "Seven Principles of Software Development", septiembre de 1996, disponible en http://c2.com/cgi/wikiSevenPrinciplesOfSoftwareDevelopment.
[HUN95] Hunt, D., A. Bailey y B. Taylor, *The Art of Facilitation*, Perseus Book Group, 1995
[HUN99] Hunt A., D. Thomas y W. Cunningham, *The Pragmatic Programmer*, Addison-Wesley, 1999
[JUS99] Justice, T. *et al.*, *The Facilitator's Fieldbook*, AMACOM, 1999
[KAN93] Kanner, C., J. Falk y H. Q. Nguyen, *Testing Computer Software*, 2a. ed., Van Nostrand Reinhold, 1993
[KAN96] Kaner, S. *et al.*, *The Facilitator's Guide to Preparatory Decision Making*, New Society Publishing, 1996.
[KAR94] Karten, N., *Managing Expectations*, Dorset House, 1994
[KER78] Kernighan, B. y P. Plauger, *The Elements of Programming Style*, 2a. ed., McGraw-Hill, 1978.
[KNU98] Knuth, D., *The Art of Computer Programming*, 3 volúmenes, Addison-Wesley, 1998.
[MCC93] McConnnell, S., *Code Complete*, Microsoft Press, 1993.
[MCC97] McConnell, S., "Software's Ten Essentials", en *IEEE Software*, vol. 14, núm. 2, marzo-abril, 1997, pp. 143-144.
[MYE78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978
[MYE79] Myers, G., *The Art of Software Testing*, Wiley, 1979.
[PAR72] Parnas, D.L., "On Criteria to Be Used in Decomposing Systems into Modules", en *CACM*, vol. 14, núm. 1, abril de 1972, pp. 221-227.
[POL45] Polya, G., *How to Solve It*, Princeton University Press, 1945.
[ROS75] Ross, D., J. Goodenough y C. Irvine, "Software Engineering: Process, Principles and Goals", en *IEEE Computer*, vol. 8, núm. 5, mayo de 1975
[SHA95a] Shaw, M. y D. Garlan, "Formulations and Formalisms in Software Architecture", *Volume 1000—Lecture Notes in Computer Science*, Springer-Verlag, 1995.
[SHA95b] Shaw, M. *et al.*, "Abstractions for Software Architecture and Tools to Support Them", en *IEEE Trans. Software Engineering*, vol. SE-21, núm. 4, abril de 1995, pp. 314-335.
[STE74] Stevens, W., G. Myers y L. Constantine, "Structured Design", en *IBM Systems Journal*, vol. 13, núm. 2, 1974, pp. 115-139
[TAY90] Taylor D. A., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990
[ULL97] Ullman, E., *Close to the Machine. Technophilia and its Discontents*, City Lights Books, 1997.
[WIR71] Wirth, N., "Program Development by Stepwise Refinement", en *CACM*, vol. 14, núm. 4, 1971, pp. 221-227.
[WOO95] Wood, J. y D. Silver, *Joint Application Design*, Wiley, 1995
[ZAH90] Zahniser, R. A., "Building Software in Groups", en *American Programmer*, vol. 3, núms. 7-8, julio-agosto de 1990.

PROBLEMAS Y PUNTOS A CONSIDERAR

5.1. Inténtese resumir en un párrafo breve los "siete principios para el desarrollo de software" de David Hooker (sección 5.1). Trátese de extraer la esencia de su guía en sólo unas cuantas oraciones y sin usar las palabras de Hooker.

5.2. ¿Existen otros puntos técnicos "esenciales" que se puedan recomendar a los ingenieros de software? Enunciar cada uno de ellos y explicar por qué se han incluido.

5.3. ¿Existen otros puntos "esenciales" de gestión que se puedan recomendar a los ingenieros de software? Enunciar cada uno de ellos y explicar por qué se ha incluido.

5.4. Un principio importante de la comunicación establece "prepare antes de comunicar". ¿Cómo podría esta preparación manifestarse por sí misma en los primeros trabajos que se realizan? ¿Cuáles productos de trabajo podrían resultar como consecuencia de la preparación oportuna?

5.5. Hágase una investigación de la "facilitación" para la actividad de comunicación (utilícense las referencias que se proporcionan u otras) y prepárese un conjunto de directrices que se enfoquen sólo en la facilitación.

5.6. ¿En qué difieren la comunicación ágil y la comunicación de la ingeniería de software tradicional? ¿En qué son similares?

5.7. ¿Por qué es necesario continuar?

5.8. Realizar una investigación de la "negociación" para la actividad de comunicación y preparar un conjunto de directrices que se enfoquen sólo en la negociación

5.9. Describir lo que significa *granularidad* en el contexto de un calendario de proyecto.

5.10. ¿Por qué son importantes los modelos en el trabajo de la ingeniería de software? ¿Siempre son necesarios? ¿Son calificadores para su respuesta acerca de la necesidad?

5.11. ¿Cuáles son los tres "dominios" que se consideran durante el modelado del análisis?

5.12. Tratar de agregar un principio adicional a los enunciados para la codificación de la sección 5.6.

5.13. ¿Qué es una prueba exitosa?

5.14. ¿Se está de acuerdo con el siguiente enunciado?: "debido a que entregamos múltiples incrementos al cliente, no debemos preocuparnos por la calidad en los primeros incrementos; los problemas se pueden resolver en iteraciones posteriores. Explíquese la respuesta

5.15. ¿Por qué es importante la retroalimentación para el equipo de software?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La comunicación con el cliente es una actividad muy importante en la ingeniería del software; no obstante, algunos profesionales no dedican tiempo a leer acerca de ella. Los libros de Pardee (*To Satsfy and Delight Your Costumer*, Dorset House, 1996) y Karten [KAR94] proporcionan una gran perspectiva de los métodos para la interacción efectiva con el cliente. En muchos libros sobre la gestión de proyectos se consideran los conceptos y principios de la comunicación y la planeación. Las ofertas útiles relativas a la gestión de proyectos incluyen: Hughs y Cotterell (*Software Project Management*, segunda edición, McGraw-Hill, 1999), Phillips (*The Software Project Manager's Handbook*, IEEE Computer Society Press, 1998), McConnell (*Software Project Survival Guide*, Microsoft Press, 1998) y Gilb (*Principles of Software Engineering Management*, Addison-Wesley, 1998).

Casi cualquier libro sobre ingeniería del software contiene una exposición útil sobre los conceptos y principios para el análisis, el diseño y las pruebas. Entre las mejores ofertas están los libros de Endres y sus colegas (*Handbook of Software and Systems Engineering*, Addison-Wesley, 2003), Sommerville (*Software Engineering*, sexta edición, Addison-Wesley, 2000), Pfleeger (*Software Engineering: Theory and Practice*, Prentice-Hall, 2001) y Schach (*Object-Oriented and Classical Software Engineering*, McGraw-Hill, 2001). Davis ha recopilado una amplia colección de principios de software en [DAV95].

Los conceptos y principios del modelado se consideran en muchos libros dedicados al análisis de requisitos o al diseño de software. Young (*Effective Requirements Practices*, Addison-Wesley, 2001) resalta un "equipo conjunto" de clientes y desarrolladores que elaboren los requisitos colectivamente. Weigers (*Software Requirements*, Microsoft Press, 1999) presenta muchos requisitos clave de ingeniería y requisitos de las prácticas de gestión. Sommerville y Kotonya (*Requirements Engineering: Process and Techniques*, Wiley, 1998) analizan los conceptos y las técnicas de "obtención" y otros principios de la ingeniería de requisitos.

El libro de Norman (*The Design of Everyday Things*, Currency/Doubleday, 1990) es una lectura obligada para cualquier ingeniero de software que intente hacer el trabajo de diseño. Winograd y sus colegas (*Bringing Design to Software*, Addison-Wesley, 1996) han editado una excelente colección de ensayos que tratan sobre los aspectos prácticos del diseño de software. Constantine y Lockwood (*Software for Use*, Addison-Wesley, 1999) presentan los conceptos aso-

ciados con el "diseño centrado en el usuario". Tognazzini (*Tog on Software Design*, Addison-Wesley, 1995) presenta una valiosa exposición filosófica de la naturaleza del diseño y el impacto de las decisiones sobre la calidad y la capacidad del equipo para producir un software que proporcione un gran valor a su cliente.

Hay cientos de libros que tratan sobre uno o más elementos de la actividad de construcción Kernighan y Plauger [KER78] escribieron un texto clásico sobre el estilo de programación, McConnell [MCC93] presenta directrices pragmáticas para la construcción práctica de software; Bentley [BEN99] sugiere una amplia variedad de perlas de programación; Knuth [KNU98] ha escrito una serie clásica de tres volúmenes sobre el arte de la programación, y Hunt [HUN99] sugiere directrices pragmáticas de programación. La bibliografía sobre pruebas ha florecido en la década pasada. Myers [MYE79] se conserva como un clásico. Los libros de Whitaker (*How to Break Software*, Addison-Wesley, 2002), Kaner y sus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) y Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) presentan conceptos y principios importantes sobre las pruebas, así como una guía pragmática considerable.

En Internet existe una amplia variedad de fuentes de información sobre la práctica de la ingeniería del software. En el sitio web de SEPA se puede encontrar una lista actualizada de referencias en la red mundial, las cuales son relevantes para la práctica de la ingeniería de software:
http://www.mhhe.com/pressman.

CAPÍTULO

6

INGENIERÍA DE SISTEMAS

CONCEPTOS CLAVE

Hace casi 500 años, Maquiavelo dijo: "No hay nada más difícil de llevar a cabo, más peligroso de realizar o de éxito más incierto que encabezar la introducción de un nuevo orden de cosas". Durante los 50 últimos años, los sistemas basados en computadora han introducido un nuevo orden. Aunque la tecnología ha tenido varios avances desde la época de Maquiavelo, sus palabras aún siguen vigentes

La ingeniería del software ocurre como consecuencia de un proceso llamado *ingeniería de sistemas*. En lugar de concentrarse sólo en el software, esta disciplina se centra en una variedad de elementos mientras analiza, diseña y organiza aquellos elementos de un sistema que pueden ser un producto, un servicio o una tecnología para la transformación o control de información.

El proceso de ingeniería de sistemas asume distintas formas, según el dominio de aplicación en que se utilice. La *ingeniería de procesos de negocios* se aplica cuando el contexto del trabajo se enfoca en una empresa. Cuando se va a construir un producto (en este contexto un producto incluye todo, desde un teléfono inalámbrico hasta un sistema de control de tráfico aéreo), al proceso se le llama *ingeniería de producto*

Tanto la ingeniería de procesos de negocio como la de producto intentan poner orden en el desarrollo de sistemas basados en computadora. Aunque cada uno de ellos se utiliza en un dominio de aplicación diferente, ambos buscan poner al software en su contexto. Es decir, tanto la ingeniería de procesos de

UN VISTAZO RÁPIDO

¿Qué es? Antes de que sea posible construir el software, por medio de la ingeniería, se debe entender el "sistema" en que éste reside. Para lograrlo es necesario determinar el objetivo general del sistema; se debe identificar el papel que tienen el hardware, el software, las personas, las bases de datos, los procedimientos y otros elementos del sistema; y se deben identificar, analizar, especificar, modelar, validar y gestionar los requisitos operacionales. Estas actividades son el fundamento de la ingeniería de sistemas.

¿Quién lo hace? Un ingeniero de sistemas trabaja para entender los requisitos de un sistema al trabajar con el cliente, usuarios futuros y otros interesados

¿Por qué es importante? Existe un antiguo proverbio que dice: "los árboles no dejan ver el bosque". En este contexto, el "bosque" es el sistema y los árboles son los elementos tecnológicos (incluido el software) que se requieren para realizar el sistema. Si se construyen los elementos tecnológicos de una manera precipitada antes de entender el sistema, sin duda se cometerán errores que decepcionarán al cliente. Antes de preocuparse por los árboles se debe entender el bosque.

¿Cuáles son los pasos? Se identifican los objetivos y requisitos operacionales más detallados al obtener información del cliente; se analizan los requisitos para evaluar su claridad, si está completa y es consistente; se crea una especificación, que por lo general está incorporada a

un modelo de sistema, que después lo validan los participantes y clientes. Por último, se gestionan los requisitos del sistema para asegurar que los cambios se controlan de manera apropiada

¿Cuál es el producto obtenido? Se debe producir una representación efectiva del sistema, como consecuencia de la ingeniería del mismo. Se puede realizar a través de un prototipo, una especificación o incluso un modelo simbólico, pero debe comunicar las características operativas, funcionales y de comportamiento del sistema que se va a construir e incorporarlo dentro de la arquitectura del sistema

¿Cómo puedo estar seguro de que lo he hecho correctamente? Mediante una revisión de todos los productos de trabajo obtenidos, para verificar su claridad, si está completo y es consistente. Es importante mencionar que los cambios en los requisitos de un sistema deben gestionarse con métodos sólidos de GCS (capítulo 27).

negocios como la ingeniería de producto,[1] trabajan para asignar un papel al software de computadora y, al mismo tiempo, establecer los enlaces que unen al software con otros elementos de un sistema basado en computadora

Este capítulo se centrará en las necesidades de gestión y en las actividades específicas del proceso que permitan a la organización de software asegurarse de hacer las cosas correctas en el tiempo correcto y del modo correcto.

6.1 SISTEMAS BASADOS EN COMPUTADORA

La palabra *sistema* tal vez sea el término más usado y del que más se abusa en el léxico técnico. Se habla de sistemas políticos, sistemas educativos, de sistemas de aviación y sistemas de fabricación, de sistemas bancarios y sistemas de locomoción La palabra dice muy poco. El adjetivo se utiliza para describir el *sistema* y así entender el contexto en el que se usa la palabra. El diccionario *Webster* define sistema de la siguiente manera:

> 1 Un conjunto o disposición de cosas relacionadas que forman una unidad o un todo orgánico, 2. Una serie de hechos, principios, reglas, etcétera, clasificado y dispuestos de manera ordenada que muestran un plan lógico de la unión de las partes; 3. Un método o plan de clasificación o disposición; 4 Una manera establecida de hacer algo; método; procedimiento.

En el diccionario aparecen cinco definiciones más, pero no se sugiere un sinónimo preciso. *Sistema* es una palabra especial. Al retomar la definición del diccionario *Webster*, un sistema basado en computadora se define como:

> Un conjunto o disposición de elementos que están organizados para cumplir una meta predefinida al procesar información.

1 En realidad, el término *ingeniería de sistemas* se emplea con frecuencia en este contexto. Sin embargo, para los propósitos de este libro "ingeniería de sistemas" es genérico y abarca la ingeniería de procesos del negocio y la ingeniería de producto

Es posible que la meta sea apoyar una función de negocio o desarrollar un producto que pueda venderse para generar beneficios. Para cumplir la meta, un sistema basado en computadora emplea una variedad de elementos del sistema:

CONSEJO

...se debe caer en la ... de tener una ... "centrado en el ...". Para ... se deben ... todos los ... de un ... antes de ... en el ...

Software. Programas de computadora, estructuras de datos y documentación que sirven para hacer efectivo el método, procedimiento o control lógico que se requiere.

Hardware. Dispositivos electrónicos que proporcionan capacidad de cálculo, dispositivos de interconexión (por ejemplo, conmutadores de red, dispositivos de telecomunicación) que permiten el flujo de datos, y dispositivos electromecánicos (como sensores, motores, bombas) que proporcionan una función externa, del mundo real

Personas. Usuarios y operadores del hardware y software.

Bases de datos. Una extensa y organizada recopilación de información a la cual se tiene acceso a través de software y que persiste a través del tiempo.

Documentación. Información descriptiva (por ejemplo, modelos, especificaciones, manuales, archivos de ayuda en línea, sitios web) que detalla el uso y operación del sistema.

Procedimientos. Los pasos que definen el uso específico de cada elemento del sistema o el contexto de procedimiento en que reside el sistema

Estos elementos se combinan de varias maneras para transformar la información. Por ejemplo, un departamento de mercadotecnia transforma la información bruta de ventas en un perfil del comprador típico del producto; un robot transforma un archivo de órdenes que contiene instrucciones específicas en un conjunto de señales de control que provocan alguna acción física específica. La creación de un sistema de información para asesorar al departamento de mercadotecnia y un software de control para el robot requiere de la ingeniería de sistemas.

> "Nunca confíes en una computadora que no puedas lanzar por la ventana."
>
> **Steve Wozniak**

PUNTO CLAVE

... complejos ... realidad una ... de ... s que a ... sistemas.

Una característica complicada de los sistemas basados en computadora es que tal vez constituyen un macroelemento de un sistema aún mayor. El *macroelemento* es un sistema basado en computadora que es parte de un sistema mayor basado también en computadora. Por ejemplo, un *sistema de automatización de una fábrica* se considera una jerarquía de sistemas. En el nivel más bajo de la jerarquía se encuentra una máquina de control numérico, robots y dispositivos de entrada de información. Cada uno de éstos es un sistema basado en computadora por derecho propio Los elementos de la máquina de control numérico incluyen hardware electrónico y electromecánico (por ejemplo, procesador y memoria, motores, sensores), software (para comunicaciones y control de la máquina), personas (el operador de la máquina), una base de datos (el programa de CN almacenado), documentación y procedi-

mientos. Podría aplicarse una descomposición similar al robot y al dispositivo de entrada de información. Todos son sistemas basados en computadora.

En el siguiente nivel de la jerarquía se define una *célula de fabricación*. Ésta es un sistema basado en computadora que puede tener elementos propios (por ejemplo, computadoras, instalaciones mecánicas), y también integra los macroelementos que se han denominado máquina de control numérico, robot y dispositivo de entrada de información.

En resumen, la célula de fabricación y sus macroelementos están compuestos de elementos del sistema con las etiquetas genéricas: software, hardware, personas, bases de datos, procedimientos y documentación. En algunos casos los macroelementos pueden compartir un elemento genérico. Por ejemplo, el robot y la máquina de CN podría operarlas el mismo operador (el elemento personas). En otros casos, los elementos genéricos son exclusivos de un sistema.

El papel del ingeniero de sistemas es definir los elementos de un sistema específico basado en computadora en el contexto de la jerarquía global de sistemas (macroelementos). En las secciones siguientes se examinan las tareas que constituyen la ingeniería de sistemas de computadoras.

6.2 LA JERARQUÍA DE LA INGENIERÍA DE SISTEMAS

Referencia Web

El Consejo Internacional de Ingeniería de Sistemas (INCOSE) proporciona varios buenos recursos útiles en www.incose.org.

Sin importar su dominio de enfoque, la ingeniería de sistemas abarca una serie de métodos para navegar de arriba hacia abajo y de abajo hacia arriba en la jerarquía ilustrada en la figura 6.1. El proceso de la ingeniería de sistemas por lo general comienza con una "visión global". Es decir, se examina el dominio entero del negocio o producto para asegurarse de que se puede establecer el contexto tecnológico o de negocios apropiado. La visión global es refinada para enfocarse totalmente en un dominio específico de interés. Dentro de un dominio especial se analiza la necesidad de elementos del sistema (por ejemplo, información, software, hardware, personas). Al final se inicia el análisis, diseño y construcción del elemento del sistema deseado. En la parte alta de la jerarquía se establece un contexto muy amplio, y en el de la parte baja se conducen actividades técnicas detalladas, realizadas por la disciplina de ingeniería correspondiente (por ejemplo, ingeniería de hardware o software).[2]

PUNTO CLAVE

El buen ingeniero de sistemas comienza con un entendimiento claro del contexto —la visión global— y después, de manera progresiva, el enfoque se delimita hasta la comprensión de los detalles técnicos.

Dicho de una manera un poco más formal, la *visión global* (VG) la compone un conjunto de dominios (D_i) en donde cada uno de ellos puede ser un sistema o un sistema de sistemas por derecho propio.

$$VG = \{D_1, D_2, D_3, \ldots, D_n\}$$

Cada dominio lo componen *elementos* (E_i) específicos, los cuales tienen un papel para cumplir el objetivo y las metas del dominio o componente:

2 Sin embargo, en algunas situaciones los ingenieros del sistema deben considerar primero los elementos individuales del sistema. Mediante el uso de este enfoque, los subsistemas se describen de abajo hacia arriba al considerar primero los componentes detallados que forman el subsistema.

$D_i = \{E_1, E_2, E_3, ..., E_m\}$

Por último, cada elemento se implementa al especificar los *componentes* (C_k) técnicos que logran la función necesaria para un elemento:

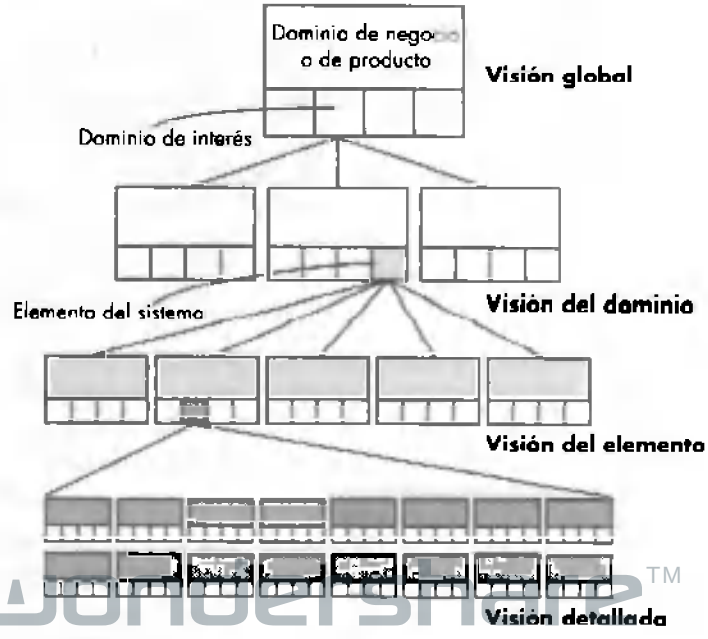
$E_j = \{C_1, C_2, C_3, ..., C_k\}$

En el contexto de software un componente podría ser un programa de computadora, un componente reutilizable de un programa, un módulo, una clase u objeto o incluso un enunciado en lenguaje de programación.

> "Siempre diseño las cosas considerándolas en su contexto inmediato superior: una silla en un cuarto, un cuarto en una casa, una casa en un vecindario, un vecindario en un plan urbano."
>
> **Eliel Saarinen**

Es importante notar que el ingeniero de sistemas estrecha más el enfoque de trabajo conforme avanza hacia abajo en la jerarquía descrita. Sin embargo, la visión global muestra una clara definición de la funcionalidad general que le permitirá al ingeniero entender el dominio y el sistema o producto en el contexto apropiado.

Figura 6.1 La jerarquía de la ingeniería de sistemas



6.2.1 Modelado del sistema

El *modelado de sistemas* es un elemento importante del proceso de ingeniería de sistemas. Sin importar que el enfoque esté en la visión global o en la visión detallada, el ingeniero crea modelos que [MOT92]:

¿Qué se logra con el modelo de la ingeniería de sistemas?

- Definen los procesos que satisfacen las necesidades de la visión que se considera
- Representen el comportamiento de los procesos y los supuestos en los que se basa el comportamiento.
- Definen de modo explícito las entradas exógenas[3] y endógenas de información al modelo.
- Representan todas las uniones (incluidas las salidas) que permiten al ingeniero entender mejor la visión.

Al construir un modelo del sistema el ingeniero debe considerar algunas restricciones

1. *Supuestos* que reducen el número de permutaciones y variaciones posibles, lo que permite al modelo reflejar el problema de una manera razonable. Por ejemplo, un producto de representación tridimensional que utiliza la industria del entretenimiento para crear animaciones realistas. Un dominio del producto permite la representación de formas humanas en tres dimensiones. Las entradas a este dominio comprenden la habilidad de adaptar movimiento de un actor humano vivo, de un video o de la creación de modelos gráficos. El ingeniero de sistemas hace ciertos supuestos sobre el intervalo de movimiento humano permitido (por ejemplo, las piernas no pueden enrollarse alrededor del torso) de modo que pueda limitarse el proceso y la gama de entradas.
2. *Simplificaciones* que permiten la creación del modelo a tiempo. Para ilustrarlo se puede considerar una compañía de productos de oficina que vende y suministra una amplia variedad de fotocopiadoras, escáneres y equipos similares El ingeniero de sistemas modela las necesidades de la organización suministradora y trabaja para entender el flujo de información que engendra una orden de suministro. Aunque una orden de suministro puede originarse desde muchas fuentes, el ingeniero toma en cuenta sólo dos de ellas: la demanda interna y la petición externa. Esto permite una partición simplificada de entradas que se requiere para generar la orden de suministro.

PUNTO CLAVE

Un ingeniero de sistemas considera los siguientes factores al determinar soluciones alternativas: supuestos, simplificaciones, limitaciones, restricciones y preferencias del cliente.

3. *Limitaciones* que ayudan a delimitar el sistema. Por ejemplo, se modela un sistema de aeronáutica para un avión de próxima generación. Como el avión tiene un diseño de dos motores, el dominio de monitoreo para la propulsión será modelado para acomodar un máximo de dos motores y sus numerosos sistemas asociados.
4. *Restricciones* que guían la manera de crear el modelo y tomar el enfoque al implementarlo. Por ejemplo, la infraestructura tecnológica para el sistema de representación tridimensional descrito antes utiliza procesadores duales basados en G5. La complejidad de cálculo de los problemas debe restringirse para encajar en los límites de proceso que imponen estos procesadores.

3 Las entradas *exógenas* unen un elemento de una visión dada con otros elementos en el mismo nivel o en otros niveles; las entradas *endógenas* unen componentes individuales de un elemento en una visión particular.

5. *Preferencias* que indican la arquitectura preferida para todos los datos, funciones y tecnología. La solución preferida a veces entra en conflicto con otros factores restrictivos. Sin embargo, la satisfacción del cliente muchas veces se toma en cuenta hasta el punto de realizar su enfoque preferido.

El modelo de sistema resultante (desde cualquier visión) puede reclamar una solución automática por completo, semiautomática o un enfoque manual. De hecho, con frecuencia es posible caracterizar modelos de cada tipo que sirvan como soluciones alternativas del problema que se tiene entre manos. En esencia, el ingeniero de sistemas tan sólo modifica la influencia relativa de diferentes elementos del sistema (personas, hardware, software) para crear modelos de cada tipo

> "Las cosas simples deben ser simples. Las cosas complejas deben ser posibles."
>
> **Alan Kay**

6.2.2 Simulación del sistema

Muchos sistemas basados en computadora interactúan con el mundo real en forma reactiva. Es decir, los eventos del mundo real los monitorean el hardware y el software que componen el sistema basado en computadora y, con base en estos eventos, el sistema impone control sobre las máquinas, los procesos e incluso sobre la gente que genera los eventos. Los sistemas de tiempo real y sistemas empotrados a menudo pertenecen a la categoría de sistemas reactivos.

Muchos sistemas de la categoría de los reactivos controlan máquinas o procesos (como aerolíneas comerciales o refinerías de petróleo) que deben operar con un grado muy alto de confiabilidad. Si el sistema falla podrían ocurrir pérdidas económicas o humanas significativas. Por esta razón, el modelado del sistema y las herramientas de simulación se utilizan para ayudar a eliminar sorpresas cuando se construyen sistemas reactivos basados en computadora. Estas herramientas se aplican durante el proceso de ingeniería de sistemas, cuando se especifica el papel del hardware, el software, las bases de datos y las personas. El modelado y las herramientas de simulación permiten al ingeniero de sistemas probar una especificación del sistema.

HERRAMIENTAS DE SOFTWARE

Herramientas de simulación del sistema

Objetivo: Las herramientas de simulación del sistema proporcionan al ingeniero de software la capacidad de predecir el comportamiento de un sistema de tiempo real antes de que éste se construya. Además, estas herramientas permiten al ingeniero de software desarrollar maquetas del sistema en tiempo real, lo que permite al cliente tener una visión del funcionamiento, operación y respuesta antes de la implementación real.

Mecánica: Las herramientas de esta categoría permiten al equipo definir los elementos de un sistema basado en computadoras, y después ejecutar varias simulaciones para entender mejor las características operacionales y el desempeño general del sistema.

Existen dos amplias categorías de simulación del sistema: 1) herramientas de propósitos generales que pueden modelar de manera virtual cualquier sistema basado en computadoras, y 2) herramientas de propósitos especiales, que están diseñadas para emplearlas en un dominio de aplicación específico (como en sistemas de aerolíneas, sistemas de manufactura, sistemas electrónicos).

Herramientas representativas[4]

CSIM, desarrollado por Lockheed Martin Advanced Technology Labs (www.atl.external.lmco.com), es un simulador de eventos discretos de propósitos generales para sistemas orientados a diagramas de edificios

Simics, desarrollado por Virtutech (www.virtutech.com), es una plataforma de simulación de sistema que puede modelar y analizar sistemas basados en hardware y software

SLX, desarrollado por Wolverine Software (www.wolverine.com), proporciona bloques de construcción de propósito general para modelar el desempeño de una amplia variedad de sistemas.

En http://www.idsia.ch/~andrea/simtools.html se puede encontrar una serie de vínculos a varias fuentes de simulación de sistemas.

6.3 INGENIERÍA DE PROCESOS DE NEGOCIOS: UNA VISIÓN GENERAL

La meta de la *ingeniería de procesos de negocios* (IPN) es definir arquitecturas que permitan que un negocio utilice información de manera efectiva. Cuando las necesidades de tecnología de información de una compañía se observan de manera global, casi no hay duda de que se requiera la ingeniería de sistemas. No sólo se requiere la especificación de la arquitectura de cómputo apropiada, sino también se debe desarrollar la arquitectura de software que puebla la configuración única de fuentes de cómputo de la organización. La ingeniería de procesos de negocios es un enfoque que crea un plan general para implementar la arquitectura de cómputo [SPE93].

¿Cuáles son las arquitecturas que se definen y desarrollan como parte de la IPN?

Se deben analizar y diseñar tres arquitecturas diferentes dentro del contexto de objetivos y metas de negocios.

- Arquitectura de datos
- Arquitectura de aplicaciones
- Infraestructura de la tecnología

La *arquitectura de datos* proporciona un marco de trabajo para las necesidades de información de un negocio o de una función de negocios. Los ladrillos de la arquitectura son los objetos de datos que utiliza el negocio. Un objeto de los datos contiene un conjunto de atributos que define algún aspecto, cualidad, característica o descriptor de los datos que se describen.

Una vez definido un conjunto de datos se identifican sus relaciones. Una *relación* indica la forma en que los objetos están conectados entre sí. Como ejemplo se pue-

4 Las herramientas mostradas aquí son una muestra de esta categoría. En la mayoría de los casos los nombres están registrados por sus respectivos desarrolladores.

den considerar los objetos **cliente** y **productoA**. Los dos objetos pueden conectarse por la relación *compra*; es decir, un **cliente** *compra* **productoA** o **productoA** *es comprado* por un **cliente**. Los objetos de datos (que pueden existir cientos o hasta miles para una actividad de negocios importante) fluyen entre funciones de negocio, están organizados dentro de una base de datos y se transforman para ofrecer información que satisface las necesidades del negocio.

CONSEJO

...de ...no debe ...en PFI ni ... Sin embargo, ...o que estos ...no se han ...el ingeniero ...r a los ...que el ...del proyecto es ...

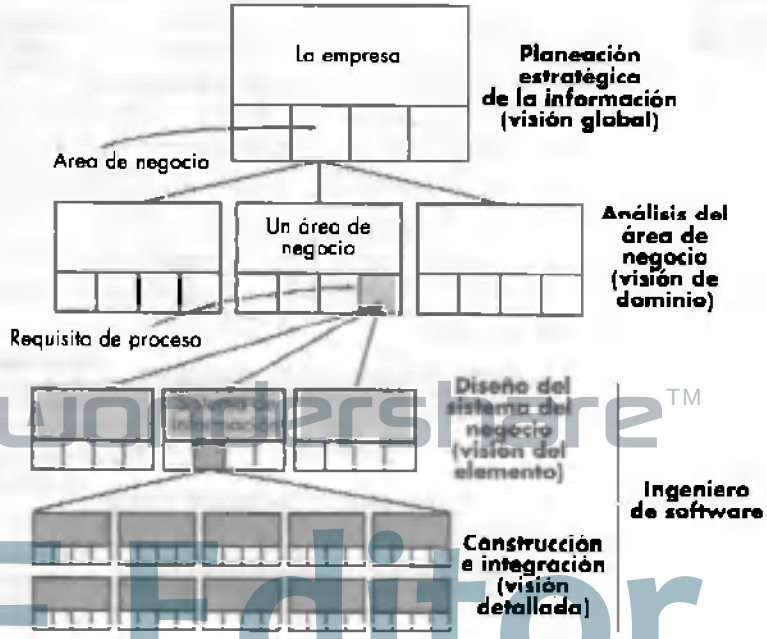
La *arquitectura de aplicación* abarca aquellos elementos de un sistema que transforman objetos dentro de la arquitectura de datos por algún propósito del negocio. En el contexto de este libro se considera que la arquitectura de aplicación es el sistema de programas (software) que realiza esta transformación. Sin embargo, en un contexto más amplio, la arquitectura de aplicación puede incorporar el papel de las personas (quienes son transformadores y usuarios de información) y procedimientos de negocios que no han sido automatizados.

La *infraestructura tecnológica* proporciona el fundamento para las estructuras de datos y de aplicación. La infraestructura comprende el hardware y el software con que se apoyan las aplicaciones y los datos. Esto incluye computadoras, sistemas de operación, redes de computadora, enlaces de telecomunicaciones, tecnologías de almacenamiento y la arquitectura (por ejemplo, cliente, servidor) diseñada para implementar estas tecnologías.

En la figura 6.2 se define e ilustra una jerarquía de proceso de negocios para modelar estas arquitecturas de sistema.

Figura 6.2

...jerarquía ...ería del ...de ...



6.4 INGENIERÍA DE PRODUCTO: UNA VISIÓN GENERAL

La meta de la *ingeniería de producto* es traducir el deseo del cliente, de una serie de capacidades definidas, a un producto del trabajo. Para conseguir esta meta la ingeniería de producto —como la ingeniería de procesos de negocios— debe crear una arquitectura y una estructura. La arquitectura abarca cuatro componentes de sistema distintos: software, hardware, datos (y bases de datos) y personas. Se establece una infraestructura de soporte e incluye la tecnología requerida para unir los componentes y la información (como documentos, CD-ROM, video) que se emplea para dar soporte a los componentes.

CONSEJO

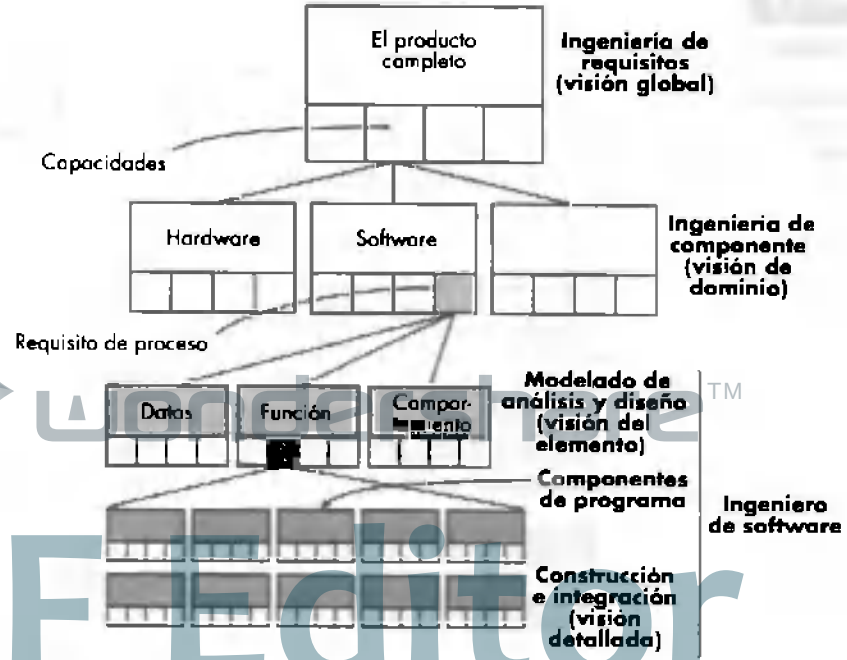
A menudo se utiliza en este contexto el modelo concurrente del proceso (capítulo 3). Cada disciplina de ingeniería trabaja en paralelo. Se debe estar seguro de que se promueva la comunicación mientras cada disciplina desempeña su trabajo

Como lo muestra la figura 6.3, la visión global se consigue mediante la ingeniería de requisitos (capítulo 7). Los requisitos generales del producto se obtienen del cliente. Estos requisitos comprenden necesidades de información y control, funcionalidad del producto y comportamiento, desempeño general del producto, diseño, restricciones de la interfaz y otras necesidades especiales. Una vez que se conocen estos requisitos, el trabajo de la ingeniería de requisitos es asignar función y comportamiento a cada uno de los cuatro componentes antes descritos.

Una vez hecha la asignación comienza la ingeniería de componentes del sistema. La ingeniería de componentes del sistema es en realidad un conjunto de actividades concurrentes que dirige por separado cada uno de los componentes del sistema: ingeniería de software, ingeniería de hardware, ingeniería humana e ingeniería de

FIGURA 6.3

La jerarquía de la ingeniería de productos.



bases de datos. Cada una de estas disciplinas de ingeniería toma una visión de dominio específica, pero es importante señalar que las disciplinas de ingeniería deben establecer y mantener una comunicación activa entre ellas. Parte del papel de la ingeniería de requisitos es establecer los mecanismos de interfaz que permitan que esto suceda.

La visión de elemento para la ingeniería de producto es la disciplina de ingeniería aplicada a un componente asignado. Para la ingeniería de software esto significa actividades de modelado del análisis y diseño (cubierto en detalle en capítulos posteriores) y actividades de construcción y despliegue que abarcan: generación de código, pruebas y tareas de soporte. Los modelos de análisis de tareas asignan requisitos a las representaciones de datos, función y comportamiento. El diseño convierte el modelo de análisis en diseños de datos, arquitectónicos, de interfaz y en el nivel de componentes del software.

HogarSeguro

Ingeniería de sistema preliminar

El escenario: Lugar de trabajo de ingeniería del software después de la junta de inicio de *HogarSeguro*.

Los actores: Jamie Lazar, miembro del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, miembro del equipo de software.

La conversación:

Jamie: Creo que estuvo muy bien.

Vinod: Sí, pero todo lo que hicimos fue ver el sistema en general, todavía tenemos que reunir muchos detalles para hacer el software.

Jamie: Por eso tenemos juntas adicionales programadas en los próximos cinco días. Por cierto, me gustaría que dos de los "clientes" estuvieran aquí en las próximas semanas. Ya sabes, que estén con nosotros para que podamos comunicarnos en realidad y...

Vinod: ¿Y qué opinaron los demás?

Jamie: Bueno, me miraron como si estuviera loco, pero [el gerente de ingeniería del software] le gustó lo –es ágil– así que está hablando con ellos.

Vinod: Yo estaba tomando notas con mi PDA durante la y obtuve una lista de funciones básicas.

Jamie: Qué bien, déjame ver.

Ed: Ya les mandé a los dos una copia por correo electrónico. Revísenla y luego hablamos.

Vinod: ¿Qué le parece después de la comida?

(Jamie y Vinod recibieron el siguiente texto de Ed.)

Notas preliminares de la estructura/funcionalidad de *HogarSeguro*:

- El sistema utilizará una o más PC, varios paneles de control manuales y montables en la pared, varios sensores y varios controladores de dispositivos/aplicaciones.
- Todos se comunicarán por protocolos inalámbricos (por ejemplo, 802.11b) y serán diseñados para la construcción de casas nuevas y la aplicación en casas existentes.
- Todo el hardware, a excepción de nuestra nueva caja inalámbrica, estará fuera del anaquel.

Funcionalidad básica del software obtenida de la conversación de inicio.

Funciones de seguridad de la casa:

- Sensor de movimiento de puerta/ventana para monitorear un acceso no autorizado (robos).
- Monitoreo de fuego y humo.
- Monitoreo de nivel de agua en sótano (por ejemplo, inundación o rompimiento del calentador de agua).

- Monitoreo de movimiento en el exterior
- Cambio de colocación de seguridad por Internet.

Funciones de vigilancia en la casa:

- Conectar a una o más cámaras de video colocadas fuera/dentro de la casa
- Controlar panorama/zoom en las cámaras.
- Definir zonas de monitoreo de las cámaras
- Mostrar tomas de la cámara en PC.
- Tener acceso a las tomas de la cámara vía Internet.
- Grabar digital y selectivamente las tomas de la cámara.
- Mostrar de nueva las tomas de la cámara

Funciones de la gestión de la casa:

- Controlar alumbrado
- Controlar instrumentos
- Controlar HVAC
- Controlar equipo de audio/video en la casa
- Habilitar la casa dentro de una modalidad "vacaciones/viaje" con un conjunto de botones:
  - Disponer aparatos/alumbrado/HVAC para que actúen de manera apropiada
  - Disponer un mensaje en la máquina contestadora.
  - Contactar vendedores para suspender entrega de periódicos, correo, etcétera

Funciones de la gestión de comunicación:

- Funciones de la máquina contestadora.
  - Lista de llamados a través de un identificador
  - Hora del mensaje
  - Texto del mensaje a través de un sistema de reconocimiento de voz
- Funciones del correo electrónico (todas las funciones estándares del correo electrónico)
  - Mostrar correo electrónico estándar
  - Lectura de voz por correo electrónico vía acceso telefónico
- Directorio telefónico personal
- Vínculo con el PDA.

Otras funciones:

Por definirse.

Todas las funciones serán accesibles vía Internet con una contraseña apropiada como protección.

6.5 MODELADO DEL SISTEMA

Debido a que un sistema puede representarse con diferentes grados de abstracción (por ejemplo, la visión global, la visión de dominio, la visión de elemento), los *modelos de sistema* tienden a ser jerárquicos o estratificados por naturaleza. En la parte más alta de la jerarquía se presenta un modelo del sistema completo (la visión global). Los objetos principales de datos, las funciones de procesamiento y los comportamientos se representan sin incluir el componente del sistema que implementará los elementos del modelo de visión global. A medida que la jerarquía se refina o estratifica se modela el detalle al nivel de componentes (en este caso, representaciones de hardware, software, etcétera). Al final, los modelos de sistemas evolucionan a modelos de ingeniería (los cuales se refinan después) que son específicos para la disciplina de ingeniería apropiada.

6.5.1 Modelado Hatley-Pirbhai

Todo sistema basado en computadora puede modelarse como transformación de la información al emplear una plantilla de entrada-proceso-salida. Hatley y Pirbhai [HAT87] han ampliado esta visión para incluir dos características adicionales del sistema: procesamiento de la interfaz del usuario y mantenimiento y procesamiento de

autocomprobación. Aunque estas características adicionales no están presentes en todos los sistemas basados en computadora, son comunes y su especificación hace que cualquier modelo de sistema sea más robusto.

Con el uso de la representación de entrada, procesamiento, salida, procesamientos de la interfaz del usuario y procesamiento de autocomprobación, un ingeniero de sistemas puede crear un modelo de componentes de sistema que deje un fundamento para etapas posteriores en cada una de las disciplinas de ingeniería.

PUNTO CLAVE

El modelo Kelly-Hatley representa la entrada, el procesamiento y la salida con la interfaz del usuario y el mantenimiento/autocomprobación.

En el desarrollo de un modelo de sistema se utiliza una plantilla modelo del sistema [HAT87]. El ingeniero de sistemas asigna elementos de sistema a cada una de las cinco regiones de procesamiento dentro de la plantilla: 1) interfaz del usuario, 2) entrada, 3) funcionamiento y control del sistema, 4) salida, y 5) mantenimiento y autocomprobación.

Al igual que casi todas las técnicas de modelado utilizadas en la ingeniería de sistemas y de software, la plantilla modelo del sistema le permite al analista crear una jerarquía en detalle. En el nivel más alto de la jerarquía está el *diagrama de contexto del sistema* (DCS). El diagrama de contexto "establece los límites de información entre el sistema que implementa y el ambiente en el que opera el sistema" [HAT87]. Es decir, el DCS define todos los productores externos de información que el sistema utiliza, todos los consumidores externos de información que el sistema crea, y todas las entidades que se comunican a través de la interfaz o realizan mantenimiento y autocomprobación.

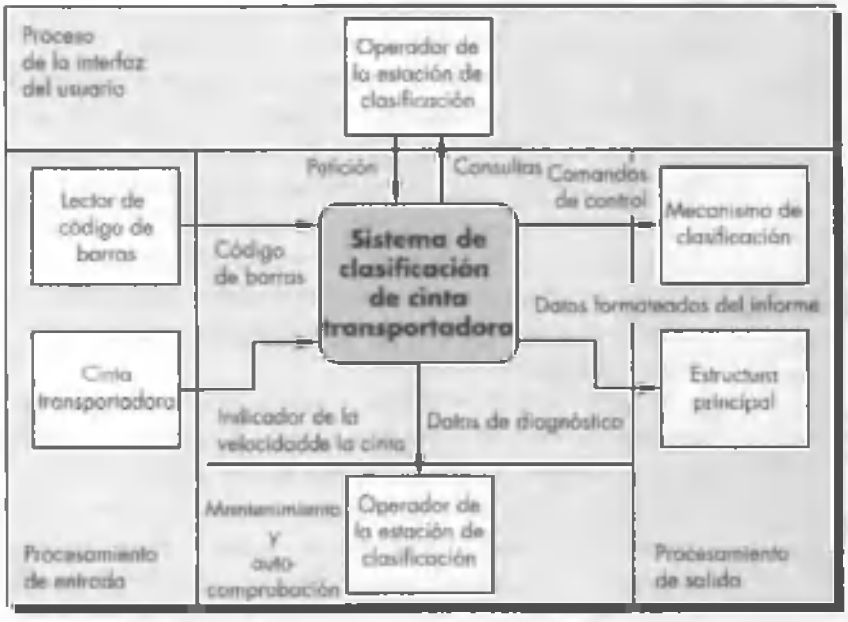
Para ilustrar el uso del DCS se considerará un *sistema de clasificación de cinta transportadora* (SCCT) descrito en la siguiente declaración (un tanto confusa) de objetivos:

> El SCCT debe desarrollarse de manera que las cajas que se mueven a lo largo de la cinta transportadora sean identificadas y ordenadas en uno de los seis contenedores al final de la cinta. Las cajas pasarán a través de una estación clasificadora, donde se identificarán. Con base en un número de identificación impreso en un lateral de la caja y un código de barras, las cajas se mandarán a los contenedores apropiados. Las cajas pasan en un orden aleatorio y están igualmente espaciadas. La línea se mueve con lentitud.
>
> Una computadora de escritorio localizada en la estación clasificadora ejecuta todo el software del SCCT, interactúa con el lector de código de barras para leer números de parte en cada caja, interactúa con el equipo de monitoreo de la línea transportadora para obtener la velocidad de la línea transportadora, almacena todos los números de parte clasificados, interactúa con un operador de la estación clasificadora para producir varios reportes y diagnósticos, manda señales de control al hardware para clasificar las cajas, y se comunica con un sistema de automatización central de la fábrica.

El DCS para el SCCT se muestra en la figura 6.4. El diagrama se divide en cinco segmentos principales. El segmento de arriba representa el procesamiento de la interfaz del usuario, y los segmentos de la izquierda y de la derecha muestran el procesamiento de entrada y de salida, respectivamente. El segmento central contiene funciones de control y proceso, y el segmento de abajo se enfoca en el mantenimiento

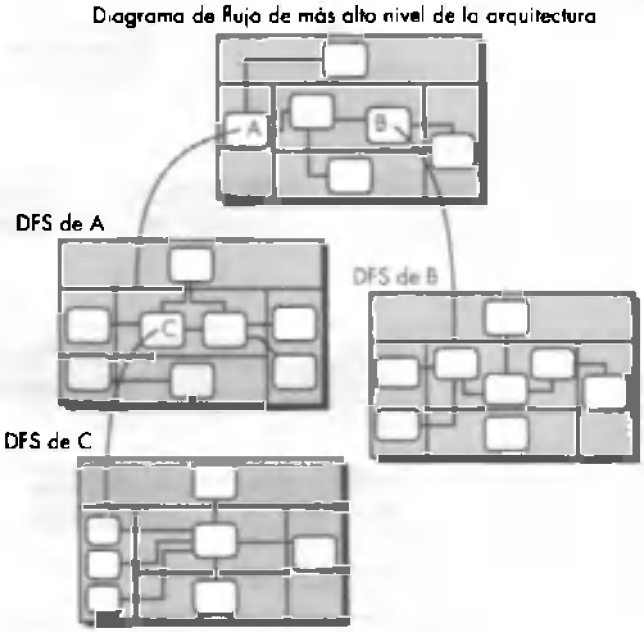
FIGURA 6.4

Diagrama de contexto del sistema del SCCT.



y la autocomprobación. Cada caja que se muestra en la figura representa una *entidad externa*; es decir, un productor o consumidor de información del sistema. Por ejemplo, el lector de código de barras produce información que es introducida al sistema SCCT. El símbolo para el sistema completo (o, a niveles más bajos, subsistemas principales) es un rectángulo con las esquinas redondeadas. Por lo tanto, el SCCT se representa en la región de procesamiento y control al centro del DCS. Las flechas etiquetadas que se muestran en el DCS representan información (datos y control) que va de un ambiente externo hacia el sistema SCCT. La entidad externa lector de código de barras produce entrada de información etiquetada como código de barras. En esencia, el DSC coloca cualquier sistema en el contexto del ambiente externo.

El ingeniero de sistemas refina el diagrama de contexto de arquitectura al estudiar con más detalle el rectángulo sombreado de la figura 6.4. Se identifican los subsistemas principales que permiten funcionar al sistema clasificador de cinta transportadora dentro del contexto definido por el DCS. Los subsistemas principales se definen en un *diagrama de flujo del sistema* (DFS) que se obtiene del DCS. El flujo de información a través de las regiones del DCS se utiliza para guiar al ingeniero de sistemas en el desarrollo del DFS, un esquema más detallado del SCCT. El diagrama de flujo del sistema muestra los subsistemas principales y el flujo de las líneas de información importantes (datos y control). Además, la plantilla del sistema divide el proceso del subsistema en cada una de las cinco regiones de proceso previamente estudiadas. En este punto, cada uno de los subsistemas puede contener uno o más elementos del sistema (por ejemplo, hardware, software, personas) tal y como los ha asignado el ingeniero de sistemas.



El diagrama de flujo del sistema (DFS) inicial se convierte en el nodo superior de una jerarquía de DFS. Cada rectángulo redondeado del DFS original puede expandirse en otra plantilla de arquitectura dedicada a ella en forma exclusiva. Este proceso se ilustra de manera esquemática en la figura 6.5. Cada uno de los DFS del sistema puede utilizarse como punto de partida de subsiguientes fases de ingeniería para el subsistema que se describe.

En los subsiguientes trabajos de ingeniería se pueden especificar (delimitar) los subsistemas y la información que fluyen entre ellos. Un relato descriptivo de cada subsistema y una definición de todos los datos que fluyen entre los subsistemas son elementos importantes de la especificación del sistema.

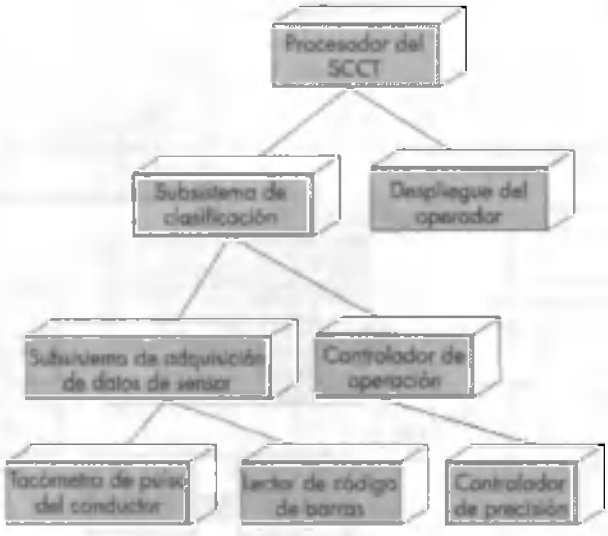
6.5.2 Modelado del sistema con UML

El UML proporciona una cantidad impresionante de diagramas que pueden utilizarse para el análisis y diseño al nivel de software y del sistema.[5] Para el SCCT se modelan cuatro elementos importantes del sistema. 1) el hardware que permite el SCCT; 2) el software que implementa el acceso a la base de datos y la clasificación; 3) el operador que acata varias peticiones del sistema; y 4) la base de datos que contiene información relevante del código de barras y el destino

5 En los capítulos del 8 al 11 se presenta una exposición más detallada de los diagramas de UML. Para una exposición completa del UML, el lector interesado debe consultar [SCH02], [LAR01] o [BEN99].

FIGURA 6.6

Diagrama de despliegue del hardware de SCCT.



El hardware del SCCT se puede modelar en el nivel del sistema mediante un *diagrama de despliegue* de UML, como se ilustra en la figura 6.6. Cada caja tridimensional muestra un elemento del hardware que es parte de la arquitectura física del sistema. En algunos casos, los elementos del hardware tendrán que diseñarse y construirse como parte del proyecto. Sin embargo, en muchos casos los elementos del hardware se pueden adquirir ya construidos. El reto para el equipo de ingeniería es realizar la interfaz de los elementos del hardware de manera apropiada.

Referencia Web

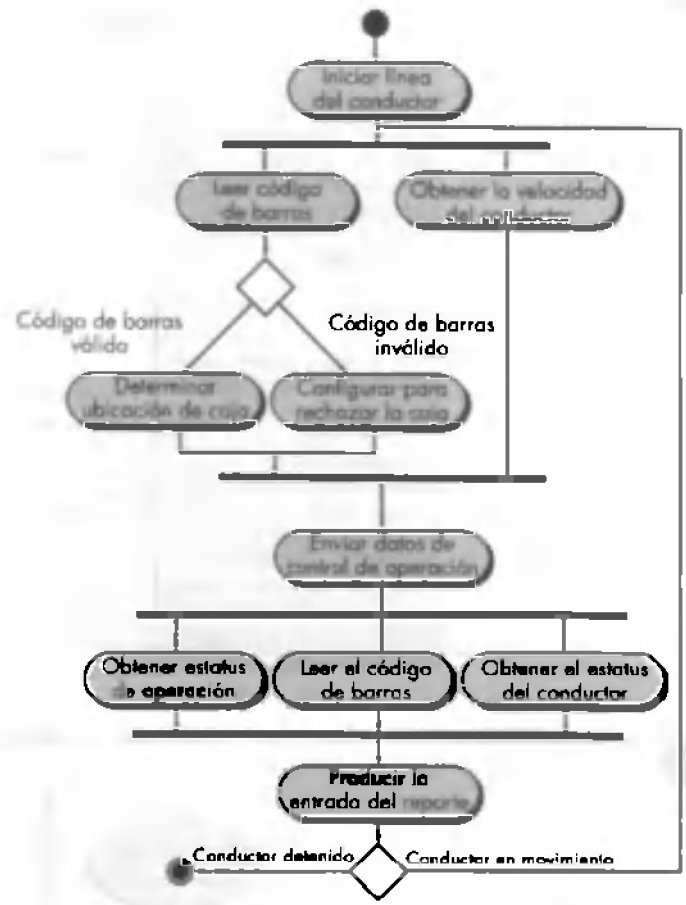
En **www.rational.com/uml/index.jsp** se puede encontrar una especificación completa de la sintáctica y semántica del UML (explicada en capítulos posteriores).

Los elementos del software para el SCCT se pueden modelar en una variedad de formas mediante el uso de UML. Los aspectos de procedimiento del software del SCCT se pueden representar mediante un *diagrama de actividad* (figura 6.7). Esta notación del UML es similar al diagrama de flujo y se utiliza para representar lo que sucede mientras el sistema realiza sus funciones. Los rectángulos redondeados implican una función específica del sistema; las flechas representan el flujo a través del sistema; el rombo de decisión representa una decisión ramificada (cada flecha que sale del rombo está etiquetada); las líneas sólidas horizontales implican la realización de actividades en paralelo.

Otra notación de UML que se puede usar para modelar software es el *diagrama de clase* (junto con muchos diagramas relacionados con las clases que se examinan en apartados posteriores de este libro). En el nivel de la ingeniería del sistema las clases[6] se extraen en un enunciado del problema. Para el SCCT las clases candidatas

6 En capítulos anteriores se destacó que una clase representa un conjunto de entidades que forman parte del dominio del sistema. El sistema puede almacenar o transformar estas entidades o pueden servir como un productor o consumidor de la información que el sistema produce.

Figura 6.7
Diagrama de actividad para SCCT



podrían ser de: **caja, línea de conducción, lector de código de barras, controlador de maniobras, solicitud del operador, reporte, producto** y otras. Cada clase encapsula un conjunto de atributos que representa toda la información necesaria acerca de la clase. Una descripción de clase también contiene un conjunto de operaciones que se aplican a la clase en el contexto del sistema SCCT. En la figura 6.8 se muestra un diagrama de clase de UML la clase **caja**.

El operador del SCCT se puede modelar con un diagrama de UML de tipo casos de uso como se muestra en la figura 6.9. El diagrama de caso de uso ilustra la forma en la que un actor (en este caso el operador que se representa con una figura adherida) interactúa con el sistema. Cada óvalo etiquetado dentro de la caja (la cual representa la frontera del sistema SCCT) implica un caso de uso —un escenario escrito que describe una interacción con el sistema.

FIGURA 6.8

Diagrama de clase de UML para la clase caja.

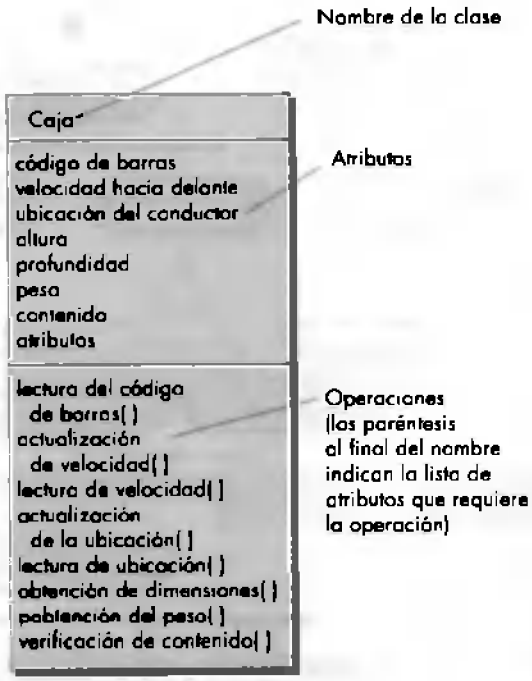
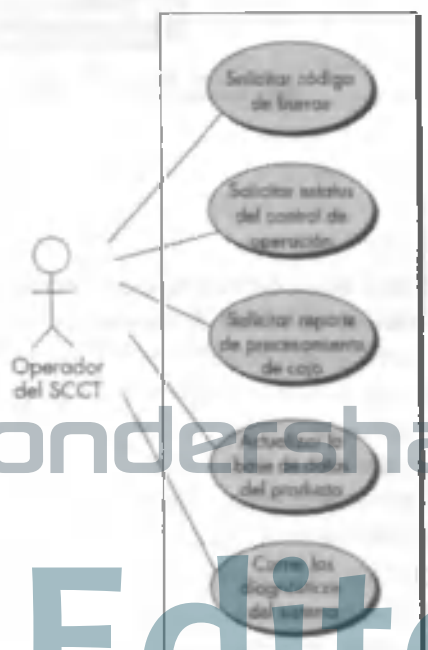


FIGURA 6.9

Diagrama de caso de uso para el operador SCCT.



HERRAMIENTAS DE SOFTWARE

Herramientas de modelado de sistemas

Objetivo: Las herramientas de modelado de sistemas proporcionan al ingeniero de software la capacidad de modelar todos los elementos de un sistema basado en computadoras al usar una notación específica para la herramienta.

Mecánica: Las mecánicas de las herramientas varían. Por lo general, las herramientas de esta categoría ayudan al ingeniero de sistemas a modelar 1) la estructura de todos los elementos funcionales del sistema; 2) el comportamiento estático y dinámico del sistema, 3) la interfaz máquina-humano

Herramientas representativas[7]

Describe, desarrollado por Embarcadero Technologies (www.embarcadero.com), es una adaptación de herramientas de modelado basadas en UML que puede representar sistemas de software o sistemas completos.

Rational XDE and Rose, desarrollado por Rational Technologies (www.rational.com), proporciona una adaptación basada en UML de herramientas de desarrollo y modelado para sistemas basados en computadoras, la cual se utiliza de manera amplia.

Real-Time Studio, desarrollado por Artisan Software (www.artisansw.com) es una conjunto de herramientas de modelado y desarrollo que dan soporte al desarrollo de sistemas en tiempo real.

Telelogic Tau, desarrollado por Telelogic (www.telelogic.com), es un conjunto con herramientas basadas en UML que da soporte al modelado de diseño y análisis, y tiene vínculos con características de construcción de software.

6.6 RESUMEN

Un sistema de alta tecnología comprende varios componentes: hardware, software, personas, bases de datos y procedimientos. La ingeniería de sistemas ayuda a traducir las necesidades del cliente en un modelo de sistema que emplea uno o más de estos componentes

La ingeniería de sistemas comienza al adoptar una "visión global". Se analiza el dominio del negocio o producto para establecer todos los requisitos básicos. El enfoque se reduce entonces a una "visión de dominio", donde cada uno de los elementos del sistema se analiza en forma individual. Cada elemento se asigna a uno o más componentes de ingeniería, los cuales se estudian aplicando la disciplina de ingeniería correspondiente.

La ingeniería del proceso de negocios es un enfoque de la ingeniería de sistemas mediante el cual se definen arquitecturas que permitan a un negocio utilizar la información de manera eficaz. El objetivo de la ingeniería del proceso de negocios es crear una arquitectura de datos, una arquitectura de aplicación y una infraestructura de tecnología comprensibles que satisfagan las necesidades de la estrategia de negocio y los objetivos y metas de cada área del negocio.

La ingeniería de productos es un enfoque de la ingeniería de sistemas que comienza con el análisis del sistema. El ingeniero de sistemas identifica las necesi-

7 Las herramientas mostradas aquí son una muestra dentro de esta categoría. En la mayoría de los casos los nombres están registrados por sus respectivos desarrolladores

dades del cliente, determina la factibilidad económica y técnica, y asigna funciones y rendimientos al software, el hardware, las personas y las bases de datos; es decir a los componentes clave de la ingeniería.

Referencias

[BEN99] Bennett, S., S. McRobb y R. Farmer, *Object-Oriented Systems Analysis and Design Using UML*, McGraw-Hill, 1999.

[HAR93] Hares, J. S., *Information Engineering for Advanced Practitioner*, Wiley, 1993, pp. 12-13.

[HAT87] Hatley, D. J. e I. A. Pirbhai, *Strategies for Real Time System Specification*, Dorset House, 1987.

[LAR01] Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2a. ed., Prentice-Hall, mayo de 2001.

[MAR90] Martin, J., *Information Engineering: Book II—Planning and Analysis*, Prentice-Hall, 1990.

[MOT92] Motamarri, S., "Systems Modeling and Description", en *Software Engineering Notes*, vol. 17, núm. 2, abril de 1992, pp. 57-63.

[SCH02] Schmuller, J., *Teach Yourself UML in 24 Hours*, 2a. ed., Sams Publishing, 2002.

[SPE93] Spewak, S., *Enterprise Architecture Planning*, QED Publishing, 1993.

[THA97] Thayer, R. H. y M. Dorfman, *Software Requirements Engineering*, 2a. ed., IEEE Computer Society Press, 1997.

Problemas y puntos a considerar

6.1. Encontrar tantos sinónimos como se pueda de la palabra "sistema". ¡Buena suerte!

6.2. Construir un "sistema de sistemas" jerárquico para un sistema, producto o servicio con el cual se esté familiarizado. La jerarquía se debe extender hasta los elementos simples del sistema (hardware, software, etcétera) de al menos una rama de cada estructura.

6.3. Seleccionar un sistema o producto grande con el que esté familiarizado. Definir el conjunto de dominios que definan la visión global del sistema o producto. Describir el conjunto de elementos que componen uno o dos de los dominios. Para un elemento, identificar los componentes técnicos que deben desarrollarse.

6.4. Seleccionar algún sistema o producto grande con el cual esté familiarizado. Establecer las suposiciones, simplificaciones, limitaciones, restricciones y preferencias que se deberían hacer para construir un modelo de sistema de modelo eficaz (y realizable).

6.5. La ingeniería de procesos del negocio requiere definir *datos, arquitectura de aplicaciones*, además de una *infraestructura de aplicaciones*. Describir cada uno de estos términos mediante un ejemplo.

6.6. Un ingeniero de sistemas puede tener una de tres procedencias: el desarrollo de sistemas, el cliente o una organización externa. Discutir los pros y los contras de cada procedencia. Describir un ingeniero de sistemas "ideal".

6.7. El profesor entregará una descripción de alto nivel de un sistema o producto basado en computadoras.

a) Desarrollar un conjunto de preguntas que se debería realizar como ingeniero de sistemas.

b) Proponer al menos dos ubicaciones diferentes para el sistema con base en las respuestas a sus preguntas.

c) En clase, comparar sus ubicaciones con las de sus compañeros.

6.8. Desarrollar un diagrama de contexto del sistema para el sistema basado en computadoras que se haya elegido (o uno asignado por su profesor).

6.9. Aunque la información hasta este punto está muy entrecortada, trátese de desarrollar un diagrama de desarrollo, un diagrama de actividad, un diagrama de clase y un diagrama de caso de uso con UML para el producto *HogarSeguro*.

6.10. Realizar una investigación bibliográfica y escribir un documento breve que describa cómo funcionan las herramientas de modelado y simulación. Alternativa· recopile bibliografía de dos o más vendedores de herramientas de modelado y simulación y evalúe sus similitudes y diferencias

6.11. ¿Existen características de un sistema que no se puedan establecer durante las actividades de la ingeniería de sistemas? Describir las características, si existen, y explicar por qué su consideración se debe retrasar a fases posteriores del desarrollo.

6.12. ¿Existen situaciones en las que la especificación formal del sistema se pueda abreviar o eliminar por completo? Expliquese la respuesta.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Los libros de Hatley y sus colegas (*Process for Systems Architecture and Requirements Engineering*, Dorset House, 2001), Buede (*The Engineering Design of Systems: Models and Methods*, Wiley, 1999), Weiss y sus colegas (*Software Product-Line Engineering*, Addison-Wesley, 1999), Blanchard y Fabrycky (*System Engineering and Analysis*, 3a. ed., Prentice-Hall, 1998), Armstrong y Sage (*Introduction to Systems Engineering*, Wiley, 1997), y Martin (*Systems Engineering Guidebook*, CRC Press, 1996) presentan el proceso de la ingeniería del sistema (con un énfasis distinto en la ingeniería) y proporcionan una guía muy valiosa Blanchard (*System Engineering Management*, segunda edición, Wiley, 1997) y Lacy (*System Engineering Management*, McGraw-Hill, 1992) exponen aspectos de gestión de la ingeniería del sistema

Chorafas (*Enterprise Architecture and New Generation Systems*, St. Lucie Press, 2001) presenta ingeniería de información y arquitecturas de sistema para la "siguiente generación" de soluciones de TI; se incluyen sistemas basados en Internet Wallnau y sus colegas (*Building Systems from Comercial Componennts*, Addison-Wesley, 2001) se enfoca en los aspectos de la ingeniería de sistemas basada en componentes para productos y sistemas de información. Lozinsky (*Enterprise-Wide Software Solutions: Integration Strategies and Practices*, Addison-Wesley, 1998) abarca el uso de paquetes de software como una solución que permite a las compañías pasar de los sistemas heredados a los procesos de negocio modernos. Una exposición muy valiosa del riesgo y la ingeniería del sistema se presenta en el libro de Bradley (*Elimination of Risk in Systems*, Tharsis Boooks, 2002).

Davis (*Business Process Modeling with Aris: A Practical Guide*, Springer-Verlag, 2001), Bustard y sus colegas (*System Models for Business Process Improvement*, Artech House, 2000), y Scheer (*Business Process Engineering. Reference Models for Industrial Enterprises*, Springer-Verlag, 1998) describen los métodos de modelado del proceso de negocios para sistemas de información y de toda una empresa

Davis y Yen (*The Information System Consultant's Handbook: Systems Analysis and Design*, CRC Press, 1998) presentan una cobertura enciclopédica de los aspectos del análisis y diseño de sistemas en el dominio de los sistemas de información. Una ayuda excelente del IEEE por Thayer y Dorfman [THA97] discute la interrelación entre los análisis al nivel de sistema y al nivel de software

Law y sus colegas (*Simulation Modeling and Analysis*, McGraw-Hill, 1999) analizan técnicas de modelado y simulación de sistemas para una amplia variedad de dominios de aplicación.

Para los lectores involucrados de manera activa en el trabajo de sistemas o que están interesados en un tratamiento más elaborado del tópico, los libros de Gerald Weinberg (*An Introduction to General System Thinking*, Wiley, Interscience, 1976, y *On the Design of Stable Systems*, Wiley-Interscience, 1979) se han convertido en clásicos y ofrecen una excelente exposición sobre el "pensamiento general de sistemas", lo que de manera implícita conduce a un enfoque general del análisis y diseño de sistemas. Otros libros más recientes de Weinberg (*General Principles of Systems Design*, Dorset House, 1998 y *Rethinking Systems Analysis and Design*, Dorset House, 1998) continúan la tradición de este primer trabajo

En Internet existe una amplia variedad de fuentes de información sobre la ingeniería de sistemas y materias relacionadas. En el sitio Web de SEPA, **http://www.mhhe.com/pressman**, se puede encontrar una lista actualizada de referencias en la red mundial que son relevantes para la ingeniería del sistema, la ingeniería de la información, la ingeniería de proceso del negocio y la ingeniería del producto.

CAPÍTULO

7

INGENIERÍA DE REQUISITOS

La comprensión de los requisitos de un problema está entre las tareas más difíciles que enfrenta un ingeniero de software. Cuando se piensa por primera vez acerca de ello, la ingeniería de requisitos no parece tan difícil. Después de todo, ¿el cliente no sabe lo que se requiere? ¿Los usuarios finales no deberían entender bien las características y funciones que les proporcionarán un beneficio? Es sorprendente, pero en muchas ocasiones la respuesta a estas preguntas es: "no". Y aun si los clientes y usuarios finales son explícitos en sus necesidades, estos requisitos pueden cambiar durante el proyecto La ingeniería de requisitos es difícil

En el prólogo a un libro de Ralph Young [YOU01] sobre las prácticas efectivas en los requisitos, el autor de este libro escribió:

> Es tu peor pesadilla Un cliente entra en tu oficina, se sienta, te mira directo a los ojos, y dice: "Yo sé que usted piensa que entiende lo que digo, pero lo que usted no entiende es que lo que digo no es realmente lo que quiero decir". Esto sucede de manera invariable cuando el proyecto está avanzado, después de que se han realizado los compromisos relativos al tiempo de entrega, las reputaciones están en juego y el dinero está en serio peligro.
>
> Todos los que hemos trabajado en el negocio de los sistemas y el software por más de unos cuantos años hemos vivido esta pesadilla, y sólo unos pocos de nosotros hemos aprendido a continuar aun con esta circunstancia. Nosotros tenemos dificultades cuando tratamos de obtener requisitos de nuestros clientes. Tenemos problemas al comprender la información que adquirimos Con frecuencia, registramos los re-

UN VISTAZO RÁPIDO

¿Qué es? La ingeniería de requisitos ayuda a los ingenieros de software a entender mejor el problema en cuya solución trabajarán. Incluye el conjunto de tareas que conducen a comprender cuál será el impacto del software en el negocio, qué es lo que el cliente quiere que interactuarán los usuarios finales con el sistema

¿Quién lo hace? Los ingenieros de software (a veces referidos como *ingenieros de sistemas* o *analistas* en el mundo de la TI) y otros interesados (gerentes, clientes y usuarios finales) participan en la ingeniería de requisitos

¿Por qué es importante? El diseño y la construcción de un elegante programa de computadora que resuelva el problema incorrecto no satisface las necesidades de nadie. Por lo tanto, es muy importante entender lo que el cliente quiere antes de comenzar a diseñar y construir un sistema basado en computadora.

¿Cuáles son los pasos? La ingeniería de requisitos empieza con la fase de inicio, la cual es una tarea que define el ámbito y la naturaleza del problema que debe resolverse Después continúa con la obtención, que es una tarea que ayuda al cliente a definir sus necesidades; posteriormente sigue con la elaboración, que es la

fase donde se refinan y modifican los requisitos básicos. Cuando el cliente ha definido el problema se lleva a cabo la negociación, donde se define cuáles son las prioridades, cuáles aspectos son esenciales y en qué momento se requieren. Por último, el problema se especifica de alguna manera, y después es validado y revisado para asegurar que la concepción del problema que tiene el ingeniero de software coincide con la percepción del cliente.

¿Cuál es el producto obtenido? El objetivo del proceso de la ingeniería de requisitos es darle a todas las partes una explicación escrita del problema. Esto puede lograrse por medio de varios productos de trabajo: escenarios de uso, listas de funciones y características, modelos de análisis o alguna especificación.

¿Cómo puedo estar seguro de que lo he hecho correctamente? El ingeniero de software revisa los productos de trabajo de la ingeniería de requisitos junto con el cliente y los usuarios finales para asegurarse que haya entendido lo que en realidad pretendían decirle. Es necesario hacer una advertencia: aun después de que todas las partes están de acuerdo, las cosas cambian, y continuarán haciéndolo a través de la vida del proyecto.

quisitos de una manera desorganizada e invertimos muy poco tiempo en verificar lo que registramos. Permitimos que el cambio nos controle en lugar de establecer mecanismos para controlarlo. En resumen, fallamos al establecer un cimiento sólido para el sistema o software. Cada uno de estos problemas representa un reto. Cuando éstos se combinan, la imagen es desalentadora incluso para los gerentes y profesionales del sofware más experimentados. Pero existen soluciones.

Sería deshonesto decir que la ingeniería de requisitos es la "solución" para los retos que se han enunciado. Pero proporciona un enfoque sólido para abordar dichos desafíos.

7.1 Un puente hacia el diseño y la construcción

Las actividades de diseño y construcción de software de computadora son desafiantes, creativas y hasta divertidas. De hecho, la construcción es tan irresistible que muchos desarrolladores de software quieren entrar en ella antes de comprender con claridad de qué es lo que se necesita. Ellos argumentan que las cosas se aclararán mientras construyen; que los interesados en el software serán capaces de entender mejor las necesidades sólo después de examinar las primeras iteraciones del software; que las cosas cambian tan rápido que la ingeniería de requisitos es una pérdida de tiempo; que la línea de base produce un programa que funciona y todo lo demás es secundario. Lo que hace seductores a estos argumentos es que contienen elementos de verdad.[1] Pero cada uno de ellos es imperfecto y puede conducir a un proyecto de software fallido.

1 En particular, esto es cierto para los proyectos chicos (menos de un mes) que implican un esfuerzo relativamente pequeño. Conforme el software crece en tamaño y complejidad, estos argumentos comienzan a derrumbarse.

"La parte más difícil de construir un sistema de software es decidir qué construir. Ninguna parte del trabajo estropea tanto el sistema resultante si se hace mal. Ninguna parte es más difícil de rectificar después."

Fred Brooks

La ingeniería de requisitos, como todas las demás actividades de la ingeniería del software, debe adaptarse a las necesidades del proceso, el proyecto, el producto y las personas que realizan el trabajo. Desde la perspectiva del proceso del software, la ingeniería de requisitos (IR) es una acción de la ingeniería del software que comienza durante la actividad de comunicación y continúa en la actividad de modelado.

En algunos casos se elige un enfoque abreviado. En otros, cada una de las tareas definidas para comprender los requisitos se debe llevar a cabo de manera rigurosa. Sobre todo, el equipo de software debe adaptar su enfoque a la IR, lo que no significa abandono. Es esencial que el equipo de software haga un esfuerzo real por entender los requisitos de un problema *antes* de intentar resolverlo.

PUNTO CLAVE

La ingeniería de requisitos tiende un puente hacia el diseño y la construcción. Pero ¿dónde se origina el puente? Se puede argumentar que comienza al pie de los participantes del proyecto (es decir, gerentes, clientes, usuarios finales), donde se definen las necesidades del negocio, se describen los escenarios de los usuarios, se delinean las características y funciones, y se identifican las restricciones del proyecto. Otros quizá sugieran que comienza con la definición más amplia del sistema, en el que el software es sólo un componente (capítulo 6) del dominio del sistema que es aún mayor. Pero sin importar el punto de inicio, el trabajo a lo largo del puente se inicia en la parte alta del proyecto, lo que permite que el equipo de software examine el contexto del trabajo de software que será realizado; las necesidades específicas que el diseño y la construcción deben abordar; las prioridades que indican el orden en el que se debe completar el trabajo; y la información, las funciones y los comportamientos que tendrán un impacto profundo en el diseño resultante.

7.2 TAREAS DE LA INGENIERÍA DE REQUISITOS

La ingeniería de requisitos proporciona el mecanismo apropiado para entender lo que el cliente quiere, analizar las necesidades, evaluar la factibilidad, negociar una solución razonable, especificar la solución sin ambigüedades, validar la especificación, y administrar los requisitos conforme éstos se transforman en un sistema operacional [THA97]. El proceso de la ingeniería de requisitos se lleva a cabo a través de siete distintas funciones: *inicio, obtención, elaboración, negociación, especificación, validación y gestión*.

Resulta importante destacar que algunas de estas funciones de la ingeniería de requisitos ocurren en paralelo y que todas deben adaptarse a las necesidades del proyecto. Todas están dirigidas a definir lo que el cliente quiere, y todas sirven para establecer una base sólida respecto del diseño y la construcción de lo que obtendrá el cliente.

7.2.1 Inicio

¿Cómo se inicia un proyecto de software? ¿Es un evento aislado que se convierte en el catalizador para un nuevo sistema o producto basado en computadora? ¿O la necesidad evoluciona con el tiempo? No existen respuestas definitivas para estas preguntas.

> "Por lo general, las semillas de los desastres más importantes en software se siembran en los primeros tres meses desde el comienzo del proyecto."
>
> **Capers Jones**

En algunos casos, una conversación informal es todo lo que se necesita para precipitar un esfuerzo importante de ingeniería del software. Pero en general, la mayoría de los proyectos comienza cuando se identifica una necesidad de negocios o se descubre un nuevo mercado o servicio potencial. Los participantes de la comunidad de negocios (es decir, los gerentes, gente de mercadotecnia, gerentes de producto) definen un caso de negocios para la idea, tratan de identificar la amplitud y profundidad del mercado, hacen un análisis preliminar de factibilidad, e identifican una descripción funcional del ámbito del proyecto. Toda esta información está sujeta a cambios (una situación probable), pero es suficiente para suscitar conversaciones con la organización de ingeniería del software.[2]

Al *inicio*[3] del proyecto los ingenieros de software hacen una serie de preguntas libres de contexto, las cuales se exponen en la sección 7.3.4. El objetivo es establecer una comprensión básica del problema, las personas que quieren una solución, la naturaleza de la solución que se desea, y la efectividad de la comunicación preliminar entre el cliente y el desarrollador.

7.2.2 Obtención

En verdad parece muy simple preguntarle al cliente, a los usuarios y otros interesados cuáles son los objetivos para el sistema o producto, qué es lo que se debe lograr, de qué forma el producto satisface las necesidades del negocio y por último cómo se utilizará el sistema o producto día a día. Pero no es simple, es muy difícil.

Christel y Kang [CRI92] identifican una serie de problemas que ayudan a entender por qué es difícil la *obtención* de requisitos

¿Por qué es difícil comprender con claridad lo que quiere el cliente?

- **Problemas de ámbito.** El límite del sistema está mal definido o los clientes/usuarios especifican detalles técnicos innecesarios que pueden confundir, en lugar de clarificar, los objetivos generales del sistema.

2 Si se va a construir un sistema basado en computadora, las discusiones comienzan con la ingeniería del sistema, una actividad que define la visión global y la visión de dominio para el sistema (capítulo 6).

3 Los lectores del capítulo 3 recordarán que el proceso unificado define una "fase de inicio" más completa, la cual incluye las tareas de inicio, obtención y elaboración tal como se examinan en este capítulo.

- **Problemas de comprensión.** Los clientes/usuarios no están seguros por completo de qué es lo que se necesita, comprenden poco acerca de las capacidades y limitaciones de su ambiente de cómputo, no comprenden del todo el dominio del problema, tienen dificultades al comunicar necesidades al ingeniero de sistemas, omiten información que consideran "obvia", especifican requisitos que chocan con las necesidades de otros clientes/usuarios, o especifican requisitos ambiguos o inestables
- **Problemas de volatilidad.** Los problemas cambian conforme transcurre el tiempo.

Para ayudar a superar estos problemas, los ingenieros de requisitos deben realizar en forma organizada la actividad de recopilación de requisitos.

7.2.3 Elaboración

La información conseguida con el cliente durante el inicio y la obtención se expande y se refina durante la *elaboración*. Esta actividad de la ingeniería de requisitos se enfoca en el desarrollo de un modelo técnico refinado de las funciones, características y restricciones del software.

La elaboración es una acción del modelado del análisis (capítulo 8) y se compone de una serie de tareas de modelado y refinamiento. La elaboración se conduce mediante la creación y el refinamiento de escenarios del usuario que describen la forma en que el usuario final (y otros actores) interactuarán con el sistema Cada escenario del usuario se analiza para obtener clases de análisis: entidades del dominio de negocios visibles para el usuario final. Se definen los atributos de cada clase de análisis y se identifican los *servicios*[4] que requiere cada clase Se identifican las relaciones y la colaboración entre las clases y se produce una variedad de diagramas de UML complementarios.

El resultado final de la elaboración es un modelo de análisis que define el dominio de la información, las funciones y el comportamiento del problema

INFORMACIÓN

Modelado del análisis

Supóngase por un momento que es necesario especificar todos los requisitos para la [illegible] de una cocina gourmet. Se conocen las [illegible]s de la sala, la ubicación de las puertas y [illegible] y el espacio disponible en la pared. especificar por completo lo que se va a construir hacer una lista de todos los gabinetes y aplicaciones (su fabricante, modelo, número y dimensiones). Después se podrían especificar las contrapartes (laminado, granito, etcétera), uniones de plomería, pisos y los techos. Esta lista podría constituir una especificación útil, pero no proporciona un modelo de lo que se desea Para completar el modelo se podría crear una representación tridimensional que muestre la posición

4 También se utilizan los términos *operaciones* y *métodos*

de los gabinetes y aplicaciones y las relaciones entre ellos. A partir del modelo, sería más fácil evaluar la eficiencia del flujo de trabajo (un requisito para todas las cocinas), y la apariencia estética del salón (un requisito personal, pero muy importante.)

Los modelos de análisis se construyen por una razón muy parecida a la del desarrollo de un plano de trabajo o una representación tridimensional para el caso de la cocina. Es importante evaluar cada componente del sistema en relación con los otros. Esto permite determinar cómo encajan los requisitos en esta visión y evaluar la "estética" del sistema como ha sido concebido.

7.2.4 Negociación

Dados los recursos limitados del negocio, no resulta inusual que los clientes y usuarios pidan más de lo que se puede lograr. También es relativamente común que diferentes clientes o usuarios propongan requisitos que entran en conflicto entre sí a argumentar que su versión es "esencial para nuestras necesidades especiales".

CONSEJO

En una negociación eficaz no debe haber ganador ni perdedor. Ambas partes ganan porque se solidifica un "trato" con el que los dos pueden vivir.

El ingeniero de requisitos debe conciliar estos conflictos por medio de un proceso de *negociación*. Se pide a los clientes, usuarios y otros interesados que ordenen sus requisitos y después discutan los conflictos relacionados con la prioridad. Se identifican y analizan los riesgos asociados con cada requisito (para obtener más detalles véase el capítulo 25). Se hacen "estimaciones" preliminares del esfuerzo requerido para su desarrollo y después se utilizan para evaluar el impacto de cada requisito en el costo del proyecto y sobre el tiempo de entrega. Mediante un enfoque iterativo, los requisitos se eliminan, combinan o modifican de forma que cada parte alcance cierto grado de satisfacción.

7.2.5 Especificación

En el contexto de los sistemas basados en computadora (y en software), el término especificación tiene significados diferentes para personas distintas. Una especificación puede ser un documento escrito, un conjunto de modelos gráficos, un modelo matemático formal, una colección de escenarios de uso, un prototipo o cualquier combinación de éstos.

PUNTO CLAVE

La formalidad y el formato de una especificación varían con el tamaño y la complejidad del software que se va a construir

Algunos sugieren que para una especificación se debe desarrollar y utilizar una "plantilla estándar" [SOM97] argumentan que esto conduce a que los requisitos sean presentados de una manera más consistente y por ende más entendible. Sin embargo, algunas veces es necesario ser flexible mientras se desarrolla una especificación. Respecto de sistemas grandes el mejor enfoque podría ser un documento escrito que combinara descripciones en el lenguaje natural y modelos gráficos. Por otro lado, en cuanto a productos o sistemas más pequeños, podría ser que no se necesite más que escenarios de uso, cuando dichos sistemas residan en ambientes técnicos que se comprendan bien.

La especificación es el producto de trabajo final que genera la ingeniería de requisitos. Sirve como base para las actividades de ingeniería de software subsecuentes

Describe la función y el desempeño de un sistema basado en computadora y las restricciones que regirán su desarrollo.

7.2.6 Validación

La calidad de los productos de trabajo procedentes de la ingeniería de requisitos se evalúa durante un paso de *validación*. La validación de requisitos examina la especificación para asegurar que todos los requisitos de software se han establecido de manera precisa; que se han detectado las inconsistencias, omisiones y errores y que éstos han sido corregidos, y que los productos de trabajo cumplen con los estándares establecidos para el proceso, proyecto y producto

El mecanismo primario para la validación de requisitos es la revisión técnica formal (capítulo 26). El equipo de revisión que valida los requisitos incluye ingenieros de software, clientes, usuarios y otros interesados que examinan la especificación y buscan errores en el contenido o la interpretación, áreas que tal vez requieran una clarificación, información faltante, inconsistencias (que es un problema importante cuando se desarrollan productos o sistemas grandes), conflictos entre los requisitos, o requisitos irreales (inalcanzables).

INFORMACIÓN

Lista de verificación para la validación de requisitos

Con frecuencia resulta útil examinar cada requisito frente a una serie de preguntas en forma de lista de verificación. Enseguida se presenta un pequeño subconjunto de las preguntas que deben formularse:

- ¿Los requisitos están establecidos de manera clara? ¿Estos pueden malinterpretarse?
- ¿La fuente del requisito (por ejemplo, una persona, una regulación o un reglamento) está identificada? ¿El enunciado final del requisito ha sido examinado por la fuente original o comparándolo con ella?
- ¿El requisito está restringido en términos cuantitativos?
- ¿Cuáles otros requisitos están relacionados con éste? ¿Están registrados de manera clara por medio de una matriz de referencias cruzadas u otro mecanismo?
- ¿El requisito viola alguna restricción del dominio del sistema?
- ¿El requisito se puede probar? Si es así, ¿se pueden especificar las pruebas (algunas veces llamadas criterios de validación) para ejercitar el requisito?
- ¿El requisito es rastreable para cualquier modelo de sistema que haya sido creado?
- ¿El requisito es rastreable para los objetivos generales del sistema o producto?
- ¿La especificación está estructurada de una forma que conduzca a su comprensión, referencia y traducción fácil en productos de trabajo más técnicos?
- ¿Se ha creado algún índice para la especificación?
- ¿Los requisitos asociados con el rendimiento, el desempeño y las características operacionales se han establecido de manera clara? ¿Cuáles requisitos parecen ser implícitos?

7.2.7 Gestión de requisitos

En el capítulo 6 se estableció que los requisitos para los sistemas basados en computadoras cambian y que el deseo de cambiarlos persiste durante la vida del sistema. La gestión de requisitos es un conjunto de actividades que ayudan al equipo de proyecto a identificar, controlar y rastrear los requisitos y los cambios a éstos en cual-

quier momento mientras se desarrolla el proyecto.[5] Muchas de estas actividades son idénticas a las actividades de la gestión de la configuración del software (GCS) que se tratan en el capítulo 27.

La gestión de requisitos comienza con la identificación. Cada requerimiento se asigna a un solo identificador. Una vez identificados los requisitos se desarrollan las tablas de rastreabilidad. En la figura 7.1 se muestra de manera esquemática una tabla de rastreabilidad, cada una de ellas relaciona los requisitos con uno o más aspectos del sistema o de su ambiente. Entre las muchas tablas de rastreabilidad posibles están las siguientes:

CONSEJO

Cuando un sistema es grande y complejo, la determinación de las conexiones entre los requisitos puede ser una tarea redituable. Se recomienda el uso de las tablas de rastreabilidad para facilitar un poco el trabajo

Tabla de rastreabilidad de las características. Muestra la manera en que los requisitos se relacionan con las características del sistema/producto observables para el cliente.

Tabla de rastreabilidad de la fuente. Identifica la fuente de cada requisito.

Tabla de rastreabilidad de dependencia. Indica la forma en que los requisitos están relacionados entre sí.

Tabla de rastreabilidad del subsistema. Establece categorías entre los requisitos de acuerdo con el(los) subsistema(s) que gobierna(n).

Tabla de rastreabilidad de la interfaz. Muestra la forma en que los requisitos se relacionan con las interfases internas y externas del sistema.

En muchos casos, estas tablas de rastreabilidad se mantienen como parte de la base de datos de los requisitos de forma que pueda buscárseles con rapidez para entender cómo el cambio en un requisito afectará diferentes aspectos del sistema que se construirá.

FIGURA 7.1

Tabla genérica de rastreabilidad.

Requisito \ Aspecto específico del sistema o su ambiente	A01	A02	A03	A04	A05			Aii
R01			✓		✓			
R02	✓		✓					
R03	✓			✓				✓
R04		✓			✓			
R05	✓	✓		✓				
Rnn	✓		✓					

5 La gestión formal de requisitos se inicia sólo para proyectos grandes, los cuales tienen cientos de requisitos identificables. En los proyectos pequeños esta función de la ingeniería de requisitos es bastante menos formal.

HERRAMIENTAS DE SOFTWARE

Ingeniería de requisitos

Objetivo: Las herramientas de la ingeniería de requisitos ayudan en la recopilación, [illegible]o, gestión y validación de requisitos.

[illegible]ica: La mecánica de las herramientas varía. Por lo [illegible], las herramientas de la ingeniería de requisitos [illegible]n una variedad de modelos gráficos (por [illegible]o, UML) que muestran los aspectos de información, [illegible]miento y comportamiento de un sistema. Estos [illegible]s forman la base para todas las otras actividades [illegible] proceso del software.

[illegible]ntas representativas[6]

[illegible] Systems Guide, Inc. ha preparado una lista [illegible]e completa de herramientas para la [illegible] de requisitos, ésta se puede encontrar en [illegible].systemsguild.com/Guildsite/Robs/retools.html. El modelado de requisitos se estudia en el capítulo 8 [illegible]erramientas que se presentan a continuación se [illegible] en la gestión de requisitos.

[illegible], desarrollado por Cybernetic Intelligence GMBH [illegible]easy-rm.com), construye un diccionario/glosario específico del proyecto que contiene descripciones y atributos detallados de los requisitos.

OnYourMark Pro, desarrollado por Omni-Vista (www.omni-vista.com), construye una base de datos de los requisitos, establece relaciones entre éstos, y permite a los usuarios analizar la relación entre los requisitos y los calendarios/costos.

Rational RequisitePro, desarrollado por Rational Software (www.rational.com), permite a los usuarios desarrollar una base de datos de los requisitos, representa las relaciones entre éstos y los organiza, prioriza y rastrea.

RTM, desarrollado por Integrated Chipware (www.chipware.com), es una herramienta para la descripción y rastreabilidad de requisitos que también soporta ciertos aspectos del control del cambio y gestión de las pruebas.

Se debe hacer notar que muchas tareas de la gestión de requisitos se pueden realizar con una simple hoja de cálculo o un sistema pequeño para el manejo de bases de datos.

7.3 INICIO DEL PROCESO DE LA INGENIERÍA DE REQUISITOS

En un escenario ideal, los clientes y los ingenieros de software trabajan juntos en el mismo equipo.[7] En tales casos, la ingeniería de requisitos se trata sólo de guiar conversaciones significativas con colegas que son miembros bien conocidos del equipo. Sin embargo, en la realidad a menudo es bastante diferente.

Los clientes pueden estar en una ciudad o país diferente, pueden tener sólo una idea vaga de lo que se requiere, tal vez tengan opiniones conflictivas acerca del sistema que se construirá, quizá su conocimiento técnico sea limitado y tengan un tiempo limitado para interactuar con el ingeniero de requisitos. Ninguna de estas situaciones es deseable, pero son muy comunes, y el equipo de software con frecuencia se ve obligado a trabajar dentro de las restricciones que impone esta situación.

En las secciones siguientes se examinan los pasos requeridos para iniciar la ingeniería de requisitos; es decir, para comenzar un proyecto de forma que se mantenga en movimiento hacia una solución exitosa.

6 Las herramientas mencionadas aquí son una muestra dentro de esta categoría. En la mayoría de los casos los nombres están registrados por sus respectivos desarrolladores.

7 Este enfoque se recomienda para todos los proyectos y es una parte integral de la filosofía para el desarrollo ágil de software.

7.3.1 Identificación de los interesados

PUNTO CLAVE

Un interesado es cualquiera que participe en forma directa en el sistema que se va a desarrollar u obtiene beneficios de éste.

Sommerville y Sawyer [SOM97] definen a los *interesados* como "todos aquellos que se benefician en una forma directa o indirecta del sistema que está en desarrollo". Ya se ha identificado a los sospechosos usuales: gerentes de operaciones de negocios, gerentes de producto, gente de mercadotecnia, clientes internos y externos, usuarios finales, consultores, ingenieros de producto, ingenieros de software, ingenieros de soporte y mantenimiento y otros. Cada interesado tiene una visión diferente del sistema, obtiene beneficios diferentes cuando éste se desarrolla de manera exitosa, y está abierto a diferentes riesgos si el esfuerzo de desarrollo llegara a fallar.

En el inicio el ingeniero de requisitos puede crear una lista de personas que contribuirán durante la obtención de requisitos (sección 7.4). La lista inicial crecerá conforme se establezca contacto con los interesados, ya que a cada uno de ellos se le preguntará: "¿Con quién más piensa que debería hablar?"

7.3.2 Reconocimiento de múltiples puntos de vista

Debido a que existen muchos clientes diferentes, los requisitos del sistema se explorarán desde diversos puntos de vista. Por ejemplo, el grupo de mercadotecnia está interesado en funciones y características que estimulen al mercado potencial, que hagan que el nuevo sistema sea fácil de vender. Los gerentes de negocios están interesados en un grupo de características que se puedan construir sin rebasar un presupuesto y que estén listas para llegar a nichos de mercado definidos. Los usuarios finales pueden desear características con las que estén familiarizados y sean fáciles de aprender y utilizar. Los ingenieros de software quizá estén interesados en funciones que den el soporte de la infraestructura y características más accesibles al mercado. Los ingenieros de soporte se pueden enfocar en la facilidad de mantenimiento del software.

> "Coloque a tres interesados en una habitación y pregúnteles qué tipo de sistema quieren. Es probable que obtenga cuatro o más opiniones diferentes"
>
> **Anónimo**

Cada uno de estos componentes (y otros) proporcionarán información al proceso de la ingeniería de requisitos. Conforme se recopila información desde múltiples puntos de vista, los requisitos que surjan pueden ser inconsistentes o entrar en conflicto con algún otro. El trabajo del ingeniero de requisitos es categorizar toda la información de los interesados (incluidos los requisitos inconsistentes y conflictivos) en una forma que permita a quienes tomen la decisiones elegir un conjunto de requisitos para el sistema que sean consistentes de manera interna.

7.3.3 Trabajo con respecto a la colaboración

En los capítulos previos se ha mencionado que los clientes (y otros interesados) deberían colaborar entre sí (evitando peleas insignificantes) y con los profesionales

de la ingeniería del software si se desea obtener un sistema exitoso. Pero, ¿cómo se logra esta colaboración?

El trabajo del ingeniero de requisitos es identificar áreas en común (es decir, los requisitos en los que todos los interesados están de acuerdo) y áreas de conflicto o inconsistencia (esto es, los requisitos solicitados por un interesado que entran en conflicto con las necesidades de otro). Ésta es, por supuesto, la última categoría que presenta un desafío.

INFORMACIÓN

Utilización de los "puntos de prioridad"

Una forma de resolver los conflictos entre requisitos, al mismo tiempo que se entiende importancia relativa de todos, es la utilización de de "votación" basado en puntos de prioridad. interesados reciben la misma cantidad de puntos que pueden "gastarse" en cualquier número de Se presenta una lista de requisitos y los indican la importancia relativa de cada uno (desde su punto de vista) al asignarle uno o más puntos de prioridad. Los puntos asignados no se pueden reutilizar. Una vez que los puntos de prioridad del interesado se han agotado, dicha persona no puede realizar ninguna otra acción sobre los requisitos. Los puntos totales que asignen a cada requisito todos los interesados indican la importancia general de cada requisito

La colaboración no significa, necesariamente, que los requisitos se definan por consenso. En muchos casos, los interesados colaboran al proporcionar su visión de los requisitos, pero un fuerte "campeón de proyecto" (por ejemplo, un gerente de negocios o un técnico importante) puede tomar la decisión final acerca de cuáles requisitos se aceptan.

7.3.4 Formulación de las primeras preguntas

En este capítulo se ha destacado que las preguntas formuladas al inicio del proyecto deben ser "libres de contexto" [GAU89]. El primer conjunto de preguntas libres de contexto se enfoca en el cliente y otros interesados, metas generales y en los beneficios. Por ejemplo, el ingeniero de requisitos podría preguntar:

- ¿Quién está detrás de la solicitud de este trabajo?
- ¿Quién usará la solución?
- ¿Cuál será el beneficio económico de una solución exitosa?
- ¿Existe otra fuente para la solución requerida?

Estas preguntas ayudan a identificar a todos los participantes que tendrían interés en el software que será construido. Además, estas preguntas identifican el beneficio medible de una implementación exitosa y las alternativas posibles para personalizar el desarrollo del software.

> "Es mejor saber algo de las preguntas que todo de las respuestas."
>
> James Thurber

La siguiente serie de preguntas permite que el equipo de software comprenda mejor el problema y deja que el cliente exprese sus percepciones acerca de una solución:

¿Cuáles son las preguntas que ayudan a comprender en forma preliminar el problema?

- ¿Cómo podría caracterizarse un buen resultado generado por una solución exitosa?
- ¿Cuáles problemas debería atacar esta solución?
- ¿Podría usted describir o mostrar el ambiente de negocios en el que se utilizará la solución?
- ¿Los aspectos especiales del desempeño o las restricciones afectarán la forma en que se busque la solución?

La serie final de preguntas se enfoca en la efectividad de la actividad de comunicación en sí misma. Gause y Weinberg [GAU89] las llaman las "metapreguntas" y proponen la siguiente lista abreviada:

- ¿Es usted la persona adecuada para contestar esta pregunta? ¿Sus respuestas son "oficiales"?
- ¿Mis preguntas son relevantes para su problema?
- ¿Estoy haciendo demasiadas preguntas?
- ¿Alguien más puede proporcionar información adicional?
- ¿Debería preguntarle alguna otra cosa?

> "El que pregunta es un tonto durante cinco minutos; el que no pregunta es un tonto por siempre."
>
> **Proverbio chino**

Estas preguntas (y otras) ayudarán a "romper el hielo" y a iniciar la conversación; esencial para la obtención exitosa. Pero un formato de reunión de pregunta y respuesta no es un enfoque que haya sido exitoso de manera contundente. De hecho, la sesión de preguntas y respuestas se debe usar sólo para el primer encuentro, y después se debe reemplazar por un formato de obtención de requisitos que combine elementos de resolución de problemas, negociación y especificación. En la sección 7.4 se presenta un enfoque de este tipo.

7.4 Obtención de requisitos

El formato de pregunta y respuesta descrito en la sección 7.3.4 es útil en la etapa inicial, pero no es un enfoque que haya sido exitoso de manera decisiva para una obtención de requisitos más detallada. de hecho, la sesión de preguntas y respuestas se debe usar sólo para el primer encuentro y después se debe reemplazar por un formato de obtención de requisitos que combine elementos de la resolución, elaboración, negociación y especificación del problema. En la siguiente sección se presenta un enfoque de este tipo.

7.4.1 Recopilación conjunta de requisitos

Cuando se desea estimular un esfuerzo conjunto y orientado al equipo de recopilación de requisitos, un equipo de participantes y desarrolladores trabajan juntos para identificar el problema, proponer elementos de solución, negociar diferentes enfoques y especificar un conjunto preliminar de requisitos para la solución [ZAH90].[8]

Se han propuestos muchos y diferentes enfoques para la recopilación conjunta de requisitos. Cada uno utiliza un escenario muy diferente, pero todos aplican alguna variación de las siguientes directrices básicas:

¿Cuáles son las directrices básicas para llevar a cabo una reunión conjunta de recopilación de requisitos?

- Las reuniones las dirige alguno de los asistentes, ya sea un ingeniero de software o un cliente (junto con otros participantes interesados).
- Se establecen reglas para la preparación y la participación.
- Se sugiere una agenda que sea tan formal como para cubrir todos los puntos importantes, pero tan informal como para estimular el flujo de ideas.
- Un moderador (puede ser un cliente, un desarrollador o un agente externo) controla la reunión.
- Se utiliza un "mecanismo de definición" (pueden ser hojas de trabajo, gráficos, hojas adheribles, un tablero electrónico, un mensajero electrónico o un foro virtual).
- La meta es identificar el problema, proponer elementos de solución, negociar diferentes enfoques y especificar un conjunto de requisitos de solución preliminares en una atmósfera que conduzca al cumplimiento de la meta

Para entender mejor el flujo de los eventos (conforme éstos ocurren), se presenta un escenario breve que describe la secuencia de eventos que conducen a la reunión para la recopilación de requisitos y que ocurren durante la reunión y después de ésta

> "Dedicamos mucho tiempo —la mayoría del esfuerzo del proyecto— no a la implementación ni a las pruebas, sino a tratar de decidir qué es lo que se va a construir."
>
> **Brian Lawrence**

Durante la fase de inicio (sección 7.3), las preguntas y respuestas básicas establecen el ámbito del problema y la percepción global de una solución. Fuera de estas reuniones iniciales, los participantes escriben una "solicitud de producto" de una o dos páginas. Se fijan un lugar, una hora y una fecha para la reunión y se selecciona un moderador. Los miembros del equipo de software y otras organizaciones intere-

8 A este enfoque se le llama algunas veces técnica de especificación facilitada de la aplicación (FAST, por sus siglas en inglés)

sadas son invitados a asistir. La solicitud de producto se distribuye a todos los asistentes antes de la fecha de reunión

Referencia Web

El desarrollo conjunto de aplicaciones (JAD, por sus siglas en inglés) es una técnica popular para la recopilación de requisitos. En **www.carolla.com/wp-jad.htm** se puede encontrar una buena descripción de esta técnica.

Mientras revisa la solicitud de producto en los días previos a la reunión, se pide a cada asistente hacer una lista de objetos que son parte del ambiente que rodea al sistema, otros objetos que éste producirá, y objetos que utiliza para realizar sus funciones. Además, se le pide a cada asistente que elabore una lista de los servicios (procesos o funciones) que manipulan o interactúan con los objetos. Por último, también se preparan listas de las restricciones (por ejemplo, costo, tamaño, reglas del negocio) y de los criterios de rendimiento (por ejemplo, velocidad, exactitud). Se informa a los asistentes que no se espera que las listas sean exhaustivas, sino que reflejen la percepción que cada persona tiene del sistema

Como un ejemplo,[9] considérese el fragmento de un documento previo a la reunión, escrito por una persona de mercadotecnia involucrada en el proyecto de *HogarSeguro*. Esta persona escribe la siguiente narración acerca de la función de seguridad en el hogar que será parte de *HogarSeguro*.

> Nuestra investigación indica que el mercado para los sistemas de administración del hogar está creciendo a una tasa de 40 por ciento anual. La primera función de *HogarSeguro* que saquemos al mercado debería ser la función de seguridad en el hogar. La mayoría de la gente está familiarizada con los "sistemas de alarma", por lo que dicha función sería fácil de vender
>
> La función de seguridad en el hogar protegería contra o reconocería una variedad de "situaciones" indeseables como una entrada ilegal, fuego, inundaciones, niveles de monóxido de carbono y otras. Utilizará nuestros sensores inalámbricos para detectar cada situación, el usuario podrá programarla y llamará por teléfono automáticamente a una oficina de monitoreo cuando detecte alguna situación

CONSEJO

Si un sistema o producto servirá a muchos usuarios se debe tener la completa seguridad de que los requisitos se obtienen de una muestra representativa de los usuarios. Si sólo uno de los usuarios define todos los requisitos, el riesgo de rechazo es alto

En realidad, otros podrían contribuir a este relato durante la reunión para la recopilación de requisitos, y mucha más información estaría disponible. Pero aun con información adicional, la ambigüedad podría estar presente, es probable que existieran omisiones y podrían ocurrir errores. Por ahora, la "descripción funcional" anterior será suficiente.

El equipo de recopilación de requisitos lo componen representantes de los departamentos de mercadotecnia, de ingeniería de hardware y software y de manufactura. Se utilizará un moderador externo.

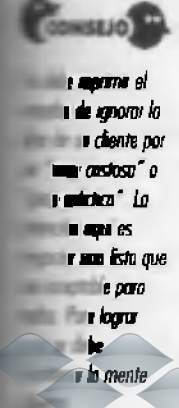
Cada persona desarrolla las listas previamente descritas. Los objetos descritos para *HogarSeguro* podrían incluir el panel de control, los detectores de humo, los sensores en puertas y ventanas, los detectores de movimiento, una alarma, un evento (cuando algún sensor se active), una pantalla, una PC, números telefónicos, una

9 El ejemplo de *HogarSeguro* (con extensiones y variaciones) se utiliza para ilustrar métodos importantes de la ingeniería del software en muchos de los capítulos que siguen. Como ejercicio, sería útil realizar una reunión para la recopilación de requisitos propia y desarrollar una serie de listas para ésta

llamada telefónica y otros. La lista de servicios podría incluir la *configuración* del sistema, la *colocación* de la alarma, el *monitoreo* de sensores, la *marcación* telefónica, la *programación* del panel de control y la *lectura* de pantalla (obsérvese que los servicios actúan sobre los objetos). De una manera similar, cada asistente elaborará listas de restricciones (por ejemplo, el sistema debe reconocer cuando los sensores no estén en funcionamiento, debe ser amigable para el usuario, debe tener una interfaz directa con la línea telefónica) y criterios de rendimiento (por ejemplo, el evento de un sensor debe ser reconocido en un segundo o menos; se debe implementar un esquema para la prioridad de los eventos).

> "Los hechos no dejan de existir sólo porque son ignorados."
>
> **Aldous Huxley**

Cuando se inicia la reunión para la recopilación de documentos, el primer tópico que se trata es la nesesidad y la justificación para el nuevo producto, todos deben estar de acuerdo en que la necesidad del producto se justifica. Una vez establecido el acuerdo, cada participante presenta sus listas para examinarlas. Las listas pueden fijarse en las paredes del salón usando hojas grandes de papel, pegarse en los muros mediante hojas adhesivas o escribirse en un pizarrón. De manera alternativa, las listas pueden haber sido colocadas en un boletín electrónico, en un sitio Web interno, o situadas dentro de un ambiente de foro de discusión (*chat room*) para revisarlas antes de la reunión. En forma ideal, cada asunto en la lista debe permitir manipularse por separado para que las listas se puedan combinar, los asuntos puedan borrarse y se les puedan realizar adiciones. En esta etapa la crítica y el debate están estrictamente prohibidos.



Después de que las listas individuales sobre el área de un tópico se hayan presentado, el grupo crea una lista combinada. Dicha lista elimina los asuntos redundantes, incorpora ideas nuevas surgidas durante el debate, pero no borra nada. Después de haberse creado las listas combinadas para todas las áreas de los distintos tópicos, el moderador coordina el debate. La lista combinada se reduce, se incrementa o replantea para reflejar de manera apropiada el producto/sistema que se desarrollará. El objetivo es desarrollar una lista consensada en el área de cada tópico (objetos, servicios, restricciones y rendimiento). Después dichas listas se integran para la acción posterior.

Cuando se completan las listas consesadas, el equipo se divide en subequipos menores; cada uno trabaja para desarrollar *miniespecificaciones* para uno o más asuntos de cada una de las listas.[10] Cada miniespecificación es una explicación concisa de la palabra o frase contenida en la lista. Por ejemplo, la miniespecificación para el objeto **Panel de control** de *HogarSeguro* podría ser:

10 En lugar de crear miniespecificaciones, muchos equipos de software eligen desarrollar escenarios del usuario llamados casos de uso. Éstos se consideran con detalle en la sección 7.5.

El **Panel de control** es una unidad empotrada en la pared con un tamaño aproximado de 9 × 5 pulgadas. El panel de control tiene conexión inalámbrica con los sensores y una PC. La interacción con el usuario ocurre por medio de un tablero de 12 teclas. Una pantalla LCD de 2 × 2 pulgadas proporciona la retroalimentación del usuario. El software provee sugerencias e imágenes interactivas y funciones similares.

Después, cada subequipo presenta sus miniespecificaciones a todos los asistentes para comentarlas. Se realizan las adiciones, anulaciones y elaboraciones posteriores. En algunos casos, el desarrollo de las miniespecificaciones descubrirá nuevos requisitos de objetos, servicios, restricciones y rendimiento que se agregarán a las listas originales. Durante los debates el equipo encontrará algún asunto que no pueda resolverse durante la reunión. Se mantendrá una lista de asuntos pendientes para que después se pueda actuar sobre estas ideas.

Después de completar las miniespecificaciones, cada asistente hace una lista de criterios de validación para el producto/sistema y la presenta al equipo. Entonces se crea una lista consensada de criterios de validación. Por último, uno o más participantes (o agentes externos) reciben el encargo de escribir las especificaciones completas mediante el uso de todas los asuntos tratados en la reunión.

HogarSeguro

Guiar una reunión para la recopilación de requisitos

El escenario: Una sala de reuniones. Se realiza la primera reunión para la recopilación de requisitos.

Los actores: Jamie Lazar, miembro del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, miembro del equipo de software; Doug Miller, miembro del equipo de software; tres miembros de mercadotecnia; un representante de ingeniería del producto; y un moderador.

La conversación:

Moderador (apuntando hacia el pizarrón blanco): Entonces ésa es la lista actual de objetos y servicios para la función de seguridad en el hogar.

Persona de mercadotecnia: Desde nuestra perspectiva, esta lista abarca toda la función.

Vinod: ¿Nadie mencionó que los usuarios querían toda la funcionalidad de HogarSeguro accesible por Internet? Eso incluiría la función de seguridad en el hogar, ¿no?

Persona de mercadotecnia: Sí, es correcto . tendremos que agregar esa funcionalidad y los objetos apropiados.

Facilitador: ¿Eso también agrega algunas restricciones?

Jamie: Sí, tanto técnicas como legales.

Representante de producción: ¿Y eso qué significa?

Jamie: Debemos estar seguros de que alguien externo no pueda entrar en el sistema, desarmarlo y robar la casa o hacer algo peor. Gran parte de la responsabilidad recae en nosotros.

Doug: Muy cierto.

Mercadotecnia: Pero aun así necesitamos la conectividad por Internet. sólo asegúrense de impedir que cualquier intruso entre.

Ed: Es más fácil decirlo que hacerlo y. .

Moderador (interrumpiendo): No quiero debatir este asunto ahora. Anotemos que es una acción que debe realizarse y continuemos.

(Doug, que lleva el registro de la reunión, hace la anotación correspondiente.)

Moderador: Siento que aún hay más cosas por considerar aquí.

(El grupo utiliza los siguientes 45 minutos en refinar y expandir los detalles de la función de seguridad en el hogar.)

7.4.2 Despliegue de la función de calidad

El *despliegue de la función de calidad* (QFD, por sus siglas en inglés) es una técnica que traduce las necesidades del cliente en requisitos técnicos para el software. El QFD "se concentra en aumentar la satisfacción del cliente desde el proceso de la ingeniería del software [ZUL92]." Para lograr lo anterior, el QFD resalta una comprensión de lo que es valioso para el cliente y después despliega estos valores durante el proceso de ingeniería. El QFD identifica tres tipos de requisitos [ZUL92]:

Requisitos normales. Reflejan los objetivos y metas establecidos para un producto o sistema durante las reuniones con el cliente. Si estos requisitos están presentes, el cliente estará satisfecho. Algunos ejemplos de requisitos normales podrían ser los tipos de gráficos en pantalla, las funciones específicas del sistema, y los niveles de rendimiento solicitados.

Requisitos esperados. Están implícitos en el producto o sistema y pueden parecer tan obvios, aunque son fundamentales, que el cliente no los establece de manera explícita. Su ausencia causaría una insatisfacción significativa. Algunos ejemplos de requisitos esperados son la facilidad de la interacción humano/máquina, corrección y confiabilidad operacional general y facilidad en la instalación del software.

Requisitos estimulantes. Reflejan las características que van más allá de las expectativas del cliente y que prueban ser muy satisfactorias cuando están presentes. Por ejemplo, un software procesador de palabras se solicita con características estándar El producto entregado contiene varias capacidades de configuración de página que son bastante satisfactorias e inesperadas

En la actualidad, el QFD abarca la totalidad del proceso de ingeniería [PAR96] Sin embargo, muchos conceptos del QFD son aplicables a la actividad de obtención de requisitos En los párrafos siguientes se presenta una visión general de dichos conceptos (adaptados para el software de computadora).

> "A menudo las expectativas fallan, y entre más lo hacen más prometen."
>
> **William Shakespeare**

En las reuniones con el cliente, la *función de despliegue* se aplica para determinar el valor de cada función que se requiere para el sistema. El *despliegue de la información* identifica los datos de los objetos y eventos que debe consumir y producir el sistema Los datos están ligados a las funciones. Por último, el *despliegue de tareas* examina el comportamiento del sistema o producto dentro del contexto de su entorno El *análisis de valor* se realiza para determinar la prioridad relativa de los requisitos determinados durante cada uno de los tres despliegues.

El QFD utiliza entrevistas y observación del cliente, sondeos y exploración de los datos históricos (por ejemplo, los reportes de problemas) como datos crudos para la actividad de recopilación de requisitos. Después, estos datos se traducen en una

tabla de requisitos —llamada la tabla de la *voz del cliente*— que se revisa con el cliente. Una variedad de diagramas, matrices y métodos de evaluación se utilizan para obtener los requisitos esperados y tratar de conseguir los requisitos estimulantes [BOS91].

7.4.3 Escenarios del usuario

Conforme se recopilan los requisitos se comienza a materializar una visión general de las funciones y características del sistema. Sin embargo, resulta difícil continuar con actividades de ingeniería del software más técnicas mientras el equipo de software no entienda la manera en que las distintas clases de usuarios finales aplicarán estas funciones y características. Para lograrlo, los desarrolladores y usuarios pueden crear un conjunto de escenarios que identifican una cadena de uso para el sistema que se va a construir. Los escenarios, llamados con frecuencia *casos de uso* [JAC92], proporcionan una descripción de cómo se usará el sistema. Los casos de uso se examinan con un mayor detalle en la sección 7.5.

HOGARSEGURO

Desarrollo de un escenario de uso preliminar

El escenario: Una sala de reuniones. Continúa la primera reunión para la recopilación de requisitos.

Los actores: Jamie Lazar, miembro del equipo de software, Vinod Raman, miembro del equipo de software, Ed Robbins, miembro del equipo de software, Doug Miller, miembro del equipo de software; tres miembros de mercadotecnia, un representante de ingeniería del producto; y un moderador.

La conversación:

Moderador: Hemos estado hablando acerca de la seguridad para el acceso a la funcionalidad de *HogarSeguro*, la cual será accesible por Internet. Me gustaría tratar algo.

Desarrollemos un escenario de uso para el acceso a la funcion de seguridad en el hogar.

Jamie: ¿Cómo?

Moderador: Podemos hacerlo de un par de formas, pero por ahora me gustaría mantener las cosas realmente informales. Dinos (y apunta a una persona de mercadotecnia) cómo visualizas el acceso al sistema.

Persona de mercadotecnia: Humm ..., bueno, es lo que haría si estuviera fuera de casa y tuviera que dejar a alguien dentro, digamos una persona de limpieza o un técnico en reparaciones, quien no tendría el código de seguridad.

Moderador (sonriendo): Ésa es la razón por la que lo harías ... dime cómo lo harías realmente.

Persona de mercadotecnia: Humm ..., la primera cosa que necesitaría es una PC. Entraría en un sitio web que tendríamos para todos los usuarios de *HogarSeguro*, introduciría mi identificación de usuario y ...

Vinod (interrumpiendo): La página web tendría que estar asegurada, codificada, para garantizar que estamos seguros y...

Moderador (interrumpiendo): Ésa es una buena información, Vinod, pero es técnica. Vamos a enfocarnos en la forma en que el usuario final usará esta capacidad, ¿de acuerdo?

Vinod: No hay problema.

Persona de mercadotecnia: Entonces, como estaba diciendo, entraría en el sitio web e introduciría mi identificación de usuario y dos niveles de contraseña.

Jamie: ¿Qué pasa si olvido mi contraseña?

Moderador (interrumpiendo): Buen punto, Jamie, pero no vamos a profundizar en eso ahora. Haremos una nota del tema y la llamaremos una "excepción". Estoy seguro que existirán otras.

Persona de mercadotecnia: Después de introducir las contraseñas, aparecerá una pantalla representando todas las funciones de SafeHome. Seleccionaría la función de seguridad en el hogar. El sistema podría requerir que yo verificara quién soy, digamos preguntándome mi dirección, teléfono o algo así.

...desplegaría una imagen del panel de control del ... de seguridad, junto con una lista de las funciones ... puedo realizar activar el sistema, desactivar el ... desactivar uno o más sensores. Supongo que ... podría permitirme reconfigurar las zonas de ... y otras cosas como ésa, pero no estoy seguro.

(Mientras la persona de mercadotecnia continúa hablando, Doug toma las notas correspondientes. Dichas notas forman la base para el primer caso de uso informal. De manera alternativa, se le pudo haber pedido a la persona de mercadotecnia que escribiera el escenario, pero esto se haría fuera de la reunión.)

7.4.4 Productos de trabajo de la obtención

Los productos de trabajo producidos como consecuencia de la obtención de requisitos variará de acuerdo con el tamaño del sistema o producto que se vaya a construir. La mayoría de los sistemas incluye los siguientes productos de trabajo:

¿Qué información genera la compilación de requisitos?

- Un enunciado de necesidad y factibilidad.
- Un enunciado limitado del ámbito del sistema o producto.
- Una lista de clientes, usuarios y otros interesados que participaron en la obtención de requisitos.
- Una descripción del ambiente técnico del sistema
- Una lista de requisitos (de manera preferente organizados por función) y las restricciones de dominio aplicables a cada uno.
- Un conjunto de escenarios de uso que proporcionen un discernimiento de la utilización del sistema o producto en diferentes condiciones de operación.
- Cualesquiera prototipos desarrollados para definir de mejor forma los requisitos

Cada uno de estos productos de trabajo los revisa toda la gente que ha participado en la obtención de requisitos.

7.5 DESARROLLO DE CASOS DE USO

...CLAVE

...el punto ...actor, ...papel ...a las ...cuando ...con el

En un libro que analiza la manera de escribir casos de uso eficaces, Alistair Cockburn [COC01] menciona que "un caso de uso captura un contrato... [que] describe el comportamiento del sistema en diferentes condiciones mientras éste responde a la petición de uno de sus usuarios". En esencia, un *caso de uso* cuenta una historia estilizada de la manera en que un usuario final (el cual desempeña uno de varios papeles posibles) interactúa con el sistema en un conjunto específico de circunstancias. La historia puede ser un texto narrativo, un esquema de tareas o interacciones, una descripción basada en una plantilla o una representación por medio de diagramas. Sin importar su forma, un caso de uso muestra el software o sistema desde el punto de vista del usuario final.

El primer paso al escribir un caso de uso consiste en definir el conjunto de "actores" que estarán involucrados en la historia. Los *actores* son las diferentes personas

(o dispositivos) que utilizan el sistema o producto dentro del contexto de la función y el comportamiento que se describirá. Los actores representan los papeles que juegan las personas (o dispositivos) conforme el sistema opera. Definido de una manera más formal, un actor es algún elemento que se comunica con el sistema o producto y que es externo al sistema en sí mismo. Cada actor tiene una o más metas cuando utiliza el sistema.

Es importante señalar que un actor y un usuario final no son necesariamente lo mismo. Un usuario típico puede desempeñar varios papeles al usar un sistema, mientras que un actor representa una clase de entidad externa (con frecuencia, pero no siempre, una persona) que desempeña sólo un papel en el contexto del caso de uso. Como un ejemplo, considérese al operador de una máquina (un usuario) que interactúa con la computadora de control para una célula de manufactura que contiene varios robots y máquinas de control numérico. Después de la revisión cuidadosa de los requisitos, el software para la computadora de control requiere cuatro diferentes modos (actores) para su interacción: modo de programación, modo de prueba, modo de monitoreo y modo de resolución de problemas. Por lo tanto, se pueden definir cuatro actores: el programador, el que realiza las pruebas, el que monitorea y el que resuelve los problemas. En algunos casos el operador de la máquina puede desempeñar todos estos papeles. En otras situaciones, son personas diferentes las que pueden desempeñar el papel de cada actor.

Referencia Web

www.rational.com/products/whitepapers/100622.jsp puede bajarse un excelente documento sobre los casos de uso

Como la obtención de requisitos es una actividad evolutiva, no todos los actores se identifican durante la primera iteración. Durante ésta es posible identificar los actores primarios [JAC92], mientras que los actores secundarios se identifican conforme se aprende más acerca del sistema. Los *actores primarios* interactúan para lograr la función requerida del sistema y obtienen el beneficio que se espera de éste. Ellos trabajan de manera directa y frecuente con el software. Los actores secundarios dan soporte al sistema de manera que los actores primarios puedan hacer su trabajo.

Ya identificados los actores pueden desarrollarse los casos de uso. Jacobson [JAC92] sugiere varias preguntas[11] que se deberían contestar mediante un caso de uso.

¿Qué se necesita saber para desarrollar un caso de uso eficaz?

- ¿Quién(es) es(son) el(los) actor(es) primario(s)?
- ¿Cuáles son las metas del actor?
- ¿Cuáles son las condiciones previas que deben existir antes de comenzar la historia?
- ¿Cuáles son las tareas o funciones principales que realiza el actor?
- ¿Cuáles excepciones podrían considerarse mientras se describe la historia?
- ¿Cuáles son las variaciones posibles en la interacción del actor?

11 Las preguntas de Jacobson se han extendido para proporcionar una visión más completa del contenido del caso de uso.

- ¿Cuál es la información del sistema que el actor adquirirá, producirá o cambiará?
- ¿El actor tendrá que informar al sistema acerca de cambios en el medio ambiente externo?
- ¿Cuál es la información que el actor desea del sistema?
- ¿El actor quiere ser informado acerca de cambios inesperados?

Como se recordará, los requisitos básicos de *HogarSeguro* definen tres actores: el **propietario de la casa** (un usuario), un **administrador de la configuración** (probablemente la misma persona que el **propietario**, pero en una función diferente), los **sensores** (dispositivos agregados al sistema), y el **subsistema de monitoreo** (la estación central que monitorea la función de seguridad en el hogar donde está instalado *HogarSeguro*). Para los propósitos de este ejemplo sólo se considera al actor **propietario**. Éste interactúa con la función de seguridad en el hogar en diferentes formas mediante el uso el panel de control de la alarma o una PC:

- Ingresa una contraseña para permitir todas las demás interacciones.
- Indaga acerca del estatus de una zona de seguridad.
- Indaga acerca del estatus de un sensor.
- Presiona el botón de pánico en caso de emergencia.
- Activa/desactiva el sistema de seguridad.

Si se considera la situación en la cual el propietario utiliza el panel de control, el caso de uso básico para la activación del sistema se presenta de la siguiente manera:[12]

FIGURA 7.2
Panel de control de HogarSeguro.

12 Nótese que este caso de uso difiere de la situación en la cual se entra en el sistema a través de Internet. En este caso, la interacción se lleva a cabo por medio del panel de control, el acceso es diferente que cuando se utiliza una PC.

1 El propietario observa el panel de control de *HogarSeguro* (figura 7.2) para determinar si el sistema está listo para entrar. Si el sistema no está listo se despliega un mensaje de no listo sobre la pantalla LCD, y el propietario debe cerrar en forma física puertas y ventanas para que el mensaje desaparezca. (Un mensaje de no listo implica que un sensor se encuentra abierto; es decir, que una puerta o ventana está abierta.)

2 El propietario utiliza el teclado para introducir una contraseña de cuatro dígitos. La contraseña se compara con la clave almacenada en el sistema. Si la contraseña es incorrecta, el panel de control emitirá un sonido y se reiniciará para recibir otra entrada. Si la contraseña es correcta, el panel de control esperará la siguiente acción.

3 El propietario selecciona e introduce en *casa* o *salida* (véase la figura 7.2) para activar el sistema. En casa activa sólo los sensores del perímetro (los sensores para la detección de movimiento interno se desactivan). Salida activa todos los sensores.

4. Cuando se realiza la activación, el propietario puede observar una luz roja de alarma.

El caso de uso básico presenta una historia de alto nivel que describe la interacción entre el actor y el sistema.

En muchas ocasiones, los casos de uso tienen una mayor elaboración para proporcionar más detalles acerca de la interacción. Por ejemplo, Cockburn [COC01] sugiere la siguiente plantilla para las descripciones detalladas de los casos de uso.

CONSEJO

Con frecuencia, los casos de uso se escriben de manera informal. Sin embargo, se recomienda el uso de la plantilla mostrada aquí para asegurar que se consideren todos los aspectos clave.

Caso de uso:	*Inicio de monitoreo*
Actor primario:	Propietario de la casa.
Meta en el contexto:	Establecer el sistema para monitorear los sensores cuando el propietario salga de la casa o permanezca dentro ella
Condiciones previas:	El sistema ha sido programado para una contraseña y reconocer diferentes sensores.
Activador:	El propietario decide "iniciar" el sistema, es decir, encender las funciones de alarma.

Escenario:

1 Propietario: observa el panel de control.
2 Propietario: introduce la contraseña.
3. Propietario: selecciona "en casa" o "salida".
4 Propietario: observa la luz roja de alarma para indicar que *HogarSeguro* está en operación.

Excepciones:

1 El panel de control *no está listo*: el propietario verifica todos los sensores para determinar cuáles están abiertos; los cierra.
2 La contraseña es incorrecta (el panel de control emite un sonido): el propietario introduce de nuevo la contraseña correcta.
3 La contraseña no es reconocida: debe contactarse el subsistema de monitoreo y respuesta para reprogramar la contraseña.
4. Se selecciona *en casa*: el panel de control emite un sonido doble y se enciende la luz de *en casa*; se activan los sensores del perímetro.

5 Se selecciona *salida*· el panel de control emite un sonido triple y se enciende la luz de *salida*; se activan todos los sensores

Prioridad: Esencial, debe implementarse.

Disponible desde: El primer incremento.

Frecuencia de uso: Muchas veces al día

Canal hacia el actor: A través de la interfaz del panel de control.

Actores secundarios: Técnicos de soporte, sensores.

Canales hacia los actores secundarios:

Técnico de soporte· línea telefónica

Sensores. interfaces alámbricas e inalámbricas

Asuntos pendientes:

1. ¿Debería haber una forma de activar el sistema sin el uso de una contraseña o con una clave abreviada?
2. ¿El panel de control debería desplegar otros mensajes de texto?
3. ¿De cuánto tiempo dispone el propietario para introducir la contraseña desde el momento en que presiona la primera tecla?
4. ¿Existe alguna forma de desactivar el sistema antes de que éste se active en realidad?

PUNTO CLAVE

Los asistentes pueden [illegible] cada caso de [illegible] asignarle su [illegible] relativo.

Los casos de uso para las otras interacciones del **propietario** se desarrollarían de una manera similar. Es importante señalar que cada caso de uso debe revisarse con cuidado. Si algún elemento de la interacción es ambiguo, existe la posibilidad de que una revisión del caso de uso descubra el problema

HOGARSEGURO

Desarrollo de un diagrama de alto nivel para un caso de uso

El escenario: Una sala de [illegible]nes. Continúa la reunión para la recopilación de [illegible]quisitos.

Los actores: Jamie Lazar, miembro del equipo de [illegible]ware; Vinod Raman, miembro del equipo de software; [illegible] Robbins, miembro del equipo de software; Doug [illegible], miembro del equipo de software; tres miembros de [illegible]rodotecnia; un representante de ingeniería del producto; y un moderador

La conversación:

[illegible]rador: Hemos invertido bastante tiempo [illegible] acerca de la función de seguridad en el hogar [illegible] *HogarSeguro*. Durante el descanso dibujé un diagrama de caso de uso para resumir los escenarios [illegible]s que son parte de esta función. Denle un vistazo [illegible] los asistentes ven la figura 7.3)

Jamie: Apenas estoy comenzando a aprender la notación del UML. Entonces, ¿la función de seguridad en el hogar la representa la caja grande con los óvalos en su interior? ¿Y los óvalos representan los casos de uso que escribimos en texto?

Moderador: Sí. Y las figuras pegadas representan actores, es decir, las personas o cosas que interactúan con el sistema según se describe en el caso de uso... ah, y utilicé el cuadro etiquetado para representar un actor que no es una persona, en este caso son sensores.

Doug: ¿Eso es legal en el UML?

Moderador: La legalidad no es el punto. Lo importante es comunicar la información. Yo pienso que el uso de una figura que parece una persona pero que representa un dispositivo puede confundirnos. Entonces he adoptado un poco las cosas. No creo que esto represente un problema

Vinod: De acuerdo, entonces tenemos narrativas de casos de uso para cada uno de los óvalos ¿Necesitamos desarrollar narrativas más detalladas con base en plantillas? He leído acerca de ellas
Moderador: Probablemente, pero eso puede esperar hasta que hayamos considerado otros funciones de *HogarSeguro*.

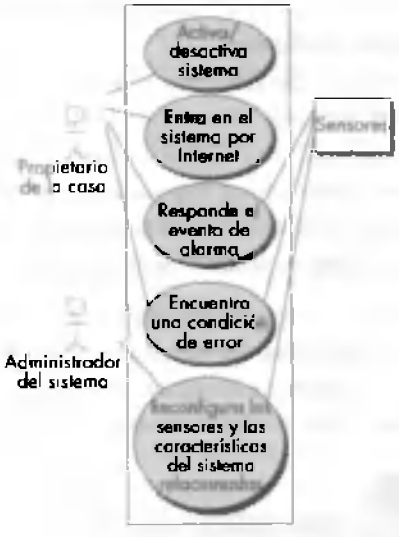
Persona de mercadotecnia: Esperen, he estado viendo este diagrama, y de repente me di cuenta que hemos olvidado algo

Moderador: ¿De verdad? ¿Qué es lo que olvidamos?

(La reunión continúa)

FIGURA 7.3

Diagrama de caso de uso para la función de seguridad en el hogar de *HogarSeguro*.



HERRAMIENTAS DE SOFTWARE

Desarrollo de casos de uso

Objetivo: Ayuda en el desarrollo de casos de uso al proporcionar plantillas que sólo requieren el llenado de espacios en blanco, para así crear casos de uso eficaces. La mayoría de las funcionalidades para los casos de uso están incluidas en un conjunto de funciones más amplio para la ingeniería de requisitos

Herramientas representativas[13]

Clear Requirement Workbench, desarrollado por LiveSpecs Software (www.livespecs.com), proporciona soporte automatizado para la creación y evaluación de casos de uso, así como una variedad de otras funciones para la ingeniería de requisitos

La vasta mayoría de las herramientas para el modelado del análisis, basadas en UML, proporcionan soporte gráfico y en texto para el desarrollo y modelado de casos de uso.

Objects by Design, una fuente para herramientas de UML (www.objectsbydesign.com/tools/umltools_byCompany.html) proporciona vínculos completos para conocer herramientas de este tipo.

En UseCases.org (www.usecases.org) se puede encontrar una variedad de plantillas para desarrollar casos de uso, así como una base de datos para soportarlos

13 Las herramientas mencionadas aquí son una muestra de esta categoría. En la mayoría de los casos los nombres están registrados por sus respectivos desarrolladores

7.6 CONSTRUCCIÓN DEL MODELO DE ANÁLISIS

El objetivo del modelo de análisis es describir los dominios requeridos de información, funcionamiento y comportamiento para un sistema basado en computadoras. El modelo cambia en forma dinámica conforme los ingenieros de software aprenden más acerca del sistema que se va a construir y los interesados entienden mejor lo que necesitan. Por esta razón el modelo de análisis es una representación de los requisitos en un momento determinado, por lo que se espera que éste cambie.

Conforme el modelo de análisis evoluciona, ciertos elementos se volverán relativamente estables, por lo que proporcionarán una base sólida para las tareas de diseño que siguen. Sin embargo, otros elementos del modelo pueden ser más volátiles, lo que indicará que el cliente aún no entiende por completo los requisitos para el sistema.

El modelo de análisis y los métodos utilizados para construirlo se describen con detalle en el capítulo 8. En las secciones siguientes se presenta una breve visión general.

7.6.1 Elementos del modelo de análisis

Existen muchas maneras de buscar los requisitos para un sistema basado en computadora. Algunas personas involucradas con el software dicen que lo mejor es seleccionar un modo de representación (por ejemplo, el caso de uso) y aplicarlo sin tomar en cuenta todos los modos restantes. Otros profesionales creen que resulta valioso utilizar varios modos de representación para mostrar el modelo de análisis. Las diferentes formas de representación obligan al equipo de software a considerar los requisitos desde distintos puntos de vista, un enfoque que tiene mayores probabilidades de descubrir omisiones, inconsistencias y ambigüedades.

Los elementos específicos del modelo de análisis los determina el método de modelado que se utilice (capítulo 8). Sin embargo, existe un conjunto de elementos genéricos común a la mayoría de los modelos de análisis:

Elementos basados en escenarios. El sistema se describe, desde el punto de vista del usuario, por medio de un enfoque basado en escenarios. Por ejemplo, los casos de uso básicos y sus correspondientes diagramas de caso de uso (figura 7.3) evolucionan para convertirse en casos de uso más elaborados basados en plantillas. Los elementos del modelo de análisis basados en escenarios con frecuencia son los primeros que se desarrollan durante la elaboración del modelo. Por tal motivo, sirven como una entrada para la creación de otros elementos de modelado.

CONSEJO

Siempre es una buena idea involucrar a los interesados. Una de las mejores formas de hacerlo es pedirle a cada uno que elabore casos de uso que describan la forma en que se utilizará el software.

Un enfoque algo diferente dentro del modelado basado en escenarios muestra las actividades o funciones que han sido definidas como parte de la tarea de obtención de requisitos. Estas funciones existen dentro de un contexto de procesamiento. Esto es, la secuencia de actividades (también se pueden utilizar los términos funciones u operaciones) que describe el procesamiento dentro de un contexto limitado se define como parte del modelo de análisis. Como la mayoría de los elementos del modelo de análisis (y otros modelos de la ingeniería de software), las actividades (funcio

nes) se pueden representar en muchos grados diferentes de abstracción. Los modelos en esta categoría pueden definirse de manera iterativa. Cada iteración proporciona detalles adicionales del procesamiento. Como un ejemplo, en la figura 7.4 se presenta un diagrama de actividad en UML para la obtención de requisitos.[14] Se muestran tres niveles de elaboración.

FIGURA 7.4

Diagramas de actividad para la obtención de documentos.

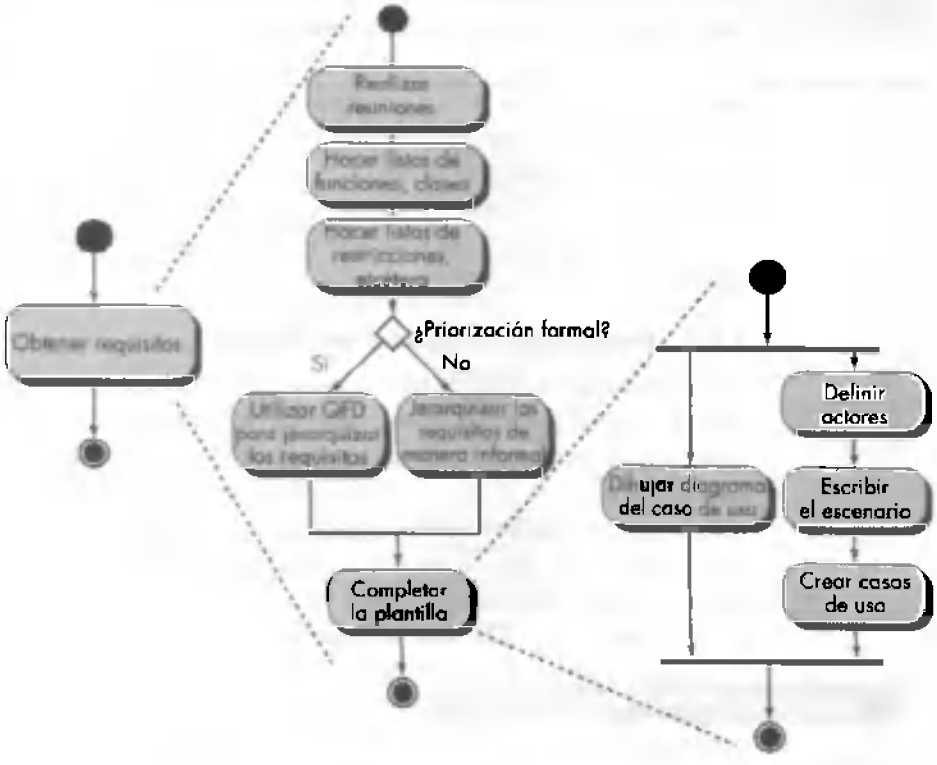


FIGURA 7.5

Diagrama de clase para el Sensor.

14 El diagrama de actividad es bastante parecido al diagrama de flujo: un diagrama gráfico para representar las secuencias y lógica del flujo de control (capítulo 11).

Elementos basados en clases. Cada escenario de uso implica un conjunto de "objetos" que se manipula mientras un actor interactúa con el sistema. Estos objetos se clasifican en clases: una colección de clases con atributos similares y comportamientos en común. Por ejemplo, se puede usar un diagrama de clase para mostrar una clase de **Sensor** para la función de seguridad de *HogarSeguro* (figura 7.5). Obsérvese que el diagrama lista los atributos de los sensores (por ejemplo, **nombre/id**, **tipo**) y las operaciones [por ejemplo, *identificar( ), habilitar( )*] que pueden aplicarse para modificar dichos atributos. Además de los diagramas de clase, otros elementos del modelado del análisis muestran la forma en que las clases colaboran con uno y otro y las relaciones e interacciones entre las clases. Lo anterior se examina con mayor detalle en el capítulo 8.

Elementos de comportamiento. El comportamiento de un sistema basado en computadora puede tener un profundo efecto sobre el diseño que se elija, así como en el enfoque de implementación que se aplique. Por lo tanto, el modelo de análisis debe proporcionar elementos de modelado que muestren el comportamiento.

El diagrama de estado (capítulo 8) es un método para representar el comportamiento de un sistema al mostrar sus estados y los eventos que ocasionan que dicho sistema cambie de estado. Un estado es cualquier forma de comportamiento observable. Además, el diagrama de estado indica las acciones (por ejemplo, la activación del proceso) que se toman como consecuencia de un evento particular.

Para *ilustrar un diagrama* de estado, considérese el estado de lectura de comandos de una fotocopiadora de oficina. En la figura 7.6 se presenta el diagrama de estado correspondiente en UML. Un rectángulo redondeado representa un estado. El rectángulo se divide en tres áreas: 1) el nombre del estado (por ejemplo, Lectura de comandos), 2) las *variables de estado* que indican la manera en que el estado se

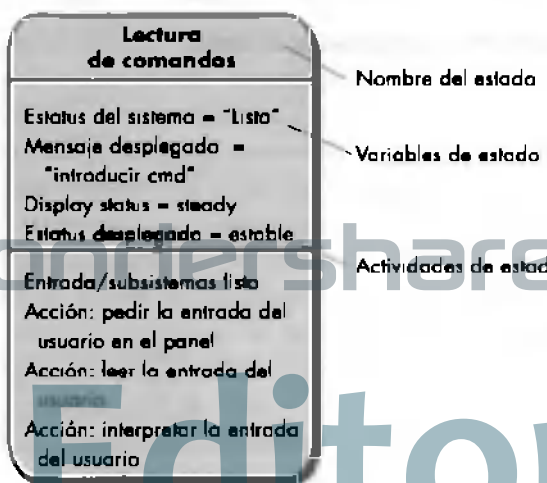


Figura 7.6 Notación en UML para el diagrama de estado.

HogarSeguro

Modelado preliminar del comportamiento

El escenario: Una sala de reuniones. Continúa la reunión para la recopilación de requisitos.

Los actores: Jamie Lazar, miembro del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, miembro del equipo de software; Doug Miller, miembro del equipo de software, tres miembros de mercadotecnia; un representante de ingeniería del producto; y un moderador.

La conversación:

Moderador: Estamos a punto de terminar de hablar acerca de la funcionalidad de seguridad en el hogar de *HogarSeguro*. Pero antes de hacerlo, me gustaría discutir el comportamiento de la función.

Persona de mercadotecnia: Yo no entiendo lo que quieres decir con comportamiento.

Ed (risenda): Es cuando le das al producto un "tiempo fuera" si se comporta mal.

Moderador: No exactamente. Déjenme explicarles. [El moderador les explica al equipo de recopilación de requisitos los conceptos básicos del modelado del comportamiento.]

Persona de mercadotecnia: Esto parece un poco técnico, no sé si yo pueda ayudar en esta parte.

Moderador: Seguro que puedes. ¿Cuál es el comportamiento que observas desde el punto de vista del usuario?

Persona de mercadotecnia: Eh... bueno, el sistema estará monitoreando los sensores, estará leyendo comandos del propietario de la casa. Desplegará su estatus.

Moderador: Ves, tú puedes hacerlo.

Jamie: También revisará la PC para determinar si existe alguna entrada desde ahí, por ejemplo un acceso basado en Internet o información de configuración.

Vinod: Sí, de hecho, la configuración del sistema es un estado por derecho propio.

Doug: Me parece que van muy rápido. Vamos a pensarlo un poco más... ¿Existe alguna forma de hacer un diagrama de esto?

Moderador: Sí existe, pero vamos a posponerlo para después de la reunión.

manifiesta a sí mismo en el mundo exterior, y 3) las *actividades de estado* que indican la forma en que se ingresa al estado (entrada/) y las acciones (do:) invocadas mientras se permanece en el mismo.

Elementos orientados al flujo. Cuando la información fluye a través de un sistema basado en computadora, ésta se transforma. El sistema acepta la entrada en una variedad de formas, aplica funciones para transformarla y produce una salida también en formas diferentes. La entrada puede ser una señal de control que transmite un transductor, una serie de números que teclea un operador humano, un paquete de información transmitido a través de una liga de red, o un voluminoso archivo de datos obtenido de un almacenamiento secundario. La transformación puede incluir una sola comparación lógica, un algoritmo numérico complejo o un enfoque de interferencia de reglas perteneciente a un sistema experto. La salida puede encender una sola luz de LED o producir un reporte de 200 páginas. En efecto, es posible crear un modelo de flujo para cualquier sistema basado en computadora, sin importar su tamaño o complejidad. En el capítulo 8 se presenta una exposición más detallada del modelado del flujo.

7.6.2 Patrones de análisis

Cualquiera que lleve a cabo ingeniería de requisitos en más de unos cuantos proyectos de software comienza a darse cuenta que ciertas cosas suceden de manera recurrente en todos los proyectos dentro de un dominio de aplicación específico.[15] Éstos pueden denominarse *patrones de análisis* [FOW97] y representan algo (por ejemplo, una clase, una función o un comportamiento) dentro del dominio de aplicación que puede reutilizarse al modelar muchas aplicaciones.

Geyer-Shultz y Hahsler [GEY01] sugieren dos beneficios que pueden asociarse con el uso de patrones de análisis:

> Primero, los patrones de análisis aceleran el desarrollo de modelos de análisis abstractos que capturan los requisitos principales del problema concreto al proporcionar modelos reutilizables del análisis, los cuales incluyen ejemplos, así como una descripción de las ventajas y limitaciones. Segundo, los patrones de análisis facilitan la transformación del modelo de análisis en un modelo de diseño al sugerir patrones de diseño y soluciones confiables para problemas comunes.

Los patrones de análisis se integran al modelo respectivo mediante una referencia al nombre del patrón. Éstos también se encuentran almacenados en un depósito para que los ingenieros de requisitos puedan utilizar los servicios de búsqueda y así encontrarlos y reutilizarlos.

Información Web
/ se puede y recursos patrones

La información acerca de un patrón de análisis se presenta en una plantilla estándar que tiene la siguiente forma[GEY01]:[16]

¿Existe alguna plantilla recomendada para describir patrones?

> **Nombre del patrón:** un descriptor que captura la esencia del patrón. El descriptor se utiliza dentro del modelo de análisis cuando se hace alguna referencia al patrón.
>
> **Intención:** describe aquello que el patrón pretende lograr o representar o el problema que ataca dentro del contexto de un dominio de aplicación.
>
> **Motivación:** un escenario que ilustra la forma en que el patrón se puede utilizar para atacar el problema.
>
> **Fuerzas y contexto:** una descripción de los aspectos externos (fuerzas externas) capaces de afectar la manera en que se utiliza el patrón, así como de los asuntos externos que serán resueltos cuando se aplique el patrón. Los aspectos externos pueden incluir cuestiones relacionadas con los negocios, restricciones técnicas externas, y asuntos relacionados con las personas.
>
> **Solución:** una descripción de la forma en que se aplica el patrón para resolver el problema, poniendo especial atención en los aspectos estructurales y de comportamiento.
>
> **Consecuencias:** se enfoca en lo que sucede cuando se aplica el patrón y en los cambios que se producen durante su aplicación.

15 En algunas situaciones las cosas se repiten sin importar el dominio de aplicación. Por ejemplo, las características y funciones de las interfases del usuario son comunes, independientemente del dominio de aplicación que se considere.

16 En la bibliografía se ha propuesto una variedad de plantillas de patrón. Los lectores interesados pueden consultar [FOW97], [BUS96] y [GAM95], entre muchas otras fuentes.

Diseño: examina la manera en que el patrón de análisis se puede lograr por medio de patrones de diseño conocidos.

Usos conocidos: ejemplos de usos en sistemas reales.

Patrones relacionados: uno o más patrones de análisis que están relacionados con el patrón en cuestión, porque el patrón de análisis 1) por lo general se utiliza junto con el patrón en estudio, 2) es similar en el sentido estructural a dicho patrón, 3) es una variación del mismo.

En el capítulo 8 se presentan ejemplos de patrones de análisis, así como otros análisis de este tópico.

INFORMACIÓN

Patrones

Los patrones se pueden ver en casi cualquier actividad de la vida diaria

Considérense las películas de acción y aventuras –de manera más específica las películas policiacas de acción y aventuras con matices de comedia–. Se pueden definir patrones para el *HéroeyCompañero*, *CapitánJefedelHéroe*, *CriminalconCorazón* y muchos más

Por ejemplo, el *CapitánJefedelHéroe* de manera invariable es más viejo, usa corbata (el héroe no), les grita en forma constante al *HéroeyCompañero*, usualmente es quien da el perfil cómico, o puede usarse en un papel más malévolo para poner trabas burocráticas o intereses personales en el camino del *HéroeyCompañero*. Se ha establecido un patrón dramático.

Para un ejemplo algo más técnico considérese un teléfono celular. Los siguientes patrones son obvios: *Llamar*, *BuscarNúmero*, *VerMensajes* entre muchos otros. Cada uno de estos patrones puede describirse una vez y después reutilizarse en el software para cualquier teléfono celular

7.7 NEGOCIACIÓN DE REQUISITOS

En un contexto ideal de la ingeniería de requisitos, las tareas de inicio, obtención y elaboración determinan los requisitos con el suficiente detalle como para emprender los pasos subsecuentes de la ingeniería del software. Desgraciadamente, esto sucede muy rara vez. En realidad, el cliente y el desarrollador entran en un proceso de *negociación*, en el cual se le debe pedir al cliente un balance de la funcionalidad, el rendimiento y otras características del sistema o producto frente al costo y el tiempo de colocación en el mercado. El objetivo de esta negociación es desarrollar un plan de proyecto que satisfaga las necesidades del cliente al mismo tiempo que refleja las restricciones del mundo real (por ejemplo, tiempo, gente, presupuesto) a las que está sometido el equipo de software.

> "Un acuerdo es el arte de dividir un pastel de tal forma que cada uno piense que se quedó con la rebanada más grande."
>
> **Ludwig Erhard**

Las mejores negociaciones son aquellas que buscan un resultado del tipo "ganar-ganar".[17] Esto es, el cliente gana al obtener el sistema o producto que satisface la

17 Se han escrito docenas de libros sobre las aptitudes para la negociación (por ejemplo, [LEW00], [FAR97], [DON96]). Ésta es una de las competencias más importantes que un ingeniero o gerente de software joven (o no tan joven) puede aprender. Se recomienda leer al menos uno de los libros mencionados

mayoría de sus necesidades, y el equipo de software gana al trabajar con presupuestos y límites de tiempo realistas y alcanzables.

Bohem [BOE98] define un conjunto de actividades de negociación en el inicio de cada iteración del proceso del software. En lugar de la actividad sencilla de comunicación con el cliente, se definen las siguientes actividades:

1. Identificación de los interesados clave en el sistema o subsistema.
2. Determinación de las "condiciones ganadoras" de los interesados.
3. Negociación de las condiciones ganadoras de los interesados para reconciliarlas en conjunto de condiciones del tipo ganar-ganar para todos los involucrados (incluido el equipo de software).

La conclusión exitosa de estos pasos iniciales asegura un resultado del tipo ganar-ganar, el cual se convierte en el criterio clave para continuar con las actividades subsecuentes de la ingeniería del software.

INFORMACIÓN

El arte de la negociación

El aprendizaje del arte de la negociación efectiva es una actividad que sirve a través de técnica y personal. La consideración de las directrices puede resultar muy valiosa:

1. *Reconocer que no es una competencia.* Para ser exitosa, ambos lados deben tener el sentimiento de haber ganado o logrado algo. Las dos partes tendrán que llegar a un acuerdo.
2. *Diseñar una estrategia.* Decidir que es lo que se desearía lograr; qué es lo que la otra parte quiere alcanzar, y qué es lo que se va a hacer para que ambas cosas sucedan.
3. *Escuchar de manera activa.* No se debe pensar en formular una respuesta mientras la otra parte está hablando. Es necesario escuchar. Es posible que se obtenga un conocimiento que ayudará a negociar de mejor manera la posición propia.
4. *Enfocarse en los intereses de la otra parte.* Si se quieren evitar los conflictos no se debe tomar una posición inflexible.
5. *No dejar que se vuelva personal.* Enfocarse en el problema que debe ser resuelto.
6. *Ser creativo.* Cuando existan situaciones de estancamiento no se debe tener miedo de pensar fuera de los cánones.
7. *Estar listo para pactar.* Una vez que se ha llegado a un acuerdo, no es necesario esperar: se debe pactar dicho convenio y seguir adelante.

HOGARSEGURO

El inicio de una negociación

El escenario: Oficina de Lisa después de la primera reunión para la de requisitos.

Los actores: Doug Miller, gerente de ingeniería de software y Lisa Pérez, gerente de mercadotecnia.

La conversación:

Lisa: Bien, escuché que en la primera reunión no les fue muy bien.

Doug: En realidad sí. Enviaste algunos buenos elementos a la reunión... contribuyeron bastante.

Lisa (sonriendo): Sí claro, ellos me dijeron que llegaron y que no fue una actividad muy tranquila que digamos.

Doug (riendo): Me aseguraré de quitarme la gorra de técnico la próxima vez que te visite. Mira, Lisa, yo creo que podemos tener un problema en terminar todas las funcionalidades para la función de seguridad en el hogar.

para las fechas de las que está hablando tu gerencia. Es muy pronto, ya lo sé, pero ya he estado haciendo un pequeño respaldo de la planeación y...

Lisa: Debemos tenerlo para esa fecha, Doug. ¿De cuál funcionalidad estás hablando?

Doug: Me parece que podemos sacar toda la función de seguridad en el hogar para la fecha límite, pero tendremos que retrasar el acceso por Internet hasta la segunda entrega.

Lisa: Doug, el acceso por Internet es lo que da a *HogarSeguro* la calidad de "novedoso". Vamos a construir toda nuestra campaña de mercadotecnia alrededor de esto. ¡Debemos tenerlo!

Doug: Entiendo tu situación, de verdad. El problema es que para darte acceso a Internet necesitaremos tener un sitio Web completamente seguro, ya construido y en funcionamiento. Para eso se necesita tiempo y gente. También tendremos que construir funcionalidades adicionales en la primera entrega ... no creo que lo podamos hacer con los recursos que tenemos ahora.

Lisa (frunciendo el ceño): Ya veo, pero debes encontrar una forma de hacerlo. Es crucial para las funciones de seguridad en el hogar y para las otras funciones también... las otras funciones pueden esperar hasta las siguientes entregas... estaré de acuerdo con eso.

Lisa y Doug parecen estar en un callejón sin salida, y aún así deben negociar una solución a este problema. En esta situación, ¿pueden "ganar" los dos? En el papel de un mediador, ¿cuál sería una sugerencia apropiada?

7.8 Validación de requisitos

Al crear cada elemento del modelo de análisis, éste se examina para conocer su consistencia, sus omisiones y ambigüedades. A los requisitos que representa el modelo el cliente les da jerarquía y se agrupan en paquetes de requisitos que se implementan como incrementos de software y se le entregan. Una revisión del modelo de análisis se enfoca en las siguientes preguntas:

¿Cuando se revisan los requisitos, ¿cuáles preguntas deben hacerse?

- ¿Cada requisito es consistente con el objetivo general del sistema/producto?
- ¿Todos los requisitos han sido especificados con el grado apropiado de abstracción? Esto es, ¿algunos requisitos proporcionan un grado de detalle técnico que sea inapropiado en esta etapa?
- ¿El requisito es necesario en realidad o representa una característica agregada irrelevante para el objetivo del sistema?
- ¿Cada requisito está limitado y no es ambiguo?
- ¿Cada requisito tiene una atribución? Esto es, ¿existe una fuente (por lo general, específica, individual) determinada para cada requisito?
- ¿Algunos requisitos entran en conflicto con otros?
- ¿Cada requisito es alcanzable en el ambiente técnico que recibirá al sistema producto?
- ¿Cada requisito se puede probar una vez que éste haya sido implementado?
- ¿El modelo de requisitos refleja de manera apropiada la información, la función y el comportamiento del sistema que será construido?
- ¿El modelo de requisitos se ha sometido a "partición" para que exponga en forma progresiva información más detallada acerca del sistema?

- ¿Se han usado patrones de requisitos para simplificar el modelo de requisitos? ¿Todos los patrones se han validado de manera apropiada? ¿Todos los patrones son consistentes con los requisitos del cliente?

Éstas y otras preguntas deben realizarse y contestarse para asegurar que el modelo de requisitos es un reflejo exacto de las necesidades del cliente y que proporciona una base sólida para el diseño.

7.9 Resumen

Antes de que el diseño y la construcción de un sistema basado en computadora puedan comenzar, es necesario entender los requisitos. Esto se logra realizando una serie de tareas de ingeniería de requisitos, la cual se lleva a cabo durante las actividades de comunicación con el cliente y modelado que han sido definidas para el proceso genérico del software. Los miembros del equipo de software realizan siete funciones distintas dentro de la ingeniería de requisitos: inicio, obtención, elaboración, negociación, especificación, validación y gestión.

Al inicio del proyecto el desarrollador y el cliente (así como otros interesados) establecen los requisitos básicos del problema, definen las restricciones predominantes del proyecto y especifican las características y funciones más importantes que deben estar presentes en el sistema para que éste alcance sus objetivos. Esta información es expandida y refinada durante la obtención, una actividad para la recopilación de requisitos que emplea reuniones que encabeza un moderador facilitadas, el QFD y el desarrollo de escenarios de uso.

La elaboración posterior expande los requisitos hacia un modelo de análisis; es decir, una colección de elementos del modelo basados en escenarios, en actividades y en clases, de comportamiento y orientados al flujo. En la creación de estos elementos se puede utilizar una variedad de notaciones de modelado. El modelo puede referirse a patrones de análisis, características del dominio del problema que son recurrentes a través de diferentes aplicaciones.

Cuando se identifican los requisitos y se crea el modelo de análisis, el equipo de software, el cliente y otros interesados en el proyecto negocian la prioridad, disponibilidad y costo relativo de cada requisito. El objetivo de esta negociación es desarrollar un plan de proyecto realista. Además, cada requisito y el modelo de análisis como un todo se validan contrastándolos con las necesidades del cliente para asegurar que se construirá el sistema correcto.

Referencias

[BOE98] Boehm, B. y A. Egyed, "Software Requirements Negotiation: Some Lessons Learned", en *Proc. Intl. Conf. Software Engineering*, ACM/IEEE, 1998, pp. 503-506.

[BOS91] Bossert, J. L., *Quality Function Deployment: A Practitioner's Approach*, ASQC Press, 1991.

[BUS96] Buschmann, F. *et al.*, *Pattern-Oriented Software Architecture: A System Pattern*, Wiley, 1996.

[COC01] Cockburn, A., *Writing Effective Use Cases*, Addison-Wesley, 2001.
[CRI92] Christel, M. G. y K. C Kang, "Issues in Requirements Elicitation", en *Software Engineering Institute*, CMU/SEI-92-TR-12 7, septiembre de 1992.
[DON96] Donaldson, M. C. y M Donaldson, *Negotiating for Dummies*, IDG Books Worldwide 1996.
[FAR97] Farber, D. C., *Common Sense Negociation. The Art of Winning Gracefully*, Bay Press, 1997
[FOW97] Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1977.
[GAM95] Gamma, E. *et al.*, *Design Patterns. Elements of Reusable Object-Oriented Software* Addison-Wesley, 1995.
[GAU89] Gause, D. C. y G M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.
[GEY01] Geyer-Schultz, A. y M. Hahsler, *Software Engineering with Analysis Patterns*, Technical Report 01/2001, Institut für Informationsverarbeitung und-wirtschaft, Wirschaftsuniversität Wien, noviembre de 2001, obtenido de http://wwwai.wu-wien ac.at/~hahsler/research/vir-lib_working2001/virlib/.
[JAC92] Jacobson. I., *Objetc-Oriented Software Engineering*, Addison-Wesley, 1992.
[LEW00] Lewicki, R., D. Saunders y J Minton, *Essentials of Negotiation*, McGraw-Hill, 2002
[PAR96] Pardee, W., *To Satisfy and Delight Your Costumer*, Dorset House, 1996
[SOM97] Somerville, I. y P. Sawyer, *Requirements Engineering*, Wiley, 1997.
[THA97] Thayer, R H y M Dorfman, *Software Requirements Engineering*, 2a. ed , IEEE Computer Society Press, 1997
[YOU01] Young, R , *Effective Requirements Practices*, Addison-Wesley, 2001.
[ZAH90] Zahniser, R. A., "Building Software in Groups", en *American Programmer*, vol 3. núms 7-8, julio-agosto de 1990
[ZUL92] Zultner, R., "Quality Function Deployment for Software· Satisfying Costumers", en *American Programmer*, febrero de 1992, pp. 28-41.

PROBLEMAS Y PUNTOS A CONSIDERAR

7.1. ¿Por qué varios desarrolladores de software no prestan mucha atención a la ingeniería de requisitos? ¿Se llegan a dar circunstancias en las que se puede omitir?

7.2. ¿Qué implica el "análisis de factibilidad" cuando se examina dentro del contexto de la función inicio?

7.3. A usted se le ha dado la responsabilidad de obtener requisitos de un cliente que dice es demasiado ocupado para reunirse con usted. ¿Qué debe hacer?

7.4. Exponer algunos de los problemas que pueden surgir cuando los requisitos deben obtenerse de tres o cuatro clientes diferentes.

7.5. ¿Por qué se dice que el modelo de análisis representa una foto instantánea de un sistema en el tiempo?

7.6. Suponga que ha convencido al cliente (usted es un excelente vendedor) de cada demanda que ha hecho como desarrollador ¿Eso lo convierte en un negociador experto? ¿Por qué?

7.7. Desarrollar al menos tres "preguntas de contexto libre" adicionales que pueda hacerle a algún interesado durante la fase de inicio.

7.8. A través de este capítulo se ha hecho referencia al "cliente". Describa al "cliente" para los desarrolladores de sistemas de información, para constructores de productos basados en computadora, para constructores de sistemas Debe tenerse precaución· pueden existir más clientes en este problema de lo que se imagina.

7.9. Desarrolle un "paquete" que facilite la recopilación de requisitos. El equipo debe incluir un conjunto de directrices para realizar una reunión de recopilación de requisitos y una serie de materiales que puedan utilizarse para facilitar la creación de listas y otros dispositivos que puedan ayudar en la definición de requisitos.

7.10. El profesor hará grupos de cuatro o cinco estudiantes. La mitad del grupo representará el papel del departamento de mercadotecnia, y la otra mitad, el de ingeniería del software. Lo que se pretenderá es definir los requisitos para la función de seguridad de *HogarSeguro*, descrita en este capítulo. Realizar una reunión de recopilación de requisitos mientras se utilizan las directrices presentadas en este capítulo.

7.11. Desarrolle un caso de uso para una de las siguientes actividades:

a) Hacer un retiro en cajero automático
b) Utilizar su tarjeta de crédito para una comida en un restaurante.
c) Comprar la despensa con una cuenta de cobro en línea
d) Buscar libros (sobre un tema específico) a través de una librería en línea
e) Una actividad que defina su instructor.

7.12. ¿Qué representan las "excepciones" en los casos de uso?

7.13. Explicar con brevedad cada uno de los elementos de un modelo de análisis. Indicar con qué contribuye cada elemento al modelo, cómo es que cada modelo es único y qué información general presenta cada modelo.

7.14. Describir con argumentos propios un patrón de análisis.

7.15. Con la plantilla presentada en la sección 7.6.2, sugerir uno o más patrones para una aplicación que aplique el instructor.

7.16. ¿Qué significa "ganar-ganar" en el contexto de la negociación durante la actividad de ingeniería de requisitos?

7.17. ¿Qué se cree que suceda cuando la validación de requisitos descubre un error? ¿Quién es el indicado para corregir el error?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Debido a que es primordial para la creación exitosa de cualquier sistema complejo basado en computadora, la ingeniería de requisitos se expone en una gran cantidad de libros. Hull y sus colegas (*Requirements Engineering*, Springer-Verlag, 2002), Bray (*An Introduction to Requirements Engineering*, Addison Wesley, 2002), Arlow (*Requirements Engineering*, Addison-Wesley, 2001), Gilb (*Requirements Engineering*, Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development*, Addison-Wesley, 1999) y Sommerville y Kotonya (*Requirements Engineering Proceses and Techniques*, Wiley, 1998) son sólo algunos libros dedicados a este tema. Dan Berry (http //se.uwaterloo ca/~dberry/bib html) ha publicado una amplia variedad de escritos acerca de tópicos relacionados con la ingeniería de requisitos.

Lauesen (*Software Requirements· Styles and Techniques*, Addison-Wesley, 2002) presenta una amplia muestra de notaciones y métodos para el análisis de requisitos. Weigers (*Software Requirements*, Microsoft Press, 1999) y Leffingwell y sus colegas (*Managing Software Requirements· A Unified Approach*, Addison-Wesley, 2000) presentan una colección útil de las mejores prácticas de requisitos y sugieren guías pragmáticas para casi todos los aspectos del proceso de la ingeniería de requisitos.

Robertson y Robertson (*Mastering the Requirements Process*, Addison Wesley, 1999) presentan un estudio de caso muy detallado que ayuda a explicar todos los aspectos del análisis de requisitos y el modelo de análisis de software. Kovitz (*Practical Software Requirements: A Manual of Content and Style*, Manning Publications, 1998) explica paso a paso un enfoque para el análisis de requisitos y una guía de estilo para aquellos que deben desarrollar especificaciones de requisitos. Jackson (*Software Requirements Analysis and Specification. A Lexicon of Practices, Principles and Prejudices*, Addison-Wesley, 1995) presenta una visión sugerente del tema de la A a la Z (de manera literal). Ploesch (*Assertions, Scenarios and Prototypes*, Springer-Verlag, 2003) explica técnicas avanzadas para desarrollar requisitos de software.

Windle y Abreo (*Software Requirements Using the Unified Process*, Prentice-Hall, 2002) exponen la ingeniería de requisitos dentro del contexto del proceso unificado y la notación del UML. Alexander y Steven (*Writing Better Requirements*, Addison-Wesley, 2002) presentan un breve conjunto de directrices para escribir requisitos claros, representarlos como escenarios y revisar el resultado final.

El modelado de casos de uso es a menudo el conductor de la creación de todos los demás aspectos del modelo de análisis. Bittner y Spence (*Use-Case Modeling*, Addison-Wesley, 2002) examinan el tema de manera amplia, así como Cockburn [COC01], Armour y Miller (*Advanced Use-Case Modeling: Software Systems*, Addison-Wesley, 2000), Kulak y sus colegas (*Use Cases. Requirements in Context*, Addison-Wesley, 2000), y Schneider y Winters (*Applying Use Cases*, Addison-Wesley, 1998).

En Internet se puede disponer de una amplia variedad de fuentes de información sobre análisis e ingeniería de requisitos. En el sitio web de SEPA, **http://www.mhhe.com/pressman,** se puede encontrar una lista actualizada de referencias en la red mundial que son relevantes para el análisis y la ingeniería de requisitos.

CAPÍTULO

8

MODELADO DEL ANÁLISIS

En el ámbito técnico, la ingeniería de software comienza con una serie de tareas de modelado que conducen a una especificación de requisitos y a una representación completa del diseño del software que se construirá. El *modelo de análisis*, que en realidad es una serie de modelos, es la primera representación técnica de un sistema.

En un libro sobre métodos de modelado del análisis, Tom DeMarco [DEM79] describe el proceso de la siguiente manera:

> Al observar los problemas y fallas reconocidas de la fase de análisis es necesario agregarle los siguientes objetivos:
>
> - Los productos del análisis deben tener una elevada facilidad de mantenimiento. Esto se aplica en particular al documento final [especificación de requisitos de software]
> - Los problemas de gran tamaño deben tratarse con un método efectivo de partición. La especificación del tipo de las novelas victorianas ya no sirve
> - Deben utilizarse gráficas cuando sea posible.
> - Se debe diferenciar entre consideraciones lógicas [esenciales] y físicas [de implementación].
>
> Como mínimo se necesita:
>
> - Algo que ayude a hacer una partición de los requisitos y a documentarla antes de la especificación.
> - Algunos medios para el seguimiento y evaluación de las interfases.
> - Herramientas nuevas para describir la lógica y la táctica, algo mejor que un texto narrativo.

Aunque DeMarco escribió acerca de los atributos del modelado del análisis hace más de un cuarto de siglo, sus contribuciones se siguen aplicando en la notación y los métodos modernos de modelado del análisis.

UN VISTAZO RÁPIDO

¿Qué es? La palabra escrita es un vehículo maravilloso para la comunicación, pero no es, necesariamente, la mejor forma de representar los requisitos para el software de computadora. El modelado del análisis utiliza una combinación de formatos en texto y diagramas para representar los requisitos de los datos, las funciones y el comportamiento de una manera que es relativamente fácil de entender y, aun más importante, conduce a una revisión para lograr la corrección, la integridad y la consistencia.

¿Quién lo hace? Un ingeniero de software (algunas veces llamado analista) construye el modelo empleando requisitos obtenidos del cliente.

¿Por qué es importante? Para validar los requisitos del software es necesario examinarlos desde algunos puntos de vista diferentes. El mode-

lado del análisis representa los requisitos en múltiples "dimensiones", con lo que se incrementa la probabilidad de encontrar errores, de que surjan inconsistencias y de que se descubran omisiones.

¿Cuáles son los pasos? Los requisitos de información, funcionales y de comportamiento se modelan mediante varios tipos de diagramas. El modelado basado en escenarios representa el sistema desde el punto de vista del usuario. El modelado orientado al flujo indica cómo se transforman los objetos de datos al realizarse las funciones del procesamiento. El modelado basado en clases define objetos, atributos y relaciones. El modelado del comportamiento presenta los estados del sistema y sus clases, así como el impacto de los eventos sobre sus estados. Una vez que se han creado los modelos preliminares, éstos se refinan y analizan para evaluar su calidad, integridad y consistencia. Después, el modelo de análisis final lo validan los interesados.

¿Cuál es el producto obtenido? Para el modelo de análisis es posible elegir una amplia variedad de tipos de diagramas. Cada una de estas representaciones ofrece una visión de uno o más de los elementos del modelo.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Los productos de trabajo del modelado del análisis deben revisarse en lo relativo a su corrección, integridad y consistencia. Estos deben reflejar las necesidades de todos los interesados y establecer una base desde la cual pueda conducirse el diseño.

8.1 ANÁLISIS DE REQUISITOS

El *análisis de requisitos* genera la especificación de características operacionales del software; indica la interfaz del software con otros elementos del sistema, y establece las restricciones que debe tener el software. El análisis de requisitos permite que el ingeniero de software (a veces llamado en este contexto *analista* o *modelador*) se extienda sobre requerimientos básicos establecidos durante tareas anteriores a la ingeniería de requisitos y construya modelos que representen escenarios del usuario, actividades funcionales, clases de problemas y sus relaciones, el comportamiento de las clases y el sistema y, a medida que se transforma, el flujo de datos.

PUNTO CLAVE

El modelo de análisis y la especificación de requisitos proporciona un medio para evaluar la calidad una vez que el software esté construido.

El análisis de requisitos le proporciona al diseñador de software una representación de información, función y comportamiento que puede trasladar a diseños arquitectónicos, de interfaz y en el nivel de componentes. Por último, el modelo de análisis y la especificación de requisitos ofrecen al desarrollador y al cliente los medios para evaluar la calidad una vez construido el software.

Por medio del modelado del análisis el ingeniero de software se centra primero en el *qué*, no en el *cómo*. ¿Qué objetos manipula el sistema, qué funciones debe desempeñar el sistema, qué comportamientos muestra el sistema, qué interfaces se definen y qué restricciones se aplican?[1]

En capítulos anteriores se estableció que en esta etapa tal vez no fuera posible realizar una especificación completa de requisitos. Quizá el desarrollador no esté

1 Es necesario mencionar que conforme los clientes se vuelven más refinados en el sentido tecnológico existe una tendencia hacia la especificación tanto del *cómo* como del *qué*. Sin embargo, el enfoque primario debe permanecer en el *qué*.

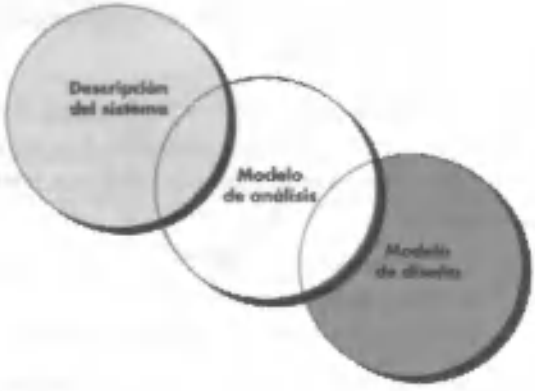


FIGURA 8.1 El modelo de análisis como un puente entre la descripción del sistema y el modelo de diseño

seguro de qué enfoque específico realizará la función y si se desempeñará de manera apropiada. Estas realidades favorecen un enfoque iterativo para el análisis y el modelado de requisitos. El analista debe modelar lo que se conoce y utilizar ese modelo como base para diseñar un incremento de software.[2]

8.1.1 Filosofía y objetivos generales

El modelo de análisis debe cumplir tres objetivos primarios: 1) describir lo que requiere el cliente, 2) establecer una base para la creación de un diseño de software, y 3) definir un conjunto de requisitos que puedan validarse una vez construido el software. El modelo de análisis llena el vacío entre una descripción al nivel de sistema (capítulo 6) —que detalla la funcionalidad general del sistema, la cual se logra al aplicar software, hardware, datos, humanos— y otros elementos del sistema y del diseño de software (capítulo 9) —que detallan la arquitectura de aplicación del software, la interfaz con el usuario y la estructura en el nivel de componentes—. Esta relación se ilustra en la figura 8.1.

> "Los problemas dignos de atacar demuestran su valor devolviendo el golpe."
>
> Piet Hein

Es importante puntualizar que algunos elementos del modelo de análisis están presentes (en un grado más alto de abstracción) en la descripción del sistema, y que esas tareas de ingeniería de requisitos en realidad comienzan como parte de la ingeniería de sistemas. Además, todos los elementos del modelo de análisis son identificables de manera directa en las partes del modelo del diseño. No siempre es posible una división clara de tareas de análisis y diseño entre estas dos importantes actividades del modelado. De modo invariable, como parte del análisis se realiza algún diseño y algún análisis se efectúa durante el diseño

2 De manera alternativa, el equipo de software puede elegir la creación de un prototipo (capítulo 3) en un esfuerzo encaminado a entender mejor los requisitos para el sistema

8.1.2 Reglas prácticas de análisis

Arlow y Neustadt [ARL02] sugieren varias reglas prácticas que deben seguirse para crear el modelo de análisis:

- *El modelo debe centrarse en los requisitos visibles dentro del problema o dominio de negocio. El grado de abstracción debe ser alto de forma relativa.* "No se debe perder tiempo en detalles" [ARL02] que tratan de explicar cómo funcionará el sistema
- *Cada elemento del modelo de análisis debe agregarse a un acuerdo general de los requisitos de software y proporcionar una visión interna del dominio de información, función y comportamiento del sistema.*
- *Debe retrasarse la consideración de la infraestructura y otros modelos no funcionales hasta el diseño.* Por ejemplo, es posible que se requiera una base de datos, pero las clases necesarias para implementarla, las funciones que se requieren para acceder a ella y el comportamiento que se exhibirá cuando se utilice debe considerarse sólo después de que se haya completado el análisis de dominio del problema.
- *Se debe minimizar el acoplamiento de todo el sistema.* Es importante representar las relaciones entre clases y funciones Sin embargo, si el nivel de "interconexión" es extremadamente alto se deben hacer esfuerzos para reducirlo
- *Se debe tener la seguridad de que el modelo de análisis proporciona valor a todos los interesados* Cada circunscripción tiene su propio uso del modelo. Por ejemplo, los interesados relacionados con los negocios deben utilizar el modelo para validar los requisitos; los diseñadores, como base para el diseño; la gente de aseguramiento de la calidad, como ayuda para planear pruebas de aceptación
- *El modelo debe mantenerse tan simple como sea posible* No se deben agregar diagramas adicionales cuando éstos no ofrezcan información nueva. No se deben utilizar formas de notación nuevas cuando basta con una simple lista

8.1.3 Análisis del dominio

Referencia Web

En www.literia.com/English/SoftwareEngineering/SE_mod5.asp pueden encontrarse muchos recursos útiles para el análisis del dominio.

Al examinar la ingeniería de requisitos (capítulo 7) se estableció que los patrones de análisis a menudo ocurren de nuevo en muchas aplicaciones dentro de un dominio de negocio específico. Si estos patrones se definen y se clasifican por categoría de una manera que permitan al ingeniero o al analista de software reconocerlos y reutilizarlos, la creación del modelo de análisis se acelera. Un factor de mayor importancia es que la probabilidad de aplicar patrones de diseño reutilizables y componentes ejecutables de software aumenta en forma sustancial. Esto ofrece tiempo al mercado y reduce los costos del desarrollo.

FIGURA 8.2 Entrada y salida para el análisis del dominio.

¿Pero, en primer lugar, cómo se reconocen los patrones de análisis? ¿Quiénes los definen, los asignan a una categoría y los preparan para aplicarlos en proyectos subsecuentes? Las respuestas a estas preguntas residen en el *análisis del dominio*. Firesmith [FIR93] describe el análisis del dominio de la siguiente manera:

> El análisis del dominio de software es la identificación, el análisis y la especificación de requisitos comunes de un dominio específico de aplicación para, de manera típica, reutilizarlo en múltiples proyectos dentro de ese dominio de aplicación [El análisis del dominio orientado a objetos es] la identificación, el análisis y la especificación de capacidades comunes reutilizables dentro de un dominio específico de aplicación, en términos de objetos, clases, subensamblajes y marcos de trabajo.

El "dominio de aplicación específico" puede variar desde aeronáutica hasta servicios bancarios, desde videojuegos en multimedia hasta software aplicado en instrumental médico. La meta del análisis o de dominio es directa: encontrar o crear aquellas clases de análisis o funciones y características comunes que se aplican ampliamente para que puedan reutilizarse.[3]

> "El gran arte del aprendizaje es entender poco a poco"
>
> **John Locke**

Referencia Web

En cierta forma, el papel de un analista de dominio es similar al de un maestro forjador de herramientas en un ambiente de manufactura pesada. El trabajo de este último es diseñar y fabricar instrumentos que puedan ser usados por mucha gente que realiza trabajos similares. El papel del analista de dominio[4] es descubrir y definir

3 Una visión complementaria del análisis del dominio "involucra el modelado del dominio de forma que los ingenieros de software y otros interesados puedan aprender más de él... no todas las clases del dominio resultan necesariamente en el desarrollo de clases reutilizables" [LET03].

4 No debe suponerse que si se cuenta con la colaboración de un analista del dominio, un ingeniero de software no tiene por qué comprender el dominio de aplicación. Todos los miembros de un equipo de software deben tener algún conocimiento del dominio en el cual se colocará el software.

patrones de análisis reutilizables, clases de análisis e información relacionada que pueda usar mucha gente en aplicaciones parecidas.

La figura 8.2 [ARA89] ilustra entradas y salidas clave para el proceso de análisis de dominio. Las fuentes de conocimiento del dominio se examinan en un intento por identificar objetos que pueden ser reutilizados a través del dominio.

8.2 Enfoques de modelado del análisis

Una visión del modelado del análisis, llamado *análisis estructurado*, considera que los datos y el proceso que transforman los datos son entidades separadas. Los objetos de datos se modelan en una forma que define sus atributos y relaciones. Los procesos que manipulan los objetos de los datos se modelan de tal manera que muestran cómo transforman los datos, mientras los objetos de datos fluyen por el sistema.

Un segundo enfoque del modelado del análisis, llamado *análisis orientado a objetos*, se centra en la definición de clases y en la manera en que éstas colaboran entre ellas para efectuar los requisitos del cliente. El UML y el proceso unificado (capítulo 3) están orientados a objetos en forma predominante.

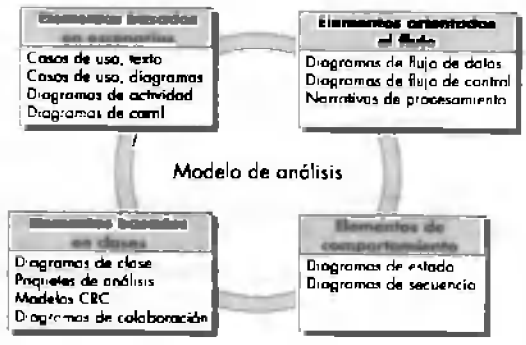
> "El [a]nálisis es frustrante, lleno de relaciones interpersonales complejos, indefinido y difícil. En pocas palabras, es fascinante. Una vez que estás engarzado, el viejo placer de la construcción de sistemas nunca será suficiente para satisfacerte."
>
> **Tom DeMarco**

Aunque el modelo de análisis propuesto en este capítulo combina características de ambos enfoques, es común que los equipos de software elijan uno y excluyan todas las representaciones del otro. El cuestionamiento no es cuál es el mejor, sino qué combinación de representaciones le proporcionará a los interesados el mejor modelo de requisitos de software y el puente más efectivo para el diseño de software.

El modelado del análisis conduce a la derivación de cada uno de los elementos de modelado mostrados en la figura 8.3. No obstante, el contenido específico de cada elemento (por ejemplo, los diagramas con que se construyen el elemento y el modelo) puede diferir de proyecto a proyecto. Como ya se ha puntualizado muchas veces en este libro, el equipo de software debe trabajar para mantenerlo simple. Sólo se deben utilizar aquellos elementos que agreguen valor al modelo.

> "¿Por qué debemos construir modelos? ¿Por qué no construimos el sistema y ya? La respuesta es que podemos construir modelos de tal forma que resaltemos o enfaticemos ciertas características críticas de un sistema, al mismo tiempo que quitamos énfasis a otros aspectos del sistema."
>
> **Ed Yourdon**

FIGURA 8.3
Elementos del modelo de análisis.



8.3 CONCEPTOS DEL MODELADO DE DATOS

Referencia Web
...información sobre ... de datos

El modelado del análisis a menudo comienza con el *modelado de datos*. El ingeniero o analista de software define todos los objetos de datos que se procesan dentro del sistema y las relaciones entre los objetos de datos, además de otra información pertinente para las relaciones.

8.3.1 Objetos de datos

Un *objeto de datos* es una representación de casi cualquier información compuesta que el software debe entender. *Información compuesta* se refiere a algo que tiene muchas propiedades o atributos diferentes. Por lo tanto, "anchura" (un valor individual) no sería un objeto de datos válido, pero las **dimensiones** (la incorporación de altura, anchura y profundidad) podrían definirse como un objeto.

¿Cómo se manifiestan a sí mismos los datos dentro del contexto de una aplicación?

Un objeto de datos puede ser una entidad externa (por ejemplo, cualquier cosa que produzca o consuma información), una cosa (por ejemplo, un reporte o un despliegue) un suceso (como una llamada telefónica) o un evento (como una alarma), un papel (por ejemplo, un vendedor), una unidad organizacional (como un departamento de contaduría), un lugar (como un almacén), o una estructura (como un archivo). Por ejemplo, una persona o un auto pueden verse como un objeto de datos en el sentido de que cualquiera de ellos puede definirse en términos de un conjunto de atributos. La descripción del objeto de datos incorpora el objeto y todos sus atributos.

PUNTO CLAVE
Un objeto de datos es una representación de información compuesta que se manipula con software.

Un objeto de datos encapsula sólo datos: no existe alguna referencia dentro de un objeto de datos a las operaciones actúen sobre los datos [5] Por lo tanto, el objeto de datos puede representarse como una tabla, tal como se muestra en la figura 8.4. Los encabezados de la tabla reflejan los atributos del objeto. En este caso, un auto se define en términos de **marca, modelo, número de serie, tipo de carrocería, color** y **propietario.** El contenido de la tabla representa ejemplos específicos del objeto de datos. Por ejemplo, un Chevy Corvette es una muestra del objeto de datos auto.

5 Esta distinción separa los objetos de datos y las clases u objetos definidos como parte del enfoque orientado a objetos.

FIGURA 8.4

Representación tabular de objetos de datos.

Nombres de atributos

Une los objetos de datos entre sí, en este caso, propietario

identificador | Atributos descriptivos | Atributos referenciales

Marca	Modelo	# de id	Tipo	Color	Propietario
Lexus	LS400	AB123...	Sedan	Blanco	RSP
Chevy	Corvette	X456...	Deportivo	Rojo	CCD
BMW	750iL	XZ765...	Coupé	Blanco	LIL
Ford	Taurus	Q12A45...	Sedán	Azul	BLF

Instancia

PUNTO CLAVE

Los atributos definen a un objeto de datos, describen sus características y, en algunos casos, hacen referencia a otro objeto

8.3.2 Atributos

Los *atributos* definen las propiedades de un objeto de datos y toman una de las tres características diferentes. Se pueden utilizar para 1) nombrar una ocurrencia del objeto de datos, 2) describir la ocurrencia o 3) hacer referencia a otra ocurrencia en otra tabla. Además, se debe definir uno o más atributos como un identificador; es decir, el atributo identificador se convierte en una "clave" cuando se desea encontrar una ocurrencia del objeto de datos. En algunos casos, los valores para el (los) identificador(es) son únicos, aunque esto no es un requisito. En referencia al objeto de datos **auto**, un identificador razonable podría ser el número de serie.

El conjunto de atributos apropiado para un objeto de datos se determina mediante la comprensión del contexto del problema. Los atributos para **auto** sirven bien para una aplicación que utilice el Departamento de vehículos de motor, pero estos atributos serían inútiles para una compañía automotriz que necesite un software para el control de fabricación. En este último caso, los atributos para auto tal vez incluirían también **número de serie, tipo de carrocería** y **color**, pero además tendrían que adicionarse muchos más atributos (como **código interior, tipo de tren de manejo, designador de paquete de ajuste, tipo de transmisión**) para hacer de **auto** un objeto significativo en el contexto de control de fabricación.

Referencia Web

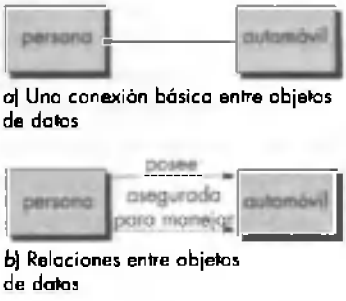
"normalización" es importante para todos aquellos que intentan realizar modelado de datos. En **www.datamodel.org** puede encontrarse una introducción útil

INFORMACIÓN

Objetos de datos y clases OO, ¿son lo mismo?

Cuando se debate acerca de los objetos de datos es común que surja una pregunta: ¿un objeto de datos es lo mismo que una clase orientada a objetos? La respuesta es: "no"

Un objeto de datos define un elemento compuesto de los datos, esto es, incorpora una colección de elementos de datos individuales (atributos) y da un nombre a la colección de elementos (el nombre del objeto de datos). Una clase OO encapsula atributos de los datos, pero también incorpora las operaciones que manipulan los datos implicados por dichos atributos. Además, la definición de clases implica una infraestructura completa que es parte del enfoque de la ingeniería de software orientado a objetos. Las clases se comunican entre sí a través de mensajes; pueden organizarse en jerarquías; proporcionan características heredadas para objetos que son una instancia para una clase



a) Una conexión básica entre objetos de datos

b) Relaciones entre objetos de datos

Figura 8.5 Relaciones entre objetos de datos.

8.3.3 Relaciones

Los objetos de datos están conectados entre sí de muchas maneras diferentes. Por ejemplo, dos objetos de datos, **persona** y **auto**, pueden representarse con la simple notación ilustrada en la figura 8.5a. Se establece una conexión entre **persona** y **auto** porque los objetos se relacionan entre sí. ¿Pero, cuáles son las relaciones? La respuesta se determina entendiendo el papel de las personas (propietarios, en este caso) y de los autos dentro del contexto del software que se construirá. Se puede definir un conjunto de parejas objeto/relación que definan las relaciones relevantes. Por ejemplo:

CLAVE Las relaciones indican cómo se relacionan los objetos de datos entre sí.

- Una persona *posee* un auto.
- Una persona *está asegurada para conducir* un auto

Las relaciones *posee* y *está asegurada para conducir* definen las conexiones relevantes entre **persona** y **auto**. En la figura 8.5b se ilustran estas parejas objeto/relación de manera gráfica. Las flechas de la figura 8.5b ofrecen información importante acerca de la direccionalidad de la relación y a menudo reducen la ambigüedad o las malas interpretaciones.

8.3.4 Cardinalidad y modalidad

Los elementos del modelado de datos —objetos de datos, atributos y relaciones— ofrecen la base para entender el dominio de información de un problema. Sin embargo, también es necesario comprender información adicional relacionada con estos elementos básicos.

Hasta este punto se ha definido un conjunto de objetos y se han representado las parejas objeto/relación que los limitan. Pero un simple par que establece que **objetoX** se *relaciona* con **objetoY** no proporciona suficiente información para los propósitos de la ingeniería del software. Se debe entender cuántas ocurrencias del **objetoX** están relacionadas con cuántas ocurrencias del **objetoY.** Esto conduce al concepto del modelado de datos llamado *cardinalidad*.

El modelo de datos debe ser capaz de representar el número de ocurrencias de los objetos en una relación dada. Tillmann [TIL93] define la cardinalidad de un par objeto/relación de la siguiente manera: "Cardinalidad es la especificación del núme-

¿Cómo se maneja una situación en la que un objeto de datos está relacionado con la ocurrencia de muchos otros objetos de datos?

ro de ocurrencias de un [objeto] que puede relacionarse con el número de ocurrencias de otro [objeto]". Por ejemplo, un objeto puede relacionarse sólo con otro objeto (una relación 1:1); un objeto puede relacionarse con muchos objetos (una relación 1:N); un número de ocurrencias de un objeto puede relacionarse con algún otro número de ocurrencias de otro objeto (una relación M:N).[6] La cardinalidad también define "el número máximo de objetos que puede participar en una relación" [TIL93]. Sin embargo, no indica si un objeto particular de datos debe participar o no en la relación. El modelo de datos agrega modalidad al par objeto/relación para especificar esta información.

INFORMACIÓN

Diagramas de entidad-relación

La pareja objeto-relación es la piedra angular del modelo de datos. Estas parejas pueden representarse de manera gráfica mediante el diagrama de entidad-relación (DER).[7] El DER lo propuso originalmente Peter Chen [CHE77] para el diseño de sistemas de bases relacionales, y después otros lo han ampliado. Con el DER se identifica un conjunto de componentes primarios: objetos de datos, atributos, relaciones e indicadores de varios tipos. El propósito primordial del DER es representar objetos de datos y sus relaciones.

Ya se ha hecho una introducción de la notación rudimentaria para el DER. Los objetos de datos se representan por medio de un rectángulo etiquetado. Las relaciones se representan mediante una línea etiquetada que conecta objetos. En algunas variaciones del DER la línea de conexión contiene un rombo que está etiquetado con la relación. Las conexiones entre objetos de datos y relaciones se establecen mediante una variedad de símbolos especiales que indican su cardinalidad y modalidad.

Para más información sobre el modelado de datos y el diagrama de entidad-relación el lector interesado puede consultar [THA00]

La *modalidad* de una relación es de 0 si no hay una necesidad explícita para que ocurra la relación o la relación es opcional. La modalidad es 1 si una ocurrencia de la relación es obligatoria.

> "Para que un sistema de información sea útil, confiable, adaptable y económico debe estar basado primero en el modelado de datos, y sólo de manera secundaria en el análisis del proceso ... porque la estructura de datos se mantiene de forma inherente a la cantidad, mientras que el proceso es relativo a la técnica."
>
> Duncan Dunsire

HERRAMIENTAS DE SOFTWARE

Modelado de datos

Objetivo: Los herramientas de modelado de datos ... de datos ... -que se ... la capacidad de representar objetos de datos, sus ... datos y otros proyectos de sistemas de información.

6 Por ejemplo, un tío puede tener muchos sobrinos y un sobrino puede tener muchos tíos.

7 Aunque el DER todavía se usa en algunas aplicaciones para el diseño de bases de datos, en la actualidad la notación en UML es la más utilizada para el diseño de datos.

...n un medio automatizado para crear ...s de entidad-relación, diccionarios de objetos de ... y modelos relacionados

...ica: Las herramientas en esta categoría permiten ... describir objetos de datos y sus relaciones. En ... casos utilizan la notación del DER; en otras ... modelan las relaciones por medio de otros ... Además permiten la creación de un modelo ... de datos al generar un esquema de base de datos ... SMBD

...tas representativas[8]

ERWin, desarrollado por Computes Associates ... com), ayuda en el diseño de objetos de datos, ... propias y elementos clave para bases de datos.

..., desarrollado por Embarcadero Software ... embarcadero.com), brinda soporte al modelado ... relación.

Oracle/Designer, desarrollado por Oracle Systems (www.oracle.com), modela procesos de negocios, entidades de datos y relaciones que se transforman en diseños a partir de los cuales se generan aplicaciones completas y bases de datos

MetaScope, desarrollado por Madrone Systems (www madronesystems.com), es una herramienta para el modelado de datos de bajo costo que da soporte a la representación gráfica de datos.

ModelSphere, desarrollado por Magna Solutions GMBH (www magnasolutions com), da soporte a una variedad de herramientas de modelado relacional.

VisibleAnalyst, desarrollado por Visible Systems (www.visible.com), da soporte a una variedad de funciones de modelado del análisis, incluido el modelado de datos.

8.4 ANÁLISIS ORIENTADO A OBJETOS

Cualquier estudio sobre el análisis orientado a objetos debería comenzar definiendo el término *orientado a objetos*. ¿Qué es un punto de vista orientado a objetos? ¿Por qué un método se considera orientado a objetos? ¿Qué es un objeto? Cuando la OO obtuvo una amplia variedad de adeptos durante las décadas de 1980 y 1990, existieron muchas opiniones diferentes (por ejemplo, [BER93], [TAY90], [STR88], [BOO86] acerca de las respuestas correctas a estas preguntas. En la actualidad ha surgido una visión coherente de la OO.

El objetivo del análisis orientado a objetos (AOO) es definir todas las clases (además de las relaciones y el comportamiento asociado con ellas) relevantes para el problema y que deben resolverse. Esto se logra llevando a cabo algunas tareas:

1. Deben comunicarse los requisitos básicos del usuario entre el cliente y el ingeniero de software
2. Deben identificarse las clases (es decir, se definen los atributos y métodos)
3. Se define una jerarquía de clases.
4. Deben representarse las relaciones de objeto a objeto (conexiones entre objetos).
5. Debe modelarse el comportamiento del objeto
6. Las tareas 1 a 5 se vuelven a aplicar de manera iterativa hasta que el modelo esté completo.

8 Las herramientas mencionadas aquí son una muestra de esta categoría. En la mayoría de los casos los nombres están registrados por sus respectivos desarrolladores.

En lugar de examinar un problema mediante un modelo más convencional del tipo entrada-procesamiento-salida (flujo de información) o un modelo derivado de forma exclusiva de las estructuras jerárquicas de información, el AOO construye un modelo orientado a las clases que se basa en la comprensión de los conceptos OO.

INFORMACIÓN

Conceptos orientados a objetos

Los conceptos orientados a objetos (OO) están bien establecidos en el mundo de la ingeniería del software. A continuación se presentan las descripciones abreviadas de conceptos OO que se encuentran con frecuencia durante el modelado del análisis. En el capítulo 10 se presentan otros objetos OO que están alineados de manera más cercana al diseño de software.

Atributos: una colección de valores de los datos que describen una clase.

Clase: encapsula los datos y las abstracciones de procedimiento requeridos para describir el contenido y el comportamiento de alguna entidad del mundo real. Dicha de otra manera, una clase es una descripción generalizada (por ejemplo, una plantilla, un patrón o un plano de trabajo) que describe una colección de objetos similares.

Objetos: instancias de una clase específica. Los objetos heredan los atributos y operaciones de una clase.

Operaciones: también llamadas *métodos y servicios* proporcionan la representación de uno de los comportamientos de una clase.

Subclase: una especialización de la superclase. Una subclase puede heredar tanto los atributos como las operaciones de una superclase.

Superclase: también llamada una *clase básica*, es una generalización de un conjunto de clases que están relacionadas con ella.

8.5 MODELADO BASADO EN ESCENARIOS

Aunque el éxito de un sistema o producto basado en computadora se mide de muchas formas, la satisfacción del usuario encabeza la lista. Si los ingenieros del software entienden la manera en que los usuarios finales (y otros actores) quieren interactuar con el sistema, el equipo de software será más capaz de caracterizar en forma apropiada los requisitos y construir modelos significativos de análisis y diseño. Por lo tanto, el modelado del análisis con UML comienza con la creación de escenarios en la forma de casos de uso, diagramas de actividad y diagramas de calles.

8.5.1 Escritura de casos de uso

Un caso de uso captura las interacciones que ocurren entre los productores y consumidores de información y del sistema en sí mismo. En esta sección se examina la forma en que se desarrollan los casos de uso como una parte de la actividad de modelado del análisis.[9]

El concepto de un caso de uso (capítulo 7) es relativamente fácil de entender. Describe un escenario de uso específico en un lenguaje directo desde el punto de

9 Los casos de uso son una parte particularmente importante del modelado del análisis que interfases con el usuario. El análisis de la interfaz se trata con detalle en el capítulo 12.

de un actor definido.[10] Pero cómo puede saberse 1) ¿sobre qué escribir? 2), ¿cuánto escribir acerca de ello? 3), ¿qué tan detallada debe ser la descripción?, y 4) ¿cómo organizar la descripción? Estas son las preguntas que deben contestarse para que los casos de uso tengan un valor como herramienta para el modelado del análisis.

> "[Los casos de uso] son simplemente una ayuda para definir lo que existe fuera del sistema (actores) y lo que debería realizar el sistema (casos de uso)."
>
> **Ivar Jacobson**

¿Sobre qué escribir? Las primeras dos tareas de la ingeniería de requisitos[11] —inicio y obtención— proporcionan la información necesaria para comenzar a escribir casos de uso. Las reuniones para la recopilación de requisitos, despliegue de la función de calidad (QFD) y otros mecanismos para la ingeniería de requisitos se utilizan para identificar a los interesados, definir el ámbito del problema, especificar las metas operativas globales, esquematizar todos los requisitos funcionales conocidos y describir las cosas (objetos) que manipulará el sistema

El desarrollo de una serie de casos de uso se comienza haciendo una lista de las funciones o actividades que realiza un actor específico. Éstas pueden obtenerse de una lista de funciones requeridas del sistema por medio de conversaciones con los clientes o usuarios finales, o mediante una evaluación de los diagramas de actividad (sección 8.5.2) desarrollados como parte del modelado del análisis

HogarSeguro

Desarrollo de otro escenario de uso preliminar

El escenario: Una sala de [illegible] durante la segunda junta para la recopilación [illegible]

Los actores: Jamie Lazar, miembro del equipo de [illegible] Ed Robbins, miembro del equipo de software, [illegible], gerente de ingeniería del software, Ires [illegible] de mercadotecnia; un representante de [illegible] de producto, y un moderador

La conversación:

Moderador: Es hora de que comencemos a hablar [illegible] de la función de vigilancia de *HogarSeguro*. [illegible] a desarrollar un escenario de usuario para el [illegible] a la función de seguridad en el hogar

Jamie: ¿Quién hace el papel del actor en esto?

Moderador: Creo que Meredith (una persona de mercadotecnia) ha estado trabajando en esa funcionalidad. ¿Por qué no haces tú el papel?

Meredith: ¿Quieres que lo hagamos igual que la última vez, no es así?

Moderador: Correcto... de la misma forma

Meredith: Bueno, es obvio que la razón para la vigilancia es permitir que el propietario esté pendiente de la casa mientras él o ella están fuera, grabar y reproducir videos que se hayan capturado [illegible] ese tipo de [illegible]

10 Un actor no es una persona específica, sino el papel que desempeña una persona (o dispositivo) dentro de un contexto específico. Un actor "llama al sistema para entregar uno de sus servicios" [COC01].

11 Estas tareas de la ingeniería de requisitos se examinan con detalle en el capítulo 7.

Ed: ¿El video será digital y se almacenará en disco?

Moderador: Buena pregunta, pero por ahora pospongamos los aspectos de implementación. ¿Meredith?

Meredith: De acuerdo, entonces básicamente hay dos partes para la función de vigilancia la primera configura el sistema, incluyendo el establecimiento de un plano de la casa –necesitamos herramientas que ayuden al propietario a hacerlo– y la segunda parte es la función de vigilancia real en sí mismo. Como el establecimiento del plano es parte de la actividad de configuración, me enfocaré en la función de vigilancia

Moderador (sonriendo): Me quitaste las palabras de la boca

Meredith: Mm Quiero tener acceso a la función de vigilancia, ya sea a través de la PC o de Internet. Siento que el acceso por Internet sería el de uso más frecuente. De cualquier manera, quiero ser capaz de desplegar vistas de las cámaras en una PC y controlar el movimiento y los acercamientos de una cámara específica. Específico la cámara seleccionada desde el plano de la casa. Quiero grabar y reproducir la salida de las cámaras de manera selectiva También quiero ser capaz de bloquear el acceso a una o más cámaras con una contraseña específica Y quiero la opción de ver pequeñas ventanas que muestren vistas de todas las cámaras y después ser capaz de seleccionar la que quiero destacar

Jamie: Esas se llaman vistas en miniatura

Meredith: Bien, entonces quiero vistas en miniatura de todas las cámaras. También quiero que la interfaz con la función de vigilancia tenga la misma apariencia que todas las otras interfases de *HogarSeguro* Quiero que sea intuitiva; es decir, que no sea necesario leer un manual para poder usarla.

Moderador: Buen trabajo, ahora entramos en esta función con un poco más de detalle.

La función de vigilancia en el hogar de *HogarSeguro* que se examina en el recuadro identifica las siguientes funciones (una lista abreviada) que realiza el actor identificado como **propietario de la casa**:

- Tener acceso a la cámara de vigilancia vía Internet.
- Seleccionar la cámara que desea ver.
- Solicitar vistas en miniatura de todas las cámaras.
- Desplegar vistas de la cámara en una ventana de una PC.
- Controlar la visión panorámica y de acercamiento en una cámara específica
- Registrar en forma selectiva la salida de cámara.
- Repetir la salida de cámara

Conforme se realizan las conversaciones posteriores con el interesado (que desempeña el papel de un propietario), el equipo de recopilación de requisitos desarrolla casos de uso para cada una de las funciones mencionadas. En general, los casos de uso se escriben primero en un estilo narrativo informal. Si se requiere mayor formalidad se rescribe el mismo caso de uso utilizando un formato estructurado similar al propuesto en el capítulo 7 y reproducido en esta sección como un recuadro.

Con fines ilustrativos, considérese la función "acceso a cámara de vigilancia-despliegue de vistas de cámara (ACV-DVC)". El interesado que desempeña el papel del **propietario** podría escribir el siguiente relato:

Caso de uso: Acceso a cámara de vigilancia-despliegue de vistas de cámara (ACV-DVC)

Actor: propietario

Si me encuentro en un lugar lejano puedo usar una PC con un software de navegación apropiado para entrar al sitio web de los productos *HogarSeguro*. Ingreso mi clave de usuario y dos niveles de contraseñas y, después de que estoy validado, tengo acceso a toda la funcionalidad de mi sistema *HogarSeguro* instalado. Para tener acceso a la vista de una cámara específica selecciono "vigilancia" de los botones desplegados para las funciones más importantes. Después escojo "seleccionar una cámara" y se despliega un plano de planta de la casa. Entonces selecciono la cámara en la que estoy interesado. En forma alterna, puedo ver simultáneamente una lista con vistas en miniatura de todas las cámaras al seleccionar "todas las cámaras" como mi opción de visualización. Una vez que he seleccionado una cámara, selecciono "vista" y una vista de un cuadro por segundo aparece en una ventana, a la cual identifica la cámara clave. Si quiero cambiar de cámara, elijo "seleccionar una cámara" y la ventana de visión original desaparece y se despliega de nuevo el plano de planta de la casa.

Una variación del caso de uso relatado presenta la interacción como una secuencia ordenada de las acciones del usuario. Cada acción se representa como un enunciado declarativo. Después de visitar la función ACV-DVC, se puede escribir:

Caso de uso: Acceso a cámara de vigilancia-despliegue de vistas de cámara (ACV-DVC)

Actor: propietario

1. El propietario entra en el sitio Web de *HogarSeguro*.
2. El propietario introduce su clave de usuario.
3. El propietario introduce dos contraseñas (cada una de al menos ocho caracteres)
4. El sistema despliega todos los botones de las funciones más importantes
5. El propietario selecciona "vigilancia" de los botones de funciones más importantes
6. El propietario elige "seleccionar una cámara".
7. El sistema despliega el plano de planta de la casa.
8. El propietario selecciona un icono de cámara del plano de planta
9. El propietario selecciona el botón "vista".
10. El sistema despliega una ventana de visión, identificado por la clave de la cámara
11. El sistema muestra salida de video dentro de la ventana de visión con una velocidad de un marco por segundo.

Es importante destacar que esta presentación secuencial no considera algunas interacciones alternativas (la narrativa tiene un flujo más libre y representa unas cuantas alternativas). Los casos de uso de este tipo se refieren algunas veces como *escenarios primarios* [SCH98].

> "Los casos de uso pueden usarse en muchos procesos [de software]. Nuestro favorito es un proceso que sea iterativo y conducido por el riesgo."
>
> **Geri Schneider y Jason Winters**

Por supuesto, para una comprensión completa de la función que se pretende describir es esencial una descripción de las interacciones alternativas. Por lo tanto, cada paso en el escenario primario se evalúa realizando las siguientes preguntas ISC:

¿Cómo se examinan cursos alternativos de acción mientras se desarrolla un caso de uso?

- ¿El acto puede ejecutar otra acción en este punto?
- ¿Es posible que el actor encuentre alguna condición de error en este punto? Si es así, ¿cuál podría ser?
- ¿Es posible que el actor encuentre algún otro comportamiento provocado por algún evento fuera de su control? Si es así, ¿cuál podría ser?

El resultado de las respuestas a estas preguntas es la creación de un conjunto de escenarios secundarios que son parte del caso de uso original, pero que representan comportamientos alternativos.

Por ejemplo, considérense los pasos 6 y 7 en el escenario primario presentado líneas atrás:

6. El propietario elige "seleccionar una cámara".
7. El sistema despliega el plano de planta de la casa

¿El actor puede ejecutar otra acción en este punto? La respuesta es: "sí". Con referencia al relato de flujo libre, el actor puede elegir ver las vistas en miniatura de todas las cámaras de manera simultánea. Por ende, un escenario secundario podría ser "Ver las vistas en miniatura de todas las cámaras"

¿Es posible que el actor encuentre alguna condición de error en este punto? Cuando un sistema basado en computadora está en funcionamiento puede ocurrir cualquier cantidad de condiciones de error. En este contexto se consideran sólo las condiciones de error que pueden ocurrir como resultado directo de las acciones descritas en los pasos 6 o 7. De nuevo, la respuesta a la pregunta es: "sí" Puede ser que nunca se haya configurado un plano de planta con iconos de las cámaras. Por lo tanto, al elegir "seleccionar una cámara" se produce una condición de error: "no existe un plano de planta configurado para esta casa".[12] Esta condición de error se convierte en un escenario secundario.

¿Es posible que el actor encuentre algún otro comportamiento en este punto? De nuevo, la respuesta a la pregunta es: "sí". Cuando se realizan los pasos 6 y 7 el sistema puede encontrar una condición de alarma Esto podría resultar en que el sistema desplegara una notificación especial de alarma (tipo, ubicación, acción del sistema) y le proporcione al actor una serie de opciones relacionadas con la naturaleza de la alarma. Como este escenario secundario puede ocurrir para casi todas las interacciones, no se convertirá en una parte del caso de uso para el **ACV-DVC**. En

12 En este caso, otro actor, el **administrador del sistema** tendría que configurar el plano de planta instalar e inicializar (es decir, asignar una ID a cada equipo) para todas las cámaras, así como realizar pruebas para estar seguro de que cada una de ellas es accesible por medio del sistema y a través del plano de planta.

vez de eso, se desarrollará un caso de uso por separado —"condición de alarma encontrada"—, el cual estará referenciado a otros casos de uso, según se requiera

En relación con las plantillas formales para los casos de uso que se muestran en el recuadro, los escenarios secundarios se representan como excepciones a la secuencia básica descrita respecto al **ACV-DVC**.

HOGARSEGURO

Plantilla de caso de uso para la vigilancia

Caso de uso: Acceso a la cámara de vigilancia-despliegue de vistas de cámara (ACV-DVC).

Propietario

en contexto: Ver la salida de las cámaras colocadas a lo largo de la casa desde cualquier ubicación remota a través de Internet.

previas: El sistema se debe configurar por completo; se deben obtener ID y contraseñas apropiadas para los usuarios.

El propietario decide echarle un vistazo a su casa mientras se encuentra fuera de ella

1. El propietario entra al sitio web de *Productos HogarSegura*.
2. El propietario introduce su ID de usuario.
3. El propietario introduce dos contraseñas (cada una de al menos ocho caracteres).
4. El sistema despliega todos los botones de las funciones más importantes.
5. El propietario selecciona "vigilancia" de los botones de las funciones más importantes.
6. El propietario selecciona "escoger una cámara".
   El sistema despliega el plano de la casa.
7. El propietario selecciona el icono de una cámara del plano de planta.
8. El propietario selecciona el botón "vista".
9. El sistema despliega una ventana de visualización que está identificada con la ID de la cámara.
11. El sistema despliega la salida de video dentro de la ventana de visualización a un cuadro por segundo.

Excepciones:

1. La ID o las contraseñas son incorrectas o no se reconocen, véase el caso de uso: "validar ID y contraseñas"
2. La función de vigilancia no está configurada para este sistema, así que el sistema despliega el mensaje de error apropiado; véase el caso de uso: "configurar la función de vigilancia".
3. El propietario selecciona "Ver vistas en miniatura para todas las cámaras"; véase el caso de uso: "Ver vistas en miniatura para todas las cámaras".
4. No está disponible un plano de planta o éste no se ha configurado, así que el sistema despliega el mensaje de error apropiado; véase el caso de uso "configurar plano de la casa".
5. Se encuentra una condición de alarma; véase el caso de uso: "condición de alarma encontrada".

Prioridad: Prioridad moderada, que se implementará después de las funciones básicas

Disponible en: El tercer incremento

Frecuencia de uso: Poco frecuente

Canal hacia el actor: A través de un browser basado en PC y conexión de Internet al sitio web de *HogarSeguro*.

Actores secundarios: Administrador del sistema, cámaras.

Canales hacia los actores secundarios:

1. Administrador del sistema: sistema basado en PC
2. Cámaras: conectividad inalámbrica

Aspectos pendientes.

1. ¿Cuál es el mecanismo que protege contra el uso no autorizado de esta capacidad por parte de los empleados de la compañía?
2. ¿La seguridad es suficiente? El ingreso no autorizado en esta característica representaría una invasión importante de la privacidad.
3. ¿La respuesta del sistema vía Internet será aceptable dado el ancho de banda requerido para las vistas de cámara?
4. ¿Se desarrollará una capacidad para proporcionar video a una tasa mayor de cuadros por segundo cuando estén disponibles conexiones con mayor ancho de banda?

Referencia Web

¿Cuándo se han terminado de escribir los casos de uso? Para una exposición valiosa de este tópico, véase **cetips.org/use-cases-done.htmlcetips.org/use-cases-done.html**.

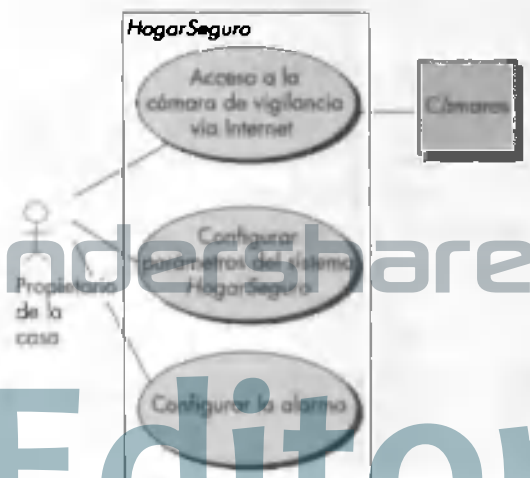
En muchos casos no es necesario crear una representación gráfica de un escenario de uso. Sin embargo, la representación diagramática puede facilitar la comprensión, en particular cuando el escenario es complejo. Como se mencionó en el capítulo 7, el UML proporciona una capacidad para hacer diagramas de casos de uso preliminar para el producto *HogarSeguro*. Cada caso de uso se representa mediante un óvalo. En esta sección sólo se ha examinado en detalle el caso de uso para el **ACV-DVC.**

8.5.2 Desarrollo de un diagrama de actividad

El diagrama de actividad en UML (que se trató en forma breve en los capítulos 6 y 7) complementa el caso de uso al proporcionar una representación gráfica del flujo de interacción dentro de un escenario específico. De manera similar al diagrama de flujo, un diagrama de actividad utiliza rectángulos redondeados para indicar una función específica del sistema, flechas para representar el flujo a través del sistema, rombos de decisión para mostrar una ramificación por decisión (cada flecha que sale del rombo se etiqueta), y líneas horizontales sólidas para indicar que ocurren actividades paralelas

FIGURA 8.6

Diagrama preliminar de caso de uso para el sistema *HogarSeguro*.



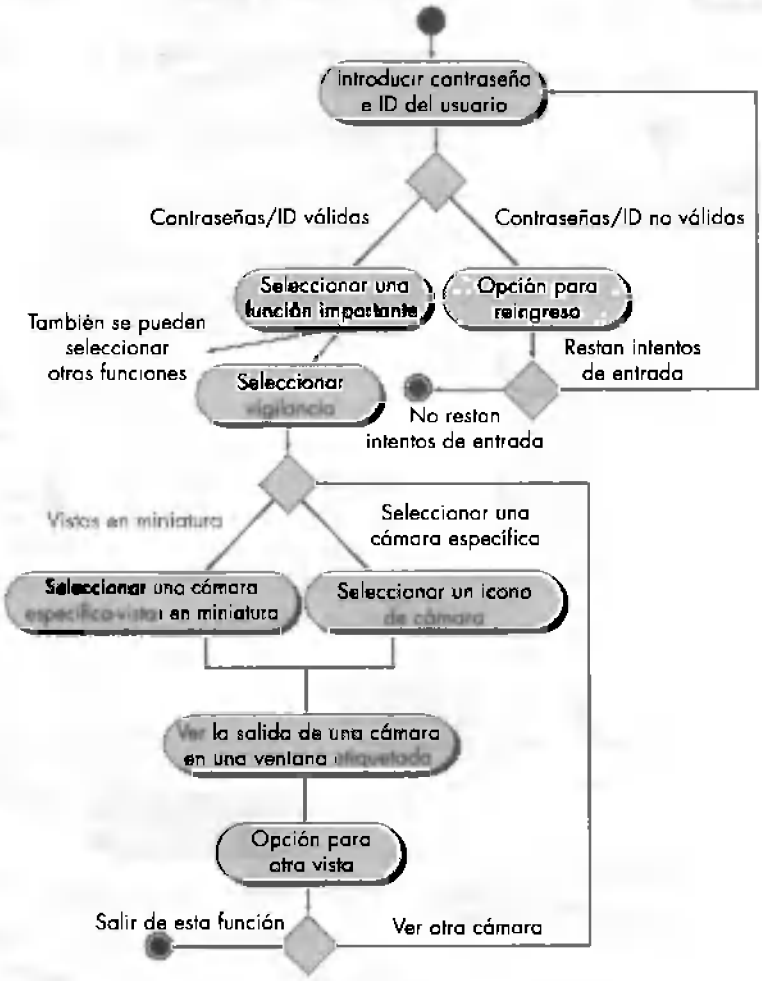


FIGURA 8.7 Diagrama de actividad para la función de acceso a la cámara de vigilancia-despliegue en vistas de cámara.

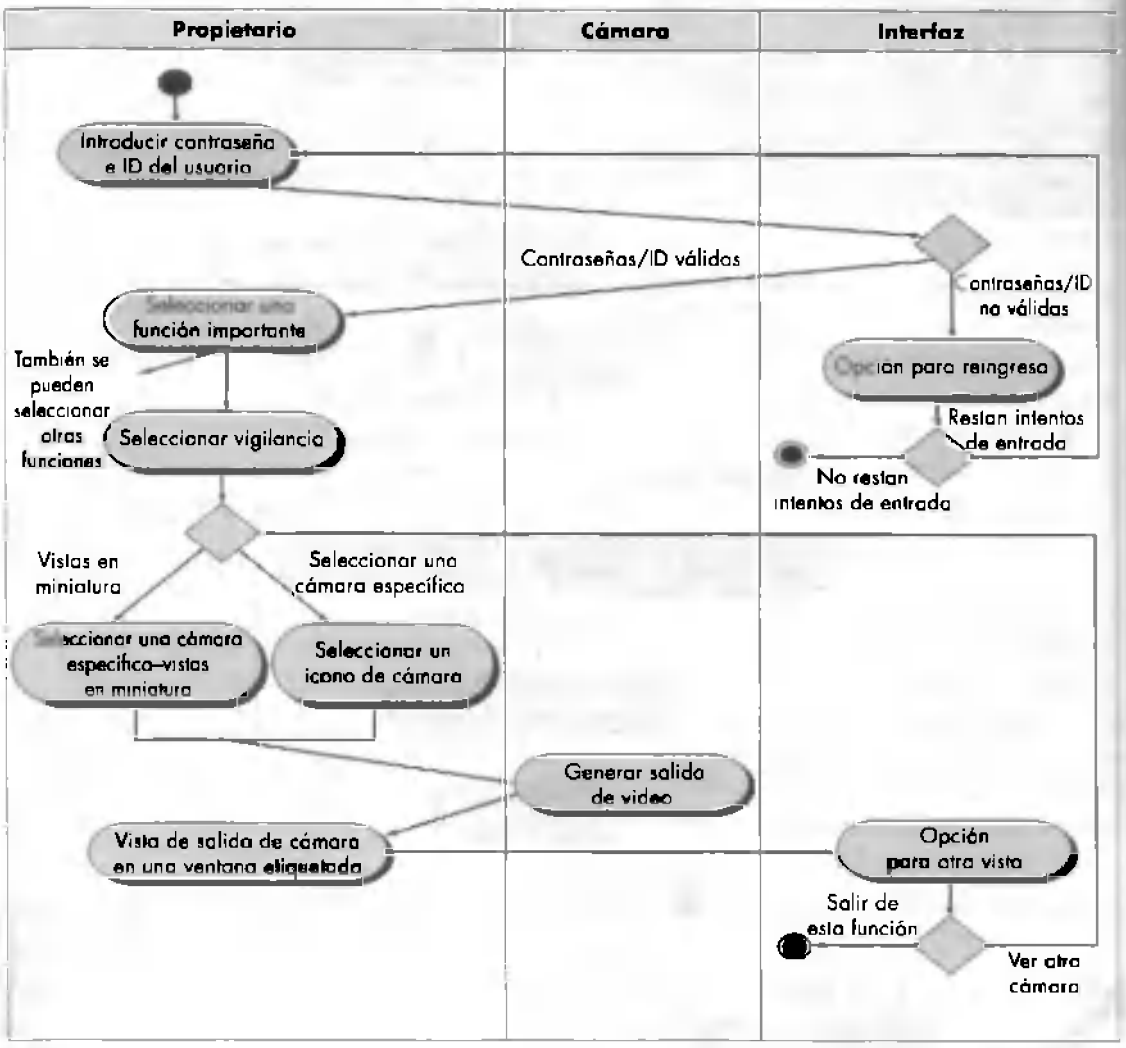
En la figura 8.7 se muestra un diagrama de actividad para la función de **ACV-DVC**. Se debe resaltar que el diagrama de actividad agrega detalles adicionales que no se mencionan de manera directa (pero sí implícita) en el caso de uso. Por ejemplo, un usuario puede intentar ingresar la **IDusuario** y la **contraseña** sólo un número limitado de veces. Esto se representa mediante un rombo de decisión debajo de *opción para reingreso*.

PUNTO CLAVE Los diagramas de actividad de UML representan las acciones y funciones que ocurren dentro de una función.

8.5.3 Diagramas de carril

El *diagrama de carril* de UML es una variación útil del diagrama de actividad, ya que permite al modelador la representación del flujo de actividades descritas por el caso de uso y, al mismo tiempo, indicar qué actor (si hay múltiples actores involucrados en una función específica) o clase de análisis tiene la responsabilidad de la acción descrita mediante un rectángulo de actividad. Las responsabilidades se representan

FIGURA 8.8 Diagrama de carril para la función de Acceso a la cámara de vigilancia-despliegue de vistas de cámara.



PUNTO CLAVE

Un diagrama de carriles en UML representa el flujo de las acciones y los documentos e indica cuáles actores realizan cada una de ellas.

como segmentos paralelos que dividen el diagrama en forma vertical, como los carriles de una alberca.

Existen tres clases de análisis —**Propietario**, **Interfaz** y **Cámara**— con responsabilidades directas o indirectas en el contexto del diagrama de actividad representado en la figura 8.7. Respecto de la figura 8.8, el diagrama de actividad se reorganiza de forma que las actividades asociadas con una clase de análisis particular pertenezcan al carril correspondiente a dicha clase. Por ejemplo, la clase **Interfaz** representa la interfaz con el usuario de acuerdo con la visión del propietario. El diagrama de actividad considera dos opciones que son responsabilidad de la interfaz: *opción para el rein*

greso y *opción para otra vista* Estas opciones y las decisiones asociadas con ellas pertenecen al carril de **Interfaz**. Sin embargo, las flechas conducen desde ese carril de regreso al carril de **propietario**, donde ocurren las acciones del propietario.

8.6 MODELADO ORIENTADO AL FLUJO

El modelado de datos orientado al flujo es una de las notaciones de análisis utilizadas con mayor amplitud en la actualidad [13] Aunque el *diagrama de flujo de datos* (DFD) y los diagramas y la información relacionados no son una parte formal de UML, pueden utilizarse para complementar los diagramas en UML y proporcionar un conocimiento adicional de los requisitos y el flujo del sistema.

El DFD tiene una visión del sistema del tipo entrada-proceso-salida. Esto es, los objetos de datos fluyen hacia el interior del software, se transforman mediante elementos de procesamiento, y los objetos de datos resultantes fluyen al exterior del software. Los objetos de datos se representan mediante flechas rotuladas y las transformaciones se representan por medio de círculos (también llamadas *burbujas*) El DFD se presenta en una forma jerárquica. Esto es, el primer modelo de flujo de datos (algunas veces llamado un *DFD de nivel 0* o *diagrama de contexto*) representa el sistema como un todo. Los diagramas de flujo de datos subsecuentes refinan el diagrama de contexto, ya que proporcionan una cantidad creciente de detalles con cada nivel subsiguiente.

> "El propósito de los diagramas de flujo de datos es proporcionar un puente semántico entre los usuarios y los desarrolladores de sistemas."
>
> **Kenneth Kozar**

8.6.1 Creación de un modelo de flujo de datos

CONSEJO

...asonos que el DFD ...a mejo... y que no ...en lo moderno ...visión que ...a forma de ...l potencial al nivel Si es de ...recomienda el DFD

El diagrama de flujo de datos permite que el ingeniero de software desarrolle modelos del dominio de información y del dominio funcional al mismo tiempo. A medida que el DFD se refina hacia grados mayores de detalle, el analista desempeña una descomposición funcional implícita del sistema. Al mismo tiempo, el refinamiento del DFD resulta en un correspondiente refinamiento de datos mientras se mueve hacia los procesos que incorporan la aplicación

Unas pocas directrices simples permiten obtener una ayuda invaluable durante la creación de un diagrama de flujo de datos: 1) el nivel 0 del diagrama de flujo de datos debe representar al software/sistema como una sola burbuja; 2) la entrada y la salida primaria deben establecerse con cuidado; 3) la refinación debe comenzar por el aislamiento de procesos, objetos de datos y almacenamientos de datos candidatos a ser representados en el siguiente nivel; 4) todas las flechas y burbujas se deben rotular con nombres significativos; 5) se debe mantener la continuidad del flujo de infor-

13 El modelo de flujo de datos es una actividad de modelado esencial en el *análisis estructurado*

FIGURA 8.9

DFD al nivel de contexto para la función de seguridad de *HogarSeguro*.

mación al cambiar de nivel a nivel;[14] y 6) la refinación de las burbujas debe hacerse una por una. Existe una tendencia natural a complicar de más el diagrama de flujo de datos. Esto ocurre cuando el analista intenta mostrar muchos detalles demasiado pronto o representar aspectos de procedimiento del software en lugar de elementos del flujo de información.

El uso del DFD y de la notación relacionada se ilustra de nuevo considerando la función de seguridad en el hogar de *HogarSeguro*. En la figura 8.9 se muestra un DFD al nivel de contexto para la función de seguridad. Las entidades externas primarias (cajas) producen información para el uso del sistema y consumen información que éste genera. Las flechas rotuladas representan objetos de datos o jerarquías de objetos de datos. Por ejemplo, **comandos y datos del usuario** abarcan todos los comandos de configuración, todos los comandos de activación/desactivación, todas las interacciones y todos los datos que se ingresan para calificar o expandir un comando.

¡PUNTO CLAVE

La continuidad del flujo de información debe mantenerse conforme se refina cada nivel del DFD. Esto significa que la entrada y salida en un nivel deben ser las mismas que la entrada y salida en un nivel refinado

El DFD de nivel 0 ahora se expande a un modelo de flujo de datos de nivel 1. ¿Pero cómo se logra esto? Un enfoque simple, pero efectivo, es realizar un "análisis gramatical" [ABB83] sobre la narrativa que describe la burbuja al nivel de contexto. Esto es, se aíslan todos los sustantivos y verbos en la *narrativa de procesamiento de HogarSeguro*[15] obtenida durante la primera reunión para la recopilación de requisitos. Con propósitos ilustrativos, considérese el siguiente texto narrativo de procesamiento con la primera aparición de todos los sustantivos subrayados y la primera aparición de todos los verbos en itálicas.[16]

14 Esto es, los objetos de datos que fluyen hacia el sistema o cualquier transformación en algún nivel, deben ser los mismos objetos de datos (o sus partes constituyentes) que fluyen hacia la transformación en un nivel más refinado.

15 Una narrativa del procesamiento es similar en estilo al caso de uso, pero algo diferente en propósito. La narrativa del procesamiento proporciona una descripción general de la función que será desarrollada. No es un escenario escrito desde el punto de vista de alguno de los actores.

16 Debe notarse que se omiten los sustantivos o verbos que son sinónimos o que no tienen una ingerencia directa en el proceso de modelación. También se debe notar que, cuando se considere el modelado basado en clases de la sección 8.7, se usará un análisis gramatical similar.

...s gramatical
...s de
...nero puede
...r un
...paso inicial
...este alguna
...para definir
...s de datos y las
...s que
...ellos

La función de seguridad de HogarSeguro *permite* al propietario *configurar* el sistema de seguridad cuando éste se *instala*, *monitorear* todos los sensores que se *conectan* al sistema de seguridad y que *interactúan* con el propietario a través de Internet, una PC o un panel de control.

Durante la instalación, la PC de *HogarSeguro* se utiliza para *programar* y *configurar* el sistema. A cada sensor se le asigna un número y tipo, se programa una contraseña maestra para *habilitar* o *deshabilitar* el sistema, y algunos números telefónicos *ingresan* para *marcarse* cuando se presenta un evento en los sensores.

Cuando se *reconoce* un evento en los sensores, el software *solicita* una alarma audible que el propietario especifica durante las actividades de configuración del sistema, el software marca el número telefónico de un servicio de monitoreo, *proporciona* información acerca de la ubicación, *reporta* la naturaleza del evento que se ha detectado. El número telefónico se remarca cada 20 segundos hasta que se *obtiene* una conexión telefónica.

El propietario *recibe* información de seguridad a través de un panel de control, la PC o un buscador que en forma colectiva se denomina una interfaz. La interfaz *despliega* mensajes de opción e información del estatus del sistema en el panel de control, la PC o la ventana del buscador. La interacción del propietario asume la siguiente forma..

En relación con el análisis gramatical comienza a surgir un patrón. Los verbos son los procesos de *HogarSeguro*; esto es, al final pueden representarse como burbujas en un DFD subsecuente. Los sustantivos son entidades externas (cajas), objetos de datos o de control (flechas), o almacenamientos de datos (líneas dobles). Después debe observarse que los sustantivos y verbos se puedan asociar de distintas formas

...de nivel 1
...la función
...seguridad de
...eguro.

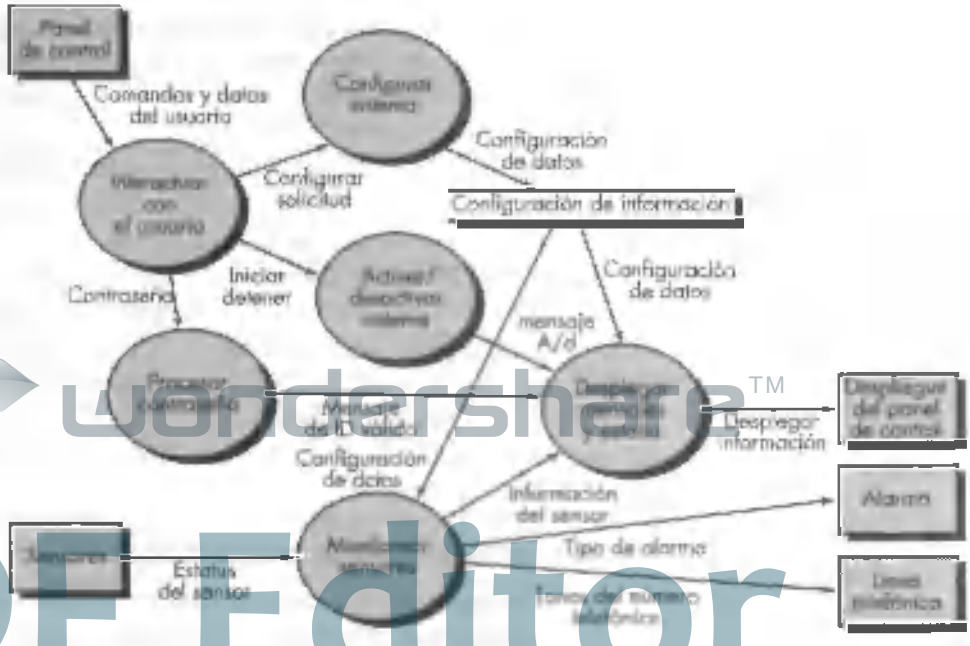


FIGURA 8.11

DFD de nivel 2 que refina el proceso de monitoreo de sensores.

CONSEJO

Se debe tener la seguridad de que toda la narrativa de procesamiento que se intenta analizar está escrita con el mismo grado de abstracción.

entre sí. Por ejemplo, a cada sensor se le asigna un número y un tipo; por lo tanto el número y el tipo son atributos del objeto de datos **sensor**. Entonces, al realizar un análisis gramatical sobre la narrativa de procesamiento para una burbuja en cualquier nivel del DFD, se puede generar mucha información útil acerca de cómo proceder con la refinación para el siguiente nivel. En la figura 8.10 se presenta un DFD de nivel 1 para el cual se ha utilizado esta información. El proceso al nivel de contexto mostrado en la figura 8.9 se ha expandido a seis procesos obtenidos de un examen del análisis gramatical. De manera similar, el flujo de información entre los procesos en el nivel 1 ha sido obtenido del análisis. Además, se mantiene la continuidad del flujo de información entre los niveles 0 y 1.

Los procesos representados en el DFD de nivel 1 se refinan después hacia niveles más bajos. Por ejemplo, es posible refinar el proceso *monitorear sensores* hacia un DFD de nivel 2 como se muestra en la figura 8.11. De nuevo, debe señalarse que se mantiene la continuidad del flujo de información entre los niveles.

La refinación de los DFD continúa hasta que cada burbuja realiza una sola función; es decir, hasta que el proceso que representa la burbuja desempeñe una función que podría implementarse con facilidad como un componente de programa. En el capítulo 9 se examina un concepto, llamado *cohesión*, con el cual se evalúa la simplicidad de una función dada. Por ahora, se busca refinar los DFD hasta que cada burbuja tenga "un solo significado".

8.6.2 Creación de un modelo de control del flujo

En muchos tipos de aplicaciones el modelo de datos y el diagrama de flujo de datos son todo lo que se necesita para obtener una visión significativa de los requisitos del

software. Sin embargo, como ya se ha mencionado, existe una clase muy grande de aplicaciones que están guiadas por eventos en lugar de datos, que producen información de control en lugar de reportes o despliegues, y que procesan información con un especial interés por el tiempo y el rendimiento. Dichas aplicaciones requieren aplicar el *modelado de control del flujo*, además del modelado del flujo de datos.

Ya se ha mencionado que un evento o elemento de control se implementa como un valor booleano (por ejemplo, verdadero o falso, encendido o apagado, 1 o 0) o una lista discreta de condiciones (vacío, saturado, lleno). En la selección de los eventos que son candidatos potenciales se sugieren las siguientes directrices:

¿Cómo se seleccionan los eventos potenciales para un diagrama de control del flujo, un diagrama de estado o una especificación de control?

- Hacer una lista de todos los sensores que el software "lee".
- Listar todas las condiciones de interrupción
- Listar todos los "interruptores" que maneja un operador
- Listar todas las condiciones de datos.
- De acuerdo con el análisis de sustantivos y verbos aplicado a la narrativa de procesamiento, revisar todos los "elementos de control" como posibilidades de entradas y salidas del control de flujo.
- Describir el comportamiento de un sistema mediante la identificación de sus estados; determinar el grado en el que se alcanza cada estado, y definir las transiciones entre los estados
- Enfocarse en posibles omisiones —un error muy común al especificar el control—; por ejemplo, se puede preguntar: "¿existe alguna otra manera de alcanzar este estado o salir de él?".

8.6.3 Especificación de control

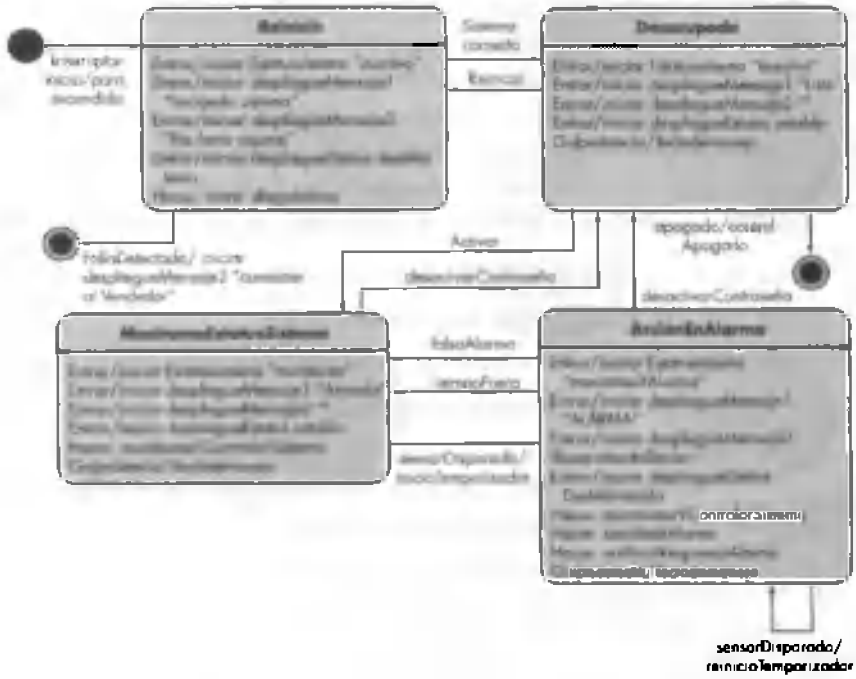
La *especificación de control* (EC) representa el comportamiento del sistema (en el nivel desde el cual está referido) de dos maneras diferentes.[17] La EC contiene un diagrama de estado que es una especificación secuencial del comportamiento. También puede contener una tabla de activación del programa· una especificación combinatoria del comportamiento.

En la figura 8.12 se muestra un diagrama de estado preliminar[18] para el modelo de control del flujo en el nivel 1 para *HogarSeguro* El diagrama indica cómo responde el sistema a diferentes eventos conforme éste atraviesa los cuatro estados definidos en este nivel. Al revisar el diagrama de estado, un ingeniero de software puede determinar el comportamiento del sistema y, aún más importante, puede encontrar si existen "hoyos" en el comportamiento especificado

17 Después, en este mismo capítulo, se presenta notación adicional para el modelado del comportamiento.

18 La notación para el diagrama de estado se muestra aquí de conformidad con la notación del UML. En el análisis estructurado se cuenta con un "diagrama de transición de estado", pero el formato del UML es superior en contenido y representación de información

FIGURA 8.12 Diagrama de estado para la función de seguridad de *HogarSeguro*.



Por ejemplo, el diagrama de estado (figura 8.12) indica que las transiciones desde el estado *desocupado* pueden ocurrir si el sistema se reinicia, activa o apaga. Si el sistema se activa (es decir, se enciende el sistema de alarma) ocurre una transición hacia el estado *MonitoreoEstatusSistema*, los mensajes que se despliegan cambian como se muestra, y se solicita el proceso *SistemaControlyMonitoreo*. Con el estado *Monitoreo Stratus Sistema* ocurren dos transiciones: 1) al desactivar el sistema se presenta una transición de regreso al estado *desocupado*; 2) cuando se dispara un sensor ocurre una transición hacia el estado *AcciónEnAlarma*. Durante la revisión se consideran todas las transiciones y el contenido de todos los estados

HOGARSEGURO

Modelado del flujo de datos

El escenario: Cubículo de Jamie, después que ha concluido la última reunión para la recopilación de requisitos.

Los actores: Jamie, Vinod y Ed, miembros del equipo de ingeniería del software de *HogarSeguro*.

La conversación:

(Jamie ha bosquejado los modelos que se muestran en las figuras 8.9, 8.10, 8.11 y 8.12 y se los muestra a Ed y Vinod).

Jamie: Tomé un curso de ingeniería del software en la universidad, y ahí aprendí estas cosas. El profesor dijo

está un poco pasada de moda, pero ¿saben
Me ayuda a clarificar las cosas.

Es genial. Pero aquí no veo ninguna clase ni ningún

esta es sólo un modelo de flujo con algunos
de comportamiento incluidas.

¿Entonces, estos DFD representan una visión E-P-
software, no?

¿E-P-S?

Entrada-proceso-salida. En realidad los DFD son
... Si los observas por un momento,
la forma en que los objetos fluyen a través del
y cómo éstos se transforman.

como si pudiéramos convertir cada burbuja
componente ejecutable al menos en el nivel más
del DFD.

Eso es lo mejor parte, si se puede De hecho,
forma de traducir los DFD a una arquitectura

Ed: ¿De verdad?

Jamie: Sí, pero primero debemos desarrollar un modelo de análisis completo, y éste no lo es.

Vinod: Bueno, es un primer paso Pero vamos a tener que abordar elementos basados en clases y también aspectos del comportamiento, aunque este diagrama de estado hace algo de eso

Ed: Tenemos mucho trabajo por hacer y no mucho tiempo para hacerlo.

(Doug –el gerente de ingeniería del software– entra en el cubículo.)

Doug: Entonces, ¿los primeros días estarán dedicados al desarrollo del modelo de análisis, eh?

Jamie (con orgullo): Ya comenzamos.

Doug: Bien, tenemos mucho trabajo por hacer y no mucho tiempo para hacerlo.

(Los tres ingenieros de software se miran entre sí y sonríen.)

La EC describe el comportamiento del sistema, pero no brinda información acerca del trabajo interior de los procesos que activa. La notación de modelado que proporciona esta información se estudia en la sección 8.6.4.

8.6.4 Especificación de proceso

La *especificación de proceso* (EP) se utiliza para describir todos los procesos del modelo de flujo que aparecen en el nivel final de refinación. El contenido de la especificación de proceso puede incluir texto narrativo, una descripción en lenguaje de diseño de programas (LDP)[19] del algoritmo del proceso, ecuaciones matemáticas, tablas, diagramas o gráficas Al proporcionar una EP para acompañar cada burbuja en el modelo de flujo, el ingeniero de software crea una "miniespecificación" que puede servir como guía para el diseño del componente del software que implementará el proceso.

PUNTO CLAVE

Para ilustrar el uso de la EP, considérese la transformación *procesamiento de contraseña* representada en el modelo de flujo para *HogarSeguro* (figura 8.10). La EP para esta función podría tomar la forma:

EP: procesamiento de contraseña (en el panel de control). La transformación *procesamiento de contraseña* valida la contraseña en el panel de control para la función de se-

19 El lenguaje de diseño de programas (LDP) mezcla la sintaxis del lenguaje de programación con la narrativa en texto para proporcionar detalles del diseño del procedimiento El LDP se estudia en el capítulo 11

guridad de *HogarSeguro*. El *procesamiento de contraseña* recibe una contraseña de cuatro dígitos a partir de la función de *interacción con el usuario*. La contraseña se compara primero con la contraseña maestra almacenada en el sistema. Si la contraseña maestra coincide [Mensaje de ID válida = verdadero] se pasa a la función de *mensaje y despliegue del estatus*. Si la contraseña maestra no coincide, se comparan los cuatro dígitos con una tabla de contraseñas secundarias (éstas se pueden asignar a invitados o trabajadores que necesitan entrar en la casa cuando el propietario no esté). Si la contraseña coincide con alguna entrada dentro de la tabla [mensaje de Id válida = verdadero], se pasa a la función de *mensaje y despliegue del estatus*. Si no existe alguna coincidencia [mensaje de Id válida = falso], se pasa a la función de mensaje y despliegue del estatus.

Si en esta etapa se desean tener detalles algorítmicos adicionales, se podría incluir también una representación en lenguaje de diseño de programas como parte de la EP. Sin embargo, muchos profesionales del software creen que la versión en LDP se puede posponer hasta que comience el diseño de componentes

HERRAMIENTAS DE SOFTWARE

Análisis estructurado

Objetivo: Las herramientas del análisis estructurado ayudan a un ingeniero de software a crear modelos de datos, modelos de flujo y modelos de comportamiento de una manera que permita la verificación de la continuidad y la consistencia, así como su fácil edición y extensión. Los modelos creados utilizando estas herramientas brindan al ingeniero de software una visión de la representación del análisis y ayudan a eliminar errores antes de que éstos se propaguen en el diseño o, aun peor, en la misma implementación.

Mecánica: Las herramientas de esta categoría utilizan un "diccionario de datos" como la base de datos central para la descripción de todos los objetos de datos. Una vez que las entradas del diccionario están definidas, pueden crearse diagramas de entidad-relación y se pueden desarrollar las jerarquías de objetos. Las características de diagramación del flujo de datos permiten crear fácilmente este modelo gráfico y también proporciona características para la creación de especificaciones de control (EC) y especificaciones de proceso (EP). Las herramientas de análisis también ayudan al ingeniero de software en la creación de modelos de comportamiento que usan los diagramas de estado como notación operativa.

Herramientas representativas[20]

AxiomSys, desarrollado por STG, Inc. (www.stgcase.com), proporciona un paquete completo de herramientas para el análisis de la estructura que incluye extensiones de Hartley-Pirbhai para el modelado de sistemas en tiempo real.

MacA&D, WinA&D, desarrollados por Excel Software (www.excelsoftware.com), proporcionan un conjunto de herramientas simples y baratas para el análisis y el diseño en máquinas Mac y Windows.

MetaCASE Workbench, desarrollado por Metacase Consulting (www.metacase.com) es una metaherramienta utilizada para definir un método de análisis o diseño (incluido en análisis estructurado): sus conceptos, reglas, notaciones y generadores.

System Architect, desarrollado por Popkin Software (www.popkin.com), proporciona un amplio rango de herramientas de análisis y diseño, incluye herramientas para el modelado de datos y el análisis estructurado.

20 Las herramientas mencionadas aquí representan una muestra de esta categoría. En la mayoría de los casos los nombres están registrados por sus respectivos desarrolladores.

8.7 MODELADO BASADO EN CLASES

¿Qué se debe hacer para desarrollar los elementos basados en clases de un modelo de análisis: clases y objetos, atributos, operaciones, paquetes, modelos CRC y diagramas de colaboración? Las secciones siguientes presentan una serie de directrices informales que ayudarán a identificarlos y representarlos.

8.7.1 Identificación de clases de análisis

Al observar el interior de una habitación se verá que existe un conjunto de objetos físicos que pueden identificarse, clasificarse y definirse con facilidad (en términos de atributos y operaciones). Pero cuando se "observa" el espacio del problema de una aplicación de software, quizá las clases (y los objetos) sean más difíciles de comprender.

> "El problema realmente difícil es descubrir cuáles son los objetos correctos [clases] en primer lugar."
>
> **Carl Argila**

Se puede iniciar la identificación de clases al examinar el enunciado del problema o (de acuerdo con la terminología aplicada previamente en este capítulo) al realizar un "análisis gramatical" sobre las narrativas desarrolladas para el sistema que se va a construir o de los casos de uso. Las clases se determinan al subrayar cada sustantivo e introduciéndolas en una simple tabla. Los sinónimos deben anotarse. Si la clase se requiere para implementar una solución, entonces es parte del espacio de solución; por otro lado, si una clase sólo se necesita para describir una solución es parte del espacio del problema. ¿Qué se debe buscar después de que todos los sustantivos han sido aislados? Las *clases* se manifiestan en una de las siguientes formas:

¿De qué forma se manifiestan a sí mismas las clases [de análisis como elementos del espacio de solución?

- *Entidades externas* (por ejemplo, otros sistemas, dispositivos, personas) que producen o consumen información que usará un sistema basado en computadora.
- *Cosas* (por ejemplo, reportes, despliegues, letras, señales) que son parte del dominio de la información para el problema.
- *Sucesos o eventos* (por ejemplo, una transferencia de propiedad o la consecución de una serie de movimientos de robot) que ocurren dentro del contexto de la operación del sistema.
- *Papeles* (por ejemplo, gerente, ingeniero, personal de ventas) que desempeñan personas que interactúan con el sistema.
- *Unidades organizacionales* (por ejemplo, división, grupo, equipo) relevantes para alguna aplicación.
- *Sitios* (por ejemplo, el piso de manufactura o el puerto de carga) que establecen el contexto del problema y la función global del sistema.

- *Estructuras* (por ejemplo, sensores, vehículos de cuatro ruedas o computadoras que definen una clase de objetos o clases de objetos relacionadas.

Esta categorización es una de las muchas que se han propuesto en la bibliografía. Por ejemplo, si los desarrolladores del software para un sistema de observación médica definen un objeto con el nombre de **ImagenInvertida** o **InversiónDeImagen**, podrían estar cometiendo un error sutil. La **Imagen** obtenida del software podría, por supuesto, ser una clase (es una cosa integrante del dominio de la información). La inversión de la imagen es una operación que se aplica a la clase. Probablemente la *inversión( )* se definiría como una operación para la clase **Imagen**, pero no se establecería como una clase diferente para connotar "inversión de imagen". Como establece Cashman [CAS89]: "El objetivo de la orientación hacia los objetos es encapsular, pero aun así mantener separados, los datos y las operaciones sobre los datos".

Para ilustrar la forma en que las clases de análisis pueden definirse durante las primeras etapas del modelado, se utiliza de nuevo la función de seguridad de *HogarSeguro*. En la sección 8.6.1 se realizó un "análisis gramatical" sobre la narrativa del procesamiento[22] para la función de seguridad. Al extraer los sustantivos se puede proponer una serie de clases potenciales:

Clase potencial	Clasificación general
propietario	papel o entidad externa
sensor	entidad externa
panel de control	entidad externa
instalación	ocurrencia
sistema (alias sistema de seguridad)	cosa
número, tipo	no objetos, atributos del sensor
contraseña maestra	cosa
número telefónico	cosa
evento del sensor	ocurrencia
alarma audible	entidad externa
servicio de monitoreo	unidad organizacional o entidad externa

La lista podría extenderse hasta que todos los sustantivos en la narrativa del procesamiento hayan sido considerados. Obsérvese que cada entrada en la lista ha sido

21 Otra categorización importante —la cual define entidad, frontera y clases de controlador— se examina en la sección 8.7.4.

22 Es importante notar que esta técnica debe usarse también para todos los casos de uso desarrollados como parte de la actividad para la recopilación de requisitos (obtención). Esto es, los casos de uso pueden analizarse gramaticalmente para extraer clases de análisis potenciales.

llamada como un objeto potencial. Cada uno de ellos debe considerarse a fondo antes de tomar una decisión final.

> "Las clases luchan, algunas clases triunfan, otras son eliminadas."
>
> **Mao Zedong**

Coad y Yourdon [COA91] sugieren seis características de selección que se deben usar cuando un analista considera cada clase potencial para incluirlas en el modelo de análisis.

¿Cómo se determina si una clase potencial debe, de hecho, convertirse en una clase de análisis?

1. *Información referida* La clase potencial será útil durante el análisis sólo si la información acerca de ella debe recordarse para que el sistema pueda funcionar.
2. *Servicios requeridos*. La clase potencial debe tener un conjunto de operaciones identificables que puedan cambiar el valor de sus atributos de alguna manera
3. *Atributos múltiples*. Durante el análisis de requisitos el enfoque debe estar en la información "importante"; una clase con un solo atributo puede, de hecho, ser útil durante el diseño, pero probablemente es mejor representarla como un atributo de otra clase durante la actividad de análisis.
4. *Atributos comunes*. Es posible definir un conjunto de atributos para la clase potencial, y estos atributos pueden aplicarse en todas las instancias de la clase.
5. *Operaciones comunes*. Es posible definir un conjunto de operaciones para la clase potencial, y estas operaciones pueden aplicarse en todas las instancias de la clase.
6. *Requisitos esenciales*. Las entidades externas que aparecen en el espacio del problema, y que producen o consumen información esencial para la operación de cualquier solución para el sistema, se definirán casi siempre como clases en el modelo de requisitos.

Considerarla una clase legítima para incluirla en el modelo de requisitos requiere que una clase potencial satisfaga todas (o casi todas) estas características. La decisión de incluir clases potenciales en el modelo de análisis es algo subjetiva, y una evaluación posterior puede ocasionar que se descarte o reinstale una clase. Sin embargo, el primer paso del modelado basado en clases es la definición de clases, y se deben tomar decisiones (incluso subjetivas). Con esto en mente, se aplican las características de selección a la lista de clases potenciales de *HogarSeguro*:

Clase potencial	Número de característica que aplica
propietario	rechazada, 1 y 2 fallan aunque aplica 6
sensor	aceptada, todas aplican
panel de control	aceptada, todas aplican

instalacion	rechazada
sistema (alias función de seguridad)	aceptada: todas aplican
número, tipo	rechazada. falla 3, atributos del sensor
contraseña maestra	rechazada. falla 3
número telefónico	rechazada: falla 3
evento del sensor	aceptada: todas aplican
alarma audible	aceptada: aplican 2, 3, 4, 5, 6
servicio de monitoreo	rechazada: 1 y 2 fallan aunque aplica 6

Se debe señalar que 1) la lista anterior no está completa (se tendrían que agregar clases adicionales para terminar el modelo); 2) algunas de las clases potenciales rechazadas se convertirán en atributos para las clases aceptadas (por ejemplo, **número**, **tipo** son atributos de **sensor**, y **contraseña maestra** y **número telefónico** se pueden convertir en atributos de **sistema**); 3) la existencia de enunciados diferentes del problema podría ocasionar decisiones distintas de "aceptación o rechazo" (por ejemplo, si cada propietario tuviera una contraseña diferente o si pudiera identificarse por su voz, la clase **Propietario** satisfaría las características 1 y 2 y ésta habría sido aceptada).

8.7.2 Especificación de atributos

Los atributos describen una clase que ha sido seleccionada para incluirla en el modelo de análisis. En esencia, los atributos son los que definen la clase, lo cual clarifica qué significa la clase en el contexto del espacio del problema.

PUNTO CLAVE

Los atributos son el conjunto de objetos de datos que definen por completo la clase dentro del contexto del problema

En el desarrollo de un conjunto de atributos significativo para una clase de análisis un ingeniero de software puede estudiar de nuevo un caso de uso y seleccionar aquellas "cosas" que "pertenecen" de manera razonable a la clase. Además, se debe contestar la siguiente pregunta para cada clase: ¿Cuáles elementos de datos (compuestos o elementales) definen esta clase en el contexto del problema?

Esto se ilustra considerando la clase **sistema** definida para *HogarSeguro*. Se ha mencionado que el propietario puede configurar la función de seguridad para reflejar la información del sensor, información de la respuesta de alarma, información de la activación/desactivación, información de la identificación, y así sucesivamente. Estos elementos de datos compuestos se pueden representar de la siguiente manera:

información de identificación = ID del sistema + verificación del número telefónico + estatus del sistema
información de la respuesta de alarma = tiempo de retraso + número telefónico
información de la activación/desactivación = contraseña maestra + número de intentos permisibles + contraseña temporal

Algunos de los datos a la derecha del signo de igual podrían refinarse hasta un nivel elemental, pero para los propósitos de este capítulo constituyen una lista razonable de atributos para la clase **sistema** (parte sombreada de la figura 8.13).

Los sensores son parte del sistema global de *HogarSeguro*, y aun así no están enlistados como elementos de datos o como atributos en la figura 8.13. Ya se ha definido **sensor** como una clase, y varios objetos de **sensor** se asociarán con la clase **sistema**. En general, se evita la definición de un elemento como un atributo si más de uno de los elementos se asociará con la clase

8.7.3 Definición de operaciones

Las operaciones definen el comportamiento de un objeto. Aunque existen muchos tipos distintos de operaciones, éstas se pueden dividir, por lo general, en tres grandes categorías: 1) operaciones que manipulan los datos de alguna manera (por ejemplo, al agregar, borrar, reformatear, seleccionar), 2) operaciones que realizan un cómputo, 3) operaciones que preguntan acerca del estado de un objeto, y 4) operaciones que monitorean un objeto para la ocurrencia de un evento de control. Estas funciones se ejecutan al operar sobre atributos o asociaciones (sección 8.7.5). Por lo tanto, una operación debe tener "conocimiento" de la naturaleza de los atributos y asociaciones de la clase.

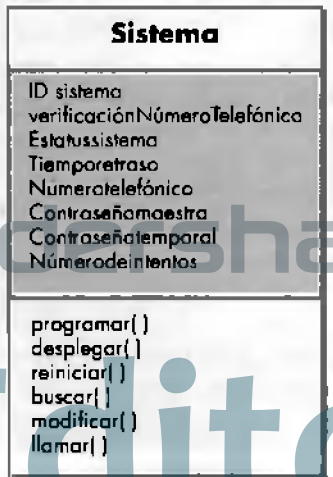
CONSEJO

se definen las para una análisis, el debe estar en to al problema las comporta- requeridos implementa-

Como una primera iteración en la obtención de un conjunto de operaciones para una clase de análisis, el analista puede estudiar otra vez la narrativa de un procesamiento (o caso de uso) y seleccionar aquellas operaciones que pertenezcan de manera razonable a la clase. Esto se logra estudiando de nuevo el análisis gramatical y aislando los verbos. Algunos de estos verbos serán opciones legítimas y podrán conectarse con facilidad a una clase específica. Por ejemplo, en la narrativa del procesamiento presentada párrafos atrás en este capítulo, se observa que "al sensor se le asigna un número y un tipo" o "se programa una contraseña maestra para habilitar y deshabilitar el sistema". Estas frases indican algunos hechos.

FIGURA 8.13

Diagrama de clase para la clase del sistema.



- Que una operación de *asignar( )* es relevante para la clase **sensor**.
- Que una operación de *programar( )* está encapsulada por la clase **sistema**.
- Que *habilitar( )* y *deshabilitar( )* son operaciones que se aplican a la clase **sistema**.

En una investigación posterior tal vez la operación *programar( )* sea dividida en una serie de suboperaciones más específicas que se requieren para configurar el sistema. Por ejemplo, *programar( )* implica la especificación de números telefónicos, la configuración de características del sistema (como al crear la tabla de sensores, al introducir las características de la alarma) y la introducción de contraseña(s). Sin embargo, por ahora *programar( )* se especifica como una sola operación.

HogarSeguro

Modelos de clase

El escenario: Cubículo de Ed, al comenzar el modelado del análisis.

Los actores: Jamie, Vinod y Ed, miembros del equipo de ingeniería del software de *HogarSeguro*.

La conversación:

[Ed ha estado trabajando en la extracción de clases a partir de la plantilla de caso de uso para el **"Acceso a la cámara de vigilancia-despliegue de vistas de cámara"** [presentado en un recuadro anterior en este capítulo] y está mostrando a sus colegas las clases que ha extraído.

Ed: Entonces, cuando el propietario quiere escoger una cámara, él o ella debe elegirla de un plano de planta. He definido una clase llamada **PlanodePlanta.** Aquí está el diagrama.

[Todos miran la figura 8.14.]

Jamie: Entonces **PlanodePlanta** es una clase que se une a las paredes que están compuestas por segmentos de pared, puertas y ventanas, y también las cámaras; eso es lo que significan esas líneas etiquetadas, ¿no?

Ed: Sí, esas líneas se llaman "asociaciones". Las clases están asociadas entre sí de acuerdo con las asociaciones que les he mostrado. [Las asociaciones se estudian en la sección 8.7.5.]

Vinod: Entonces el plano de planta real está hecho de paredes y contiene cámaras y sensores colocados dentro de estas paredes. ¿Cómo sabe el plano de planta dónde colocar esos objetos?

Ed: No lo sabe, pero los otras clases sí. Miren los atributos de abajo, digamos, **SegmentosdePared** los cuales se usan para construir una pared. El segmento de pared tiene unas coordenadas de inicio y final, y la operación de *dibujo( )* hace el resto.

Jamie: Y lo mismo pasa para las puertas y ventanas. Parece como si las cámaras tuvieran unos pocos de atributos extra.

Ed: Sí, los necesito para poder dar la información del movimiento y el acercamiento.

Vinod: Tengo una pregunta. ¿Por qué la cámara tiene una ID, pero las otras clases no?

Ed: Vamos a necesitar identificar cada cámara para propósitos del despliegue.

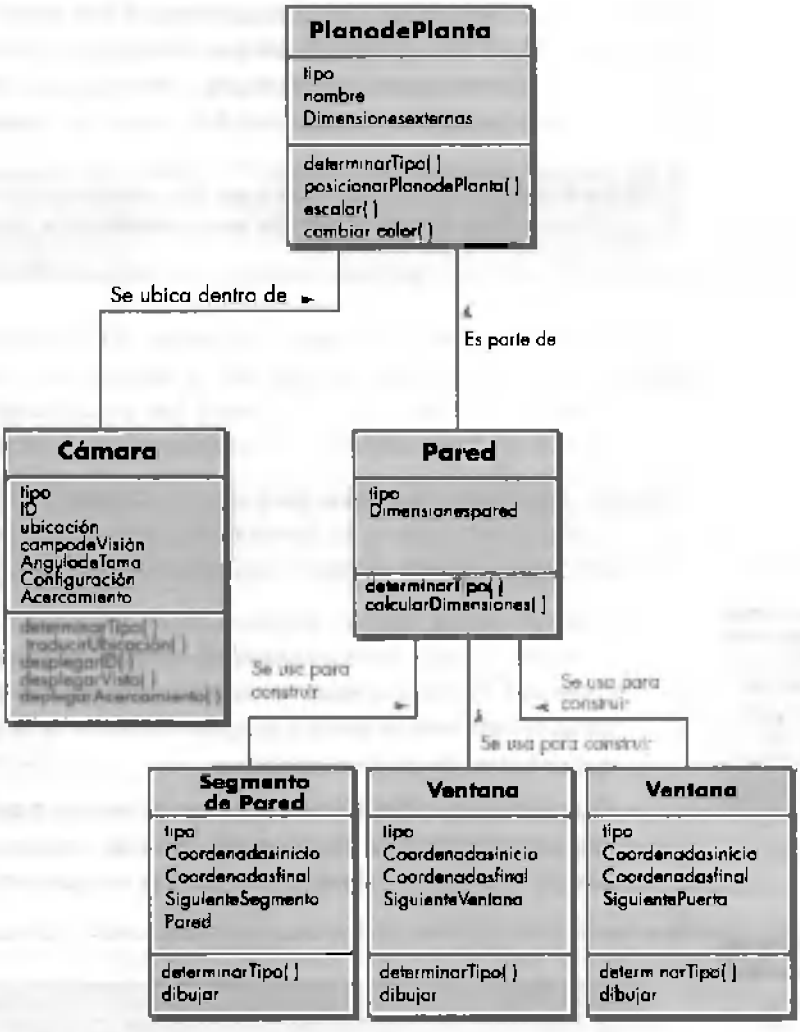
Jamie: Tiene sentido para mí, pero tengo algunas otras preguntas. [Jamie hace preguntas que resultan en modificaciones menores.]

Vinod: ¿Tienes tarjetas CRC para cada una de estas clases? Si es así, debemos seguirlas, sólo hay que estar seguros de que no se ha omitido nada.

Ed: No estoy seguro de cómo hacerlas.

Vinod: No es difícil, y los resultados son muy buenos. Les mostraré.

FIGURA 8.14
Diagrama de clase para PlanodePlanta (véase el análisis en el recuadro).



8.7.4 Modelado de Clase-Responsabilidad-Colaborador (CRC)

El modelado *de Clase-Responsabilidad-Colaborador (CRC)* [WIR90] proporciona un medio simple para identificar y organizar las clases relevantes para los requisitos del sistema o producto. Ambler [AMB95] describe el modelado CRC de la siguiente forma:

> Un modelo CRC en realidad es una colección de tarjetas índice estándar que representan clases. Las tarjetas se dividen en tres secciones. A lo largo del borde superior de la tarjeta se escribe el nombre de la clase. En el cuerpo de la tarjeta se listan las responsabilidades de la clase a la izquierda y los colaboradores a la derecha.

En realidad, el modelo CRC puede utilizar tarjetas índice reales o virtuales. El objetivo es desarrollar una representación organizada de las clases. Las *responsabilidades*

son los atributos y las operaciones relevantes para la clase. Dicho de manera más simple, una responsabilidad es "cualquier cosa que la clase sabe o hace" [AMB95]. Los *colaboradores* son aquellas clases que se requieren para que una clase reciba la información necesaria para completar una responsabilidad. En general, una colaboración implica ya sea una solicitud de información o la solicitud de alguna acción.

> "Uno de los propósitos de las tarjetas CRC es fallar al inicio, fallar constantemente y fallar sin que sea caro. Es mucho más barato tirar un bulto de tarjetas que reorganizar una gran cantidad de código fuente."
>
> C. Horstmann

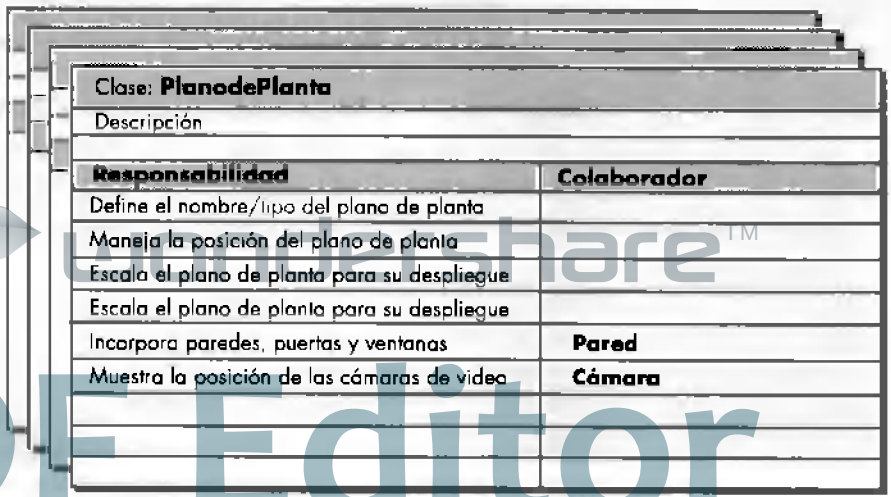
En la figura 8.15 se ilustra una tarjeta índice CRC simple para la clase **Planodeplanta**. La lista de responsabilidades que se muestra en la tarjeta CRC es preliminar y está sujeta a adiciones o modificaciones. Las clases **Pared** y **Cámara** se anotan enseguida de la responsabilidad que requerirá su colaboración.

Clases. Las directrices básicas para identificar clases y objetos ya se han presentado en este mismo capítulo. La taxonomía de los tipos de clases que se presentó en la sección 8.7.1 se puede extender considerando las siguientes categorías:

Referencia Web
En www.[illegible] encontrarse una [illegible] casos.

- *Clases de entidad*, también llamadas clases de *modelo* o *negocios*, se extraen de manera directa del enunciado del problema (por ejemplo, **Planodeplanta** y **Sensor**). De manera típica, estas clases representan clases que se almacenarán en una base de datos y que persisten durante la aplicación (a menos que se borren de manera específica).
- *Clases de frontera*. Se utilizan para crear la interfaz (por ejemplo, pantallas interactivas o reportes impresos) que el usuario ve y con la cual interactúa cuando se utiliza el software. Las clases de entidad contienen información

FIGURA 8.15
Una carta índice del modelo CRC.



importante para los usuarios, pero no se despliegan a sí mismas. Las clases de frontera están diseñadas con la responsabilidad de manejar la forma en que los objetos de entidad se presentan a los usuarios. Por ejemplo, una clase de frontera llamada **VentanadeCámara** tendría la responsabilidad de desplegar la salida de una cámara de vigilancia para el sistema *HogarSeguro*.

- Las *clases de controlador* manejan una "unidad de trabajo" [UML03] desde el inicio hasta el final. Esto es, las clases de controlador se pueden diseñar para manejar 1) la creación o actualización de los objetos de entidad; 2) la inmediatez de objetos de frontera conforme éstos obtienen información de objetos de entidad; 3) la comunicación compleja entre conjuntos de objetos; y 4) la validación de datos comunicados entre los objetos o entre el usuario y la aplicación. En general, las clases de controlador no se consideran sino hasta que ha comenzado el diseño.

> "Los objetos pueden clasificarse de manera científica en tres grandes categorías, los que no funcionan, los que se descomponen y los que se pierden."
>
> **Russell Baker**

Responsabilidades. En las secciones 8.7.2 y 8.7.3 se han presentado las directrices básicas para identificar responsabilidades (atributos y operaciones). Wirfs-Brock y sus colegas [WIR90] sugieren cinco directrices para determinar las responsabilidades de las clases:

¿Cuáles directrices pueden aplicarse para la asignación de responsabilidades a las clases?

1. **La inteligencia del sistema se debe distribuir entre las clases para abordar de mejor manera las necesidades del problema.** Cada aplicación abarca un cierto grado de inteligencia; esto es, lo que el sistema sabe y puede hacer. Esta inteligencia puede distribuirse entre las clases de varias maneras diferentes. Las clases "poco inteligentes" (aquellas que tienen menos responsabilidades) pueden modelarse para actuar al servicio de unas cuantas clases "muy inteligentes" (las que tienen muchas responsabilidades). Aunque este enfoque hace que el flujo de control en un sistema sea directo, tiene unas cuantas desventajas: *a)* concentra toda la inteligencia en unas pocas clases, lo que dificulta los cambios, y *b)* tiende a requerir más clases, y por ende, un mejor esfuerzo de desarrollo.

   Si la inteligencia del sistema se distribuye con mayor amplitud entre las clases de una aplicación, cada objeto sabe y hace sólo unas cuantas cosas (las cuales, por lo general, son bien enfocadas), y la cohesión del sistema mejora. Lo anterior aumenta la facilidad de mantenimiento del software y reduce el impacto de los efectos colaterales debidos al cambio.

   Para determinar si la inteligencia del sistema está bien distribuida las responsabilidades anotadas en cada tarjeta índice del modelo CRC deben evaluarse y así comprobar si alguna clase tiene una lista de responsabilidades

demasiado larga. Esto indica una concentración de inteligencia.[23] Además, las responsabilidades para cada clase deben mostrar el mismo grado de abstracción.

2. **Cada responsabilidad debe establecerse tan general como sea posible.** Esta directriz implica que las responsabilidades generales (tanto atributos como operaciones) deben estar en la parte alta de la jerarquía de la clase (debido a que son genéricas son aplicables en todas las subclases).
3. **La información y el comportamiento relacionado con ella deben estar dentro de la misma clase.** Con esto se logra el principio OO llamado *encapsulación*. Los datos y los procesos que manipulan los datos deben empaquetarse como una unidad cohesiva
4. **La información relativa a una cosa debe localizarse con una sola clase, no distribuirse entre muchas de ellas**. Una sola clase debe tomar la responsabilidad de almacenar y manipular un tipo específico de información En general, esta responsabilidad no se puede compartir entre varias clases Si la información se distribuye, el software se vuelve más difícil de mantener y más desafiante de probar.
5. **Las responsabilidades pueden compartirse entre clases relacionadas cuando esto es apropiado.** Existen muchos casos en los que una variedad de objetos relacionados deben mostrar el mismo comportamiento al mismo tiempo Como un ejemplo, considérese un videojuego que debe desplegar las siguientes clases: **Jugador, CuerpoJugador, BrazosJugador, PiernasJugador, CabezaJugador** Cada una de estas clases tiene sus propios atributos (por ejemplo, **posición, orientación, color, velocidad**) y todos deben actualizarse y desplegarse cuando el usuario manipula un *joystick*. Por lo tanto, las responsabilidades *actualizar( )* y *desplegar( )* deben compartirlas cada uno de los objetos mencionados. El **Jugador** sabe cuando algo ha cambiado y se requiere *actualizar( )*. Colabora con los otros objetos para lograr una nueva posición u orientación, pero cada objeto controla su propio despliegue.

Colaboraciones. Las clases cumplen sus responsabilidades en una de dos formas 1) una clase puede utilizar sus propias operaciones para manipular sus propios atributos, y con ello cumplir con una responsabilidad particular, o 2) una clase puede colaborar con otras clases.

Wirfs-Brock y sus colegas [WIR90] definen las *colaboraciones* de la siguiente manera:

> Las colaboraciones representan las solicitudes que un cliente hace a un servidor con el fin de cumplir una responsabilidad Una colaboración es la materialización del contrato entre el cliente y el servidor. Se dice que un objeto colabora con otro objeto si, para cumplir con una responsabilidad, necesita enviar algunos mensajes al otro objeto Una

23 En tales casos puede ser necesario dividir las clases en múltiples clases o subsistemas completos para distribuir la inteligencia de manera más eficaz

colaboración sencilla fluye en una dirección, lo que representa una solicitud del cliente al servidor. Desde el punto de vista del cliente, cada una de sus colaboraciones está asociada con una responsabilidad particular que ha implementado el servidor.

Las colaboraciones identifican las relaciones entre clases. Cuando un conjunto de clases colabora para lograr algún requisito, éste puede organizarse en un subsistema (un aspecto de diseño).

Las colaboraciones se identifican al determinar si una clase puede cumplir cada responsabilidad por sí misma. Si no es así, entonces se requiere de la interacción con otra clase y, por ende, una colaboración.

Como un ejemplo, considérese la función de seguridad de *HogarSeguro*. Como parte del procedimiento de activación, el objeto **PaneldeControl** debe determinar si algún sensor está abierto. Se define una responsabilidad llamada *determinar-estatus-sensor( )* Si los sensores están abiertos, **PaneldeControl** debe establecer un atributo de **estatus** como "no listo" La información de los sensores se obtiene de cada objeto **sensor**. Por lo tanto, la responsabilidad *determinar-estatus-control( )* trabaja en colaboración con **sensor**

Para ayudarse en la identificación de los colaboradores, el analista puede examinar tres relaciones genéricas diferentes entre las clases [WIR90]: 1) la relación *es-parte-de*, 2) la relación *tiene-conocimiento-de*, y 3) la relación *depende de*. Cada una de las tres relaciones genéricas se considera con brevedad en los siguientes párrafos.

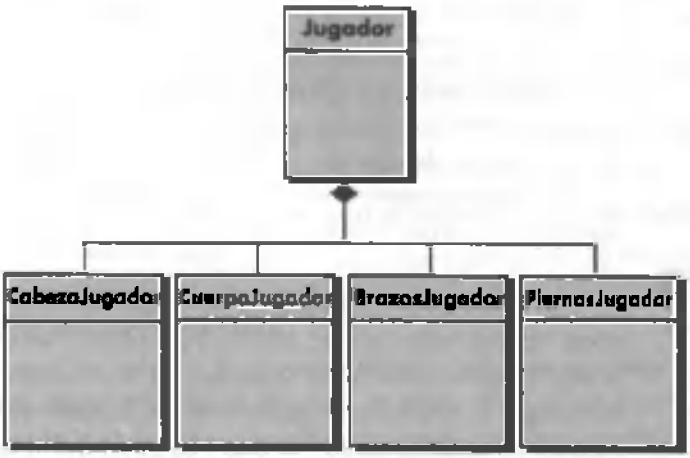
Todas las clases que son parte de una clase agregada se conectan a ésta a través de una relación del tipo *es-parte-de*. Considérense las clases definidas para el videojuego ya mencionado, la clase **CuerpoJugador** *es-parte-de* **Jugador**, al igual que **BrazosJugador, PiernasJugador** y **CabezaJugador**. En UML estas relaciones se representan como la agregación mostrada en la figura 8.16.

Cuando una clase debe obtener información de otra clase se establece la relación *tiene-conocimiento-de* La responsabilidad *determinar-estatus-sensor( )* mencionada antes ejemplifica una relación del tipo *tiene-conocimiento-de*.

La relación *depende-de* implica que dos clases tienen una dependencia que no se logra mediante las relaciones *tiene-conocimiento-de* o *es-parte-de*. Por ejemplo, **cabezaJugador** siempre debe estar conectada a **CuerpoJugador** (a menos que el videojuego sea violento en particular). Aun así, cada objeto puede existir sin el conocimiento directo del otro. Un atributo del objeto **CabezaJugador** llamado **posición central** está determinado desde la posición central de **CuerpoJugador**. Esta información se obtiene a través de un tercer objeto, **Jugador**, quien la adquiere de **CuerpoJugador**. Por ende, **CabezaJugador** *depende-de* **CuerpoJugador.**

En todos los casos, el nombre de la clase del colaborador se registra en la tarjeta índice del modelo CRC enseguida de la responsabilidad que ha producido la colaboración. Por lo tanto, la tarjeta índice contiene una lista de responsabilidades y las colaboraciones correspondientes que permiten que las responsabilidades puedan cumplirse (figura 8.15).

FIGURA 8.16
Una clase agregada compuesta.



Cuando se ha desarrollado un modelo CRC completo, los representantes del clientes y la ingeniería del software pueden revisar el modelo utilizando el siguiente enfoque [AMB95]:

1. Todos los participantes en la revisión (del modelo CRC) reciben un subconjunto de las tarjetas índice del modelo CRC. Las tarjetas que colaboran se deben separar (es decir, ningún revisor debe tener dos que colaboren).
2. Todos los escenarios de caso de uso (y los diagramas de caso de uso correspondientes) deben organizarse en categorías
3. El líder de la revisión lee el caso de uso en forma deliberada. Cuando el líder llega a una clase nombrada pasa una señal a la persona que tiene la tarjeta índice de clase correspondiente. Por ejemplo, un caso de uso para *HogarSeguro* contiene la siguiente narrativa:

   El propietario observa el panel de control de *HogarSeguro* para determinar si el sistema está listo para la entrada. Si no lo está, el propietario podrá cerrar físicamente ventanas y puertas para que se presente el indicador de listo. [Un indicador de no-listo implica que un sensor está abierto; es decir, que esa puerta o ventana está abierta.]

   Cuando el líder de la revisión llega a "panel de control", en la narrativa del caso de uso, la señal se pasa a la persona que posee la carta índice del **Panelde-control**. La frase "implica que un sensor está abierto" requiere que la tarjeta índice contenga una responsabilidad que validará esta implicación (lo cual se logra mediante la responsabilidad *determinar-estatus-sensor( )*). Enseguida de la responsabilidad escrita en la tarjeta está el colaborador **sensor**. Entonces, la señal se pasa a la clase **sensor**.
4. Cuando se pasa la señal, la persona que posee la tarjeta de clase debe describir las responsabilidades anotadas en ella. El grupo determina si una (o más) de las responsabilidades satisface el requisito del caso de uso.

5. Si las responsabilidades y las colaboraciones anotadas en las tarjetas índice no satisfacen el caso de uso, se hacen las modificaciones necesarias a la tarjeta. Esto puede incluir la definición de clases nuevas (y corresponden a las tarjetas de índice de CRC) o la especificación de responsabilidades o colaboraciones nuevas revisadas sobre las tarjetas existentes

Esta forma de operación continúa hasta que se termina con el caso de uso. Cuando se han revisado todos los casos de uso se continúa con el modelado del análisis.

HOGARSEGURO

Modelos CRC

El **escenario:** Cubículo de Ed, durante el modelado del análisis

Los actores: Vinod y Ed, miembros del equipo de ingeniería del software de *HogarSeguro*

La conversación:

[Vinod ha decidido enseñarle a Ed cómo desarrollar tarjetas CRC mediante un ejemplo.]

Vinod: Mientras tú has estado trabajando en la interfaz y Jamie ha estado involucrado con la seguridad, yo he estado trabajando en la función de administración del hogar.

Ed: ¿Cuál es el estatus de eso? Mercadotecnia cambia su punto de vista a cada momento

Vinod: Aquí está el primer corte del caso de uso para esa función. Lo hemos refinado un poco, pero debe darte una idea general.

Caso de uso: Función de administración del hogar de *HogarSeguro*

Narrativa: Queremos utilizar la interfaz de administración del hogar en una PC o con una conexión inalámbrica para controlar dispositivos electrónicos que tengan controladoras de interfaz inalámbricos. El sistema debe permitirme encender y apagar luces específicos, ejecutar aplicaciones conectadas a una interfaz inalámbrica, configurar los sistemas de calefacción y aire acondicionado con las temperaturas que yo defina; para esto, quiero seleccionar los dispositivos de un plano de planta de la casa. Cada dispositivo debe estar identificado sobre el plano de la planta. Como una característica opcional, quiero controlar todos los dispositivos audiovisuales: audio, televisión, DVD, grabadoras digitales, etcétera

Con una sola selección, quiero ser capaz de configurar la casa completa para varias situaciones. Una es *en casa*; la otra, *fuera de casa*; una tercera, *salida por la noche*, y una cuarta, *viaje largo*. Todas estas situaciones tendrán configuraciones que se aplicarán a todos los dispositivos. En los estados *salida por la noche* y *viaje largo* el sistema debe encender y apagar luces a intervalos aleatorios (para aparentar que alguien está en casa) y controlar el sistema de calefacción y aire acondicionado. Debo ser capaz de invalidar estas configuraciones a través de Internet con la protección de una contraseña apropiada.

Ed: ¿La gente de hardware tiene concebidas todas las interfases inalámbricas?

Vinod (sonriendo): Están trabajando en ellas, digamos que no es un gran problema. De cualquier manera, extraje una serie de clases para la administración del hogar, y podemos utilizar alguna de ellas como ejemplo. Usemos la clase **InterfazAdministraciónHogar.**

Ed: De acuerdo. . entonces, las responsabilidades son... los atributos y operaciones para la clase, y las colaboraciones son las clases hacia las que apuntan las responsabilidades

Vinod: Creo que no entendiste la CRC.

Ed: Tal vez un poco, pero continúa

Vinod: Entonces, aquí está mi definición de clase para **InterfazAdministraciónHogar**

Atributos:

Panelopciones: proporciona información en botones que permiten al usuario seleccionar una funcionalidad.

Panelsituacion: proporciona información en botones que permiten al usuario seleccionar la situación

PlanodePlanta el mismo que el objeto de vigilancia, pero este despliega los dispositivos

IconosdeDispositivo: información sobre iconos que representan luces, aplicaciones, cámaras, etcétera.

PanelesdeDispositivo: simulación de una aplicación o panel de control de un dispositivo; permite el control.

Operaciones:

DesplegarControl(), seleccionarControl(), desplegarSituación(), seleccionarSituación(), entrarPlanodePlanta(), seleccionarIconoDispositivo(), desplegarPanelDispositivo(), entrar PanelDispositivo(),..

Clase: *InterfazAdministraciónHogar*

Responsabilidad	Colaborador
desplegarControl	**Paneloperaciones** (clase)
seleccionarControl	**Paneloperaciones** (clase)
desplegarSituación	**PanelSituación** (clase)
seleccionarSituación	**PanelSituación** (clase)
entrarPlanodePlanta	**PlanodePlanta** (clase)
•	
•	
•	

Ed: Entonces cuando se invoca la operación *entrarplanodePlanta()*, ésta colabora con el objeto **PlanodePlanta** como el que desarrollamos para la vigilancia. Espera, aquí tengo su descripción. (Ven la figura 8.14)

Vinod: Exactamente. Y si quisiéramos revisar todo el modelo de clase, podríamos comenzar con esta carta índice, después ir a la carta índice del colaborador, y de ahí, a una de los colaboradores de los colaboradores, y así sucesivamente.

Ed: Buena forma de encontrar omisiones o errores

Vinod: Si.

Una asociación define una relación entre clases. La multiplicidad define cuántos de una clase están relacionados con cuántos de otra clase.

8.7.5 Asociaciones y dependencias

En muchos casos, dos clases de análisis se relacionan entre sí de alguna manera, parecida a la forma en que se relacionan dos objetos de datos (sección 8.3.3). En UML estas relaciones se llaman *asociaciones*. Véase de nuevo la figura 8.14; la clase **PlanodePlanta** se define al identificar un conjunto de asociaciones entre **PlanodePlanta** y otras dos clases, **Cámara** y **Pared**. La clase **Pared** se asocia con tres clases que permiten que se construya una pared, **SegmentodePared, Ventana** y **Puerta**.

En algunos casos, una asociación se puede definir en forma más extensa al indicar *multiplicidad* (el término *cardinalidad* ya se usó antes en este capítulo). En referencia a la figura 8.14, un objeto de **Pared** se construye con uno o más objetos de **SegmentosdePared**. Además, el objeto **Pared** puede contener 0 o más objetos de **Ventana** y 0 o más objetos de **Puerta**. Estas restricciones de multiplicidad se ilustran en la figura 8.17, donde "uno o más" se representa mediante 1..* y "0 o más" por medio 0..*. En UML el asterisco indica una frontera superior ilimitada en el rango.

En muchos casos existe una relación cliente-servidor entre dos clases de análisis. En tales casos, una clase de cliente depende de alguna manera de la clase de servidor y se establece una *relación independencia*. Las dependencias se definen mediante un estereotipo. Un *estereotipo* es un "mecanismo de extensibilidad" [ARL02] den

24 Otras relaciones de multiplicidad —una a una, una a muchas, muchas a muchas, una a un rango específico con límites inferior y superior, y otras— se pueden indicar como parte de una asociación

Figura 8.17
Multiplicidad.

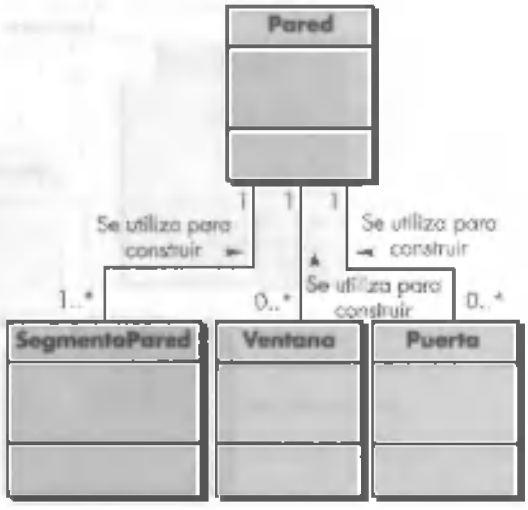
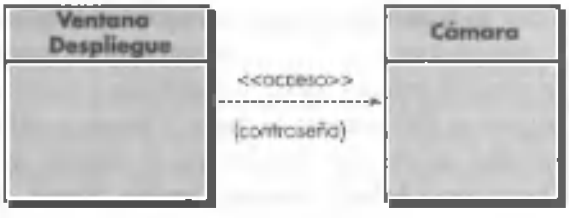


Figura 8.18
Dependencias.



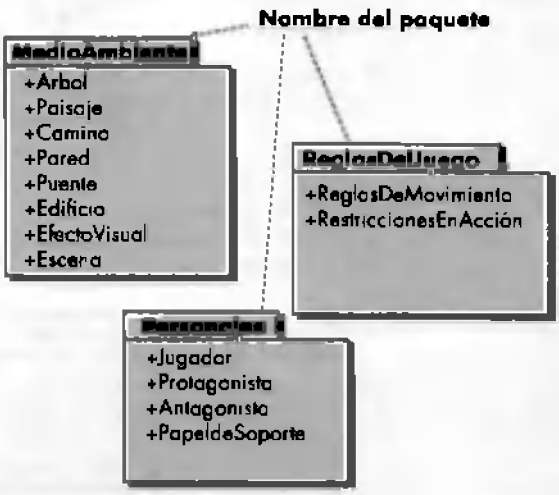
tro del UML que permite a un ingeniero de software definir un elemento de modelado especial cuya semántica define el cliente. En UML los estereotipos se representan dentro de comillas angulares (por ejemplo, <<estereotipo>>).

Como una ilustración de una dependencia simple dentro del sistema de vigilancia de *HogarSeguro*, un objeto de **Cámara** (en este caso, la clase de servidor) proporciona una imagen de video o un objeto de **VentanadeDespliegue** (en este caso, la del cliente). La relación entre estos dos objetos no es una asociación simple, aun así existe una asociación de dependencia. En un caso de uso escrito para la vigilancia (que no se muestra), el modelador aprende que se debe proporcionar una contraseña especial para ver ubicaciones específicas de cámara. Una forma de lograr esto es que **Cámara** pida una contraseña y después dé permiso a **VentanadeDespliegue** para producir la imagen de video. Esto se puede representar como se muestra en la figura 8.18, donde <<acceso>> implica que el uso de la salida de la cámara está controlado mediante una contraseña especial.

8.7.6 Paquetes de análisis

Una parte importante del modelado del análisis es la categorización. Esto es, los diferentes elementos del modelo de análisis (por ejemplo, casos de uso, clases de

FIGURA 8.19
Paquetes.



análisis) se clasifican de una manera que los empaquete como una agrupación —llamada un *paquete de análisis*—, a la cual se le asigna un nombre representativo.

¡PUNTO CLAVE

Un paquete se utiliza para ensamblar una colección de clases relacionadas.

Para ilustrar la utilización de paquetes de análisis considérese el videojuego que se presentó párrafos atrás. Al desarrollar el modelo de análisis para el videojuego se obtiene un gran número de clases. Algunas se enfocan en el ambiente del juego; por ejemplo, las escenas visuales que el usuario ve mientras se desarrolla el juego. Clases como **Árbol, Paisaje, Camino, Pared, Puente, Edificio, EfectoVisual** podrían estar dentro de esta categoría. Otras se enfocan en los personajes dentro del juego al describir sus características físicas, acciones y restricciones. Se podrían definir clases como **Jugador** (la cual se describió con anterioridad), **Protagonista**, **Antagonista** y **PapelesdeSoporte**. Además, otras describen las reglas del juego: cómo navega el jugador a través del ambiente. Aquí son candidatas clases como **ReglasDeMovimiento** y **RestriccionesEnAcción**. Pueden existir muchas otras categorías. Estas clases pueden representarse como los paquetes de análisis que se muestran en la figura 8.19.

El signo de más que precede al nombre de la clase de análisis en cada paquete indica que las clases tienen visibilidad pública y que, por lo tanto, son accesibles desde otros paquetes. Aunque no se muestran en esta figura, hay otros símbolos que pueden preceder a un elemento dentro de un paquete. Un signo de menos indica que un elemento está oculto de todos los otros paquetes, y un símbolo # indica que un elemento es accesible sólo a las clases contenidas dentro de un paquete dado.

8.8 CREACIÓN DE UN MODELO DE COMPORTAMIENTO

Los diagramas de clase, las tarjetas índice CRC y otros modelos orientados a las clases tratados en la sección 8.7 representan elementos estáticos del modelo de análi-

sis. Ahora es tiempo de hacer una transición al comportamiento dinámico del sistema o producto. Para lograrlo el comportamiento del sistema debe presentarse como una función de los elementos específicos y el tiempo.

El *modelo de comportamiento* indica la forma en que el software responderá a los eventos o estímulos externos. En la creación del modelo el analista debe realizar los siguientes pasos:

1. Evaluar todos los casos de uso para entender por completo la secuencia de interacción dentro del sistema.
2. Identificar los eventos que conducen la secuencia de interacción y entender la forma en que estos eventos se relacionan con las clases específicas.
3. Crear una secuencia para cada caso de uso.
4. Construir un diagrama de estado para el sistema
5. Revisar el modelo de comportamiento para verificar su exactitud y consistencia.

Cada uno de estos pasos se expone en las secciones siguientes

8.8.1 Identificación de eventos con el caso de uso

Como se mencionó en la sección 8.5, el caso de uso representa una secuencia de actividades que implica a los actores y al sistema En general, siempre que el sistema y un actor intercambian información ocurre un evento. Si se recuerda la explicación anterior acerca del modelado del comportamiento en la sección 8.6.3, será importante puntualizar que el evento no es la información que se ha intercambiado, sino el *hecho* de que la información haya sido intercambiada.

Los puntos del intercambio de información se obtienen examinando un caso de uso. A manera de ilustración, se reconsiderará el caso de uso para una pequeña parte de la función de seguridad de *HogarSeguro*.

> El propietario utiliza el teclado para introducir una contraseña de cuatro dígitos. La contraseña se compara con la contraseña válida almacenada en el sistema. Si la contraseña es incorrecta, el panel de control emitirá un sonido por una sola vez y se reiniciará para esperar otra entrada. Si la contraseña es correcta, el panel de control esperará la acción posterior.

Las partes subrayadas del escenario del caso de uso indican eventos. Se debe identificar a un autor para cada evento; la información intercambiada se debe anotar, y cualquier condición o restricción debe registrarse

Como ejemplo de un evento típico, considérese la frase subrayada del caso de uso "el propietario utiliza el teclado para introducir una contraseña de cuatro dígitos". En el contexto del modelo de análisis, el objeto, **propietario**,[25] transmite un evento al

25 En este ejemplo se asume que cada usuario (propietario) que interactúa con *HogarSeguro* tiene una contraseña de identificación y que, por lo tanto, es un objeto legítimo

objeto **PaneldeControl**. El evento podría llamarse *introducción de contraseña*. La información transferida son los cuatro dígitos que constituyen la contraseña. Aunque ésta no es una parte esencial del modelo de comportamiento. Es importante señalar que algunos eventos tienen un impacto explícito sobre el flujo de control del caso de uso, mientras que otros no tienen impacto directo sobre el flujo de control. Por ejemplo, el evento *introducción de contraseña* no cambia de manera explícita el flujo de control del caso de uso, pero los resultados del evento *comparación de contraseña* (derivado de la interacción "la contraseña se compara con la contraseña válida almacenada en el sistema") tendrá un impacto explícito sobre la información y el flujo de control del software de *HogarSeguro*.

Una vez que se han identificado todos los eventos, éstos se ubican con los objetos involucrados. Los objetos pueden ser responsables de generar eventos (por ejemplo, **Propietario** genera el evento de *introducción de contraseña*) o de recibir eventos que han ocurrido en cualquier sitio (por ejemplo, **PaneldeControl** recibe el resultado binario del evento *comparación de contraseña*).

8.8.2 Representaciones de estado

En el contexto del modelado del comportamiento se pueden considerar dos diferentes caracterizaciones de los estados: 1) el estado de cada clase conforme el sistema realiza su función, y 2) el estado del sistema como se observa desde el exterior conforme el sistema realiza su función.[26]

PUNTO CLAVE

El sistema tiene estados que representan un comportamiento específico observable desde el exterior, una clase tiene estados que representan su comportamiento cuando el sistema realiza sus funciones.

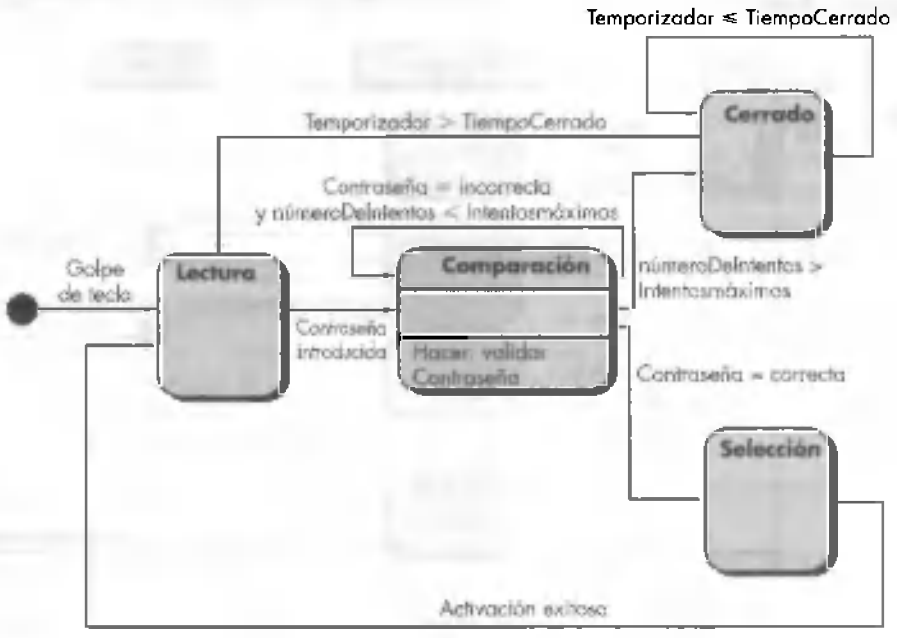
El estado de una clase implica características tanto pasivas como activas [CHAS]. Un *estado pasivo* es simplemente el estatus actual de todos los atributos de un objeto. Por ejemplo, el estado pasivo de la clase **Jugador** (en la aplicación de videojuego estudiada con anterioridad) incluiría los atributos de **posición** y **orientación** del **Jugador**, así como otras características relevantes para el juego (por ejemplo, un atributo que indique la **existencia de deseos mágicos**). El *estado activo* de un objeto indica el estatus actual del objeto cuando éste está sujeto a una transformación o a un procesamiento continuos. La clase **Jugador** podría tener los siguientes estados activos: *en movimiento, en descanso, herido, en curación, atrapado, perdido*, etcétera. Debe ocurrir un evento (algunas veces llamado un *disparador*) para obligar a un objeto a hacer una transición de un estado activo a otro.

En los párrafos siguientes se explican dos diferentes representaciones del comportamiento. La primera indica la forma en que una clase individual cambia sus estados con base en eventos externos, y la segunda muestra el comportamiento del software como una función del tiempo.

Diagramas de estado para clases de análisis. Un componente de un modelo del comportamiento es un diagrama de estado en UML que representa los estados activos para cada clase y los eventos (disparadores) que ocasionan cambios entre

26 Los diagramas de estado presentados en la sección 8.6.3 muestran el estado del sistema. La exposición en esta sección se enfocará al estado de cada clase dentro del modelo de análisis.

Figura 8.20
Diagrama de estado para la clase PanelDeControl.



estos estados activos. En la figura 8.20 se ilustra un diagrama de estado para la clase **PaneldeControl** en la función de seguridad de *HogarSeguro*

Cada flecha en la figura 8.20 representa una transición desde un estado activo de una clase hasta otro. Las etiquetas mostradas para cada flecha representan el evento que dispara la transición. Aunque el modelo de estado activo proporciona un discernimiento activo de la "historia de vida" de una clase, es posible especificar información adicional para proporcionar mayor profundidad en la comprensión del comportamiento de una clase Además de especificar el evento que ocasiona la transición, el analista puede especificar una guardia y una acción [CHA93]. Una *guardia* es una condición booleana que debe satisfacerse para que ocurra la transición Por ejemplo, la guardia para la transición desde el estado de "lectura" al estado de "comparación" de la figura 8 20 se puede determinar al examinar el caso de uso:

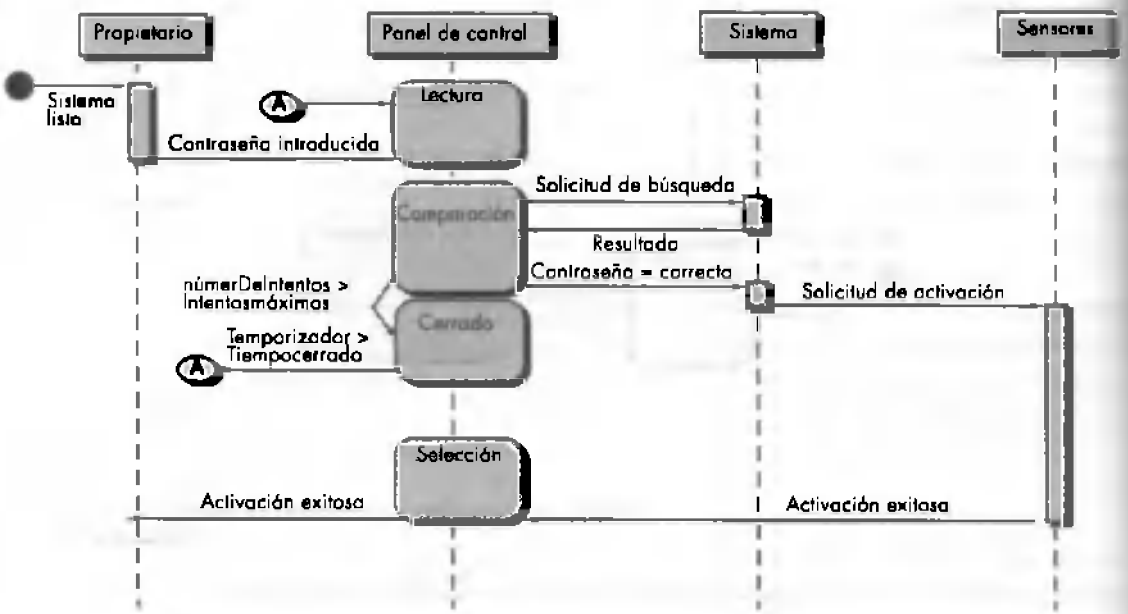
si (introducción de contraseña = 4 dígitos), entonces comparar con contraseña almacenada

En general, la guardia para una transición por lo regular depende del valor de uno o más atributos de un objeto. En otras palabras, la guardia depende del estado pasivo del objeto

Una *acción* sucede de manera concurrente con la transición del estado o como consecuencia de éste, y por lo general implica una o más operaciones (responsabilidades) del objeto. Por ejemplo, una acción conectada con el evento *contraseña introducida* (figura 8.20) es una operación llamada ***ValidarContraseña( )*** que da acceso a

CLAVE
de un
de estado
al
sin
las clases
un
de secuencia
nto al
forma en
clases se
de estado o

FIGURA 8.21 Diagrama de secuencia (parcial) para la función de seguridad de *HogarSeguro*.



un objeto de **contraseña** y realiza una comparación dígito por dígito para validar la contraseña introducida

Diagramas de secuencia. El segundo tipo de representación de comportamiento, llamado un *diagrama de secuencia* en UML, indica cómo los eventos causan transiciones de objeto a objeto. Una vez que se han identificado los eventos al examinar un caso de uso, el modelador crea un diagrama de secuencia: una representación de cómo los eventos causan un flujo de un evento a otro como una función del tiempo. En esencia, el diagrama de secuencia es una versión abreviada del caso de uso. Representa clases clave y eventos que causan que el comportamiento fluya de clase a clase.

En la figura 8.21 se ilustra un diagrama de secuencia parcial de la función de seguridad de *HogarSeguro*. Cada flecha representa un evento (derivado de un caso de uso) e indica cómo el evento canaliza el comportamiento entre los objetos de *HogarSeguro*. El tiempo se mide de manera vertical (hacia abajo), y los rectángulos verticales delgados representan el tiempo invertido en procesar una actividad. Los estados se pueden mostrar a lo largo de una línea de tiempo vertical.

El primer evento, *sistema listo*, se deriva del ambiente externo y canaliza el comportamiento a un objeto de **propietario**. El propietario introduce una contraseña. Se pasa un evento de *solicitud de búsqueda* al **sistema**, el cual busca la contraseña en una base de datos simple y regresa un *resultado* (*encontrado* o *no encontrado*) al **PaneldeControl** (ahora en el estado de *comparación*). Una contraseña válida resul-

ta en un evento *contraseña=correcta* para el **Sistema,** el cual activa los sensores con un evento de *solicitud de activación*. Por último, el control se pasa de regreso al propietario con el evento *activación exitosa*.

Una vez que se ha desarrollado un diagrama de secuencia completo, todos los eventos que ocasionan transiciones entre objetos del sistema se pueden cotejar con un conjunto de eventos de entrada y eventos de salida (de un objeto). Esta información es útil en la creación de un diseño eficaz para el sistema que será construido

HERRAMIENTAS DE SOFTWARE

Modelado del análisis generalizado en UML

Objetivo: Las herramientas para el modelado del análisis proporcionan la capacidad de modelos basados en escenarios, modelos en clases y modelos de comportamiento mediante UML.

: Las herramientas en esta categoría soportan rango de diagramas en UML requeridos para un modelo de análisis (estas herramientas soportan el modelado del diseño). Además de la diagramas, las herramientas en esta categoría la verificación de la consistencia y la corrección los diagramas en UML; 2) proporcionan vínculos diseño y la generación de código; 3) construyen de datos que ayudan a la administración y de grandes modelos en UML requeridos para complejos.

as representativas[27]

herramientas soportan un rango completo en UML requeridos para el modelado del

ArgoUML, una herramienta de fuente abierta (argouml.tigris.org).

Control Center, desarrollado por TogetherSoft (www.togethersoft.com).

Enterprise Architect, desarrollado por Sparx Systems (www.sparxsystems.com.au).

Object Technology Workbench (OTW), desarrollado por OTW Software (www.otwsoftware.com)

Power Designer, desarrollado por Sybase (www.sybase.com).

Rational Rose, desarrollado por Rational Corporation (www.rational.com)

System Architect, desarrollado por Popkin Software (www.popkin.com)

UML Studio, desarrollado por Pragsoft Corporation (www.pragsoft.com).

Visio, desarrollado por Microsoft (www.microsoft.com).

Visual UML, desarrollado por Visual Object Modelers (www.visualuml.com).

8.9 RESUMEN

El objetivo del modelado del análisis es crear una variedad de representaciones que muestran los requisitos del software para la información, la función y el comportamiento. Esto se logra aplicando dos diferentes filosofías de modelado (pero potencialmente complementarias): el análisis estructurado y el análisis orientado a objetos. El análisis estructurado considera al software como un transformador de infor-

27 Las herramientas mencionadas aquí son una muestra de esta categoría. En la mayoría de los casos los nombres están registrados por sus respectivos desarrolladores.

mación. Éste ayuda al ingeniero de software a identificar los objetos de datos, sus relaciones y la manera en la cual estos objetos de datos se transforman mientras fluyen a través de las funciones de procesamiento del software. El análisis orientado a objetos examina un dominio de problema definido como un conjunto de casos de uso en un esfuerzo por extraer clases que definen el problema. Cada clase tiene un conjunto de atributos y operaciones. Las clases están relacionadas entre sí en una variedad de formas diferentes y se moldean mediante la aplicación de diagramas de UML. El modelo de análisis lo componen cuatro elementos de modelado: modelos basados en escenarios, modelos de flujo, modelos basados en clases y modelos de comportamiento.

Los modelos basados en escenarios muestran los requisitos del software desde el punto de vista del usuario. El caso de uso —una descripción narrativa o basada en una plantilla de una interacción entre un actor y el software— es el elemento primario del modelado. El caso de uso, derivado durante la obtención de requisitos, define los casos clave para una función o interacción específica. El grado de formalidad y detalle del caso de uso varía, pero el resultado final proporciona la entrada necesaria a las otras actividades de modelado del análisis. Los escenarios también pueden describirse por medio de un diagrama de actividad: una representación gráfica del tipo de un diagrama de flujo que muestra el flujo del procesamiento dentro de un escenario específico. Los diagramas de carril ilustran la forma en que el flujo de procesamiento incumbe a varios actores o clases.

Los modelos de flujo se enfocan en el flujo de objetos de datos conforme las funciones de procesamiento los transforman. Los modelos de flujo, que se derivan del análisis estructurado, utilizan el diagrama de flujo de datos; ésta es una notación de modelado que muestra la manera en que una entrada se transforma en una salida conforme los objetos de datos se mueven a través del sistema. Cada función del software que transforma datos se describe mediante una especificación del proceso narrativa. Además del flujo de datos, este elemento de modelado también muestra el flujo de control (una representación que ilustra la forma en que los eventos afectan el comportamiento del sistema).

El modelado basado en clases utiliza información derivada de elementos de modelado orientado al flujo y basado en escenarios para extraer clases candidatas, atributos y operaciones de las narrativas basadas en texto. Se establecen los criterios para la definición de una clase. La tarjeta índice clase-responsabilidad-colaborador puede usarse en la definición de relaciones entre las clases. Además, se puede aplicar una variedad de notaciones de modelado en UML para definir jerarquías, relaciones, asociaciones, agregaciones y dependencias entre las clases. Los paquetes de análisis se utilizan para categorizar y agrupar clases de manera que sean más manejables para los sistemas grandes.

Los primeros tres elementos del modelado del análisis proporcionan una visión estática del software. Los modelos de comportamiento muestran un desempeño dinámico. El modelo de comportamiento utiliza la entrada de elementos basados

escenarios, orientados al flujo y basados en clases para representar los estados de las clases de análisis y del sistema como un todo. Esto se logra identificando los estados, definiendo los eventos que ocasionan que una clase (o sistema) tenga una transición de un estado a otro, e identificando las acciones que suceden cuando se realiza la transición. En el modelado del comportamiento se utiliza la notación en UML de los diagramas de estado y los diagramas de secuencia

REFERENCIAS

[ABB83] Abbott, R., "Program Design by Informal English Descriptions", en *CACM*, vol. 26, núm. 11, noviembre de 1983, pp. 892-894

[AMB95] Ambler, S., "Using Use-Cases", en *Software Development*, julio de 1995, pp. 53-61

[ARA89] Arango, G. y R. Prieto-Diaz, "Domain Analysis: Concepts and Research Directions", en *Domain Analysis: Acquisition of Reusable Information for Software Construction*, (Arango, G. y R. Prieto-Diaz, eds.), IEEE Computer Society Press, 1989

[ARL02] Arlow, J. e I. Neustadt, *UML and the Unified Process*, Addison-Wesley, 2002

[BER93] Berard, E. V., *Essays on Objetc-Oriented Software Engineering*, Addison-Wesley, 1993

[BOO86] Booch, G., "Object-Oriented Development", en *IEE Trans. Software Engineering*, vol. SE-12, núm. 2, febrero de 1986, pp. 211ff

[BUD96] Budd, T., *An Introduction to Objetc Oriented Programming*, 2a. ed., Addison-Wesley, 1996.

[CAS89] Cashman, M., "Object-Oriented Domain Analysis", en *ACM Software Engineering Notes*, vol. 14, núm. 6, octubre de 1989, p. 67

[CHA93] de Champeaux, D., D. Lea y P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993

[CHE77] Chen, P., *The Entity-Relationship Approach to Logical Database Design*, QED Information Systems, 1977.

[COA91] Coad, P. y E. Yourdon, *Object Oriented Analysis*, 2a. ed., Prentice-Hall, 1991.

[COC01] Cockburn, A., *Writing Effective Use Cases*, Addison-Wesley, 2001

[DAV93] Davis, A., *Software Requirements: Objects, Functions and States*, Prentice-Hall, 1993.

[DEM79] DeMarco, T., *Structured Analysis and System Specification*, Prentice-Hall, 1979

[FIR93] Firesmith, D. G., *Object-Oriented Requirements Analysis and Logical Design*, Wiley, 1993

[LET03] Lethbridge, T., comunicación personal sobre el análisis del dominio, mayo de 2003.

[OMG03] Object Management Group, OMG Unified Modeling Language Specification, versión 1.5, marzo de 2003, disponible en http://www.rational.com/uml/resources/documentation/

[SCH02] Schmuller, J., *Teach Yourself UML in 24 Hours*, 2a. ed., SAMS Publishing 2002.

[SCH98] Schneider, G. y J. Winters, *Applying Use Cases*, Addison-Wesley, 1998

[STR88] Stroustrup, B., "What is Object-Oriented Programming?", en IEEE Software, vol. 5, núm. 3, mayo de 1988, pp. 10-20

[TAY90] Taylor, D. A., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990

[THA00] Thalheim, B., *Entity Relationship Modeling*, Springer-Verlag, 2000.

[TIL93] Tillmann, G., *A Practical Guide to Logical Data Modeling*, McGraw-Hill, 1993.

[UML03] The UML Café, "Customers Don't Print Themselves", disponible en http://www.theumlcafe.com/a0079.htm, mayo de 2003.

[WIR90] Wirfs-Brock, R., B. Wilkerson y L. Weiner, *Designing Object-Oriented Software*, Prentice-Hall, 1990.

PROBLEMAS Y PUNTOS A CONSIDERAR

8.1. ¿Es posible comenzar a codificar inmediatamente después de haber creado el modelo de análisis? Explicar la respuesta y después justificar los puntos en contra.

8.2. Un análisis práctico es aquel en el cual el modelo "debe enfocarse en los requisitos que son visibles dentro de los dominios del problema o del negocio". ¿Cuáles tipos de requisitos *no* son visibles en estos dominios? Dar algunos ejemplos.

8.3. ¿Cuál es el propósito del análisis del dominio? ¿Cómo se relaciona con el concepto de los patrones de requisitos?

8.4. En unas cuantas líneas, trátese de describir las diferencias primordiales entre el análisis estructurado y el análisis orientado a objetos

8.5. ¿Es posible desarrollar un modelo de análisis eficaz sin desarrollar los cuatro elementos que se muestran en la figura 8.3? Explicar la respuesta.

8.6. Supóngase que han pedido construir uno de los siguientes sistemas.

a) Un registro de cursos basado en red para una universidad.
b) Un sistema procesador de órdenes basado en Internet para una tienda de computadoras
c) Un sistema simple de facturación para un negocio pequeño.
d) El software que reemplace un Rolodex y que se encuentre dentro de un teléfono inalámbrico.
e) Un libro de cocina automático que esté construido dentro de un horno eléctrico o de microondas.

Selecciónese el sistema que se considere interesante y descríbanse sus objetos de datos, relaciones y atributos

8.7. Dibujar un modelo al nivel de contexto (DFD de nivel 0) para uno de los cinco sistemas que se listan en el problema 8.6. Escribir una narrativa del procesamiento para el sistema al nivel de contexto.

8.8. Utilizar el DFD al nivel de contexto desarrollado en el problema 8.7 para dibujar los diagramas de flujo de los niveles 1 y 2. Para comenzar, utilícese un "análisis gramatical" en la narrativa del procesamiento al nivel de contexto. Recuérdese especificar todo el fluido de información mientras rotula todas las flechas que se encuentran entre las burbujas. Úsense nombres significativos para cada transformación

8.9. Desarróllense especificaciones de control (EC) y especificaciones de proceso (EP) para el sistema que seleccionó en el problema 8.6. Trátese de que el modelo sea lo más completo posible.

8.10. El departamento de obras públicas de una ciudad grande ha decidido desarrollar un sistema de rastreo y reparación de baches basado en la Web (SRRB). Se incluye la siguiente descripción:

> Los ciudadanos pueden entrar al sitio Web y reportar la ubicación y severidad de los baches. Cuando éstos se reportan entran a un "sistema de reparación del departamento de obras públicas", donde se les asigna un número de identificación, junto con la dirección de la calle, el tamaño (en una escala de 0 a 10), la ubicación (en la orilla de la calle, en medio, etcétera), el distrito (determinado por la dirección de la calle), y la urgencia de la reparación (determinada por el tamaño del bache). Los datos de la orden de trabajo están asociados con cada bache e incluyen la ubicación y el tamaño del bache, número de identificación de la reparación, cantidad de personal necesario, horas aplicadas a la reparación, estado del bache (trabajo en progreso, reparado, reparado en forma temporal, no reparado), cantidad de material de relleno utilizado, y costo de la reparación (cálculo de las horas aplicadas, número de personas, material y equipo utilizados). Por último, se crea un archivo de daños para registrar información sobre averías reportadas debido a los baches, el cual incluye nombre del ciudadano, dirección, número telefónico, tipo de daño, precio del daño en dólares. El SRRB es un sistema basado en la Web; todas las peticiones se hacen en forma interactiva

Con base en una notación de análisis estructurada, desarrolle un modelo de análisis para el SRRB

8.11. Describir los términos orientados a objetos *encapsulación* y *herencia*.

8.12. Escribir un caso de uso basado en una plantilla para el sistema de gestión para el hogar *HogarSeguro*, descrito de manera informal en un recuadro ubicado enseguida de la sección 8.7.4.

8.13. Dibujar un diagrama de caso de uso en UML para el sistema SRRB presentado en el problema 8. 10 Tendrán que hacerse varios supuestos sobre la manera en que el usuario interactúa con este sistema

8.14. Desarrollar un modelo de clase para el sistema SRRB presentado en el problema 8.10

8.15. Desarrollar un conjunto completo de tarjetas índice del modelo CRC para el sistema SRRB presentado en el problema 8 10.

8.16. Encabezar una revisión de las tarjetas índice de CRC con sus colegas. ¿Cuántas clases adicionales, responsabilidades y colaboradores fueron agregados como consecuencia de la revisión?

8.17. Describir la diferencia entre una asociación y una dependencia para una clase de análisis.

8.18. ¿Qué es un paquete de análisis y cómo debe utilizarse?

8.19. ¿De qué manera difiere un diagrama de estado para clases de análisis de los diagramas de estado presentados para el sistema completo?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Se han publicado docenas de libros sobre análisis estructurado En la mayoría se trata el tema de una manera adecuada, pero sólo en unos cuantos se presenta un trabajo en verdad excelente. DeMarco y Plauger (*Structured Analysis and System Specification*, Pearson, 1985) es un clásico que sigue siendo una buena introducción a la notación básica. Los libros de Kendal y Kendal (*Systems Analysis and Design*, 5a ed., Prentice-Hall, 2002) y Hoffer *et al.* (*Modern Systems Analysis and Design*, Addison-Wesley, 3a. ed., 2001) son referencias valiosas El libro de Yourdon (*Modern Structured Analysis*, Yourdon-Press, 1989) sobre el tema se conserva entre las publicaciones más completas a la fecha.

Allen (*Data Modeling for Everyone*, Wrox Press, 2002), Simpson y Witt (*Data Modeling Essentials*, 2a. ed , Coriolis Group, 2000), Reingruber y Gregory (*Data Modeling Handbook*, Wiley, 1995) presentan guías detalladas para crear modelos de datos relacionados con la calidad industrial. Un interesante libro de Hay (*Data Modeling Patterns*, Dorset House, 1995) presenta patrones de modelos de datos típicos que se encuentran en muchos negocios diferentes Un tratamiento detallado del modelado del comportamiento puede encontrarse en Kowal (*Behavior Models: Specifying User's Expectations*, Prentice-Hall, 1992).

Los casos de uso son la base del análisis orientado a objetos Los libros de Bittner y Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn [COC01], Armour y Miller (*Advanced Use-Case Modeling: Software Systems*, Addison-Wesley, 2000) y Rosemberg y Scot (*Use-Case Driven Object Modeling with UML: A Practical Approach*, Addison-Wesley, 1999) proporcionan una guía valiosa en la creación y uso de este importante mecanismo de representación y logro de requerimientos.

Arlow y Neustadt han escrito apreciables análisis del UML [ARL02], Schmuller [SCH02], Fowler y Scott (*UML Distilled*, 2a. ed , Addison-Wesley, 1999), Booch y sus colegas (*The UML User Guide*, Addison-Wesley, 1998) y Rumbaugh y sus colegas (*The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998).

Los métodos de análisis y diseño que apoyan el proceso unificado los explica Larman (*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified process*, 2a. ed., Prentice-Hall, 2001), Dennis y sus colegas (*System Analysis and Design: An Object-Oriented Approach with UML*, Wiley, 2001), y Rosenberg y Scott (*Use-Case Driven Object Modeling with* UML, Addison-Wesley, 1999), Balcer y Mellor (*Executable UML: A Foundation for*

Model Driven Architecture, Addison-Wesley, 2002) exponen la semántica general del UML, los modelos que se pueden crear, y una forma de considerar el UML como un lenguaje ejecutable. Starr (*Executable UML: How to Build Class Models*, Prentice-Hall, 2001) ofrece una guía útil y sugerencias detalladas para crear clases de diseño y análisis efectivos.

En Internet se dispone de una gran variedad de fuentes de información sobre el modelado del análisis. En el sitio SEPA se puede encontrar una lista actualizada de referencias de la red que son notables para el modelado del análisis:
http://www.mhhe.com/pressman.

CAPÍTULO

9

INGENIERÍA DEL DISEÑO

La ingeniería del diseño abarca un conjunto de principios, conceptos y prácticas que conducen al desarrollo de un sistema o producto de alta calidad Los principios del diseño (explicados en el capítulo 5) establecen una filosofía primordial que guían al diseñador en el trabajo que desempeña. Es necesario comprender los conceptos del diseño antes de que se apliquen las mecánicas de la práctica del diseño, y la práctica del diseño mismo conduce a la creación de varias representaciones del software, el cual sirve como guía para la actividad de construcción que sigue

La *ingeniería del diseño* no es una frase común dentro del contexto de la ingeniería del software Sin embargo, debería serlo El diseño es una actividad primordial de la ingeniería. A principios de la década de 1990, Mitch Kapor, el creador de Lotus 1-2-3, presentó un "manifiesto sobre el diseño de software" en *Dr Dobbs Journal* Ahí afirma:

> ¿Qué es el diseño? Es el lugar en donde una persona se puede parar con un pie en dos mundos —el mundo de la tecnología y el de la gente y los propósitos humanos— e intenta unirlos...
>
> El crítico de arquitectura romana Vitruvius aportó la noción de que las construcciones bien diseñadas eran aquellas que mostraban firmeza, comodidad y placer. Lo mismo debe decirse del buen software *Firmeza*: el programa no debe tener ningún error que inhiba su función *Comodidad*: un programa debe cumplir con los propósitos para los que fue creado *Placer* la experiencia de usar el programa debe ser agradable. Aquí se presentan los principios de una teoría de diseño para software

UN VISTAZO RÁPIDO

¿Qué es? El diseño es lo que casi cualquier ingeniero quiere hacer Es el sitio donde manda la creatividad, donde los requisitos del cliente, las necesidades de negocio y las consideraciones técnicas se unen en la formulación de un producto o sistema. El diseño crea una representación o modelo del software, pero a diferencia del modelo de análisis (que se enfoca en la descripción de los datos, las funciones y el comportamiento requeridos), el modelo de diseño proporciona detalles acerca de las estructuras de datos, las arquitecturas, las interfaces y los componentes del software que son necesarios para implementar el sistema.

¿Quién lo hace? Los ingenieros de software encabezan cada una de las tareas de diseño.

¿Por qué es importante? El diseño permite al ingeniero de software modelar el sistema o producto que se va a construir. Este modelo puede evaluarse en relación con su calidad y mejorarse antes de generar código, de realizar pruebas y de que los usuarios finales se vean involucrados a gran escala. El diseño es el sitio en el que se establece la calidad del software.

¿Cuáles son los pasos? El diseño presenta el software de diferentes formas. Primero, debe representarse la arquitectura del sistema o producto. Después, se modelan las interfaces que conectan el software con los usuarios finales,

con otros sistemas y dispositivos y con los propios componentes que lo constituyen. Por último, se diseñan los componentes del software que se utilizarán en la construcción del el sistema. Cada una de estas visiones representa una acción de diseño diferente, pero todas deben ajustarse a un conjunto de conceptos básicos del diseño que determinan todo el trabajo de diseño.

¿Cuál es el producto obtenido? Un modelo que abarca representaciones arquitectónicas, de interfaz, en el nivel de componentes y de despliegue.

¿Cómo puedo estar seguro de que lo he hecho correctamente? El modelo de diseño lo evalúa el equipo de software en un esfuerzo encaminado a determinar si éste contiene errores, inconsistencias u omisiones; si existen mejores alternativas; y si el modelo puede implementarse dentro de las restricciones, el itinerario y el costo que han sido establecidos.

La meta de la ingeniería del diseño es producir un modelo de representación que muestre firmeza, comodidad y placer. Para lograrlo, un diseñador debe practicar la diversificación y después la convergencia. Belady [BEL81] establece que "la diversificación es la adquisición de un repertorio de alternativas, la materia prima del diseño: componentes, soluciones de componentes y conocimiento, todo contenido en catálogos, libros de texto y en la mente". Una vez que se ha integrado este conjunto de información, el diseñador debe elegir y tomar elementos del repertorio que cumplan los requisitos definidos por la ingeniería de requisitos (capítulo 7) y el modelo de análisis (capítulo 8). Cuando esto ocurre, se consideran y se rechazan las alternativas, y el ingeniero de diseño converge en "una configuración particular de componentes y, por lo tanto, en la creación del producto final" [BEL81].

La diversificación y la convergencia demandan intuición y juicio. Estas cualidades están basadas en la experiencia de construir entidades similares, un conjunto de principios que guían cómo evoluciona el modelo, un conjunto de criterios que permiten juzgar la calidad, y un proceso de iteración que conduce a una representación del diseño final.

La ingeniería del diseño para el software de computadora está en un cambio continuo, en la medida en que evolucionan mejores métodos, mejores análisis y una comprensión más amplia. Aun en la actualidad, la mayoría de las metodologías de diseño de software carecen de profundidad, flexibilidad y naturaleza cuantitativa que por lo general se asocian con disciplinas de diseño de ingeniería más clásicas. Sin embargo, existen métodos para el diseño de software, se dispone de criterios para la calidad del diseño, y es posible aplicar notación de diseño. En este capítulo se explorarán los conceptos y principios fundamentales aplicables a todo el diseño de software, los elementos del modelo del diseño y el impacto de los patrones sobre el proceso de diseño. En los capítulos 10, 11 y 12 se examina una variedad de métodos de diseño de software mientras se aplican al diseño arquitectónico, de interfaz y en el nivel de componentes.

9.1 DISEÑO DENTRO DEL CONTEXTO DE LA INGENIERÍA DEL SOFTWARE

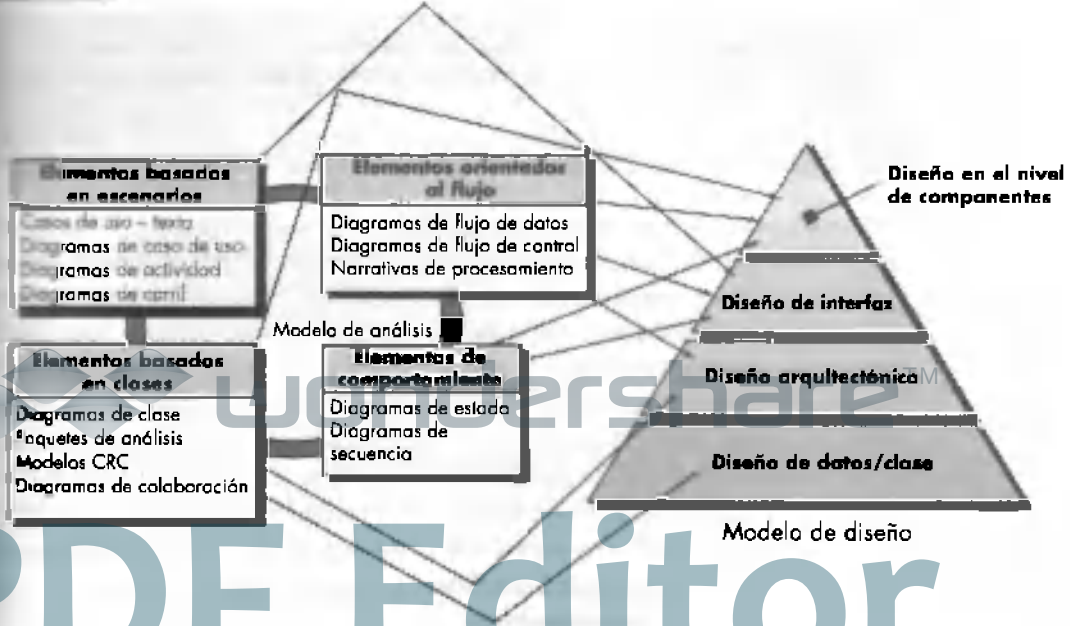
El diseño del software se encuentra en el núcleo técnico de la respectiva ingeniería y se aplica de manera independiente al modelo de software que se utilice. Una vez que se analizan y especifican los requisitos, el diseño del software es la última acción de la ingeniería correspondiente dentro de la actividad del modelado, la cual establece una plataforma para la construcción (generación de código y pruebas).

> "El milagro más común de la ingeniería de software es la transición del análisis al diseño y del diseño al código."
> **Richard Due**

Cada uno de los elementos del modelo de análisis (capítulo 8) proporciona la información necesaria para crear los cuatro modelos de diseño que se requieren para una especificación completa de diseño. En la figura 9.1 se ilustra el flujo de información durante el diseño del software. Los requisitos del software que muestran los elementos basados en escenarios, basados en clases, orientados al flujo y de comportamiento alimentan la tarea de diseño. Mediante la notación de diseño y de los métodos de diseño que se exponen en capítulos posteriores, la tarea de diseño produce un diseño de datos-clase, un diseño arquitectónico, un diseño de interfaz y un diseño de componentes.

El diseño de datos-clase transforma los modelos de análisis y clases (capítulo 8) en las clases de diseño y las estructuras de datos que se requieren para implemen-

FIGURA 9.1 Transformación del modelo de análisis en un modelo de diseño.



tar el software. Las clases y relaciones que definen las tarjetas índice CRC y el contenido detallado de datos que muestran los atributos de clase y otras notaciones proporcionan la base para la actividad de diseño de datos. Una parte del diseño de clases puede ocurrir en conjunto con el diseño de la arquitectura del software. El diseño de clase más detallado se realiza a medida que se diseña cada componente del software.

El diseño arquitectónico define la relación entre los elementos estructurales más importantes del software, los estilos arquitectónicos y patrones de diseño que pueden usarse para satisfacer los requisitos definidos por el sistema, y las restricciones que afectan la manera en que se pueden implementar los patrones arquitectónicos [SHA96]. La representación del diseño arquitectónico —el marco de trabajo de un sistema basado en computadora— puede derivarse de la especificación del sistema, del modelo de análisis y de la interacción de los subsistemas definidos dentro del modelo de análisis.

El diseño de la interfaz describe la forma en que el software se comunica con los sistemas que interactúan con él y con los humanos que los utilizan. Una interfaz implica un flujo de información (por ejemplo, datos o control) y un tipo de comportamiento específico. Por lo tanto, los escenarios de uso y los modelos de comportamiento proporcionan mucha de la información que se requiere en el diseño de la interfaz.

El diseño al nivel de componentes transforma los elementos estructurales de la arquitectura del software en una descripción procedimental de los componentes de éste. La información obtenida de los modelos basados en clases, los modelos de flujo y los modelos de comportamiento sirven como base para el diseño de componentes.

> "Existen dos formas de construir un diseño de software. Una forma es hacerlo tan simple que obviamente no hay deficiencias, y la otra es hacerlo tan complicado que no existen deficiencias obvias. El primer método es mucho más difícil."
>
> **C.A.R. Hoare**

Durante el diseño se toman decisiones que al final incidirán en el éxito de la construcción del software, así como en, con el mismo grado de importancia, la facilidad con que el software puede mantenerse. Pero, ¿por qué es tan importante el diseño?

La importancia del diseño del software puede describirse con una sola palabra *calidad*. El diseño es la etapa en la que se fomentará la calidad en la ingeniería del software. El diseño proporciona las representaciones del software susceptibles de evaluar respecto de la calidad. El diseño es la única forma en que, de manera exacta, un requisito del cliente se puede convertir en un sistema o producto de software terminado. El diseño del software sirve como fundamento para todas las actividades subsecuentes de la ingeniería del software y del soporte de éste. Sin diseño se corre el riesgo de construir un sistema inestable, el cual fallará cuando se realicen cambios pequeños; que será difícil de probar; cuya calidad no podrá evaluarse sino hasta

etapas tardías del proceso del software, cuando queda poco tiempo y ya se ha gastado mucho dinero en él.

9.2 PROCESO Y CALIDAD DEL DISEÑO

El diseño del software es un proceso iterativo mediante el cual los requisitos se traducen en un "plano" para construir el software. Al inicio, el plano representa una visión holística del software. Es decir, el diseño se representa en un grado alto de abstracción, el cual puede rastrearse de manera directa hasta conseguir el objetivo específico del sistema y requisitos más detallados de comportamiento, funcionales y de datos. A medida en que ocurren las iteraciones del diseño, un refinamiento subsiguiente conduce a representaciones del diseño a grados mucho más bajos de abstracción. Estos grados aún se pueden rastrear hasta los requisitos, pero la conexión es más sutil.

A través del proceso del diseño, la calidad en evolución de éste se evalúa con una serie de revisiones técnicas formales o con revisiones de diseño explicadas en el capítulo 26. McGlaughlin [MCG91] sugiere tres características que sirven como guía en la evaluación de un buen diseño:

- El diseño debe implementar todos los requisitos explícitos contenidos en el modelo de análisis, y debe ajustarse a todos los requisitos implícitos que desea el cliente
- El diseño debe ser una guía legible y comprensible para quienes generan código y quienes realizan pruebas y, en consecuencia, dan soporte al software
- El diseño debe proporcionar una imagen completa del software —dando dirección a los dominios de datos, funcionales y de comportamiento— desde una perspectiva de implementación.

Cada una de estas características es en realidad una meta del proceso de diseño. Pero, ¿cómo se alcanza cada una de ellas?

> "Escribir una brillante pieza de código que funcione es una cosa; diseñar algo que pueda soportar a largo plazo un negocio es otra muy diferente."
>
> **C. Ferguson**

Directrices de calidad. Con el fin de evaluar la calidad de una representación de diseño se deben establecer los criterios técnicos para un buen diseño. En secciones posteriores de este capítulo se expondrán los conceptos de diseño que también sirven como criterios de calidad del software. Por ahora se presentan las siguientes directrices:

1. Un diseño debe presentar una estructura arquitectónica que *a)* se haya creado mediante patrones de diseño reconocibles, *b)* la integren componentes que

¿Cuáles son las características de un buen diseño?

exhiban buenas características de diseño (éstas se explican más adelante en este capítulo), y *c)* pueda implementarse de manera evolutiva,[1] para que de esta forma facilite la implementación y las pruebas.

2. Un diseño debe ser modular, esto es, el software deberá dividirse de manera lógica en elementos o subsistemas.
3. Un diseño debe contener distintas representaciones de los datos, la arquitectura, las interfaces y los componentes.
4. Un diseño debe conducir a estructuras de datos que sean apropiadas para las clases que habrán de implementarse y que procedan de patrones de datos reconocibles.
5. Un diseño debe conducir a componentes que presenten características funcionales independientes.
6. Un diseño debe conducir a interfases que reduzcan la complejidad de las conexiones entre los componentes y el ambiente externo.
7. Un diseño debe obtenerse por medio de un método repetible que se base en la información obtenida durante el análisis de requisitos del software
8. Un diseño debe representarse por medio de una notación que comunique de manera eficaz su significado.

Estas directrices de diseño no se logran por casualidad. El proceso de diseño del software fomenta el buen diseño aplicando principios fundamentales de diseño, una metodología sistemática y una revisión cuidadosa.

INFORMACIÓN

Evaluación de la calidad del diseño: la revisión técnica formal

El diseño es importante porque permite que un equipo de software evalúe la calidad[2] del software antes de implementarlo; es decir, en un momento en el que los errores, omisiones o inconsistencias son fáciles de corregir y no resultan caros. Pero ¿cómo se evalúa la calidad durante el diseño? El software no se puede comprobar porque no existe un software ejecutable al cual aplicarle pruebas. ¿Qué debe hacerse?

Durante el diseño, la calidad se evalúa al realizar una serie de revisiones técnicas formales (RTF). Las RTF se tratan con detalle en el capítulo 26,[3] pero resulta valioso hacer un resumen de la técnica en este punto. Una RTF es una reunión que dirigen miembros del equipo de software. Por lo general participan dos, tres o cuatro personas, dependiendo del ámbito de la información de diseño que se revisará. Cada persona desempeña un papel: el *líder de revisión* planea la reunión, establece la agenda y después realiza la reunión; el *relator* toma notas para que nada se olvide; el *productor* es la persona cuyo producto de trabajo (por ejemplo, el diseño de un componente del software) se revisa. Antes de la reunión, cada persona en el equipo de revisión recibe una copia del producto de trabajo del

1 Para sistemas más pequeños algunas veces el diseño puede desarrollarse en forma lineal

2 Los factores de calidad tratados en el capítulo 15 pueden ayudar al equipo de revisión mientras evalúa la calidad

3 En este punto se podría considerar la revisión de la sección 26.4. Las RTF son una parte crítica del proceso de diseño y un mecanismo importante para lograr la calidad del diseño

se le pide que lo lea en busca de errores, o ambigüedades. Cuando la reunión comienza, es detectar todos los problemas del producto para que éstos puedan corregirse antes de que implementación. La RTF tiene una duración típica de entre 90 minutos y dos horas. Al concluir la RTF, el equipo de revisión determina si se requieren acciones posteriores por parte del productor antes de que el producto de trabajo del diseño pueda aprobarse como parte del modelo de diseño final.

> "La calidad no es algo que se ponga encima de los sujetos y objetos como adorno en un árbol de navidad."
> **Robert Pirsig**

Atributos de calidad. Hewlett-Packard [GRA87] desarrolló un conjunto de atributos de calidad; entre ellos están la funcionalidad, la facilidad de uso, la confiabilidad, el desempeño y la soportabilidad. Estos atributos de calidad representan un objetivo para todo el diseño de software:

de
a
el
se vio o
embargo,
que
de
son
tratarse

- La *funcionalidad* se estima al evaluar el conjunto de características y capacidades del programa, la generalidad de las funciones que se entregan y la seguridad del sistema en su totalidad.
- La *facilidad de uso* se valora al considerar los factores humanos (capítulo 12), la estética, consistencia y documentación generales.
- La *confiabilidad* se evalúa al medir la frecuencia y severidad de las fallas, la precisión de los resultados de salida, la media del momento de fallas (MMF), la habilidad para recuperarse de las fallas y la previsibilidad del programa
- El *desempeño* se mide con la velocidad de procesamiento, tiempo de respuesta, consumo de recursos, rendimiento y eficacia
- La *soportabilidad* combina la habilidad de extender el programa (extensibilidad), la adaptabilidad y la serviciabilidad —estos tres atributos representan un concepto más común, *facilidad de mantenimiento*— además, resistencia a pruebas, compatibilidad, configurabilidad (habilidad para organizar y controlar elementos de la configuración de software) (capítulo 27), la facilidad con que puede instalarse el sistema, y la facilidad con que se pueden localizar los problemas.

No todos los atributos de la calidad del software tienen el mismo peso cuando se desarrolla el diseño del software. Tal vez una aplicación tenga un especial interés en la seguridad. Es posible que otra demande desempeño con un enfoque particular en la velocidad de procesamiento. Una tercera puede centrarse en la confiabilidad. Sin importar el peso, es importante puntualizar que estos atributos de calidad deben considerarse al comienzo del diseño, *no* después de que el diseño esté completo y haya comenzado la construcción.

CONJUNTO DE TAREAS

Conjunto de tareas genéricas para el diseño

1. Examinar el modelo del dominio de la información y diseñar las estructuras de datos apropiadas para los objetos de datos y sus atributos
2. Por medio del modelo de análisis, seleccionar un estilo arquitectónico (patrón) que sea apropiado para el software.
3. Dividir el modelo de análisis en subsistemas de diseño y ubicar estos subsistemas dentro de la arquitectura.
   Asegurarse de que cada subsistema es cohesivo en su funcionamiento.
   Diseñar las interfaces del subsistema.
   Ubicar clases o funciones de análisis para cada subsistema.
4. Crear un conjunto de clases de diseño o componentes.
   Traducir cada descripción de las clases de análisis en una clase de diseño
   Verificar cada clase de diseño contra los criterios de diseño; considerar los aspectos de la herencia
   Definir métodos y mensajes asociados con cada clase de diseño.
   Evaluar y seleccionar patrones de diseño para una clase de diseño o un subsistema.
   Revisar las clases de diseño y modificarlos según se requiera.
5. Diseñar cualquier interfaz requerido con sistemas o dispositivos externos
6. Diseñar la interfaz del usuario.
   Revisar los resultados del análisis de tareas.
   Especificar la secuencia de acción con base en escenarios del usuario.
   Crear un modelo de comportamiento de la interfaz
   Definir los objetos de la interfaz y mecanismos de control.
   Revisar el diseño de la interfaz y modificarlo según se requiera
7. Conducir el diseño al nivel de componentes.
   Especificar todos los algoritmos a un grado de abstracción relativamente bajo.
   Refinar la interfaz de cada componente.
   Definir estructuras de datos al nivel de componente
   Revisar cada componente y corregir todos los errores descubiertos.
8. Desarrollar un modelo de despliegue.

9.3 CONCEPTOS DEL DISEÑO

A través de la historia de la ingeniería del software ha evolucionado un conjunto de conceptos fundamentales de diseño de software. Aunque el grado de interés en cada concepto ha variado con los años, han pasado la prueba del tiempo. Cada uno ofrece al ingeniero de software un fundamento sobre el cual pueden aplicarse métodos de diseño más elaborados.

M. A. Jackson [JAC75] dijo una vez: "El comienzo de la sabiduría para [un ingeniero de software] es reconocer la diferencia entre hacer que un programe funcione y conseguir que lo haga del modo correcto". Los conceptos fundamentales del diseño de software ofrecen el marco de trabajo necesario para hacer las cosas "del modo correcto".

9.3.1 Abstracción

Cuando se considera una solución modular a cualquier problema se pueden exponer muchos *grados de abstracción*. En un alto grado de abstracción una solución se establece en términos generales con el lenguaje del entorno del problema. En los grados de menor abstracción se proporciona una descripción más detallada de la solución.

> "La abstracción es una de las formas fundamentales en las que el humano se enfrenta a la complejidad."
>
> **Grady Booch**

CONSEJO

...para ...abstracciones ...y de ...son útiles ...en ...Si pueden ...dominio ...de ...será aún

En la medida en que cambian los diferentes grados de abstracción se trabaja para crear abstracciones procedimentales y de datos. Una *abstracción procedimental* se refiere a una secuencia de instrucciones que tiene una función específica y limitada. El nombre de abstracción procedimental implica estas funciones, pero se omiten detalles específicos. Un ejemplo de abstracción procedimental sería la palabra *abrir* para una puerta. *Abrir* implica una larga secuencia de pasos procedimentales (por ejemplo, caminar a la puerta, alcanzar la manija, darle vuelta a la manija y empujar la puerta, hacerse a un lado para abrir paso a la puerta que se abre, etc.).[4]

Una *abstracción de datos* es una colección nombrada de datos que describe un objeto de datos. En el contexto de abstracción procedimental, *abrir* se puede definir como una abstracción de datos llamada **puerta**. Como cualquier objeto de datos, la abstracción de datos para **puerta** abarcaría una serie de atributos que la describan (por ejemplo, **puerta, tipo, dirección de apertura, mecanismo de apertura, peso, dimensiones**). Se puede decir que la abstracción procedimental *abrir* emplearía la información contenida en los atributos de la abstracción de datos **puerta**.

9.3.2 Arquitectura

La *arquitectura del software* alude a "la estructura general del software y las formas en que la estructura proporciona una integridad conceptual para un sistema" [SHA95a]. En su forma más simple, la arquitectura es la estructura u organización de los componentes del programa (módulos), la manera en que éstos componentes interactúan, y la estructura de datos que utilizan los componentes. En un sentido más amplio, sin embargo, los componentes pueden generalizarse para representar elementos importantes del sistema y sus interacciones

> "Una arquitectura de software es el producto del trabajo de desarrollo que ofrece el mayor rendimiento de la inversión con respecto a la calidad, el tiempo y el costo."
>
> **Len Bass et al.**

Referencia Web

...puede

Una de las metas del diseño de software es derivar una representación arquitectónica de un sistema. Esta representación sirve como el marco de trabajo a partir del cual se conducen actividades de diseño más detalladas. Un conjunto de patrones

4 Sin embargo, debe notarse que un conjunto de operaciones puede reemplazarse con otro, siempre que la función implicada por la abstracción de procedimiento sea la misma. Por lo tanto, los pasos requeridos para implementar abrir podrían cambiar en forma sustancial si la puerta fuera automática y estuviera unida a un sensor.

CONSEJO

No debe dejarse que la arquitectura suceda por sí sola. Si eso pasa, el resto del tiempo de proyecto se invertirá en tratar de obligarla a ajustarse al diseño. Se recomienda diseñar la arquitectura de manera explícita.

arquitectónicos permite que un ingeniero de software reutilice conceptos en el nivel de diseño.

El diseño arquitectónico puede representarse al usar uno o más de muchos modelos diferentes [GAR95]. Los *modelos estructurales* representan la arquitectura como una colección organizada de componentes del programa. Los *modelos del marco trabajo* incrementan el grado de abstracción del diseño al intentar identificar marcos de trabajo repetibles del diseño arquitectónico que se encuentran en tipos de aplicaciones similares. Los *modelos dinámicos* abordan los aspectos conductuales de la arquitectura del programa, al indicar cómo puede cambiar la configuración de la estructura o el sistema, como función de los eventos externos. Los *modelos del proceso* se centran en el diseño del proceso técnico o de negocios que el sistema debe contener. Por último, los *modelos funcionales* pueden utilizarse para representar la jerarquía funcional de un sistema. El diseño arquitectónico se expone en el capítulo 10.

9.3.3 Patrones

Brad Appleton define un *patrón de diseño* de la siguiente manera: "Un patrón es una semilla de conocimiento, la cual tiene un nombre y transporta la esencia de una solución probada a un problema recurrente dentro de cierto contexto en medio de intereses en competencia" [APP98]. Dicho de otro modo, un patrón de diseño describe una estructura de diseño que resuelve un problema de diseño particular dentro de un contexto específico y en medio de "fuerzas" que pueden tener un impacto en la manera en que se aplica y utiliza el patrón.

> "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y después describe la esencia de la solución a dicho problema, de tal forma que puedas usar esta solución un millón de veces más, sin nunca hacerlo dos veces de la misma forma."
>
> **Christopher Alexander**

La finalidad de cada patrón de diseño es proporcionar una descripción que le permita al diseñador determinar 1) si el patrón es aplicable al trabajo actual, 2) si el patrón se puede reutilizar (por ende, ahorrar tiempo del diseño), y 3) si el patrón puede servir como guía para desarrollar un patrón similar, pero diferente en cuanto a la funcionalidad o estructura. Los patrones de diseño se exponen con mayor detalle en la sección 9.5.

9.3.4 Modularidad

Los patrones de arquitectura y diseño de software materializan la *modularidad*, es decir, el software se divide en componentes con nombres independientes y que es posible abordar en forma individual. Estos componentes llamados *módulos* se integran para satisfacer los requisitos del problema.

Se ha establecido que la "modularidad es el atributo particular del software que permite que un programa sea manejable de manera intelectual" [MYE78]. El software monolítico (es decir, un programa grande compuesto por un módulo sencillo) -

puede entenderlo con facilidad un ingeniero de software. El número de rutas de control, la amplitud de las referencias, el número de variables y la complejidad general imposibilitaría comprenderlo. Este punto se ilustra con el siguiente argumento, basado en observaciones de solución de problemas humanos.

Considérense dos problemas, p_1 y p_2. Si la complejidad observada para p_1 es mayor que la de p_2 se deduce que el esfuerzo requerido para resolver p_1 es mayor que el esfuerzo necesario para resolver p_2. Como un caso general, este resultado es obvio en el sentido intuitivo; la resolución de un problema difícil toma más tiempo.

También se deduce que la complejidad observada de dos problemas, cuando están combinados, con frecuencia es mayor que la suma de las complejidades observadas cuando cada una de ellas se toma por separado: esto conduce a una estrategia de "divide y vencerás" (es más fácil resolver un problema complejo cuando éste se divide en piezas más manejables). Esto tiene implicaciones importantes con respecto a la modularidad y al software. De hecho, es un argumento para la modularidad.

Es posible concluir que si el software se subdivide en forma indefinida, el esfuerzo requerido para desarrollarlo se reducirá en forma sensible. Por desgracia, hay otras fuerzas que entran en juego, lo que ocasiona que esta conclusión sea (tristemente) inválida. En relación con la figura 9.2, el esfuerzo (costo) para desarrollar un solo módulo de software decrece conforme se incrementa el número de módulos. Si se tiene el mismo conjunto de requisitos, más módulos significan un tamaño individual menor. Sin embargo, a medida que crece el número de módulos, el esfuerzo (costo) asociado con la integración de los módulos también crece. Estas características conducen también a la curva total del costo o el esfuerzo que se muestra en la figura. Existe un número M de módulos que resultaría en un costo de desarrollo mínimo, aunque hasta el momento no se tiene la sofisticación necesaria para predecir M con seguridad.

Las curvas que se muestran en la figura 9.2 proporcionan una guía útil cuando se considera la modularidad. Ésta debe aplicarse, pero se debe tener cuidado de per-

Figura 9.2 Modularidad y costo del software

manecer en la vecindad de *M*. Debe evitarse la modularidad excesiva o insuficiente pero ¿cómo puede conocerse la vecindad de *M*? ¿Qué tan modular debe hacerse el software? Las respuestas a estas preguntas requieren comprender otros conceptos de diseño que se considerarán después, en este mismo capítulo.

Un diseño y el programa resultante se modularizan de manera que el desarrollo se pueda planear con mayor facilidad; se puedan definir y entregar incrementos del software; los cambios puedan ajustarse con mayor facilidad; las pruebas y la eliminación de errores se pueda conducir con más eficiencia, y el mantenimiento se pueda conducir sin efectos laterales de consideración.

9.3.5 Ocultación de información

El concepto de modularidad conduce a todos los diseñadores de software a formularse una pregunta fundamental: ¿cómo puede descomponerse una solución de software para obtener el mejor conjunto de módulos? El principio de *ocultación de información* [PAR72] sugiere que los módulos "se caracterizan por las decisiones de diseño que (cada uno) oculta a los otros". En otras palabras, los módulos deben especificarse y diseñarse de manera que la información (procedimiento y datos) que esta dentro del módulo sea inaccesible para otros módulos que no necesiten esa información.

PUNTO CLAVE

La finalidad de la ocultación de información es reservar los detalles de las estructuras de datos y de los procesamientos de procedimiento detrás de una interfaz del módulo. El conocimiento de los detalles no debe estar al alcance de los usuarios del módulo.

La ocultación implica que se puede conseguir una modularidad efectiva al definir un conjunto de módulos independientes que se comuniquen entre sí y que intercambien sólo la información necesaria para lograr la función del software. La abstracción ayuda a definir las entidades de procedimiento (o información) que conforman el software. La ocultación define y fortalece las restricciones de acceso para los detalles de procedimiento dentro de un módulo y para cualquier estructura de datos local que utilice el módulo [ROS75].

El uso de la ocultación de información, como un criterio de diseño para sistemas modulares, proporciona los mayores beneficios cuando se requieren modificaciones durante la realización de las pruebas y, después, en el curso de mantenimiento del software. Como la mayoría de los datos y procedimientos está oculta de las otras partes del software, existe una probabilidad menor de introducir errores inadvertidos al realizar las modificaciones y propagarlos a otros lugares dentro del software.

9.3.6 Independencia funcional

El concepto de *independencia funcional* es la suma directa de la modularidad y de los conceptos de abstracción y ocultación de información. En referencias obligadas sobre el diseño de software, Wirth [WIR71] y Parnas [PAR72] aluden a las técnicas de refinamiento que mejoran la independencia de los módulos. Trabajos posteriores de Stevens, Myers y Constantine [STE74] consolidaron el concepto.

La independencia funcional se consigue al desarrollar módulos con una función "determinante" y una "aversión" a la interacción excesiva con otros módulos. Dicho de otra manera, se desea diseñar el software de tal manera que cada módulo aborde una subfunción específica de los requisitos y tenga una sola interfaz cuando se

observe desde otras partes de la estructura del programa. Es justo preguntarse por qué es importante la independencia.

El software con una modularidad efectiva, es decir, con módulos independientes, es más fácil de desarrollar porque la función se puede fraccionar y las interfaces se simplifican (considérense las ramificaciones cuando el desarrollo se realiza en equipo). Los módulos independientes son más fáciles de mantener (y probar) porque se limitan los efectos secundarios que originan las modificaciones al diseño o al código, se reduce la propagación de errores, y es posible emplear módulos reutilizables. En resumen, la independencia funcional es una clave para el buen diseño, y el diseño es la clave para lograr la calidad del software.

La independencia se evalúa aplicando dos criterios cualitativos: cohesión y acoplamiento. La *cohesión* es una medida de la fuerza funcional relativa de un módulo. El *acoplamiento* es una medida de la interdependencia relativa entre los módulos.

La cohesión es una extensión natural del concepto de ocultación de información descrito en la sección 9.3.5. Un módulo cohesivo realiza una sola tarea, para lo que requiere muy poca interacción con otros componentes en otras partes del programa. Dicho de manera sencilla, un módulo cohesivo debe (idealmente) hacer sólo una cosa.

El acoplamiento es una medida de la interconexión entre los módulos de una estructura de software. El acoplamiento depende de la complejidad de la interfaz entre los módulos, el punto donde se realiza una entrada o referencia a un módulo, y los datos que pasan a través de la interfaz. En el diseño de software se intenta conseguir el acoplamiento más bajo posible. Una conectividad sencilla entre los módulos da como resultado un software más fácil de entender y menos propenso a experimentar el "efecto ola" [STE74], el cual se presenta cuando surgen problemas en un lugar y después se propagan a través del sistema.

9.3.7 Refinamiento

El *refinamiento* paso a paso es una estrategia de diseño descendente que propuso inicialmente Niklaus Wirth [WIR71]. El desarrollo de un programa se realiza al refinar de manera sucesiva los niveles de detalle procedimentales. Una jerarquía se desarrolla al descomponer el enunciado macroscópico de una función (una abstracción procedimental) paso a paso hasta alcanzar las oraciones del lenguaje de programación.

En realidad, el refinamiento es un proceso de *elaboración*. Se inicia con el enunciado de una función (o una descripción de datos) que se define con un alto grado de abstracción. Esto es, el enunciado describe los datos o la función de manera conceptual, pero no proporciona información acerca de los trabajos internos de la función o de la estructura interna de los datos. El refinamiento hace que el diseñador trabaje sobre el enunciado original y que proporcione más y más detalles conforme se realiza cada refinamiento sucesivo (elaboración).

La abstracción y el refinamiento son conceptos complementarios. La abstracción le permite a un diseñador especificar procedimientos y datos sin considerar detalles

de grado menor. El refinamiento ayuda al diseñador a revelar los detalles de grado menor mientras se realiza el diseño. Ambos conceptos auxilian al diseñador en la creación de un modelo de diseño completo a medida que evoluciona la actividad de diseño.

> "No fallé. Sólo encontré 10 000 formas fallidas de hacer las cosas."
>
> Thomas Edison

Referencia Web

En www.refactoring.com se pueden encontrar excelentes recursos para la refabricación.

9.3.8 Refabricación

Una actividad importante de diseño que sugieren muchos métodos ágiles (capítulo 4) es la *refabricación*, técnica de reorganización que simplifica el diseño (o cualquier parte de un componente sin cambiar su función o comportamiento Fowler [FOW99] define la refabricación de la siguiente manera: "La refabricación es el proceso de cambiar un sistema de software de tal forma que no se altere el comportamiento externo del código [diseño] y aún así se mejore su estructura interna."

Cuando un software se refabrica el diseño existente se examina en busca de redundancias, elementos de diseño inútiles, algoritmos innecesarios o insuficientes, estructuras de datos inapropiadas o construidas de manera incorrecta, o cualquier otra falla de diseño que se pueda corregir para lograr un mejor diseño. Por ejemplo, una primera iteración de diseño podría producir un componente que muestra poca cohesión (realiza tres funciones que tienen muy pocas relaciones entre sí). El diseñador puede decidir que el componente debe refabricarse en tres componentes distintos, cada uno de ellos con una elevada cohesión. El resultado será un software más fácil de integrar, probar y mantener.

HOGARSEGURO

Conceptos de diseño

El escenario: Cubículo de Vinod, comienza el modelado del diseño

Los actores: Vinod, Jamie y Ed, miembros del equipo de ingeniería del software de *HogarSeguro*. También, Shakira, un nuevo miembro del equipo

La conversación:

[Los cuatro miembros del equipo acaban de regresar de un seminario matutino, titulado "Aplicación de conceptos básicos de diseño", que ofreció un profesor local de ciencia computacional.]

Vinod: ¿Obtuvieron algo del seminario?

Ed: Ya sabía la mayor parte de las cosas, pero creo que no es una mala idea escucharlas de nuevo

Jamie: Cuando realizaba mis estudios profesionales en SC nunca entendí realmente por qué la ocultación de información era tan importante como dicen.

Vinod: Porque... es una técnica para reducir la propagación del error en un programa. En realidad, creo que la independencia funcional también se logra lo mismo

Shakira: Yo no tengo un título en SC, entonces muchas de las cosas que mencionó el instructor son nuevas para mí. Y yo pueda generar un buen código y rápido. No veo por qué este asunto es tan importante.

Jamie: He visto tu trabajo, Shak, y ¿sabes qué? Tú haces muchas de estas cosas en forma natural .. por eso es que tus diseños y tu código funcionan.

Shakira (sonriendo): Bueno, yo siempre trato de código, mantenerlo enfocado en una cosa, simples y restringidos las interfases, reutilizar código siempre que puedo. . . ese tipo de cosas.

Ed: Modularidad, independencia funcional, ocultación, ¿ves?

Jamie: Todavía recuerdo el primer curso de ...ación que tomé .. nos enseñaron a refinar el ... iterativamente

Vinod: Tú sabes que lo mismo puede aplicarse al ...

Ed: El único concepto del que no había escuchado antes ... "refabricación"

Shakira: Se utiliza en Programación Extrema, creo que eso dijo.

Ed: Sí, no es por completo diferente al refinamiento, sólo lo haces después de que el diseño o el código han sido completados. Si me preguntan a mí, es un tipo de paso hacia la optimización del software.

Jamie: Regresemos al diseño de *HogarSeguro*. Creo que deberíamos poner estos conceptos en nuestra lista de revisión mientras desarrollamos el modelo de diseño para *HogarSeguro*.

Vinod: Estoy de acuerdo. Pero igual de importante, comprometámonos todos a pensar en ellos conforme desarrollamos el diseño.

9.3.9 Clases de diseño

En el capítulo 8 se mencionó que el modelo de análisis define un conjunto completo de clases de análisis. Cada una de estas clases describe algún elemento del dominio del problema, con enfoque en los aspectos del problema visibles para el usuario o el cliente. El grado de abstracción de una clase de análisis es relativamente alto.

Conforme evoluciona el modelo de diseño, el equipo de software debe definir un conjunto de *clases de diseño* que 1) refine las clases de análisis al proporcionar detalles del diseño que permitirán la implementación de las clases, y 2) produzca un conjunto nuevo de clases de diseño que implementen una infraestructura de software para soportar la solución del negocio. Se sugieren cinco diferentes tipos de clases de diseño, cada uno representa una capa distinta de la arquitectura de diseño [AMB01]:

¿Qué tipos de clases el diseñador?

- Las *clases de interfaz con el usuario* definen todas las abstracciones necesarias para la interacción humano-computadora (IHC). En muchos casos, la IHC ocurre dentro del contexto de una *metáfora* (por ejemplo, un libro de verificación, un formato de orden, una máquina de fax) y las clases de diseño para la interfaz pueden ser representaciones visuales de los elementos de la metáfora.
- Las *clases del dominio de negocios* a menudo son refinamientos de las clases de análisis definidas antes. Las clases identifican los atributos y servicios (métodos) necesarios para implementar algún elemento del dominio de negocios.
- Las *clases de proceso* implementan abstracciones del negocio en un nivel más bajo, las cuales se requieren para manejar por completo las clases del dominio de negocios.
- Las *clases persistentes* representan almacenamientos de datos (por ejemplo, una base de datos) que persistirán más allá de la ejecución del software.

- Las *clases de sistema* implementan las funciones de gestión y control del software que permiten que el sistema opere y se comunique dentro de su entorno de computación y con el mundo exterior

A medida que evoluciona el modelo de diseño, el equipo de software debe desarrollar un conjunto completo de atributos y operaciones para cada clase de diseño. El grado de abstracción se reduce conforme cada clase de análisis se transforma en una representación del diseño. Esto es, las clases de análisis representan objetos (y servicios asociados que se aplican a éstos) usando la jerga del dominio de negocios. Las clases de diseño presentan un mayor detalle técnico, pues son una guía para la implementación.

Arlow y Neustadt [ARL02] sugieren revisar cada clase de diseño para asegurar que la misma esté "bien formada". Ellos definen cuatro características de una clase de diseño bien formada:

¿Qué es una clase de diseño "bien formada"?

Completa y suficiente. Una clase de diseño debe ser la encapsulación completa de todos los atributos y métodos que se pueden esperar, en forma razonable (con base en una interpretación reconocible del nombre de la clase), que existan para la clase. Por ejemplo, la clase **escena** definida para el software de edición de video está completa sólo si contiene todos los atributos y métodos que pueden asociarse de manera razonable con la creación de una escena de video. La suficiencia asegura que la clase de diseño contenga sólo aquellos métodos que sean suficientes para lograr el objetivo, ni más ni menos.

Primitivismo. Los métodos asociados con una clase de diseño deben enfocarse en el cumplimiento de un servicio para la clase. Una vez que el servicio ha sido implementado con un método, la clase no debe proporcionar otra forma de complementar la misma cosa. Por ejemplo, la clase **videoClip** del software de edición de video podrían tener atributos **punto-inicial** y **punto-final** para indicar los puntos de inicio y fin del clip (nótese que el video bruto cargado en el sistema puede ser más largo que el clip que se usa). Los métodos *establecerPuntoInicial( )* y *establecerPuntoFinal( )* proporcionan los únicos medios para configurar los puntos de inicio y fin del clip.

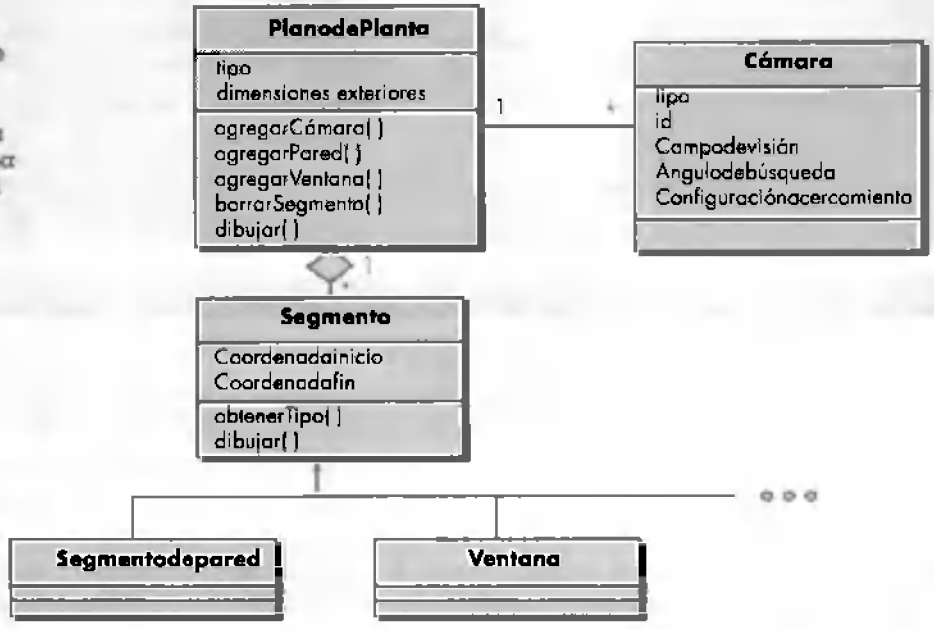
Cohesión alta. Una clase de diseño cohesiva tiene un conjunto de responsabilidades pequeño y enfocado, y aplica atributos y métodos de manera sencilla para implementar dichas responsabilidades. Por ejemplo, la clase **VideoClip** del software para la edición de video podría contener un conjunto de métodos para editar el videoclip. Mientras cada método se enfoque sólo en atributos asociados con el clip se mantendrá la cohesión.

Acoplamiento bajo. Dentro del modelo de diseño es necesario que las clases de diseño colaboren con alguna otra. Sin embargo, la colaboración se debe mantener en un mínimo aceptable. Si un modelo de diseño tiene un acoplamiento alto (todas las clases de diseño colaboran con todas las otras clases de diseño), el sistema es difícil de implementar, probar y mantener a través del tiempo. En general, las c

de diseño dentro de un subsistema deben tener sólo un conocimiento limitado de las clases en otros subsistemas. Esta restricción, llamada la *Ley de Deméter* [LIE03], sugiere que un método sólo debe enviar mensajes a métodos de clases vecinas.[5]

FIGURA 9.3

Clase de diseño para el plano de planta y agregación compuesta para la clase (véase la exposición en el recuadro).



HOGARSEGURO

Refinación de una clase de análisis en una clase de diseño

El escenario: Cubículo de Ed, continúa el modelado del diseño

Los actores: Vinod y Ed, miembros del equipo de ingeniería del software de *HogarSeguro*.

La conversación:

Ed trabaja en la clase **PlanodePlanta** [véase el debate en el recuadro de la sección 8.7.4 y la figura 8.14] y la ha refinado para el modelo de diseño.

Ed: Entonces ¿recuerdas la clase **PlanodePlanta,** no? Se utiliza como una parte de las funciones de vigilancia y administración del hogar.

Vinod (afirmando con la cabeza): Sí, me parece recordar que la usamos como parte de nuestros análisis de CRC para la administración del hogar.

Ed: Lo hicimos. De cualquier manera, la estoy refinando para el diseño. Quiero mostrar cómo implementaremos

5 Una forma menos formal de enunciar la Ley de Deméter es: "Cada unidad debe hablar sólo con sus amigos; no con extraños."

realmente la clase **PlanodePlanta.** Mi idea es implementarla como un conjunto de listas ligadas [una estructura de datos específica]. Entonces ... tuve que refinar la clase de análisis **Planodeplanta** (figura 8.14) y, en realidad, hasta simplificarla.

Vinod: La clase de análisis sólo mostraba cosas en el dominio del problema, bueno, realmente sobre la pantalla de la computadora, que fueran visibles para el usuario final ¿no?

Ed: Sí, pero para la clase de diseño **PlanodePlanta** tengo que agregar algunas cosas que son implementación específica. Necesitaba mostrar que **PlanodePlanta** es una agregación de segmentos –y por ende la clase **Segmento**– y que la clase **Segmento** está compuesta de listas para segmentos de pared, ventanas, puertas y cosas así. La clase **Cámara** colabora con **PlanodePlanta** y, obviamente, puede haber muchos cámaras en el plano de planta.

Vinod: Bueno, veamos una fotografía de esta nueva clase de diseño **PlanodePlanta.**

[Ed le muestra el esquema de la figura 9.3.]

Vinod: De acuerdo, veo lo que estás tratando de hacer. Esto te permite modificar fácilmente el plano de planta porque puedes agregar nuevos elementos o borrar otros de la lista –la agregación– sin ningún problema.

Ed (afirmando con la cabeza): Sí, yo creo que funcionará.

Vinod: Yo también.

9.4 EL MODELO DE DISEÑO

El *modelo de diseño* puede verse en dos dimensiones diferentes, como se ilustra en la figura 9.4. La dimensión del *proceso* indica la evolución del modelo de diseño conforme se ejecutan las tareas de diseño como una parte del proceso del software. La dimensión de *abstracción* representa el grado de detalle a medida que cada elemento del modelo de análisis se transforma en un equivalente del diseño y después se refina de una manera iterativa. En la figura, la línea punteada indica la frontera entre los modelos de análisis y diseño. En algunos casos se distingue con claridad entre los modelos de análisis y diseño; en otros, el modelo de análisis se combina con el diseño y la distinción resulta menos obvia.

PUNTO CLAVE

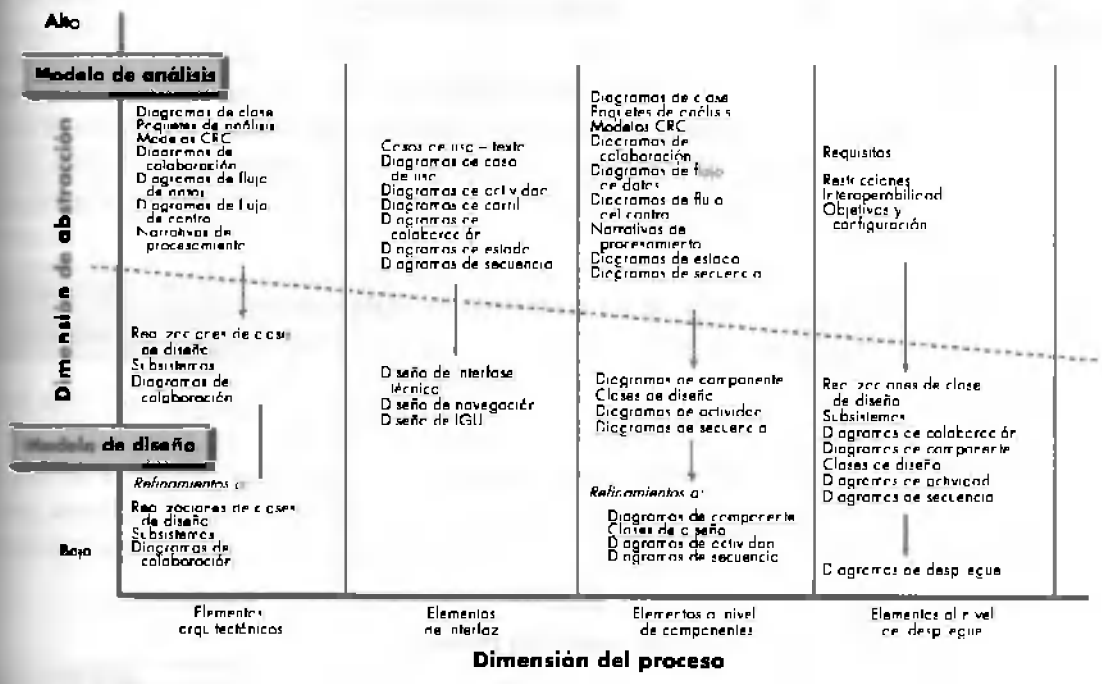
El modelo de diseño tiene cuatro elementos importantes: datos, arquitectura, componentes e interfaz.

Los elementos del modelo del diseño utilizan muchos de los diagramas en UML aplicados en el modelo de análisis. La diferencia es que estos diagramas están refinados y elaborados como parte del diseño; se proporciona un mayor detalle para la implementación específica y se resaltan la estructura y el estilo arquitectónicos, los componentes que residen dentro de la arquitectura y las interfaces entre los componentes y con el mundo exterior.

> "Las preguntas acerca de si el diseño es necesario o evitable están bastante fuera de lugar: el diseño es inevitable. La alternativa al buen diseño es el mal diseño y no su inexistencia."
>
> **Douglas Martin**

Sin embargo, es importante mencionar que los elementos del modelo anotados a lo largo del eje horizontal no siempre se desarrollan de una manera secuencial. En la mayoría de los casos, el diseño arquitectónico preliminar establece la plataforma y lo siguen el diseño de interfaz y el diseño al nivel de componentes, los cuales a menudo se realizan en paralelo. El modelo de despliegue con frecuencia se retrasa hasta que el diseño ha sido desarrollado por completo.

FIGURA 9.4 Dimensiones del modelo de diseño



9.4.1 Elementos del diseño de datos

Al igual que otras actividades de la ingeniería del software, el *diseño de datos* (algunas veces llamado *arquitectura de datos*) crea un modelo de datos o información que se representa con un alto grado de abstracción (la visión de los datos del cliente/usuario). Después, este modelo de datos se refina en representaciones que de manera progresiva tienen una implementación específica y que pueden procesarse mediante el sistema basado en computadora. En muchas aplicaciones de software la arquitectura de los datos tendrá una profunda influencia sobre la arquitectura del software que los debe procesar

PUNTO CLAVE

...el diseño
...enfoca en
...bases de
...de los
..., el
...datos
...las
...de datos
...queren para
...r los
...datos

La estructura de los datos siempre ha sido una parte importante del diseño del software. Al nivel de los componentes del sistema, las estructuras del diseño de datos y los algoritmos con que se manipulan son esenciales para la creación de aplicaciones de alta calidad. Al nivel de aplicación, la traducción de un modelo de datos (obtenido como una base de la ingeniería de requisitos) a una base de datos es esencial para alcanzar los objetivos de negocio de un sistema. Al nivel de negocios, la colección de información almacenada en bases de datos dispersas y reorganizadas en una "conjunción de datos" permite la explotación de datos o el descubrimiento de un conocimiento que puede tener un impacto sobre el éxito del mismo negocio. En

cada caso, el diseño de datos juega un papel importante. El diseño de datos se explica con mayor detalle en el capítulo 10.

PUNTO CLAVE

El modelo arquitectónico se deriva del dominio de aplicación, del modelo de análisis y de los estilos y patrones disponibles.

9.4.2 Elementos del diseño arquitectónico

El *diseño arquitectónico* para el software es el equivalente al plano de planta de una casa. Este plano muestra la configuración general de las habitaciones, su tamaño, forma y las relaciones entre ellas, y las puertas y ventanas que permiten el movimiento hacia y desde los cuartos. El plano de planta proporciona una visión global de la casa. Los elementos del diseño arquitectónico dan una visión general del software.

> "Puedes usar un borrador en la tabla de diseño o un martillo en el sitio de construcción."
>
> **Frank Lloyd Wright**

El modelo arquitectónico [SHA96] se obtiene a partir de tres fuentes: 1) la información acerca del dominio de aplicación para el software que se va a construir; 2) los elementos del modelo de análisis específico, tales como diagramas de flujo de datos o clases de análisis, sus relaciones y colaboraciones para el problema que se tiene a la mano, y 3) la disponibilidad de patrones (sección 9.5) y estilos arquitectónicos (capítulo 10).

9.4.3 Elementos de diseño de interfaz

El *diseño de interfaz* para software es el equivalente a un conjunto de dibujos detallados (y especificaciones) para puertas, ventanas y utilidades externas de una casa. Estos dibujos representan el tamaño y forma de las puertas y ventanas, la manera en que operan, la manera en que las conexiones de las utilidades (como agua, energía eléctrica, gas, teléfono) llegan a la casa y se distribuyen entre las habitaciones representadas en el plano de planta. Estos dibujos indican dónde está localizado el timbre de la puerta, si hay un intercomunicador que anuncie la presencia de un visitante y cómo está instalado el sistema de seguridad. En esencia, los dibujos (y especificaciones) detallados para las puertas, ventanas y utilidades externas indican cómo fluyen las cosas y la información desde y hacia la casa y dentro de las habitaciones que son parte del plano de planta. Los elementos del diseño de interfaz para software muestran cómo fluye la información hacia o fuera del sistema y cómo éste está comunicado entre los componentes definidos como parte de la arquitectura.

> "El público está más familiarizado con el diseño malo que con el bueno. En efecto, está condicionado a preferir el mal diseño porque es con lo que vive. Lo nuevo es amenazante, lo viejo es seguro."
>
> **Paul Rand**

Existen tres elementos importantes del diseño de interfaz: 1) la interfaz con el usuario (IU), 2) interfaces externas a otros sistemas, artefactos, redes u otros productores o consumidores de información; y 3) interfaces internas entre varios com-

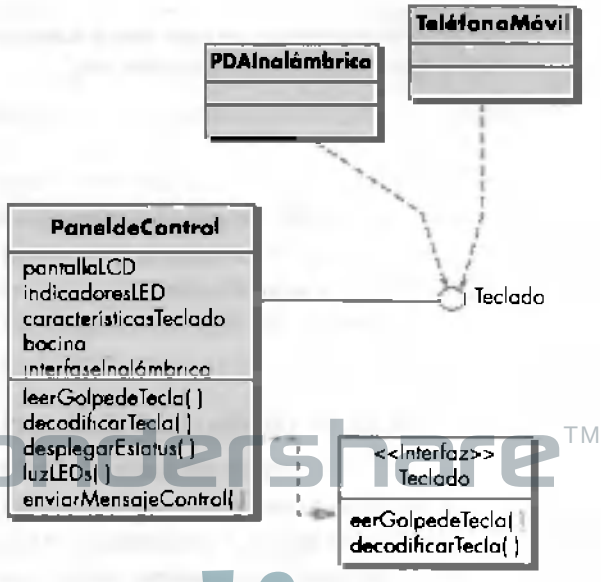
ponentes de diseño. Estos elementos de diseño de interfaz permiten al software comunicarse de manera externa y permiten la comunicación y colaboración interna entre los componentes que pueblan la arquitectura del software.

El diseño de la IU es una acción primordial de la ingeniería de software, y se considera con detalle en el capítulo 12. El diseño de una IU incorpora elementos estéticos (por ejemplo, distribución, color, gráficas, mecanismos de interacción), elementos ergonómicos (por ejemplo, información y ubicación de la distribución, metáforas, navegación de la IU), y elementos técnicos (como patrones de la IU, componentes reutilizables). En general, la IU es un subsistema único dentro de la arquitectura de aplicación general.

El diseño de las interfases externas requiere información definitiva acerca de la entidad hacia donde se manda o recibe la información. En todos los casos, esta información debe recopilarse durante la ingeniería de requisitos (capítulo 7) y verificarse una vez que comience el diseño de la interfaz.[6] El diseño de interfases externas debe incorporar revisión de errores y (cuando sea necesario) características apropiadas de seguridad.

El diseño de las interfaces internas está cercanamente alineado con el diseño al nivel de los componentes (capítulo 11). Las realizaciones del diseño de clases de análisis representan todas las operaciones y esquemas de mensajes requeridos para permitir la comunicación y colaboración entre las operaciones de varias clases. Cada

FIGURA 9.5



6 No resulta poco común que las características de la interfaz cambien con el tiempo. Por lo tanto, un diseñador debe asegurar que la especificación para la interfaz se mantenga actualizada.

Referencia Web
En www.useit.com puede encontrarse información muy valiosa sobre el diseño de la IU

mensaje debe ser diseñado para ajustarse a la transferencia de información de requisitos y los requerimientos funcionales específicos de la operación que ha sido solicitada.

En algunos casos, una interfaz se modela de una manera muy parecida a una clase. El UML define una *interfaz* de la siguiente manera [OMG01]: "Una interfaz es un determinante de las operaciones [públicas] visibles de manera externa de una clase, componente u otro clasificador (incluidos los subsistemas) sin especificación de estructura interna". Dicho de un modo más simple, una interfaz es un conjunto de operaciones que describe parte del comportamiento de una clase y proporciona acceso a esas operaciones.

Por ejemplo, la función de seguridad de *HogarSeguro* emplea un panel de control que permite al propietario de la casa controlar ciertos aspectos de la función de seguridad. En una versión avanzada del sistema, las funciones del panel de control pueden implementarse vía PDA inalámbrico o teléfono móvil.

La clase **PaneldeControl** (figura 9.5) proporciona el comportamiento asociado con un teclado y, por lo tanto, debe implementar operaciones de *leerTeclaPresionada* y *decodificarTecla( )*. Si estas operaciones se suministrarán a otra clase (en este caso a **PDAInalámbrico** y **TeléfonoMóvil**), resulta inútil definir una interfaz como la que se muestra en la figura. La interfaz llamada **Teclado** se muestra como un estereotipo de <<interfaz>> o como un círculo pequeño y etiquetado que se conecta a la clase con una línea. La interfaz se define sin atributos y con el conjunto de operaciones necesarias para lograr el comportamiento de un teclado.

> "Un error común que cometen las personas cuando tratan de diseñar algo completamente a prueba de tontos es subestimar la ingenuidad de los que son completamente tontos."
>
> **Douglas Adams**

La línea punteada con un triángulo abierto en su extremo (figura 9.5) indica que la clase **PaneldeControl** proporciona operaciones de **Teclado** como parte de su comportamiento. En UML esto se caracteriza como una *realización*. Esto es, parte del comportamiento de **PaneldeControl** se implementará al realizar las operaciones de **Teclado**. Estas operaciones se proporcionarán a otras clases que entren a la interfaz.

9.4.4 Elementos de diseño al nivel de componentes

El diseño al nivel de componentes para el software equivale a un conjunto de dibujos detallados (y especificaciones) para cada cuarto en una casa. Estos dibujos muestran el alambrado y la instalación sanitaria dentro de cada cuarto, la ubicación de los receptáculos eléctricos e interruptores, llaves, lavabos, tinas, desagües y armarios. También describen los pisos que se usarán, los moldes que se aplicarán, y cualquier otro detalle asociado con el cuarto. El diseño al nivel de componentes para software describe por completo el detalle interno de cada componente del software. Para

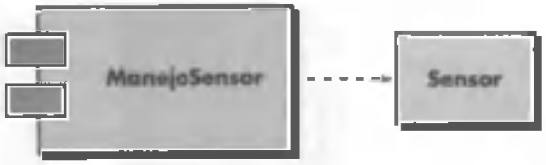


FIGURA 9.6 Diagrama de componente en UML para ManejoSensor.

lograrlo el diseño al nivel de componentes define estructuras de datos para todos los objetos de datos locales, así como detalle algorítmico para todo el procesamiento que ocurre dentro de un componente y una interfaz que permite el acceso a todas las operaciones de los componentes (comportamientos).

> "Los detalles no son los detalles. Ellos hacen el diseño."
>
> **Charles Eames**

Dentro del contexto de la ingeniería del software orientada a objetos, un componente se representa a manera de diagrama en UML como se muestra en la figura 9.6. En esta figura se representa un componente llamado **ManejoSensor** (parte de la función de seguridad de *HogarSeguro*). El componente está conectado a una clase llamada **Sensor**, la cual está asignada a éste mediante una flecha punteada. El componente **ManejoSensor** realiza todas las funciones asociadas con los sensores de *HogarSeguro*, entre las que se encuentran su monitoreo y configuración. En el capítulo 11 se presenta una explicación más a fondo acerca de los diagramas de componente.

Los detalles de diseño de un componente se pueden modelar a muchos grados distintos de abstracción. En la representación del procesamiento lógico se puede utilizar un diagrama de actividad. El flujo detallado de procedimiento para un componente puede representarse, ya sea mediante un pseudocódigo (una representación del tipo de lenguaje de programación que se describe en el capítulo 11), o de algún formato diagramático (por ejemplo, un diagrama de actividad o un diagrama de flujo).

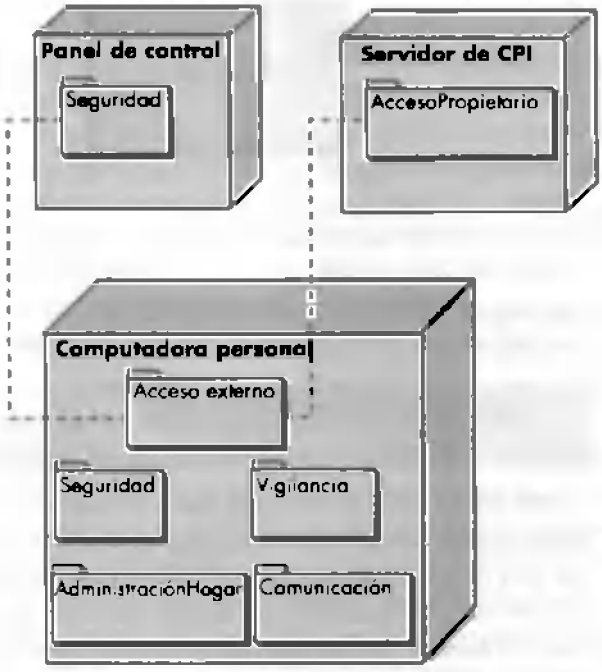
9.4.5 Elementos de diseño al nivel del despliegue

Los elementos de diseño al nivel del despliegue indican cómo se ubicarán la funcionalidad y los subsistemas dentro del entorno computacional físico que soportará al software. Por ejemplo, los elementos del producto *HogarSeguro* están configurados para operar dentro de tres entornos de computación primarios: una PC doméstica, el panel de control de *HogarSeguro* y un servidor ubicado en CPI Corp. (lo que proporciona acceso al sistema a través de Internet).

Durante el diseño se desarrolla un diagrama de despliegue en UML y después se refina, como se muestra en la figura 9.7. En ésta se muestran tres ambientes computacionales (en realidad, debería haber más, si se incluyen sensores, cámaras y otros). Se indican los subsistemas (funcionalidad) que se alojan dentro de cada ele-

FIGURA 9.7

Diagrama de despliegue en UML para *HogarSeguro*.



PUNTO CLAVE

Los diagramas de despliegue comienzan en un formato descriptor, donde el entorno de despliegue se describe en términos generales. Después se utiliza un formato de instancia y se describen de manera explícita los detalles de configuración.

mento de cómputo. Por ejemplo, la computadora personal aloja subsistemas que implementan seguridad, vigilancia, administración del hogar y características de comunicación. Además, se ha diseñado un subsistema de acceso externo para controlar todos los intentos de acceso al sistema *HogarSeguro* desde una fuente externa. Cada subsistema sería elaborado para indicar los componentes que implementa.

El diagrama mostrado en la figura 9.7 está en *forma de descriptor*. Esto significa que el diagrama de despliegue muestra el entorno computacional, pero no indica de manera explícita detalles de configuración. Por ejemplo, no se identifica la "computadora personal". Podría ser una "Wintel" PC o una Macintosh, una estación de trabajo Sun o una Linux-box. Estos detalles se proporcionan cuando el diagrama de despliegue se revisa en *forma de instancia* durante etapas posteriores del diseño o cuando comienza la construcción. Se identifica cada instancia del despliegue (una configuración de hardware con un nombre específico)

"Deja todo y retírate, toma una pequeña relajación. Cuando regresas al trabajo, tu juicio será más seguro. Toma algo de distancia porque entonces el trabajo parece más pequeño, una mayor parte del mismo puede ser capturado en una mirada y la falta de armonía y proporción se observa con más facilidad."

Leonardo DaVinci

9.5 Diseño de software basado en patrones

Los mejores diseñadores en cualquier campo de trabajo tienen la misteriosa habilidad de vislumbrar patrones que caracterizan un problema y los patrones correspondientes que pueden combinarse para crear una solución. A través del proceso de diseño, un ingeniero de software debe buscar toda oportunidad para reutilizar patrones de diseño existentes (cuando cumplen las necesidades de un diseño) en vez de crear nuevos.

9.5.1 Descripción de un patrón de diseño

Las disciplinas maduras de la ingeniería utilizan miles de patrones de diseño. Por ejemplo, un ingeniero mecánico utiliza un eje de dos pasos como un patrón de diseño clave. Los atributos (diámetros del eje, dimensiones del orificio de las llaves, etcétera) y las operaciones (por ejemplo, la rotación del giro y la conexión del giro) son inherentes al patrón. Un ingeniero eléctrico utiliza un circuito integrado (un patrón de diseño en extremo complejo) para resolver un elemento específico de un problema nuevo. Los patrones de diseño pueden describirse empleando la plantilla [MAI03] que se muestra en el recuadro "Plantilla del patrón de diseño".

INFORMACIÓN

Plantilla del patrón de diseño

Nombre del patrón: describe la esencia del patrón en un nombre corto, pero expresivo.

...n: describe el patrón y lo que hace.

...conocido-como: lista los sinónimos para el ...

...on: proporciona un ejemplo del problema.

...lidad: anota situaciones específicas de diseño en ... es aplicable el patrón

...: describe las clases que se requieren para ...ntar el patrón.

Participantes: describe las responsabilidades de las clases que se requieren para implementar el patrón

Colaboraciones: describe cómo colaboran los participantes para llevar a cabo sus responsabilidades

Consecuencias: describe las "fuerzas de diseño" que afectan al patrón y los intercambios potenciales que deben considerarse cuando se implementa el patrón.

Patrones relacionados: patrones de diseño relacionados mediante referencias cruzados.

...CLAVE ...de diseño ...y aquellos ...e la solución ...n la forma ...se puede ...el diseño.

Una descripción del patrón de diseño puede considerar también un conjunto de fuerzas de diseño. Las *fuerzas de diseño* describen requisitos no funcionales (por ejemplo, facilidad de mantenimiento, portabilidad) asociados con el software en el que se aplicará el patrón. Además, las fuerzas definen las limitaciones que restringen la manera en que se implementará el diseño. En esencia, las fuerzas de diseño describen el ambiente y las condiciones que deben existir para que el patrón del diseño sea aplicable. Las características del patrón (clases, responsabilidades y colaboraciones) indican los atributos ajustables del diseño para permitir que el patrón se ajuste a una variedad de problemas [GAM95]. Estos atributos representan características del diseño que pueden buscarse (por ejemplo, a través de una base de datos)

para que sea factible encontrar un patrón apropiado. Por último, la guía asociada con el uso de un patrón de diseño indica las ramificaciones de las decisiones de diseño.

> "Los patrones están a medio hornear, lo que significa que siempre debes terminarlos y adaptarlos a tu propio entorno."
>
> **Martin Fowler**

Los nombres de los patrones de diseño deben elegirse con cuidado. Uno de los problemas técnicos clave en la reutilización de software es la falta de habilidad para encontrar patrones reutilizables existentes, a pesar de que existen cientos o miles de patrones. La búsqueda del patrón "correcto" tiene un apoyo inmenso si se cuenta con un nombre significativo del patrón

9.5.2 Utilización de patrones en el diseño

Los patrones de diseño pueden usarse durante el diseño del software. Una vez que se ha desarrollado el modelo de análisis (capítulo 8), el diseñador puede examinar una representación detallada del problema que debe resolver y las restricciones que impone el problema. La descripción del problema se examina en varios grados de abstracción para determinar si es flexible para uno o más de los siguientes tipos de patrones de diseño.

¿Qué tipos de patrones de diseño están disponibles para el ingeniero de software?

Patrones arquitectónicos. Estos patrones definen la estructura general del software, indican las relaciones entre los subsistemas y los componentes del software y definen las reglas para especificar las relaciones entre los elementos (clases, paquetes, componentes, subsistemas) de la arquitectura.

Patrones de diseño. Estos patrones se aplican a un elemento específico del diseño como un agregado de componentes para resolver algún problema de diseño. Las relaciones entre los componentes o los mecanismos para efectuar la comunicación de componente a componente.

Idiomas. A veces llamados *patrones de código*, estos patrones específicos de lenguaje por lo general implementan un elemento algorítmico o un componente de protocolo de interfaz específico o un mecanismo de comunicación entre los componentes.

Cada uno de los tipos de patrones difiere en el grado de abstracción con el que está representado y con el grado en el que proporciona una guía directa para la actividad de construcción (en este caso, codificación) del proceso de software

9.5.3 Marcos de trabajo

En algunos casos es necesario proporcionar una infraestructura esquelética específica de implementación, llamada *marco de trabajo*, para el trabajo de diseño. Esto es, el diseñador puede seleccionar una "*miniarquitectura reutilizable* que ofrezca el

PUNTO CLAVE

...de trabajo es ...o de código ... con clases ... que han ... para ... el problema en ...

portamiento y la estructura genérica para una familia de abstracciones de software, junto con un contexto... que especifique su colaboración y uso dentro de un dominio dado" [APP98].

Un marco de trabajo no es un patrón arquitectónico, sino un esqueleto con una colección de "puntos de conexión" (también llamados ganchos y ranuras) que le permiten adaptarse a un dominio de un problema específico. Los puntos de conexión permiten al diseñador integrar clases o funcionalidad específicas del problema dentro del esqueleto. En un contexto orientado al objeto, un marco de trabajo es una colección de clases que cooperan.

En esencia, el diseñador de un marco de trabajo argumentará que una miniarquitectura reutilizable se puede aplicar a todo el software que se desarrollará dentro de un dominio limitado de aplicación. Para que sean más efectivos, los marcos de trabajo se aplican sin cambios. Se pueden agregar elementos de diseño adicionales, pero sólo a través de los puntos de conexión que permiten que el diseñador desarrolle el esqueleto del marco de trabajo.

9.6 Resumen

La ingeniería de diseño comienza cuando la primera iteración de la ingeniería de requisitos llega a su fin. La finalidad del diseño de software es aplicar un conjunto de principios, conceptos y prácticas que conducen al desarrollo de un sistema o producto de alta calidad. La meta del diseño es crear un modelo de software que implemente todos los requisitos del cliente de manera correcta y complazca a aquellos que lo usen. Los ingenieros de diseño deben examinar por medio de muchas alternativas de diseño y converger en la solución que mejor cumpla las necesidades de los interesados en el proyecto.

El proceso de diseño avanza de una visión general de software a una visión más estrecha que define el detalle requerido para implementar un sistema. El proceso comienza con un enfoque en la arquitectura Se definen los subsistemas; se establecen mecanismos de comunicación entre los subsistemas; se identifican los componentes; y se desarrolla una descripción detallada de cada componente. Además, se diseñan las interfases internas, externas y del usuario.

Los conceptos de diseño han evolucionado en la primera mitad del siglo de trabajo de la ingeniería del software. Estos conceptos describen atributos del software de computadora que deben estar presentes sin importar el proceso de ingeniería del software que se elija, los métodos de diseño que se apliquen, o los lenguajes de programación que utilicen

El modelo de diseño abarca cuatro elementos diferentes. En la medida en que se desarrolla cada uno de estos elementos evoluciona una visión más completa del diseño. El elemento arquitectónico utiliza información derivada del dominio de aplicación, el modelo de análisis y catálogos disponibles para patrones y estilos que deriven una representación estructural completa del software, sus sistemas y com-

ponentes. Los elementos de diseño de interfaz modelan interfaces internas y exter-
nas y la interfaz del usuario. Los elementos al nivel de componentes definen cada
uno de los módulos (componentes) que pueblan la arquitectura. Por último, los ele-
mentos de diseño al nivel de despliegue asignan la arquitectura, sus componentes y
las interfases a la configuración física que albergará el software.

El diseño basado en patrones es una técnica que reutiliza elementos de dis
que han probado ser exitosos en el pasado. Cada patrón arquitectónico, patrón de
diseño o idioma se cataloga, se documenta por completo y se considera cuidado-
mente cuando se evalúa para incluirlo en una aplicación específica. Los marcos de
trabajo, una extensión de los patrones, ofrecen un esqueleto arquitectónico para el
diseño de subsistemas completos dentro de un dominio de aplicación específica.

REFERENCIAS

[AMB01] Ambler, S., *The Object Primer*, Cambridge Univ. Press, 2a. ed., 2001.

[APP98] Appleton, B., "Patterns and Software: Essential Concepts and Terminology", puede obtenerse en http://www.enteract.com/~bradapp/docs/patterns-intro.html.

[ARL02] Arlow, J. e I. Neusdadt, *UML and the Unified Process*, Addison-Wesley, 2002.

[BEL81] Belady, L., prólogo de *Software Design: Methods and Techniques* (L. J. Peters, autor), Yourdon Press, 1981.

[FOW00] Fowler, M. *et al.*, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

[GAM95] Gamma, E. *et al.*, *Design Patterns*, Addison-Wesley, 1995.

[GAR95] Garlan, D. y M. Shaw, "An Introduction to Software Architecture", en *Advances in Software Engineering and Knowledge Engineering*, vol. I (V. Ambriola y G. Tortora, eds.) World Scientific Publishing Company, 1995.

[GRA87] Grady, R. B. y D. L. Casswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987.

[JAC75] Jackson, M. A., *Principles of Program Design*, Academic Press, 1975.

[LIE03] Lieberherr, K., "Demeter: Aspect-Oriented Programming", mayo de 2003, disponible en http://www.ccs.neu.edu/home/lieber/LoD.html.

[MAI03] Maioriello, J., "What Are Design Patterns and Do I Need Them?", developer.com, 2003, disponible en http://www.developer.com/design/article.php/1474561.

[MCG91] McGlaughlin, R., "Some Notes on Program Design", en *Software Engineering Notes*, vol. 16, núm. 4, octubre de 1991, pp. 53-54.

[MYE78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978.

[OMG01] Object Management Group, *OMG Unified Modeling Language Specification*, versión 1.4, septiembre de 2001.

[PAR72] Parnas, D. L., "On Criteria to be used in Decomposing Systems into Modules", CACM, vol. 14, núm. 1, abril de 1972, pp. 221-227.

[ROS75] Ross, D., J. Goodenough y C. Irvine, "Software Engineering: Process, Principles and Goals", en *IEEE Computer*, vol. 8, núm. 5, mayo de 1975.

[SCH02] Schmuller, J., *Teach Yourself UML*, SAMS Publishing, 2002.

[SHA96] Shaw, M. y D. Garlan, *Software Architecture*, Prentice-Hall, 1996.

[STA02] "Metaphor", en *The Stanford HCI Learning Space*, 2002, http://hci.stanford.edu/hcils/concepts/metaphor.html.

[STE74] Stevens, W., G. Myers y L. Constantine, "Structured Design", en *IBM Systems Journal*, vol. 13, núm. 2, 1974, pp. 115-139.

[WIR71] Wirth, N., "Program Development by Stepwise Refinement", en *CACM*, vol. 14, núm. 4, 1971, pp. 221-227.

PROBLEMAS Y PUNTOS A CONSIDERAR

9.1. ¿Se diseña un software cuando se "escribe" un programa? ¿Qué es lo que hace que el diseño de software sea diferente a la generación de código?

9.2. Si un diseño de software no es un programa (de hecho no lo es), ¿entonces, qué es?

9.3. ¿Cómo se evalúa la calidad de un diseño de software?

9.4. Examinar el conjunto de tareas presentadas para un diseño. ¿Cuándo se evalúa la calidad dentro del conjunto de tareas? ¿Cómo se logra esto?

9.5. Dar ejemplos de tres abstracciones de datos y abstracciones procedimentales que puedan utilizarse para manipularlas

9.6. Describir con argumentos propios la arquitectura de software.

9.7. Sugerir un patrón de diseño relacionado con una categoría de cosas cotidianas (por ejemplo, productos electrónicos, automóviles, aparatos). Documentar el patrón con ayuda de la plantilla que se presenta en la sección 9.5.

9.8. ¿Existe algún caso en el que los problemas complejos requieran de menos esfuerzo para resolverse? ¿Cómo afectaría ese caso el argumento para la modularidad?

9.9. se debe implementar un diseño modular como software monolítico? ¿Cómo se puede lograr esto? ¿El desempeño es la única justificación para la implementación del software monolítico?

9.10. Explicar la relación entre el concepto de ocultación de información como un atributo de modularidad efectiva y el concepto de independencia del módulo.

9.11. ¿Cómo se relacionan los conceptos de acoplamiento y portabilidad del software? Dar ejemplos que apoyen la explicación.

9.12. Aplicar un "enfoque de refinamiento paso a paso" para desarrollar tres grados diferentes de abstracción procedimental para uno o más de los siguientes programas: *a)* Desarrollar una máquina que expida cheques que, al dar una cantidad numérica en dólares, imprima la cantidad en palabras que por lo general se requiere en un cheque; *b)* resolver de manera iterativa la raíz de una ecuación trascendental; *c)* desarrollar una tarea simple que planee algoritmos para un sistema operativo.

9.13. Realizar una pequeña investigación sobre programación extrema y escribir un texto breve acerca de la refabricación para un proceso de desarrollo de software ágil.

9.14. Visitar un depósito de patrones de diseño (en la web) y navegue por unos minutos a través de los patrones. Elegir uno y presentarlo ante los compañeros de clase.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Donald Norman ha escrito dos libros (*The Design of Everyday Things*, Doubleday, 1990, y *The Psychology of Everyday Things*, HarperCollins, 1988) que se han convertido en clásicos en la bibliografía sobre diseño y "debe" leerlos cualquiera que diseñe cualquier cosa que usen los humanos. Adams (*Conceptual Blockbusting*, 3a. ed., Addison-Wesley, 1986) ha escrito un libro que es una lectura esencial para los diseñadores que quieran ampliar su manera de pensar. Por último, un texto clásico de Polya (*How to Solve It*, Princeton University Press, 2a ed., 1988) proporciona un proceso de resolución de problemas genérico que puede ayudar a los diseñadores de software al enfrentarse con problemas complejos.

Dentro de la misma tradición, Winograd *et al.* (*Bringing Design to Software*, Addison-Wesley, 1996) analiza los diseños de software que funcionan, los que no funcionan y por qué. Un libro fascinante editado por Wixon y Ramsey (*Field Methods Casebook for Software Design*, Wiley,

1996] sugiere métodos de investigación de campo (muy parecidos a los que utilizan los antropólogos) para entender cómo los usuarios finales hacen el trabajo que hacen, y después ofrece una guía para diseñar el software que satisfaga sus necesidades. Beyer y Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs*, Academic Press, 1997) ofrecen otra visión del diseño de software que integra al cliente-usuario en cada aspecto del proceso de diseño de software.

McConnell (*Code Complete*, Microsoft Press, 1993) presenta una excelente exposición de los aspectos prácticos de diseñar software para computadora de alta calidad. Robertson (*Simple Program Design*, 3a. ed., Kboyd y Fraser Publishing, 1999) ofrece una útil explicación introductoria del diseño de software para quienes comienzan su estudio acerca del tema. Fowler y sus colegas (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) exponen técnicas para el mejoramiento incremental de los diseños de software.

En la década pasada se han escrito muchos libros sobre diseños basados en patrones para ingenieros de software. Gamma y sus colegas [GAM95] han escrito el libro fundamental sobre el tema. Otros libros de Douglass (*Real-Time Design Patterns*, Addison-Wesley, 2002), Metsker (*Design Patterns Applied*, Wrox Press, 2002), Marinescu y Roman (*EJB Design Patterns*, Wiley, 2001) sitúan patrones de diseño en ambientes de aplicación y lenguajes específicos. Además, los libros clásicos del arquitecto Christopher Alexander (*Notes on the Synthesis of Form*, Harvard University Press, 1964, y *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977) debe leerlos el diseñador de software que pretenda comprender a fondo los patrones de diseño.

En Internet se dispone de una amplia variedad de fuentes de información sobre ingeniería de diseño. Una lista actualizada de referencias en la red mundial relevantes para el diseño de software y la ingeniería de diseño puede encontrarse en el sitio web de SEPA: **http://www.mhhe.com/pressman.**

CAPÍTULO

10

DISEÑO ARQUITECTÓNICO

CONCEPTOS CLAVE

El diseño se ha descrito como un proceso de varios pasos en el cual las representaciones de la estructura de los datos y el programa, las características de la información y el detalle procedimental se sintetizan a partir de los requisitos. Esta descripción la amplía Freeman [FRE80]:

> [D]iseño es una actividad relacionada con la toma de decisiones, a menudo de naturaleza estructural. Comparte con la programación una preocupación relacionada con abstraer la representación de la información y las secuencias del procesamiento, pero el grado de detalles es muy diferente en los extremos. El diseño construye representaciones coherentes y bien planeadas de los programas, que se concentran en las interrelaciones entre las partes al nivel más elevado y las operaciones lógicas en los niveles inferiores ..

Como se indicó en el capítulo 9, el diseño está orientado a la información Los métodos de diseño del software se derivan de la consideración de cada uno de los tres dominios del análisis del modelo Los dominios de la información, la función y el comportamiento sirven como guía para el diseño del software.

En este capítulo se presentarán los métodos requeridos para crear "representaciones coherentes y bien planeadas" de las capas de los datos y la arquitectura del modelo de diseño El objetivo es proporcionar un enfoque sistemático del diseño arquitectónico: los planos preliminares que se utilizan

UN VISTAZO RÁPIDO

¿Qué es? El diseño arquitectónico representa la estructura de datos y los componentes del programa necesarios para construir un sistema computacional. Asume el estilo arquitectónico que tomará el sistema, la estructura y las propiedades de los componentes que constituyen el sistema y las interrelaciones entre todos los componentes arquitectónicos de un sistema.

¿Quién lo hace? Aunque un ingeniero de software puede diseñar los datos y la arquitectura, a menudo el trabajo se asigna a especialistas cuando se construyen sistemas grandes y complejos. Un diseñador de base de datos o de almacén de datos crea la arquitectura de datos del sistema. El "arquitecto del sistema" selecciona un estilo arquitectónico apropiado para los requisitos derivados durante la ingeniería del sistema y el análisis de los requisitos del software

¿Por qué es importante? Nadie trataría de construir una casa sin un plano, ¿verdad? Tampoco empezaría a trazar planos bosquejando la distribución de la fontanería Necesitaría un panorama general (la propia casa) antes de preocuparse por los detalles Eso es lo que hace el diseño arquitectónico proporciona una vista general y asegura que se obtenga lo que se desea.

¿Cuáles son los pasos? El diseño arquitectónico empieza con el diseño de los datos y luego pasa a la derivación de una o más representaciones de la estructura arquitectónica del sistema. Se analizan los estilos o patrones arquitectónicos alternos para derivar la estructura que se amolda mejor a los requisitos del cliente. En

cuanto se selecciona una opción se elabora la arquitectura empleando un método de diseño apropiado.

¿Cuál es el producto obtenido? Un modelo que abarca la arquitectura de los datos y la estructura del programa se crea durante esta etapa del diseño. Además, se describen las propiedades de los componentes y sus relaciones (interacciones).

¿Cómo puedo estar seguro de que lo he hecho correctamente? En cada etapa se revisan los productos resultantes del diseño del software para verificar la claridad, la corrección, el grado en que se han completado y su consistencia con los requisitos y entre unos y otros.

10.1 ARQUITECTURA DEL SOFTWARE

En su notable libro sobre el tema, Shaw y Garlan [SHA96] analizan la arquitectura del software de la siguiente manera:

> Desde la primera vez que un programa se dividió en módulos, los sistemas de software han tenido arquitecturas, y los programadores han sido responsables de las interacciones entre los módulos y las propiedades globales del ensamblaje. Históricamente, las arquitecturas han estado implícitas (como accidentes de implementación o sistemas heredados del pasado). Los buenos desarrolladores de software han adoptado con frecuencia uno o varios patrones arquitectónicos como estrategia para la organización del sistema, pero los emplean de manera informal y no tienen medios para hacerlos explícitos en el sistema resultante

Hoy, la arquitectura del software efectiva y su representación y diseño explícitos se han vuelto temas dominantes en la ingeniería del software.

> "La arquitectura de un sistema es un marco conceptual completo que describe su forma y estructura (sus componentes y la manera en que se integran)."
>
> **Jerrold Grochow**

10.1.1 ¿Qué es la arquitectura?

Cuando se analiza la arquitectura de un edificio vienen a la mente muchos atributos diferentes. En el aspecto más simple se considera la forma general de la estructura física. Pero, en realidad, la arquitectura es mucho más, es la manera en que los diversos componentes de un edificio se integran para formar un todo cohesionado. Es la manera en que el edificio se amolda a su ambiente y se combina con otros edificios vecinos. Es el grado en el cual el edificio cumple con el propósito establecido en que satisface las necesidades de su propietario. Es la percepción estética de la estructura —el impacto visual del edificio— y la manera en que las texturas, los colores y los materiales se combinan para crear la fachada externa y el "entorno viviente" del interior. Son pequeños detalles: el diseño de la iluminación, el tipo de piso, la colocación de las cosas que cuelgan de las paredes, la lista es casi interminable. Finalmente, se trata de un arte.

¿Pero qué pasa con la *arquitectura del software*? Bass, Clement y Kazman [BAS03] definen este término elusivo de la siguiente manera:

> La arquitectura del software de un programa o sistema de cómputo es la estructura o las estructuras del sistema, que incluyen los componentes del software, las propiedades visibles externamente de esos componentes y las relaciones entre ellos

La arquitectura no es el software operativo. En cambio, es una representación que permite que un ingeniero del software: 1) analice la efectividad del diseño para cumplir con los requisitos establecidos, 2) considere opciones arquitectónicas en una etapa en que aún resulta relativamente fácil hacer cambios al diseño, y 3) reduzca los riesgos asociados con la construcción del software.

> "Cásese cuanto antes con su arquitectura y después arrepiéntase a su gusto"
>
> Barry Boehm

Esta definición destaca el papel de los "componentes del software" en cualquier representación arquitectónica. En el contexto del diseño arquitectónico, un componente de software es algo tan simple como un módulo del programa o una clase orientada a objetos, pero también se extiende para incluir bases de datos y *middleware* que permita configurar una red de clientes y servidores.

En este libro, el diseño de la arquitectura del software considera dos niveles de la pirámide del diseño (figura 9.1): el *diseño de datos* y el *diseño arquitectónico*. En el contexto del análisis anterior, el diseño de los datos permite representar el componente de datos de la arquitectura en sistemas convencionales y definiciones de clase (atributos y operaciones de encapsulamiento) de los sistemas orientados a objetos. El diseño arquitectónico se concentra en la representación de la estructura de los componentes del software, sus propiedades e interacciones.

10.1.2 ¿Por qué es importante la arquitectura?

En un libro dedicado a la arquitectura del software, Bass y sus colegas [BAS03] identifican tres razones clave por las cuales la arquitectura del software es importante:

- Las representaciones de la arquitectura del software permiten la comunicación entre todas las partes (participantes) interesadas en el desarrollo de un sistema de cómputo.
- La arquitectura destaca las decisiones iniciales relacionadas con el diseño que tendrán un impacto profundo en todo el trabajo de la ingeniería del software que le sigue y, lo que también resulta importante, en el éxito final del sistema como entidad operacional.
- La arquitectura "constituye un modelo relativamente pequeño e intelectualmente comprensible de cómo está estructurado el sistema y cómo trabajan juntos sus componentes" [BAS03].

El modelo de diseño arquitectónico y los patrones arquitectónicos que contiene son transferibles. Es decir, los estilos y patrones arquitectónicos (sección 10.3.1) se aplican al diseño de otros sistemas y representan un conjunto de abstracciones que permiten a los ingenieros de software describir la arquitectura de maneras predecibles.

10.2 Diseño de datos

La acción de *diseño de datos* traduce los objetos de datos definidos como parte del modelo de análisis (capítulo 8) en estructuras globales al nivel de componentes de software y, cuando es necesario, una arquitectura de base de datos al nivel de aplicación. En algunas situaciones debe diseñarse y construirse una base de datos específicamente para un nuevo sistema. Sin embargo, en otras, se emplean una o más bases de datos existentes.

10.2.1 Diseño de datos al nivel arquitectónico

Hoy los negocios grandes y pequeños están inundados de datos. No resulta poco común que incluso un negocio de tamaño moderado tenga docenas de bases de datos que sirven a muchas aplicaciones que abarcan cientos de gigabytes de datos. El reto consiste en extraer información útil de este entorno de datos, sobre todo cuando la información deseada tiene funcionalidad cruzada (por ejemplo, información que sólo se obtiene si los datos específicos de mercadotecnia están relacionados de manera cruzada con los datos de ingeniería del producto).

> "La calidad de los datos marca la diferencia entre un almacén y un basurero de datos."
>
> Jarrett Rosenberg

Referencia Web

En el siguiente sitio Web se obtiene información acerca de [illegible] almacén de datos: www.datawarehouse.com.

Para resolver este desafío la comunidad de empresas de la tecnología de la información (TI) ha desarrollado la técnica de *minería de datos*, también denominada *descubrimiento de conocimiento en bases de datos* (DCBD), que recorre bases de datos existentes con el fin de extraer información apropiada en el ámbito de los negocios. Sin embargo, la existencia de múltiples bases de datos, sus diferentes estructuras, el grado de detalle que contienen y muchos otros factores hacen que la minería de datos resulte difícil dentro de un entorno existente de base de datos. Una solución alterna, denominada *almacén de datos* agrega una capa adicional a la arquitectura de datos.

Un almacén de datos es un entorno de datos independiente que no está directamente integrado en las aplicaciones cotidianas, pero que abarca todos los datos utilizados en un negocio [MAT96]. En cierto sentido, un almacén de datos es una base de datos grande e independiente que tiene acceso a los datos almacenados en las bases de datos que sirven al conjunto de aplicaciones requeridas en un negocio.

Conviene más dejar el análisis detallado del diseño de estructuras de datos, bases de datos y almacenes de datos a los libros dedicados a estos temas (por ejemplo, [DATO00], [PRE98], [KIM98]). El lector interesado debe leer la sección *Otras lecturas y fuentes de información* de este capítulo como referencia adicional.

HERRAMIENTAS DE SOFTWARE

Minería y almacenamiento de datos

Objetivo: Las herramientas de la minería de datos ayudan en la identificación de relaciones entre atributos que describen un objeto de datos específico o un conjunto de objetos de datos. Las herramientas para el almacenamiento de datos ayudan en el diseño de modelos para un almacén de datos.

Mecánica: Estas herramientas tienen diversas mecánicas. En general, las herramientas de minería aceptan conjuntos grandes de datos como entrada y permiten que el usuario las consulte para tratar de comprender mejor las relaciones entre diversos elementos de datos. Las herramientas de almacenamiento empleadas en el diseño proporcionan vínculos con la entidad u otras opciones de modelado.

Herramientas representativas[1]

Minería de datos:

Business Objects, desarrollada por Business Objects, SA (www.businessobjects.com), es un conjunto de herramientas de diseño de datos que apoya "la integración, la consulta, el informe, el análisis y el análisis de datos".

SPSS, desarrollado por SPSS, Inc. (www.spss.com), proporciona una amplia variedad de funciones estadísticas que permiten el análisis de conjuntos grandes de datos.

Almacenamiento de datos:

Industry Warehouse Studio, desarrollada por Sybase (www.sybase.com), proporciona una infraestructura de almacén de datos empaquetado que "sirve como trampolín" para iniciar el diseño de un almacén de datos.

IFW Business Intelligence Suite, desarrollada por Modelware (www.modelwarepl.com), es un conjunto de modelos, herramientas de software y diseños de base de datos que "proporcionan un camino rápido al almacén de datos y al diseño y la implementación de centros de acopio de datos".

Una lista extensa de herramientas y recursos de minería y almacenamiento de datos se encontrará en el Data Warehousing Information Center (www.dwinfocenter.org).

10.2.2 Diseño de datos al nivel de componentes

El diseño de datos al nivel de componentes se concentra en la representación de estructuras de datos a las que se tiene acceso en forma directa mediante uno o más componentes de software. Wasserman [WAS80] ha propuesto un conjunto de principios que se emplea para especificar y diseñar estas estructuras de datos. En realidad, el diseño de datos empieza durante la creación del modelo de análisis. Si se recuerda que el análisis y el diseño de requisitos suelen superponerse, se considerará el siguiente conjunto de principios (adaptado de [WAS80]) para la especificación de datos:

¿Cuáles principios aplicables al diseño de datos?

1. *Los principios del análisis sistemático aplicados a la función y el comportamiento también deben aplicarse a los datos.* También es necesario desarrollar y revisar las representaciones del flujo de datos y el contenido, identificar los objetos de datos, considerar organizaciones alternas de datos y evaluar el impacto de los datos que modelan el diseño del software.
2. *Deben identificarse todas las estructuras de datos y las operaciones que se realizarán.* El diseño de una estructura de datos eficiente debe tener en cuenta las operaciones que se realizarán en la estructura de datos. Los atributos y operaciones encapsulados dentro de una clase satisfacen este principio.

1 El autor no respalda las herramientas expuestas; sólo representan una muestra de esta categoría. En casi todos los casos los nombres son marcas registradas de sus respectivos desarrolladores.

3. *Debe establecerse un mecanismo para la definición del contenido de cada objet de datos y usarlo para definir los datos y las operaciones que se les aplican.* Los diagramas de clase (capítulo 8) definen los elementos de datos (atributos) contenidos dentro de una clase y el procesamiento (operaciones) que se apl ca a esos elementos de datos.
4. *Las decisiones del diseño al nivel de datos deben posponerse hasta una de las u timas etapas del proceso de diseño.* Un proceso de refinación paso a paso es aplicable al diseño de datos. Es decir, la organización general de los datos puede definirse durante el análisis de los requisitos, refinarse durante el tra bajo de diseño de datos y especificarse de manera detallada durante el disen al nivel de componentes.
5. *La representación de una estructura de datos sólo debe conocerse para los módulos que deben usar directamente los datos que contiene tal estructura.* El concep to de ocultación de la información y el concepto relacionado del acoplamient (capítulo 9) proporcionan conocimientos importantes sobre la calidad de un diseño de software.
6. *Debe desarrollarse una biblioteca de estructuras de datos útiles y también las operaciones que pueden aplicárseles.* Esto se logra con una biblioteca de clases.
7. *Un diseño de software y un lenguaje de programación deben dar soporte a la especificación y la realización de los tipos de datos abstractos.* La implementación de una sofisticada estructura de datos llega a volverse excesivamente difícil s no existen medios para la especificación directa de la estructura en el lengua je de programación elegido para la implementación.

Estos principios forman una base para un enfoque de diseño de datos, al nivel de componentes, que se integre a las actividades de análisis y diseño.

10.3 Estilos y patrones arquitectónicos

Cuando un constructor estadounidense usa la frase "colonial con una sala al centro" (*centre hall colonial*) para describir una casa, la mayoría de quienes estén familiarizados con las casas en Estados Unidos evocará una imagen general del aspecto de la casa y de su plano general. El constructor ha usado un *estilo arquitectónico* como mecanismo descriptivo para diferenciar la casa de otros estilos (por ejemplo, marco en A, rancho elevado, Cape Cod). Pero algo más importante es que el estilo arquitectónico también es una plantilla para la construcción. Resulta necesario proporcionar mayores detalles de la casa. Se deben especificar sus dimensiones finales, agregar características personalizadas, determinar los materiales de construcción, pero el estilo ("colonial con sala al centro") es el que guía el trabajo del constructor.

"En el fondo de la mente de todo artista hay un patrón o tipo de arquitectura."

G. K. Chesterton

¿Qué es un estilo arquitectónico?

El software que se construye para sistemas de cómputo también muestra uno o muchos estilos arquitectónicos. Cada estilo describe una categoría de sistemas que abarca 1) un conjunto de componentes (por ejemplo, una base de datos, módulos computacionales) que realizan una función requerida por el sistema; 2) un conjunto de conectores que permiten la "comunicación, coordinación y cooperación" entre los componentes; 3) restricciones que definen cómo se integran los componentes para formar el sistema, y 4) modelos semánticos que permiten a un diseñador, mediante el análisis de las propiedades conocidas de las partes que lo integran (BAS03), comprender las propiedades generales de un sistema.

Información Web

Un estilo arquitectónico es una transformación impuesta al diseño de todo un sistema. El objetivo es establecer una estructura para todos los componentes del sistema. En caso de que una arquitectura existente se vaya a someter a reingeniería (capítulo 31), la imposición de un estilo arquitectónico desembocará en cambios fundamentales en la estructura del software, incluida una reasignación de la funcionalidad de los componentes [BOS00].

Un *patrón arquitectónico*, al igual que un estilo, impone una transformación en el diseño de una arquitectura. Sin embargo, un patrón difiere de un estilo en varios elementos fundamentales: 1) el alcance de un patrón es menor, ya que se concentra en un aspecto en lugar de hacerlo en toda la arquitectura; 2) un patrón impone una regla sobre la arquitectura, pues describe la manera en que el software manejará algún aspecto de su funcionalidad al nivel de la infraestructura (por ejemplo, concurrencia) [BOS00]; 3) los patrones arquitectónicos tienden a abarcar aspectos específicos del comportamiento dentro del contexto de la arquitectura (por ejemplo, cómo maneja una aplicación en tiempo real la sincronización o las interrupciones). Los patrones se usan junto con un estilo arquitectónico para determinar la forma de la estructura general de un sistema. En la siguiente sección se expondrán estilos y patrones arquitectónicos de uso común en el software.

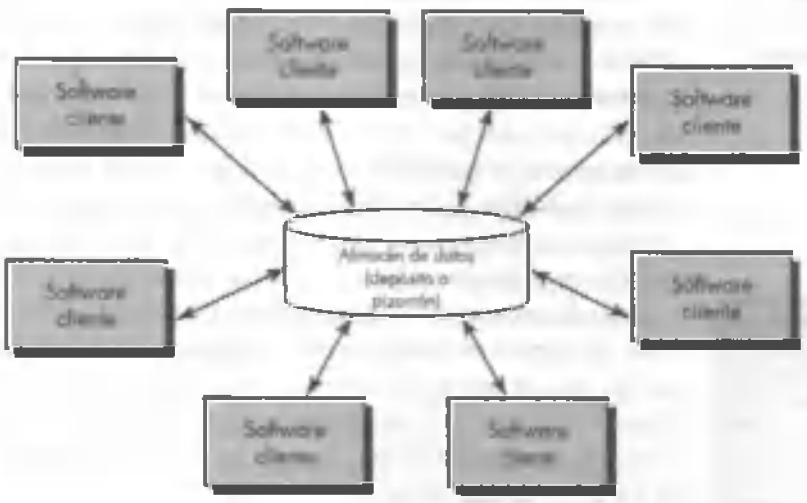
10.3.1 Una breve taxonomía de estilos arquitectónicos

Aunque se han creado millones de sistemas de cómputo en los últimos 50 años, la gran mayoría puede clasificarse (consúltense [SHA96], [BUS96], [BAS03]) en un número relativamente pequeño de estilos arquitectónicos:

Arquitectura centrada en datos. Un almacén de datos (por ejemplo, un archivo o una base de datos) se encuentra en el centro de esta arquitectura; otros componentes tienen acceso a él y cuentan con la opción de actualizar, agregar, eliminar o, por otra parte, modificar los datos de ese almacén. En la figura 10.1 se ilustra un estilo típico centrado en datos. El software cliente tiene acceso a un almacén central. En algunos casos éste es pasivo. Es decir, el software cliente accede a los datos independientemente de cualquier cambio hecho a los datos o a las acciones de otro software cliente. Una variación de este enfoque transforma el depósito en un "pizarrón" que envía notificaciones al software cliente cuando cambian los datos de interés para el cliente.

FIGURA 10.1

Arquitectura centrada en datos.

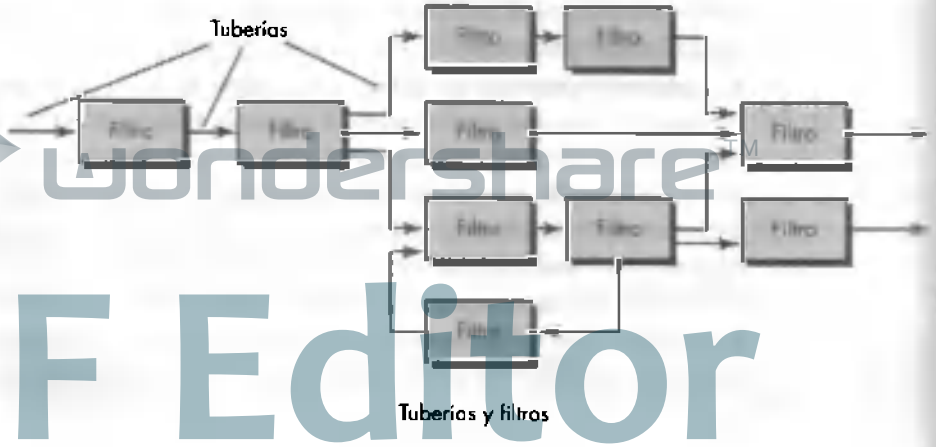


Una arquitectura centrada en datos promueve la *capacidad de integración* [BAS... Esto significa que es posible cambiar componentes existentes y agregar nuevos componentes cliente a la arquitectura sin preocuparse por otros clientes (ya que los componentes cliente operan en forma independiente). Además, es posible pasar datos entre clientes empleando el mecanismo del pizarrón (es decir, el componente pizarrón sirve para coordinar la transferencia de información entre clientes). Los componentes cliente ejecutan los procesos de manera independiente.

Arquitectura de flujo de datos. Esta arquitectura se aplica cuando los datos de entrada se habrán de transformar en datos de salida mediante una serie de componentes para el cálculo o la manipulación. Una estructura de tuberías y filtros (figura 10.2) tiene un conjunto de componentes, denominados *filtros*, conectados por *tuberías* que transmiten datos de un componente al siguiente. Cada filtro funciona sin tomar en

FIGURA 10.2

Arquitectura de flujo de datos.



cuenta si los componentes tienen flujo ascendente o descendente; está diseñado para esperar la entrada de datos con cierta forma y producir su salida (al siguiente filtro) de una forma específica. Sin embargo, no es necesario que el filtro conozca el funcionamiento de los filtros vecinos.

> "En las disciplinas de la ingeniería se usan ampliamente patrones y estilos de diseño."
>
> **Mary Shaw y David Garlan**

Si el flujo de datos degenera en una sola línea de transformaciones se denomina *procesamiento por lotes secuencial*. Esta estructura acepta un procesamiento por lotes de datos y luego aplica una serie de componentes secuenciales (filtros) para transformarlos.

Arquitectura de llamada y retorno. Este estilo arquitectónico permite que un diseñador de software (arquitecto del sistema) obtenga una estructura de programa que resulta relativamente fácil modificar y cambiar de tamaño. En esta categoría hay dos subestilos [BAS03]:

- *Arquitectura de programa principal/subprograma*. Esta estructura de programa clásica separa la función en una jerarquía de control donde un programa "principal" invoca a varios componentes de programa, que a su vez pueden invocar a otros componentes. En la figura 10.3 se ilustra una arquitectura de este tipo.
- *Arquitectura de llamada de procedimiento remoto* Los componentes de una arquitectura de programa principal/subprograma se distribuyen entre varias computadoras de una red.

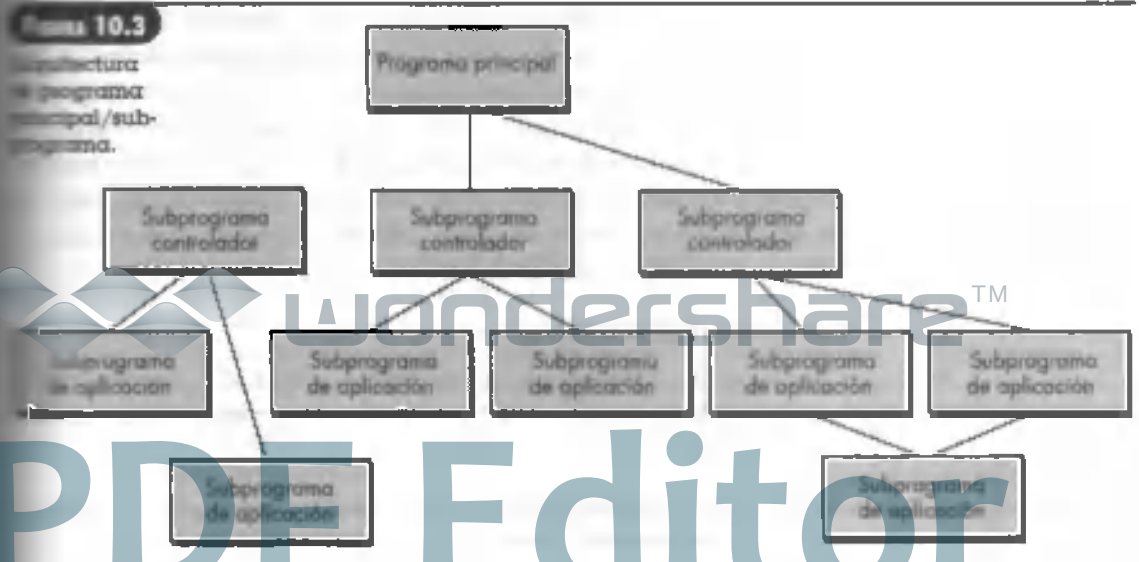
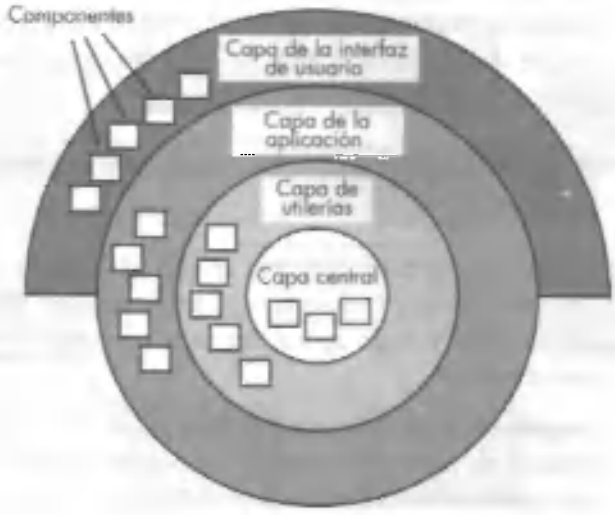


Figura 10.3 Arquitectura de programa principal/subprograma.

Figura 10.4
Arquitectura estratificada.



Arquitectura orientada a objetos. Los componentes de un sistema encapsulan los datos y las operaciones que deben aplicarse para manipular los datos. La comunicación y coordinación entre componentes se consigue mediante el paso de mensajes.

Arquitectura estratificada. En la figura 10.4 se ilustra la estructura básica de una arquitectura estratificada. Hay varias capas definidas; cada una de ellas realiza operaciones que se acercan progresivamente al conjunto de instrucciones de la máquina. En la capa externa los componentes sirven a las operaciones de interfaz de usuario. En la capa interna los componentes sirven como interfaz con el sistema operativo. Las capas intermedias proporcionan servicios de utilería y de software de aplicaciones.

Estos estilos arquitectónicos sólo son un pequeño subconjunto de los que dispone el diseñador de software.[2] Una vez que la ingeniería de requisitos define las características y restricciones del sistema que habrá de construirse, podrán elegirse el estilo arquitectónico o la combinación de estilos que mejor combinen con las características y restricciones. En muchos casos será apropiado más de un estilo y podrán diseñarse y evaluarse distintas opciones. Por ejemplo, en muchas aplicaciones con base de datos se combina un estilo por capas (apropiado para casi todos los sistemas) con una arquitectura centrada en datos.

10.3.2 Patrones arquitectónicos

Si el constructor decide construir una casa colonial con sala al centro sólo podrá aplicar un estilo arquitectónico. Los detalles del estilo (por ejemplo, número de dif-

2 Consúltese [BOS00], [HOF00], [BAS03], [SHA97], [BUS96] y [SHA96] para contar con un análisis detallado de los estilos y patrones arquitectónicos.

HOGARSEGURO

Elección de un estilo arquitectónico

El escenario: Cubículo de Jamie, tras continúa el modelado del diseño

actores: Jamie y Ed, integrantes del equipo de ería del software *HogarSeguro*.

aversación:

(haciendo el ceño): Hemos estado modelando la con UML... ya sabes, clases, relaciones y ese tipo Así que supongo que lo adecuado será aplicar la orientada a objetos.[3]

: Pero

Pero... tengo problemas para visualizar lo que es la ectura orientada a objetos. Conozco la arquitectu- llamada y retorno, una tipo de jerarquía de proce- ncional, pero orientada a objetos... no sé, me amorfo.

(sonriendo): Amorfo, ¿eh?

lo que quiero decir es que no logro visualizar estructura real, sólo el diseño de clases flotando en el espacio.

Jamie: Bueno, eso no es cierto. Hay jerarquías de clase ... piensa en la jerarquía [agregación] que hicimos para el objeto **PlanoCasa** [figura 9.3]. Una arquitectura orientada a objetos es una combinación de esa estructura y las interconexiones (ya sabes, colaboraciones) entre las clases. Lo mostraremos al describir por completo los atributos y las operaciones, los mensajes que se intercambian y la estructura de las clases.

Ed: Voy a dedicar una hora a correlacionar una arquitectura de llamado y retorno, luego regresaré aquí y pensaré en la arquitectura orientada a objetos.

Jamie: Doug no tendrá problema con eso. Él dijo que debemos considerar arquitecturas alternas. Por cierto, no hay ninguna razón para no combinar ambas arquitecturas.

Ed: Bueno, en eso estoy.

meneas, fachada de la casa, colocación de puertas y ventanas) variarán considerablemente, pero una vez que se ha tomado la decisión de la arquitectura general de la casa, el estilo se impondrá al diseño.[4]

Los patrones arquitectónicos difieren un poco.[5] Por ejemplo, cada casa (y todo estilo arquitectónico para casas) emplea un patrón *cocina*, el cual define la necesidad de colocar los artículos básicos de cocina, un fregadero, alacenas y, posiblemente,

3 Podría argumentarse que la arquitectura de *HogarSeguro* debe considerarse en un nivel más elevado que la arquitectura indicada. *HogarSeguro* tiene diversos subsistemas (funcionalidad de monitoreo de la casa, el sitio de monitoreo de la compañía y el subsistema que se ejecuta en la PC del propietario). Dentro de los subsistemas prevalecen los procesos concurrentes (por ejemplo, el monitoreo de sensores) y el manejo de eventos. A este nivel, algunas decisiones arquitectónicas se toman durante la ingeniería del sistema y el producto (capítulo 6), pero el diseño arquitectónico dentro de la ingeniería del software muy bien tendría que considerar estos aspectos.

4 Esto indica que tendrá una sala central y un pasillo, que las habitaciones estarán colocadas a la izquierda y la derecha de la sala, que la casa tendrá dos (o más) pisos, que los dormitorios estarán en la planta alta, etc. Estas "reglas" se imponen una vez que se ha tomado la decisión de usar el estilo colonial con sala al centro.

5 Es importante observar que no hay un acuerdo universal sobre la terminología. Algunas personas (por ejemplo, [BUS96]) usan los términos estilos y patrones como sinónimos, mientras que otros hacen la sutil distinción sugerida en esta sección.

reglas para ubicar cosas relacionadas con el flujo de trabajo en la habitación. Además, el patrón podría especificar la necesidad de cubiertas y acabados, iluminación, interruptores de pared o una isla central, pisos, etc. Obviamente, hay más de un diseño de cocina, pero todo diseño se concebirá dentro del contexto de la "solución" que sugiere el patrón *cocina*.

Como ya se indicó, los patrones arquitectónicos para el software definen un enfoque específico para el manejo de alguna característica de comportamiento del sistema. Bosch [BOS00] define varios patrones arquitectónicos. En los siguientes párrafos se presentan ejemplos representativos.

¡PUNTO CLAVE

Una arquitectura del software tiene varios patrones arquitectónicos que atienden temas como la concurrencia, la persistencia y la distribución.

Concurrencia. Muchas aplicaciones deben manejar varias tareas de manera tal que simulen paralelismo (es decir, esto ocurre cada vez que un solo procesador maneja varias tareas "paralelas" o componentes). Una aplicación tiene varias maneras de manejar la concurrencia, y cada una se presenta mediante un patrón arquitectónico diferente. Por ejemplo, un enfoque consiste en usar un patrón de manejo de procesos del sistema operativo que ofrece características integradas del sistema operativo que permiten la ejecución concurrente de los componentes. El patrón también incorpora funcionalidad del sistema operativo que maneja la comunicación entre los procesos, la calendarización y otras funciones requeridas para alcanzar la concurrencia. Otro método sería definir un calendarizador de tareas al nivel de aplicación. Un patrón calendarizador de tareas contiene un conjunto de objetos activos, y cada uno de ellos incluye una operación tic( ) [BOS00]. El calendarizador invoca periódicamente tic( ) para cada objeto, que luego realiza las funciones que debe realizar antes de regresar el control al calendarizador, mismo que invoca de nuevo la operación tic( ) para el siguiente objeto concurrente.

Persistencia. Los datos persisten si sobreviven después de la ejecución del proceso que los creó. Los datos persistentes se almacenan en una base de datos o un archivo; en un momento posterior, otros procesos tienen la opción de leerlos o modificarlos. En los entornos orientados a objetos la idea de un objeto persistente extiende un poco más el concepto de persistencia. Los valores de todos los atributos del objeto, el estado general de éste y otra información complementaria se almacenan para su aplicación y recuperación posterior. En general, se emplean dos patrones arquitectónicos para lograr la persistencia: 1) un patrón de *sistema de administración de base de datos* que aplica las capacidades de almacenamiento y recuperación de un sistema de administración de base de datos a la arquitectura de la aplicación, o 2) un patrón de *persistencia al nivel de la aplicación* que construye características de persistencia en la arquitectura de ésta (por ejemplo, el software de procesamiento de palabras que maneja su propia estructura de documento).

Distribución. El problema de la distribución dirige la manera en que se comunican entre sí los sistemas, o los componentes de éstos, en un entorno distribuido. Este problema incluye dos elementos: 1) la manera en que las entidades se conectan entre sí, y 2) la naturaleza de la comunicación. El patrón arquitectónico más común

tablecido para dirigir el problema de la distribución es el de *corredor*. Un corredor actúa como "intermediario" entre el componente cliente y un componente servidor. El cliente envía un mensaje al corredor (que contiene toda la información apropiada para que se realice la comunicación), el cual completa la conexión. CORBA (capítulo 30) es un ejemplo de una arquitectura de corredor.

Antes de elegir cualquiera de los patrones arquitectónicos indicados en los párrafos anteriores, debe evaluarse si es apropiado para la aplicación y el estilo arquitectónico general, además de evaluar su facilidad de mantenimiento, confiabilidad, seguridad y desempeño.

10.3.3 Organización y refinamiento

Debido a que el proceso de diseño suele dejar a un ingeniero de software con varias opciones arquitectónicas, es importante establecer un conjunto de criterios de diseño para evaluar un diseño arquitectónico. Las siguientes preguntas [BAS03] proporcionan una visión específica del estilo arquitectónico que se ha derivado.

¿Cómo se modela un estilo arquitectónico que se ha derivado?

Control. ¿Cómo se maneja el control dentro de la arquitectura? ¿Existe una jerarquía de control distintiva y, si es así, cuál es la función de los componentes dentro de esta jerarquía de control? ¿Cómo transfieren los campos el control dentro del sistema? ¿Cómo se comparte el control entre los componentes? ¿Cuál es la topología del control (es decir, cuál es la forma geométrica que asume el control)? ¿Está sincronizado el control o los componentes operan asincrónicamente?

Datos. ¿Cómo se comunican los datos entre los componentes? ¿El flujo de datos es continuo o los objetos de datos se pasan esporádicamente al sistema? ¿Cuál es el modo de transferencia de los datos (por ejemplo, los datos se pasan de un componente a otro o están disponibles globalmente para compartirse entre los componentes del sistema)? ¿Existen componentes de datos (por ejemplo, un pizarrón o almacén) y, de ser así, cuál es su papel? ¿Cómo interactúan los componentes funcionales con los de datos? ¿Los componentes de datos son pasivos o activos (es decir, interactúan activamente con otros componentes del sistema)? ¿Cómo interactúan los datos y el control dentro del sistema?

Estas preguntas proporcionan al diseñador una evaluación temprana de la calidad del diseño y sientan las bases para un análisis más detallado de la arquitectura.

10.4 DISEÑO ARQUITECTÓNICO

Cuando empieza el diseño arquitectónico debe ponerse en contexto el software que se habrá de desarrollar; es decir, el diseño debe definir las entidades externas (otros sistemas, otros dispositivos, otras personas) con las que interactúa el software y también la naturaleza de la interacción. Esta información suele adquirirse del modelo de análisis y toda la demás información reunida durante la ingeniería de requisitos. Una vez que se ha modelado el contexto y que se han descrito todas las interfaces exter-

nas del software, el diseñador especifica la estructura del sistema al definir y refinar los componentes del software que implementan la arquitectura. Este proceso prosigue de manera iterativa hasta que se obtiene una estructura arquitectónica completa

> "Un doctor entierra sus errores, pero un arquitecto sólo puede aconsejar a su cliente que plante vides."
>
> **Frank Lloyd Wright**

10.4.1 Representación del sistema en el contexto

En el capítulo 6 se indica que un ingeniero del sistema debe modelar el contexto diagrama de contexto del sistema (figura 6.4) cumple con este requisito al representar el flujo de la información dentro y fuera del sistema, la información del usuario el procesamiento relevante de soporte. Al nivel de diseño arquitectónico, un arquitecto del software aplica un diseño de contexto arquitectónico (DCA) para modelar la manera en que el software interactuará con las entidades ubicadas más allá de sus límites. La estructura genérica de los diagramas de contexto arquitectónico se muestran en la figura 10.5.

¡PUNTO CLAVE

El contexto arquitectónico representa la manera en que el software interactúa con las entidades externas a sus límites.

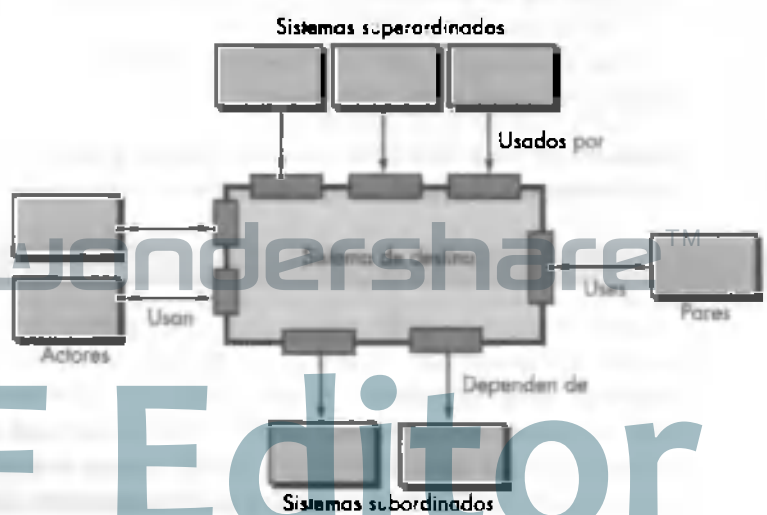
¿Cómo interactúan los sistemas entre sí?

Si se toma como referencia la figura, los sistemas que interactúan con el sistema *de destino* (el sistema para el que se está desarrollando un diseño arquitectónico) se representan así:

- *Sistemas superordinados*, los que emplean el sistema de destino como parte de algún esquema de procesamiento de nivel más elevado.
- *Sistemas subordinados:* los que utiliza el sistema de destino y que proporcionan los datos o el procesamiento necesarios para completar la funcionalidad del sistema de destino

FIGURA 10.5

Diagrama de contexto arquitectónico (adaptado de [BOS00]).



- *Sistemas al nivel de par:* los que interactúan de igual a igual (es decir, la información la producen o consumen los pares y el sistema de destino).
- *Actores:* las entidades (personas, dispositivos) que interactúan con el sistema de destino produciendo o consumiendo información necesaria para el procesamiento de requisitos.

Cada una de estas entidades externas se comunica con el sistema de destino mediante una interfaz (los pequeños rectángulos sombreados).

Para ilustrar la utilización del DCA considérese de nuevo la función de seguridad casera del producto *HogarSeguro*. El controlador general y el sistema de Internet del producto *HogarSeguro* son superdordinados a la función de seguridad y se muestran arriba de la función en la figura 10.6. La función de vigilancia es un *sistema par* y emplea (es empleado por) la función de seguridad en versiones posteriores del producto. Los páneles de propietario y control son actores que actúan como productores y consumidores de la información utilizada, producida (o de ambos tipos) por el software de seguridad. Por último, el software de seguridad emplea los sensores, que se muestran como subordinados de éste.

Como parte del diseño arquitectónico tendrían que especificarse los detalles de cada interfaz mostrada en la figura 10.6. En esta etapa deben identificarse todos los datos que fluyen al interior o el exterior del sistema de destino

10.4.2 Definición de arquetipos

Un *arquetipo* es una clase o un patrón que representa una abstracción central importantísima en el diseño de una arquitectura para el sistema de destino. En general, se requiere un conjunto relativamente pequeño de arquetipos para diseñar incluso sistemas relativamente complejos. La arquitectura del sistema de destino la integran

Punto CLAVE
Los arquetipos son los bloques abstractos de construcción de un diseño arquitectónico.

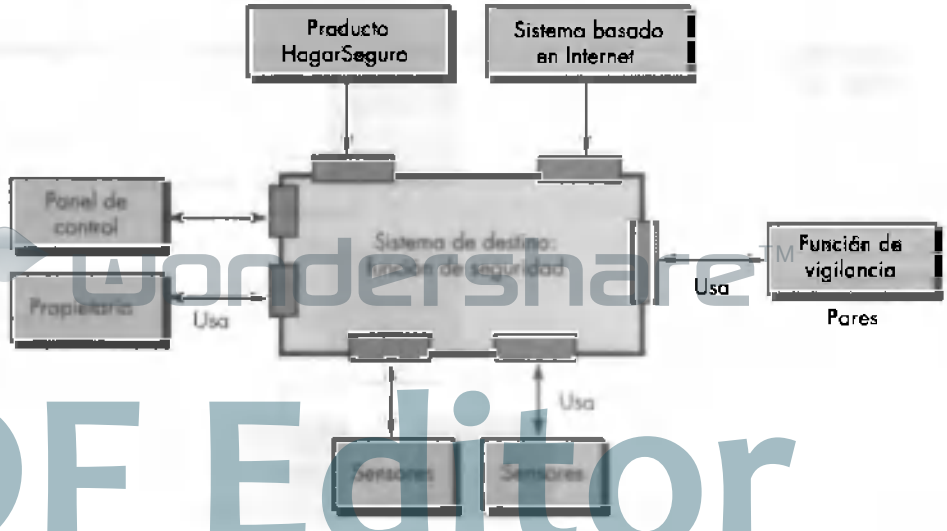


FIGURA 10.6
Diagrama de contexto arquitectónico para la función de seguridad de HogarSeguro.

arquetipos, que representan elementos estables de la arquitectura pero que pueden dar paso a instancias de muchas maneras diferentes, con base en el comportamiento del sistema.

En muchos casos, los arquetipos se derivan al examinar las clases de análisis definidas como parte del modelo de análisis. Si se continúa el análisis de la función de seguridad casera de *HogarSeguro* se definirían los siguientes arquetipos.

- **Nodo.** Representa una colección cohesiva de elementos de entrada y salida en la función de seguridad casera. Por ejemplo, un nodo estaría integrado por 1) varios sensores y 2) varios indicadores de alarma (salida).
- **Detector.** Una abstracción que abarca todo el equipo de sensores que alimenta información en el sistema de destino.
- **Indicador.** Una abstracción que representa todos los mecanismos (por ejemplo, sirena de alarma, luces parpadeantes, timbre) para indicar que está ocurriendo una condición de alarma.
- **Controlador.** Una abstracción que describe el mecanismo que permite el armado o desarmado de un nodo. Si los controladores residen en una red tienen la capacidad de comunicarse entre sí.

Cada uno de estos arquetipos se describe con la notación UML, como se muestra en la figura 10.7.

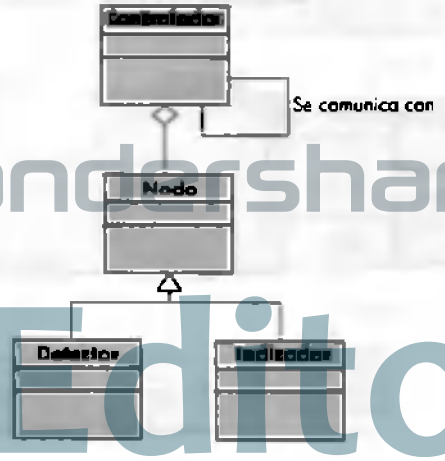
Recuérdese que los arquetipos forman la base de la arquitectura pero son abstracciones que deben refinarse aún más a medida que avanza el diseño arquitectónico. Por ejemplo, **Detector** podría refinarse en una jerarquía de clase de sensores.

10.4.3 Refinamiento de la arquitectura en componentes

A medida que se refina la arquitectura del software en componentes, la estructura del sistema empieza a emerger. Pero ¿cómo se eligen estos componentes? Para re-

FIGURA 10.7

Relaciones UML para los arquetipos de la función de seguridad de *HogarSeguro* (adaptado de [BOS00]).



ponder, el diseñador de la arquitectura empieza con las clases descritas como parte del modelo de análisis.[6] Estas clases de análisis representan entidades dentro del dominio de la aplicación (negocio) que deben atenderse dentro de la arquitectura del software. Por tanto, el dominio de la aplicación es una fuente para la derivación y el refinamiento de los componentes. Otra fuente es el dominio de la infraestructura La arquitectura debe adecuarse a muchos componentes de infraestructura que habilitan los componentes de la aplicación, pero que no tienen conexión de negocios con el dominio de la aplicación Por ejemplo, los componentes de administración de memoria, de comunicación, de base de datos y de administración de tareas suelen integrarse en la arquitectura del software

La interfaz descrita en el diagrama de contexto de la arquitectura (sección 10.4 1) indica que uno o más componentes especializados procesan los datos que fluyen por la interfaz. En algunos casos (por ejemplo, una interfaz gráfica de usuario) debe diseñarse una arquitectura completa de subsistemas con muchos componentes

> "La estructura de un sistema de software proporciona la ecología en que nace, madura y muere el código. Un hábitat bien diseñado permite el éxito en la evolución de todos los componentes necesarios de un sistema de software."
>
> **R. Pattis**

Continuando con la función de seguridad casera de *HogarSeguro*, se definirá el conjunto de componentes de nivel superior que atienden la siguiente funcionalidad:

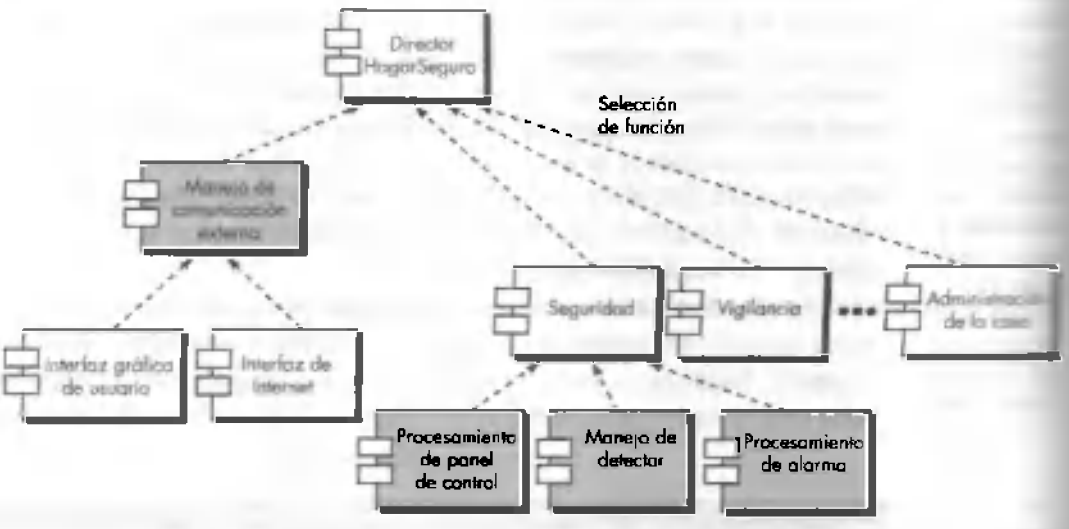
- *Administración de comunicación externa*: coordina la comunicación de la función de seguridad con entidades externas; por ejemplo, sistemas de Internet, notificación externa de alarma.
- *Procesamiento del panel de control*: maneja toda la funcionalidad del panel de control.
- *Manejo del detector*: coordina el acceso a todos los detectores conectados al sistema
- *Procesamiento de alarma*: verifica todas las condiciones de alarma y actúa sobre ellas.

Cada uno de estos componentes de nivel superior tendría que elaborarse iterativamente y luego posicionarse dentro de la arquitectura general de *HogarSeguro*. Para cada uno se definirían clases de diseño (con los atributos y las operaciones apropiadas). Sin embargo, es importante observar que los detalles de diseño de todos los atributos y las operaciones sólo se especificarían hasta la realización del diseño en el nivel de componentes (capítulo 11).

En la figura 10.8 se ilustra la estructura arquitectónica general (representada como un diagrama de componentes UML). El componente *Manejo de comunicación externa*

6 Si se elige un enfoque convencional (no orientado a objetos) es posible derivar los componentes del modelo de flujo de datos En la sección 10.6 se analizará este enfoque

FIGURA 10.8 Estructura general de la arquitectura de *HogarSeguro* con componentes de nivel



adquiere las transacciones provenientes de los componentes que procesan la *interfaz gráfica de usuario* de *HogarSeguro* y la *interfaz de Internet*. El componente *director de HogarSeguro* maneja esta información y selecciona la función de producto apropiada (en este caso, seguridad). El componente *procesamiento de panel de control* interactúa con el propietario para armar o desarmar la función de seguridad. El componente *manejo de detector* agrupa los sensores para detectar una condición de alarma, y el componente *procesamiento de alarma* produce una salida cuando se detecta la alarma.

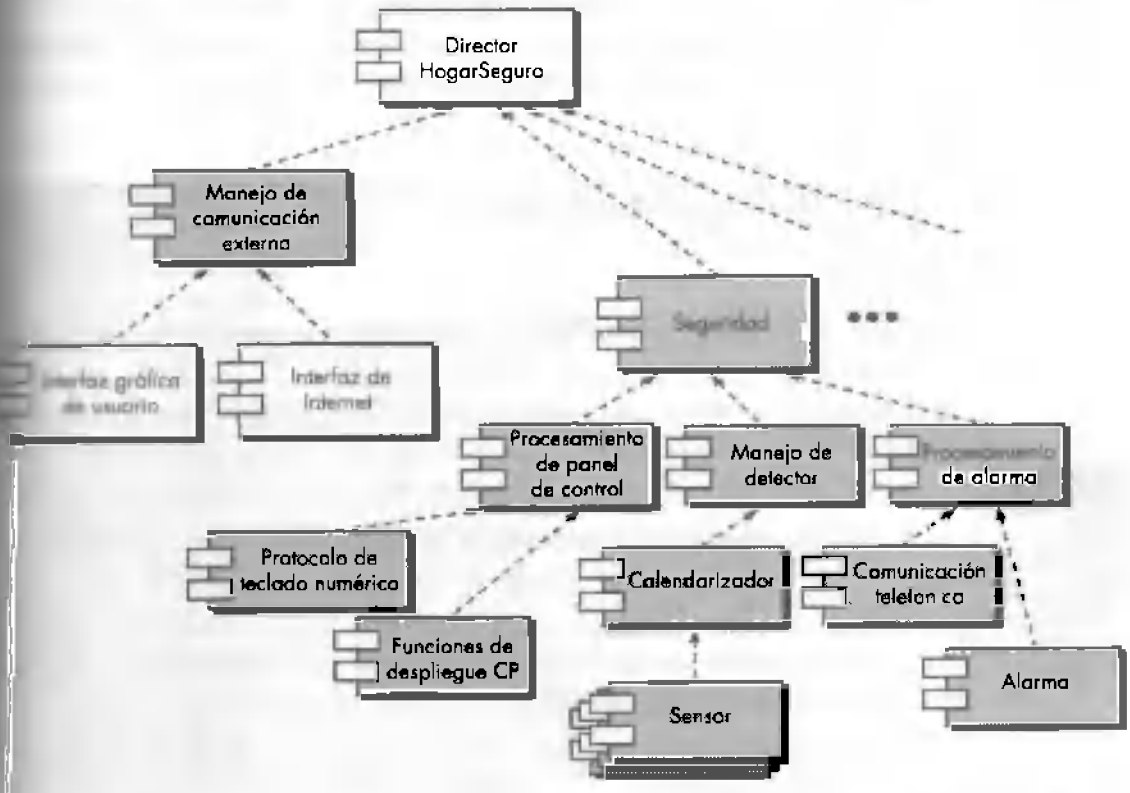
10.4.4 Descripción de la creación de instancias del sistema

El diseño arquitectónico que se ha modelado hasta este punto a todavía es de un nivel relativamente alto. Se ha representado el contexto del sistema; se han definido los arquetipos que indican las abstracciones importantes dentro del dominio del problema; es evidente la estructura general del sistema; y se han identificado los principales componentes del software. Sin embargo, aún se necesita mayor refinamiento (recuérdese que todo el diseño es iterativo).

Esto se logra desarrollando una *instancia* de la arquitectura. Es decir, la arquitectura se aplica a un problema específico con el propósito de demostrar que la estructura y los componentes son apropiados.

En la figura 10.9 se ilustra una instancia de la arquitectura *HogarSeguro* para el sistema de seguridad. Los componentes que muestra la figura 10.8 se refinan aún más para mostrar detalles adicionales. Por ejemplo, el componente *manejo de detector* interactúa con el componente de infraestructura *calendarizador* que implementa el agrupamiento "concurrente" de cada objeto *sensor* del sistema de seguridad. Una elaboración similar se realiza para cada uno de los componentes representados en la figura 10.8

Figura 10.9 Instancia de la función de seguridad con elaboración de componentes.



HERRAMIENTAS DE SOFTWARE

Diseño arquitectónico

Objetivo: Las herramientas de diseño arquitectónico modelan la estructura general del [illegible] al representar interfases, dependencias, [illegible]es e interacciones de los componentes.

[illegible]canica: Las herramientas cuentan con mecánicas [illegible]as. En casi todos los casos, la capacidad del diseño [illegible]nico es parte de la funcionalidad de las [illegible]nientas automatizadas para modelado de análisis y [illegible]

Herramientas representativas[7]

[illegible]n, desarrollada por Synthis Corp. *(www.synthis.com)*, es una herramienta especializada para el diseño y la construcción de arquitectura específicas de componentes Web.

ObjectiF, desarrollada por microTOOL GmbH *(www.microtool.com)*, es una herramienta de diseño UML que lleva a arquitecturas (por ejemplo, Coldfusion, J2EE, Fusebox) sensibles a la ingeniería de software basada en componentes (capítulo 30).

Rational Rose, desarrollado por Rational (www.rational.com), es una herramienta de diseño basada en UML que soporta todos los aspectos del diseño arquitectónico.

7 Las herramientas expuestas sólo representan una muestra de esta categoría. En casi todos los casos los nombres son marcas registradas por sus respectivos desarrolladores.

10.5 EVALUACIÓN DE DISEÑOS ARQUITECTÓNICOS ALTERNOS

En el mejor de los casos, el diseño produce varias opciones arquitectónicas que se evalúan para determinar cuál es la más apropiada respecto al problema que habrá de resolverse. En las siguientes secciones se analizaran diseños arquitectónicos alternos

> "Tal vez sea en el sótano. Déjame ir escaleras arriba y revisar"
>
> M. C. Escher

10.5.1 Un método de análisis de compensación para la arquitectura

El Instituto de Ingeniería del Software (SEI, por sus siglas en inglés) ha desarrollado un *método de análisis de compensación para la arquitectura* (MACA) [KAZ98] que define un proceso de evaluación iterativo para las arquitecturas del software. Las siguientes actividades de análisis del diseño se realizan iterativamente:

Referencia Web
Información a fondo sobre MACA se obtendrá en www.sei.cmu.edu/ata/ata_method.html.

1. *Recopilar escenarios.* Se desarrolla un conjunto de casos de uso (capítulos 7 y 8) para representar el sistema desde el punto de vista del usuario.
2. *Deducir requisitos, restricciones y descripción de entornos.* Esta información se requiere como parte de la ingeniería de requisitos y se usa para asegurarse de que se atiendan todas las preocupaciones de los participantes
3. *Describir los estilos/patrones arquitectónicos que se han elegido para dirigir los escenarios y requisitos*
4. *Evaluar los atributos de calidad al considerar cada atributo de manera aislada.* Entre los atributos de calidad para la evaluación del diseño arquitectónico se incluyen confiabilidad, desempeño, seguridad, facilidad de mantenimiento, flexibilidad, facilidad de prueba, portabilidad, facilidad de reutilización e interoperabilidad.
5. *Identificar la sensibilidad de los atributos de calidad respecto de varios atributos arquitectónicos para un estilo arquitectónico específico.* Esto se logra haciendo pequeños cambios en la arquitectura y determinando la sensibilidad al cambio de un atributo de calidad, como el desempeño. Cualquier atributo al que afecte significativamente la variación en la arquitectura se considerará un punto de sensibilidad.
6. *Analizar las arquitecturas alternas (desarrolladas en el paso 3) empleando el análisis de sensibilidad aplicado en el paso 5.* El SEI describe este enfoque de la siguiente manera [KAZ98]:

Una vez determinados los puntos de sensibilidad arquitectónica se encuentran los puntos en que se requiere compensación con sólo identificar los elementos arquitectónicos a los

que son sensibles varios atributos. Por ejemplo, el desempeño de una arquitectura cliente-servidor sería muy sensible al número de servidores (el desempeño aumentará, dentro de cierto rango, al aumentar el número de servidores). . Por tanto, el número de servidores es un punto de compensación para esta arquitectura

Estos seis pasos representan la primera iteración MACA. Con base en los resultados de los pasos 5 y 6 será posible eliminar algunas opciones arquitectónicas, modificar una o más de las arquitecturas restantes y representarlas con más detalle, y luego aplicar de nueva cuenta los pasos de MACA [8]

HogarSeguro

Evaluación de la arquitectura

El escenario: Oficina de Doug as continúa el modelado del diseño

s: Vinod, Jamie, Shakira y Ed, integrantes de ingeniería del *software HogarSeguro* Doug Miller, jefe del grupo de ingeniería del

rsación:

Sé que están derivando un par de arquitecturas , para el producto *HogarSeguro*, y eso es Supongo que mi pregunta es ¿cómo elegiremos la

Estoy trabajando en un estilo de llamada y retorno, y Jamie o yo vamos a derivar una arquitectura a objetos.

Muy bien, ¿y cómo la elegiremos?

Tomé un curso de diseño en mis años de d, y recuerdo que hay varias maneras de

Las hay, pero son un poco académicas Mira, que podemos hacer nuestra evaluación y elegir la casos de uso y escenarios

¿No es lo mismo?

No cuando estás hablando de evaluación arquitectónico. Ya tenemos un conjunto completo de casos de uso. Así que apliquemos cada una de las arquitecturas y veamos cómo reacciona el sistema; es decir, cómo funcionan los componentes y los conectores en el contexto del caso de uso.

Ed: Es una buena idea. Eso nos asegura que no dejaremos nada fuera

Vinod: Cierto, pero también nos indica si el diseño arquitectónico no es directo, si tiene que doblarse como moño para hacer el trabajo, por decirlo así

Jamie: ¿No es lo mismo un escenario que un caso de uso?

Vinod: No, en este caso un escenario es algo diferente

Doug: Estás hablando de un escenario de calidad o uno de cambio, ¿o no?

Vinod: Sí. Lo que hacemos es regresar con los participantes y preguntarles cómo cambiará *HogarSeguro* en los siguientes tres años, por decir algo. Ya sabes, nuevas versiones, características, ese tipo de cosas. Construimos un conjunto de escenarios de cambio También desarrollamos un conjunto de escenarios de calidad que definen los atributos que nos gustaría ver en la arquitectura del software.

Jamie: Y los aplicamos a las opciones

Vinod: Exacto. El estilo que maneje mejor los casos de uso y los escenarios es el que escogeremos.

8 El *método de análisis de la arquitectura del software* (MAAS) es una alternativa a MACA y vale la pena que los lectores interesados en el análisis arquitectónico lo examinen. Un artículo sobre MAAS se descarga de http://www.sei.cmu.edu/publications/articles/saam_metho-propert-sas.htm

10.5.2 Complejidad arquitectónica

Una técnica útil para evaluar la complejidad general de una arquitectura determinada consiste en considerar las dependencias entre componentes dentro de la arquitectura. Estas dependencias las orienta la información, el flujo de control, o ambas dentro del sistema. Zhao [ZHA98] sugiere tres tipos de dependencias:

Dependencias compartidas que representan relaciones de dependencia entre consumidores que usan el mismo recurso o los productores que producen para los mismos consumidores. Por ejemplo, supónganse dos componentes *u* y *v* que se refieren a los mismos datos globales. Entonces existe una relación de dependencia compartida entre *u* y *v*.

Dependencias de flujo que representan las relaciones de dependencia entre productores y consumidores de recursos. Por ejemplo, para dos componentes u y *v*, si u debe completarse antes de que el control pase a *v* (prerrequisito) o si u se comunica con *v* mediante parámetros, entonces existe una relación de dependencia de flujo entre *u* y *v*.

Dependencias restringidas que representan restricciones al flujo relativo de control entre un conjunto de actividades. Por ejemplo, si dos componentes *u* y *v* no pueden ejecutarse al mismo tiempo (exclusión mutua), entonces existe una relación de dependencia restringida entre *u* y *v*.

Las dependencias compartidas y de flujo que señala Zhao son similares al concepto de acoplamiento descrito en el capítulo 9. Acoplamiento es un concepto importante del diseño que se aplica al nivel de arquitectura y de componentes. En el capítulo 15 se expondrán métricas simples para evaluar el acoplamiento.

10.5.3 Lenguajes de descripción arquitectónica

El arquitecto de una casa tiene un conjunto de herramientas estandarizadas y de notación que permite representar el diseño de una manera poco ambigua y fácil de comprender. Aunque el arquitecto del software puede dibujar con notación UML se necesitan otras formas de diagramación, y unas cuantas herramientas relacionadas para aplicar un enfoque más formal a la especificación de un diseño arquitectónico.

El *lenguaje de descripción arquitectónica* (LDA) proporciona una semántica y sintaxis para describir una arquitectura del software. Hofmann y sus colegas [HOF] sugieren que un LDA debe proporcionar al diseñador la capacidad de separar componentes arquitectónicos, de integrar componentes individuales en bloques arquitectónicos mayores y de representar interfaces (mecanismos de conexión) entre componentes. Una vez que se hayan establecido las técnicas descriptivas, basadas en el lenguaje para diseño arquitectónico, es más probable que se establezcan los métodos de evaluación efectivos para la arquitectura a medida que el diseño evoluciona.

HERRAMIENTAS DE SOFTWARE

Lenguajes de descripción arquitectónica

El siguiente resumen de varios LDA importantes lo preparó Rickard Land [LAN02] y se [illegible] con el permiso del autor. Debe observarse que [illegible] cinco LDA se han desarrollado con fines de [illegible]ción y no son productos comerciales

[illegible] (pasat.stanford.edu/rapide/) [LUC95] construye a partir de la noción de conjuntos parciales ordenados

[illegible] (www.cs.cmu.edu/~UniCon) [SHA96] define arquitecturas de software desde el punto de vista de abstracciones útiles para los diseñadores

[illegible] (www.cs.cmu.edu/~able/aesop/) [GAR94] aliende el problema de reutilización del estilo.

Wright (www.cs.cmu.edu/~able/wright/) [ALL97] formaliza los estilos arquitectónicos empleando predicados, lo que permite comprobar la estática para determinar la consistencia y el grado en que se ha completado una arquitectura.

Acme (www.cs.cmu.edu/~acme/)[GAR00] es un LDA de segunda generación.

UML (www.uml.org/) incluye muchos de los artefactos necesarios para descripciones arquitectónicas; no es tan completo como otros LDA

10.6 CORRELACIÓN DEL FLUJO DE DATOS EN UNA ARQUITECTURA DEL SOFTWARE

Los estilos analizados en la sección 10.3.1 representan arquitecturas radicalmente diferentes; por tanto, no debe sorprender que no exista una completa correlación (o mapeo) que logre la transición del modelo de análisis a diversos estilos arquitectónicos. En realidad, no hay correlación práctica para algunos estilos arquitectónicos. El diseñador debe abocarse a la traducción de requisitos en diseño para estos estilos mediante las técnicas analizadas en la sección 10.4

Ilustrar un enfoque de la correlación arquitectónica requiere tener en cuenta una técnica de correlación aplicada a la arquitectura de *llamada y retorno* (una estructura muy común en muchos tipos de sistemas). Esta técnica de correlación permite que un diseñador derive arquitecturas de llamada y retorno razonablemente complejas a partir de diagramas de flujo de datos dentro del modelo de análisis. La técnica, a veces denominada *diseño estructurado*, se presenta en libros de Myers [MYE78] y Yourdon y Constantin [YOU79].

El diseño estructurado suele caracterizarse como un método de diseño orientado a flujo de datos, ya que proporciona una conveniente transición de un diagrama de flujo de datos (capítulo 8) a una arquitectura del software. El tipo de flujo de información es el controlador para el enfoque de correlación o mapeo.

10.6.1 Flujo de transformación

La información debe entrar y salir del software en una forma lógica para "la realidad externa". Por ejemplo, los datos escritos en un teclado, los tonos de una línea telefónica y las imágenes de video en una aplicación multimedia son medios de información de la realidad externa. Estos datos externos deben convertirse en una forma interna para el procesamiento. La información ingresa en el sistema por rutas que

transforman los datos externos en una forma interna. Estas rutas se identifican como *flujo de entrada*. En el núcleo del software ocurre una transición. Los datos entrantes se pasan por un *centro de transformación* y empiezan a moverse por rutas que ahora los llevan "fuera" del software. Al desplazamiento de los datos por estas rutas se le denomina *flujo de salida*. El flujo general de datos ocurre de manera secuencial y sigue una o unas cuantas rutas "en línea recta".[9] Cuando un segmento de un diagrama de flujo de datos muestra estas características está presente el flujo de *transformación*.

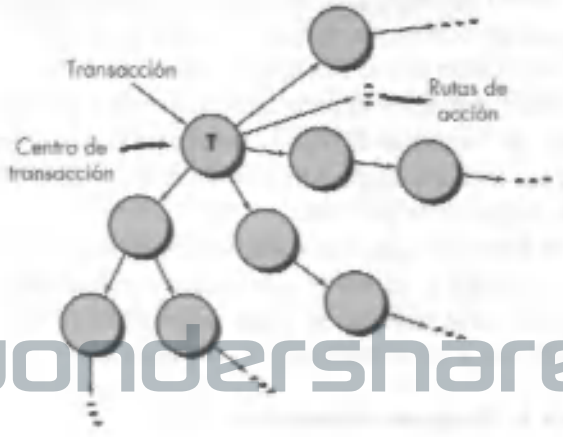
10.6.2 Flujo de transacción

Al flujo de información suele caracterizarlo un solo elemento de datos, llamado *transacción*, que dispara otro flujo de datos por una de muchas rutas. Cuando un diagrama de flujo de datos (DFD) asume la forma mostrada en la figura 10.10 está presente el flujo de transacción.

Al flujo de transacción lo caracterizan los datos que se desplazan por un camino entrante que convierte la información proveniente del exterior en una transacción. Ésta se evalúa y, con base en su valor, se inicia el flujo por una de muchas *rutas de acción*. Al elemento que concentra y distribuye el flujo de información, del que emanan muchas rutas de acción, se le denomina *centro de transacción*.

Debe observarse que, dentro del DFD de un sistema grande, tienen que estar presentes los flujos de transformación y transacción. Por ejemplo, en una transacción orientada a flujo, el flujo de información por un camino de acción puede tener características del flujo de transformación.

FIGURA 10.10
Flujo de transacción.



9 Una correlación obvia para este tipo de flujo de información es la arquitectura de flujo de datos descrita en la sección 10.3.1. Sin embargo, hay muchos casos en que esta arquitectura tal vez no sea la mejor elección para un sistema complejo. Entre los ejemplos se incluyen sistemas que experimentarán cambios sustanciales con el tiempo o sistemas en los cuales el procesamiento asociado con el flujo de datos no necesariamente resulta secuencial.

10.6.3 Correlación de transformaciones

La *correlación de transformaciones* es un conjunto de pasos de diseño que permite que un DFD con características de flujo de transformación se correlacione con un estilo arquitectónico específico. Ilustrar este enfoque requiere considerar de nuevo la función de seguridad *HogarSeguro*.[10] Un elemento del modelo de análisis es un conjunto de diagramas de flujo de datos que describen el flujo de información dentro de la función de seguridad. Con el fin de correlacionar estos diagramas en una arquitectura se llevan a cabo los siguientes pasos de diseño

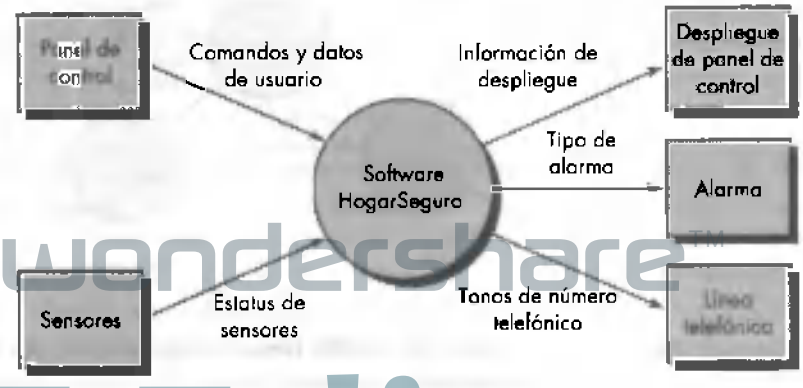
Paso 1. Revisar el modelo fundamental del sistema. El modelo fundamental del sistema o diagrama de contexto describe la función de seguridad como una sola transformación, que representa a los productores y consumidores externos de los datos que fluyen al interior y hacia fuera de la función. En la figura 10.11 se describe un modelo de nivel 0; en la 10.12 se muestra un flujo de datos refinado para la función de seguridad

CONSEJO

Si el DFD se refina con más en este momento, se debe hacer un esfuerzo por crear burbujas que muestren una cohesión elevada.

Paso 2. Revisar y refinar los diagramas de flujo de datos para el software. La información obtenida de los modelos de análisis se refina para producir mayor detalle. Por ejemplo, se examina el DFD de nivel 2 para *supervisar sensores* (figura 10.13) y se deriva un diagrama de flujo de datos de nivel 3, como se muestra en la figura 10.14. En el nivel 3 cada transformación del DFD muestra una cohesión relativamente alta (capítulo 9) Es decir, el proceso implícito en una transformación realiza una función única y distintiva que puede implementarse como un componente del software *HogarSeguro*. Por tanto, el DFD de la figura 10.14 contiene suficiente detalle para un "primer corte" en el diseño arquitectónico del subsistema *supervisar sensores*, y se sigue adelante sin mayor refinamiento

FIGURA 10.11
DFD al nivel de contexto para la función de seguridad de HogarSeguro.



10 Sólo se considera una parte de la función de seguridad de *HogarSeguro* que usa el panel de control. No se tendrán en cuenta otras características expuestas al principio de este libro y en este capítulo.

FIGURA 10.12

DFD de nivel 1 para la función de seguridad de *HogarSeguro*

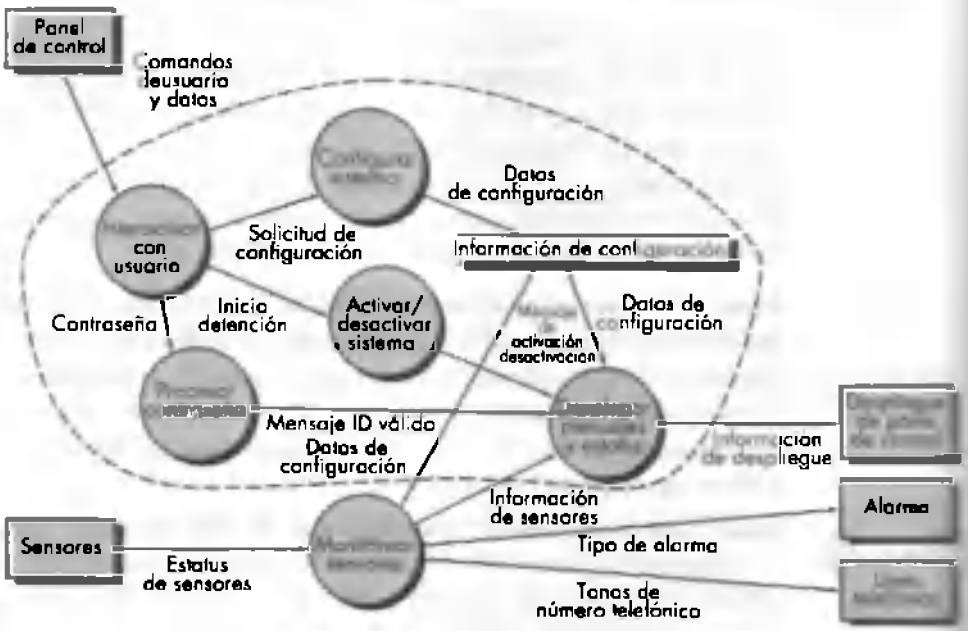
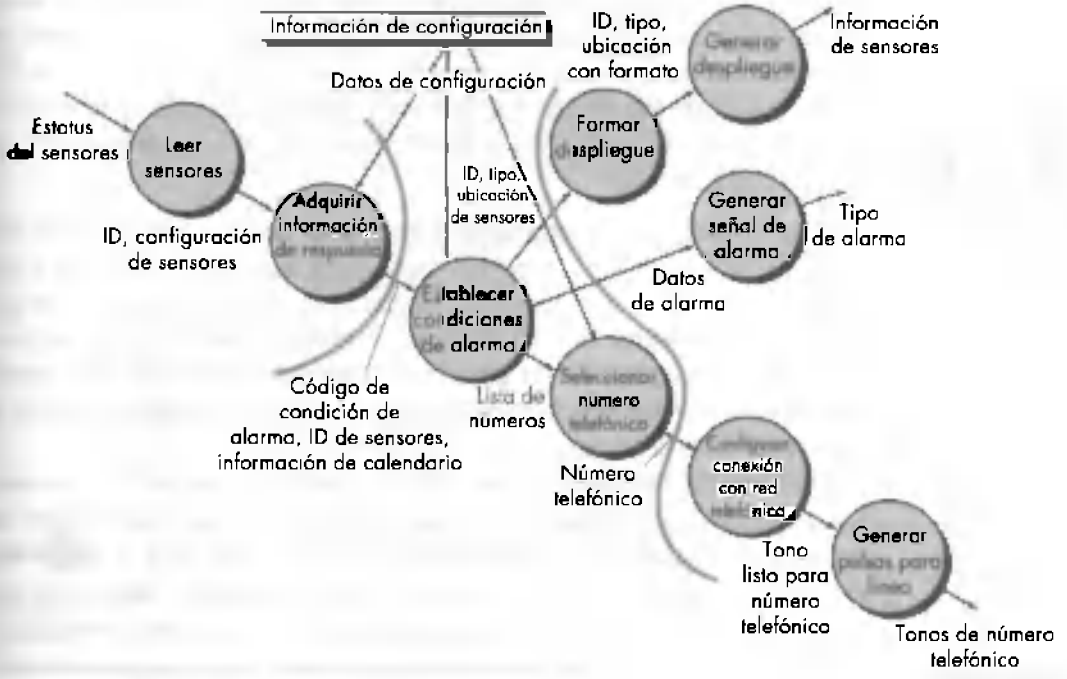


FIGURA 10.13

DFD de nivel 2 que refina la transformación *monitorear sensores*

Paso 3. Determinar si el DFD tiene características de flujo de transformación o de transacción. En general, siempre es posible representar el flujo de información dentro de un sistema como una transformación. Sin embargo, cuando se encuentra una característica obvia de transacción (figura 10.10), se recomienda una correlación de diseño diferente. En este paso el diseñador selecciona las car

FIGURA 10.14 DFD de nivel 3 para *monitorear sensores* con límites de flujo.



ticas de flujo global (de todo el software) con base en la naturaleza prevaleciente del DFD. Además, se aislan las regiones locales del flujo de transformación o transacción. Estos *subflujos* se aprovechan para refinar la arquitectura del programa derivado de una característica global descrita antes. Por ahora, la atención se centrará sólo en el flujo de datos del subsistema *monitorear* sensores descrito en la figura 10 14

Al evaluar el DFD (figura 10 14) se aprecia que los datos entran al software por una ruta de entrada y salen por tres rutas de salida No participa un centro de transacción distintivo (aunque la transformación establece condiciones de alarma que podrían percibirse como tales). Por tanto, se supondrá una característica general de transformación para el flujo de la información.

Paso 4. Aislar el centro de transformación al especificar límites de flujo de entrada y salida. En la sección anterior, el flujo de entrada se describió como un camino que convierte la información con forma externa en información interna; el flujo de salida hace la conversión inversa Los límites de los flujos de entrada y salida están abiertos a la interpretación. Es decir, diferentes diseñadores seleccionarán puntos ligeramente diferentes del flujo como posición de los límites. En realidad, las soluciones alternas de diseño se obtienen al modificar la posición de los límites de flujo Aunque debe tenerse cuidado al seleccionar los límites, la variación en una burbuja a lo largo del camino de flujo generalmente tendrá poco impacto en la estructura final del programa

Los límites de flujo en este ejemplo se ilustran como curvas sombreadas que corren verticalmente por el flujo de la figura 10.14. Las transformaciones (burbujas) que constituyen el centro de transformación caen dentro de los dos límites sombreados que corren de arriba abajo en la figura. Podría respaldarse el argumento de que es posible reajustar un límite (por ejemplo, podría proponerse un límite de flujo de entrada que separe *leer sensores* y *adquirir información de respuesta*). En este paso de diseño debe resaltarse la selección de límites razonables, no la larga iteración sobre la posición de los límites.

CONSEJO

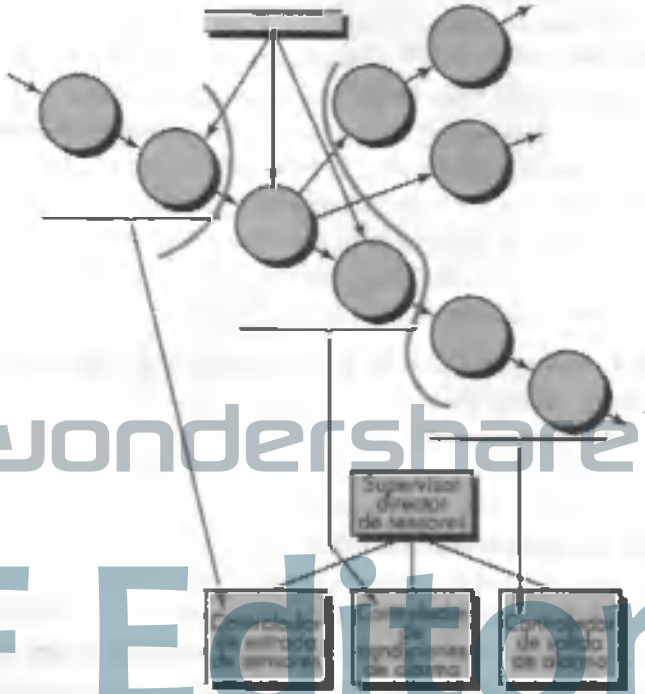
No se debe adoptar un enfoque dogmático en esta etapa. Tal vez sea necesario establecer dos o más controladores para procesamiento o cálculo de entrada, con base en la complejidad del sistema que habrá de construirse. Si el sentido común dicta este enfoque, ¡úsese!

Paso 5. Realizar una "factorización de primer nivel". La arquitectura del programa que se ha derivado a partir de los resultados de la correlación lleva a una distribución descendente del control. La *factorización* genera una estructura de programa en que los componentes descendentes se encargan de la toma de decisiones y los componentes de bajo nivel realizan más trabajo de entrada, cálculo y salida. Los componentes de nivel intermedio se encargan de parte del control y realizan cantidades moderadas de trabajo.

Al encontrar el flujo de transformación se correlaciona un DFD con una estructura específica (una arquitectura de llamada y retorno) que proporciona control para el procesamiento de la información de entrada, de transformación y de salida. En la figura 10.15 se muestra esta factorización de primer nivel para el subsistema *supervisar sensores*. Un controlador principal (llamado *supervisar director de sensores*) resi-

FIGURA 10.15

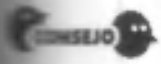
Factorización de primer nivel para *supervisar sensores*.



en la parte superior de la estructura del programa y coordina las siguientes funciones subordinadas de control:

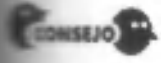
- Un controlador de procesamiento de información entrante, llamado *controlador de entrada de sensores*, coordina la recepción de todos los datos de entrada.
- Un controlador de flujo de transformación, llamado *controlador de condiciones de alarma*, supervisa todas las operaciones de los datos en forma adecuada para el trabajo interno (por ejemplo, un módulo que invoca varios procedimientos de transformación de datos).
- Un controlador de procesamiento de información saliente, llamado *controlador de salida de alarma*, coordina la producción de información de salida

Aunque una estructura de tres picos se desprende de la figura 10.15, flujos complejos en sistemas grandes llegan a pedir dos o más módulos de control para cada una de las funciones genéricas de control descritas. El número de módulos en el primer nivel debe limitarse al mínimo posible para realizar las funciones de control y aun mantener buenas características de independencia funcional.



Paso 6. Realice una "factorización de segundo nivel". La factorización de segundo nivel se logra al correlacionar las transformaciones individuales (burbujas) de una DFD con los módulos apropiados dentro de la arquitectura. Si se empieza en el límite del centro de transformación y se desplaza hacia fuera por rutas de entrada y luego por rutas de salida, las transformaciones se correlacionarán en niveles subordinados de la estructura del software. En la figura 10.16 se muestra el enfoque general de la factorización de segundo nivel.

Aunque en la figura 10.16 se ilustra una correlación uno a uno entre transformaciones de DFD y módulos de software, suelen ocurrir correlaciones diferentes. Es posible combinar dos o hasta tres burbujas y representarlas como un componente, o expandir una sola burbuja en dos o más componentes. Consideraciones prácticas y medidas de la calidad del diseño dictan el resultado de la factorización de segundo nivel. La revisión y el refinamiento producen cambios en la estructura, pero sirven como diseño de "primera iteración"



La factorización de segundo nivel para el flujo de entrada se realiza de la misma manera. La factorización se realiza nuevamente al desplazarse hacia fuera, desde el límite del centro de transformación en el lado del flujo de entrada. El centro de transformación del software del subsistema *monitorear sensores* se correlaciona de manera un poco diferente. Cada conversión de datos o transformación de datos de la parte de la transformación del DFD se correlaciona con un módulo que está subordinado al controlador de transformación. En la figura 10.17 se muestra una arquitectura de primera iteración completa

Los componentes correlacionados de la manera anterior y mostrados en la figura 10.17 representan un diseño inicial de la arquitectura del software. Aunque el nombre de los componentes indica una función, debe escribirse una breve explica-

FIGURA 10.16

Factorización de segundo nivel para *supervisar sensores*.

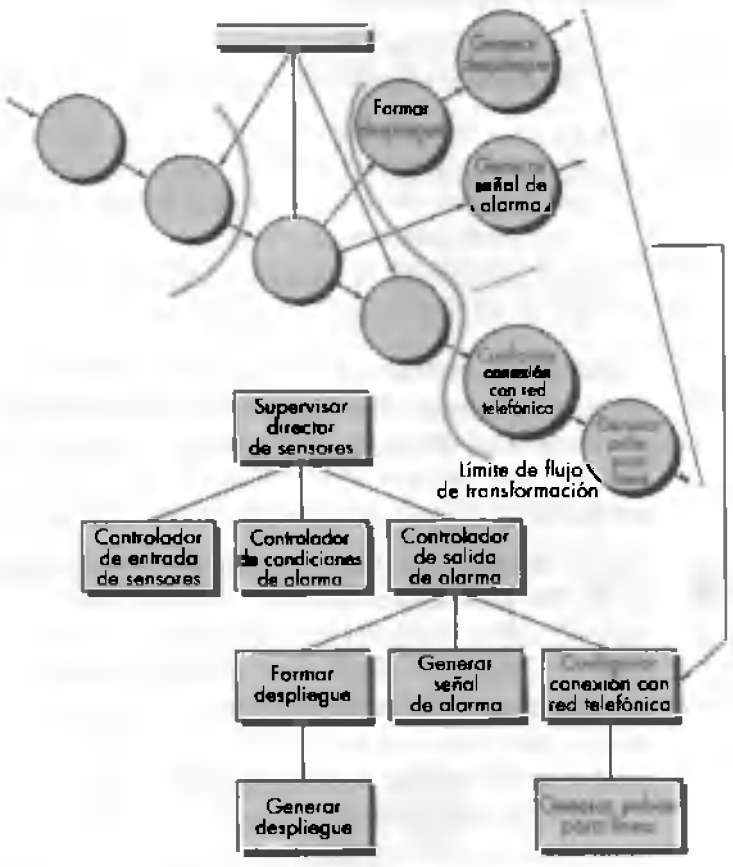
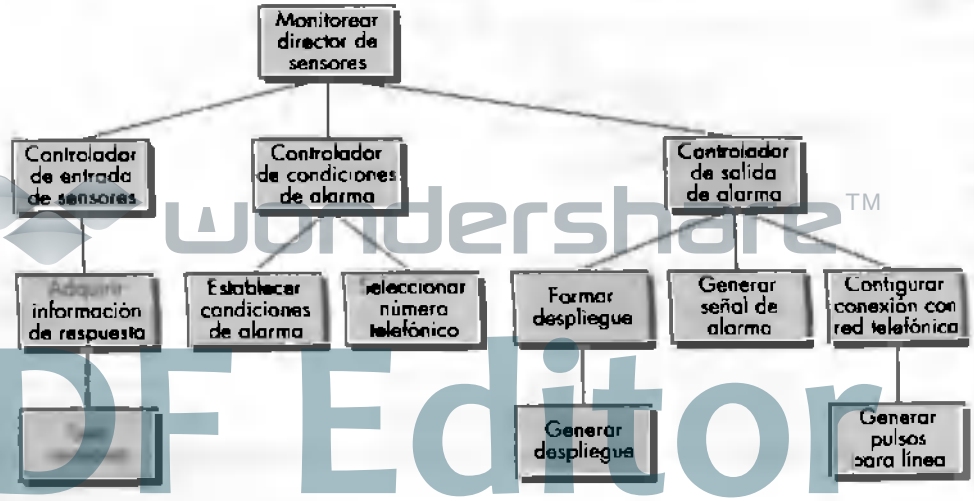


FIGURA 10.17 Estructura de primera iteración para *supervisar sensores*.



ción de su procesamiento (adaptado de la especificación creada durante el modelado del análisis)

Paso 7. Refinar la arquitectura de primera iteración empleando diseño heurístico para mejorar la calidad del software. Siempre es posible refinar una arquitectura de primera iteración si se aplican conceptos de independencia funcional (capítulo 9) Los componentes se expanden o contraen para producir una factorización sensible, una buena cohesión, un acoplamiento mínimo y, lo que es más importante, una estructura que se implemente sin dificultades, se pruebe sin confusión y se mantenga sin problemas.

Los refinamientos los determinan los métodos de análisis y evaluación descritos brevemente en la sección 10.5, además de las consideraciones prácticas y el sentido común. Por ejemplo, hay ocasiones que el controlador del flujo de datos de entrada resulta totalmente innecesario, que se requiere algún procesamiento de entrada en un componente subordinado al controlador de transformación, que no puede evitarse el acoplamiento elevado debido a los datos globales, o que no logran alcanzarse las características óptimas de la estructura. Los requisitos del software, junto con el juicio humano, deben servir para tomar la decisión final.

El objetivo de los siete pasos precedentes es desarrollar una representación arquitectónica del software Es decir, una vez definida la estructura, es posible evaluar y refinar la arquitectura del software al tener un panorama general de él Las modificaciones hechas en este momento requieren poca información adicional, pero tendrán un fuerte impacto en la calidad del software.

El lector debe hacer una breve pausa y considerar la diferencia entre el enfoque del diseño descrito y el proceso de "escribir programas". Si el código es la única representación del software, el desarrollador tendrá gran dificultad para evaluar o refinar a voluntad, en un nivel global u holístico. En realidad, tendrá dificultad para "ver el bosque entre los árboles".

HogarSeguro

Refinamiento de una arquitectura de primera iteración

El escenario: Cubículo de Jamie, [illegible]s se continúa con el modelado del diseño

actores: Jamie y Ed, integrantes del equipo de in[illegible]ría del software *HogarSeguro*

conversación:

[illegible] acaba de completar un diseño de primera iteración [illegible] subsistema *monitorear sensores*. Se detiene para pe[illegible] su opinión a Jamie.]

Ed: Mira, aquí está la arquitectura que he obtenido [Ed muestra a Jamie la figura 10.17, que ella estudia por unos momentos.]

Jamie: Está muy bien, pero creo que podemos hacer algunas cosas para simplificarla y mejorarla.

Ed: ¿Cómo cuáles?

Jamie: Bueno, ¿por qué usas el componente *controlador de entrada de sensores*?

Ed: Porque necesitas un sensor para la correlación.

Jamie: En realidad no. El controlador no hace mucho, porque estamos manejando un solo camino de flujo para los datos de entrada. Podemos eliminar el controlador sin exponernos a efectos colaterales.

Ed: Puedo vivir con eso. Haré el cambio y...

Jamie (sonriendo): ¡Espera! También podemos reducir los componentes *establecer condiciones de alarma* y *seleccionar número telefónico*. En realidad el controlador de transformación que muestras no es necesario, y la pequeña reducción de la cohesión resulta tolerable.

Ed: Simplificación, ¿eh?

Jamie: Así es. Y mientras hacemos refinamientos, sería buena idea reducir los componentes *formar despliegue* y *generar despliegue*. El formato del despliegue del panel de control es simple. Podemos definir un nuevo módulo llamado *producir despliegue*.

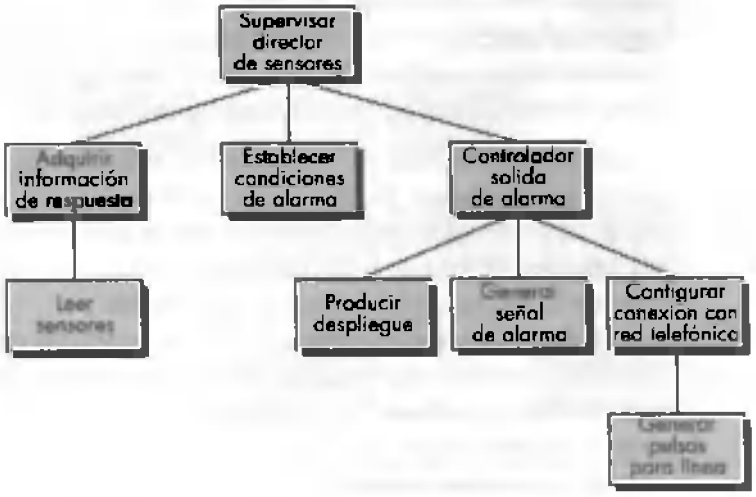
Ed (trazando un boceto): ¿De modo que esto es lo que crees que podemos hacer?

(Muestra a Jamie la figura 10.18.)

Jamie: Es un inicio.

FIGURA 10.18

Estructura refinada del programa para *supervisar sensores*.



10.6.4 Correlación de transacciones

En muchas aplicaciones de software, un solo elemento de datos dispara varios flujos de información que afectan una función relacionada con el elemento de datos que dispara. El elemento de datos, llamado *transacción*, y sus correspondientes características de flujo se analizaron en la sección 10.6.2. En esta sección se considerarán los pasos de diseño empleados para correlacionar el flujo de transacción con una arquitectura de software.

La correlación de transacciones se ilustrará si se considera el subsistema de interacción con el usuario de la función de seguridad de *HogarSeguro*. En la figura 10.12 se muestra el flujo de datos de nivel 1 para este subsistema. Al refinar el DFD se deriva un diagrama de flujo de datos de nivel 2, como se muestra en la figura 10.19. El objeto de datos **comandos de usuario** fluye dentro del sistema y genera un flujo de información adicional por una de tres rutas de acción. Un solo elemento

de datos, **tipo de comando**, hace que el flujo de datos se expanda hacia fuera del concentrador. Por tanto, la característica general del flujo de datos está orientada a la transacción.

Debe observarse que el flujo de información a lo largo de dos de las tres rutas de acción acomoda el flujo de entrada adicional (por ejemplo, **parámetros y datos del sistema** son entradas del camino de acción "configurar"). Todas las rutas de acción fluyen en una sola transformación, *desplegar mensajes y estatus*.

Los pasos del diseño para la correlación de transacciones son similares y en algunos casos idénticos a los pasos para correlación de transformaciones (sección 10.6.3). Una diferencia importante se encuentra en la correlación del DFD con la estructura del software.

Paso 1. Revisar el modelo fundamental del sistema.

Paso 2. Revisar y refinar los diagramas de flujo de datos para el software.

Paso 3. Determinar si el DFD tiene características de flujo de transformación o de transacción.

Los pasos 1, 2 y 3 son idénticos a los correspondientes en la correlación de transformaciones. El DFD que se muestra en la figura 10.19 tiene una característica de flujo de transformación clásico. Sin embargo, el flujo por las rutas de acción que emanan de la burbuja *invocar procesamiento de comandos* parece contar con características de flujo de transformación. Por tanto, deben determinarse límites de flujo para ambos tipos.

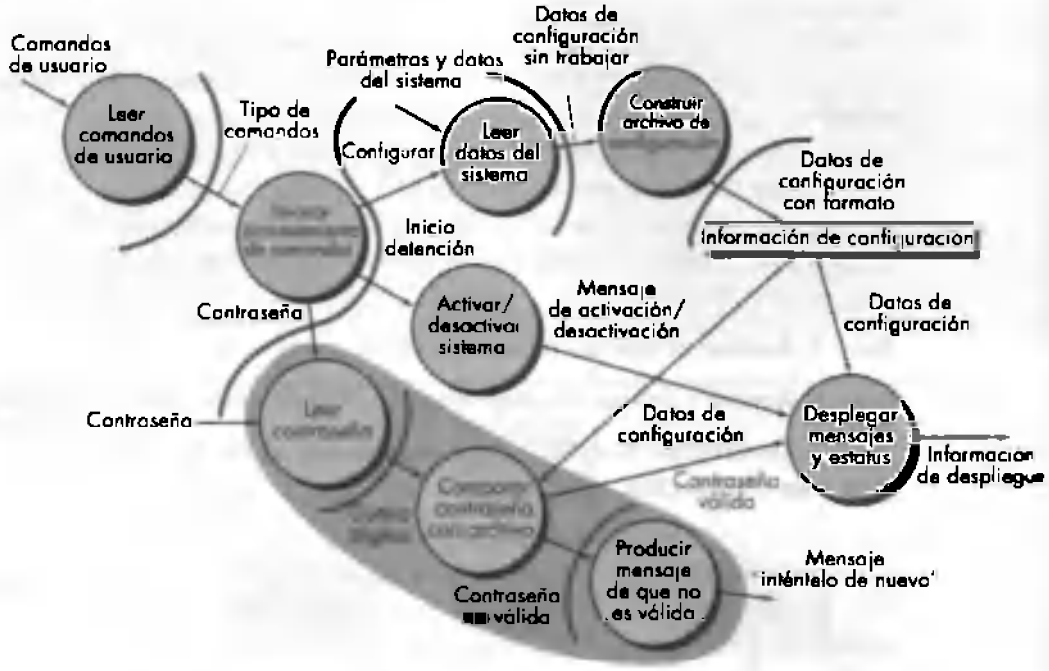
Paso 4. Identificar el centro de transacción y las características de flujo en cada una de las rutas de acción. La ubicación del centro de transacción se desprende directamente del DFD. El centro de transacción se encuentra en el origen de varias rutas de acción que fluyen de él de manera radial. En el caso del flujo mostrado en la figura 10.19, la burbuja *invocar procesamiento de comandos* es el centro de transacción.

PUNTO CLAVE

El camino entrante (es decir, el camino del flujo en que se recibe la transacción) y todas las rutas de acción deben estar aislados. Es necesario evaluar la característica de flujo individual de cada camino de acción. Por ejemplo, el camino "contraseña" (mostrado dentro de un área sombreada en la figura 10.19) tiene características de transformación. El flujo de entrada, de transformación y de salida se indican con límites.

Paso 5. Correlacionar el DFD con una estructura de programa sensible al procesamiento de la transacción. El flujo de transacción se correlaciona con una arquitectura que contiene una rama entrante y una para despacho. La estructura de la rama entrante se desarrolla de manera muy parecida a la correlación de transformaciones. Si se empieza en el centro de transacción, las burbujas ubicadas a lo largo del camino entrante se correlacionan con módulos. La estructura de la rama para despacho contiene un módulo despachador que controla todos los módulos de ac-

FIGURA 10.19 DFD de nivel 2 para el subsistema de interacción con el usuario.

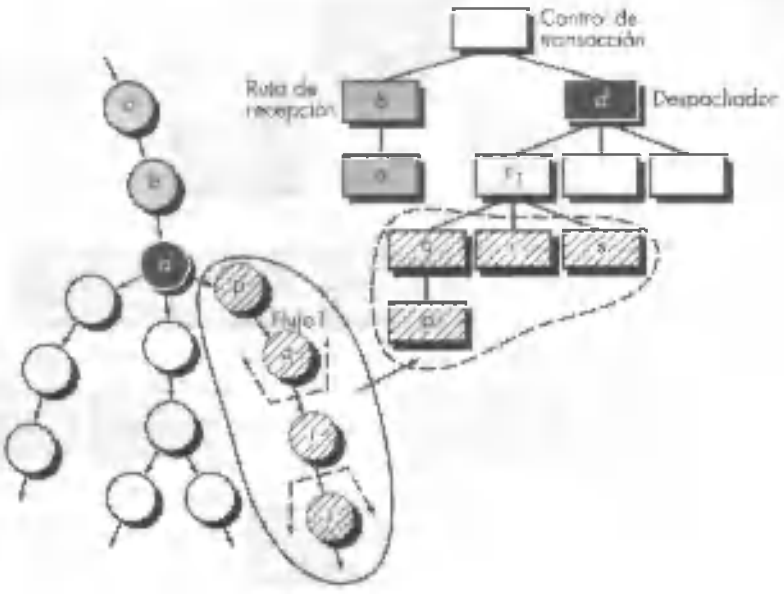


ción subordinados. Cada camino de flujo de acción del DFD se correlaciona con una estructura que corresponde a sus características de flujo específicas. Este proceso se ilustra esquemáticamente en la figura 10.20.

Si se toma en cuenta el flujo de datos del subsistema de interacción con el usuario, la factorización de primer nivel para el paso 5 se muestra en la figura 10.21. Las burbujas *leer comandos del usuario* y *activar/desactivar sistema* se correlacionan directamente con la arquitectura sin necesidad de módulos de control inmediato. El centro de transacción, *invocar comando de procesamiento*, se correlaciona directamente con el módulo despachador del mismo nombre. Se crean controladores para la configuración del sistema y el procesamiento de la contraseña, como se ilustra en la figura 10.21.

Paso 6. Factorizar y refinar la estructura de transacción y la de cada camino de acción. Cada camino de acción del DFD tiene sus propias características de flujo de información. Ya se ha observado que es posible encontrar los flujos de transformación o transacción. La "subestructura" relacionada con el camino de acción se desarrolla empleando los pasos de diseño analizados en esta sección y en la anterior.

Como ejemplo, considérese el flujo de información de procesamiento de la contraseña que se muestra en la figura 10.19 (dentro del área sombreada). El flujo



tra caracteristicas de transformación clásicas. Se ingresa una contraseña (flujo entrante) y se transmite a un centro de transformación donde se compara contra las contraseñas almacenadas. Si no se obtiene una coincidencia, se producen una alarma y un mensaje de precaución (flujo saliente). El camino "configurar" se dibuja de manera similar empleando correlación de transformaciones En la figura 10.22 se muestra la arquitectura del software resultante

Paso 7. Refinar la arquitectura de primera iteración empleando diseño heurístico para mejorar la calidad del software. Este paso para la correlación de transacciones es idéntico al de transformaciones En ambos enfoques de diseño, cri-

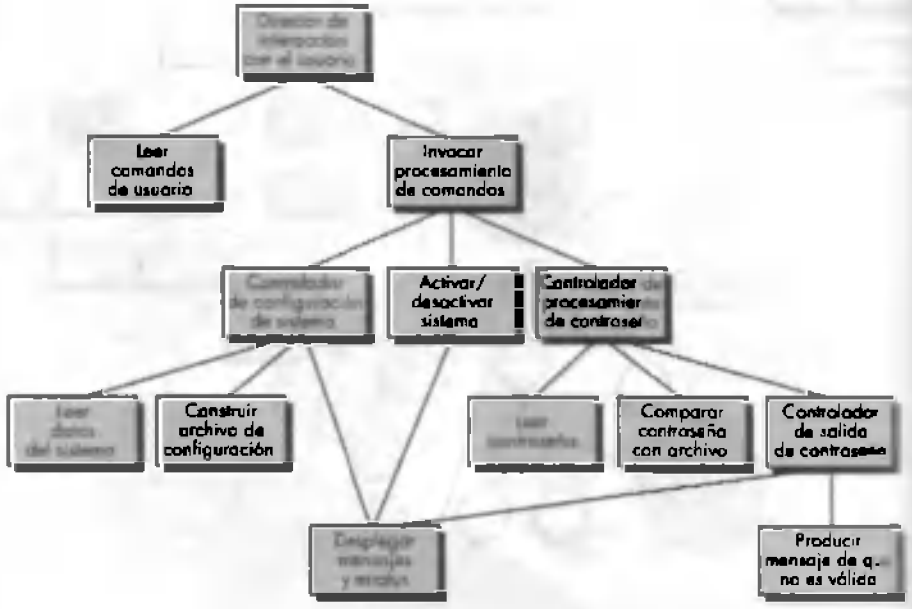


FIGURA 10.22
Arquitectura de primera iteración para el subsistema de interacción con el usuario.

terios como independencia del módulo, factibilidad (eficacia de la implementación la prueba) y facilidad de mantenimiento deben ponderarse cuidadosamente cuando se propongan modificaciones estructurales

> "Hágalo lo más simple posible, pero no más simple de lo necesario"
>
> Albert Einstein

10.6.5 Refinamiento del diseño arquitectónico

Cualquier análisis del refinamiento del diseño debe prologarse con el siguiente comentario: recuerde que un "diseño óptimo" que no funciona tiene un mérito cuestionable. El diseñador de software debe preocuparse por desarrollar una representación del software que cumpla con todos los requisitos funcionales y de desempeño así como la aceptación del mérito basado en las medidas y la heurística del diseño.

Debe estimularse el refinamiento de la arquitectura del software durante las primeras etapas del diseño. Como se analizó al principio de este capítulo, es posible derivar, refinar y evaluar estilos arquitectónicos alternos para determinar cuál es el "mejor" enfoque. Este método para afrontar la optimización es uno de los verdaderos beneficios que se obtienen de desarrollar una representación de la arquitectura del software.

Es importante indicar que a menudo la simplicidad estructural refleja elegancia y eficiencia. El objetivo del refinamiento del diseño debe ser el uso del menor número de componentes que permitan una integración efectiva de los módulos y de la ca

tructura de datos menos compleja que sirva adecuadamente para los requisitos de información.

10.7 Resumen

La arquitectura del software proporciona un concepto holístico del sistema que habrá de construirse. Describe la estructura y la organización de los componentes del software, sus propiedades y la conexión entre ellos. Entre los componentes del software se incluyen los módulos del programa y las diversas representaciones de datos que éste manipula Por tanto, el diseño de datos es una parte integral de la derivación de la arquitectura del software. La arquitectura destaca las decisiones iniciales del diseño y proporciona un mecanismo para considerar los beneficios de estructuras de sistema alternas.

El diseño de datos traduce los objetos de datos (definidos en el modelo de análisis) a estructuras de datos que residen dentro del software Los atributos que describen el objeto, la relación entre los objetos de datos y su utilización dentro del programa influyen en la elección de las estructuras de datos. En un grado más elevado de abstracción, el diseño de datos lleva a la definición de una arquitectura para una base de datos o un almacén de datos.

El ingeniero del software tiene a su disposición varios estilos y patrones arquitectónicos. Cada estilo describe una categoría del sistema que abarca 1) un conjunto de componentes que realizan una función requerida por un sistema, 2) un conjunto de conectores que permiten la comunicación, coordinación y cooperación entre componentes, 3) restricciones que definen la manera en que se integran los componentes para formar el sistema, y 4) modelos semánticos que permiten a un diseñador comprender las propiedades generales de un sistema

En sentido general, el diseño arquitectónico se realiza aplicando varios pasos En primer lugar, el sistema debe estar representado en el contexto; es decir, el diseñador debe definir las entidades externas que interactúan con el software y la naturaleza de la interacción. Una vez que se ha especificado el contexto, el diseñador debe identificar un conjunto de abstracciones de nivel superior, llamadas arquetipos, que representan elementos centrales del comportamiento o la función del sistema; después de que se han definido las abstracciones, el diseño empieza a acercarse al dominio de la implementación. Se identifican los componentes y se representan dentro del contexto de una arquitectura que los soporta. Por último, se crean instancias específicas de la arquitectura para "probar" el diseño en un contexto real.

Como un simple ejemplo de diseño arquitectónico, el método de correlación o mapeo presentado en este capítulo emplea las características del flujo de datos para derivar un estilo arquitectónico de uso común. Un diagrama de flujo de datos se correlaciona con una estructura del programa empleando uno o dos enfoques de correlación (correlación de transformación o de transacción). Una vez que se ha derivado una arquitectura, se elabora ésta y luego se compara contra los criterios de calidad

REFERENCIAS

[AHO83] Aho, A. V., J. Hopcroft y J. Ullmann, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[ALL97] Allen R., "A Formal Approach to Software Architecture", tesis de doctorado, Carnegie Mellon University, Número de reporte técnico CMU-CS-97-144, 1997.

[BAR00] Barroca, L. y P. Hall (eds.), *Software Architecture: Advances and Applications*, Springer-Verlag, 2000.

[BAS03] Bass, L., P. Clements y R. Kazman, *Software Architecture in Practice*, 2a. ed., Addison-Wesley, 2003.

[BOS00] Bosch, J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.

[BUS96] Buschmann, F., *Pattern-Oriented Software Architecture*, Wiley, 1996.

[DAT00] Date, C. J., *An Introduction to Database Systems*, 7a. ed., Addison-Wesley, 2000.

[DIK00] Dikel, D., D. Kane y J. Wilson, *Software Architecture: Organizational Principles and Patterns*, Prentice-Hall, 2000.

[FRE80] Freeman, P., "The Context of Design", en *Software Design Techniques*, 3a. ed. (P. Freeman y A. Wasserman, eds.), IEEE Computer Society Press, 1980, pp. 2-4.

[GAR94] Garlan D., R. Allen y J. Ockerbloom, "Exploiting Style in Architectural Design Environments", en *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering*, 1994.

[GAR00] Garlan D., R. T. Monroe y D. Wile, "Acme: Architectural Description of Component-Based Systems", en *Foundations of Component-Based Systems*, G. T. Leavens y M. Sitaraman (eds.), Cambridge University Press, 2000.

[HOF00] Hofmeister, C., R. Nord y D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.

[HOF01] Hofmann, C. *et al.*, "Approaches to Software Architecture", se descarga de: http://citeseer.nj.nec.com/84015.html.

[KAZ98] Kazman, R. *et al.*, *The Architectural Tradeoff Analysis Method*, Software Engineering Institute, CMU/SEI-98-TR-008, julio de 1998.

[KIM98] Kimball, R., L. Reeves, M. Ross y W. Thornthwaite, *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses*, Wiley, 1998.

[LAN02] Land R., A Brief Survey of Software Architecture, reporte técnico, Departamento de Ingeniería de Cómputo, Universidad Mälardalen, Suecia, febrero de 2002.

[LUC95] Luckham D. C. *et al.*, "Specification and Analysis of System Architecture Using Rapide", en *IEEE Transactions on Software Engineering*, ejemplar "Special Issue on Software Architecture", 1995.

[MAT96] Mattison, R., *Data Warehousing: Strategies, Technologies, and Techniques*, McGraw-Hill, 1996.

[MYE78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978.

[PRE98] Preiss, B. R., *Data Structures and Algorithms: With Object-Oriented Design Patterns in C++*, Wiley, 1998.

[SHA96] Shaw, M. y D. Garlan, *Software Architecture*, Prentice-Hall, 1996.

[SHA97] Shaw, M. y P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", en *Proc. COMPSAC*, Washington, DC, agosto de 1997.

[WAS80] Wasserman, A., "Principles of Systematic Data Design and Implementation", en *Software Design Techniques* (P. Freeman y A. Wasserman, eds.), 3a. ed., IEEE Computer Society Press, 1980, pp. 287-293.

[YOU79] Yourdon, E. y L. Constantine, *Structured Design*, Prentice-Hall, 1979.

[ZHA98] Zhao, J., "On Assessing the Complexity of Software Architectures", en *Proc. Intl. Software Architecture Workshop*, ACM, Orlando, FL, 1998, pp. 163-167.

PROBLEMAS Y PUNTOS A CONSIDERAR

10.1. Empleando la arquitectura de una casa o un edificio como metáfora, realizar comparaciones con la arquitectura del software. ¿En qué son similares las disciplinas de la arquitectura clásica y la del software? ¿En qué se diferencian?

10.2. Escribir un artículo de tres a cinco páginas que presente directrices para seleccionar estructuras de datos basadas en la naturaleza del problema. Empezar delineando las estructuras de datos clásicas encontradas en el trabajo del software y luego describir los criterios para seleccionar, a partir de éstas, tipos particulares de problemas.

10.3. Explicar la diferencia entre una base de datos que sirve a una o más aplicaciones de negocios convencionales y un almacén de datos

10.4. Presentar dos o tres ejemplos de aplicaciones para cada uno de los estilos arquitectónicos indicados en la sección 10.3.1.

10.5. Algunos de los estilos arquitectónico indicados en la sección 10.3.1 son de naturaleza jerárquica, otros no. Elaborar una lista de cada tipo ¿Cómo se implementarían los estilos arquitectónicos que no son jerárquicos?

10.6. Los términos *estilo arquitectónico, patrón arquitectónico* y *marco conceptual* suelen encontrarse en el análisis sobre la arquitectura del software Investigar un poco (utilizar la Web) y describir la diferencia entre cada uno de estos términos y sus contrapartes.

10.7. Seleccionar una aplicación con la que se esté familiarizado Responder cada una de las preguntas planteadas para control y datos en la sección 10.3.3.

10.8. Investigar la MACA (visitar el sitio Web de SEI) y presentar un análisis detallado de los seis pasos presentados en la sección 10 5 1

10.9. Algunos diseñadores sostienen que todos los flujos de datos deben considerarse orientados a la transformación. Analizar la manera en que esta convención afectará la arquitectura del software que se deriva cuando un flujo orientado a la transacción se trata como transformación. Utilizar un flujo de ejemplo para ilustrar puntos importantes

10.10. Si no se ha hecho, completar el problema 8.10. Emplear los métodos de diseño descritos en este capítulo para desarrollar una arquitectura del software para el PHTRS.

10.11. Empleando un diagrama de flujo de datos y una descripción del procesamiento, describir un sistema de cómputo que tenga distintas características de flujo de transformación Definir los límites del flujo y correlacionar el diagrama de flujo de datos con la arquitectura del software empleando la técnica descrita en la sección 10.6.3

10.12. Empleando un diagrama de flujo de datos y una descripción del procesamiento, describir un sistema de cómputo que tenga distintas características de flujo de transacción Defina los límites del flujo y correlacione el diagrama de flujo de datos con la arquitectura del software empleando la técnica descrita en la sección 10 6.4.

Otras lecturas y fuentes de información

La literatura sobre arquitectura de software ha aumentado a lo largo de la década pasada. libros de Fowler (*Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003), Clements y sus colegas (*Documentig Software Architecture: View and Beyond*, Addison-Wesley, 2002), Schmidt y sus colegas (*Pattern-Oriented Software Architectures*, dos volúmenes, Wiley, 2000), Bosch [BOS00], Dikel y sus colegas [DIK00], Hofmeister y sus colegas [HOF00] Bass, Clements y Kazman [BAS03], Shaw y Garlan [SHA96] y Buschmann *et al* [BUS96] proporcionan un análisis a fondo del tema Un trabajo previo de Garlan (*An Introduction to Software Architecture*, Software Engineering Institute, CMU/SEI-94-TR-021,1994) Tiene una excelente introducción Clements y Northrop (*Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001) trata el diseño de arquitecturas que dan soporte a líneas de productos de sofware. Clements y sus colegas (*Evaluating Software Architectures*, Addison-Wesley 2002) considera los temas asociados con la evaluación de alternativas de arquitectura y la selección de la mejor arquitectura para un problema de dominio dado.

Libros específicos sobre implementación de arquitectura tratan el diseño arquitectónico dentro de un ambiente de desarrollo o tecnología especiales. Wallnau y sus colegas (*Building Systems from Commercial Components*, Addison-Wesley, 2001) presentan métodos para construir arquitecturas basadas en componentes. Pritchard (*COM and CORBA Side-by-Side*, Addison-Wesley 1999), Mowbray (*CORBA Design Patterns*, Wiley, 1997) y Mark *et al.* (*Objetet Management Architecture Guide, Wiley, 1996*) provee lineaminttos de diseño detallados para la estructura de soporte CORBA, Shanley (*Protected Mode Software Architecture*, Addison-Wesley 1996) proporciona asesoría sobre diseño arquitectónico para cualquier sistema operativo basado en tiempo real para PC, sistemas operativos multipropósito o *drivers*.

La investigación actual sobre arquitectura de software se documenta anualmente en los *Proceedings of the International Workshop en Software Architecture*, patrocinados por la ACM y otras organizaciones de computación, y los *Proceedings of the International Conference on Software Engineering*. Barroca y Hall [BAR00] presentan un útil estudio de investigación reciente.

El modelado de datos es un requisito para un buen diseño en esta materia. Los libros de Teory (*Database Modeling and Design*, Academic Press, 1998); Schmidt (*Data Modeling for Information Professionals*, Prentice-Hall, 1998); Bobak (*Data Modeling and Design for Today's Architectures*, Artech House, 1997); Silverstone, Graziano e Inmon (*The Data Model Resource Book*, Wiley, 1997); Date [DAT00], y Reingruber y Gregory (*The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models*, Wiley, 1994) contiene presentaciones detalladas sobre notación de modelado de datos, heurístic y aspectos del diseño de bases de datos. El diseño de almacenes de datos se ha vuelto más importante en los últimos años. Los libros de Humphreys, Hawkins y Dy (*Data Warehousing. Architecture and Implementation*, Prentice-Hall, 1999); Kimball *et al.* [KIM98] e Inmon [INM95] tratan el tópico con mucho detalle.

El estudio general del diseño de software con discución de aspectos de arquitectura y diseño de datos puede encontrarse en la mayoría de los libros dedicados a la ingeniería de software. Tratamientos más rigurosos del tema se hallan en Feijs (*A Formalization of Design Methods*, Prentice-Hall, 1993) Witt *et al.* (*Software Architecture and Design Principles*, Thomson Publishing, 1994) y Budgen (*Software Design*, Addisson-Wesley, 1994).

Presentaciones completas de diseño orientado al flujo de datos pueden encontrarse en Myers [MYE78], Yourdon y Constantine [YOU79], y Page-Jones (*The Practical Guide to Structured Systems Design*, 2a. ed. Prentice Hall, 1998). Estos libros están dedicados sólo al diseño se incluyen extensos análisis del flujo de datos.

Una amplia variedad de fuentes de información sobre el diseño arquitectónico están disponibles en Internet. Una lista actualizada de referencias en la World Wide Web que son relevantes para el diseño arquitectónico puede encontrarse en el sitio Web de SEPA: **http://www.mhhe.com/pressman.**

CAPÍTULO

11

DISEÑO AL NIVEL DE COMPONENTES

El diseño al nivel de componentes se presenta después de que se ha completado la primera iteración del diseño arquitectónico. En esta etapa ya se han establecido los datos generales y la estructura del programa. El objetivo es traducir el modelo de diseño en un software operacional. Pero el grado de abstracción del modelo de diseño existente es relativamente elevado, y el del programa operacional, bajo. La traducción llega a ser desafiante, abriendo la puerta para el ingreso de errores sutiles que resultan difíciles de encontrar y corregir en etapas posteriores del proceso de software. En una famosa conferencia, Edsgar Dijkstra, una de las personas que más ha contribuido a nuestra comprensión del diseño de software, afirmó [DIJ72]:

> Al parecer, la diferencia entre el software y muchos otros productos es que en éstos, como regla general, mayor calidad representa precio más elevado. Quienes desean software realmente confiable descubrirán que deben encontrar un medio para evitar la mayor parte de los errores desde el principio y como resultado, el proceso de programación se volverá mucho más económico... los buenos programadores... no deben desperdiciar su tiempo depurando, deben evitar los errores desde el principio.

Aunque estas palabras fueron pronunciadas hace muchos años, aún son válidas. Cuando el modelo de diseño se traduce en código fuente deben seguirse una serie de principios de diseño que no sólo realicen la traducción, sino que "eviten la introducción de errores desde el principio".

Es posible representar el diseño al nivel de componentes empleando un lenguaje de programación. En esencia, el programa se crea con el modelo de diseño arquitectónico como guía. Un enfoque alterno consiste en representar el diseño al nivel de componentes empleando alguna representación intermedia (por ejemplo, gráfica, tabular o basada en texto) que se traduzca fácilmente en código fuente. Independientemente del mecanismo con que se represente el diseño al nivel de componentes, las estructuras de datos, las interfaces y los algoritmos definidos deben adecuarse a diversas líneas generales de diseño bien definidas, que ayuden a evitar errores a medida que evoluciona el diseño procedimental. En este capítulo se examinaran esas líneas generales y los métodos disponibles para seguirlas.

UN VISTAZO RÁPIDO

¿Qué es? Un conjunto completo de componentes de software se define durante el diseño arquitectónico, pero las estructuras de datos internas y el procesamiento de detalles de cada componente no se representan en un grado de abstracción parecido al código. El diseño al nivel de componentes define las estructuras de da-

tos, los algoritmos, las características de la interfaz y los mecanismos de comunicación asignados a cada componente de software

¿Quién lo hace? Un ingeniero de software realiza el diseño al nivel de componentes.

¿Por qué es importante? Antes de construir el software se debe tener la capacidad de determinar si funcionará bien. El diseño al nivel de componentes representa el software de tal manera que permite revisar si los detalles del diseño son correctos y consistentes con las representaciones iniciales del diseño (es decir, los diseños de datos, arquitectura e interfaz). Proporciona una manera de evaluar si funcionarán las estructuras, las interfaces y los algoritmos.

¿Cuáles son los pasos? Las representaciones al nivel de diseños de datos, arquitectura e interfaz representan la base del diseño al nivel de componentes. La definición de clase o la narrativa de procesamiento de cada componente se traduce en un diseño detallado que usa diagramas o formas de texto que especifican estructuras de datos internas, detalle de la interfaz local y lógica de procesamiento. La notación de diseño abarca diagramas UML y representaciones complementarias. El diseño procedimental se especifica mediante un conjunto de construcciones de programación estructurada.

¿Cuál es el producto obtenido? El diseño de cada componente, representado en una notación gráfica, tabular o textual, es el principal producto de trabajo creado durante el diseño a nivel de componentes.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Se realiza un recorrido o una inspección del diseño. Éste se examina para determinar si las estructuras de datos, las interfaces, las secuencias y condiciones lógicas son correctas y para ver si arrojan los datos apropiados o la transformación de control asignada al componente durante las primeras etapas del diseño.

11.1 ¿QUÉ ES UN COMPONENTE?

De manera general, un componente es un bloque de construcción modular para el software de cómputo. De manera más formal, la especificación unificada de lenguaje de modelado de OMG [OMG01] define un componente como "una parte modular, desplegable y reemplazable de un sistema que encapsula implementación y expone un conjunto de interfaces".

Como se analizó en el capítulo 10, los componentes pueblan la arquitectura del software y, por tanto, ayudan a cumplir con los objetivos y requisitos del sistema a construcción. Debido a que los componentes residen en el interior de la arquitectura del software, deben comunicarse y colaborar con otros componentes y con entidades (como otros sistemas, dispositivos, personas) que existen fuera de los límites del software.

> "Los detalles no son sólo detalles. Integran el diseño."
>
> **Charles Eames**

El verdadero significado del término "componente" variará dependiendo del punto de vista del ingeniero de software que lo usa. En la siguiente sección se revisarán tres conceptos importantes de lo que es un componente y la manera en que se modela a medida que se realiza el modelado del diseño.

11.1.1 Concepto orientado a objetos

En el contexto de la ingeniería del software orientada a objetos, un componente contiene un conjunto de clases que colaboran entre sí.[1] Cada clase de un componente se ha elaborado completamente para incluir todos los atributos y las operaciones relevantes para su implementación. Como parte de la elaboración del diseño, también deben definirse todas las interfaces (mensajes) que permiten que las clases se comuniquen y colaboren con otras clases de diseño. Para lograrlo, el diseñador empieza con el modelo de análisis y elabora clases de análisis (en el caso de componentes que se relacionan con el dominio del problema) y clases de infraestructura (en el caso de componentes que proporcionan servicios de soporte para el dominio del problema).

Este proceso de elaboración del diseño se ilustra imaginando que el software se construirá para una imprenta sofisticada. El objetivo general del software es recopilar las necesidades del cliente en el mostrador, cotizar un trabajo de impresión y pasarlo a una planta de producción automatizada. Durante la ingeniería de los requisitos se deriva una clase de análisis denominada **TrabajoImprenta**. Los atributos y las operaciones definidos durante el análisis se observan en la parte superior izquierda de la figura 11.1. Durante el diseño arquitectónico se define **TrabajoImprenta** como un componente de la arquitectura de software y se representa empleando la notación abreviada de UML que se muestra en la parte central derecha de la figura. Observe que **TrabajoImprenta** tiene dos interfaces, *calcularTrabajo*, que proporciona la capacidad de cotizar el trabajo, e *iniciarTrabajo*, que pasa el trabajo a la planta de producción. Éstas se representan empleando los símbolos que se muestran a la izquierda del recuadro del componente.

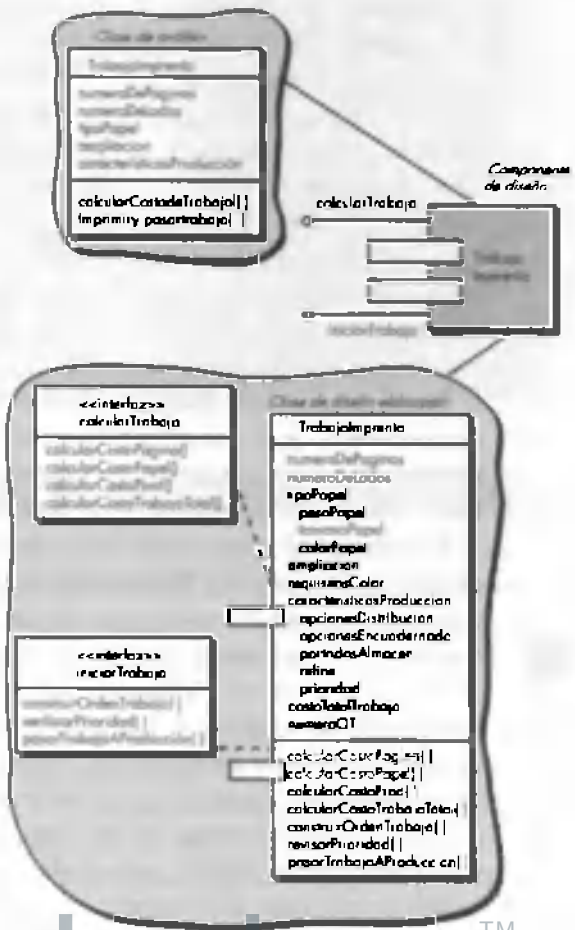
El diseño al nivel de componentes empieza en este punto. Deben elaborarse los detalles del componente **TrabajoImprenta** para que proporcionen la información suficiente que guíe la implementación. La clase de análisis original se elabora para dar forma a todos los atributos y las operaciones requeridos para implementar la clase como el componente **TrabajoImprenta**. Tomando como referencia la parte inferior derecha de la figura 11.1, la clase de diseño **TrabajoImprenta** elaborada contiene información de atributos más detallada, además de una descripción expandida de las operaciones requeridas para implementar el componente. Las interfaces *calcularTrabajo* e *iniciarTrabajo* llevan implícitas la comunicación y colaboración con otros componentes (que no se muestran aquí). Por ejemplo, la operación *calcularCostoPagina( )* (parte de la interfaz *calcularTrabajo*) colaboraría con un componente **TablaPrecios** que contiene la información de precios de los trabajos. La operación *verificarPrioridad( )* (parte de la interfaz *iniciarTrabajo*) colaboraría con el componente **ColaTrabajos** para determinar los tipos y las prioridades de los trabajos en espera (o en cola) que se encuentran en producción.

Esta actividad de elaboración se aplica a cada componente definido como parte del diseño arquitectónico. Una vez completado, se elabora aún más cada atributo,

1 En algunos casos un componente podría contener una sola clase.

operación e interfaz. Deben especificarse las estructuras de datos apropiadas para cada atributo. Además, también se diseña el detalle algorítmico que se requiere para implementar la lógica de procesamiento asociada con cada operación. Esta actividad de diseño procedimental se analizará más adelante, en este mismo capítulo. Por último, se diseñan los mecanismos necesarios para implementar la interfaz. En el caso del software orientado a objetos, esto abarca la descripción de todos los mensajes que se requieren para realizar la comunicación entre objetos dentro del sistema.

FIGURA 11.1 Elaboración de un componente de diseño.



11.1.2 El concepto convencional

En el contexto de la ingeniería del software convencional, un componente es un elemento funcional de un programa que incorpora la lógica del procesamiento, las estructuras internas de los datos necesarios para implementar dicha lógica, y una interfaz que permita la invocación del componente y el paso de los datos. Un componente convencional, también denominado módulo, reside dentro de la arquitectura

del software y representa uno de tres papeles importantes: 1) como *componente de control* que coordina la invocación de todos los demás componentes del dominio del problema, 2) como *componente del dominio* del problema que implementa una función completa o parcial requerida por el cliente, o 3) como *componente de infraestructura* responsable de funciones que soportan el procesamiento requerido en el dominio del problema.

Como los componentes orientados a objetos, los componentes del software convencional se derivan del modelo de análisis. Sin embargo, en este caso el elemento de datos orientado al flujo del modelo de análisis sirve como base para la derivación. Cada transformación (burbuja) representada en los niveles inferiores del diagrama de flujo de datos (capítulo 8) se correlaciona directamente (sección 10.6) con una jerarquía de módulos. Los componentes de control (módulos) residen cerca de la parte superior de la jerarquía (arquitectura) y los componentes del dominio del problema tienden a residir hacia la parte inferior de la jerarquía. Para lograr una modularidad efectiva, se aplican conceptos de diseño como la independencia funcional (capítulo 9) a medida que se elaboran los componentes.

> "Invariablemente se descubre que un sistema complejo que funciona ha evolucionado a partir de un sistema simple que también funcionaba."
>
> **John Gall**

Este proceso de elaboración del diseño de componentes convencionales se ilustra considerando de nuevo el software que se habrá de construir para un sofisticado centro de fotocopiado. Un conjunto de diagramas de flujo de datos se derivaría durante el modelado del análisis. Se supondrá que éstos se correlacionan (sección 10.6) dentro de la arquitectura que se muestra en la figura 11.2. Cada recuadro representa un componente de software. Tómese en cuenta que los recuadros con pantalla gris tienen una función equivalente a las operaciones definidas en la clase **TrabajoImprenta** analizada en la sección 11.1.1. Sin embargo, en este caso cada operación se representa como un módulo separado que se invoca como se muestra en la figura. Con otros módulos se controla el procesamiento y, por tanto, son componentes de control.

CONSEJO

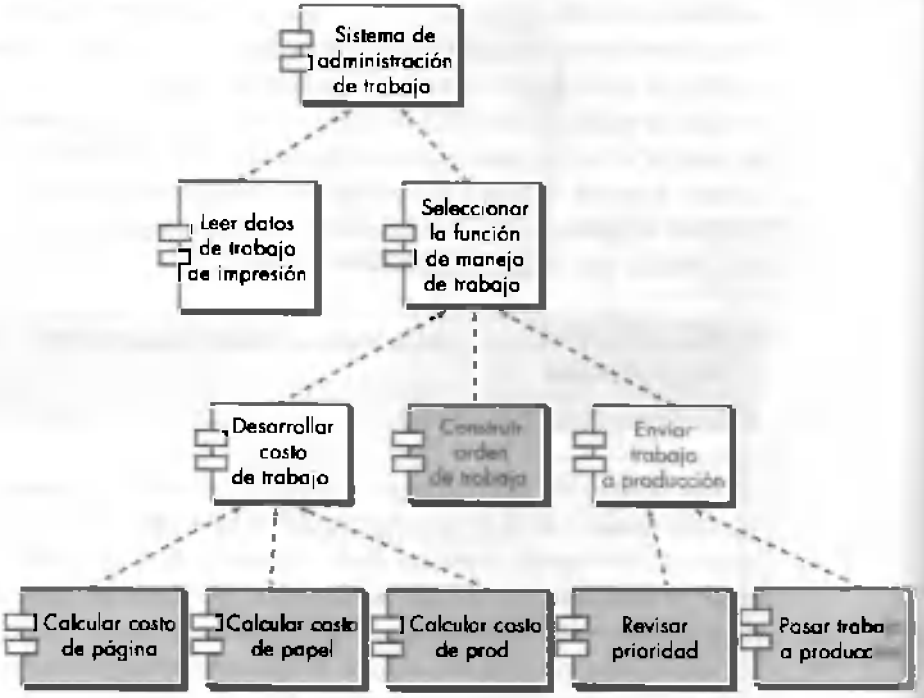
...se elabora el ... cada ...nte de ..., el punto ...desplaza al ...de estructuras ...específicas y ...procedimen... ...manipular los ...s de datos. ...o, no debe ...que la ...o que debe ...r los ...s o la ...global de ...e servir a ...s componentes.

Durante el diseño al nivel de componentes, se elabora cada módulo mostrado en la figura 11.2. La interfaz del módulo se define de manera explícita. Es decir, se representa cada objeto de datos o de control que fluye por la interfaz. El algoritmo que permite que el módulo realice su función está diseñado con el enfoque de refinamiento paso a paso analizado en el capítulo 9. El comportamiento del módulo suele representarse con un diagrama de estado.

Para ilustrar este proceso considérese el módulo *CalcularCostoPagina*. Su objetivo es calcular el costo de impresión por página a partir de las especificaciones del cliente. Los datos necesarios para realizar esta función son: **número de páginas en el documento, número total de documentos que se producirán, impresión por una o ambas caras, color**

o blanco y negro y tamaño. Estos datos se pasan a *CalcularCostoPagina* mediante la terfaz del módulo Éste usa los datos y determina un costo por página que se basa en el tamaño y la complejidad del trabajo (una función de todos los datos pasados a módulo con la interfaz). El costo por página es inversamente proporcional al tamaño del trabajo y directamente proporcional a su complejidad.

FIGURA 11.2
Gráfica de estructura de un sistema convencional.



En la figura 11.3 se representa el diseño al nivel de componentes empleando una notación UML modificada El módulo *CalcularCostoPagina* tiene acceso a los datos a invocar al módulo *obtenerDatosTrabajo*, que permite el paso de todos los datos relevantes al componente, y una interfaz de base de datos, *accesarBDCostos*, que permite que el módulo tenga acceso a una base de datos con todos los costos de impresión. A medida que prosigue el diseño se elabora el módulo *CalcularCostoPagina* para proporcionar el detalle del algoritmo y la interfase (figura 11.3). El detalle del algoritmo se representa empleando el texto de seudocódigo que se muestra en la figura con un diagrama de actividad de UML. Las interfaces se representan como una colección de objetos o elementos de datos de entrada y salida La elaboración del diseño continúa hasta que se proporcione detalle suficiente para guiar la construcción del componente.

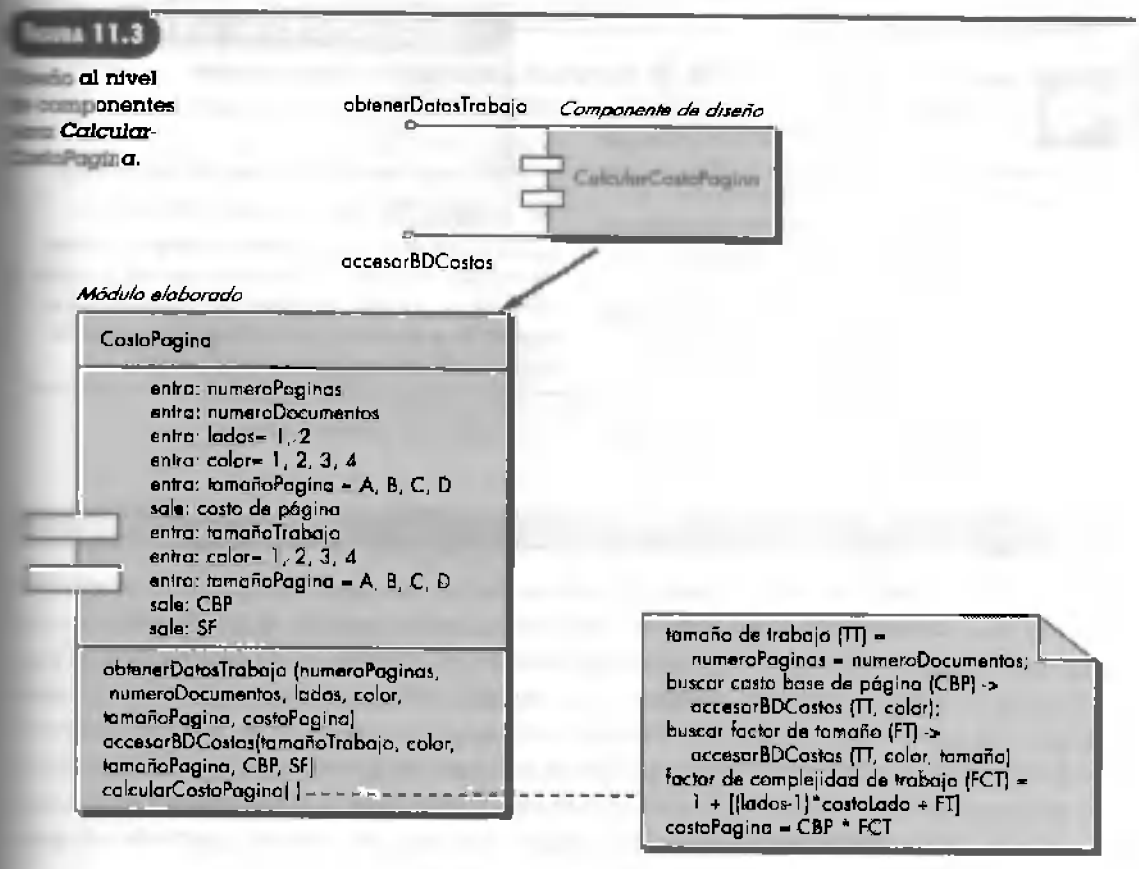


FIGURA 11.3 Diseño al nivel de componentes para *CalcularCostoPagina*.

11.1.3 Un concepto relacionado con el proceso

En los conceptos orientado a objetos y convencional del diseño al nivel de componentes presentados en las secciones anteriores, se supone que los componentes se han diseñado desde cero. Es decir, que el diseñador debe crear un nuevo componente basado en especificaciones derivadas del modelo de análisis. Hay, por supuesto, otro enfoque.

En la década pasada, la comunidad de la ingeniería del software ha destacado la necesidad de construir sistemas que usen los componentes de software existentes. En esencia, un catálogo de componentes probados al nivel de diseño o de código queda a disposición del ingeniero de software a medida que avanza en el trabajo de diseño. Mientras se desarrolla la arquitectura del software, se eligen del catálogo los componentes o patrones de diseño y se usan para poblar la arquitectura. Debido a que estos componentes se han creado con la reutilización en mente, se encuentra a disposición del diseñador una descripción completa de su interfaz, la función o las funciones que realiza y la comunicación y colaboración que requiere. La ingeniería del software basada en componentes se analizará de manera muy detallada en el capítulo 30.

HERRAMIENTAS DE SOFTWARE

Middleware e ingeniería de software basada en componentes

Uno de los elementos clave que lleva al éxito o el fracaso de la ingeniería del software basada en componentes es la disponibilidad de middleware. Éste es una colección de componentes de infraestructura que permiten que los componentes del dominio del problema se comuniquen con otros en una red o dentro de un sistema complejo. Quienes desean usar ingeniería del software basada en componentes a medida que avanza el proceso de software cuentan con tres estándares competidores:

OMG CORBA (http://www.corba.org/)

Microsoft COM
(http://www.microsoft.com/com/tech/complus.asp)

Sun JavaBeans (http://java.sun.com/products/ejb/).

Los anteriores sitios Web presentan una amplia variedad de tutoriales, manuales, herramientas y recursos generales sobre estos importantes estándares de middleware. En el capítulo 30 se encontrará más información acerca de la ingeniería del software basada en componentes.

11.2 DISEÑO DE COMPONENTES BASADOS EN CLASES

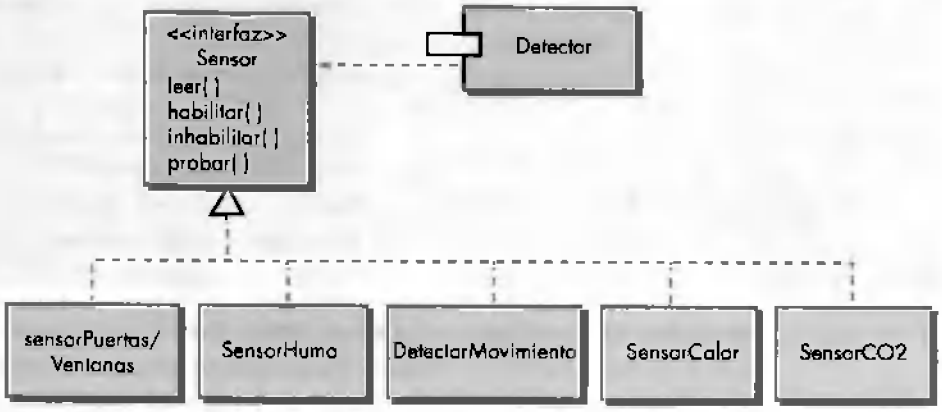
Como ya se ha observado, el diseño al nivel de componentes se basa en información desarrollada como parte del modelo de análisis (capítulo 8) y representada como parte del modelo arquitectónico (capítulo 10). Cuando se elige un método de ingeniería de *software basado en componentes, el diseño al nivel de éstos se concentra* en la elaboración de las clases de análisis (clases específicas del dominio del problema), y la definición y la afinación de las clases de infraestructura. La descripción detallada de los atributos, las operaciones y las interfaces empleados por estas clases representa el detalle de diseño requerido como precursor de la actividad de construcción.

11.2.1 Principios básicos de diseño

Hay cuatro principios básicos de diseño aplicables al diseño al nivel de componentes y se han adoptado ampliamente cuando se aplica ingeniería de software orientada a objetos. La motivación elemental para la aplicación de estos principios es crear diseños que sean más sensibles al cambio y reducir la propagación de efectos secundarios cuando ocurren cambios. Estos principios pueden usarse para guiar al diseñador a medida que desarrolla un componente de software.

El principio abierto-cerrado (PAC). *"[El componente de] un módulo debe estar abierto para extensión, pero cerrado para modificación"* [MAR00]. Esta frase parece una contradicción, pero representa una de las características más importantes de un buen diseño al nivel de componentes. Para expresarlo de manera simple, el diseñador debe especificar el componente de manera que permita extenderlo (dentro del dominio funcional que atiende) sin necesidad de modificaciones internas al propio componente (al nivel de código o lógica). Para ello, el diseñador crea abstracciones que sirven como memoria intermedia entre la funcionalidad que tal vez habrá de extenderse y la propia clase de diseño.

Figura 11.4
...amiento
...PAC



Por ejemplo, suponga que la función de seguridad *HogarSeguro* usa la clase **Detector** que debe revisar el estatus de cada tipo de sensor de seguridad. Es probable que con el tiempo aumenten el número y los tipos de sensores de seguridad. Si la lógica de procesamiento interno está implementada como una secuencia de construcciones si-entonces-si_no (if-then-else), donde cada una atiende un tipo de sensor diferente, la adición de un nuevo tipo de sensor requerirá lógica de procesamiento interno adicional (un si-entonces-si_no adicional). Esto es una violación del PAC.

Una manera de cumplir con el PAC en el caso de la clase **Detector** se ilustra en la figura 11.4. La interfaz *sensor* presenta una vista consistente de sensores para el componente **Detector.** Si se agrega un nuevo tipo de sensor no se requieren cambios en la clase **Detector** (componente). Se preserva el PAC.

HOGARSEGURO

El PAC en acción

La escena: Cubículo de Vinod

Los actores: Vinod y Shakira, integrantes del equipo de ingeniería del software *HogarSeguro*

La conversación:

Vinod: Acabo de recibir una llamada de Doug (el ge... del equipo). Dice que marketing quiere agregar un ... sensor.

Shakira (sonriendo): ¡Otra vez!

Vinod: Sí... y no vas a creer con lo que han salido aho...

Shakira: Sorpréndeme.

Vinod (riendo): Lo llamaron sensor detector de ladridos.

Shakira: ¿Qué significa?

Vinod: Es para la gente que deja sus mascotas en departamentos o condominios o casas muy cercanas. El perro empieza a ladrar. El vecino se enoja y se queja. Con este sensor, si el perro ladra durante más de un minuto, por decir algo, el sensor detona una alarma especial que llama al teléfono celular del dueño.

Shakira: No bromees.

Vinod: Es en serio. Doug quiere saber cuánto tiempo nos tomará agregarlo a la función de seguridad.

Shakira (pensando por un momento): No mucho... mira [le muestra a Vinod la figura 11.4]. Hemos aislado las clases de sensores reales tras la interfaz sensor. Siempre y cuando tengamos especificaciones del sensor de perros, se agrega en un tris. Lo único que ten-

go que hacer es crear un componente apropiado... a sea, una clase, para él. No hay ninguna necesidad de cambiar el componente Detector.

Vinod: Entonces le puedo decir a Doug que no hay mucho problema.

Shakira: Conociendo a Doug, nos tendrá ocupados y no enviará la cosa esa contra perros hasta la próxima versión

Vinod: No está mal, pero ¿lo podrías implementar ahora mismo si él lo quisiera?

Shakira: Sí, la manera en que diseñamos la interfaz nos permite hacerlo sin mucho esfuerzo.

Vinod (pensando por un momento): ¿Alguna vez has oído hablar del "Principio Abierto-Cerrado"?

Shakira (encogiendo los hombros): Nunca

Vinod (sonriendo): No importa

Principio de sustitución de Liskov (PSL). *"Debe tenerse la opción de sustituir subclases con sus clases principales."* [MAR00] Este principio del diseño, que originalmente propuso Barbara Liskov [LIS88], sugiere que un componente que use una clase principal debe seguir funcionando apropiadamente si, en cambio, se pasa al componente una clase derivada. El PSL exige que cualquier clase derivada de una clase principal se apegue a cualquier contrato implícito entre la clase principal y los componentes que la usan. En el contexto de esta explicación, un "contrato" es una *condición previa* que debe ser verdad después de que el componente usa una clase principal y una *condición posterior* que debe ser cierta después de que el componente usa una clase principal. Cuando un diseñador crea clases *derivadas*, también deben apegarse a las condiciones previas y posteriores.

Principio de inversión de la dependencia (PID). *"Dependa de las abstracciones, no de las concreciones."* [MAR00] Como hemos visto en el análisis del PAC, las abstracciones son el lugar donde un diseño se puede extender sin grandes complicaciones. Cuanto más dependa un componente de otros componentes concretos (en lugar de abstractos, como la interfaz), más difícil será extenderlos.

Si se prescinde del diseño y se pasa directamente al código, sólo recuérdese que éste es la "concreción" final. Así que se estará violando el PID

Principio de segregación de la interfaz (PSI). *"Es mejor tener muchas interfaces específicas del cliente que una interfaz de propósito general."* [MAR00] Hay muchos casos en que varios componentes de cliente usan las operaciones proporcionadas por una clase de servidor. El PSI sugiere que el diseñador debe crear una interfaz especializada para servir a cada categoría importante del cliente. Sólo las operaciones importantes para una categoría especial de clientes deben especificarse en la interfaz para esos clientes. Si varios clientes necesitan las mismas operaciones, deben especificarse en cada una de las interfaces especializadas.

Por ejemplo, piense en la clase **PlanoCasa** que se usa en *HogarSeguro* para funciones de seguridad y vigilancia. En el caso de las funciones de seguridad, **PlanoCasa** sólo se emplea durante las actividades de configuración y utiliza las operaciones *colocarDispositivo( )*, *mostrarDispositivo( )*, *agruparDispositivo( )* y *eliminarDispositivo* para colocar, mostrar, agrupar y eliminar sensores del plano. La función de vigilancia de *HogarSeguro* usa las cuatro operaciones indicadas para seguridad, pero sólo requiere operaciones espaciales para manejar las cámaras: *mostrarPV( )* y *mostrarIDC*

positivo( ). Por tanto, el PSI sugiere que los componentes de cliente de las dos funciones de *HogarSeguro* tengan interfaces especializadas y definidas para ellas. La interfaz de seguridad sólo abarcaría las operaciones *colocarDispositivo( )*, *mostrarDispositivo( )*, *agruparDispositivo( )* y *eliminarDispositivo( )*. La interfaz de vigilancia incorporaría las cuatro operaciones anteriores, además de *mostrarPV( )* y *mostrarIDDispositivo( )*

Aunque los principios de diseño al nivel de componentes proporcionan una guía útil, los propios componentes no existen en el vacío. En muchos casos, los componentes o las clases individuales se organizan en subsistemas o paquetes Es razonable preguntar, ¿cómo debe presentarse esta actividad de empaquetamiento? Exactamente, ¿cómo deben organizarse los componentes a medida que avanza el diseño? Martin [MAR00] sugiere principios adicionales de empaquetamiento que son aplicables al diseño al nivel de componentes.

Punto CLAVE
...componente reutilizable más que ...diseño técnico requiere ...ción efectivos mecanismos ...capítulo

Principio de equivalencia entre reutilización y versión (PER). *"La esencia de la reutilización es la misma que de la versión."* [MAR00] Cuando las clases o componentes se diseñan para reutilizarlos, hay un contrato explícito entre el desarrollador de la entidad reutilizable y la persona que la usará. El desarrollador se compromete a establecer un sistema de control de versiones que dé soporte y mantenga las versiones anteriores de la entidad mientras los usuarios actualizan lentamente la versión más actual En lugar de atender cada clase individualmente, lo aconsejable sería agrupar las clases reutilizables en paquetes que pueden manejarse y controlarse a medida que evolucionan nuevas versiones

Principio del cierre común (PCC). *"Las clases que cambian juntas deben permanecer juntas "* [MAR00] Las clases deben empaquetarse de manera que sean un todo coherente Es decir, cuando las clases se empaquetan como parte de un diseño, deben atender la misma área de funciones o comportamientos. Cuando alguna característica de esa área debe cambiar, es probable que sólo deban modificarse las clases del paquete. Esto lleva a un control de cambios y a un manejo de las versiones más efectivos.

Principio común de la reutilización (PCR). *"Las clases que no se reutilizan juntas no deben agruparse juntas."* [MAR00] Cuando una o más clases cambian en un paquete, también cambia el número de versión del paquete. Todas las demás clases o los demás paquetes que dependen de ese paquete deben actualizarse ahora a la versión más reciente del paquete y probarse para asegurar que la nueva versión funciona sin incidentes. Si no hubo cohesión al agrupar las clases, es posible que cambie una clase que no tenga relación con las demás. Esto requerirá un proceso innecesario de integración y prueba. Por ello, sólo deben incluirse en un mismo paquete las clases que se reutilizarán juntas

11.2.2 Líneas generales de diseño al nivel de componentes

Además de los principios analizados en la sección 11.2.1, es posible aplicar un conjunto pragmático de líneas generales de diseño a medida que avanza el diseño al ni-

vel de componentes. Estas líneas generales se aplican a componentes, sus interfaces y las características de dependencia y herencias que impactan el diseño resultante. Ambler [AMB02] sugiere las siguientes líneas generales:

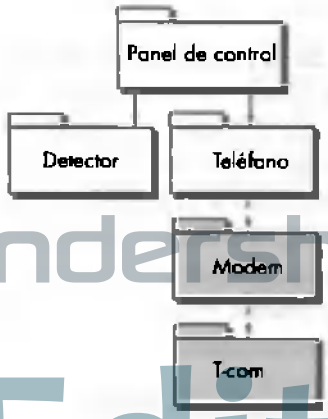
¿Qué se debe tomar en cuenta cuando se nombran los componentes?

Componentes. Deben definirse convenciones de asignación de nombres para componentes especificados como parte del modelo arquitectónico, y luego refinarse y elaborarse como parte del diseño al nivel de componentes. Los nombres del modelo arquitectónico deben extraerse del dominio del problema y tener algún significado para los participantes que ven el modelo arquitectónico. Por ejemplo, el nombre de clase **PlanoCasa** tiene significado para quienes lo leen, sin importar sus antecedentes técnicos. Por otra parte, los componentes de infraestructura o las clases elaboradas al nivel de componentes deben tener un nombre que refleje el significado específico de la implementación. Si se habrá de manejar una lista vinculada como parte de la implementación de **PlanoCasa**, la operación *manejarLista( )* resulta apropiada, aunque una persona sin conocimientos técnicos podría malinterpretarla.[2]

También vale la pena usar estereotipos para ayudar a identificar la naturaleza de los componentes al nivel de diseño detallado. Por ejemplo, <<infraestructura>> podría usarse para identificar un componente de infraestructura; <<basededatos>> podría usarse para identificar una base de datos que sirve a una o más clases del diseño o a todo el sistema; <<tabla>> se usaría para identificar una tabla dentro de una base de datos.

Interfaces. Las interfaces proporcionan información importante acerca de la comunicación y la colaboración (además de ayudar a lograr el PAC). Sin embargo, una representación libre de las interfaces tiende a complicar los diagramas del componente. Ambler [AMB02] recomienda que 1) cuando los diagramas se vuelvan más complejos se use la representación de línea y círculo para una interfaz, en lugar

FIGURA 11.5
Cohesión de capa.



2 Es improbable que una persona de marcadotecnia o de la organización del cliente (un tipo sin antecedentes técnicos) examine el detalle de la información de diseño

enfoque más formal del recuadro UML y la flecha con línea de guiones; 2) por razones de consistencia, las interfaces deben fluir desde la izquierda del recuadro del componente; 3) sólo deben mostrarse las interfaces relevantes del componente en cuestión, aunque estén disponibles otras Estas recomendaciones pretenden simplificar la naturaleza visual de los diagramas de componentes UML.

Dependencias y herencia. Para mejorar la legibilidad es buena idea modelar las dependencias de izquierda a derecha y la herencia de abajo (clases derivadas) hacia arriba (clases principales). Además, las interdependencias entre los componentes deben representarse mediante interfaces, en lugar de hacerlo mediante la representación de una dependencia de componente a componente Siguiendo la filosofía del PAC, esto ayudará a mejorar las opciones de mantenimiento del sistema.

11.2.3 Cohesión

En el capítulo 9 se describió la cohesión como la "función única" de un componente. En el contexto del diseño al nivel de componentes para sistemas orientados a objetos, la *cohesión* implica que un componente o una clase sólo encapsula atributos y operaciones relacionadas estrechamente entre sí y con la clase del propio componente. Lethbridge y Laganiére [LET01] definen varios tipos diferentes de cohesión (que aparecen en la lista ordenados según su grado de cohesión):[3]

Funcional. Exhibido principalmente para operaciones, este grado de cohesión se presenta cuando un módulo realiza un solo cálculo y luego devuelve un resultado

De capa. Exhibido para paquetes, componentes y clases, este tipo de cohesión ocurre cuando una capa superior tiene acceso a los servicios de una inferior, pero ésta no tiene acceso a aquélla. Piénsese, por ejemplo, en la necesidad de que la función de seguridad de *HogarSeguro* haga una llamada telefónica al exterior si se dispara una alarma. Sería posible definir un conjunto de paquetes en capas como se muestra en la figura 11.5 Los paquetes con pantalla gris contienen componentes de infraestructura. El acceso se tiene del paquete del panel de control hacia abajo.

De comunicación. Todas las operaciones con acceso a los mismos datos se definen dentro de una clase. En general, esa clase sólo se concentra en los datos en cuestión, accesándolos y almacenándolos

Resulta relativamente fácil implementar, probar y mantener las clases y los componentes que muestran cohesión funcional, de capa y de comunicación. El diseñador debe luchar por alcanzar estos grados de cohesión. Sin embargo, hay muchos casos en que se encuentran los siguientes niveles inferiores de cohesión:

Secuencial. Los componentes o las operaciones están agrupados de manera que el primero permita la entrada al siguiente, y así sucesivamente. El objetivo es implementar una secuencia de operaciones

3 En general, mientras mayor sea el grado de cohesión, más fácil será implementar, probar y mantener el componente

Procedimental. Las componentes o las operaciones están agrupados de manera que permiten la invocación de uno inmediatamente después de que se invoque el anterior, aunque no se hayan pasado datos entre ellos.

Temporal. Las operaciones que se realizan reflejan un comportamiento o estado específico, como una operación que se realiza al principio o todas las operaciones realizadas cuando se detecta un error.

Utilitaria. Se han agrupado componentes, clases u operaciones que existen dentro de la misma categoría, pero que no tienen otra relación. Por ejemplo, una clase llamada **Estadistica** muestra cohesión utilitaria si contiene todos los atributos y las operaciones necesarios para calcular seis medidas estadísticas simples.

Estos grados de cohesión son menos deseables y deben evitarse cuando existen otras opciones de diseño. Sin embargo, es importante tomar en cuenta que a veces los temas pragmáticos de diseño e implementación obligan al diseñador a optar por los grados inferiores de cohesión.

HOGARSEGURO

Cohesión en acción

El escenario: Cubículo de Jamie.

Los personajes: Jamie y Ed, integrantes del equipo de ingeniería del software que trabajan en la función de vigilancia.

La conversación:

Ed: Tengo un diseño preliminar del componente de cámara.

Jamie: ¿Puedo hacerle una rápida revisión?

Ed: Supongo que sí... pero en realidad quisiera tu opinión sobre algo.

[Jamie le hace un gesto para que siga hablando.]

Ed: Originalmente definimos cinco operaciones para cámara. Mira... [muestra la lista a Jamie].

determinarTipo( ) me dice el tipo de cámara.

traducirUbicacion( ) me permite mover la cámara por el plano de la casa.

desplegarID( ) obtiene el ID de la cámara y la muestra junto al icono de ésta.

desplegarVista( ) me muestra gráficamente el campo de vista de la cámara.

desplegarZoom( ) me muestra gráficamente la amplicación de la cámara.

Ed: He diseñado cada uno por separado, y son operaciones muy simples. De modo que sería buena idea combinar todas las operaciones de despliegue en una sola que se llamara *desplegarCámara( )*; mostraría el ID, la vista y el zoom. ¿Qué te parece?

Jamie (haciendo una mueca): No estoy seguro de que sea una buena idea.

Ed (frunciendo el ceño): ¿Por qué? Todas estas pequeñas operaciones pueden causar dolores de cabeza.

Jamie: El problema de combinarlos es que perdemos cohesión. Tú sabes, la operación *desplegarCámara( )* no tendría una sola función.

Ed (un poco exasperado): ¿Y eso qué? Todo esto tendrá más de cien líneas de código. Creo que será fácil implementarlo.

Jamie: ¿Y qué pasaría si mercadotecnia decide cambiar la manera en que representamos el campo de vista?

Ed: Simplemente me meto en la operación *desplegarCámara( )* y elaboro el módulo.

Jamie: ¿Y qué pasa con los efectos colaterales?

Ed: ¿A qué te refieres?

[illegible]: Bueno, digamos que haces el cambio pero, sin [illegible] cuenta, creas un problema con el despliegue del ID [illegible] No sería tan descuidado

[illegible]: Tal vez no, pero qué pasaría si una persona de [illegible] tiene que hacer el módulo dentro de dos años. [illegible]z no comprenda la operación tan bien como tú y, [illegible] sabe, podría ser descuidado.

Ed: ¿De modo que estás en contra de él?

Jamie: Tú eres el diseñador [illegible] es tu decisión, [illegible] sólo asegúrate de comprender las consecuencias de una baja cohesión.

Ed (pensándolo por un momento): Tal vez [illegible] mos ir con diferentes operaciones de despliegue.

Jamie: Buena decisión

11.2.4 Acoplamiento

En exposiciones precedentes del análisis y el diseño se observó que la comunicación y la colaboración son elementos esenciales de cualquier sistema orientado a objetos. Sin embargo, hay un lado oscuro en esta importante (y necesaria) característica. A medida que aumenta la cantidad de comunicación y colaboración (es decir, a medida que crece el grado de "conectividad" entre las clases), también aumenta la complejidad del sistema. Y a medida que ésta aumenta, la dificulta de implementar, probar y mantener el software también lo hace.

El *acoplamiento* es una medida cualitativa del grado al que las clases se conectan entre sí. A medida que las clases (y los componentes) se vuelven más interdependientes, el acoplamiento aumenta. Un objetivo importante en el diseño al nivel de componentes consiste en mantener el acoplamiento lo más bajo posible.

El acoplamiento de clase se manifiesta de varias maneras. Lethbridge y Laganière [LET01] definen las siguientes categorías de acoplamiento:

[illegible] que se ena- [illegible] diseño de cada [illegible] de soft- [illegible] atención se [illegible] al diseño de [illegible] es de datos [illegible] s y a los di- [illegible] procedimentales [illegible] manipular los es- [illegible] s de datos. Sin [illegible] o, no deben [illegible] se la arquitectu- [illegible] debe albergar [illegible] componentes a los [illegible] globales [illegible] que pueden [illegible] muchos com-

Acoplamiento del contenido. Ocurre cuando un componente "modifica subrepticiamente datos internos de otro" [LET01]. Esto viola la ocultación de la información, que es un concepto básico del diseño.

Acoplamiento común. Ocurre cuando varios componentes usan una variable global. Aunque esto es necesario en algunas ocasiones (por ejemplo, para establecer valores predeterminados en toda una aplicación), el acoplamiento común puede llevar a la propagación incontrolable de errores y a efectos colaterales imprevisibles cuando se hacen cambios.

Acoplamiento de control. Se presenta cuando la *operación A( )* invoca la *operación B( )* y pasa una marca de control a *B*. La marca de control "dirige" entonces el flujo lógico dentro de *B*. El problema con esta forma de acoplamiento es que un cambio no relacionado en *B* puede causar la necesidad de cambiar el significado de la marca de control que pasa *A*. Si esto se omite, se presentará un error.

Acoplamiento de estampa. Ocurre cuando **ClaseB** se declara como tipo para un argumento de una operación de **ClaseA**. Debido a que **ClaseB** ahora es parte de la definición de **ClaseA**, la modificación del sistema se vuelve más compleja

Acoplamiento de datos. Ocurre cuando las operaciones pasan cadenas largas de argumentos de datos. El "ancho de banda" de la comunicación entre clases o componentes crece y la complejidad de la interfaz aumenta. La prueba y el mantenimiento son más difíciles.

Acoplamiento de llamada a rutina. Ocurre cuando una operación invoca a otra. Este grado de acoplamiento es común y, a menudo, muy necesario. Sin embargo, aumenta la conectividad de un sistema.

Acoplamiento de uso de tipo. Ocurre cuando el componente **A** usa un tipo de datos definido en el componente **B** (por ejemplo, esto ocurre cada vez que "una clase declara una variable de instancia o una local como si tuviera otra clase para su tipo [LET01]). Si cambia la definición del tipo, también deben cambiar todos los componentes que usan la definición.

Acoplamiento de inclusión o importación. Ocurre cuando el componente **A** importa o incluye un paquete o el contenido del componente **B**.

Acoplamiento externo. Ocurre cuando un componente se comunica o colabora con componentes de infraestructura (como las funciones del sistema de operación, la capacidad de la base de datos, las funciones de comunicación). Aunque este tipo de acoplamiento es necesario, debe limitarse a un pequeño número de componente o clases dentro de un sistema.

El software debe comunicarse interna y externamente. Por tanto, el acoplamiento es fundamental. Sin embargo, el diseñador debe trabajar para reducir el acoplamiento cada vez que sea posible y comprender las ramificaciones de un acoplamiento elevado cuando no pueda evitarse.

HogarSeguro

Acoplamiento en acción

El escenario: Cubículo de Shakira.

Los actores: Vinod y Shakira, integrantes del equipo de ingeniería del software *HogarSeguro* que trabajan en la función de seguridad.

La conversación:

Shakira: Tuve lo que consideraba una estupenda idea... Luego lo pensé un poco mejor y no me pareció tan buena. Por último, la rechacé, pero pensé que sería conveniente compartirla contigo.

Vinod: Seguro, ¿cuál es la idea?

Shakira: Bueno, cada uno de los sensores reconoce una condición de alarma de cada tipo, ¿verdad?

Vinod (sonriendo): Por eso los llamamos sensores, Shakira.

Shakira (exasperada): No seas sarcástico, Vinod. Tienes que trabajar en tus habilidades interpersonales.

Vinod: ¿Qué me estabas diciendo?

Shakira: Bien, de todas maneras, pensaba ¿Por qué no crear una operación en cada objeto sensor denominada *hacerLlamada( )* que colaboraría directamente con el componente **LlamadaSaliente**, bueno, con una interfaz al componente **LlamadaSaliente**.

Vinod (pensativo): ¿En lugar de hacer que esa colaboración ocurra fuera de un componente como **PanelControl** o algo así?

Shakira: Claro... pero luego me dije, eso significa que cada objeto sensor estará conectado al componente **LlamadaSaliente** y que eso significa que está indirecta

...coplado al mundo exterior y . bueno, sólo pen... ...e complicaba un poco las cosas.

...: Estoy de acuerdo. En este caso, es mejor idea ...que la interfaz del sensor pase información a **Pa...al** y hacer que inicie la llamada saliente. Ade... ...diferentes sensores podrían dar diferentes números telefónicos. No querrás que el sensor almacene esa información porque si cambia

Shakira: No parece muy adecuado.

Vinod: El diseño de la heurística para acoplamiento nos indica que no es correcto.

Shakira: De todos modos. .

11.3 CONDUCCIÓN DEL DISEÑO AL NIVEL DE COMPONENTES

Al principio de este capítulo se observó que el diseño al nivel de componentes es de naturaleza elaborativa. El diseñador debe transformar la información del análisis y los modelos arquitectónicos en una representación de diseño que proporcione suficiente detalle para guiar la actividad de construcción (codificación y prueba). Los siguientes pasos representan un conjunto de tareas típicas para el diseño al nivel de componentes, cuando se aplica a un sistema orientado a objetos.

Paso 1. Identificar todas las clases de diseño que corresponden al dominio del problema. Usando los modelos de análisis y arquitectónico, cada clase de análisis y componente arquitectónico está elaborado como se describió en la sección 11.1.1.

CONSEJO

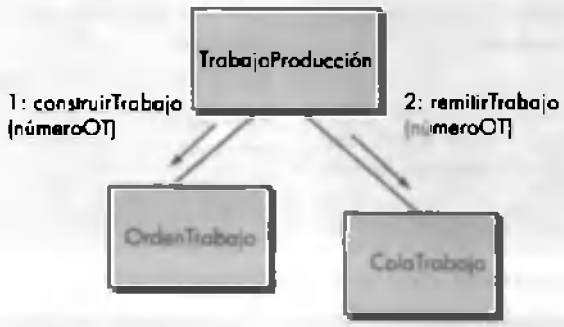
...trabajando ...o que no ...a obje... ...meros tres ...concentran ...ento de ...de datos y ...miento ...(transfor... ...definidas ...del mode...

Paso 2. Identificar todas las clases de diseño que corresponden al dominio de la infraestructura. Estas clases no se describen en el modelo del análisis y a menudo faltan en el modelo arquitectónico, pero deben describirse en este punto. Como ya se ha indicado, entre las clases y los componentes de esta categoría se incluyen componentes de interfaz gráfica de usuario, del sistema operativo, de administración de objetos y datos, y otros.

Paso 3. Elaborar todas las clases de diseño que no se adquieran como componentes reutilizables. La elaboración requiere que se describan de manera detallada todas las interfases, los atributos y las operaciones necesarios para implementar la clase. Al realizar esta tarea debe tomarse en cuenta el diseño de la heurística (por ejemplo, la cohesión y el acoplamiento de componentes).

Paso 3a. Especificar los detalles del mensaje cuando las clases o los componentes colaboran. El modelo del análisis emplea el diagrama de colaboración para mostrar la manera en que las clases de análisis colaboran entre sí. A medida que avanza el diseño al nivel de componentes, a veces es útil mostrar los detalles de estas colaboraciones al especificar la estructura de mensajes que se pasan entre los objetos de un sistema. Aunque esta actividad de diseño es opcional, puede usarse como precursora de la especificación de interfaces que muestran la manera en que se comunican y colaboran los componentes del sistema.

FIGURA 11.6
Diagrama de colaboración con envío de mensajes.



En la figura 11.6 se ilustra un diagrama simple de colaboración para el sistema de impresión analizado antes. Tres objetos, **TrabajoProduccion, OrdenTrabajo** y **ColaTrabajo**, colaboran para preparar el envío de un trabajo de impresión al flujo de producción. Los mensajes se pasan entre objetos como lo ilustran las flechas en la figura. Durante el modelado del análisis los mensajes se especifican como se muestra en la figura. Sin embargo, a medida que avanza el diseño, cada mensaje se elabora al expandir su sintaxis de la siguiente manera [BEN02].

[condición guardia] expresión de secuencia (valor devuelto): = nombre del mensaje (lista de argumentos)

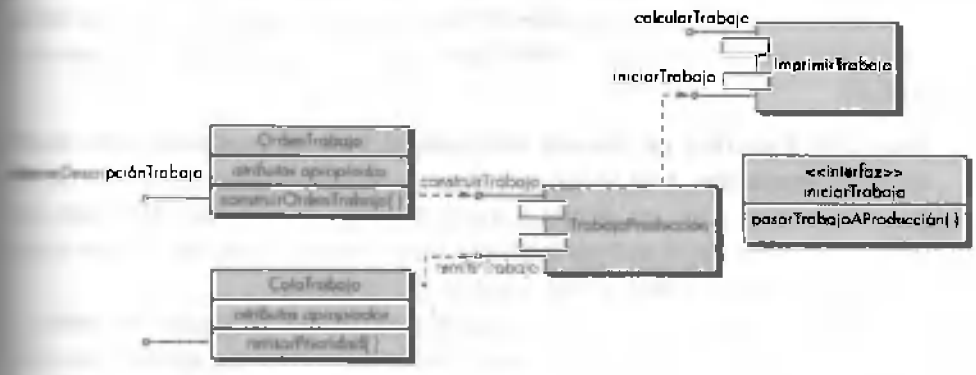
donde una **[condición guardia]** está escrita en lenguaje de restricción de objeto (OCL por sus siglas en inglés)[4] y especifica cualquier conjunto de condiciones que deben cumplirse antes de enviar el mensaje; **expresión de secuencia** es un valor entero (u otro indicador de orden, como 3.1.2) que indica el orden secuencial en que se envía un mensaje; **(valor devuelto)** es el nombre de la información que devuelve la operación invocada por el mensaje; **nombre del mensaje** identifica la operación que se invoca y **(lista de argumentos)** es la lista de los atributos que se pasan a la operación

Paso 3b. Identificar las interfaces apropiadas para cada componente. En el contexto del diseño al nivel de componentes, una interfaz UML es un "grupo de operaciones externamente visibles (es decir, públicas). La interfase no contiene estructura interna; no tiene atributos ni asociaciones..." [BEB02]. Definida de manera más formal, una interfaz es el equivalente a una clase abstracta que proporciona una conexión controlada entre las clases de diseño. La elaboración de una interfaz se muestra en la figura 11.1. En esencia, las operaciones definidas para la clase de diseño están ordenadas en una o más clases abstractas. Cada operación dentro de la clase abstracta (la interfaz) debe tener cohesión; es decir, debe mostrar procesamiento que se concentra en una función o subfunción limitada.

Tomando como referencia la figura 11.1, podría argumentarse que la interfaz *iniciarTrabajo* no muestra suficiente cohesión. En realidad, realiza tres subfunciones:

4 El OCL se analiza brevemente en la sección 11.4 y en el capítulo 28

Figura 11.7 Refactorización de definiciones de interfases y clases para ImprimirTrabajo.



ferentes: construir una orden de trabajo, revisar la prioridad del trabajo y pasar un trabajo a producción. El diseño de la interfaz debe refactorizar. Un enfoque sería reexaminar las clases del diseño y definir una nueva clase **OrdenTrabajo** que se ocuparía de todas las actividades asociadas con la elaboración de una orden de trabajo. La operación *construirOrdenTrabajo( )* se vuelve una parte de esa clase. De igual manera, se podría definir una clase **FilaTrabajo** que incorporaría la operación *revisarPrioridad( )*. Una clase **TrabajoProduccion** abarcaría toda la información asociada con un trabajo de producción que se pasará a la planta de producción. La interfaz *iniciarTrabajo* tomaría entonces la forma mostrada en la figura 11.7. Ahora esta interfase es cohesiva, y se concentra en una sola función. Las interfaces asociadas con **TrabajoProduccion, OrdenTrabajo** y **FilaTrabajo** tienen una sola función

Paso 3c. Elaborar atributos y definir los tipos y las estructuras de datos necesarios para implementarlos. En general, las estructuras y los tipos de datos con que se describen atributos se definen dentro del contexto del lenguaje de programación que habrá de usarse para la implementación. UML define el tipo de datos de un atributo empleando la siguiente sintaxis:

```
nombre : tipo-expresión = valor-inicial {propiedad cadena}
```

donde **nombre** es nombre del atributo y **tipo-expresión** es el tipo de datos; **valor-inicial** es el valor que toma el atributo cuando se crea un objeto y **propiedad cadena** define una propiedad o característica del atributo.

Durante la primera iteración de diseño al nivel de componentes, los atributos suelen describirse por nombre. Tomando como referencia una vez más la figura 11.1, la lista de atributos de **TrabajoImprenta** sólo incluye los nombres de los atributos; sin embargo, a medida que avanza la elaboración del diseño, cada atributo se define empleando el formato de atributos UML indicado. Por ejemplo, **Tipo-pesoPapel** se define de la siguiente manera:

```
Tipo-pesopapel: string = "A"{contiene 1 de 4 valores: A, B, C o D}
```

que define **Tipo-pesopapel** como una variable de cadena variable inicializada con el valor A y que toma uno de cuatro valores del conjunto {A,B,C,D}.

Si un atributo aparece varias veces en varias clases de diseño, y tiene una estructura relativamente compleja, es mejor crear una clase separada para acomodar el atributo.

Paso 3-D. Describir de manera detallada el flujo de procesamiento dentro de cada operación. Esto se logra empleando un seudocódigo basado en un lenguaje de programación (sección 11.5.5) o el diagrama de actividad UML. Cada componente de software se elabora mediante varias interacciones que aplican el concepto de refinamiento paso a paso (capítulo 9).

La primera iteración define cada operación como parte de la clase de diseño. En cada caso, la operación debe estar caracterizada de manera que asegure una cohesión elevada; es decir, la operación debe realizar una sola función o sustitución definida. La siguiente iteración hace poco más que expandir el nombre de la operación. Por ejemplo, la operación *calcularCostoPapel( )* observada en la figura 11.1 se expandiría de la siguiente manera:

```
calcularCostoPapel (peso, tamano, color): numerico
```

Esto indica que *calcularCostoPapel( )* requiere los atributos **peso**, **tamaño** y **color** como entrada y devuelve un valor numérico (en realidad un valor en pesos) como salida.

> "De haber tenido más tiempo, habría escrito una carta más corta."
>
> Blas Pascal

CONSEJO

La elaboración se usa paso a paso mientras se refina el diseño del componente. Siempre debe preguntarse: "¿Hay una manera de simplificar esto mientras sigue arrojando el mismo resultado?"

Si el algoritmo requerido para implementar *calcularCostoPapel( )* es simple y se comprende ampliamente, tal vez sea innecesario elaborar diseño adicional. El ingeniero de software responsable de la codificación proporcionará el detalle necesario para implementar la operación. Sin embargo, si el algoritmo es más complejo o concreto, se requiere mayor elaboración de diseño en esta etapa. En la figura 11.2 se describe un diagrama de actividad UML para *calcularCostoPapel( )*. Cuando se emplean diagramas de actividad para especificación de diseño al nivel de componentes, suelen representarse en un nivel de abstracción un poco más elevado que el código fuente. Más adelante, en este mismo capítulo, se analizará un método para el uso de seudocódigo para especificar el diseño.

Paso 4. Describir fuentes de datos persistentes (bases de datos y archivos) e identificar las clases necesarias para manejarlas. Las bases de datos y los archivos suelen trascender la descripción del diseño de un componente individual. En casi todos los casos estos almacenes de datos persistentes suelen especificarse inicialmente como parte del diseño arquitectónico. Sin embargo, a medida que avanza la elaboración del diseño, a veces resulta útil proporcionar detalles adicionales de la estructura y organización de estas fuentes de datos persistentes.

Paso 5. Desarrollar y elaborar representaciones del comportamiento de una clase o un componente. Los diagramas de estado UML se usaron como parte del modelo de análisis para representar el comportamiento del sistema que se observa externamente y el comportamiento más localizado de clases individuales de análisis. Durante el diseño al nivel de componentes, suele ser necesario modelar el comportamiento de una clase de diseño

Al comportamiento dinámico de un objeto (la instanciación de una clase de diseño mientras se ejecuta el programa) lo afectan eventos externos y el estado actual del objeto (modo o comportamiento). Para comprender el comportamiento dinámico de un objeto, el diseñador debe examinar todos los casos de uso relevantes durante el periodo de vida de la clase de diseño. Estos casos de uso proporcionan información que ayuda al diseñador a delinear los eventos que afectan al objeto y a los estados en que reside éste mientras pasa el tiempo y ocurren los eventos. La transi-

FIGURA 11.8 **Diagrama de actividad UML para *calcularCostoPapel( )*.**

ción entre estados (impulsados por los eventos) se representan empleando una gráfica de estado UML [BEN02] como se ilustra en la figura 11.9.

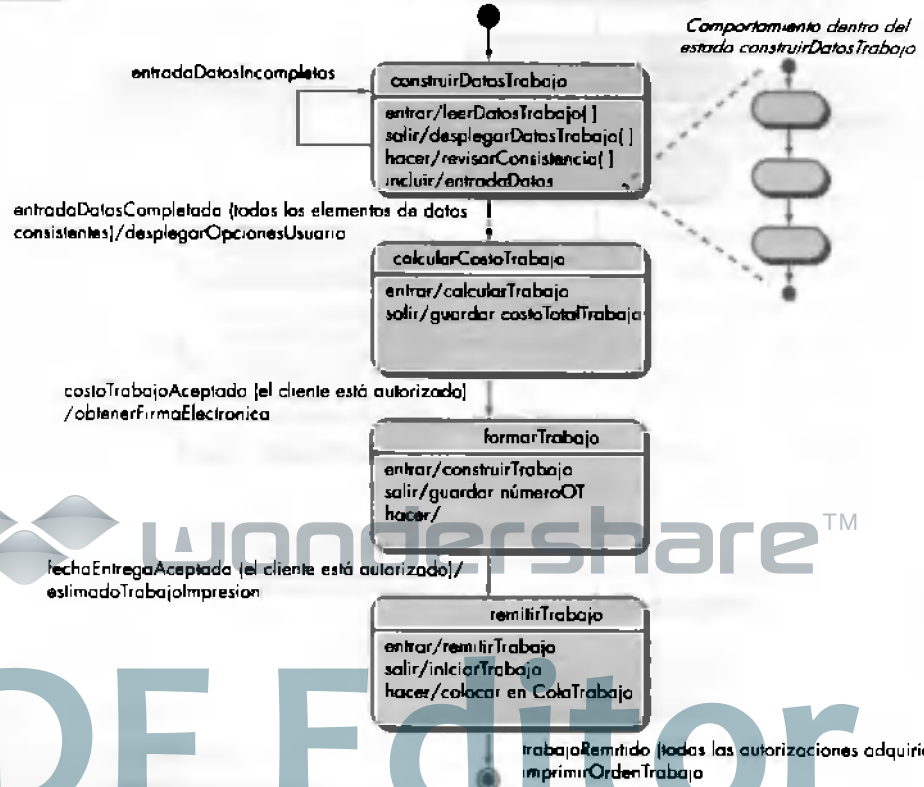
La transición de un estado (representado por un rectángulo con esquinas redondeadas) a otro ocurre como consecuencia de un evento que toma esta forma

nombre-evento (lista-parametros) [condicion-guardia] / expresion de accion

donde **nombre-evento** identifica el evento; **lista-parametros** incorpora datos asociados con el evento; **condición-guardia** está escrita en lenguaje de restricción de objeto (OCL) y especifica una condición que debe cumplirse antes de que pueda ocurrir el evento, y **expresion de accion** define una acción antes de que ocurra cuando tiene lugar la transición

Tomando como referencia la figura 11.9, cada estado puede definir acciones de *entrada/* y *salida/* que ocurren mientras se presentan las transiciones de entrada y salida. En casi todos los casos, estas acciones corresponden a operaciones relevantes para la clase que se está modelando. El indicador *hacer/* proporciona un mecanismo para indicar las actividades que ocurren mientras se encuentra en el estado, y el indicador *incluir/* proporciona un medio para elaborar el comportamiento al incrementar más detalle en la gráfica de estado dentro de la definición de un estado.

FIGURA 11.9 Fragmento de diagrama de estado para la clase TrabajoImprenta.



Es importante observar que el modelo de componentes a menudo contiene información que no resulta inmediatamente obvia en otros de componentes de diseño. Por ejemplo, el cuidadoso examen de la gráfica de estado de la figura 11.9 indica que el comportamiento dinámico de la cláusula **TrabajoImprenta** depende de dos aprobaciones del cliente, derivadas de los datos de costos y la calendarización del trabajo de impresión. Sin aprobaciones (la condición guardia asegura que el cliente tiene autorización para aprobar) no se remitirá el trabajo de impresión porque no hay manera de alcanzar el estado *remitirTrabajo*.

Paso 6. Elaborar diagramas de despliegue para proporcionar detalles de la implementación adicional. Los diagramas de despliegue (capítulo 9) se usan como parte del diseño arquitectónico y se representan en forma de descriptor. Así, se representan las principales funciones del sistema (a menudo representadas como subsistemas) dentro del contexto del entorno de cómputo que las albergará.

Durante el diseño al nivel de componentes pueden elaborarse diagramas de despliegue para representar la ubicación de paquetes clave de componentes. Sin embargo, por lo general los componentes se representan individualmente dentro de un diagrama de componente La razón de esto es evitar la complejidad del diagrama. En algunos casos, los diagramas de despliegue se elaboraron en forma de instancias. Esto significa que el hardware específico y el o los entornos del sistema operativo que se usarán son específicos y que se indica la ubicación de los paquetes de componentes dentro de este entorno.

Paso 7. Factorizar todas las representaciones del diseño al nivel de componentes y siempre deben considerarse alternativas. A lo largo del libro se ha destacado que el diseño es un proceso iterativo. El primer modelo al nivel de componentes que se cree no será tan completo, consistente o exacto como la enésima iteración que aplique al modelo Es esencial usar la refactorización mientras se realiza el trabajo de diseño.

Además, un diseñador no debe tener una visión estrecha Siempre hay soluciones opcionales para el diseño, y los mejores diseñadores toman en cuenta todas (o casi todas) antes de definir el modelo de diseño final. Desarrollan opciones y examinan cada una de manera cuidadosa, empleando los principios del diseño y los conceptos presentados en los capítulos 5, 9 y 11

11.4 LENGUAJE DE RESTRICCIÓN DE OBJETOS

La amplia variedad de diagramas disponible como parte de UML proporciona a un diseñador un rico conjunto de formas de representación para el modelo de diseño. Sin embargo, las representaciones gráficas no suelen bastar. El diseñador necesita un mecanismo para representar explícita y formalmente la información que restringe algún elemento del modelo de diseño. Es posible, por supuesto, describir restricciones en un lenguaje natural, pero este método lleva invariablemente a la inconsisten-

cia y la ambigüedad. Por tanto, lo apropiado parece un lenguaje más formal, que tome en cuenta la teoría de conjuntos y los lenguajes formales de especificación (capítulo 28), pero que tenga una cantidad menor de sintaxis matemática que un lenguaje de programación.

El *lenguaje de restricción de objetos* (OCL) complementa al UML al permitir que un ingeniero de software use gramática y sintaxis formales para construir instrucciones sin ambigüedades relacionadas con varios elementos del modelo de diseño (por ejemplo, clases y objetos, eventos, mensajes, interfaces). En el OCL las instrucciones se construyen en cuatro partes: 1) un contexto que define la situación limitada en que es válida la instrucción; 2) una propiedad que representan algunas características del contexto (por ejemplo, si el contexto es una clase, una propiedad sería un atributo); 3) una *operación* (aritmética, orientada a conjuntos) que manipula o califica una propiedad, y 4) *palabras clave* (como **if, then, else, and, or, not, implies**) con que se especifican expresiones condicionales.

PUNTO CLAVE

El OCL proporciona gramática y sintaxis formales para describir los elementos de diseño al nivel de componentes

Como ejemplo simple de una expresión OCL, considérese la condición guardia colocada en el evento *costoTrabajoAceptado* que causa una transición entre los estados **calcularCostoTrabajo** y **formarTrabajo** dentro del diagrama de gráfica de estado para la clase **TrabajoImprenta** (figura 11.9). En el diagrama, la condición guardia se expresa en lenguaje natural y especifica que la autorización sólo se presentará si el cliente está autorizado para aprobar el costo del trabajo. En el OCL, la expresión tomaría la forma:

```
cliente
    tiene.autoridadAutorización = "sí"
```

donde un atributo booleano, **autoridadAutorización**, de la clase **Cliente** (en realidad una instancia específica de la clase) debe tener el valor **sí** para satisfacer la condición guardia.

Cuando se crea el modelo de diseño suele haber instancias (consulte la sección 11.2.1) en que deben satisfacerse las condiciones previas y posteriores antes de completar alguna opción especificada en el diseño. El OCL proporciona una herramienta poderosa para especificar condiciones previas y posteriores de manera formal. Como ejemplo, piense en una extensión al sistema de la imprenta (analizado a lo largo de este capítulo) en que el cliente proporciona un límite de costo superior para el trabajo de impresión y una fecha de entrega límite, al mismo tiempo que se especifican otras características del trabajo. Si el costo y la entrega estimada exceden esos límites, el trabajo no se entregará y debe notificarse al cliente. En el OCL un conjunto de condiciones previas y posteriores se especificaría de la siguiente manera:

```
context TrabajoImprenta::validate(limiteSuperiorCosto : Integer, reqEnvioCliente :
Integer)
  pre: limiteSuperiorCosto > 0
```

```
        and reqEnvioCliente > 0
        and tiene.autorizacionTrabajo = "no"
    post: if tiene.costoTotalTrabajo <= limiteSuperiorCosto
            and tiene.fechaEnvio <= reqEnvioCliente
        then
            tiene.autorizacionTrabajo = "si"
        endif
```

Esta declaración OCL define una invariante (condiciones que deben existir antes **[pre]** y después **[post]** de algún comportamiento). Al principio, la condición previa establece que el cliente debe especificar el costo límite y la fecha de entrega, y que la autorización debe estar en "no" Después de determinar los costos y la fecha de envío, se aplica la condición posterior. También debe tomarse en cuenta que la expresión **tiene.autorizacionTrabajo = "si"** no está asignando el valor "sí"; en cambio, está declarando que **autorizacionTrabajo** debe tener el valor "sí" en el momento en que termine la operación

Una descripción completa del OCL está más allá del alcance de este libro.[5] Los lectores interesados deben consultar [WAR98] y [OMG01] para conocer detalles adicionales

HERRAMIENTAS DE SOFTWARE

UML/OCL

Objetivo: Existe una amplia variedad de herramientas UML para ayudar al diseñador en todas las etapas del diseño Algunas de estas herramientas proporcionan soporte al OCL.

Mecánica: Las herramientas de esta categoría permiten a un diseñador crear todos los diagramas de UML necesarios para construir un modelo de diseño completo. Lo más importante es que muchas herramientas proporcionan una sintaxis y una semántica sólidas, verificación y manejo de control de versión y cambios (capítulo 27) Cuando se proporciona capacidad de OCL, las herramientas permiten que el diseñador cree expresiones OCL y, en algunos casos, las "compile" para varios tipos de evaluación y análisis

Herramientas representativas[6]

ArgoUML, distribuido por Tigress.org (http://argouml.tigris.org/), da soporte a UML y OCL completo, e incluye varias herramientas de ayuda para el diseño, que van más allá de la generación de diagramas UML y expresiones OCL.

Dresden OCL toolkit, desarrollado por Frank Finger en la Universidad Tecnológica de Dresden (http://dresden-ocl.sourceforge.net/), es un juego de herramientas basada en un compilador OCL que abarca varios módulos que analizan, revisan el tipo y normalizan las restricciones OCL.

OCL parser, desarrollado por IBM (http://www3.ibm.com/software/ad/library/standards/OCL-download.html), está escrito en Java y está disponible gratuitamente para la comunidad orientada a objetos con el fin de que se estimule el uso de OCL con modeladores UML

5 Sin embargo, en el capítulo 29 se presentara una exposicion más amplia del OCL (presentada en el contexto de los métodos formales)

6 Las herramientas mencionadas aquí representan una muestra de esta categoría. En casi todos los casos los nombres de las mismas son marcas registradas de sus respectivos desarrolladores

11.5 DISEÑO DE COMPONENTES CONVENCIONALES

Los fundamentos del diseño al nivel de componentes para componentes convencionales de software[7] se integraron a principios de la década de 1960 y adquirieron solidez con el trabajo de Edsgar Dijkstra y sus colegas ([BOH66], [DIJ65], [DIJ76]). A nales de esos años, Dijkstra y otros propusieron el uso de un conjunto de construcciones lógicas restringidas, a partir de las cuales se pudiera formar cualquier programa. Las construcciones destacaban el "mantenimiento del dominio funcional" decir, cada construcción tenía una estructura lógica predecible, a la que se ingresaba en la parte superior y se abandonaba en la inferior, lo que permitía al lector seguir con mayor facilidad el flujo del procedimiento.

PUNTO CLAVE

La programación estructurada es una técnica de diseño que restringe el flujo de la lógica a tres construcciones: secuencia, condición y repetición

Las construcciones son secuencia, condición y repetición. *Secuencia* implemen los pasos de procesamiento esenciales en la especificación de cualquier algoritmo. *Condición* proporciona las funciones para el procesamiento seleccionado con base en algún evento lógico, y *repetición* permite los bucles. Estas tres construcciones fundamentales para la programación estructurada, que es una importante técnica diseño al nivel de componentes.

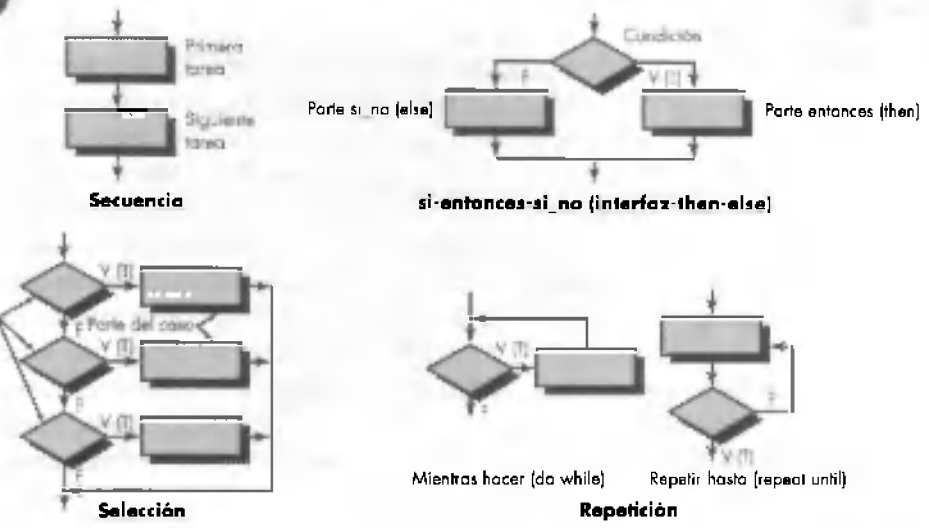
Las construcciones estructuradas se propusieron para restringir el diseño procedimental del software a un número pequeño de operaciones. La complejidad de métricas (capítulo 15) indica que el uso de las construcciones estructuradas reduce la complejidad del programa y, por tanto, mejora las opciones de lectura, prueba mantenimiento. El uso de un número limitado de construcciones lógicas también contribuye a un proceso de comprensión humana que los psicólogos llaman *mentación*. Para comprender este proceso, piénsese en la manera en que está leyendo esta página. No se leen letras individuales sino que se reconocen patrones o grupos de palabras o frases formados por varias letras. Las construcciones estructuradas son grupos lógicos que le permiten a un lector reconocer elementos procedimentales de un módulo, en lugar de leer el diseño o código línea por línea. La comprensión se mejora cuando hay patrones lógicos fácilmente reconocibles.

11.5.1 Notación gráfica del diseño

Ya se ha analizado el diagrama de actividad UML en este capítulo y en los capítulos 7 y 8. El diagrama de actividad permite a un diseñador representar secuencia, condición y repetición (todos elementos de la programación estructurada) y es el descendiente de una representación de diseño gráfico anterior (aún usado ampliamente) llamado *diagrama de flujo*.

Un diagrama de flujo, como uno de actividad, es muy simple gráficamente. Un diamante representa una condición lógica, y las flechas muestran el flujo de control.

[7] Un componente de software convencional implementa un elemento de procesamiento que forma una función o subfunción en el dominio del problema, o alguna capacidad en el dominio de la infraestructura. A menudo denominados *módulos*, *procedimientos* o *subrutinas*, los componentes convencionales no encapsulan datos de la misma manera que los componentes orientados a objetos.



En la figura 11.10 se ilustran tres construcciones estructuradas. La *secuencia* se representa como dos cajas de procesamiento conectadas por una línea (flecha) de control. La *condición*, también llamada *si-entonces-si_no*, se describe como un diamante de decisión que, si es verdadero, causa que ocurra la parte-entonces del procesamiento, y si es falso, invoca la parte *si_no*. La *repetición* se representa empleando dos formas ligeramente diferentes. La parte *hacer mientras* prueba una condición y ejecuta una tarea de bucle de manera repetitiva, siempre y cuando la condición siga siendo verdadera. Una parte *repetir hasta* ejecuta primero la tarea de bucle, luego prueba una condición y repite la tarea hasta que la condición falla. La construcción de *selección* (o *seleccionar caso*) que muestra la figura es en realidad una extensión de *si-entonces-si_no*. Sucesivas decisiones prueban un parámetro hasta que ocurre una condición verdadera y se ejecuta la ruta de procesamiento de la *parte de caso*.

En general, el uso dogmático y exclusivo de las construcciones estructuradas introduce ineficiencia cuando se requiere un escape de un conjunto de bucles anidados o de condiciones anidadas. Lo que es más importante, la complicación adicional de todas las pruebas lógicas junto con la ruta de escape llega a oscurecer el flujo de control del software, aumenta la posibilidad de error y tiene un impacto negativo en la legibilidad y la capacidad de mantenimiento. ¿Qué podemos hacer?

Se le dejan dos opciones al diseñador: 1) se rediseña la representación procedimental para que la "rama de escape" no sea necesaria en una ubicación anidada del flujo de control, o 2) se violan las construcciones estructuradas de una manera controlada; es decir, se diseña una rama restringida fuera del flujo anidado. La opción 1 es obviamente el enfoque ideal, pero la 2 puede acomodarse sin infringir el espíritu de la programación estructurada.

Debe usarse una tabla de decisión cuando un conjunto complejo de condiciones y acciones se encuentran dentro de un componente.

11.5.2 Notación tabular del diseño

En muchas aplicaciones de software tal vez se requiera un módulo para evaluar una combinación compleja de condiciones y seleccionar las acciones apropiadas basadas en esas condiciones. Las *tablas de decisión* [HUR83] proporcionan una notación que traduce acciones y condiciones (descritas en una narrativa de procesamiento) a una forma tabular. Es difícil malinterpretar una tabla, y hasta puede usarse como entrada legible para una máquina a un algoritmo orientado a tablas.

Una tabla de decisión se divide en cuatro cuadrantes. El de la esquina superior izquierda contiene una lista de todas las condiciones. El cuadrante de la esquina inferior izquierda contiene una lista de todas las acciones posibles, basada en combinaciones de condiciones. Los cuadrantes de la derecha forman una matriz que indica combinaciones de condición y las acciones correspondientes que ocurrirán para una combinación específica. Por tanto, cada columna de la matriz puede interpretarse como una *regla de procesamiento*. Los siguientes pasos se aplican para desarrollar una tabla de decisión:

1. Presentar una lista de todas las acciones que puedan asociarse con un procedimiento (o módulo) específico.
2. Presentar una lista de todas las condiciones (o decisiones tomadas) durante la ejecución de un procedimiento.
3. Asociar conjuntos específicos con acciones específicas, eliminando combinaciones específicas de condiciones; como opción, desarróllese cada posible permutación de las condiciones.
4. Definir reglas al indicar cuáles acciones ocurren para un conjunto de condiciones.

Para ilustrar el uso de una tabla de decisión, piénsese en el siguiente extracto de un caso de uso informal que se ha propuesto para el sistema de la imprenta:

> Tres tipos de clientes están definidos: un cliente regular, uno de plata y uno de oro. Los tipos se asignan según la cantidad de negocios que el cliente realiza con la imprenta (en un periodo de 12 meses). Un cliente regular recibe precios de impresión y fechas de entrega normales. Un cliente de plata obtiene un descuento de 8 por ciento sobre todas las cotizaciones y se coloca adelante de los clientes regulares en la cola de impresión. Un cliente de oro obtiene una reducción del 15 por ciento sobre los precios cotizados y se coloca adelante de los clientes regulares y de plata en la cola de trabajo. Es posible aplicar un descuento especial de x porcentaje adicional a los otros descuentos a la cotización de cualquier cliente, a discreción de la administración.

En la figura 11.11 se ilustra una representación de una tabla de decisión relacionada con el anterior caso de uso informal. Cada una de las seis reglas indica una de las seis condiciones viables. Como regla general, la tabla de decisión se usa de manera efectiva para complementar otras notaciones de diseño procedimental.

FIGURA 11.11
Tabla de decisión resultante.

	Reglas					
Condiciones	**1**	**2**	**3**	**4**	**5**	**6**
Cliente regular	V (T)	V (T)				
Cliente plata			V (T)	V (T)		
Cliente oro					V (T)	V (T)
Cliente especial	F	V (T)	F	V (T)	F	V (T)
Acciones						
Sin descuento	✓					
Aplicar 8 por ciento de descuento			✓	✓		
Aplicar 15 por ciento de descuento					✓	✓
Aplicar x porcentaje de descuento adicional		✓		✓		✓

11.5.3 Lenguaje de diseño de programas

El *lenguaje de diseño de programas* (PDL, por sus siglas en inglés), también denominado *lenguaje común estructurado* o *seudocódigo*, es "un lenguaje rudimentario porque utiliza el vocabulario de un idioma (como el inglés) y la sintaxis general de otro (es decir, un lenguaje estructurado de programación)" [CAI75]. En este capítulo, PDL se usa como referencia genérica para un lenguaje de diseño.

A primera vista, PDL parecería un lenguaje de programación. La diferencia entre PDL y un lenguaje de programación real radica en el uso de texto narrativo (como el inglés) incrustado directamente dentro de las instrucciones en PDL. Dado el uso de texto narrativo incrustado directamente en una estructura sintáctica, no es posible compilar PDL. Sin embargo, algunas herramientas pueden traducirlo en un "esqueleto" de lenguaje de programación, en una representación gráfica de diseño, o en ambas (por ejemplo, un diagrama de flujo). Estas herramientas también producen mapas de anidamiento, un índice de operación de diseño, tablas de referencia cruzada y otra información diversa.

Un lenguaje de diseño de programas puede ser una simple transposición de un lenguaje como Ada, C o Java. La sintaxis básica de PDL debe incluir construcciones para definición de componentes, descripción de interfaces, declaración de datos, estructuración de bloques y construcciones de condiciones, de repetición y de entrada/salida. Debe tomarse en cuenta que PDL puede extenderse para incluir palabras clave para multitareas, procesamiento concurrente (o ambas opciones), manejo de interrupciones, sincronización de interprocesos y muchas otras características. El diseño de la aplicación para la que se está usando PDL debe dictar la forma final del lenguaje de diseño. El formato y la semántica de algunas de estas construcciones de PDL se presentan en el ejemplo siguiente. Para ilustrar el uso de PDL, consideramos un diseño procedimental para la función de seguridad *HogarSeguro* analizada en capítulos anteriores. El sistema supervisa las alarmas para detectar fuego, humo, robo,

CONSEJO

...idad usar su ...da programa- ...base para ...o se pue- ...un esque- ...código ...con texto ...mientras se ...el diseño.

agua y temperatura (por ejemplo, rompimiento del horno cuando el propietario está ausente en el invierno), produce un timbre de alarma y llama a un sistema de monitores, generando un mensaje de voz sintetizado. En el PDL siguiente ilustramos algunas de las construcciones importantes anotadas en secciones anteriores.

Recuerde que PDL *no* es un lenguaje de programación. El diseñador puede adaptarlo como se requiera sin preocuparse por errores de sintaxis. Sin embargo, el diseño del software de supervisión tendría que revisarse (¿se observa algún problema?) y refinarse antes de que pueda escribirse el código. El siguiente PDL[8] proporciona una elaboración del diseño procedimental para una versión anterior de un componente de manejo de alarmas.

```
componente manejoAlarma
El objetivo de este componente es manejar los interruptores y las entradas del panel de
control a partir de los sensores por el tipo y actuar en cualquier condición de alarma que
sea encontrado.
   establecer valores por defecto para estatusSistema (valor devuelto), todos los elemen-
   tos de datos
   inicializar todos los puertos del sistema y reiniciar todo el hardware
   revisar interruptoresPanelControl (ipc)
      si ipc = "probar" entonces invocar alarma fijar en "encendido"
      si ipc = "alarmaApagado" entonces invocar alarma fijar en "apagado"
      •
      •
      •
      valor por defecto ipc = ninguno
   restablecer todo valoresSeñal e interruptores
   hacer para todos los sensores
      invocar verificarSensor procedimiento regresa valorSeñal
      si valorSeñal > límite [tipoAlarma]
         entonces teléfono.mensaje = mensaje [tipoAlarma]
            fijar timbreAlarma en "encendido" para alarmaTiempoSegundos
            fijar estatus sistema = "condiciónAlarma"
            parempieza
               procedimiento alarma con "encendido", alarmaTiempoSegundos
               invocar procedimiento teléfono fijar en tipoAlarma, númeroTeléfono
            partermina
         si_no omitir
      termina si
   termina hacerpara
termina manejoAlarma
```

8 El nivel de detalle que representa el PDL se define localmente. Algunas personas prefieren una descripción orientada al lenguaje más natural, mientras que otras prefieren algo más parecido a un código.

Obsérvese que el diseñador del componente de manejo de alarma ha usado las construcciones **parempieza...parlarmina** que especifica un bloque paralelo. Todas las tareas especificadas en el bloque **parempieza** se ejecutan en paralelo. En este caso, no se toman en cuenta los detalles de implementación.

HERRAMIENTAS DE SOFTWARE

Lenguaje de diseño de programas

Objetivo: Aunque la inmensa mayoría de los ingenieros de software que usa PDL o seudocódesarrolla una versión que se adapta del lenguaje de ación que tratan de emplear para la implementaexisten varias herramientas de PDL

ica: En algunos casos, las herramientas aplican inversa al código fuente (una triste realidad en donde algunos programas no tienen absolutaninguna documentación). Otras permiten al diseñacrear PDL con una ayuda automatizada

ientas representativas[9]

desarrollado por Caine, Farber y Gordon p://www.cfg.com/pdl81/lpd.html), da soporte a la creación de diseños con el uso de una versión definida de PDL

DocGen, distribuido por Software Improvment Group (http://www.software-improvers.com/DocGen.htm), es una herramienta de ingeniería inversa que genera documentación parecida a PDL a partir de código Ada y C.

PowerPDL, desarrollado por Iconix (http://www.iconixsw.com/SpecSheets/PowerPDL.html), le permite a un diseñador crear PDL basado en diseños y luego traducir el seudocódigo a formas que puedan generar otras representaciones de diseño.

11.5.4 Comparación entre notaciones de diseño

La notación de diseño debe llevar a una representación procedimental fácil de comprender y revisar. Además, la notación debe mejorar la capacidad de "codificar en" para que el código, en realidad, se convierta en un subproducto natural del diseño Por último, la representación de diseño debe tener la capacidad de darle mantenimiento fácilmente para que el diseño siempre represente el programa de manera correcta

Una pregunta natural que surge en cualquier análisis de la notación de diseño sería: ¿Cuál notación es realmente mejor, dados los atributos indicados líneas antes? Cualquier respuesta es subjetiva y está abierta al debate. Sin embargo, parece que el lenguaje de diseño de programas ofrece la mejor combinación de características. El PDL puede incrustarse directamente en los listados de código fuente, mejorando la documentación y facilitando más el mantenimiento del diseño. La edición se hace en cualquier editor de texto o sistema de procesamiento de palabras, ya existen procesadores automáticos, y la posibilidad de "generación automática de código" es buena.

Sin embargo, de esto no se desprende que cualquier otra notación sea necesariamente inferior a PDL, o que "no sea buena" en atributos específicos. La naturaleza

9 Las herramientas expuestas aquí el autor no las respalda; sólo representan una muestra de las herramientas incluidas en esta categoría En casi todos los casos, los nombres de las herramientas son marcas registradas de sus respectivos desarrolladores

gráfica de los diagramas de actividad y los de flujo proporciona una perspectiva s bre el flujo de control que muchos diseñadores prefieren. El contenido tabular pre so de las tablas de decisión es una herramienta excelente para aplicaciones orient das a tablas. Y muchas otras representaciones de diseño (por ejemplo, nidos de tri), no presentados en este libro, ofrecen sus propios beneficios únicos. En el an sis final, la elección de una herramienta de diseño estará relacionada de manera estrecha con factores humanos que con atributos técnicos.

11.6 RESUMEN

La acción de diseño al nivel de componentes abarca una secuencia de tareas que ducen lentamente el grado de abstracción con que se representa el software. E seño al nivel de componentes describe finalmente el software en un grado de tracción cercano al código

Es posible tomar dos enfoques distintos de diseño al nivel de componentes, lo depende de la naturaleza del software que habrá de desarrollarse. El concepto tado a objetos se enfoca en la elaboración de clases de diseño que provienen del problema como del dominio de la infraestructura. El concepto convencional fina tres tipos principales de componentes o módulos: de control, de dominio problema y de infraestructura. En ambos casos se aplican principios básicos de seño y conceptos que llevan a software de mayor calidad. Cuando se considera de un punto de vista del proceso, el diseño al nivel de componentes se basa en ponentes de software reutilizables y en patrones de diseño que son elementos te de la ingeniería de software basada en componentes.

El diseño al nivel de componentes orientado a objetos se basa en clases principios y conceptos importantes guían al diseñador a medida que se elaboran clases Principios como el principio abierto-cerrado y el de inversión de la dep cia, además de conceptos como acoplamiento y cohesión, guían al ingeniero de ware en la construcción de componentes de software susceptibles de probars plementarse y mantenerse. Para realizar diseño al nivel de componentes e contexto, las clases se elaboran al especificar detalles de los mensajes, identif terfaces apropiadas, elaborar atributos y definir estructuras de datos para imp tarlos, describir el flujo de procesamiento dentro de cada operación y repres comportamiento en un nivel de clase o componente. En todo caso, la iteración seño es una actividad esencial.

El diseño al nivel de componentes convencional requiere la representación tructuras de datos, interfaces y algoritmos para un módulo de programa con lle suficiente para servir como guía en la generación de código fuente en leng programación. Para lograr esto, el diseñador usa una de varias notaciones ño que representan detalles al nivel de componentes en formatos gráficos, o de texto.

La programación estructurada es una filosofía de diseño procedimental que restringe el número y tipo de construcciones lógicas para representar el detalle del algoritmo. El objetivo de la programación estructurada es ayudar al diseñador a definir algoritmos que sean menos complejos y, por tanto, más fáciles de leer, probar y mantener

REFERENCIAS

[AMB02] Ambler, S., "UML Component Diagramming Guidelines", disponible en http://www.modelingstyle.info/, 2002.

[BEN02] Bennett, S., S. McRobb y R. Farmer, *Object-Oriented Analysis and Design*, 2a. ed., McGraw-Hill, 2002

[BOH66] Bohm, C. y G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", en *CACM*, vol. 9, núm. 5, mayo de 1966, pp. 366-371.

[CAI75] Caine, S. y K. Gordon, "PDL—A Tool for Software Design", en *Proc. National Computer Conference*, AFIPS Press, 1975, pp. 271-276.

[DIJ65] Dijkstra, E., "Programming Considered as a Human Activity", en *Proc. 1965 IFIP Congress*, North-Holland Publishing Co., 1965

[DIJ72] Dijkstra, E., "The Humble Programmer", 1972 ACM Turing Award Lecture, *CACM*, vol. 15, núm. 10, octubre de 1972, pp. 859-866.

[DIJ76] Dijkstra, E., "Structured Programming", en *Software Engineering, Concepts and Techniques* (J. Buxton *et al*., eds.), Van Nostrand-Reinhold, 1976

[HUR83] Hurley, R. B., *Decision Tables in Software Engineering*, Van Nostrand-Reinhold, 1983.

[LET01] Lethbridge, T. y R. Laganiere, *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, McGraw-Hill, 2001

[LIS88] Liskov, B., "Data Abstraction and Hierarchy", en *SIGPLAN Notices*, vol. 23, núm. 5, mayo de 1988.

[MAR00] Martin, R., "Design Principles and Design Patterns", descargado de http://www.objectmentor.com, 2000

[OMG01] *OMG Unified Modeling Specification*, Object Management Group, version 1.4, septiembre de 2001.

[WAR98] Warmer, J. y A. Klepp, *Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

PROBLEMAS Y PUNTOS A CONSIDERAR

11.1. El término *componente* suele ser difícil de definir. Primero proporciónese una definición genérica y luego definiciones más explícitas para software orientado a objetos y convencional. Por último, elíjanse tres lenguajes de programación con los que se esté familiarizado e ilústrese la manera en que cada uno define un componente.

11.2. ¿Por qué son necesarios los componentes de control en el software convencional y no lo son en el orientado a objetos?

11.3. Describase el paradigma orientado a objetos mediante argumentos propios. ¿Por qué es importante crear abstracciones que sirvan como interfaz entre componentes?

11.4. Describase el DIP mediante argumentos propios. ¿Qué pasaría si un diseñador depende excesivamente de las concreciones?

11.5. Selecciónense tres componentes que se hayan desarrollado recientemente y evalúense los tipos de cohesión de cada uno. Si se tuviera que definir el principal beneficio de una cohesión elevada, ¿cuál sería?

11.6. Selecciónense tres componentes que se hayan desarrollado recientemente y evalúense los tipos de acoplamiento de cada uno. Si tuviera que definir el principal beneficio de un acoplamiento elevado, ¿cuál sería?

11.7. ¿Es razonable decir que los componentes del dominio del problema nunca deben mostrar acoplamiento externo? Si se está de acuerdo, ¿cuáles tipos de componente mostrarían este tipo de acoplamiento?

11.8. Investíguese y desarróllese una lista de categorías típicas para los componentes de infraestructura

11.9. ¿Qué es una condición guardia y cuándo se usa?

11.10. ¿Cuál es el papel de las interfaces en un diseño al nivel de componentes basado en clases?

11.11. Los términos *atributos públicos* y *privados* suelen usarse en trabajo de diseño al nivel de componentes. ¿Qué significa cada uno y cuáles conceptos de diseño tratan de imponer?

11.12. ¿Qué es una fuente de datos persistentes?

11.13. Desarróllese 1) una clase de diseño elaborada; 2) descripciones de interfaz; 3) un diagrama de actividad para una de las operaciones dentro de la clase; 4) un diagrama de estado detallado para una de las clases de *HogarSeguro* que se han analizado en capítulos anteriores.

11.14. ¿Es lo mismo refinamiento por pasos que factorización? Si no, ¿cuáles son sus diferencias?

11.15. Investíguese un poco y describanse tres o cuatro construcciones OCL u operadores que no se hayan analizado en la sección 11.4.

11.16. Selecciónese una pequeña parte de un programa existente (de unas 50 a 75 líneas de código). Aíslense las construcciones de programación estructurada dibujando cajas alrededor de ellas en el código fuente. ¿El extracto del programa tiene construcciones que violan la filosofía de programación estructurada? De ser así, rediséñese el código para que se amolde a las construcciones de programación estructurada. De lo contrario, ¿qué nota en los recuadros que está dibujando?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Los principios de diseño, los conceptos, las líneas generales y las técnicas para clases y componentes orientados a objetos se revisan en muchos libros sobre ingeniería de software, análisis y diseño orientados a objetos. Entre las muchas fuentes de información se encuentran Bennett y sus colegas [BEN02], Larman (*Applying UML and Patterns*, Prentice-Hall, 2001), Rutledge y Laganiere [LET01] y Nicola y sus colegas (*Streamlined Object Modeling. Patterns, and Implementation*, Prentice-Hall, 2001). Schach (*Object-Oriented and Classical software Engineering*, quinta edición, McGraw-Hill, 2001), Dennis y sus colegas (Systems Analysis and Design An Object Oriented Approach with UML, Wiley, 2001), Graham (*Object-Oriented Methods: Principles and Practice*, Addison-Wesley, 2000), Richter (*Designing Flexible Object-Oriented Systems with UML*, Macmillan, 1999), Stevens y Pooley (*Using UML: Software Engineering with Objects and Components*, edición revisada, Addison-Wesley, 1999) y Riel (*Object-Oriented Design Heuristics*, Addison-Wesley, 1996).

El concepto de diseño por contrato es un útil paradigma de diseño. Libros de Mitchell y Kim (*Design by Contract by Example*, Addison-Wesley, 2001) y Jezequel y sus colegas (*Patterns and Contracts*, Addison-Wesley, 1999) analizan este tema en forma detallada. Metsker (*Design Patterns Java Workbook*, Addison-Wesley, 2002) y Shalloway y Trott (*Design Patterns Explained: A New Perspective on Object Oriented Design*, Addison-Wesley, 2001) toman en cuenta el impacto de los patrones en el diseño de componentes de software. La iteración de diseño es esencial para la creación de diseños de alta calidad. Fowler (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) proporciona una guía útil que puede aplicarse a medida que evoluciona el diseño.

El trabajo de Linger, Milis y Witt (*Structured Programming -Theory and Practice*, Addison-Wesley, 1979) sigue siendo un tratado definitivo sobre el tema. El texto contiene un análisis además de explicaciones detalladas de las ramificaciones de la programación e

Otros libros que se concentran en los temas de diseño procedimental para sistemas tradicionales son los de Robertson (*Simple Program Design*, tercera edición, Course Technology, 2000), Farrell (*A Guide to Programming Logic and Design*, Course Technology, 1999), Bentley (*Programming Pearls*, 2a. edición, Addison-Wesley, 1999) y Dahl (*Structured Programming*, Academic Press, 1997).

Relativamente, pocos libros recientes se han dedicado en exclusiva al diseño al nivel de componentes. En general, los libros de lenguaje de programación atienden el diseño procedimental con algún detalle, pero siempre en el contexto del lenguaje que se introduce en el libro. Hay disponibles cientos de títulos.

Una amplia variedad de fuentes de información sobre diseño al nivel de componentes se encuentra en Internet. Una lista actualizada de referencias en World Wide Web que resultan relevantes para el diseño al nivel de componentes se encuentra en el sitio Web de SEPA: **http:/ /www.mhhe.com/pressman**

CAPÍTULO

12 DISEÑO DE LA INTERFAZ DE USUARIO

CONCEPTOS CLAVE

El plano de una casa (su diseño arquitectónico) no estaría completo sin representación de puertas, ventanas y conexiones de agua, electricidad léfono (sin mencionar la televisión por cable) Las "puertas, ventanas nexiones" del software de computación integran el diseño de la interfa usuario.

El diseño de la interfaz se concentra en tres áreas: 1) el diseño de inter entre componentes de software; 2) el diseño de interfases entre el softv otros productores y consumidores de información que no son humanos cir, otras entidades externas), y 3) el diseño de la interfaz entre un ser h (es decir, el usuario) y la computadora. Este capítulo se concentrará exc mente en la tercera categoría de *diseño de la interfaz: la del usuario*

En el prólogo de su libro clásico acerca del diseño de interfaces de us Ben Shneiderman [SHN90] afirma:

> La frustración y ansiedad son parte de la vida diaria de muchos usuarios de sist de información computarizados Luchan por aprender el lenguaje de comandos sistemas de selección de menús que presuntamente deben ayudarles a realizar su bajo. Algunas personas se encuentran con casos tan serios de choque de comput ra, terror terminal o neurosis de red, que evitan el empleo de sistemas de cómp

Los problemas que refiere Shneiderman son reales. Es cierto que las ces gráficas de usuario, las ventanas, los iconos y las selecciones hecha ratón han eliminado gran parte de los más terribles problemas relacionad las interfases. Pero aun en un "mundo de ventanas", todo mundo ha enc

UN VISTAZO RÁPIDO

¿Qué es? El diseño de la interfaz de usuario crea un medio de comunicación efectiva entre un ser humano y una computadora. Siguiendo un conjunto de principios de diseño de interfaces, el diseñador identifica los objetos y las acciones de la interfaz y luego crea un formato de pantalla que forma la base de un prototipo de interfaz de usuario

¿Quién lo hace? Un ingeniero de software diseña la interfaz de usuario al aplicar un proceso iterativo basado en principios de diseño ampliamente aceptados.

¿Por qué es importante? Si el uso del software es difícil, si lleva al usuario a cometer errores o si frustra sus esfuerzos por alcanzar sus obje no le gustará, sin importar su capacidad funciones que ofrezca. La interfaz tiene qu correcta porque moldea la percepción de rio acerca del software.

¿Cuáles son los pasos? El diseño de la inter usuario empieza con la identificación requisitos de éste, la tarea y el ambient vez identificadas las tareas del usuario, se y analizan los escenarios de éste para de conjunto de objetos y acciones para la Esto constituye la base para la creación matos de pantalla que representan el dise fico y la ubicación de los iconos; la defin texto descriptivo en pantalla; la especific

signación de nombres a las ventanas, además de la especificación de los elementos principales y secundarios de los menús. Se recurre a herramientas para crear prototipos y finalmente implementar el modelo de diseño; por último, se evalúa la calidad del resultado.

¿Cuál es el producto obtenido? Se crean los escenarios del usuario y se genera el formato de pantalla. Se desarrolla un prototipo de la interfaz y se modifica de manera interactiva.

¿Cómo puedo estar seguro de que lo he hecho correctamente? El usuario realiza una "prueba de manejo" del prototipo. La información que proporciona esta prueba se emplea para la siguiente modificación iterativa del prototipo.

interfases de usuario difíciles de aprender y usar, confusas, poco intuitivas, imperdonables y, en muchos casos, totalmente frustrantes. Sin embargo, alguien dedicó tiempo y energía a la construcción de tales interfaces, y es improbable que el constructor haya generado estos problemas a propósito.

El diseño de la interfaz de usuario requiere el estudio de las personas y el conocimiento tecnológico adecuado. ¿Quién es el usuario? ¿Cómo aprende a interactuar con un nuevo sistema de cómputo? ¿Cómo interpreta la información que produce el sistema? ¿Qué espera del sistema? Éstas son sólo algunas de las muchas preguntas que deben plantearse y responderse como parte del diseño de la interfaz de usuario.

12.1 Las reglas de oro

En su libro sobre el diseño de interfaces, Theo Mantel [MAN97] acuñó tres "reglas de oro" para el diseño de la interfaz:

1. Dar el control al usuario
2. Reducir la carga en la memoria del usuario
3. Lograr que la interfaz sea consistente.

Estas reglas de oro forman la base de un conjunto de principios de diseño de interfases de usuario que servirán de guía en esta importante acción de diseño del software.

12.1.1 Dar el control al usuario

Se le preguntó a un usuario clave, durante la sesión de acopio de requisitos para un nuevo e importante sistema de información, acerca de los atributos de la interfaz gráfica orientada a ventanas. "Lo que en verdad me gustaría", dijo el usuario solemnemente, "es un sistema que me lea la mente. Que sepa lo que quiero hacer antes de que deba hacerlo y que me permita hacerlo fácilmente. Eso es todo, y nada más".

Mi primera reacción fue mover la cabeza y sonreír, pero me detuve por un momento. No había absolutamente nada de malo en la solicitud del usuario. Quería un sistema que reaccionara a sus necesidades y que le ayudara a hacer las cosas. Quería controlar la computadora; no que ésta lo controlara.

La mayor parte de las restricciones y limitaciones que impone el diseñador a la interfaz pretenden simplificar el modo de interacción. ¿Para quienes? En muchos ca-

sos, el diseñador introduce limitaciones y restricciones para simplificar la implementación de la interfaz. Así, tal vez se tenga como resultado una interfaz fácil de construir, pero cuyo uso resulta frustrante.

Mandel [MAN97] define varios principios de diseño que permiten al usuario mantener el control:

Definir los modos de interacción de forma que el usuario no realice acciones innecesarias o indeseables. Un modo de interacción es el estado actual de la interfaz. Por ejemplo, si se elige *corrector ortográfico* en un menú de un procesador de palabras, el software pasa a un modo corrector ortográfico. No hay ninguna razón para obligar al usuario a que permanezca en este modo si desea editar el texto. Debe darse al usuario la opción de entrar y salir de él sin esfuerzo

Proporcionar una interacción flexible. Debido a que diferentes usuarios tienen distintas preferencias de interacción, deben ofrecerse las opciones correspondientes. Por ejemplo, tal vez el software le permita al usuario interactuar mediante movimiento del ratón, un lápiz digitalizador o comandos seleccionados con el teclado, mediante reconocimiento de voz. Pero no todas las acciones son adecuadas para todos los mecanismos de interacción. Por ejemplo, imagine la dificultad de utilizar comandos seleccionados con el teclado (o entrada de voz) para dibujar una forma compleja

Incluir las opciones de interrumpir y deshacer la interacción del usuario. Aunque se encuentre en medio de una secuencia de acciones, un usuario debe contar con la opción de interrumpir la secuencia para hacer otra cosa (sin perder el trabajo que haya hecho). También debe contar con la opción de "deshacer" cualquier acción

Depurar la interacción a medida que aumentan los grados de destreza y permitir que se personalice la interacción. Con frecuencia, los usuarios terminan repitiendo la misma secuencia de interacciones. Vale la pena diseñar un mecanismo de "macro" que permita a un usuario personalizar la interfaz para facilitar la interacción

Oculte al usuario ocasional los elementos técnicos internos. La interfaz debe llevar al usuario al mundo virtual de la aplicación; no es necesario que conozca el sistema operativo, las funciones de administración de archivos u otros secretos de la tecnología de cómputo. En esencia, la interfaz nunca debe requerir que el usuario interactúe en el nivel "interno" del equipo (por ejemplo, nunca debe pedirse que el usuario escriba comandos del sistema operativo desde el interior del software de aplicación).

Diseñar interacción directa con los objetos que aparecen en la pantalla. El usuario siente que tiene el control cuando manipula los objetos necesarios para realizar una tarea de manera parecida a como lo haría con un objeto material. Por ejemplo, una interfaz de aplicación que permita al usuario "alargar" un objeto (cambiar su tamaño) es una implementación de manipulación directa.

"Siempre he deseado que mi computadora sea tan fácil de manejar como mi teléfono. Mi deseo se ha vuelto realidad. Ya no sé cómo usar mi teléfono."

Bjarne Stroustrup (creador de C++)

12.1.2 Reducir la carga en la memoria del usuario

Cuantas más cosas tenga que recordar un usuario, más probabilidades habrá de que cometa errores al interactuar con el sistema. Por ello, una interfaz de usuario bien diseñada no dependerá de la memoria de éste. Siempre que sea posible, el sistema debe "recordar" la información pertinente y ayudar al usuario con un escenario de interacción que le facilite el uso de la memoria. Mandel [MAN97] define los principios de diseño que logran que una interfaz reduzca la carga de memoria que recae en el usuario:

Reducir la demanda de memoria a corto plazo. Cuando los usuarios participan en tareas complejas, la demanda de memoria a corto plazo se torna importante. La interfaz se debe diseñar para que reduzca la necesidad de recordar acciones y resultados anteriores. Esto se logra al proporcionar pistas visuales que permitan al usuario reconocer acciones anteriores sin tener que recordarlas.

Definir valores por defecto que tengan significado. El conjunto inicial de valores por defecto debe tener un sentido para el usuario promedio, pero también contar con la posibilidad de especificar sus preferencias. Sin embargo, debe disponer de una opción "restablecer" que le permita volver a definir los valores por defecto originales.

Definir accesos directos intuitivos. Cuando se emplea la mnemotécnica para aplicar una función del sistema (por ejemplo, alt-I para solicitar la función de imprimir), debe estar unida a una acción de manera tal que resulte fácil de recordar (como la primera letra de la tarea que se solicita).

El formato visual de la interfaz debe basarse en una metáfora tomada de la realidad. Por ejemplo, en un sistema de pago de facturas se debe utilizar la metáfora de la chequera y el talonario de cheques para llevar al usuario a recorrer el proceso del pago de facturas. Esto permite que el usuario dependa de pistas visuales que comprende bien, en lugar de memorizar una misteriosa secuencia de interacciones.

Desglosar la información de manera progresiva. La interfaz debe organizarse jerárquicamente. Es decir, la información sobre una tarea, un objeto o algún comportamiento debe presentarse primero en un grado alto de abstracción. Después de que el usuario se interese por seleccionar algo con el ratón, deben presentarse más detalles. Un ejemplo común en muchas aplicaciones de procesamiento de palabras es la función de subrayado. Se trata de una entre varias funciones ubicadas en el menú *estilo de texto*. Sin embargo, no aparecen todas las posibilidades de subrayado. El usuario debe seleccionar subrayado para que se presenten a continuación todas las opciones disponibles (como subrayado sencillo, doble, de guiones, etc.).

HOGARSEGURO

Violación de una "regla de oro" de la interfaz de usuario

La escena: Cubículo de Vinod, cuando empieza el diseño de la interfaz de usuario.

Los actores: Vinod y Jamie, integrantes del equipo de ingeniería de software de *HogarSeguro*

La conversación:

Jamie: He estado pensando en la interfaz de la función de vigilancia

Vinod (sonriendo): Pensar es bueno

Jamie: Creo que podemos simplificar un poco las cosas

Vinod: Lo que significa que

Jamie: Bueno, ¿qué pasaría si eliminamos por completo el plano de la casa? Es ostentosa, pero requiere muchos esfuerzos de desarrollo. En lugar de eso, pidamos al usuario que especifique la cámara que desea ver y luego despleguemos el video en una ventana de video.

Vinod: ¿Cómo recordará el propietario cuántas cámaras están funcionando y dónde se encuentran?

Jamie (un poco irritada): Él es el dueño, debe saberlo

Vinod: ¿Pero qué pasaría si no?

Jamie: Debe saberlo

Vinod: Eso no es lo importante .. ¿qué tal si la olvida?

Jamie: Oh, podemos proporcionarle una lista de las cámaras en operación y el lugar en que se encuentran

Vinod: Eso es posible, pero ¿por qué habría de pedir una lista?

Jamie: Muy bien, proporcionemos la lista, la pida o no

Vinod: Eso está mejor. Por lo menos no tendrá que recordar cosas que podemos darle

Jamie (pensando por un momento): Pero a ti te gusta el plano, ¿o no?

Vinod: Ajá.

Jamie: ¿Cuál crees que le gustaría a mercadotecnia?

Vinod: Estás bromeando, ¿verdad?

Jamie: No.

Vinod: Uh.. el que tiene el flash . les encantan las funciones atractivas en los productos a ellos no les interesa cuál es más fácil de construir.

Jamie (suspirando): Muy bien, tal vez haré un prototipo de ambos

Vinod: Buena idea... luego dejaremos que el cliente decida

12.1.3 Lograr que la interfaz sea consistente

La interfaz debe adquirir y presentar la información de manera consistente. Esto implica que 1) toda la información visual esté organizada de acuerdo con un estándar de diseño que se mantenga en todas las presentaciones de pantalla; 2) los mecanismos de entrada se restrinjan a un conjunto limitado que se utilice de manera consistente en toda la aplicación, y 3) los mecanismos para ir de una tarea a otra se hayan definido e implementado de manera consistente. Mandel [MAN97] define un conjunto de principios de diseño que ayudan a construir una interfaz consistente:

> "Las cosas que tienen aspectos diferentes deben actuar de manera distinta. Las que tienen el mismo aspecto, deben actuar igual."
>
> **Larry Mandel**

Permitir que el usuario incluya la tarea actual en un contexto que tenga algún significado. Muchas interfaces implementan capas complejas de interacciones con docenas de imágenes en pantalla. Es importante proporcionar indicadores (por ejemplo, títulos de ventana, íconos gráficos, códigos de color consistentes

permitan al usuario conocer el contexto del trabajo que realiza. Además, el usuario debe tener la capacidad de determinar de dónde viene y cuáles son sus opciones para la transición a una nueva tarea.

Mantener la consistencia en toda una familia de aplicaciones. Un conjunto de aplicaciones (o productos) debe implementar las mismas reglas de diseño para mantener la consistencia en todas las interacciones.

Si modelos interactivos anteriores han generado expectativas en el usuario, no hacer cambios a menos que haya razones inexcusables. Una vez que una secuencia interactiva se ha convertido en un estándar *de facto* (como el empleo de alt-G para guardar un archivo), el usuario espera esto en todas las aplicaciones que encuentre. Un cambio (como el uso de alt-G para solicitar la función cambiar de tamaño) crearía confusión.

Los principios del diseño de interfases expuestos aquí y en secciones anteriores proporcionan una guía para un ingeniero de software. En la siguiente sección se examinará el proceso de diseño de la interfaz.

INFORMACIÓN

Facilidad de uso

En un brillante ensayo sobre la facilidad de uso, Larry Constantine [CON95] plantea una pregunta que tiene una fuerte relación con el tema: "¿Al final de cuentas, qué quieren los usuarios?" Responde así: "Lo que los usuarios realmente quieren son buenas herramientas. Todos los sistemas de software, desde los sistemas operativos y los lenguajes hasta la entrada de datos y las aplicaciones de apoyo a la toma de decisiones, son solo herramientas. Los usuarios finales esperan de las herramientas que construimos para ellos lo mismo que nosotros esperamos de las herramientas que usamos [illegible] sistemas fáciles de aprender y que les ayuden a hacer su trabajo. Quieren software que no los detenga, [illegible]e o confunda, que no les lleve a cometer errores o [illegible]culte la terminación del trabajo".

Constantine argumenta que la facilidad de uso no se [illegible] de mecanismos de interacción estéticos o modernos, [illegible] inteligencia integrada en la interfaz. En cambio, [illegible] cuando la arquitectura de la interfaz corresponde a [illegible]ecesidades de las personas que la usarán.

Una definición formal de facilidad de uso es elusiva. [illegible]ue y sus colegas [DON99] la definen de la siguiente manera: "Facilidad de uso es una medida de la manera en [illegible] un sistema de cómputo... facilita el aprendizaje; [illegible] a quienes aprenden a recordar lo que han [illegible]ido, reduce la posibilidad de errores; les permite ser [illegible]tes y los deja satisfechos con el sistema".

La única manera de determinar si existe "facilidad de uso" dentro de un sistema en construcción consiste en realizar una evaluación o una prueba de uso. Obsérvese a los usuarios interactuando con el sistema y respóndanse las siguientes preguntas [CON95]:

- ¿Es posible usar el sistema sin ayuda ni enseñanza continua?
- ¿Las reglas de interacción ayudan a un usuario conocedor a trabajar con eficiencia?
- ¿Los mecanismos de interacción se vuelven más flexibles a medida que los usuarios adquieren más conocimientos?
- ¿El sistema se ha adecuado al entorno físico y social en que habrá de usarse?
- ¿El usuario está al tanto del estado del sistema? ¿El usuario sabe dónde se encuentra en cada momento?
- ¿La interfaz está estructurada de manera lógica y consistente?
- ¿Los mecanismos de interacción, iconos y procedimientos son consistentes en toda la interfaz?
- ¿La interacción anticipa errores y ayuda al usuario a corregirlos?
- ¿La interfaz tolera los errores que se cometen?
- ¿La interacción es simple?

Si se responde afirmativamente a cada una de estas preguntas, es probable que se haya alcanzado la facilidad de uso.

Entre los numerosos beneficios mensurables derivados de un sistema con facilidad de uso se encuentran [DON99]: mayor nivel de ventas y satisfacción del usuario, ventaja competitiva, mejores reseñas en los medios, mejores recomendaciones de boca en boca, costos de soporte reducidos, mayor productividad del usuario final, menores costos de capacitación y documentación, además de menos probabilidades de que los usuarios insatisfechos entablen demandas.

12.2 Análisis y diseño de la interfaz de usuario

El proceso general para analizar y diseñar una interfaz de usuario empieza con la creación de diferentes modelos de función del sistema (como se percibe desde el exterior). Luego se delinean las tareas orientadas al ser humano y el equipo que se requieren para lograr el funcionamiento del sistema; se toman en cuenta los temas de diseño que se aplican a todos los diseños de interfaces; se emplean herramientas para crear prototipos e implantar finalmente el modelo de diseño, y los usuarios finales evalúan la calidad del resultado.

Referencia Web

Una excelente fuente de información sobre diseño de interfaces de usuario se encuentra en **www.useit.com.**

12.2.1 Modelos del análisis y diseño de la interfaz

Cuando se analiza y diseña una interfaz de usuario entran en juego cuatro modelos diferentes. Un ingeniero humano (o el ingeniero del software) establece un *modelo del usuario*; el ingeniero del software crea un *modelo del diseño*; el usuario final desarrolla una imagen mental que suele denominarse *modelo mental* del usuario o *percepción del sistema*, y quienes implementan el sistema crean un *modelo de la implementación*. Por desgracia, es posible que estos modelos difieran significativamente entre sí. El papel del diseñador de la interfaz es reconciliar estas diferencias y derivar una representación consistente de la interfaz.

> "Si hay que usar algún 'truco', la interfaz de usuario no es consistente."
>
> **Douglas Anderson**

CONSEJO

Hasta un usuario principiante quiere accesos directos; hasta los usuarios esporádicos y con conocimiento suelen necesitar una guía. Hay que darles lo que necesiten

El modelo del usuario establece el perfil de los usuarios finales del sistema. Para construir una interfaz de usuario efectiva, "todo diseño debe empezar por la comprensión de quiénes son los usuarios de destino, incluidos sus perfiles de edad y sexo, habilidades físicas, educación, antecedentes culturales o étnicos, motivaciones, objetivos y personalidad" [SCH90]. Además, es posible distribuir a los usuarios en las siguientes categorías:

Principiantes. No tienen conocimientos de la sintaxis[1] del sistema y cuentan con escasos conocimientos[2] de la semántica de la aplicación o del uso de la computadora en general.

1 En este contexto, *conocimiento de sintaxis* alude a los mecanismos de interacción requeridos para usar la interfaz de manera efectiva.

2 *Conocimiento de semántica* alude al sentido inherente de la aplicación (una comprensión de las operaciones que se realizan, el significado de entrada y salida, y las metas y los objetivos del sistema).

Usuarios esporádicos y con conocimientos. Tienen conocimientos razonables de semántica, pero muestran una retención relativamente baja de la información sobre sintaxis necesaria para utilizar la interfaz.

Usuarios frecuentes y con conocimientos. Cuentan con conocimientos de sintaxis y semántica suficientes para llegar al "síndrome del usuario avanzado" (es decir, individuos que buscan combinaciones de teclas y métodos abreviados para interactuar).

Un modelo del diseño de todo el sistema incorpora datos, arquitectura, interfaz y representaciones procedimentales del software. La especificación de requisitos establece ciertas restricciones que ayudan a definir el usuario del sistema, pero el diseño de la interfaz sólo suele ser incidental en relación con el modelo del diseño.[3]

El modelo mental del usuario (percepción del sistema) es la imagen del sistema que los usuarios finales llevan en la mente. Por ejemplo, si se pidiera al usuario de un sistema determinado de diseño de páginas que describiera la operación, la percepción del sistema determinaría la respuesta. La precisión de la descripción dependerá del perfil del usuario (por ejemplo, los principiantes darían cuando mucho una respuesta incompleta) y de la familiaridad general con el software en el dominio de la aplicación. Un usuario que comprenda por completo el diseño de páginas, pero que haya trabajado con el sistema una sola vez en realidad podría proporcionar una descripción más completa de su funcionamiento que el principiante que ha pasado semanas tratando de aprender el sistema.

PUNTO CLAVE

El modelo mental evalúa la manera en que el usuario percibe la interfaz y si ésta satisface sus necesidades.

> "[P]reste atención a lo que hacen los usuarios, no a lo que dicen."
>
> **Jakob Nielsen**

El modelo de la implementación combina la manifestación externa del sistema de cómputo (la apariencia de la interfaz) y toda la información de ayuda (libros, manuales, videocintas, archivos de ayuda) que describe la sintaxis y semántica del sistema. Cuando coinciden el modelo de la implementación y el modelo mental del usuario, los usuarios suelen sentirse a gusto con el software y lo usan con efectividad. Para lograr esta "combinación" de los modelos, el modelo del diseño debió desarrollarse para incluir la información del modelo del usuario, y el modelo de implementación debe reflejar con exactitud la información sintáctica y semántica de la interfaz.

Los modelos descritos en esta sección son "abstracciones de lo que el usuario está haciendo o lo que piensa que está haciendo o lo que alguien más piensa que debería estar haciendo cuando usa el sistema interactivo". [MON84]. En esencia, estos modelos permiten que el diseñador de la interfaz satisfaga un elemento clave del principio más importante del diseño de la interfaz de usuario: *Conoce al usuario y sus tareas.*

3 Así no es como deben ser las cosas. En muchos casos, el diseño de la interfaz es tan importante como el diseño arquitectónico y el nivel de componentes.

12.2.2 El proceso

El proceso de análisis y diseño de las interfaces de usuario es iterativo y se representa con un modelo espiral parecido al que se analizó en el capítulo 3. Tomando como referencia la figura 12.1, se observará que el proceso de análisis y diseño de la interfaz de usuario abarca cuatro actividades distintas de marco de trabajo [MAN97]:

1. Análisis y modelado de usuarios, tareas y entornos.
2. Diseño de la interfaz
3. Construcción (implementación) de la interfaz.
4. Validación de la interfaz

La espiral que se muestra en la figura 12.1 indica que cada una de estas tareas ocurrirá más de una vez, y cada pasada por la espiral representa la elaboración adicional de los requisitos y el diseño resultante. En casi todos los casos, la actividad de construcción incluye la creación de prototipos (la única manera práctica de validar lo que se ha diseñado).

El análisis de la interfaz se concentra en el perfil de los usuarios que interactuarán con el sistema. Se registrarán el grado de habilidad, la comprensión del trabajo y la disposición general ante el nuevo sistema; y se definirán diferentes categorías de usuarios.

> "Es mejor diseñar la experiencia del usuario que rectificarla."
>
> Jon Meads

Una vez definidos los requisitos generales se realiza un análisis más detallado de las tareas. Se identifican, describen y elaboran las tareas que el usuario realiza y

FIGURA 12.1 El proceso de diseño de la interfaz de usuario.

alcanzar los objetivos del sistema (sobre un número de pasadas iterativas por la espiral). El análisis de tareas se expone de manera más detallada en la sección 12.3

El análisis del entorno del usuario se concentra en el ambiente físico de trabajo Entre las preguntas que deben responderse están las siguientes

¿Qué necesito saber del entorno cuando empiece a diseñar la interfaz de usuario?

- ¿Dónde se localizará físicamente la interfaz?
- ¿El usuario estará sentado, de pie o realizará otras tareas sin relación con la interfaz?
- ¿El hardware de la interfaz tiene restricciones de espacio o iluminación, o lo afecta el ruido?
- ¿Hay factores humanos especiales determinados por factores ambientales?

La información reunida como parte de la actividad de análisis se utiliza para crear un modelo del análisis para la interfaz. Tomando este modelo como base, se inicia la actividad de diseño.

El objetivo del diseño de la interfaz es definir un conjunto de objetos y acciones (y sus representaciones en pantalla) que permitan que el usuario realice todas las tareas definidas, de manera tal que cumplan todos los objetivos de facilidad de uso que define el sistema. El diseño de la interfaz se estudia con mayor detalle en la sección 12.4.

Por lo general, la actividad de construcción empieza al crear un prototipo que permita evaluar los escenarios de uso. A medida que continúa el proceso de diseño iterativo, pueden usarse herramientas de desarrollo de la interfaz de usuario (consulte el recuadro de la sección 12.4) para completar la construcción de la interfaz

La validación se concentra en 1) la capacidad de la interfaz para implementar correctamente todas las tareas del usuario, acomodar todas las variaciones de las tareas y cumplir todos los requisitos generales del usuario; 2) la facilidad del uso y el aprendizaje de la interfaz, y 3) la aceptación por el usuario de que la interfaz es una herramienta útil para su trabajo.

Como ya se ha observado, las actividades descritas en esta sección ocurren de manera iterativa. Por tanto, es innecesario tratar de especificar cada detalle (para el modelo de análisis o de diseño) en el primer paso. Los siguientes pasos del proceso dan lugar al detalle de la tarea, la información del diseño y las características operativas de la interfaz.

12.3 ANÁLISIS DE LA INTERFAZ[4]

Un principio clave de todos los modelos de procesos de ingeniería del software reza: *es mejor comprender el problema antes de tratar de diseñar una solución*. En el ca-

4 Resulta razonable argumentar que esta sección debió colocarse en el capítulo 8, porque allí se estudia el análisis de requisitos Se ha incluido aquí porque el análisis y el diseño de la interfaz están íntimamente relacionados, y porque el límite entre ambos es muy difuso

so del diseño de la interfaz de usuario, comprender el problema significa comprender 1) a las personas (los usuarios finales) que interactuarán con el sistema por medio de la interfaz; 2) las tareas que los usuarios finales deben realizar para hacer su trabajo; 3) el contenido que se presenta como parte de la interfaz, y 4) el entorno en que se realizarán estas tareas. En las secciones siguientes se examinará cada uno de estos elementos del análisis de la interfaz con el fin de establecer una base sólida para el diseño de las tareas que siguen.

12.3.1 Análisis del usuario

Ya se ha indicado que cada usuario tiene una imagen mental del software (o percepción del sistema) que podría ser diferente de la imagen mental de otros. Además, es probable que la imagen mental del usuario sea muy diferente del modelo del diseño que tiene el ingeniero. El diseñador sólo lograría que coincidan la imagen mental y el modelo del diseño si trabaja para comprender a los propios usuarios, además de la manera en que ellos usarán el sistema. Es posible usar información de una amplia variedad de fuentes disponible para lograr esto:

¿Cómo sé lo que los usuarios quieren de la interfaz?

Entrevistas con el usuario. Es el enfoque más directo. Representantes del equipo de software se entrevistan con usuarios finales para comprender mejor sus necesidades, motivaciones, cultura de trabajo y gran cantidad de temas adicionales. Esto se logra mediante reuniones personales o con grupos de enfoque.

CONSEJO

Sobre todo, dedíquese tiempo a hablar con los usuarios reales, pero debe hacerse con cuidado. Una opinión fuerte no necesariamente significa que la mayoría de los usuarios esté de acuerdo.

Información de ventas. El personal de ventas se reúne con clientes y usuarios de manera regular y obtiene información que ayudará al equipo de software a clasificar a los usuarios en categorías y a comprender mejor sus necesidades.

Información de mercadotecnia. El análisis de mercado es invaluable en la definición de los segmentos del mercado, al tiempo que proporciona una comprensión de la manera en que cada segmento usaría el software de manera sutilmente diferente.

Información proveniente de soporte. El personal de soporte habla a diario con los clientes. Esto los convierte en la fuente de información más probable sobre lo que funciona y lo que no, lo que le gusta a los usuarios y lo que les disgusta, características que generan dudas y las que resultan fáciles de usar.

El siguiente grupo de preguntas (adaptado de [HAC98]) ayudará al diseñador de la interfaz a comprender mejor a los usuarios de un sistema.

PUNTO CLAVE

¿Cómo aprendemos sobre la demografía y las características de los usuarios finales?

- ¿Los usuarios son profesionales capacitados, técnicos, trabajadores de oficina u obreros?
- ¿Qué grado de educación formal tiene el usuario promedio?
- ¿Los usuarios son capaces de aprender con materiales escritos o expresan su deseo de recibir capacitación en el lugar?
- ¿Los usuarios son expertos para tipear o le tienen fobia al teclado?
- ¿Cuál es la edad promedio de la comunidad de usuarios?

- ¿Los usuarios corresponden predominantemente a algún género?
- ¿Qué compensación reciben los usuarios por su trabajo?
- ¿Los usuarios trabajan en horas normales de oficina, o siguen sus labores hasta que hayan terminado lo que están haciendo?
- ¿El software será una parte integral del trabajo de los usuarios, o se empleará ocasionalmente?
- ¿Cuál es el idioma materno de los usuarios?
- ¿Cuáles serían las consecuencias si un usuario comete un error mientras usa el sistema?
- ¿Los usuarios son expertos en el tema que atiende el sistema?
- ¿Los usuarios quieren conocer la tecnología que sustenta la interfaz?

Las respuestas a éstas y otras preguntas similares permitirán que el diseñador comprenda quiénes son los usuarios finales, qué los motiva y complace, cómo se agruparían en diferentes clases o perfiles de usuarios, cuáles son sus modelos mentales del sistema, y cómo debe caracterizarse la interfaz de usuario para que satisfaga sus necesidades.

12.3.2 Análisis y modelado de tareas

El objetivo del análisis de la tarea es responder las siguientes preguntas:

- ¿Qué trabajo hará el usuario en circunstancias específicas?
- ¿Cuáles tareas y subtareas se realizarán mientras el usuario trabaja?
- ¿Cuáles objetos específicos del dominio del problema manipulará el usuario mientras se realiza el trabajo?
- ¿En qué secuencia se presentan tareas del trabajo (el flujo de trabajo)?
- ¿Cuál es la jerarquía de las tareas?

Para responder estas preguntas, el ingeniero de software debe basarse en las técnicas de análisis expuestas en los capítulos 7 y 8; pero en este caso las técnicas se aplican a la interfaz del usuario.

Casos de uso. En capítulos anteriores se indicó que el caso de uso describe cómo un actor (en el contexto del diseño de la interfaz, un actor siempre es una persona) interactúa con un sistema. Cuando se usa como parte del análisis de tareas, el caso de uso se desarrolla para que muestre la manera en que el usuario final realiza alguna tarea específica relacionada con el trabajo. Casi siempre, el caso de uso se escribe de manera informal (un solo párrafo) en primera persona. Por ejemplo, supóngase que una pequeña empresa de software quiere construir un sistema de diseño asistido por computadora explícitamente para diseñadores de interiores. Para comprender mejor cómo hacen su trabajo, se pide a diseñadores de interiores reales que describan funciones específicas del diseño. Cuando se les pregunta: "¿cómo decide

el lugar en que colocará un mueble en una habitación?", un diseñador de interiores escribe el siguiente caso de uso informal:

> Empiezo haciendo un borrador del plano de la habitación, las dimensiones y la posición de las ventanas y puertas. Me preocupo por la manera en que entra la luz, por la vista que se tiene a través de las ventanas (si es hermosa, me gustaría llamar la atención hacia ella por el espacio que las paredes no tapan, por el flujo del movimiento en la habitación. Luego observo la lista de muebles que mi cliente y yo hemos elegido (mesas, sillas, sillones, vitrinas) y la lista de accesorios (lámparas, alfombras, cuadros, esculturas, plantas, pequeñas piezas), y también observo mis notas sobre la manera en que mi cliente desea que se distribuyan. Luego dibujo cada elemento de mis listas empleando una plantilla que se ha dibujado a escala con el plano. Etiqueto cada elemento y uso lápiz porque siempre muevo cosas. Considero varias opciones de ubicación y decido cuál es la que me gusta más. Luego dibujo una representación (una imagen tridimensional) del cuarto, para dar a mi cliente una idea del aspecto que tendrá.

Este caso de uso proporciona una descripción elemental de una tarea de trabajo importante para el sistema de diseño asistido por computadora. A partir de él, el ingeniero de software extraerá tareas y objetos, además del flujo general de la interacción. Además, también es posible concebir otras características del sistema que agradarían al diseñador de interiores. Por ejemplo, podrían tomarse fotografías digitales del panorama de cada ventana. Así, cuando se haga una representación de la imagen de la habitación se mostraría la vista real en cada ventana.

HogarSeguro

Casos de uso para el diseño de interfaces de usuario

La escena: Cubículo de Vinod, mientras se continúa con el diseño de la interfaz.

Los actores: Vinod y Jamie, integrantes del equipo de ingeniería del software de HogarSeguro.

La conversación:

Jamie: Fui a ver a mi contacto de mercadotecnia e hice que escribiera un caso de uso para la interfaz de vigilancia.

Vinod: ¿Desde el punto de vista de quién?

Jamie: Del propietario de la casa, ¿de quién más?

Vinod: También existe el enfoque del administrador. Aunque el propietario desempeñe ese papel, tendrá un punto de vista diferente. El "administrador" habilita el sistema, configura las cosas, diseña el plano de la casa, coloca las cámaras.

Jamie: Lo que hice fue que mercadotecnia representara el papel de un propietario que quiere ver un video.

Vinod: Eso está bien. Es uno de los principales comportamientos de la interfaz de la función de vigilancia. Pero tendremos que examinar también el comportamiento del administrador del sistema.

Jamie (irritada): Tienes razón.

(Jamie sale a buscar a la persona de mercadotecnia. Regresa unas horas más tarde.)

Jamie: Tuve suerte. Encontré a nuestro contacto de mercadotecnia y trabajamos juntos el caso del administrador. Básicamente, definiremos "administración" como una función aplicable a todas las funciones de HogarSeguro. He aquí lo que hicimos. (Jamie muestra a Vinod el caso de uso informal.)

Caso de uso informal: Quiero la capacidad de configurar o editar el formato del sistema en cualquier momento. Cuando configure el sistema seleccionaré la función de administración. Me pregunta si quiero definir una nueva configuración. Respondo afirmativamente y

una pantalla de dibujo que me permite el plano de la casa en una cuadrícula. Habrá para paredes, ventanas y puertas que me n el dibujo. Solamente estiro los iconos para que la dimensión correcta. El sistema desplegará las pies o metros (puedo seleccionar el sistema). Tengo la opción de seleccionar sensores y de una biblioteca y de ubicarlos en el plano. poner una leyenda a cada una, pero el sistema lo puede hacer automáticamente. Tengo la de definir los parámetros de sensores y diante menús especiales. Si selecciono editar, over los sensores o las cámaras, agregar nuevos o eliminar los que ya existen, editar el plano de la casa y los valores de cámaras y sensores. En todos los casos, espero que el sistema tenga consistencia y me ayude a no cometer errores.

Vinod (después de leer el guión): Muy bien. Tal vez haya algunos patrones de diseño útiles o algunos componentes reutilizables que podamos usar en las interfaces gráficas de usuario tomados de algún programa de diseño. Te apuesto la comida a que, si los usamos, podemos implementar parte de la interfaz del administrador, o casi toda ella.

Jamie: Estoy de acuerdo, déjame revisarlo.

Elaboración de la tarea. En el capítulo 9, se analizó la elaboración paso a paso (también *denominada descomposición funcional* o *refinamiento paso a paso*) como mecanismo para refinar las tareas de procesamiento requeridas para que el software realice alguna función deseada. El análisis de la tarea para el diseño de la interfaz emplea un enfoque elaborativo para apoyar la comprensión de las actividades humanas a las que debe adecuarse la interfaz de usuario.

El análisis de la tarea se aplica de dos maneras. Como ya lo hemos indicado, un sistema interactivo, computacional, suele usarse para reemplazar una actividad manual o semiautomática. Con el fin de comprender las tareas indispensables para alcanzar el objetivo de la actividad, un ingeniero humano[5] debe comprender las tareas que las personas realizan actualmente (al usar un método manual) y luego relacionarlas con un conjunto similar (pero no necesariamente idéntico) de tareas que se implementan en el contexto de la interfaz de usuario. Como opción, el ingeniero humano puede estudiar una especificación existente para una solución computarizada y derivar un conjunto de tareas de usuario que se adecuarán al modelo de éste, el modelo del diseño y la percepción del sistema.

se ha
una tarea no
que no
manera de
y que se
a aplicar
e implemente
de usuario.

Sin importar el enfoque general para el análisis de la tarea, un ingeniero humano debe definir y clasificar primero las tareas. Ya se ha indicado que un enfoque es la elaboración paso a paso. Por ejemplo, suponga que una pequeña compañía de software pretende construir un sistema de diseño asistido por computadora explícitamente para diseñadores de interiores. Al observar cómo trabaja uno de ellos, el ingeniero nota que el diseño de interiores abarca varias actividades importantes: distribución del mobiliario (tome en cuenta el caso de uso que ya analizamos), selección de telas y materiales, selección de tapices para paredes y cortinas para ventanas, presentación (para el cliente), presupuesto y compras. Cada una de estas importantes tareas pueden desglosarse en subtareas. Por ejemplo, de acuerdo con la información contenida en

5 En muchos casos, un ingeniero de software realiza las tareas descritas en esta sección. Lo ideal es que esta persona tenga capacitación en ingeniería humana y en diseño de interfaces de usuario.

el caso de uso, la distribución del mobiliario se refinaría de las siguientes tareas: 1) dibujar un plano de la casa tomando como base las dimensiones de la habitación; 2) ubicar las ventanas y puertas en los lugares adecuados; 3a) usar las plantillas de muebles para dibujar contornos a escala en el plano; 3b) usar las plantillas de accesorios para dibujarlos a escala en el plano; 4) mover los contornos de los muebles y accesorios para obtener el mejor lugar; 5) rotular todos los contornos de muebles y accesorios; 6) dibujar las dimensiones para mostrar la ubicación; 7) generar la vista tridimensional en perspectiva para el cliente. Un enfoque similar se usaría para cada una de las otras tareas importantes.

Es posible refinar aún más de la subtareas 1 a 7. De la 1 a la 6 se realizarán al manipular la información y ejecutar acciones dentro de la interfaz de usuario. Por otra parte, el software tiene la capacidad de realizar automáticamente la subtarea 7, que representaría poca interacción directa del usuario.[6] El modelo del diseño de la interfaz debe acomodar cada una de esas tareas para que sea consistente con el modelo del usuario (el perfil de un diseñador de interiores "típico") y la percepción del sistema (lo que el diseñador de interiores espera de un sistema automatizado).

CONSEJO

Aunque la elaboración del objeto sea útil, no debe usarse como un enfoque independiente. La voz del usuario debe tomarse en cuenta durante el análisis de la tarea.

Elaboración del objeto. En lugar de concentrarse en las tareas que un usuario debe realizar, el ingeniero de software examina el caso de uso y otra información obtenida del usuario y obtiene los objetos físicos que usa el diseñador de interiores. Estos objetos pueden ordenarse en clases. Los atributos de cada clase y una evaluación de las acciones aplicadas a cada objeto proporcionan al diseñador una lista de operaciones. Por ejemplo, la plantilla de muebles se traduce en una clase llamada **Mueble** con atributos que podrían incluir ~~tamaño~~, ~~forma~~, ~~ubicación~~ y otras. El diseñador de interiores seleccionaría el objeto de la clase *Mueble*, lo *movería* a una posición en el plano (otro objeto en el contexto), *dibujaría* el contorno de los muebles, etc. Las tareas *seleccionar*, *mover* y *dibujar* son operaciones. El modelo del análisis de la interfaz de usuario no proporcionaría una implementación para cada una de estas operaciones. Sin embargo, a medida que se elabore el diseño se definirían los detalles de cada operación.

Análisis del flujo de trabajo. Cuando distintos usuarios, cada uno representando diferentes papeles, utilizan una interfaz de usuario, a veces es necesario ir más allá del análisis de la tarea y la elaboración de objetos y aplicar el *análisis del flujo de trabajo*. Esta técnica permite que un ingeniero de software comprenda cómo se realiza un proceso de trabajo cuando se involucran varias personas (y papeles) en una compañía que pretende automatizar el proceso de prescribir y enviar medicamentos que se venden con receta. Todo el proceso[7] girará alrededor de una

[6] Sin embargo, tal vez éste no sea el caso. Es probable que el diseñador de interiores quiera cambiar la perspectiva del dibujo, el tamaño o el uso del color y otra información. El caso de uso relacionado con las representaciones de dibujos en perspectiva proporcionaría la información que necesita atender en esta tarea.

[7] Este ejemplo se ha adaptado de [HAC98].

cación Web disponible para médicos (o sus asistentes), farmacéuticos y pacientes. El flujo de trabajo se representa de manera efectiva con un diagrama de línea de flotación UML (una variante del diagrama de actividad).

Sólo se considerará una pequeña parte del proceso de trabajo: la situación que se presenta cuando un paciente pide que se le resurta una receta. En la figura 12.2 se presenta un diagrama de línea de flotación que indica las tareas y decisiones de cada uno de los tres papeles citados. Esta información podría obtenerse mediante entrevistas o casos de estudio escritos por cada actor. Independientemente de esto, el flujo de los eventos (mostrados en la figura) permite que el diseñador de la interfaz reconozca tres características clave:

1. Cada usuario implementa diferentes tareas con la interfaz; por tanto, el concepto de la interfaz diseñada para el paciente será diferente del aplicado a los farmacéuticos o médicos.
2. El diseño de la interfaz para farmacéuticos y médicos debe tener acceso a la información de fuentes secundarias (como acceso a inventarios de farmacia y a la información acerca de medicamentos alternos por parte del médico), además de desplegar esta información.
3. Es posible elaborar aún más muchas actividades indicadas en el diagrama de línea de flotación mediante análisis de la tarea, elaboración de objetos, o ambas opciones (por ejemplo, *prescripción de resurtido* podría relacionarse con una entrega por correo, o una visita a la farmacia o a un centro especial de distribución de medicamentos).

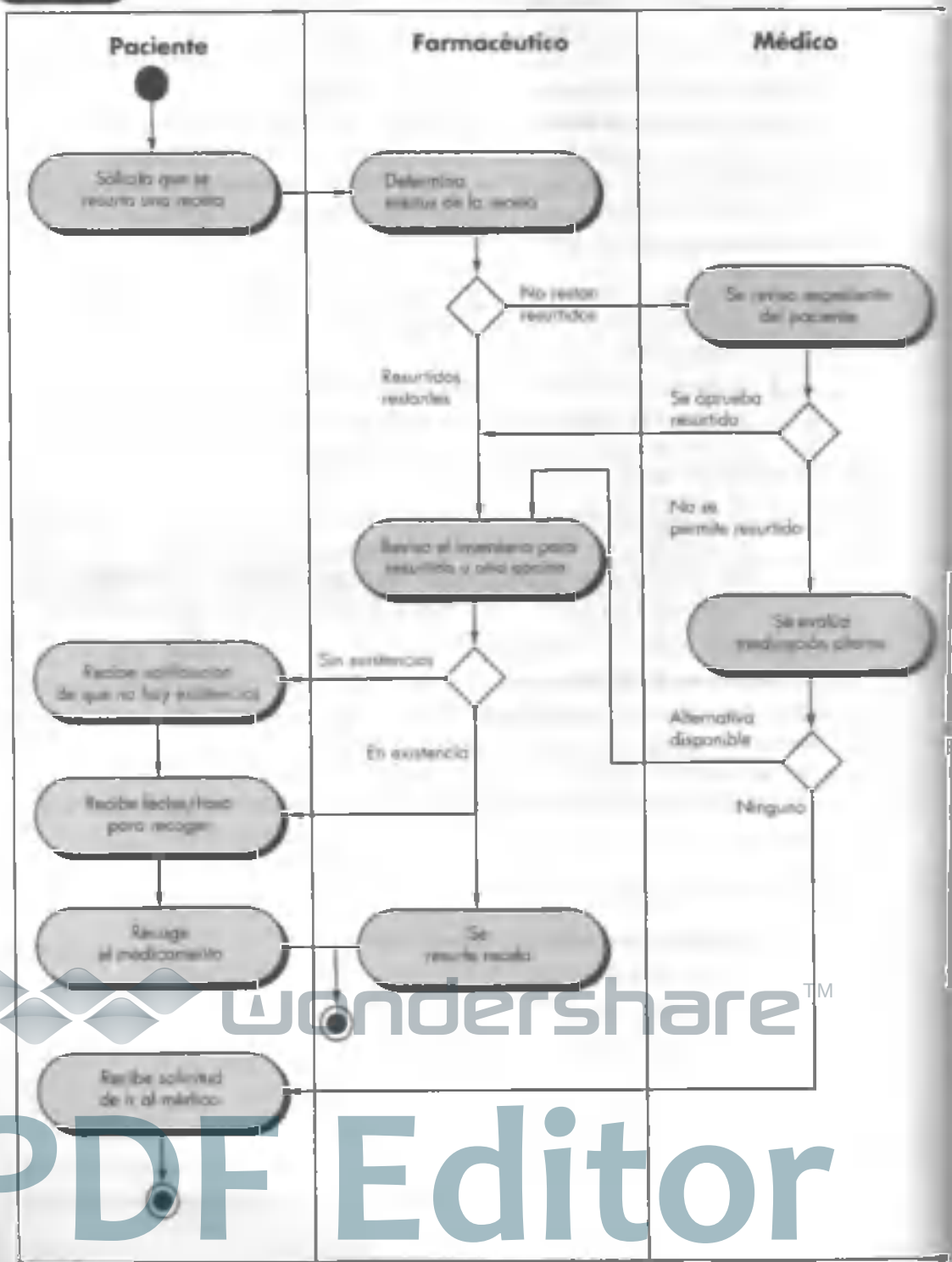
Representación jerárquica. Cuando se analiza una interfaz ocurre un proceso de elaboración. Una vez establecido el flujo de trabajo se define una jerarquía de tarea para cada tipo de usuario. La jerarquía se deriva de una elaboración paso a paso de cada tarea que el usuario haya identificado. Por ejemplo, piénsese en la tarea de usuario *solicitar que se resurta una receta*. Se desarrolla la siguiente jerarquía de tareas:

Solicitar que se resurta una receta

- *Proporcionar información de identificación*
  - *Especificar nombre*
  - *Especificar identidad de usuario*
  - *Especificar NIP y contraseña*
- *Especificar número de receta*
- *Especificar fecha en que se requiere el resurtido*

Completar la tarea *solicitar que se resurta una receta* requiere definir tres subtareas. Una de ellas, *proporcionar información de identificación*, incluye tres subtareas adicionales.

FIGURA 12.2 Diagrama de línea de flotación para la función de resurtido de recetas.



"Es mucho mejor adaptar la tecnología al usuario que obligar a éste a adaptarse a la tecnología."

Larry Marine

12.3.3 Análisis del contenido de la pantalla

Las tareas del usuario identificadas en la sección anterior llevan a la presentación de diferentes tipos de contenido. En el caso de aplicaciones modernas, el contenido de la pantalla va de informes basados en caracteres (por ejemplo, hojas de cálculo), pantallas gráficas (por ejemplo, un histograma, un modelo 3-D, la fotografía de una persona) o información especializada (como archivos de audio o video). Las técnicas de modelado del análisis estudiadas en el capítulo 8 identifican los objetos de datos de salida que produce una aplicación. Estos objetos de datos son: 1) generados por componentes (no relacionados con la interfaz) en otras partes de la aplicación; 2) adquiridas de los datos almacenados en una base de datos a la que se tiene acceso desde la aplicación, o 3) transmitida de sistemas externos a la aplicación en cuestión.

En este paso del análisis de la interfaz se consideran el formato y la estética del contenido (tal como se despliega en la interfaz). Entre las preguntas que se habrán de plantear se encuentran las siguientes:

¿Cómo se determina el formato y la estética del contenido desplegado como parte de la interfaz de usuario?

- ¿Hay diferentes tipos de datos asignados a ubicaciones consistentes en la pantalla (por ejemplo, las fotos siempre aparecen en la esquina superior derecha de la pantalla)?
- ¿El usuario puede personalizar la distribución del contenido en la pantalla?
- ¿Se ha asignado una apropiada identificación en pantalla a todo el contenido?
- ¿Cómo se segmenta un informe largo para facilitar su comprensión?
- ¿Habrá mecanismos disponibles para desplazar directamente al resumen de información en conjuntos grandes de datos?
- Se cambiará el tamaño de la salida gráfica para que quepa dentro de los límites de la pantalla o el monitor que habrá de usarse?
- ¿Cómo se usará el color para mejorar la comprensión?
- ¿Cómo se presentarán al usuario los mensajes de error y los avisos de precaución?

A medida que se responden estas (y otras) preguntas se establecerán los requisitos para la presentación del contenido.

12.3.4 Análisis del entorno de trabajo

Hackos y Redish [HAC98] analizan la importancia del análisis del entorno de trabajo cuando afirman:

> La gente no trabaja aislada, la afectan la actividad que se realiza a su alrededor, las características físicas del lugar de trabajo, el tipo de equipo que emplea y sus relaciones de trabajo con los demás. Si los productos que se diseñan no se amoldan al entorno, es posible que su uso resulte difícil o frustrante

En algunas aplicaciones la interfaz de usuario para un sistema de cómputo se coloca en un "lugar amigable para el usuario" (por ejemplo, iluminación apropiada, buena altura de la pantalla, fácil acceso al teclado), pero en otros (como el piso de una fábrica o la cabina de un avión), la iluminación es deficiente, el ruido es importante, un teclado o un ratón no son una opción, la colocación del monitor es menos que ideal. Al diseñador de la interfaz lo limitarán factores que atentan contra la facilidad de uso.

Además de los factores del entorno físico, la cultura del lugar de trabajo también incide. ¿La interacción del sistema se medirá de alguna manera (por ejemplo, tiempo por transacción o exactitud de ésta)? ¿Dos o más personas tendrán que compartir información antes de que se proporcione una entrada? ¿Cómo se dará soporte a los usuarios del sistema? Es necesario responder éstas y muchas otras preguntas relacionadas antes de iniciar el diseño de la interfaz.

12.4 PASOS DEL DISEÑO DE LA INTERFAZ

Una vez finalizado el análisis de la interfaz, se han identificado con detalle todas las tareas (objetos o acciones) que requiere el usuario final, y comenzará la actividad del diseño de la interfaz. Esta etapa, como todo el diseño de la ingeniería del software, es un proceso iterativo. Cada paso del diseño de la interfaz se da varias veces, y en cada uno de ellos se elabora y refina información desarrollada en los pasos anteriores.

Aunque se han propuesto muchos modelos diferentes para el diseño de la interfaz de usuario (por ejemplo, [NOR86], [NIE00]), todos sugieren alguna combinación de los siguientes pasos:

1. Con base en la información desarrollada durante el análisis de la información (sección 12.3), definir los objetos y las acciones de la interfaz (operaciones).
2. Definir eventos (acciones del usuario) que cambiarán el estado de la interfaz. Modelar este comportamiento.
3. Representar cada estado de la interfaz tal como lo verá el usuario final.
4. Indicar cómo interpreta el usuario el estado del sistema a partir de la información proporcionada mediante la interfaz.

En algunos casos, el diseñador de la interfaz puede empezar con borradores de cada estado de la interfaz (es decir, el aspecto de la interfaz en distintas circunstancias) y luego trabajar hacia atrás para definir objetos, acciones y otra información importante para el diseño. Independientemente de la secuencia de las tareas del diseño, éste debe 1) seguir siempre las reglas de oro analizadas en la sección 12.1; 2) modelar la manera en que se implementará la interfaz, y 3) tomar en cuenta el entorno (por ejemplo, la tecnología de despliegue, el sistema operativo, las herramientas de desarrollo) en que habrá de usarse.

"El diseño interactivo [es] una mezcla integrada de artes gráficas, tecnología y psicología"

Brad Wieners

12.4.1 Aplicación de los pasos del diseño de la interfaz

Un paso importante en el diseño de la interfaz es la definición de los objetos que ésta tendrá y las acciones que se les aplicarán. Para realizarlo se analizan los casos de uso de manera muy parecida a la descrita en el capítulo 8. Es decir, se escribe una descripción de un caso de uso. Luego se aíslan los nombres (objetos) y los verbos (acciones) para crear una lista de objetos y acciones.

Una vez definidos los objetos y las acciones, que se han elaborado de manera iterativa, se organizan por tipo. Se identifican objetos de destino, origen y aplicación. Un *objeto de origen* (como el icono de un informe) se arrastra y coloca en un *objeto de destino* (por ejemplo, un icono de impresora). La implicación de esta acción es crear un informe impreso. Un *objeto de aplicación* representa datos específicos de la aplicación que no se manipulan directamente como parte de la interacción con la pantalla. Por ejemplo, en una lista de correo se almacenan nombres para un envío de correspondencia. La propia lista podría ordenarse, combinarse o purgarse (acciones de menú), pero no arrastrarse ni colocarse mediante interacción del usuario.

Una vez que el diseñador queda satisfecho con un objeto importante y que se han definido las acciones (para una iteración de diseño) se realiza el formato de la pantalla. Como otras actividades de diseño de la interfaz, el *formato de la pantalla* es un proceso interactivo; en él se elabora el diseño gráfico y se colocan los iconos, la definición de texto descriptivo en pantalla, la especificación y la asignación de nombres a las ventanas, además de la definición de los elementos primarios y secundarios de los menús. Si una metáfora de la realidad es apropiada para la aplicación, se especifica en este momento, y el diseño se organiza de manera tal que satisfaga la metáfora.

Un breve ejemplo de los pasos del diseño indicados anteriormente se obtiene imaginando el contexto en que se sitúa un usuario del sistema *HogarSeguro* analizado en capítulos anteriores. A continuación se presenta un caso de estudio preliminar (escrito por el propietario) para la interfaz.

Caso de uso preliminar. Quiero tener acceso a mi sistema *HogarSeguro* desde cualquier lugar remoto vía Internet. Empleando software de navegador que opera en mi notebook (mientras estoy trabajando o viajando) puedo determinar el estado del sistema de alarmas, armar o desarmar el sistema, reconfigurar zonas de seguridad y ver diferentes habitaciones de la casa con las cámaras de video preinstaladas.

Para tener acceso a *HogarSeguro* desde un lugar remoto proporciono una identificación y una contraseña. Estos elementos definen los niveles de acceso (por ejemplo, no todos los usuarios pueden reconfigurar el sistema ni proporcionar seguridad). Una vez validado, puedo revisar el estatus del sistema y cambiarlo al armar o desarmar *HogarSeguro*. Puedo reconfigurar el sistema al desplegar un plano de la casa, ver cada uno de los sensores de

seguridad, desplegar cada zona configurada actualmente y modificar zonas de acuerdo con las necesidades. Puedo ver el interior de la casa con las cámaras de video colocadas de manera estratégica. Puedo hacer acercamientos y desplazamientos con las cámaras para proporcionar diferentes vistas del interior.

Con base en este caso de uso se identifican las tareas del propietario, los objetos y los elementos siguientes:

- *tiene acceso* al sistema *HogarSeguro*
- *ingresa* un **ID** y una **contraseña** para acceso remoto
- *revisa* el **estatus del sistema**
- *arma* o *desarma* el sistema HogarSeguro
- *despliega* el **plano de la casa** y las **ubicaciones de los sensores**
- *despliega* **zonas** en el plano de la casa
- *cambia* **zonas** en el plano de la casa
- *despliega* **ubicaciones de las cámaras de video** o el **plano de la casa**
- *selecciona* visualmente una **cámara de video**
- *ve* **imágenes de video**
- *desplaza* o *acerca* las **cámaras de video**

Los objetos (en negritas) y las acciones (en cursivas) se extraen de la lista de del propietario. La mayor parte de los objetos indicados son objetos de la ap Sin embargo, **ubicación de las cámaras de video** (un objeto de origen) se tra y coloca en **cámara de video** (un objeto de destino) para crear una **video** (una ventana que contiene el despliegue de video).

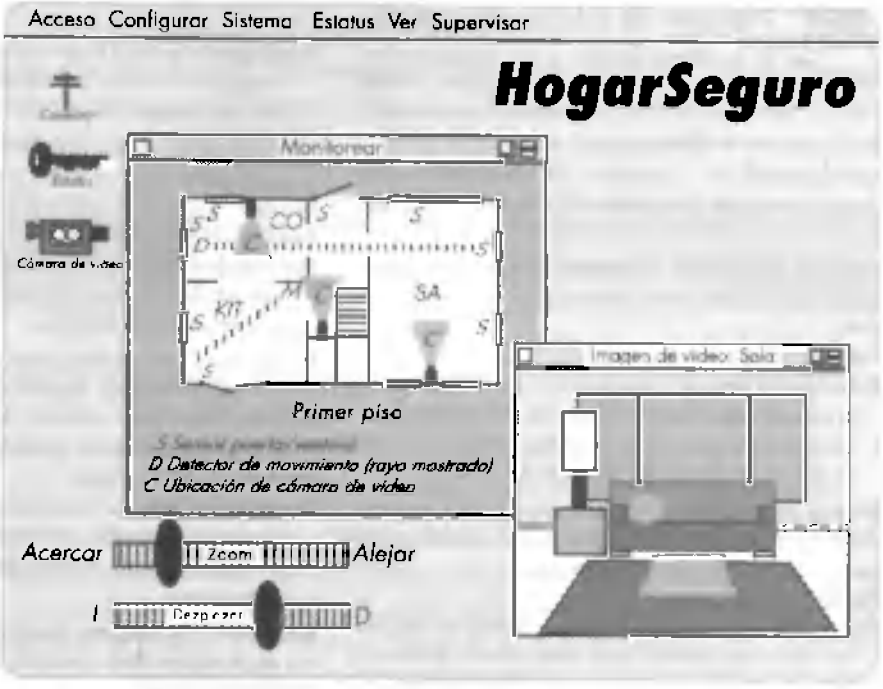
Se crea un boceto preliminar del formato de la pantalla para el monitoreo deo (figura 12.3).[8] La imagen de video se solicita seleccionando un icono de ción de las cámaras de video, *C*, localizado en el plano de la casa despliega ventana de monitoreo. En este caso se arrastra la ubicación de una cámara de en la sala, SA, y se coloca sobre el icono de cámara de video ubicado en la perior izquierda de la pantalla. Aparecerá la ventana de imagen de video, do video de flujo continuo proveniente de la cámara ubicada en la sala (SA troles deslizables de acercamiento y desplazamiento se emplean para co ampliación y la dirección de la imagen del video. Para seleccionar una vista cámara, el usuario simplemente arrastra y coloca un icono de ubicación de

CONSEJO

Aunque las herramientas automatizadas son útiles en el desarrollo de prototipos de formato, en ocasiones todo lo que se necesita es lápiz y papel

8 Considérese que esto difiere de la implementación de estas características en capítulos Esto podría considerarse un borrador del primer diseño y representa una opción digna en cuenta

FIGURA 12.3
Formato de pantalla preliminar.



diferente en el icono de la cámara emplazado en la esquina superior izquierda de la pantalla.

El boceto del formato que se muestra tendría que complementarse con una expansión de cada elemento de menú dentro de la barra de menús, indicando cuáles acciones están disponibles para el modo de monitoreo de video (estado). Durante la etapa de diseño de la interfaz se crearía un conjunto completo de bocetos para cada tarea de propietario anotada en el escenario del usuario.

12.4.2 Patrones de diseño de la interfaz de usuario

Las interfaces gráficas de usuario sofisticadas se han vuelto tan comunes que ha surgido una amplia variedad de patrones de diseño de interfaces de usuario. Como se observó al principio de este libro, un patrón de diseño es una abstracción que prescribe una solución de diseño a un problema de diseño específico, bien delimitado. Cada uno de los patrones de ejemplo (y todos los patrones de cada categoría) presentados en el recuadro siguiente también tendría un diseño completo al nivel de componentes, incluidos clases, atributos, operaciones e interfaces de diseño.

Un análisis completo de los patrones de interfaz de usuario está más allá del alcance de este libro. El lector interesado debe consultar [DUY02], [BOR01], [WEL01] y [TID02] para conocer más información.

INFORMACIÓN

Patrones de interfaz de usuario

En las últimas décadas se han propuesto cientos de patrones de interfaz de usuario. Tidwell [TID02] y Van Welie [WEL01] proporcionan taxonomías[9] de los patrones de diseño de interfaz de usuario que se han organizado en 10 categorías. En este recuadro se presentan patrones de ejemplo de cada una de estas categorías.

Interfaz de usuario completa. Proporciona líneas generales de diseño para estructura y navegación de alto nivel

Patrón: *navegación de alto nivel*

Descripción breve: Proporciona un menú de alto nivel, a menudo acoplado a un logotipo o una imagen de identificación que permite la navegación directa a cualquiera de las principales funciones del sistema.

Diseño de página. Atiende la organización general de las páginas (para sitios Web) o distintos despliegues de pantalla (para aplicaciones interactivas).

Patrón: *pila de cartas*

Descripción breve: Proporciona el aspecto de una pila de cartas con pestañas; para seleccionarlas se hace clic en cada una de ellas, que representa subfunciones específicas o categorías de contenido

Formularios y entrada. Toma en cuenta diversas técnicas de diseño para completar entradas al nivel de formularios

Patrón: *llene los espacios en blanco*

Descripción breve: Permite el ingreso de datos alfanuméricos en un "cuadro de texto"

Tablas. Proporciona guía de diseño para la creación y manipulación de todo tipo de datos tabulares

Patrón: *tabla que permite su ordenación*

Descripción breve: Despliega una larga lista de registros que se ordenan al seleccionar un mecanismo interruptor para cualquier etiqueta de la columna

Manipulación directa de datos. Atiende la edición, modificación y transformación de datos.

Patrón: *migas de pan*

Descripción breve: Proporciona una ruta completa de navegación cuando el usuario está trabajando con una jerarquía completa de páginas o pantallas de despliegue

Navegación. Ayuda al usuario en el recorrido de menús jerárquicos, páginas Web y pantalla de despliegue interactivo

Patrón: *edición en el lugar*

Descripción breve: Proporciona la capacidad de edición simple de textos para ciertos tipos de contenido en el lugar en que se despliega.

Búsqueda. Permite búsquedas de contenido específico mediante la información mantenida en un sitio Web o contenida en almacenes persistentes de datos que están accesibles para una aplicación interactiva

Patrón: *búsqueda simple*

Descripción breve: Proporciona la capacidad de buscar, en un sitio Web o una fuente de datos persistente, un elemento simple de datos descrito por una cadena alfanumérica

Elementos de página Implementa elementos específicos de un página Web o pantalla de despliegue.

Patrón: *asistente*

Descripción breve: Lleva al usuario paso a paso por una tarea compleja, proporcionando guías para completar la tarea mediante una serie de ventanas

Comercio electrónico Específicos de sitios Web estos patrones implementan elementos recurrentes de aplicaciones de comercio electrónico.

Patrón: *carrito de compras*

Descripción breve: Proporciona una lista de elementos seleccionados en una compra.

Varios Patrones que no caben fácilmente en las categorías anteriores. En algunos casos, estos patrones dependen del dominio u ocurren para clases específicas de

Patrón: *indicador de programa*

Descripción breve: Proporciona una indicación de progreso cuando se está realizando una operación

12.4.3 Temas de diseño

A medida que evoluciona el diseño de una interfaz casi siempre surgen cuatro comunes: tiempo de respuesta del sistema, funciones de ayuda para el usuario, manejo de información de error y rotulado de comandos. Por desgracia, muchos

9 En [TID02] y [WEL01] se encontrarán descripciones de patrones completos (junto con otros patrones).

ñadores no prestan atención a estos temas hasta una etapa relativamente tardía del proceso de diseño (en ocasiones el primer atisbo de un problema se presenta hasta que se dispone de un prototipo operacional). Como resultado, a veces se tiene iteración innecesaria, demoras del proyecto y frustración del cliente. Es mucho mejor abordar cada uno como elemento de diseño y tomarlo en cuenta al principio del diseño de software, cuando los cambios son fáciles y el costo es bajo.

> "Un error común que comete la gente cuando trata de diseñar algo a prueba de tontos es subestimar la ingenuidad de los verdaderamente tontos."
>
> **Douglas Adams**

Tiempo de respuesta. El tiempo de respuesta del sistema es la primera queja sobre muchas aplicaciones interactivas. En general, se mide desde el punto en que el usuario realiza alguna acción de control (como oprimir la tecla Return o hacer clic con el ratón) hasta que el software responde con la salida o la acción deseada.

El tiempo de respuesta del sistema tiene dos características importantes: duración y variabilidad. Si la respuesta del sistema se demora mucho, la frustración y el estrés del usuario son inevitables. Variabilidad es la desviación del tiempo de respuesta promedio y, en muchos sentidos, es la característica más importante del tiempo de respuesta. Una baja variabilidad permite que el usuario establezca un ritmo de interacción, aunque el tiempo de respuesta sea relativamente largo. Por ejemplo, una respuesta de 1 segundo a un comando a menudo será preferible a una respuesta que varía de 0.1 a 2.5 segundos. Cuando la variabilidad es significativa, el usuario siempre se encontrará fuera de balance, siempre se preguntará si ha ocurrido algo "diferente" tras bastidores.

Funciones de ayuda. Casi todos los usuarios de un sistema de cómputo interactivo necesitan ayuda de vez en cuando. En algunos casos, basta con una simple pregunta dirigida a un colega con experiencia. En otros, tal vez la única opción sea una investigación detallada en un conjunto de varios volúmenes de "manuales de usuario". Sin embargo, en casi todos los casos el software moderno proporciona funciones de ayuda en línea que permiten al usuario obtener una respuesta a sus preguntas o resolver un problema sin dejar la interfaz.

Deben atenderse varios temas de diseño [RUB88] cuando se toma en cuenta una opción de ayuda.

- ¿La ayuda estará disponible para todas las funciones del sistema y en todo momento durante la interacción con éste? Entre las opciones se incluye ayuda sólo para un subconjunto de todas las funciones y acciones o ayuda para todas las funciones.
- ¿Cómo necesitará la ayuda el usuario? Entre las opciones se incluyen menú de ayuda, una tecla especial de función o un comando AYUDA.

- ¿Cómo se representará la ayuda? Las opciones son una ventana separada, una referencia a un documento impreso (menos que ideal) o una sugerencia de una o dos líneas que aparece en un lugar fijo de la pantalla.
- ¿Cómo regresará el usuario a la interacción normal? Entre las opciones se incluyen un botón de regreso desplegado en la pantalla, una tecla de función o una secuencia de control.
- ¿Cómo se estructurará la información de ayuda? Las opciones son una estructura "plana" en que se tiene acceso a toda la información con el teclado, una jerarquía en capas de información que proporciona detalles crecientes a medida que el usuario la recorre, o el uso de hipertexto.

Manejo de errores. Los mensajes de error y los avisos de precaución son "malas noticias" que se entregan a los usuarios de sistemas interactivos cuando algo sale mal. En el peor de los casos, los mensajes de error y los avisos de precaución ofrecen información inútil o que puede malinterpretarse y que sólo sirve para aumentar la frustración del usuario. Algunos usuarios de computadora han encontrado un error de la forma *"La aplicación XXX se ha visto forzada a cerrarse porque se ha encontrado un error del tipo 1023"*. En algún lugar debe existir una explicación del error 1023; de lo contrario, ¿por qué habrían asignado los diseñadores ese identificador? Sin embargo, el mensaje de error no proporciona una indicación real de lo que estuvo mal ni de la forma de buscar la información adicional. Un mensaje de error presentado de esta manera no hace nada por aliviar la ansiedad del usuario ni ayuda a corregir el problema.

> "La interfaz con el infierno: 'Para corregir este error y seguir adelante, escriba cualquier número primo de 11 dígitos...'"
>
> Autor anónimo

En general, todo mensaje de error o aviso de precaución que produzca un sistema interactivo debe tener las siguientes características:

¿Qué características debe tener un "buen" mensaje de error?

- El mensaje debe describir el problema en un lenguaje que el usuario entienda.
- El mensaje debe proporcionar consejos constructivos sobre la manera de recuperarse del error.
- El mensaje debe indicar cualquier consecuencia negativa del error (por ejemplo, la posibilidad de que se corrompan los archivos de datos) para que el usuario se asegure que no han ocurrido (o para que los corrija si ya ocurrieron).
- El mensaje debe acompañarse de una pista visual o auditiva. Es decir, debe generarse un bip junto con el despliegue del mensaje, o éste debe parpadear momentáneamente o desplegarse en un color que se reconozca fácilmente como "color de error".
- El mensaje no debe contener juicios. Es decir, la redacción nunca debe culpar al usuario.

Como a nadie le gustan las malas noticias, a pocos usuarios le gustarán los mensajes de error, sin importar lo bien diseñados que estén. Pero un enfoque adecuado para los mensajes de error hará mucho por mejorar la calidad de un sistema interactivo y reducirá de manera importante la frustración del usuario cuando ocurran los problemas.

Rotulación de menús y comandos. El comando de texto escrito fue alguna vez el modo más común de interacción entre los usuarios y el software del sistema y se usaba en aplicaciones de todo tipo. Hoy el uso de interfaces orientadas a ventanas, con opción de señalar y elegir, ha reducido la dependencia de los comandos escritos, pero muchos usuarios avanzados aún prefieren este tipo de interacción. Varios temas de diseño surgen cuando se proporcionan comandos de texto o etiquetas de menú como modo de interacción:

- ¿Cada opción de menú tiene un comando correspondiente?
- ¿Qué forma tendrán los comandos? Entre las opciones se incluyen una secuencia de control (como alt-P), teclas de función o palabras escritas por el usuario
- ¿Qué tan difícil será aprender y recordar los comandos? ¿Qué puede hacerse si se olvida un comando?
- ¿El usuario tiene la opción de personalizar o abreviar los comandos?
- ¿Las etiquetas de los menús se explican por sí solas dentro del contexto de la interfaz?
- ¿Los submenús son consistentes con la función indicada en un elemento principal del menú?

Como se indicó al principio de este capítulo, es necesario definir convenciones para usar comandos en toda la aplicación. Es confuso para el usuario y a menudo lo lleva a cometer errores escribir alt-D cuando desea duplicar un objeto gráfico en una aplicación y alt-D cuando quiere deshacer una acción en otra. Es obvio que esto propiciará errores.

Referencia Web

Un sitio [illegible] de software [illegible] www.3. [illegible] /wai/ [illegible] / [illegible] / [illegible] software.

Accesibilidad de la aplicación. A medida que las aplicaciones de computadora se vuelven ubicuas, los ingenieros de software deben asegurarse de que el diseño de la interfaz tenga mecanismos que permiten un fácil acceso a quienes tienen necesidades especiales. La *accesibilidad* es un imperativo moral, legal y comercial para los usuarios (e ingenieros de software) que tienen problemas físicos. Diversas líneas generales de accesibilidad (por ejemplo, [W3C03]), muchas diseñadas para aplicaciones Web, pero a menudo aplicables a cualquier software, proporcionan sugerencias detalladas para el diseño de interfaces que alcanzan diferentes grados de accesibilidad. Otros (como [APP03], [MIC03]) proporcionan líneas generales específicas para "tecnología asistencial" que atiende las necesidades de quienes tienen discapacidades visuales, auditivas, de movilidad, del habla o de aprendizaje

Internacionalización. Los ingenieros de software y sus administradores subestiman invariablemente el esfuerzo y las habilidades necesarias para crear interfases de usuario que atiendan las necesidades de usuarios de otras localidades o que hablan diferentes idiomas. Con demasiada frecuencia, las interfases se diseñan para una localidad y un idioma y luego se espera que funcionen en otros países. El reto para los diseños de interfases es crear software "globalizado"; las interfaces de usuario deben diseñarse para contener un núcleo genérico de funciones que se entreguen a todos los usuarios. Características adicionales de localidad permiten a la interfaz personalizarse para un mercado específico.

Los ingenieros de software cuentan con varias pautas para la internacionalización (como [IBM03]). Estas pautas atienden amplios problemas de diseño (como las diferencias en formato de pantalla en varios mercados) y temas discretos de implementación (por ejemplo, diferentes alfabetos pueden crear rotulación y requisitos de espacio especiales). El estándar *Unicode* [UNI03]) se ha desarrollado para atender el desalentador desafío de manejar docenas de idiomas naturales con cientos de caracteres y símbolos.

HERRAMIENTAS DE SOFTWARE

Desarrollo de interfases de usuario

Objetivo Estas herramientas le permiten a un ingeniero de software crear una sofisticada interfaz gráfica de usuario empleando relativamente escaso desarrollo de software personalizado. Las herramientas proporcionan acceso a componentes reutilizables y convierte la creación de una interfaz en una selección entre opciones predefinidas que se ensamblan mediante la herramienta.

Mecánica. Las interfases de usuario modernas están construidas con un conjunto de componentes reutilizables acoplados con algunos componentes personalizados desarrollados para proporcionar funciones especializadas. Casi todas las herramientas de desarrollo de interfases de usuario permiten que el ingeniero de software cree una interfaz empleando opciones de "arrastrar y colocar"; es decir, el desarrollador selecciona entre opciones predefinidas (por ejemplo, constructores de formularios, mecanismos de interacción, capacidad de procesamiento de comandos) y coloca esas opciones en el contenido de la interfaz que habrá de crearse.

Herramientas de representación[10]

Macromedia Authorware, desarrollado por Macromedia Inc. (www.macromedia.com/software/), se ha diseñado para la creación de interfases y entornos de aprendizaje electrónico. Emplea características sofisticadas de construcción.

Motif Common Desktop Environment, desarrollado por The Open Group (www.osf.org/tech/desktop/cde/), es una interfaz gráfica de usuario integrada para sistemas abiertos de computación de escritorio. Entrega una interfaz gráfica simple, estándar, para la administración de datos, archivos y aplicaciones.

PowerDesigner/PowerBuilder, desarrollado por Sybase (www sybase/products/internetappdevtools), es un conjunto muy completo de herramientas CASE, que incluyen muchas opciones para el diseño y la construcción de interfases gráficas de usuario.

10 Las herramientas expuestas aquí sólo representan una muestra de esta categoría. En casi todos los casos los nombres de las mismas son marcas registradas de sus respectivos desarrolladores.

12.5 EVALUACIÓN DEL DISEÑO

Una vez que se ha creado un prototipo de interfaz de usuario operacional, debe evaluarse y determinar si satisface las necesidades del usuario. La evaluación puede abarcar un espectro de grados de formalidad que va desde una "prueba de manejo" informal, en la cual un usuario proporciona retroalimentación informal, hasta un estudio diseñado formalmente, el cual emplea métodos estadísticos para la evaluación de cuestionarios que llena una población de usuarios finales.

El ciclo de evaluación de la interfaz de usuario asume la forma mostrada en la figura 12.4. Después de completado el diseño se crea un prototipo de primer nivel. A continuación, el usuario evalúa este prototipo[11] y hace comentarios directos al diseñador acerca de la eficacia de la interfaz. Además, si se utilizan técnicas formales de evaluación (por ejemplo, cuestionarios, hojas de evaluación), es probable que el diseñador obtenga información de estos datos (por ejemplo, del 80 al 100 por ciento de los usuarios rechaza el mecanismo para guardar archivos de datos). Las modificaciones al diseño se hacen basándose en la información que proporciona el usuario, y así se crea un prototipo de segundo nivel. El ciclo de la evaluación continúa hasta que ya sean innecesarias más modificaciones al diseño de la interfaz.

FIGURA 12.4
Ciclo de evaluación del diseño de interfaz.

11 Es importante notar que los expertos en diseños ergonómomico y de interfaz también pueden dirigir revisiones de la interfaz. Dichas funciones se llaman evaluaciones heurísticas o ensayos cognitivos.

El enfoque de creación de prototipos resulta eficaz, pero ¿es posible evaluar la calidad de una interfaz de usuario antes de construir un prototipo? Si se descubren posibles problemas y se corrigen en las primeras etapas, se reducirá el número de bucles en el ciclo de evaluación y se acortará el tiempo de desarrollo. Si se ha creado un modelo de diseño de la interfaz, es posible aplicar varios criterios de evaluación [MOR81] en las primeras revisiones del diseño:

1. La longitud y complejidad de la especificación escrita del sistema y su interfaz indican la cantidad de aprendizaje necesario para los usuarios del sistema
2. El número especificado de tareas del usuario y el promedio de acciones por tarea indican el tiempo de interacción y la eficacia global del sistema.
3. La cantidad de acciones, tareas y estados del sistema que indica el modelo de diseño se relaciona con la carga de memoria que recae en los usuarios del sistema.
4. El estilo de la interfaz, las funciones de ayuda y el protocolo de manejo de errores indican en forma general la complejidad de la interfaz y el grado de aceptación del usuario.

Una vez construido el primer prototipo, el diseñador puede recopilar diversos datos cualitativos y cuantitativos que ayudarán a evaluar la interfaz. Para recopilar los datos cualitativos se pueden distribuir cuestionarios entre los usuarios del prototipo con preguntas que arrojan: 1) respuesta simple sí/no, 2) respuesta numérica, 3) respuesta con escala de valoración (subjetiva), 4) escala de Likert (por ejemplo, completamente de acuerdo, un poco de acuerdo), 5) respuesta con porcentajes (subjetiva) o 6) respuesta abierta.

Si se desean datos cuantitativos, puede aplicarse una forma de análisis de estudio del tiempo. Se observa a los usuarios durante la interacción y se usan los datos (como el número de tareas completadas correctamente en un periodo estándar, frecuencia y secuencia de acciones, tiempo que pasa "mirando" la pantalla, número y tipo de errores, tiempo de recuperación de errores, tiempo dedicado al uso de la ayuda y cantidad de referencias de ayuda por periodo estándar) como guía para la modificación de la interfaz.

Un análisis completo de los métodos de evaluación de la interfaz de usuario rebasa el alcance de este libro. Se puede consultar más información en [LEA88], [MAN97] y [HAC98].

12.6 RESUMEN

Es posible afirmar que la interfaz de usuario es el elemento más importante de un sistema o producto de cómputo. Si la interfaz está mal diseñada la capacidad del usuario se verá muy reducida para aprovechar las ventajas de una aplicación. En efecto, una interfaz débil puede llevar al fracaso una aplicación bien diseñada y con una implementación sólida.

Tres principios importantes guían el diseño de una interfaz de usuario efectiva: 1) dar el control al usuario, 2) reducir la carga en la memoria del usuario, y 3) lograr que la interfaz sea consistente. Construir una interfaz que cumpla con estos principios requiere desarrollar un proceso de diseño organizado.

El diseño de la interfaz de usuario comienza con una serie de tareas de análisis. Entre éstas se encuentra identificación del usuario, tarea y análisis y modificación de la tarea y el entorno. El análisis del usuario define los perfiles de varios usuarios finales y aplica información recopilada de diferentes fuentes de negocios y técnicas. El análisis de tareas define las tareas y acciones del usuario empleando un enfoque elaborativo u orientado a objetos, aplicando casos de uso, elaboración de tareas y objetos, análisis de flujo de trabajo y representaciones jerárquicas de tareas para comprender plenamente la interacción ser humano-computadora. El análisis ambiental identifica las estructuras física y social en que debe operar la interfaz.

Una vez identificadas las tareas, se crean y analizan los escenarios para definir un conjunto de objetos y acciones de la interfaz. Esto proporciona la base para la creación del formato de pantalla, que representa el diseño gráfico y la ubicación de los íconos, la definición de un texto descriptivo en pantalla, la especificación de las ventanas y la asignación de títulos a éstas, además de la especificación de los elementos primarios y secundarios del menú. Mientras se refina el modelo de diseño, deben tomarse en cuenta temas relacionados con el diseño, como tiempo de respuesta, estructura de comandos y acciones, manejo de errores y funciones de ayuda. El usuario dispone de varias herramientas de implementación para construir un prototipo que él mismo puede evaluar.

REFERENCIAS

[APP03] Apple Computer, *People with Special Needs*, 2003, disponible en http://www.apple.com/disability/

[BAR01] Borchers, J., *A Pattern Approach to Interaction Design*, Wiley 2001.

[CON95] Constantine, L. "What DO Users Want? Engineering Usability in Software", en *Windows Tech Journal*, diciembre de 1995, disponible en http://www.forUse.com.

[DON99] Donahue, G., S. Weinschenck y J. Nowicki, "Usability is Good Business", Compuware Corp., julio de 1999, disponible en http://www.compuware.com.

[DUY02] vanDuyne, D., J. Landay y J. Hong, *The design of Sites*, Addison-Wesley, 2002.

[HAC98] Hackos, J. y J. Redish, *User and Task Analysis for Interface Design*, Wiley, 1998.

[IBM03] IBM, *Overview of Software Globalization*, 2003, disponible en http://oss.software.ibm.com/icu/userguide/i18n.html

[LEA88] Lea, M., "Evaluating User Interfaces Designs", en *User Interface Design for Computer Systems*, Halstead Press (Wiley), 1988.

[MAN97] Mandel, T., *The Elements of User Interface Design*, Wiley, 1997.

[MIC03] *Microsoft Accesability Technology for Everyone*, 2003, disponible en htt.//www.microsoft.com/enable/.

[MON84] Monk, A. (ed), *Fundamentals of Human-Computer Interaction*, Academic Press, 1984.

[MOR81] Moran, T. P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems", en *Intl. Journal of Man-Machine Studies*, vol. 15, pp. 3-50.

[NIE00] Nielsen, J., *Designing Web Usability*, New Riders Publishing, 2000.

[NOR86] Norman, D. A., "Cognitive Engineering", en *User Centered Systems Design*, Lawrence Earlbaum Associates, 1986.

[RUB88] Rubin, T., *User Interface Design for Computer Systems*, Haldstead Press (Wiley), 1988
[SHN90] Shneiderman, B., *Designing the User Interface*, Addison-Wesley, 3a. ed., 1990.
[TID99] Tidwell, J., "Common Ground: A Pattern Language for HCI Design", disponible en http://www.mit.edu/~jtidwell/interaction_patterns.html, mayo de 1999
[TID02] Tidwell, J., "UI Patterns and Techniques", disponible en http://time.tripper.com/uipatterns/index.html, mayo de 2002.
[UNI03] Unicode Inc., *The Unicode Home Page*, 2003, disponible en http://www.unicode.org
[W3C03] World Wide Web Consortium, *Web Content Accesability Guidelines*, 2003, disponible en http://www.w3.org/TR/2003/Word-WCAG20-20030624/.
[WEL01] vanWelle M., "Interaction Design Patterns", disponible en http://www.welie.com/patterns/, 2001.

PROBLEMAS Y PUNTOS A CONSIDERAR

12.1. Describir la mejor y la peor interfaz con que se haya trabajado alguna vez y critiquenlas en relación con los conceptos presentados en este capítulo.

12.2. Desarrollar dos principios de diseño adicionales que "den el control al usuario".

12.3. Desarrollar dos principios de diseño adicionales que "reduzcan la carga en la memoria del usuario"

12.4. Desarrollar dos principios de diseño adicionales que "logren una interfaz consistente".

12.5. Se ha pedido desarrollar un sistema de banco en casa para Web; desarrollar los modelos del usuario, del diseño y mental y la implementación

12.6. Realizar un análisis detallado de tareas para el sistema del problema 12.5. Utilizar un enfoque elaborativo u orientado a objetos.

12.7. Agregar por lo menos cinco preguntas adicionales a la lista desarrollada para el análisis de contenido de la sección 12.3.5.

12.8. Siguiendo con el problema 12.6, definir objetos y acciones de interfaz para la aplicación. Identificar cada tipo de objetos

12.9. Desarrollar un conjunto de formatos de pantalla con una definición de elementos principales y secundarios del menú para el sistema del problema 12.5.

12.10. Desarrollar un conjunto de formatos de pantalla con una definición de los elementos principales y secundarios del menú para el sistema *HogarSeguro*. Es posible aplicar un enfoque diferente del que se muestra para el formato de pantalla en la figura 15.3.

12.11. Describir un enfoque propio de las funciones de ayuda del usuario para el análisis de tareas que se hayan realizado como parte del problema 12.5.

12.12. Proporcionar algunos ejemplos que ilustren por qué debe tomarse en cuenta la variabilidad del tiempo de respuesta

12.13. Desarrollar un enfoque que integre automáticamente los mensajes de error y la función de ayuda para el usuario. Es decir, que el sistema reconozca automáticamente el tipo de error y proporcione una ventana de ayuda con sugerencias para corregirlo. Realizar un diseño de software razonablemente completo que tome en cuenta las estructuras de datos y los algoritmos apropiados

12.14. Desarrollar un cuestionario de evaluación de interfaces que contenga 20 preguntas genéricas aplicables a la mayor parte de las interfaces. Pídase que 10 compañeros de clase completen el cuestionario para un sistema interactivo que todos usen. Resumir los resultados e informar de ellos a su clase.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Aunque su libro no se relaciona específicamente con interfaces ser humano-computadora, mucho de lo que Donald Norman (*The Design of Everyday Things*, edición reimpresa, Currency/D

bleday, 1990) tiene que decir sobre la psicología de un diseño efectivo se aplicará a las interfaces de usuario. Es una lectura recomendada para cualquier persona que tome con seriedad el diseño de interfaces de alta calidad.

Las interfaces gráficas del usuario son ubicuas en el mundo moderno de la computación. Ya sea empleada por un cajero automático, un teléfono móvil, una PDA, un sitio Web o una aplicación de negocios, la interfaz de usuario proporciona una ventana al software. Por ello, abundan los libros sobre el diseño de interfaces. Todos los siguientes libros tratan sobre facilidad de uso, conceptos de interfaz de usuario, principios y técnicas de diseño, además de que contienen muchos ejemplos útiles: Galitz (*The Essential Guide to User Interface Design*, Wiley, 2002), Cooper (*About Face 2.0: The Essentials of User Interface Design*, IDG Books, 2003), Beyer y Holtzblatt (*Contextual Design: A Costumer Centered Approach to Systems Design*, Morgan-Kaufmann, 2002), Raskin (*The Human Interface*, Addison-Wesley, 2000), Constantine y Lockwood (*Software for Use*, ACM Press, 1999) y Mayhew (*The Usability Engineering Lifecycle*, Morgan-Kaufmann, 1999).

Johnson (*GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan-Kaufmann, 2000) proporciona una guía útil para quienes aprenden mejor al examinar contraejemplos. Un libro que se disfruta, de Cooper (*The Inmates Are Running the Asylum*, Sams Publishing, 1999), analiza por qué los productos de alta tecnología nos atraen y la manera de diseñar unos que no lo hagan.

El análisis y modelado de tareas son actividades fundamentales del diseño de interfaces. Hackos y Redish [HAC98] han escrito un libro dedicado a estos temas y proporcionan un método detallado para concentrarse en el análisis de tareas. Wood (*User Interface Design: Bridging the Gap from User Requirements to Design*, CRC Press, 1997) aborda la actividad de análisis para interfaces y la transición a las tareas de diseño.

La actividad de evaluación se concentra en la facilidad de uso. Los libros de Rubin (*Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*, Wiley, 1994) y Nielsen (*Usability Inspection Methods*, Wiley, 1994) abordan el tema con gran detalle.

En un libro único, que podría parecer muy interesante a los diseñadores de producto, Murphy (*Front Panel: Designing Software for Embedded User Interfaces*, R&D Books, 1998) ofrece una guía detallada para el diseño de interfaces destinadas a sistemas incrustados y aborda los peligros de seguridad inherentes a los controles, el manejo de maquinaria pesada y las interfaces para sistemas médicos o de transporte. El diseño de la interfaz para productos incrustados también se estudia en el libro de Garrett (*Advanced Instrumentation and Computer I/O Design: Real-Time System Computer Interface Engineering*, IEEE, 1994).

En Internet se encuentra una amplia variedad de fuentes de información sobre el diseño de la interfaz de usuario. Una lista actualizada de referencias en la World Wide Web relevantes para el diseño de la interfaz de usuario se encuentra en el sitio Web SEPA.
http://www.mhhe.com/pressman.

CAPÍTULO

13 ESTRATEGIAS DE PRUEBA DEL SOFTWARE

CONCEPTOS CLAVE

Una estrategia de prueba del software integra los métodos de diseño de caso de pruebas del software en una serie bien planeada de pasos que desembocará en la eficaz construcción de software. La estrategia proporciona un mapa que describe los pasos que se darán como parte de la prueba, indica cuándo se planean y cuándo se dan estos pasos, además de cuánto esfuerzo, tiempo y recursos consumirán. Por tanto, cualquier estrategia de prueba debe incorporar la planeación de pruebas, el diseño de caso de pruebas, la ejecución de pruebas y la recolección y evaluación de los datos resultantes.

Una estrategia de prueba del software debe ser lo suficientemente flexible como para promover un enfoque personalizado. Al mismo tiempo, debe ser lo adecuadamente rígido como para promover una planeación razonable y un seguimiento administrativo del avance del proyecto. Shooman [SHO83] analiza estos temas:

> En muchos sentidos, la prueba es un proceso autónomo, y el número de tipos diferentes de pruebas varía tanto como los diferentes enfoques de desarrollo. Durante muchos años, la única defensa contra los errores de programación fueron el diseño cuidadoso y la inteligencia natural del programador. Ahora estamos en la era en que las técnicas modernas de diseño [y las revisiones de las técnicas formales] nos están ayudando a reducir el número de errores iniciales inherentes al código. De manera similar, diferentes métodos de prueba están empezando a apilarse en varios métodos y filosofías distintos.

Estos "enfoques y filosofías" conforman lo que se denomina estrategia. En el capítulo 14 se presentará la tecnología de prueba del software. Ese capítulo se concentrará en la estrategia de la prueba del software.

UN VISTAZO RÁPIDO

¿Qué es? El software se prueba para descubrir errores cometidos sin darse cuenta al realizar su diseño y construcción. ¿Pero cómo se realizan las pruebas? ¿Debe desarrollarse un plan formal para las pruebas? ¿Debe probarse el programa como un todo o sólo deben aplicarse pruebas a una parte pequeña? ¿Deben volver a realizarse las pruebas ejecutadas a medida que se agregan nuevos componentes a un sistema de gran tamaño? ¿Cuándo debe pedirse la participación del cliente? Éstas y muchas otras preguntas se responderán cuando desarrolle una estrategia de prueba del software.

¿Quién lo hace? El jefe de proyecto, los ingenieros de software o los especialistas en pruebas son quienes desarrollan la estrategia para la prueba del software.

¿Por qué es importante? Con frecuencia, la prueba requiere una mayor cantidad del esfuerzo dedicado al proyecto que cualquier otra actividad de ingeniería del software. Si se realiza

...n un plan, se desperdicia tiempo, se dedica un esfuerzo innecesario y, aún peor, es posible que no se detecten errores. Por tanto, lo razonable sería establecer una estrategia sistemática para probar el software.

¿Cuáles son los pasos? La prueba empieza por lo "pequeño" y avanza hacia lo "grande". Esto significa que, en las primeras etapas, la prueba se concentra en un solo componente o en un grupo pequeño de componentes relacionados y se aplica para descubrir errores en la lógica de datos y del procesamiento que se ha encapsulado en el componente. Una vez probados los componentes, deben integrarse hasta que todo el sistema se haya construido. En este punto se ejecuta una serie de pruebas de alto nivel para descubrir errores en la satisfacción de los requisitos del cliente. A medida que se descubren los errores deben diagnosticarse y corregirse empleando un proceso llamado depuración.

¿Cuál es el producto obtenido? Una Especificación de la prueba documenta el enfoque que aplicó el equipo de software a la prueba al definir un plan que detalla una estrategia general y un procedimiento que describe los pasos específicos que se darán y las pruebas que habrán de realizarse.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Al revisar la Especificación de la prueba antes de realizarla se evalúa si están completos los casos y las tareas de prueba. Un plan y un procedimiento de prueba efectivos llevarán a la construcción ordenada del software y al descubrimiento de errores en cada etapa del proceso de construcción.

13.1 UN ENFOQUE ESTRATÉGICO PARA LA PRUEBA DEL SOFTWARE

La prueba es un conjunto de actividades que se planean con anticipación y se realizan de manera sistemática. Por tanto, se debe definir una plantilla para las pruebas del software (un conjunto de pasos en que se puedan incluir técnicas y métodos específicos del diseño de casos de prueba).

Se han propuesto varias estrategias de prueba del software en distintos libros; todas proporcionan al desarrollador del software una plantilla para pruebas y todas tienen las siguientes características genéricas:

- Para realizar pruebas efectivas un equipo de software debe efectuar revisiones técnicas formales y efectivas (capítulo 26). Esto eliminará muchos errores antes de que empiece la prueba.
- La prueba comienza al nivel de componentes y trabaja "hacia fuera", hacia la integración de todo el sistema de cómputo.
- Diferentes técnicas de prueba son apropiadas en diferentes momentos.
- La prueba la dirige el desarrollador del software y (en el caso de proyectos grandes) un grupo independiente de pruebas.
- La prueba y la depuración son actividades diferentes, pero la segunda debe incluirse en cualquier estrategia de prueba.

Una estrategia para la prueba del software debe incluir pruebas de bajo nivel (necesarias para confirmar la correcta implementación de un pequeño segmento de código fuente) y de alto nivel (que validen las principales funciones del sistema a partir

de los requisitos del cliente). Una estrategia debe servir como guía para el profesional y fijar un conjunto de guías para el jefe de proyecto. Debido a que los pasos de la estrategia de prueba son simultáneos, cuando empieza a aumentar la presión a las fechas límite debe tenerse la opción de medir los avances y buscar que los problemas aparezcan lo antes posible.

13.1.1 Verificación y validación

La prueba del software es un elemento de un tema más amplio que suele denominarse verificación y validación (VyV). *Verificación* es el conjunto de actividades que aseguran que el software implemente correctamente una función específica. *Validación* es un conjunto diferente de actividades que aseguran que el software construido corresponde con los requisitos del cliente.[1] Boehm [BOE81] lo establece de otra manera

Verificación: "¿Estamos construyendo el producto correctamente?"

Validación: "¿Estamos construyendo el producto correcto?"

La definición de VyV abarca muchas de las actividades incluidas en el aseguramiento de la calidad del software y se analiza de manera detallada en el capítulo 26.

La verificación y la validación abarcan una amplia lista de actividades de aseguramiento de la calidad del software: revisiones técnicas formales, auditorías de calidad y de configuración, monitoreo del desempeño, simulación, factibilidad, revisión de la documentación y la base de datos, análisis de algoritmos, pruebas de desarrollo, de facilidad de uso, calificación y de instalación [WAL89]. Aunque las actividades de prueba tienen un papel demasiado importante en VyV, también se necesitan muchas otras actividades.

CONSEJO

No se deben tener descuidos ni considerar que las pruebas son una "red de seguridad" que atrapará todos los errores que ocurran debido a la aplicación de prácticas débiles de ingeniería del software. No lo son. Debe ponerse especial cuidado en la calidad y detección de errores en todo el proceso de la ingeniería del software.

> "Probar es la parte inevitable de cualquier esfuerzo responsable por desarrollar un sistema de software."
>
> **William [illegible]**

Las pruebas son el último bastión para la evaluación de la calidad y, de manera más pragmática, el descubrimiento de errores. Pero las pruebas no deben considerarse como una "red de seguridad". Como suele decirse: "No es posible probar la calidad. Si no está ahí antes de que empiece la prueba, no estará cuándo se termine". La calidad se incorpora al software en todo el proceso de ingeniería. La aplicación apropiada de métodos y herramientas, las revisiones técnicas formales y efectivas

[1] Debe indicarse que hay una fuerte divergencia de opinión acerca de los tipos de prueba que constituyen una "validación". Algunas personas creen que toda prueba es una verificación, y que la validación se realiza cuando el usuario revisa y aprueba los requisitos, y más delante, cuando el sistema está en condiciones de operar. Otras personas consideran que la prueba de la unidad y la integración (secciones 13.3.1 y 13.3.2) constituyen la verificación y que las pruebas de alto nivel (abordadas más adelante en este capítulo) son la validación.

junto con una administración y una medición sólidas aportan la calidad, que se confirma durante las pruebas.

Miller [MIL77] relaciona la prueba del software con el aseguramiento de la calidad al afirmar: "lo que motiva la prueba de los programas es la confirmación de la calidad del software con métodos que se puedan aplicar de manera económica y efectiva en sistemas grandes y pequeños"

13.1.2 Organización para las pruebas del software

En cualquier proyecto de software se presenta un conflicto de intereses cuando comienzan las pruebas. Ahora se pide a las personas que han construido el software que lo prueben En sí, esto parece inofensivo; después de todo, ¿quién conoce mejor un programa que la persona que lo desarrolló? Por desgracia, a esos mismos desarrolladores les interesa mucho demostrar que el programa está libre de errores, que funciona de acuerdo con los requisitos del cliente y que se completará a tiempo y sin rebasar el presupuesto. Cada uno de estos intereses mina las bondades de la prueba.

> "El optimismo es el peligro ocupacional de la programación; la prueba, el tratamiento."
>
> **Kent Beck**

Desde un punto de vista psicológico, el análisis y el diseño del software (junto con la codificación) son tareas constructivas El ingeniero del software analiza, modela y luego crea un programa de computadora, junto con su documentación. Como cualquier constructor, el ingeniero del software se sentirá orgulloso del edificio que acaba de construir y mirará con recelo a cualquiera que pretenda echarlo abajo. Cuando comienza la prueba hay un intento sutil, pero definitivo, de "romper" lo que ha construido el ingeniero del software Al ponerse en los zapatos del constructor la prueba parecerá (psicológicamente) destructiva De modo que el constructor actuará con cuidado, diseñando y ejecutando pruebas que demostrarán el buen funcionamiento del programa en lugar de descubrir errores. Por desgracia, los errores seguirán presentes. Y si el ingeniero del software no los encuentra, ¡el cliente sí lo hará!

De las consideraciones precedentes suelen inferirse erróneamente varias malas interpretaciones: 1) que el responsable del desarrollo no debería participar en el proceso de prueba, 2) que el software debe ponerse a salvo de extraños que lo prueben sin misericordia, y 3) que quienes aplican las pruebas sólo deben participar en el proyecto cuando vayan a darse los primeros pasos de esas pruebas Todas estas afirmaciones son incorrectas.

El desarrollador del software siempre será el responsable de probar las unidades individuales (componentes) del programa y asegurar que cada una realice la función o muestre el comportamiento para el que se diseñó En muchos casos, el desarrollador también aplica la prueba de integración (un paso que lleva a la construcción, y la prueba, de toda la arquitectura del software). Sólo después de que la arquitectura del software esté completa participará un grupo independiente de prueba.

PUNTO CLAVE

grupo independiente de prueba no tiene el conflicto de que experimentan los constructores del software.

CONSEJO

Si no existe un GIP dentro de una organización, tendrá que adoptarse su punto de vista propio. Al aplicar la prueba se debe tratar de destruir el software.

El papel de un *grupo independiente de prueba* (GIP) consiste en eliminar los problemas propios de dejar que el constructor pruebe lo que él mismo ha construido. La prueba independiente elimina el conflicto de intereses que, de otra manera, estaría presente. Después de todo, al personal del GIP se le paga para que encuentre errores.

Sin embargo, el ingeniero del software no debe simplemente entregar el programa al GIP y alejarse. El desarrollador y el GIP deben trabajar unidos en todo el proyecto de software para asegurar la realización de pruebas exhaustivas. Mientras éstas se realizan, el desarrollador debe estar disponible para corregir los errores que se descubran.

> "El primer error que comete la gente es pensar que el equipo de pruebas es responsable de asegurar la calidad."
>
> **Brian Marick**

El GIP es parte del equipo del proyecto de desarrollo del software, ya que participa en el análisis y diseño y además sigue participando (al planear y especificar los procedimientos de prueba) en todos los pasos de un proyecto grande. Sin embargo, en muchos casos el GIP informa a la organización de aseguramiento de calidad del software, por lo que obtiene un grado de independencia que sería imposible si fuera parte de la organización encargada de la ingeniería del software.

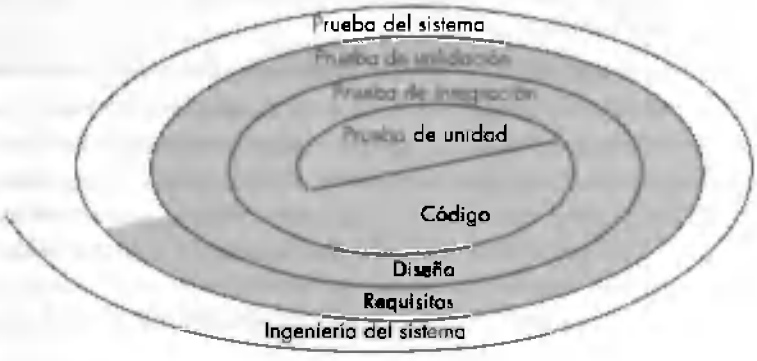
13.1.3. Estrategia de prueba para arquitecturas convencionales del software

Sería factible considerar que el proceso de ingeniería del software es equiparable a la espiral que se ilustra en la figura 13.1. Al principio, la ingeniería del sistema define el papel del software y lleva al análisis de los requisitos de éste, donde se establecen el dominio de información, la función, el comportamiento, el desempeño, las restricciones y los criterios de validación del software. Al desplazarse hacia el interior de la espiral se llega al diseño y, por último, a la codificación. El desarrollo de software de computadora requiere recorrer la espiral hacia dentro, a lo largo de una línea bien definida que disminuye el grado de abstracción tras cada vuelta.

? ¿Cuál es la estrategia general para la prueba del software?

También es posible ver una estrategia para la prueba del software en el contexto de la espiral (figura 13.1). La prueba de unidad comienza en el vértice de la espiral y se concentra en cada unidad (componente) del software, tal como se implementó en el código fuente. La prueba avanza al desplazarse hacia fuera, a lo largo de la espiral, hasta llegar a la *prueba de integración*, donde se atiende el diseño y la construcción de la arquitectura del software. Si se recorre otra vuelta hacia fuera en la espiral, se encuentra la *prueba de validación*, donde se validan los requisitos establecidos como parte del análisis de requisitos del software, comparándolos con el software que se ha construido. Por último, se llega a la *prueba del sistema*, donde se prueba el software y otros elementos del sistema. El software de computadora se prueba recorriendo la espiral hacia fuera, por una línea bien definida, de manera que en cada vuelta se ensancha el alcance de la prueba.

Si se considera el proceso desde un punto de vista procedimental, en realidad la prueba dentro del contexto de la ingeniería del software consiste en una serie de cuatro pasos que se implementan de manera secuencial. Esos pasos se muestran en la figura 13.2. Al principio, la prueba se concentra en cada componente individual, asegurando que funciona de manera apropiada como unidad (por ello se le denomina *prueba de unidad*). La prueba de unidad emplea en forma recurrente las técnicas de prueba que recorren caminos específicos en una estructura de control del componente, lo que asegura una cobertura completa y una detección máxima de errores. Enseguida deben ensamblarse o integrarse los componentes para formar el paquete de software completo. La prueba de integración atiende todos los aspectos asociados con el doble problema de verificación y construcción del programa. Las técnicas de diseño de casos de prueba que se concentran en entradas y salidas son



más dominantes durante la integración, aunque podrían usarse técnicas que re... rren rutas específicas del programa para asegurar la cobertura de los principale... caminos de control. Después de que se ha integrado (construido) el software se ap... ca un conjunto de pruebas de alto nivel. Se deben evaluar los criterios de validac... establecidos durante el análisis de requisitos. La prueba de validación proporci... un aseguramiento final de que el software cumple con todos los requisitos funci... nales, de comportamiento y desempeño.

El último paso de la prueba de alto nivel queda fuera de los límites de la inge... ría del software, pero dentro de un contexto más amplio de la ingeniería de los s... temas de cómputo. El software, una vez validado, debe combinarse con otros e... mentos del sistema (por ejemplo, hardware, personas, bases de datos). La prueba ... sistema verifica que todos los elementos encajen apropiadamente y que se logre ... función y el desempeño generales del sistema.

13.1.4 Estrategia de prueba del software para arquitecturas orientadas a objetos

La prueba de sistemas orientados a objetos plantea un conjunto diferente de desa... al ingeniero del software. La definición de prueba debe ampliarse para incluir té... cas de descubrimiento de errores (por ejemplo, revisiones técnicas formales) que ... aplican para analizar y diseñar modelos. El grado al que se han completado y la ... sistencia de las representaciones orientadas a objetos deben evaluarse a medida ... se construyen. La prueba de unidad pierde parte de su significado, y las estrate... de integración cambian de manera importante. En resumen, las estrategias y las ... ticas de prueba (capítulo 14) deben ser responsables de las características únicas ... software orientado a objetos.

¡PUNTO CLAVE

Como las pruebas convencionales, las orientadas a objetos empiezan por lo "pequeño". Sin embargo, en casi todos los casos, el elemento más pequeño probado es una clase o un paquete de clases que colaboran entre sí.

La estrategia general para el software orientado a objetos tiene una filosofía ... tica a la que se aplica a las arquitecturas convencionales, pero presenta diferen... en el enfoque. Se empieza "probando lo pequeño" y se trabaja hacia el exterior ... bando lo grande". Sin embargo, la atención cambia cuando la prueba es pequeña ... un módulo individual (el concepto convencional) a una clase que abarca atribu... operaciones y que, además, requiere comunicación y colaboración. A medida q...

HOGARSEGURO

Preparación para la prueba

La escena: Oficina de Doug Miller, mientras continúa el diseño al nivel de componentes y empieza la construcción de ciertos componentes.

Los actores: Doug Miller, jefe de ingeniería del software, Vinod, Jamie, Ed y Shakira, integrantes del equipo de ingeniería del software de *HogarSeguro*.

La conversación:

Doug: Me parece que no hemos dedicado el tiempo suficiente a hablar de las pruebas.

Vinod: Tienes razón, pero todos hemos estado un ... atareados. Y además hemos pensado en ella ... en realidad, hemos hecho más que pensarlo.

(sonriendo): Lo sé... tenemos exceso de ... pero aún tenemos que pensar en las cosas ...les.

...Me gusta la idea de diseñar pruebas de ... antes de empezar a codificar cualquiera de los ...ntes, de modo que eso es lo que hemos tratado ... Tengo un enorme archivo de pruebas que ... ejecutar una vez que esté completo el código de ... componentes.

...Ese es el concepto de Programación Extrema (un ... ágil de desarrollo de software visto en el capítulo ... ¿no?

... Así es. Aunque no estamos usando, en sí, la ...mación Extrema, decidimos que sería una buena ... diseñar pruebas de unidad antes de construir el ... (el diseño nos da toda la información que ...os).

Jamie: Yo he estado haciendo lo mismo.

Vinod: Y he tomado el papel de integrador, de modo que cada vez que uno de los muchachos me pase un componente, lo integraré y ejecutaré una serie de pruebas de regresión en el programa parcialmente integrado. He estado trabajando para diseñar un conjunto de pruebas apropiado para cada función del sistema.

Doug (a Vinod): ¿Con cuánta frecuencia ejecutarás las pruebas?

Vinod: Todos los días ... hasta que se integre bien el sistema. O sea, hasta que los incrementos de software que planeamos entregar queden integrados.

Doug: ¡Muchachos, van adelante de mí!

Vinod (sonriendo): La anticipación lo es todo en el negocio del software, ¡jefe.

integran clases dentro de una arquitectura orientada a objetos, se ejecuta una serie de pruebas de regresión para descubrir errores debidos a la comunicación y colaboración entre clases (componentes) y a los efectos colaterales que origina la adición de nuevas clases (componentes). Por último, se prueba el sistema como un todo para asegurarse de que se descubran errores en los requisitos.

13.1.5 Criterios para completar la prueba

¿Cuándo hemos terminado las pruebas?

Cada vez que se analizan las pruebas del software surge una pregunta clásica: ¿cuándo hemos terminado las pruebas (cómo sabemos que hemos probado lo suficiente)? Lo lamentable es que no hay una respuesta definitiva, sino que hay algunas respuestas pragmáticas y algunos intentos iniciales de sentar una guía empírica.

Una respuesta a la pregunta es: "nunca se termina de aplicar una prueba; la carga simplemente se desplaza de usted (el ingeniero del software) a su cliente. Cada vez que el cliente, el usuario, o ambos, ejecutan un programa de computadora, éste se está probando. Este hecho incuestionable subraya la importancia de otras actividades del aseguramiento de la calidad del software.

Otra respuesta (un poco cínica, pero correcta) es: "la prueba se termina cuando se agota el tiempo o el dinero".

Aunque algunos usarán esta respuesta como argumento, un ingeniero del software necesita criterios más rigurosos para determinar que las pruebas han sido suficientes. Musa y Ackerman [MUS89] sugieren una respuesta basada en criterios estadísticos: "No, no podemos estar completamente seguros de que el software nunca fallará, pero de acuerdo con un modelo estadístico teóricamente sólido y validado en forma experimental, hemos realizado las pruebas suficientes como para afirmar, con una confianza del 95%, que las probabilidades de tener mil horas de operaciones del

CPU libres de fallas en un entorno definido de forma probabilística es por lo menos de 0.995." Empleando el modelado estadístico y la teoría de la confiabilidad del software, pueden desarrollarse modelos de fallas del software (descubiertas durante la prueba) como una función del tiempo de ejecución (por ejemplo, consúltese [MUS89], [SIN99] o [IEE01]).

Al recopilar métricas durante la prueba del software y usar modelos existentes de la confiabilidad del software, es posible desarrollar directrices significativas para responder la pregunta: ¿Cuándo hemos terminado la prueba? Lo indiscutible es que hace falta mucho trabajo antes de que puedan establecerse reglas cuantitativas para la prueba, pero los enfoques empíricos existentes son considerablemente mejores que la simple intuición.

13.2 ASPECTOS ESTRATÉGICOS

Más adelante, en este mismo capítulo, se explorará una estrategia sistemática para la prueba del software. Pero hasta la mejor estrategia fallará si no atiende una serie de aspectos predominantes. Tom Gilb [GIL95] argumenta que deben atenderse los siguientes temas, si se desea implementar con éxito una estrategia de prueba del software:

¿Cuáles directrices llevan a una estrategia de prueba del software que tenga éxito?

Especificar los requisitos del producto de manera cuantificable mucho antes de que empiecen las pruebas. Aunque el objetivo primordial de la prueba es encontrar errores, una buena estrategia de prueba también evalúa otras características de la calidad, como las opciones de llevarla a otra plataforma, además de la facilidad de mantenimiento y uso (capítulo 15). Esto debe especificarse de manera tal que pueda medirlo para que los resultados de la prueba no resulten ambiguos.

Establecer explícitamente los objetivos de la prueba. Los objetivos específicos de la prueba se deben establecer en términos cuantificables. Por ejemplo, dentro del plan de prueba deben establecerse la efectividad y la cobertura de la prueba, el tiempo medio de falla, el costo de encontrar y corregir defectos, la densidad o la frecuencia de las fallas restantes, y las horas de trabajo por prueba de regresión [GIL95].

Comprender cuáles son los usuarios del software y desarrollar un perfil para cada categoría de usuario. Los casos de uso que describan el escenario de interacción para cada clase de usuario reducen el esfuerzo general de prueba, ya que concentran la prueba en la utilización real del producto.

Desarrollar un plan de prueba que destaque la "prueba de ciclo rápido". Gilb [GIL95] recomienda que un equipo de ingeniería del software "aprenda a probar en ciclos rápidos (2% del esfuerzo del proyecto) los incrementos en el mejoramiento de la funcionalidad, la calidad, o ambas, de manera que sean útiles para el cliente y se puedan probar en el campo". La retroalimentación que generan estas pruebas de ciclo rápido se utiliza para controlar los grados de calidad y las correspondientes estrategias de prueba.

Herramientas Web

En este sitio Web se encontró una excelente lista de herramientas para pruebas: www.io.com/~wazmo/qa/.

Construir un software "robusto" diseñado para probarse a sí mismo. El software debe diseñarse de manera tal que use técnicas antierror (sección 13.3.1). Es decir, el software debe tener la capacidad de diagnosticar ciertas clases de errores. Además, el diseño debe incluir pruebas automatizadas y de regresión.

Usar revisiones técnicas formales y efectivas como filtro previo a la prueba. Las revisiones técnicas formales (capítulo 26) llegan a ser tan efectivas como las pruebas para descubrir errores. Por tanto, las revisiones reducen la cantidad de esfuerzo de prueba que se requiere para producir software de alta calidad.

Realizar revisiones técnicas formales para evaluar la estrategia de prueba y los propios casos de prueba. Las revisiones técnicas formales posibilitan descubrir inconsistencias, omisiones y errores evidentes en el enfoque de la prueba. Esto ahorra tiempo y también mejora la calidad del producto.

Desarrollar un enfoque de mejora continua para el proceso de prueba. Es necesario medir la estrategia de prueba. Las medidas reunidas durante la prueba deben usarse como parte de un enfoque estadístico de control del proceso para la prueba del software.

> "Probar únicamente los requisitos del usuario final es como inspeccionar un edificio considerando únicamente el trabajo realizado por el diseñador de interiores, a costa de los cimientos, las vigas y la plomería."
>
> **Boris Beizer**

13.3 ESTRATEGIAS DE PRUEBA PARA EL SOFTWARE CONVENCIONAL

En la prueba del software es posible aplicar muchas estrategias. En un extremo, un equipo de software podría esperar hasta que el sistema esté totalmente construido y luego aplicar pruebas al sistema general esperando encontrar errores. Este enfoque, aunque atractivo, simplemente no funciona. Arrojará un software plagado de errores, molesto para el cliente y usuario final. En el otro extremo, un ingeniero de software podría aplicar pruebas diariamente, sin importar la parte del sistema que se construya. Este enfoque, aunque menos atractivo para muchos, es muy efectivo. Por desgracia, la mayoría de los desarrolladores de software dudan en usarlo. ¿Qué hay que hacer?

La estrategia de prueba que elige la mayor parte de los equipos de software se ubica entre estos dos extremos. Toma un enfoque incremental de las pruebas; inicia con la prueba de unidades individuales del programa, pasa a pruebas diseñadas para facilitar la integración de las unidades, y culmina con pruebas que realizan sobre el sistema construido. En las siguientes secciones se expone cada una de estas clases de prueba.

13.3.1 Prueba de unidad

La *prueba de unidad* se concentra en el esfuerzo de verificación de la unidad más pequeña del diseño del software: el componente o módulo de software. Tomando como guía la descripción del diseño al nivel de componentes, se prueban importantes caminos de control para descubrir errores dentro de los límites del módulo. El alcance restringido que se ha determinado para las pruebas de unidad limita la relativa complejidad de las pruebas y los errores que éstas descubren. Las pruebas de unidad se concentran en la lógica del procesamiento interno y en las estructuras de datos dentro de los límites de un componente. Este tipo de prueba se puede aplicar en paralelo a varios componentes.

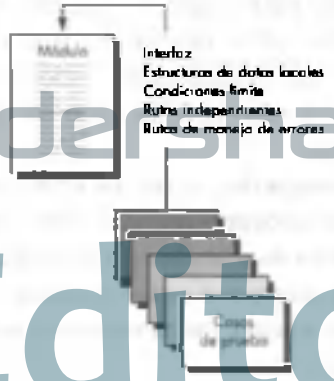
Consideraciones sobre la prueba de unidad. En la figura 13.3 se ilustran de manera esquemática las pruebas que se aplican como parte de la prueba de unidad. La interfaz del módulo se prueba para asegurar que la información fluye apropiadamente hacia dentro y hacia fuera de la unidad de programa sujeta a prueba. Se examinan las estructuras de datos locales para asegurar que los datos temporales mantienen la integridad durante todos los pasos de la ejecución de un algoritmo. Se recorren todos los caminos independientes (caminos de base) en toda la estructura para asegurar que todas las instrucciones de un módulo se hayan ejecutado por lo menos una vez. Se prueban las condiciones límite para asegurar que el módulo opera apropiadamente en los límites establecidos para restringir el procesamiento. Y, por último, se prueban todos los caminos de manejo de errores.

?) ¿Cuáles errores se encuentran comúnmente durante la prueba de unidad?

Es necesario probar el flujo de datos en la interfaz del módulo antes de iniciar cualquier otra prueba. Si los datos no entran ni salen apropiadamente, todas las demás pruebas resultarán inútiles. Además, durante la prueba de unidad deben recorrerse las estructuras de datos locales y evaluarse (si es posible) el impacto local sobre los datos globales.

Durante la prueba de unidad, una tarea esencial es la prueba selectiva de las rutas de ejecución. Se deben diseñar casos de prueba para descubrir errores debidos a cálculos incorrectos, comparaciones erróneas o flujos de control inapropiados. Entre los erro-

FIGURA 13.3
Prueba de unidad.



res más comunes en los cálculos se encuentran los siguientes: 1) aplicación incorrecta o mal entendida de la precedencia aritmética, 2) operaciones de modo mezcladas, 3) inicialización incorrecta, 4) falta de precisión, y 5) representación simbólica incorrecta de una expresión. La comparación y el flujo de control están estrechamente acoplados entre sí (es decir, el flujo cambia con frecuencia después de una comparación). Los casos de prueba deben descubrir errores como: 1) comparaciones entre diferentes tipos de datos, 2) operadores lógicos o precedencia de éstos aplicados incorrectamente, 3) expectativa de igualdad cuando los errores de precisión hacen que sea poco probable, 4) comparación incorrecta de variables, 5) terminación inapropiada o inexistente de bucles, 6) falla en la salida cuando se encuentra una iteración divergente, y 7) variables de bucle modificadas de manera inapropiada.

La prueba de límites es una de las tareas más importantes de la prueba de unidad. Con frecuencia, el software falla en sus límites. Es decir, a menudo los errores ocurren cuando se procesa el enésimo elemento de una matriz de *n* dimensiones, cuando se evoca la *i*-ésima repetición de un bucle con *i* pasos, cuando se encuentra el valor máximo o mínimo permisible. Es muy probable descubrir errores en los casos de prueba que se ejercen sobre la estructura de datos, el flujo de control y los valores de datos ubicados apenas abajo de los máximos o mínimos, sobre éstos y apenas arriba de ellos.

Información Web

Si se requiere alta [illegible] información útil sobre [illegible] variedad de [illegible] y recursos para "pruebas ágiles", www.testing.com/agile/.

Un buen diseño impone que se prevean las condiciones de error y que se configuren rutas de manejo de errores para modificar la ruta o terminar limpiamente el procesamiento cuando ocurra un error. Yourdon [YOU75] llama a este enfoque *antierror*. Por desgracia, existe la tendencia a incorporar el manejo de errores en el software y, con ello, a no probarlo nunca. Una historia real servirá como ejemplo:

> Un sistema de diseño asistido por computadora se desarrolló bajo contrato. En un módulo de procesamiento de transacciones, un bromista práctico puso el siguiente mensaje de manejo de errores después de una serie de pruebas condicionales que invocaban varias ramas del flujo de control: ¡ERROR! NO HAY FORMA DE QUE PUEDA LLEGAR HASTA AQUÍ. ¡Este "mensaje de error" lo descubrió un cliente durante la capacitación del usuario!

CONSEJO

Debe asegurarse que los pruebas de diseño ejerciten todos los casos de manejo de errores. Si no lo hacen, el usuario podría [illegible] cuando se le solicita, lo que ocasionará una [illegible] ya de por sí [illegible].

Entre los posibles errores que deben probarse cuando se evalúe el manejo de errores se encuentran los siguientes: 1) la descripción del error es ininteligible, 2) el error indicado no corresponde al encontrado, 3) la condición de error causa la intervención del sistema operativo antes de que se dispare el manejo de errores, 4) el procesamiento de la condición de excepción es incorrecto, 5) la descripción del error no proporciona información suficiente para ayudar a localizar la causa del error.

Procedimientos de prueba de unidad. La prueba de unidad suele considerarse adyacente al paso de la codificación. El diseño de las pruebas de unidad puede realizarse antes de que empiece la codificación (un enfoque ágil que suele preferirse) o después de que se ha generado el código fuente. Una revisión de la información del

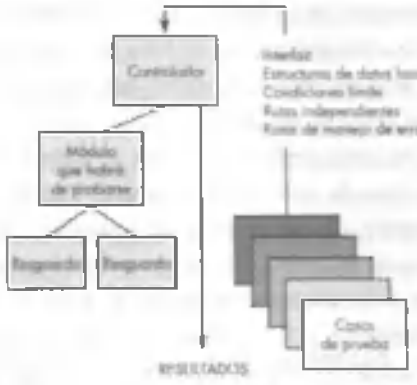


FIGURA 13.4
Entorno de prueba de unidad.

diseño proporciona una guía para establecer casos de prueba que probabl descubrirán errores en cada una de las categorías analizadas. Cada caso de pru debe relacionarse con un conjunto de resultados esperados.

Debido a que un componente no es un programa independiente, para cada pru ba de unidad se debe desarrollar software *controlador*, de resguardo, o de ambos tipos. En la figura 13.4 se ilustra el entorno para la prueba de unidad. En casi todas las aplicaciones, un controlador no es más que un "programa principal" que acepta los datos del caso de prueba, pasa estos datos al componente (que habrá de probar se) e imprime los resultados importantes. Los *resguardos* sirven para reemplazar módulos subordinados al componente que habrá de probarse (o llamados por éste). Un resguardo o "subprograma simulado" usa la interfaz del módulo subordinado, realiza una mínima manipulación de datos, proporciona verificación de la entrada y devuelve el control al módulo de prueba

CONSEJO

Hay situaciones en que no se tendrán los recursos para hacer una prueba de unidad muy completa. Entonces deben seleccionarse módulos críticos y los que tengan una elevada complejidad en ciclos, y sólo ésos deben probarse

Controladores y resguardos representan una sobrecarga de trabajo. Es decir, resulta necesario escribir ambos tipos de software (sin que suela aplicarse un diseño formal), pero no se entregan con el producto de software final. Si se les mantiene en un nivel simple, la sobrecarga real es relativamente pequeña. Por desgracia, no es posible aplicar adecuadamente una prueba de unidad a muchos componentes con un "simple" software de sobrecarga. En muchos casos es posible posponer la prueba completa hasta el paso de prueba de integración (donde también se utilizan controladores o resguardos).

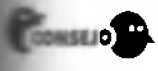
La prueba de unidad se simplifica cuando se diseña un componente con cohesión elevada. Cuando sólo se atiende una función de un componente, el número de casos de prueba se reduce y es más fácil predecir y corregir los errores

13.3.2 Prueba de integración

Un neófito en el mundo del software podría plantear una pregunta aparentemente legítima, una vez que se haya aplicado una prueba de unidad a todos los módulos: "Si todo funciona bien individualmente, ¿por qué dudan que funcione cuando

une?" El problema, por supuesto, consiste en "unir" (crear la interfaz). En una interfaz es posible perder datos, un módulo podría tener un efecto adverso e inadvertido sobre otro, la combinación de subfunciones tal vez no produzca la función principal deseada, la imprecisión aceptable en elementos individuales podría ampliarse hasta grados inaceptables y las estructuras globales de datos podrían presentar problemas. Es triste, pero la lista sigue y sigue.

La *prueba de integración* es una técnica sistemática para construir la arquitectura del software mientras, al mismo tiempo, se aplican las pruebas para descubrir errores asociados con la interfaz. El objetivo es tomar componentes a los que se aplicó una prueba de unidad y construir una estructura de programa que determine el diseño.



...enfoque de ...g" para la ...n es una ...perezoso ...al fracaso. ...realizar la ...n en incre- ...y probar ...s se avanza

A menudo se tiende a intentar una integración que no sea incremental; es decir, a construir el programa mediante un enfoque de "big bang". Se combinan todos los componentes por anticipado. Se prueba todo el programa como un todo. ¡Y se produce el caos! Se encuentra una gran cantidad de errores. La corrección es difícil, porque resulta complicado aislar las causas debido a la extensión del programa completo. Una vez corregidos esos errores, aparecen otros nuevos y el proceso continúa en un ciclo que parece interminable.

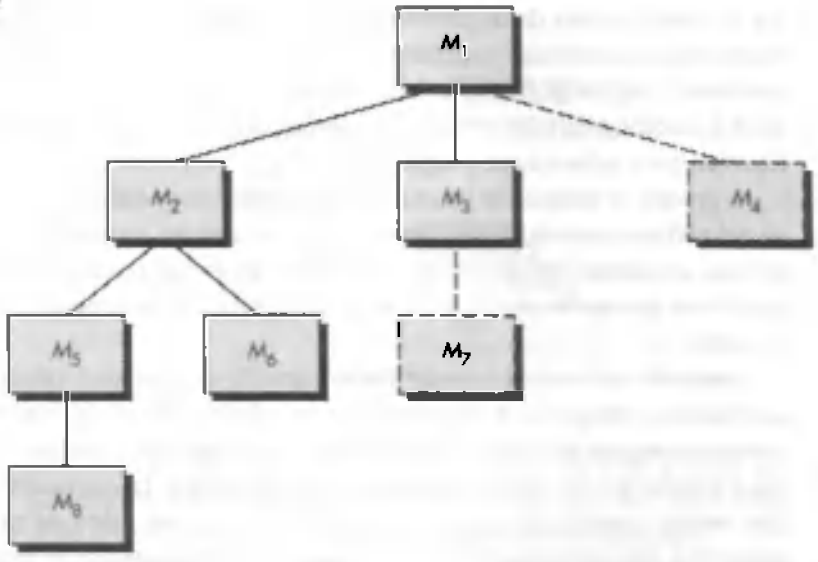
La *integración incremental* es la antítesis del enfoque del "big bang". El programa se construye y prueba en pequeños incrementos, en los cuales resulta más fácil aislar y corregir los errores, es más probable que se prueben por completo las interfaces y se vuelve factible la aplicación de un enfoque de prueba sistemática. En los siguientes párrafos se expondrán varias estrategias diferentes de integración incremental.

...se desarrolla ...o de un ...to, tiene que ...e la ...o en que ...a la integración, ...modo que los ...entes estén ...s cuando se ...necesite.

Integración descendente. *La prueba de integración descendente* es un enfoque incremental para la construcción de la arquitectura del software. Los módulos se integran al descender por la jerarquía de control, empezando con el módulo de control principal (programa principal). Los módulos subordinados al módulo de control principal se incorporan en la estructura de una de dos maneras: primero-en-profundidad o primero-en-anchura.

Tomando como referencia la figura 13.5, *la integración primero-en-profundidad* integra todos los módulos de una ruta de control principal de la estructura del programa. La selección de una ruta principal es un poco arbitraria y depende de las características específicas de la aplicación. Por ejemplo, si se elige el camino de la izquierda, se integrarían primero los módulos M_1, M_2 y M_5. A continuación, se integraría M_8 o (si es necesario para el adecuado funcionamiento de M_2) M_6. Enseguida se construyen las rutas de control central y a la derecha. La *integración primero en anchura* incorpora todos los componentes directamente subordinados en cada nivel, desplazándose horizontalmente por la estructura. En el caso de la figura, se integrarían primero los componentes M_2, M_3 y M_4. Y les seguirían M_5, M_6, etc. El proceso de integración se realiza en una serie de cinco pasos:

FIGURA 13.5
Integración descendente.



¿Cuáles son los pasos de la integración descendente?

1. El módulo de control principal se utiliza como controlador de prueba, y se sustituyen los resguardos en todos los componentes directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque de integración seleccionado (es decir, primero-en-profundidad o primero-en-anchura) se van reemplazando los resguardos subordinados, uno por uno, con los componentes reales.
3. Se aplican pruebas cada que se integra un nuevo módulo.
4. Al completar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real
5. Se aplica la prueba de regresión (que se analiza más adelante, en esta misma sección) para asegurarse de que no se han introducido nuevos errores.

El proceso continúa a partir del paso 2 hasta la construcción total de la estructura del programa

La estrategia de integración descendente verifica los principales puntos de control o decisión al principio del proceso de prueba. En una estructura de programa bien elaborada, la toma de decisiones ocurre en los niveles superiores de la jerarquía y, por tanto, se encuentran primero. Si existen problemas de control importantes resulta esencial reconocerlos desde el principio. Si se selecciona la integración primero-en-profundidad, es posible implementar y demostrar una función completa de software. Por ejemplo, imagínese una estructura de transacción clásica (capítulo 11) en que una serie compleja de entradas interactivas se solicita, adquiere y valida por medio de una ruta de entrada. Tal vez ese camino esté integrado en forma descendente. Todo el procesamiento de entrada (para el envío de las siguientes transacc

nes) podría demostrarse antes de que otros elementos de la estructura se hayan integrado. La demostración temprana de la capacidad funcional genera confianza en el desarrollador y en el cliente.

¿Cuáles problemas se encontrarán cuando se elija la integración descendente?

La estrategia descendente no parece muy complicada, pero en la práctica llegan a surgir problemas de logística. El más común se presenta cuando se requiere procesamiento en los niveles inferiores de la jerarquía para probar de manera adecuada los niveles superiores. Al principio de la prueba descendente se reemplazan los módulos de bajo nivel con resguardos; por tanto, no fluirán datos importantes hacia la parte superior de la estructura del programa. Quien aplica la prueba cuenta con tres opciones: 1) retrasar muchas de las pruebas hasta que los resguardos sean reemplazados con los módulos reales, 2) desarrollar resguardos que realicen funciones limitadas que simulen los módulos reales, o 3) integrar el software de la parte inferior de la jerarquía hacia arriba.

El primer enfoque (retrasar las pruebas hasta no reemplazar los resguardos con los módulos reales) hace perder cierto control sobre la correspondencia entre pruebas específicas y la incorporación de módulos específicos. Con esto se dificulta determinar la causa de los errores y se tiende a violar la naturaleza altamente restringida del enfoque descendente. Es posible trabajar con el segundo enfoque, pero puede llevar a un aumento importante de la sobrecarga de trabajo, a medida que los resguardos se vuelvan más y más complejos. El tercer enfoque, denominado *prueba ascendente*, se expondrá en la siguiente sección.

Integración ascendente. La *prueba de integración ascendente*, como su nombre lo indica, empieza la construcción y la prueba con módulos atómicos (es decir, componentes de los niveles más bajos de la estructura del programa). Debido a que los componentes se integran de abajo hacia arriba, siempre está disponible el procesamiento requerido para los componentes subordinados a un determinado nivel y se elimina la necesidad de resguardos. Una estrategia de integración ascendente se implementa mediante los siguientes pasos:

¿Cuáles son los pasos para una integración ascendente?

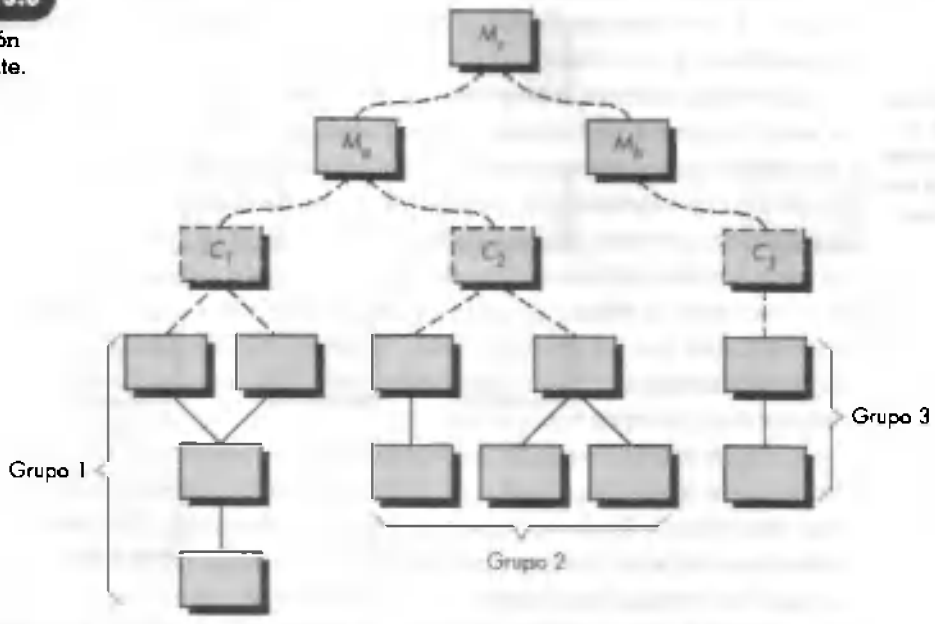
1. Se combinan los módulos de bajo nivel en grupos (también llamados construcciones) que realicen una subfunción específica del software.
2. Se escribe un controlador (un programa de control para pruebas) con el fin de coordinar la entrada y la salida de los casos de prueba.
3. Se prueba el grupo.
4. Se eliminan los controladores y se combinan los grupos ascendiendo por la estructura del programa.

CLAVE

La integración ascendente elimina la necesidad de resguardos complejos.

La integración sigue el patrón ilustrado en la figura 13.6. Los componentes se combinan para formar los grupos 1, 2 y 3. Cada uno de ellos se prueba empleando un controlador (mostrado como un recuadro con guiones). Los componentes de los grupos 1 y 2 están subordinados a M_a. Los controladores C_1 y C_2 se eliminan y los grupos interaccionan directamente con M_b. De igual manera, se elimina el controlador

FIGURA 13.6
Integración ascendente.



C_3 del grupo 3 antes de la integración con el módulo M_b. M_a y M_b se integrarán finalmente con el módulo M_c, y así sucesivamente.

A medida que la integración asciende se reduce la necesidad de controladores de prueba separados. En realidad, si los dos niveles superiores de la estructura del programa se integran de manera descendente, se reducirá de manera importante el número de controladores y se simplificará enormemente la integración de grupos.

CONSEJO

La prueba de regresión es una estrategia importante para reducir "efectos colaterales". Deben aplicarse pruebas de regresión cada vez que se haga un cambio importante al software (incluida la integración de nuevos componentes).

Prueba de regresión. Cada vez que se agrega un nuevo módulo como parte de una prueba de integración, el software cambia. Se establecen nuevas rutas de flujo de datos, ocurren nuevas entradas y salidas y se invoca una nueva lógica de control. Estos cambios llegan a causar problemas con funciones que antes funcionaban bien. En el contexto de una estrategia de prueba de integración, la aplicación de una prueba de regresión consiste en ejecutar nuevamente el mismo subconjunto de pruebas que ya se ha aplicado para asegurarse de que los cambios no han propagado efectos colaterales indeseables.

En un contexto más amplio, si las pruebas (de cualquier tipo) tienen éxito, el resultado es el descubrimiento de errores, y éstos deben corregirse. Cada vez que el software se corrige también cambia algún aspecto de la configuración del software (el programa, su documentación o los datos de soporte). La prueba de regresión es una actividad que ayuda a asegurar que los cambios (debidos a la prueba u otras razones) no introduzcan comportamiento indeseable o errores adicionales.

La prueba de regresión se aplica manualmente, al ejecutar de nueva cuenta un subconjunto de todos los casos de prueba o al emplear herramientas automáticas

captura, reproducción, o ambas. Las *herramientas de captura, reproducción*, o de ambos tipos, permiten al ingeniero del software capturar casos de prueba y resultados para reproducirlos y compararlos después. El conjunto de pruebas de regresión (el subconjunto de pruebas que se aplicarán) contiene tres clases diferentes de casos de prueba:

- Una muestra representativa de pruebas que ejercerán todas las funciones del software.
- Pruebas adicionales que se concentran en las funciones del software que probablemente el cambio afectaría
- Pruebas que se concentran en los componentes del software que han cambiado.

A medida que avanza la prueba de integración, la cantidad de pruebas de regresión llega a volverse muy grande. Por tanto, el conjunto de pruebas de regresión debe diseñarse para que sólo incluya las que atienden una o más clases de errores en cada una de las funciones principales del programa. Resulta poco práctico e ineficiente repetir cada prueba para todas las funciones del programa después de que se ha presentado un cambio.

Prueba de humo. La *prueba de humo* es un enfoque de prueba de integración que suele utilizarse mientras se desarrollan productos de software. Está diseñado como mecanismo para marcar el ritmo en proyectos en los cuales el tiempo es crítico, lo que permite que el equipo de software evalúe su proyecto con frecuencia. En esencia, el enfoque de la prueba de humo abarca las siguientes actividades:

1. Los componentes de software traducidos a código se integran en una "construcción", la cual incluye todos los archivos de datos, las librerías, los módulos reutilizables y los componentes de ingeniería que se requieren para implementar una o más funciones del producto.
2. Se diseña una serie de pruebas para exponer errores que impedirán que la construcción realice apropiadamente su función. El objetivo es descubrir errores "paralizantes" que tengan la mayor probabilidad de retrasar el proyecto de software.
3. La construcción se integra con otras construcciones, y *diariamente* se aplica una prueba de humo a todo el producto (en su forma actual). El enfoque de la integración puede ser descendente o ascendente

PUNTO CLAVE

la prueba de humo
[illegible] caracterizarse
[illegible] una estrategia
[illegible]
[illegible] Se
[illegible] el software
[illegible] nuevos
[illegible]
[illegible] y se aplica
[illegible] prueba de humo
[illegible] días.

La aplicación diaria de una prueba a todo el producto sorprenderá a algunos lectores. Sin embargo, las pruebas frecuentes dan a los jefes de proyecto y participantes una evaluación realista del avance de las pruebas de integración. McConnell [MCO96] describe así la prueba de humo:

La prueba de humo debe ejercitar todo el sistema de principio a fin. No debe ser exhaustiva, pero debe tener la capacidad de exponer problemas importantes. La prueba de humo debe ser tan completa que si la construcción la aprueba, puede suponerse que ésta es suficientemente estable como para probarla de manera más completa.

La prueba de humo proporciona varios beneficios cuando se aplica en proyectos de ingeniería del software complejos y que dependen en forma crítica del tiempo.

- *Se minimiza el riesgo en la integración.* Debido a que las pruebas de humo se aplican diariamente, desde el principio se descubren las incompatibilidades y otros errores paralizantes, por lo que se reduce la probabilidad de que tengan un fuerte impacto en el calendario cuando se descubran errores.
- *Se mejora la calidad del producto final.* Como el enfoque está orientado a la construcción (integración), es probable que la prueba de humo descubra errores funcionales, arquitectónicos y al nivel de componentes. Si estos errores se corrigen desde el principio, se obtendrá una mayor calidad en el producto.
- *Se simplifican el diagnóstico y la corrección de errores.* Como todos los elementos de prueba de integración, es probable que los errores no descubiertos en la prueba de humo estén asociados con "nuevos incrementos de software". El software que se acaba de añadir a la construcción es una posible causa de un error recién descubierto).
- *El progreso es más fácil de evaluar.* Cada día que pasa se integra más software y se demuestra que funciona. Esto mejora la moral del equipo y brinda a los jefes de proyecto una buena indicación de que se están logrando avances.

> "Trate la construcción diaria como si fuera el corazón del proyecto. Si no tiene corazón, el proyecto está muerto."
> Jim McCarthy

Opciones estratégicas. Ha habido mucha discusión (por ejemplo, [BEI8]) acerca de las ventajas y desventajas relativas de las pruebas de integración ascendente y descendente. En general, las ventajas de una estrategia tienden a convertirse en desventajas para la otra. La principal desventaja del enfoque descendente es la necesidad de resguardos y las dificultades de las pruebas asociadas. Los problemas relacionados con los resguardos se compensarían con la ventaja de probar las principales funciones de control en las primeras etapas. La principal desventaja de la integración ascendente es que "el programa, como una entidad, no existe hasta que se haya añadido el último módulo" [MYE79]. Esta desventaja la atenúa la mayor facilidad para diseñar casos de prueba y la falta de resguardos.

La selección de una estrategia de integración depende de las características del software y, en ocasiones, del calendario del proyecto. En general, la mejor opción es un enfoque combinado (a veces denominado *prueba sandwich*) que usa pruebas

Referencia Web
Apuntadores a comentarios sobre estrategias de prueba se encontrarán en: www.qalinks.com.

cendentes para los niveles superiores de la estructura del programa, junto con pruebas ascendentes para los niveles subordinados.

¿Qué es un módulo crítico y por qué identificarse?

A medida que se realiza la prueba de integración, el responsable debe identificar módulos críticos. Un *módulo crítico* tiene una o más de las siguientes características: 1) atiende varios requisitos del software, 2) tiene un alto grado de control (se encuentra en un lugar relativamente alto de la estructura del programa), 3) es complejo o propenso a errores, o 4) tiene requisitos de desempeño bien definidos. Los módulos críticos deben probarse lo antes posible. Además, las pruebas de regresión se deben concentrar en el funcionamiento de los módulos críticos.

Documentación de la prueba de integración. Un plan general para la integración del software y una descripción de pruebas específicas se documentan en la *Especificación de la prueba*. Este documento que contiene un plan de prueba, un procedimiento de prueba, es un producto de trabajo del proceso de software y se vuelve parte de la configuración del software.

El plan de prueba describe la estrategia general de integración. La prueba se divide en fases y construcciones que atienden características específicas del funcionamiento y el comportamiento del software. Por ejemplo, la prueba de integración para un sistema de diseño asistido por computadora se dividiría en las siguientes fases de prueba:

- Interacción del usuario (selección de comandos, creación de dibujos, representación del despliegue, procesamiento de errores y representación).
- Manipulación y análisis de datos (creación de símbolos, asignación de dimensiones, rotación, cálculo de propiedades físicas).
- Procesamiento y generación de despliegue (despliegues bi y tridimensionales, imágenes y gráficas).
- Administración de base de datos (acceso, actualización, integridad, desempeño).

Cada una de estas fases y subfases (denotadas entre paréntesis) delinean una amplia categoría funcional dentro del software y suelen relacionarse con un dominio específico dentro de la arquitectura del software. Por tanto, las construcciones del programa (grupos de módulos) se crean para que correspondan con cada fase. Los siguientes criterios y las pruebas correspondientes se aplican para todas las fases de prueba.

Integridad de la interfaz. Las interfaces internas y externas se prueban a medida que cada módulo (o grupo) se incorpora en la estructura.

Validez funcional. Se realizan las pruebas diseñadas para descubrir errores funcionales.

Contenido de la información. Se aplican las pruebas diseñadas para descubrir errores asociados con estructuras de datos locales o globales.

Desempeño. Se realizan las pruebas diseñadas para verificar los límites de desempeño establecidos durante el diseño del software.

Un calendario para la integración, el desarrollo de software de sobrecarga y los temas relacionados también se analizan como parte del plan de prueba. Se determinan las fechas de inicio y término para cada fase y se definen las "ventanas de disponibilidad" para los módulos de prueba de unidad. Una breve descripción del software de sobrecarga (resguardos y controladores) se concentra en las características que requieren esfuerzos especiales. Por último, se describen el entorno y los recursos de la prueba. Configuraciones poco comunes de hardware, simuladores específicos y herramientas especiales de prueba son algunos de los muchos temas que también podrían analizarse.

A continuación se describe el procedimiento detallado de prueba que se requiere para completar el plan respectivo. También se detalla el orden de integración y las pruebas correspondientes en cada paso de integración. Además, se incluye una lista de todos los casos de prueba (anotados para referencia posterior) y los resultados esperados.

Una historia de resultados, problemas o peculiaridades de las pruebas reales se registra en el *Informe de prueba* que puede adjuntarse a la *Especificación de prueba*. Si se desea, la información contenida en esta sección será vital durante el mantenimiento del software. También se presentan referencias y apéndices apropiados.

Como todos los demás elementos de una configuración de software, el formato de la especificación de prueba puede amoldarse a las necesidades locales de una organización de ingeniería del software. Sin embargo, es importante observar que una estrategia de integración (contenida en el plan de prueba) y los detalles de prueba (descritos en el procedimiento de prueba) son ingredientes esenciales y deben aparecer.

> "El mejor participante de una prueba no es el que encuentra más errores ... sino el que corrige la mayor cantidad de ..."
> **Cem Kaner et al.**

13.4 ESTRATEGIAS DE PRUEBA PARA SOFTWARE ORIENTADO A OB...

El objetivo de probar, para definirlo de manera simple, es encontrar la mayor cantidad de errores aplicando una cantidad manejable de esfuerzo en un periodo corto. Aunque este objetivo fundamental sigue sin cambio para el software orientado a objetos, la naturaleza de este software cambia la estrategia y la táctica de las pruebas (capítulo 14).

13.4.1 Prueba de unidad en el contexto orientado a objetos

Al pensar en el software orientado a objetos cambia el concepto de unidad. La encapsulación orienta la definición de clases. Esto significa que cada clase o instancia de una clase (objeto) empaqueta atributos (datos) y las operaciones (funciones) que manipulan estos datos. Una clase encapsulada suele ser el eje de las pruebas

unidad. Sin embargo, las unidades de prueba más pequeñas son las operaciones dentro de la clase. Debido a que una clase puede contener varias operaciones diferentes y a que una operación determinada puede existir como parte de varias clases diferentes, deben cambiar las tácticas aplicadas para la prueba de unidad.

PUNTO CLAVE
La prueba de clase para software orientado a objetos es equivalente a la prueba de unidad para software convencional. No es posible probar una operación de manera aislada

No es posible probar una sola operación de manera aislada (el concepto convencional de prueba de unidad) sino como parte de una clase. Para ilustrarlo, considérese una jerarquía de clase en que se define una operación *X* para la superclase y que heredan varias subclases. Cada una de éstas usa la operación *X*, pero se aplica dentro del contexto de los atributos privados y las operaciones que se han definido para la subclase. Dado que el contexto en que se emplea la operación *X* varía en formas sutiles, es necesario probar la operación en el contexto de cada una de las subclases. Esto significa que si se prueba la operación *X* de manera independiente (el enfoque de la prueba de unidad convencional) se podrá usar de manera deficiente en el contexto orientado a objetos.

La prueba de clase para el software orientado a objetos es el equivalente a la prueba de unidad para el software convencional. A diferencia de ésta, que tiende a concentrarse en el detalle algorítmico de un módulo y en los datos que fluyen por la interfaz del módulo, la prueba de clase para el software orientado a objetos se orienta mediante las operaciones que encapsula la clase y en el comportamiento de estado de la clase.

13.4.2 Prueba de integración en el contexto orientado a objetos

Debido a que el software orientado a objetos no tiene una estrategia obvia de control jerárquico, tienen poco significado las estrategias de integración descendente y ascendente tradicionales (sección 13.3.2). Además, integrar las operaciones una por una en una clase (el enfoque convencional de la integración incremental) suele resultar imposible debido a las "interacciones directas e indirectas de los componentes que integran la clase" [BER93].

PUNTO CLAVE
Una estrategia para la prueba de integración del software orientado a objetos es la prueba basada en subprocesos. Los subprocesos son conjuntos de clases que responden a una entrada o a un evento. Las pruebas basadas en el uso se concentran en las clases que no colaboran mucho con otras clases.

Hay dos estrategias diferentes para la prueba de integración de los sistemas orientados a objetos [BIN94]. La primera, la *prueba basada en subprocesos*, integra el conjunto de clases requerido para responder a una entrada o un evento del sistema. Cada subproceso se integra y prueba individualmente. La prueba de regresión se aplica para asegurar que no se presenten efectos colaterales. El segundo enfoque de integración, la *prueba basada en el uso*, empieza la construcción del sistema con la prueba de esas clases (llamadas *clases independientes*) que usan muy pocas clases de *servidor* (o ninguna). Después de que se prueban las clases independientes, se prueba la siguiente capa de clases, llamadas *clases dependientes*, que usan las clases independientes. Esta secuencia de capas de prueba de clases dependientes continúa hasta que se construye todo el sistema.

El uso de controladores y resguardos también cambia cuando se aplican pruebas de integración a los sistemas orientados a objetos. Con los controladores se prueban operaciones al nivel más bajo y grupos completos de clases. Un controlador también

se utiliza para reemplazar la interfaz de usuario, de modo que puedan aplicarse las pruebas de funcionalidad del sistema antes de la implementación de la interfaz. Los resguardos se usan en situaciones en que la colaboración entre clases es necesaria, pero en las cuales aún no se han implementado por completo una o más de las clases que colaboran.

La *prueba de grupo* es un paso en la prueba de integración del software orientado a objetos. Aquí, un grupo de clases que colaboran entre sí (determinadas por el examen del CRC y el modelo objeto-relación) se ejercita al diseñar casos de prueba que tratan de descubrir errores en las colaboraciones.

13.5 Pruebas de validación

¡PUNTO CLAVE

Como todos los demás pasos de prueba, en la validación se trata de descubrir errores, pero el punto central está en el nivel de los requisitos (o las cosas que serán inmediatamente evidentes para el usuario final).

Las *pruebas de validación* empiezan tras la culminación de la prueba de integración, cuando se han ejercitado los componentes individuales, se ha terminado de ensamblar el software como paquete y se han descubierto y corregido los errores de interfaz. En el nivel de validación o sistema desaparece la distinción entre software convencional y orientado a objetos. La prueba se concentra en las acciones visibles para el usuario y en la salida del sistema que éste puede reconocer.

La *validación* se define de muchas formas, pero una definición simple (aunque vulgar) es que se alcanza cuando el software funciona de tal manera que satisface las expectativas razonables del cliente. En este punto, un desarrollador de software experimentado protestaría: "¿Qué o quién decide lo que es una expectativa razonable?"

Las expectativas razonables se definen en la *Especificación de requisitos del software* (un documento que describe los atributos del software visibles para el usuario). La especificación contiene una sección denominada *Criterios de validación*. La información contenida en esa sección integra la base del enfoque de la prueba de validación.

13.5.1 Criterios de la prueba de validación

La validación del software se logra mediante una serie de pruebas que demuestran que se cumple con los requisitos. Un plan de prueba delinea la clase de pruebas que se aplicarán, y un procedimiento de prueba define los casos de prueba específicos. Tanto el plan como el procedimiento se diseñan para asegurar que se satisfacen todos los requisitos funcionales, que se alcanzan todas las características de comportamiento, que se cumple con todos los requisitos de desempeño, que la documentación es correcta y que se cumple también con todos los requisitos de facilidad de uso y otros requisitos especificados (por ejemplo, portabilidad, compatibilidad, recuperación de errores, facilidad de mantenimiento).

Después de que se ha dirigido cada caso de prueba de validación, existirá una de dos condiciones posibles: 1) la característica de funcionamiento o desempeño cumple con la especificación y se le acepta, o 2) se descubre una desviación de la especificación y se crea una lista de deficiencias. La desviación o el error descubierto

esta etapa de un proyecto rara vez se corrige antes de la fecha de entrega. A menudo es necesario negociar con el cliente un método para resolver las deficiencias.

13.5.2 Revisión de la configuración

La *revisión de la configuración* es un elemento importante del proceso de validación; su objetivo es asegurar que todos los elementos de la configuración del software se hayan desarrollado apropiadamente, estén catalogados y tengan el detalle suficiente para reforzar la fase de soporte del ciclo de vida del software. La revisión de la configuración, a veces denominada *auditoría*, se analizará con más detalle en el capítulo 27.

13.5.3 Pruebas alfa y beta

En la práctica es imposible que un desarrollador de software prevea cómo utilizará el usuario realmente el programa. Es posible que se malinterpreten las instrucciones de uso, que se utilicen con regularidad extrañas combinaciones de datos, que una salida en apariencia clara para el responsable de las pruebas sea ininteligible para un usuario en el campo.

Al construir software personalizado para un cliente se aplica una serie de pruebas de aceptación que permiten al cliente validar todos los requisitos. El usuario final conduce, no los ingenieros del software, las pruebas de aceptación, las cuales van desde una "prueba de manejo" informal hasta una serie de pruebas planeadas y ejecutadas de manera sistemática. En realidad, la prueba de aceptación llega a durar semanas o meses, por lo que es posible descubrir errores acumulativos que degradan el sistema

> "Si se recurre a ojos suficientes, todos los errores serán superficiales (por ejemplo, si hay una base lo suficientemente grande de personas que realizan las pruebas beta y de codesarrolladores, casi todos los problemas se caracterizarán rápidamente y la corrección será obvia para alguien)."
>
> **E. Raymond**

Si el software se desarrolla como un producto que usarán muchos clientes, no es práctico realizar pruebas de aceptación formales para cada uno. La mayoría de los constructores de productos de software emplean procesos llamados *prueba alfa* y *prueba beta* para descubrir errores que sólo el usuario final podría detectar.

Los usuarios finales son quienes aplican la *prueba alfa* en el lugar de trabajo del desarrollador. El software se utiliza en un entorno natural mientras el desarrollador "mira sobre el hombro" de los usuarios típicos y registra los errores y los problemas de uso. Las pruebas alfa se realizan en un entorno controlado.

Las *pruebas beta* se aplican en el lugar de trabajo de los usuarios finales. A diferencia de la prueba alfa, por lo general el desarrollador no está. Por tanto, la prueba beta es una aplicación "en vivo" del software en un entorno que no controla el desarrollador. El usuario final registra todos los problemas (reales o imaginarios) que

encuentra durante la prueba y los informa de manera regular al desarrollador. Como resultado de los problemas informados durante las pruebas beta, los ingenieros de software lo modifican y luego preparan la liberación del producto de software para toda la base de clientes

HogarSeguro

Preparación para validación

La escena: Oficina de Doug Miller, mientras continúa el diseño al nivel de componentes y empieza la construcción de ciertos componentes.

Los actores: Doug Miller, jefe de ingeniería de software, Vinod, Jamie, Ed y Shakira, integrantes del equipo de ingeniería del software *HogarSeguro*

La conversación:

Doug: El primer incremento estará listo para validación en ... ¿unas tres semanas?

Vinod: Es correcto. La integración va bien. Estamos realizando pruebas de humo a diario, encontrando algunos errores, pero nada que no se pueda manejar. Así que hasta ahora todo va bien.

Doug: Cuéntame un poco de la validación.

Shakira: Bueno, emplearemos todos los casos de uso como base para el diseño de nuestras pruebas. Aún no he empezado, pero estaré desarrollando pruebas para todos los casos de uso de los que soy responsable.

Ed: Lo mismo yo.

Jamie: Yo también, pero tendremos que actuar juntos para la prueba de aceptación y también para las pruebas alfa y beta, ¿verdad?

Doug: Sí, en realidad he pensado que podríamos [illegible] a un contratista que nos ayude con la validación. Tengo dinero en el presupuesto... y nos daría una perspectiva fresca.

Vinod: Creo que lo tenemos todo bajo control.

Doug: Estoy seguro de eso, pero un grupo independiente de prueba nos dará un punto de vista autónomo sobre el software.

Jamie: Estamos justos de tiempo aquí, Doug. Yo, en lo personal, no tengo tiempo para cuidar a nadie que traigas a hacer el trabajo.

Doug: Lo sé, lo sé. Pero si un GIP trabaja a partir de los requisitos y casos de uso, no requerirá mucha ayuda de ustedes.

Vinod: Todavía pienso que lo tenemos todo bajo co[illegible]

Doug: Ya te oí, Vinod, pero me voy a imponer en [illegible]. Planeemos el encuentro con el representante del GIP [illegible] adelante, esta misma semana. Dejemos que empiece y veamos que nos trae.

Vinod: Muy bien, tal vez aligere un poco la carga.

13.6 Prueba del sistema

En el inicio de este libro se destacó el hecho de que el software sólo es un elemento de un sistema de cómputo más grande. Al final, el software se incorpora a otros elementos del sistema (como hardware, personas, información), y se realiza una serie de pruebas de integración del sistema y de validación. Estas pruebas están más allá del alcance del proceso del software y no las realizan únicamente los ingenieros del software. Sin embargo, los pasos dados durante el diseño y la prueba del software mejorarán en gran medida la probabilidad de tener éxito en la integración del software en el sistema mayor.

> "Al igual que la muerte y los impuestos, las pruebas son desagradables e inevitables."
>
> Ed Y[illegible]

Un problema clásico de la prueba del sistema es "señalar con el dedo". Esto ocurre cuando se descubre un error y el desarrollador de cada elemento del sistema culpa a los demás. En lugar de caer en este absurdo, el ingeniero del software debe anticiparse a posibles problemas con la interfaz y 1) diseñar rutas de manejo de errores que prueben toda la información proveniente de otros elementos del sistema, 2) aplicar una serie de pruebas que simulen datos incorrectos u otros posibles errores en la interfaz del software, 3) registrar los resultados de las pruebas como "evidencia" en el caso de que se le culpe, y 4) participar en la planeación y el diseño de pruebas del sistema para asegurar que el software se ha probado adecuadamente

En realidad, la *prueba del sistema* abarca una serie de pruebas diferentes cuyo propósito principal es ejercitar profundamente el sistema de cómputo. Aunque cada prueba tiene un propósito diferente, todas trabajan para verificar que se hayan integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas. En las siguientes secciones se examinarán los tipos de pruebas del sistema [BEI84] que valen la pena para sistemas basados en software

13.6.1 Prueba de recuperación

Muchos sistemas de cómputo deben recuperarse de fallas y reanudar el procesamiento en un tiempo determinado. En algunos casos, un sistema debe ser tolerante con las fallas; es decir, las fallas de procesamiento no deben llevar a la caída del sistema, en general. En otros casos, una falla del sistema debe corregirse dentro de un periodo específico o se sufrirá un fuerte daño económico.

La *prueba de recuperación* es una prueba del sistema que obliga al software a fallar de varias maneras y a verificar que la recuperación se realice apropiadamente. Si la recuperación es automática (la realiza el propio sistema) debe evaluarse que sean correctos la reinicialización, los mecanismos de respaldo del sistema, la recuperación de datos y el nuevo arranque. Si la recuperación requiere intervención humana, se debe evaluar el tiempo medio de reparación (TMR) para determinar si se encuentra dentro de límites aceptables.

13.6.2 Prueba de seguridad

Cualquier sistema de cómputo que maneje información confidencial o que desencadene acciones que dañen (o beneficien) inapropiadamente a los individuos es un blanco para irrupciones impropias o ilegales. La irrupción abarca un amplio rango de actividades: hackers que tratan de entrar en los sistemas por juego, empleados disgustados que tratan de irrumpir como forma de venganza, e individuos deshonestos que buscan ganancias personales ilícitas.

La *prueba de seguridad* comprueba que los mecanismos de protección integrados en el sistema realmente lo protejan de irrupciones inapropiadas. Para citar a Beizer [BEI84]: "Por supuesto que debe probarse la seguridad del sistema para asegurar que es invulnerable a los ataques frontales, pero también a los perpetrados por los flancos o la retaguardia".

Durante la prueba de seguridad, quien la aplica desempeña el papel del individuo que desea entrar en el sistema. ¡Todo se vale! Debe tratar de obtener contraseñas por cualquier medio externo; podría atacar el sistema con software personalizado diseñado para burlar cualquier defensa que se haya construido; podría saturar el sistema, negando así el servicio a otros; podría producir errores intencionales en el sistema para tratar de tener acceso durante la recuperación; podría revisar datos sin protección, con la idea de encontrar la clave de acceso al sistema.

Si se dan el tiempo y los recursos suficientes, una buena prueba de seguridad terminará por irrumpir en el sistema. El papel del diseñador del sistema es que el costo de la irrupción sea mayor que el valor de la información que habrá de obtenerse.

13.6.3 Prueba de resistencia

Los pasos de prueba analizados antes, en este mismo capítulo, llevan a una evaluación completa de las funciones y el desempeño normales del programa. Las pruebas de resistencia están diseñadas para confrontar los programas con situaciones anormales. En esencia, la persona que realiza la prueba de resistencia se pregunta: "¿Hasta dónde puedo llevar esto antes de que falle?"

La *prueba de resistencia* ejecuta un sistema de tal manera que requiera una cantidad, una frecuencia o un volumen anormal de recursos. Por ejemplo: 1) se diseñan pruebas especiales que generen diez interrupciones por segundo, cuando la tasa promedio es de una o dos; 2) se aumenta la frecuencia de entrada de datos en cierta magnitud que permita determinar cómo responderán las funciones de entrada; 3) se ejecutan casos de prueba que requieran el máximo de memoria u otros recursos; 4) se diseñan casos de prueba que causen problemas de administración de memoria; 5) se crean casos de prueba que produzcan búsquedas excesivas de datos en el disco. En esencia, la persona que aplica la prueba tratará de sobrecargar el programa.

> "Si está tratando de encontrar verdaderos errores del sistema y no ha sometido su software a una verdadera prueba de resistencia, entonces éste es el momento de empezar".
>
> **Boris Beizer**

Una variante de la prueba de resistencia es una técnica denominada prueba de sensibilidad. En algunas situaciones (la más común de ellas ocurre con los algoritmos matemáticos), un rango muy pequeño de datos contenidos dentro de los límites de los datos válidos para un programa puede causar procesamiento extremadamente incluso erróneo, o una fuerte degradación del desempeño. Las pruebas de sensibilidad tratan de descubrir combinaciones de datos dentro de las clases de entrada válidas que causen inestabilidad o procesamiento inapropiado.

13.6.4 Prueba de desempeño

En sistemas en tiempo real e incrustados es inaceptable el software que proporciona la función requerida pero que no cumple los requisitos de desempeño. La prueba de desempeño está diseñada para probar el desempeño del software en tiempo de

ejecución dentro del contexto de un sistema integrado. La prueba de desempeño se aplica en todos los pasos del proceso de la prueba. Incluso al nivel de la unidad, el desempeño de un módulo individual debe evaluarse mientras se realizan las pruebas. Sin embargo, no es sino hasta que se encuentran totalmente integrados todos los elementos del sistema que es posible asegurar el verdadero desempeño del sistema.

Con frecuencia las pruebas de desempeño se vinculan con pruebas de resistencia y suelen requerir instrumentación de software y hardware. Es decir, a menudo resulta necesario medir con exactitud la utilización de recursos (por ejemplo, los ciclos de procesador). Mediante instrumentación externa pueden vigilarse de manera regular los intervalos de ejecución, los eventos que se registran (como las interrupciones) y los estados de muestra del equipo. Si se instrumenta un sistema, la persona que aplica la prueba descubrirá situaciones que lleven a la degradación y posibles fallas del sistema.

HERRAMIENTAS DE SOFTWARE

Planeación y administración de pruebas

Objetivo: Estas herramientas ayudan al [illegible] de software a planear la estrategia de prueba que [illegible]rá de elegirse y a manejar el proceso de prueba a [illegible] que se aplica.

[illegible]nica: Las herramientas de esta categoría atienden [illegible]ación y el almacenamiento de la prueba, la [illegible]istración y el control, el seguimiento de los [illegible]s, la integración, el rastreo de errores y la [illegible]ción de informes. Los jefes de proyecto las usan [illegible] complementar las herramientas de planeación [illegible]es aplican las pruebas usan estos herramientas para [illegible]r actividades de prueba y controlar el flujo de [illegible]ación a medida que avanza el proceso de prueba.

Herramientas representativas[2]

OTF (Object Testing Framework), desarrollado por MCG Software Inc. (www.mcgsoft.com), proporciona un marco conceptual para la administración de conjuntos de pruebas para objetos Smalltalk.

QADirector, desarrollado por Compuware Corp (www.compuware.com/qacenter), proporciona un solo punto de control para administrar todas las fases del proceso de prueba.

TestWorks, desarrollado por Software Reasearch Inc. (www.soft.com/Products/index, html), contiene un conjunto plenamente integrado de herramientas de prueba, incluidas algunas que sirven para el manejo y la generación de informes de las pruebas.

13.7 EL ARTE DE LA DEPURACIÓN

La prueba del software es un proceso que puede planearse y especificarse sistemáticamente. Se diseña el caso de prueba, se define una estrategia y se evalúan los resultados frente a las expectativas prescritas.

La depuración ocurre como consecuencia de una prueba realizada con éxito. Es decir, cuando un caso de prueba descubre un error, la depuración es la acción que lo elimina. Aunque la depuración puede y debe ser un proceso ordenado, sigue siendo un arte. Un ingeniero del software, al evaluar los resultados de una prueba, suele

2 Las herramientas expuestas aquí sólo representan una muestra de esta categoría. En casi todos los casos los nombres de las mismas son marcas registradas de sus respectivos desarrolladores.

enfrentarse con una indicación "sintomática" de un problema de software. Es decir tal vez la manifestación externa del error y su causa interna no tienen una relación obvia. La depuración es el proceso mental que conecta un síntoma con una causa

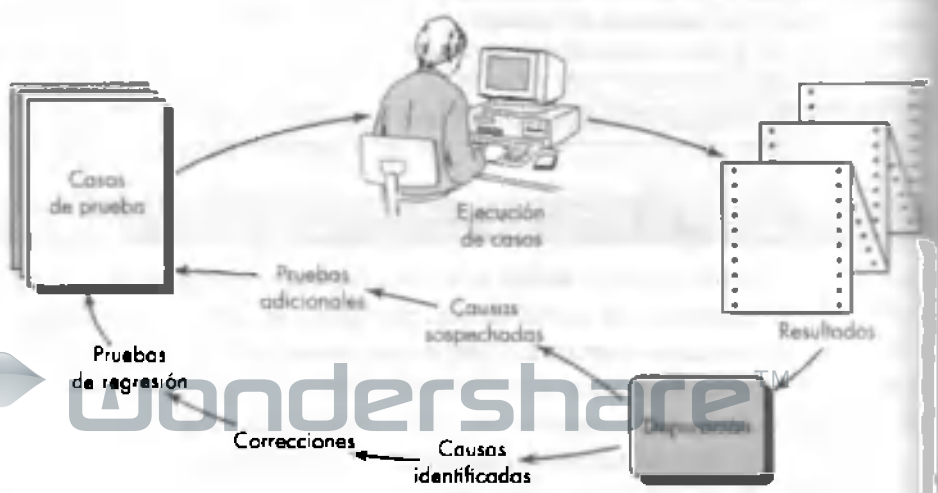
> "En cuanto empezamos la programación, descubrimos, para nuestra sorpresa, que no será fácil conseguir el programa que tenemos en mente. Es necesario descubrir la depuración. Recuerdo el momento exacto en que me di cuenta de que iba a pasar gran parte de mi vida, a partir de ese momento, en encontrar los errores de mis propios programas."
>
> **Maurice Wilkes, descubre la depuración en 1949**

13.7.1 El proceso de depuración

La depuración no es una prueba, pero siempre ocurre como consecuencia de una [illegible]. Si se toma como referencia la figura 13.7, el proceso de depuración comienza con la ejecución de un caso de prueba. Se evalúan los resultados y se encuentra una falta de correspondencia entre el desempeño esperado y el real. En muchos casos, los datos que no corresponden son síntoma de una causa que aún no aparece. La depuración trata de relacionar el síntoma con la causa, lo que conduce a corregir el error.

La depuración siempre arroja dos resultados: 1) se encuentra y se corrige la causa o 2) no se localiza la causa. En este último caso, la persona encargada de la depuración debe sospechar la causa, diseñar uno o más casos de prueba que ayuden a comprobar esa sospecha y avanzar hacia la corrección del error de manera iterativa

FIGURA 13.7
El proceso de depuración.



Al hacer esta afirmación se toma el concepto más amplio posible de la prueba. ¡No sólo el desarrollador prueba el software antes de la liberación, sino que el cliente, el usuario, o ambos, prueban software cada vez que lo usan!

¿Por qué es tan difícil la depuración? Con toda probabilidad, la respuesta se relaciona más con la psicología humana (consulte la siguiente sección) que con la tecnología del software. Sin embargo, ciertas características de los errores ofrecen algunas pistas:

¿Por qué es tan difícil la depuración?

1. El síntoma y la causa pueden estar separados geográficamente. Es decir, aquél aparece en una parte del programa mientras ésta se ubica en un sitio distante. Los componentes con un fuerte acoplamiento (capítulo 11) exacerban esta situación.
2. Es posible que el síntoma desaparezca (temporalmente) al corregir otro error.
3. Es probable que el síntoma no lo cause algún error (como en el caso de inexactitudes al redondear cifras)
4. El síntoma podría deberse a un error humano difícil de localizar
5. El síntoma podría deberse a problemas de tiempo y no de procesamiento.
6. Tal vez sea difícil reproducir con exactitud las condiciones de entrada (por ejemplo, una aplicación en tiempo real en que no está definido el orden de entrada).
7. El síntoma podría presentarse intermitentemente Esto suele ser común en sistemas empotrados que acoplan el hardware y el software de manera inextricable.
8. Probablemente el síntoma se debe a causas distribuidas entre varias tareas que se ejecutan en diferentes procesadores [CHE90]

Durante la depuración se encuentran errores que van de medianamente molestos (como un formato de salida incorrecto) a catastróficos (por ejemplo, el sistema falla y causa serios daños económicos o físicos). A medida que aumentan las consecuencias de un error, también se incrementa la presión para encontrar la causa. A menudo, debido a la presión, un desarrollador del software introduce dos errores más al tratar de corregir uno.

> "Todos saben que depurar es dos veces más difícil que escribir el programa. Por tanto, si aplica toda su inteligencia para escribirlo, ¿cómo espera siquiera depurarlo?"
>
> —Brian Kernighan

13.7.2 Consideraciones psicológicas

Por desgracia, hay evidencia de que las destrezas para la depuración son innatas en el ser humano. Ciertas personas son buenas para ella; otras no Aunque la evidencia experimental sobre la depuración está abierta a muchas interpretaciones, se han reportado grandes variaciones en la habilidad para la depuración en programadores con educación y experiencia similares.

Al comentar los aspectos humanos de la depuración, Shneiderman [SHN80] afirma:

> La depuración es una de las partes más frustrantes de la programación. Incluye elementos de resolución de problemas o de retos mentales, junto con el molesto reconocimiento de que se ha cometido un error. La creciente ansiedad y escasa voluntad de aceptar la existencia de errores aumentan la dificultad de la tarea. Por fortuna, se presenta un gran alivio y la tensión decrece cuando el error finalmente... se corrige.

Aunque sea difícil "aprender" a depurar, se proponen varios enfoques para el problema. Se examinarán en la siguiente sección.

HOGARSEGURO

Depuración

La escena: Cubículo de Ed mientras se realizan la codificación y la prueba de unidad.

Los actores: Ed y Shakira, integrantes del equipo de ingeniería del software de *HogarSeguro*.

La conversación:

Shakira (asomándose a la entrada del cubículo): Hey .. ¿dónde estabas a la hora del almuerzo?

Ed: Justo aquí... trabajando.

Shakira: Te ves mal... ¿qué es lo que pasa?

Ed (suspirando con fuerza): He estado trabajando en este <bleep> error porque lo descubrí a las 9:30 de la mañana, y son las 2:45 y aún no tengo una pista.

Shakira: Pensé que estábamos de acuerdo en no dedicar más de una hora a depurar cosas por nuestra cuenta. En ese caso, tendríamos que buscar ayuda, ¿o no?

Ed: Sí, pero.

Shakira (entrando en el cubículo): A ver, ¿cuál es el problema?

Ed: Es complicado. Y además he estado revisando esto durante, ¿cuánto?, cinco horas. No vas a encontrarlo.

Shakira: Perdóname... ¿cuál es el problema?

(Ed le explica el problema a Shakira, que lo mira durante 30 segundos sin hablar.)

Shakira (empieza a pintarse una sonrisa en su rostro): Mira, justo aquí, la variable *establecerCondicionAlarma*. ¿No debería ponerse en "falsa" antes de que inicie el bucle?

(Ed se queda viendo la pantalla sin creerlo, se inclina hacia delante y empieza a golpear su cabeza gentilmente contra el monitor. Shakira, ahora sonriendo ampliamente, se levanta y sale.)

13.7.3 Estrategias de depuración

Sin importar el enfoque que se adopte, la depuración tiene un objetivo primordial: encontrar y corregir la causa de un error del software. El objetivo se logra combinando la evaluación sistemática, la intuición y la suerte. Bradley [BRA85] describe así el enfoque de la depuración:

> La depuración es una aplicación simple y directa del método científico desarrollado hace 2 500 años. La esencia de la depuración consiste en ubicar la fuente del problema [la causa] mediante partición binaria, manejando hipótesis de trabajo que predigan nuevos valores que habrán de examinarse.

Tomemos un ejemplo sencillo, sin relación alguna con el software: en mi casa no funciona una lámpara. Si no funciona nada en la casa, la causa debe ser un fusible fundido o una falla en el suministro de energía eléctrica. Miro alrededor para ver si hay luz en el ve-

cindario. Conecto la lámpara bajo sospecha en un enchufe que funcione y un aparato en buen estado en el enchufe bajo sospecha. Y así se siguen alternando hipótesis y pruebas.

En general, se han propuesto tres estrategias de depuración [MYE79]: 1) fuerza bruta, 2) seguimiento hacia atrás y 3) eliminación de la causa.

> "El primer paso para corregir un programa es hacer que falle repetidamente (en el ejemplo más simple posible)."
>
> T. Dwff

Tácticas de depuración. La categoría de depuración por la *fuerza bruta* tal vez sea el método más común y menos eficiente para aislar la causa de un error del software. Los métodos de depuración por la fuerza bruta se aplican cuando todo lo demás falla. Al aplicar una filosofía de "dejemos que la computadora encuentre el error", se hacen descargas de memoria, se invocan señales en tiempo de ejecución y se carga el programa con instrucciones de salida. En algún lugar del pantano de información que se produce se espera encontrar una pista que pueda conducir a la causa de un error. Aunque la gran cantidad de información producida conduzca finalmente al éxito, lo más frecuente es que haga desperdiciar tiempo y esfuerzo.

El *rastreo hacia atrás* es un enfoque de depuración muy común, que se utiliza con éxito en pequeños programas. Empezando en el sitio donde se ha descubierto un síntoma, se recorre hacia atrás el código fuente (manualmente) hasta hallar el sitio de la causa. Por desgracia, a medida que aumenta el número de líneas del código, la cantidad de caminos hacia atrás se vuelve tan grande que resulta inmanejable.

El tercer enfoque para la depuración (*eliminación de causas*) lo determina la inducción o deducción e introduce el concepto de *partición binaria*. Los datos relacionados con el error se organizan para aislar las causas posibles. Se elabora una "hipótesis de la causa" y se aprovechan los datos ya mencionados para probar o desechar la hipótesis. Como opción, se elabora una lista de todas las causas posibles y se aplican pruebas para eliminar cada una de ellas. Si las pruebas iniciales indican que determinada hipótesis de causa es prometedora, se refinan los datos para tratar de aislar el error.

Depuración automatizada. Cada uno de los enfoques de depuración anteriores complementan las herramientas de depuración que proporcionan soporte semiautomatizado al ingeniero de software mientras se intentan estrategias de depuración. Hailpern y Santhanam [HAI02] resumen el estado de estas herramientas cuando indican: "...se han propuesto muchos nuevos enfoques y se dispone de muchos entornos de depuración comerciales. Los entornos de desarrollo integrado (EDI) proporcionan una manera de capturar algunos de los errores por defecto específicos del lenguaje (por ejemplo, caracteres faltantes de fin-de-instrucción, variables indefinidas, etc.) sin requerir compilación." Un área que ha atrapado la imaginación de la industria es la visualización de las construcciones de programación necesarias como medio de análisis de programas [BAE97]. Se cuenta con una amplia variedad de compiladores de depuración, ayudas dinámicas para la depuración ("trazadores"), generadores automáticos de casos de prueba y herramientas de correlación de refe-

HERRAMIENTAS DE SOFTWARE

Depuración

Objetivo: Estas herramientas proporcionan ayuda automatizada a quienes deben depurar problemas de software. El objetivo es proporcionar conocimiento difícil de obtener si se afronta el proceso de depuración manualmente.

Mecánica: Casi todas las herramientas de depuración son específicas del lenguaje de programación y del entorno.

Herramientas representativas:[4]

Jprobe ThreadAnalyzer, desarrollado por Sitraka (www.sitraka.com), ayuda en la evaluación de problemas de subprocesos: bloqueos, detenciones y condiciones de carrera que representan serios peligros para el desempeño en aplicaciones de Java.

C++ Test, desarrollado por Parasoft (www.parasoft.com), es una herramienta de prueba de unidad que soporta un amplio rango de pruebas en código C y C++. Las características de depuración ayudan al diagnóstico de los errores encontrados.

CodeMedic, desarrollado por NewPlanet Software (www.newplanetsoftware.com/medic/), proporciona una interfaz gráfica para el depurador UNIX estándar, gdb, e implementa sus características más importantes. Actualmente gdb da soporte a C/C++, Java, PalmOS, varios sistemas incrustados, lenguaje ensamblador, FORTRAN y Modula-2.

BugCollector Pro, desarrollado por Nesbitt Software Corp. (www.nesbitt.com/), implementa una base de datos multiusuario que ayuda al equipo de software registrar errores reportados y otras solicitudes de mantenimiento, y administración de flujo de trabajo de depuración.

GNATS, una aplicación freeware (www.gnu.org/software/gnats/), es un conjunto de herramientas para registrar informes de error.

rencias cruzadas. Sin embargo, las herramientas no son un sustituto de la evaluación cuidadosa basada en un modelo de diseño completo y un código fuente claro.

El factor humano. Ningún análisis de los enfoques y las herramientas de depuración estaría completo sin mencionar un poderoso aliado: ¡los demás! Un punto de vista fresco, despejado de horas de frustración, puede hacer maravillas.[5] Una máxima final para la depuración sería: "¡Cuando todo lo demás falle, pida ayuda!"

13.7.4 Corrección del error

Cuando se encuentra un error debe corregirse. Pero como ya se ha indicado, corregir un error pueden introducirse otros y, por lo tanto, causar más daño que solucionar el problema. Van Vleck [VAN89] sugiere tres preguntas simples que debería plantearse todo ingeniero del software antes de hacer la "corrección" que elimine la causa del error:

¿Cuando corrijo un error, ¿qué preguntas debo hacerme?

1. *¿La causa del error se repite en otra parte del programa?* En muchas situaciones un error se produce en un programa debido a un patrón erróneo de lógica que podría repetirse en cualquier lugar. La consideración explícita del patrón lógico puede llevar al descubrimiento de otros errores.

[4] Las herramientas expuestas aquí representan una muestra de esta categoría. En casi todos los casos los nombres de las mismas son marcas registradas de sus respectivos desarrolladores.

[5] El concepto de programación por pares (recomendada como parte del modelo de proceso de *programación extrema* analizado en el capítulo 4) proporciona un mecanismo para la depuración a medida que se diseña y codifica el software.

2. *¿Cuál es el "siguiente error" que podría introducirse con la corrección que está a punto de realizarse?* Antes de la corrección se debe evaluar el código fuente (o mejor aún, el diseño) para evaluar el acoplamiento entre las estructuras lógicas y de datos. Si la corrección se realiza en una sección del programa con un acoplamiento elevado, debe tenerse mucho cuidado cuando se haga cualquier cambio.

3. *¿Qué debió hacerse para evitar este error desde el principio?* Esta pregunta es el primer paso hacia el establecimiento de un enfoque estadístico de aseguramiento de la calidad del software (capítulo 26). Si se corrige el proceso junto con el producto, se eliminará el error del programa actual y de todos los programas futuros.

13.8 RESUMEN

La prueba ocupa el mayor porcentaje del esfuerzo técnico en el proceso del software. Apenas se empiezan a comprender las sutilezas de la planeación, la ejecución y el control sistemáticos de las pruebas.

El objetivo de la prueba del software es descubrir errores; se cumple planeando y ejecutando una serie de pasos (pruebas de unidad, integración, validación y sistema). Las pruebas de unidad e integración se concentran en la verificación funcional de cada componente y en la incorporación de componentes en la arquitectura del software. La prueba de validación demuestra el cumplimiento con los requisitos del software, y la prueba del sistema valida el software una vez que se ha incorporado a un sistema mayor.

Cada paso de prueba se completa mediante una serie de técnicas sistemáticas de prueba que ayudan a diseñar los casos de prueba. Cada paso de prueba ensancha el grado de abstracción con que se considera el software.

A diferencia de la prueba (una actividad sistemática y planeada), la depuración debe considerarse un arte. La actividad de depuración empieza con la indicación sintomática de un problema y debe rastrear la causa del error. Entre los muchos recursos disponibles durante la depuración, el más valioso es el consejo de otros integrantes del equipo de ingeniería del software.

La necesidad de crear software de la mayor calidad exige un enfoque de prueba más sistemático. Para citar a Dunn y Ullman [DUN82]:

> Lo indispensable es una estrategia general que abarque el espacio de prueba estratégico con una metodología tan deliberada como lo era el desarrollo sistemático en que se basaban el análisis, el diseño y la codificación.

En este capítulo se ha examinado el espacio de prueba estratégico, tomando en cuenta los pasos que tienen la mayor probabilidad de conseguir el principal objetivo de la prueba: encontrar y eliminar errores de manera ordenada y efectiva.

REFERENCIAS

[BAE97] Baecker, R., C. DiGiano y A. Marcus, "Software Visualization of Debugging", *Communications of the ACM*, vol. 40, núm. 4, abril de 1997 y otros ensayos en la misma e
[BEI84] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand-Reinhold
[BER93] Berard, E., *Essays on Object-Oriented Software Engineering*, vol. 1, Addison-Wesley
[BIN94] Binder, R., "Testing Object-Oriented Systems: A Status Report", en *American Pro*, vol. 7, núm. 4, abril de 1994, ruta crítica, pp. 23-28.
[BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981, p. 37.
[BRA85] Bradley, J.H., "The science and Art of Debugging", en *Computerworld*, 19 de ag, 1985, pp. 35-38.
[CHE90] Cheung, W. H., J. P. Black y E. Manning, "A Framework for Distributed Debuggi, *IEEE Software*, enero de 1990, pp. 106-115.
[DUN82] Dunn, R. y R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982.
[GIL95] Gilb, T., "What We Fail To Do In Our Current en Testing Culture", en *Testing Newsletter* (edición en linea, tln@soft.com), Software Research, Inc., enero de
[HAI02] Hailpern, B. y P. Santhanam, "Software Debugging, Testing and Verification", en Systems Journal, vol. 41, núm. 1, 2002, disponible en http://www.research.ibm.com/journal/sj/411/hailpern.html
[IEE01] *Software Reliability Engineering, 12th International Symposium*, IEEE, 2001
[MCO96] McConnell, S., "Best Practices: Daily Build and Smoke Test", en *IEEE Software*, núm. 4, julio de 1996, pp. 143-144.
[MIL77] Miller, E., "The Philosophy of Testing", en *Program Testing Techniques*, IEEE Co, Society Press, 1977, pp. 1-3.
[MUS89] Musa, J. D. y A. F. Ackerman, "Quantifying Software Validation: When to Stop Te, en *IEEE Software*, mayo de 1989, pp. 19-27.
[MYE79] Myers, G., *The Art of Software Testing*, Wiley, 1979.
[SHO83] Shooman, M. L., *Software Engineering*, McGraw- Hill, 1983.
[SHN80] Shneiderman, B., *Software Psychology*, Winthrop Publishers. 1980, p. 28.
[SIN99] Singpurwalla, N. y S. Wilson, *Statistical Methods in Software Engineering: Reliab, Risk*, Springer-Verlag, 1999.
[VAN89] Van Vleck, T., "Three Questions About Each Bug You find", en *ACM Software, Notes*, vol. 14, núm. 5, julio de 1989, pp. 62-63.
[WAL89] Wallance, D. R. y R. U. Fujii, "Software Verification and Validation: An Overview", *IEEE Software*, mayo de 1989, pp. 10-17.
[YOU75] Yourdon, E., *Techniques of Program Structure and Design*, Prentice-Hall, 1975

PROBLEMAS Y PUNTOS A CONSIDERAR

13.1. Con palabras propias, describase la diferencia entre verificación y validación. ¿L métodos de diseño de casos de prueba y estrategias de prueba?

13.2. Elabórese una lista de algunos problemas que pudieran estar asociados con la de un grupo independiente de prueba. ¿Lo integran las mismas personas que el gr aseguramiento de la calidad del software?

13.3. ¿Siempre es posible desarrollar una estrategia para probar software que secuencia de pasos de prueba descrita en la sección 13.1.3? ¿Cuáles son las complicaciones que podrían surgir para sistemas incrustados?

13.4. ¿Por qué es difícil aplicar pruebas de unidad a un módulo altamente acc

13.5. El concepto de "antierror" (sección 13.3.1) es una manera extremadamente efec proporcionar depuración integrada cuando se descubre un error.

a) Desarrollar un conjunto de directrices antierror.
b) Analizar las ventajas de usar esta técnica.
c) Analizar las desventajas de usar esta técnica.

13.6. ¿Cómo afecta la calendarización la prueba de integración?

13.7. ¿La prueba de unidad es posible (o incluso deseable) en todas las circunstancias? Proporcionar ejemplos que justifiquen la respuesta

13.8. ¿Quién debe aplicar la prueba de validación· el desarrollador o el usuario del software? Justifíquese la respuesta.

13.9. Desarrollar una estrategia de prueba completa para el sistema *HogarSeguro* analizado en todo el libro. Documéntese en una *Especificación de prueba*

13.10. Como proyecto de clase, desarrollar una *Guía de depuración* para instalarla. Deben proporcionarse consejos orientados al lenguaje y al sistema ¡qué se hayan aprendido en la escuela de la vida! Empezar con una descripción esquemática de los temas que revisarán los compañeros de clase y el profesor

Otras lecturas y fuentes de información

Casi todos los libros sobre la prueba del software analizan estrategias junto con métodos para el diseño de casos de prueba. Todos los siguientes libros analizan los principios, los conceptos, las estrategias y los métodos de prueba: Craig y Kaskiel (*Systematic Software Testing*, Artech House, 2002), Tamres (*Introducing Software Testing*, Addison-Wesley, 2002), Whittaker (*How To Break Software*, Addison-Wesley, 2002), Jorgensen (*Software Testing: A Craftman's Approach*, CRC Press, 2002), Splaine y sus colegas (*The Web Testing Handbook*, Software Quality Engineering Publishing, 2001), Patton (*Software Testing*, Sams Publishing, 2000), Kaner y sus colegas (*Testing Computer Software*, segunda edición, Wiley, 1999), Black (*Managing the Testing Process*, Microsoft Press, 1999) y Perry (*Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach*, Dorset House, 1997) también atienden las estrategias de prueba del software

Para los lectores interesados en métodos de desarrollo ágil de software, Crispin y House (*Testing Extreme Programming*, Addison-Wesley, 2002) y Beck (*Test Driven Development: By Example*, Addison-Wesley, 2002) presentan estrategias y tácticas de prueba para Programación Extrema. Kamer y sus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) presentan una colección de más de 300 "lecciones" pragmáticas (directrices) que toda persona dedicada a la prueba de software debe aprender. Watkins (*Testing IT: An Off the Shelf Testing Process*, Cambridge University Press, 2001) establece un marco conceptual de prueba efectivo para todos los tipos de software desarrollado y adquirido.

Lewis (*Software Testing and Continuous Quality Improvment*, CRC Press, 2000) y Koomen y sus colegas (*Test Process Improvment*, Addison-Wesley, 1999) analizan estrategias para la mejora continua del proceso de prueba.

Sykes y McGregor (*Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001), Bashir y Goel (*Testing Object-Oriented Software*, Springer-Verlag, 2000), Binder, *Testing Object Oriented Systems*, Addison-Wesley, 1999), Kung y sus colegas (*Testing Object-Oriented Software*, IEE Computer Society Press, 1998) y Marick (*The Craft of Software Testing*, Prentice Hall, 1997) presentan estrategias y métodos para prueba de sistemas orientados a objetos

Directrices para la depuración se encuentran en libros de Agans (*Debugging: The Nine Indispensable Rules for Finding Even The Most Elusive Hardware and Software Problems*, AMACON, 2002), Tells y Hsieh (*The Science of Debugging*, The Coreolis Group, 2001), Robbins (*Debugging Applications*, Microsoft Press, 2000) y Dunn (*Software Defect Removal*, McGraw-Hill, 1984). Rosenberg (*How Debuggers Work*, Wiley, 1996) atiende la tecnología de las herramientas de depuración. Younessi (*Object-Oriented Defect Management of Software*, Prentice-Hall, 2002) presenta técnicas para administrar los defectos que se encuentran en sistemas orientados a objetos Beizer [BEI84] presenta una interesante "taxonomía de los errores" que puede llevar a métodos efectivos para la planeación de pruebas. Ball (*Debugging Embedded Microprocessor Systems*, Newnes Publishing, 1998) atiende la naturaleza especial del software incrustado de microprocesador.

En Internet se encuentra una amplia variedad de fuentes de información sobre estrategias de prueba del software. Una lista actualizada de referencias en la World Wide Web que resultan relevantes para las estrategias de prueba del software se encuentran en el sitio Web SEPA. **http://www.mhhe.com/pressman.**

CAPÍTULO

14 TÉCNICAS DE PRUEBA DEL SOFTWARE

CONCEPTOS CLAVE

Las pruebas representan un interesante reto para los ingenieros de software, quienes por naturaleza son personas constructivas. Las pruebas requieren que el desarrollador descarte nociones preconcebidas de lo "correcto" en el software y entonces diseñe difíciles casos de prueba para "romperlo". Beizer [BEI90] describe bien esta situación cuando afirma:

> Es un mito que si realmente fuéramos buenos para programar no tendríamos que depurar errores. Si tan sólo pudiéramos concentrarnos, si todos usaran programación estructurada, diseño descendente o tablas de decisión, si los programas se escribieran en SQUISH, si tuviéramos las balas plateadas correctas, entonces no habría errores. Ése es el mito. Hay errores, dice el mito, porque somos malos en lo que hacemos y si somos malos en eso, debemos sentirnos culpables. Por tanto, el diseño de pruebas y de casos de prueba es una admisión de la falla, que instila una buena dosis de culpa. Y el tedio de probar sólo es un castigo por nuestros errores. ¿Castigo por qué? ¿Por ser seres humanos? ¿Culpables de qué? ¿De no alcanzar una perfección inhumana? ¿Por no distinguir entre lo que otro programador piensa y lo que dice? ¿Por no usar la telepatía? ¿Por no resolver los problemas de las comunicaciones humanas que han estado presentes... durante cuarenta siglos?

¿Las pruebas deben provocar culpa? ¿Las pruebas son realmente destructivas? La respuesta es ¡no!

En este capítulo se analizarán técnicas para el diseño de casos de prueba del software. Este tipo de diseño se concentra en un conjunto de técnicas para la creación de casos de prueba que cumplan con objetivos generales y con las estrategias de prueba analizadas en el capítulo 13.

UN VISTAZO RÁPIDO

¿Qué es? Una vez generado el código fuente, es necesario probar el software para descubrir (y corregir) la mayor cantidad de errores posible antes de entregarlo al cliente. Su objetivo es diseñar una serie de casos de prueba que tengan una alta probabilidad de encontrar errores, ¿pero cómo? Aquí es donde entran en escena las técnicas de prueba del software. Estas técnicas proporcionan directrices sistemáticas para probar las de diseño que 1) comprueben la lógica interna y las interfases de todo componente del software y 2) comprueben los dominios de entrada y salida del programa para descubrir errores en la función, comportamiento y desempeño.

¿Quién lo hace? Durante las etapas iniciales de este proceso, un ingeniero de software realiza todas las pruebas. Sin embargo, a medida que avanza este proceso se irán incorporando especialistas en pruebas.

¿Por qué es importante? Con las revisiones y otras actividades de aseguramiento de la calidad del software se pueden y deben descubrir errores, pero no basta con ello. El cliente prueba el programa cada vez que lo ejecuta. Por tanto, se tiene que ejecutar el programa antes de que llegue al cliente, y el objetivo específico será encontrar y eliminar todos los errores. La localización de la mayor cantidad de errores requiere aplicar pruebas de manera sistemática y diseñar casos de prueba empleando técnicas definidas.

¿Cuáles son los pasos? En aplicaciones convencionales el software se prueba desde dos perspectivas diferentes: 1) la lógica interna del programa se comprueba mediante técnicas de diseño de casos de prueba de "caja blanca", 2) los requisitos del software se comprueban empleando técnicas de diseño de casos de prueba de "caja negra". En el caso de aplicaciones orientadas a objetos, la "prueba" empieza antes de la existencia del código fuente, para una vez generado éste, se diseñará una serie de pruebas para comprobar operaciones con una clase y examinar si existen errores mientras una clase colabora con otra. A medida que las clases se integran para formar un subsistema, se aplica la prueba de uso, junto con los enfoques basados en fallas, para comprobar las clases que colaboran. Por último, los casos de uso ayudan a diseñar pruebas que permitan descubrir errores al nivel de validación del software. En todo caso, el objetivo es encontrar el número máximo de errores con la mínima cantidad de esfuerzo y tiempo.

¿Cuál es el producto obtenido? Se diseña y documenta un conjunto de casos de prueba diseñado para comprobar la lógica interna, las interfaces, las colaboraciones entre componentes y los requisitos internos; se definen los resultados esperados y se registran los resultados reales.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Cuando se empiece la prueba debe cambiarse de punto de vista. ¡El objetivo es "romper" el software! Deben diseñarse casos de prueba en forma meticulosa y revisarse que los casos de prueba creados abarquen todo lo diseñado. Además, es preciso evaluar la cobertura de la prueba y darle seguimiento a las actividades de detección de errores.

14.1 FUNDAMENTOS DE LAS PRUEBAS DEL SOFTWARE

En el capítulo 5 se analizaron los objetivos y principios fundamentales de las pruebas. Se recordará que el objetivo de las pruebas es encontrar errores y que una buena prueba es la que tiene una alta probabilidad de encontrar un error. Por tanto, cuando un ingeniero de software diseñe e implemente un sistema o un producto de cómputo, debe tener en mente la facilidad de prueba. Al mismo tiempo, las propias pruebas deben mostrar un conjunto de características para alcanzar el objetivo de encontrar la mayor cantidad de errores con un mínimo de esfuerzo.

> "Todo programa hace algo bien; pero tal vez sea lo que no queremos que haga."
>
> **Anónimo**

Facilidad de prueba. James Bach[1] proporciona la siguiente definición: "La facilidad de *prueba del software* indica simplemente si es fácil o no probar [un programa de computadora]." Las siguientes características propician la creación de software que tenga facilidad de prueba.

1 Los párrafos siguientes se usan con permiso de James Bach, (copyright, 1994) y se han adaptado del material que originalmente apareció publicado en el grupo de noticias comp.software-eng.

¿Cuáles son las características de la facilidad de prueba?

Operatividad. "Cuanto mejor funcione, con mayor eficiencia podrá probarse un sistema está diseñado e implementado con la calidad en mente, serán relativamente escasos los errores que bloquearán la ejecución de las pruebas, lo que permitirá el avance de éstas sin correcciones ni reinicios.

Observabilidad. "Lo que se ve es lo que se prueba." Las entradas proporcionadas como parte de la prueba producen salidas distintas. Los estados y las variables del sistema son visibles y pueden consultarse durante la ejecución. La salida incorrecta se identifica fácilmente. Los errores internos se detectan y reportan en forma automática El código fuente es accesible.

Controlabilidad. "Cuanto mejor se controle el software, mejor se automatizará y mejorarán las pruebas " El ingeniero de pruebas controla directamente los estados las variables de software y hardware Las pruebas pueden ser convenientemente especificadas, automatizadas y reproducidas.

Capacidad para descomponer. Al controlar el alcance de la prueba, se aislarán los problemas más rápidamente y se aplicarán las pruebas nuevamente con mayor inteligencia. El sistema de software se construye a partir de módulos independientes que también se prueban independientemente.

Simplicidad. "Cuanto menos haya que probar, más rápido se hará." El programa debe mostrar *simplicidad funcional* (por ejemplo, el conjunto de características es el mínimo necesario para satisfacer los requisitos), *simplicidad estructural* (la arquitectura aparece en módulos para limitar la propagación de fallas) y *simplicidad de código* (se adapta un estándar de codificación para facilitar la inspección y el mantenimiento.)

Estabilidad. "Cuantos menos cambios haya, menores alteraciones habrá en la prueba." Los cambios al software son poco frecuentes, se controlan cuando ocurren y no invalidan las pruebas existentes. El software se recupera bien de las fallas

Facilidad de comprensión. "Cuanta mayor información se tenga, con mayor inteligencia se aplicará la prueba." Se comprenden bien el diseño de la arquitectura las dependencias entre componentes internos, externos y compartidos. Se tiene acceso instantáneo a la documentación técnica, está bien organizada, es específica, detallada y exacta. Los cambios al diseño se comunican a quienes aplican las pruebas.

Un ingeniero usará los atributos que sugiere Bach para desarrollar una configuración de software (es decir, programas, datos y documentos) que sea sensible a la prueba.

> "Los errores son más comunes en el software, tienen más capacidad de expandirse y representan más problemas que en otras tecnologías."
>
> **David Parnas**

Características de la prueba. ¿Y qué hay con las propias pruebas? Kaner, Falk y Nguyen [KAN93] sugieren los siguientes atributos para una buena prueba:

¿Qué es una "buena" prueba?

1. *Una buena prueba tiene una elevada probabilidad de encontrar un error.* Alcanzar este objetivo requiere que la persona que aplica la prueba comprenda el software y trate de desarrollar una imagen mental de la manera en que puede fallar. Lo ideal es probar los tipos de fallas. Por ejemplo, un tipo de falla posible en una interfaz gráfica de usuario es la incapacidad de reconocer la posición correcta del ratón; por tanto, se diseñaría un conjunto de pruebas para probarlo tratando de evidenciar un error en el reconocimiento de su posición.
2. *Una buena prueba no es redundante.* El tiempo y los recursos destinados a las pruebas son limitados. No hay razón para realizar una prueba que tenga el mismo propósito que otra. Cada prueba debe tener un propósito diferente (aunque las diferencias sean sutiles).
3. *Una buena prueba debe ser "la mejor de su clase"* [KAN93] En un grupo de pruebas con un objetivo similar y recursos limitados podría optarse por la ejecución de un solo subconjunto de ellas. En este caso, debe usarse la prueba que tenga la mayor probabilidad de descubrir un tipo completo de errores.
4. *Una buena prueba no debe ser ni muy simple ni demasiado compleja.* Aunque a veces es posible combinar una serie de pruebas en un caso de prueba, los posible efectos colaterales asociados con este enfoque podrían enmascarar errores. En general, cada prueba debe ejecutarse por separado.

HOGARSEGURO

Diseño de pruebas únicas

La escena: Cubículo de Vinod.

Los actores: Vinod y Ed, integrantes del equipo de ingeniería del software *HogarSeguro*.

La conversación:

Vinod: Así que éstos son los casos de prueba que preparé para ejecutar con la operación *validaciónContraseña*.

Ed: Sí, cubren muchos de los posibles tipos de contraseñas que podría ingresar un usuario.

Vinod: Déjame ver... señalas que la contraseña correcta será 8080, ¿verdad?

Ed: Ajá.

Vinod: ¿Y especificas las contraseñas 1234 y 6789 para encontrar errores en el reconocimiento de contraseñas no válidas?

Ed: Correcto, y también pruebo contraseñas que son parecidas a la correcta, como 8081 y 8180.

Vinod: Está bien, pero no veo mucho caso en ejecutar las entradas 1234 y 6789. Son redundantes... prueban lo mismo, ¿o no?

Ed: Bueno, son valores diferentes.

Vinod: Es cierto, pero si 1234 no descubre un error... en otras palabras ... la operación *validaciónContraseña* detecta que la contraseña no es válida, así que no es probable que 6789 nos muestre algo nuevo.

Ed: Ya sé lo que quieres decir.

Vinod: No estoy tratando de ser puntilloso... sólo que tenemos tiempo limitado para las pruebas, de modo que es buena idea ejecutar pruebas que tengan una alta probabilidad de encontrar nuevos errores.

Ed: No hay problema . Pensaré un poco más en esto.

14.2 PRUEBAS DE CAJA NEGRA Y CAJA BLANCA

Hay una de dos maneras de probar cualquier producto construido (y casi cualquier cosa): 1) si se conoce la función específica para la que se diseñó el producto, se pueden pruebas, que demuestren que cada función es plenamente operacional, mientras se buscan los errores de cada función; 2) si se conoce el funcionamiento interno producto, se aplican pruebas para asegurarse de que "todas las piezas encajan", decir, que las operaciones internas se realizan de acuerdo con las especificaciones y que se han probado todos los componentes internos de manera adecuada. Al primer enfoque de prueba se le denomina prueba de caja negra; al segundo, prueba caja blanca.[2]

Las pruebas de caja negra son las que se aplican a la interfaz del software prueba de este tipo examina algún aspecto funcional de un sistema que tiene relación con la estructura lógica interna del software. La prueba de caja blanca software se basa en un examen cercano al detalle procedimental. Se prueban las tas lógicas del software y la colaboración entre componentes, al proporcionar de prueba que ejerciten conjuntos específicos de condiciones, bucles o ambos

¡PUNTO CLAVE

Las pruebas de caja blanco sólo pueden diseñarse después del diseño al nivel de componentes (o código fuente). Es necesario que los detalles lógicos del programa estén disponibles

> "Sólo hay una regla para diseñar casos de prueba: abarcar todas las funciones, pero no diseñar demasiados casos."
> Tsuneo Yamaura

A primera vista, parecería que toda prueba de caja blanca completa llevaría programa 100 por ciento correcto. Todo lo que se necesita hacer es identificar los caminos lógicos, desarrollar casos de prueba para ejercitarlos y evaluar dos; es decir, generar casos de prueba para comprobar exhaustivamente la lógica programa. Por desgracia, la prueba exhaustiva presenta ciertos problemas de tica (consúltese el análisis del recuadro). Sin embargo, la prueba de caja blanca

INFORMACIÓN

Prueba exhaustiva

Considérese un programa de cien líneas en lenguaje C. Después de alguna declaración básica de datos, el programa contiene dos bucles anidados que se ejecutan de 1 a 20 veces cada uno, lo que depende de la condición especificada en la entrada. Dentro del bucle interno se requieren cuatro construcciones si-entonces-si_no (*if-then-else*). ¡El programa tendrá alrededor de 10^{14} posibles rutas de ejecución!

Poner este número en perspectiva requiere suponer que se ha desarrollado un procesador de prueba mágico ("mágico" porque no existe) para aplicar una prueba va. El procesador desarrolla un caso de prueba, lo y evalúa los resultados en un milisegundo. Si trabaja horas diarias, 365 días al año, necesitaría 3 170 ra probar el programa. Esto causaría, indudablemente desastre en casi todos los calendarios de desarrollo.

Por tanto, es razonable asegurar que resulta aplicar una prueba exhaustiva en sistemas grandes

2 Los términos *prueba funcional* y *prueba estructurada* suelen usarse en lugar de prueba de y de caja blanca, respectivamente.

debe desecharse nunca como **impráctica**. Es posible seleccionar y comprobar un número limitado de rutas lógicas importantes; además de probar la validez de las principales estructuras de datos.

14.3 PRUEBAS DE CAJA BLANCA

La prueba de caja blanca, en ocasiones llamada prueba de caja de cristal, es un método de diseño que usa la estructura de control descrita como parte del diseño al nivel de componentes para derivar los casos de prueba. Al emplear los métodos de prueba de caja blanca, el ingeniero del software podrá derivar casos de prueba que 1) garanticen que todos las rutas independientes dentro del módulo se han ejercitado por lo menos una vez, 2) ejerciten los lados verdadero y falso de todas las decisiones lógicas, 3) ejecuten todos los bucles en sus límites y dentro de sus límites operacionales, y 4) ejerciten estructuras de datos internos para asegurar su validez.

> "Los errores pululan en los rincones y se acumulan en los límites."
>
> **Boris Beizer**

14.4 PRUEBA DE LA RUTA BÁSICA

La *prueba de la ruta básica* es una técnica de prueba de caja blanca que propuso inicialmente Tom McCabe [MCC76]. El método de la ruta básica permite que el diseñador de casos de prueba obtenga una medida de complejidad lógica de un diseño procedimental y que use esta medida como guía para definir un conjunto básico de rutas de ejecución. Los casos de prueba derivados para ejercitar el conjunto básico deben garantizar que se ejecuta cada instrucción del programa por lo menos una vez durante la prueba.

14.4.1 Notación de gráfica de flujo

CONSEJO

...eba dibujarse ...uma de ...ando lo ...lógica de ...nente sea ... El ... de flujo ...trazar ... del programa de ... más legible.

Antes de tratar el método de la ruta básica, debe presentarse una notación simple para la representación del flujo de control, llamado *gráfica de flujo* (o *gráfica del programa*).[3] La gráfica de flujo describe un flujo de control lógico empleando la notación ilustrada en la figura 14.1. Cada construcción estructurada (capítulo 11) tiene su símbolo correspondiente en la gráfica de flujo.

El uso de una gráfica de flujo se ilustra considerando la representación del diseño procedimental de la figura 14.2a. Aquí se describe la estructura de control del programa mediante un diagrama de flujo. En la figura 14.2b se correlaciona (o mapea) el diagrama de flujo con su gráfica de flujo correspondiente (suponiendo que no existen condiciones compuestas en los diamantes de decisión del diagrama de flujo). To-

3 En realidad, el método de la ruta básica se aplica sin el uso de las gráficas de flujo. Sin embargo, sirven como notación útil para comprender el flujo de control e ilustrar el enfoque

mando como referencia la figura 14.2b, cada círculo, llamado *nodo de gráfica de flujo,* representa una o más instrucciones procedimentales. Una secuencia de recuadros de proceso y un diamante de decisión se correlaciona con un solo nodo. Las flechas en la gráfica de flujo, llamadas *aristas* o *enlaces,* representan el flujo de control y son análogos a las flechas de los diagramas de flujo. Una arista debe terminar en un nodo, aunque el nodo no represente ninguna instrucción procedimental (por ejemplo, véase el símbolo en la gráfica de flujo para la construcción *if-then-else* de la figura 14.1). Las áreas que limitan aristas y nodos se denominan regiones. Cuando se cuentan las *regiones* se incluyen las áreas ubicadas fuera de la gráfica.[4]

FIGURA 14.1 Notación de gráfica de flujo.

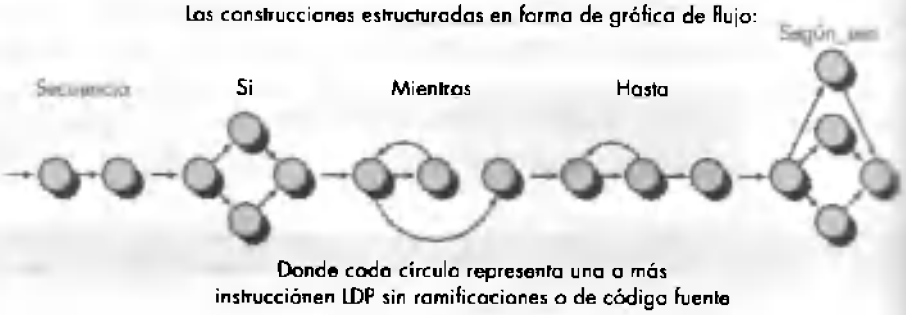
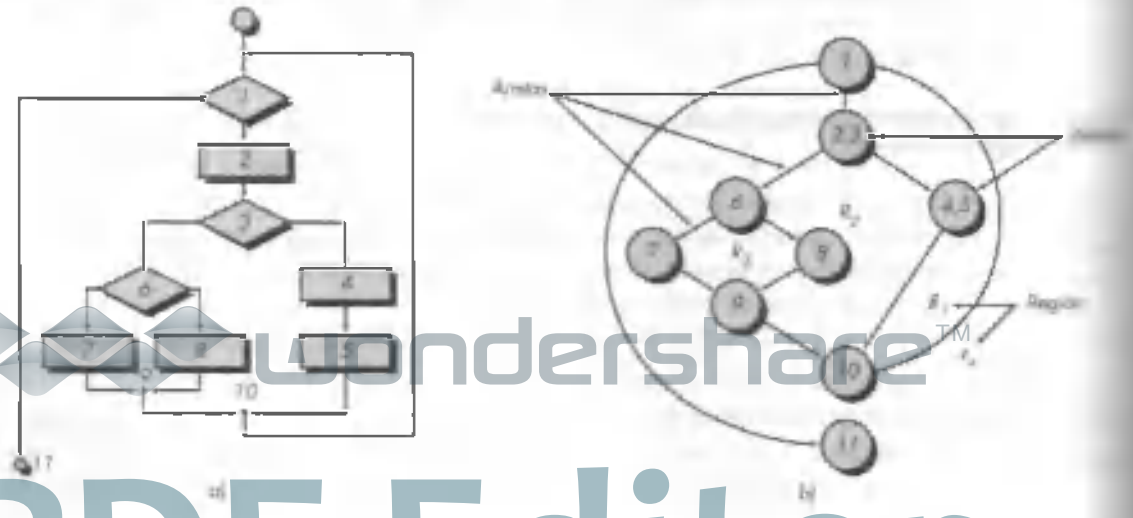


FIGURA 14.2 *a)* Diagrama de flujo y *b)* gráfica de flujo.



4 Un análisis más detallado de las gráficas y su aplicación se presentará en la sección 14.6.1

Cuando se encuentran condiciones compuestas en un diseño procedimental, la generación de una gráfica de flujo se vuelve ligeramente más complicada. Una condición compuesta ocurre cuando hay uno o más operadores booleanos (OR, AND, NAND, NOR) en una instrucción condicional. Tomando como referencia la figura 14.3, el segmento en LDP se traduce a la gráfica de flujo mostrada. Obsérvese que se crea un nodo separado para cada una de las condiciones a y b en la instrucción IF a OR b. Cada nodo que contiene una condición es un *nodo predicado* y se caracteriza porque de él emanan dos o más aristas.

14.4.2 Rutas independientes del programa

Una *ruta independiente* es cualquier ruta del programa que ingresa por lo menos un nuevo conjunto de instrucciones de procesamiento o una nueva condición. Cuando se explica desde el punto de vista de una gráfica de flujo, una ruta independiente debe recorrer por lo menos una arista que no se haya recorrido antes. Por ejemplo, a continuación se presenta un conjunto de rutas independientes en la gráfica de flujo de la figura 14.2b:

ruta 1: 1-11

ruta 2: 1-2-3-4-5-10-1-11

ruta 3: 1-2-3-6-8-9-10-1-11

ruta 4: 1-2-3-6-7-9-10-1-11

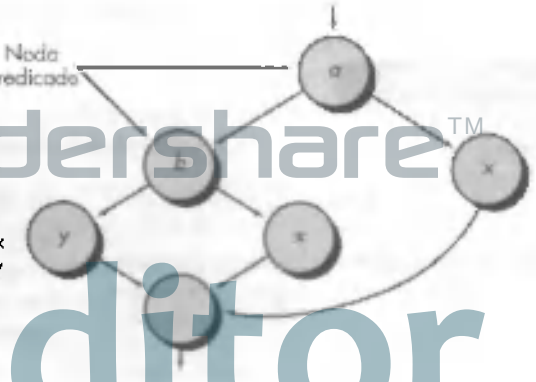
Obsérvese que cada nuevo camino ingresa una nueva arista. El camino

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

no se considera una ruta independiente porque se trata simplemente de una combinación de rutas ya especificadas y no recorre ninguna arista nueva.

Los caminos 1, 2, 3 y 4 constituyen un conjunto básico para la gráfica de flujo de la figura 14.2b. Es decir, si se diseñan pruebas para forzar la ejecución de esas rutas

FIGURA 14.3 Lógica compuesta.



IF a OR b
then procedimiento x
else procedimiento y
ENDIF

CONSEJO

La complejidad ciclomática es una métrica que resulta útil para predecir cuáles módulos tienen más probabilidad de contener errores. Se emplea para la planeación de pruebas además del diseño de casos de prueba.

(un conjunto básico), se habrán ejecutado los lados verdadero y falso de cada instrucción del programa. Debe observarse que un conjunto básico no es único. En realidad, es posible derivar varios conjuntos básicos diferentes de un diseño procedimental determinado.

¿Cómo se sabe cuántas rutas buscar? El cálculo de la *complejidad ciclomática* proporciona la respuesta. La complejidad ciclomática es una métrica de software que proporciona una medida cuantitativa de la complejidad lógica de un programa. Cuando se emplea en el contexto del método de prueba de la ruta básica, el valor calculado mediante la complejidad ciclomática define el número de rutas independientes en el conjunto básico de un programa, y proporciona un límite superior para el número de pruebas que deben aplicarse para asegurar que todas las instrucciones se hayan ejecutado por lo menos una vez.

La complejidad ciclomática se basa en la teoría gráfica y se calcula de una de tres maneras:

¿Cómo se calcula la complejidad ciclomática?

1. El número de regiones corresponde a la complejidad ciclomática.
2. La complejidad ciclomática, $V(G)$, de una gráfica de flujo, G, se define como
   $V(G) = E - N + 2$
   donde E es el número de aristas, y N, el número de nodos de la gráfica de flujo.
3. La complejidad ciclomática, V(G), de una gráfica de flujo, G, también se define como
   $V(G) = P + 1$
   donde P es el número de nodos predicado incluidos en la gráfica de flujo G.

PUNTO CLAVE

La complejidad ciclomática proporciona el límite superior del número necesario de casos de prueba para garantizar que cada instrucción del programa se haya ejecutado por lo menos una vez.

Tomando como referencia una vez más la gráfica de flujo de la figura 14.2b, la complejidad ciclomática se calcula empleando cada uno de los algoritmos que se acaban de indicar.

1. La gráfica de flujo tiene cuatro regiones.
2. $V(G)$ = 11 aristas − 9 nodos + 2 = 4
3. $V(G)$ = 3 nodos predicado + 1 = 4

HOGARSEGURO

Utilización de la complejidad ciclomática

La escena: Cubículo de Shakira.

Los actores: Vinod y Shakira, integrantes del equipo de ingeniería del software *HogarSeguro* que trabaja en la planeación de pruebas para la función de seguridad.

La conversación:

Shakira: Mira... Sé que debemos aplicar pruebas de unidad a los componentes de la función de seguridad pero hay demasiados, y si tomas en cuenta el número de operaciones que deben ejercitarse, no sé ... tal vez debe

olvidarnos de la prueba de caja blanca, integrar to- e iniciar la ejecución de las pruebas de caja negro

Vinod: ¿Crees que el tiempo es insuficiente para probar componentes, ejercitar las operaciones y luego inte-

Shakira: La fecha límite para el primer incremento está cerca de lo que pensábamos. . sí, claro, estoy preo-

Vinod: ¿Por qué no ejecutamos por lo menos pruebas caja blanca en las operaciones más propensas a

Shakira (exasperada): ¿Y exactamente cómo sé son las más propensas a error?

Vinod: V de G.

Shakira: ¿Qué?

Vinod: La complejidad ciclomática (V de G) Sólo calcu la V(G) para cada operación dentro de cada componente y verás cuáles tienen los valores más elevados Esos son los que están más propensos a error

Shakira: ¿Y cómo calculo V de G?

Vinod: Es muy fácil Aquí tienes un libro que describe cómo hacerlo

Shakira (hojeándola): Muy bien, no parece difícil. Lo probaré. Las operaciones con la V(G) más elevada serán candidatas para las pruebas de caja blanca.

Vinod: Sólo recuerdo que no hay garantías. Un componente con una V(G) baja aún puede estar propenso a errores.

Shakira: Muy bien Pero por lo menos esto me ayudará a reducir el número de componentes que necesariamente deben someterse a prueba de caja blanca

Lo más notable es que el valor de V(G) proporciona el límite superior del número de rutas independientes que forman el conjunto básico; por implicación, ofrece un límite superior del número de pruebas que debe diseñarse y ejecutarse para garantizar la cobertura de todas las instrucciones del programa

14.4.3 Derivación de casos de prueba

El método de prueba de la ruta básica se aplica a un diseño procedimental o al código fuente. En esta sección se presentará la prueba de la ruta básica como una serie de pasos. Se empleará el procedimiento promedio (descrito en PDL en la figura 14 4) como ejemplo para ilustrar cada paso en el método de diseño de casos de prueba Obsérvese que promedio, aun en el caso de un algoritmo extremadamente simple, contiene condiciones compuestas y bucles. Los siguientes pasos se aplican para derivar el conjunto básico:

1. **Utilizando el diseño o el código como base se dibuja la gráfica de flujo correspondiente.** En la creación de una gráfica de flujo se emplean los símbolos y las reglas de construcción presentadas en la sección 14.4 1 Tomando como referencia el PDL para obtener *promedio* en la figura 14.4, se crea una gráfica de flujo numerando esas instrucciones en PDL, que se correlacionarán o mapearán en los nodos correspondientes de la gráfica de flujo. En la figura 14.5 se muestra la gráfica de flujo resultante.
2. **Determínese la complejidad ciclomática de la gráfica de flujo resultante.** La complejidad ciclomática, *V*(G), se determina al aplicar el algoritmo descrito en la sección 14.4.2. Debe indicarse que podría determinarse *V*(G) sin desarrollar una gráfica de flujo, si se cuentan todas las instrucciones condicio-

FIGURA 14.4
PDL con nodos identificados.

```
PROCEDIMIENTO promedio;

  * Este procedimiento calcula el promedio de 100 o menos
    números que caen entre valores límite; también calcula la
    suma y el total de números válidos

  INTERFACE RETURNS promedio, total.entrada, total.válido;
  INTERFACE ACCEPTS valor, mínimo, máximo;

  TYPE valor[1:100] IS SCALAR ARRAY;
  TYPE promedio, total.entrada, total.válido;
     mínimo, máximo, suma IS SCALAR;
  TYPE i IS INTEGER;
  i = 1;
  total.entrada = total.válido = 0;  2
  suma = 0;
  DO WHILE valor[i] <> -999 AND total.entrada < 100  3
  4  incrementar total.entrada en 1;
     IF valor[i] >= mínimo AND valor[i] <= máximo  5
     6 THEN incrementar total.válido en 1;
        7  suma = suma + valor[i]
     ELSE omitir
  8  ENDIF
     incrementar i en 1;
  9  ENDDO
  IF total.válido > 0  10
  11  THEN promedio = suma / total.válido;
  12  ELSE promedio = -999;
  13 ENDIF
END promedio
```

nales en el PDL (para el procedimiento promedio, las condiciones compuestas cuentan como dos) y se suma 1 al resultado. Tomando como referencia la figura 14.5,

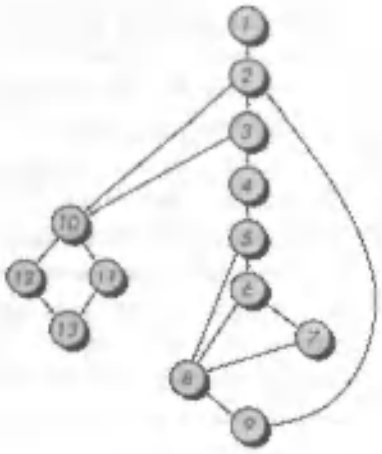
$V(G)$ = 6 regiones
$V(G)$ = 17 aristas − 13 nodos + 2 = 6
$V(G)$ = 5 nodos predicado + 1 = 6

> "Errar es de humanos, encontrar un error es de dioses."
>
> **Robert Dunn**

3. **Determínese un conjunto básico de rutas linealmente independientes.** El valor de $V(G)$ indica el número de rutas linealmente independientes de la estructura de control del programa. En el caso del procedimiento promedio se espera especificar seis caminos:

ruta 1: 1-2-10-11-13
ruta 2: 1-2-10-12-13
ruta 3: 1-2-3-10-11-13
ruta 4: 1-2-3-4-5-8-9-2-...
ruta 5: 1-2-3-4-5-6-8-9-2-...
ruta 6: 1-2-3-4-5-6-7-8-9-2-...

Figura 14.5
Gráfica de flujo para el procedimiento promedio.



Los puntos suspensivos (...) que siguen a las rutas 4, 5 y 6 indican que es aceptable cualquier ruta que se recorra en el resto de la estructura de control. A menudo vale la pena identificar nodos predicado como apoyo para derivar los casos de prueba. En este caso, los nodos 2, 3, 5, 6 y 10 son nodos predicado.

4. **Prepárense los casos de prueba que forzarán la ejecución de cada ruta en el conjunto básico.** Es necesario seleccionar los datos de manera tal que se establezcan apropiadamente las condiciones de los nodos predicado, a medida que se prueba cada ruta. Cada caso de prueba se ejecuta y compara con los resultados esperados. Una vez completados todos los casos, la persona que aplica la prueba puede estar segura de que todas las instrucciones del programa se han ejecutado por lo menos una vez.

Es importante observar que es imposible probar algunas rutas independientes (como la ruta 1 en nuestro ejemplo) por separado. Es decir, en el flujo normal del programa no puede obtenerse la combinación de los datos requeridos para recorrer la ruta. En tales casos, estas rutas se ejercitan como parte de otra prueba del camino.

Figura 14.6
Matriz de gráfica.

14.4.4 Matrices de gráficas

El procedimiento para derivar la gráfica de flujo e incluso determinar un conjunto de rutas básicas es sensible a la mecanización. Una estructura de datos denominada *matriz de gráfica* resulta muy útil para desarrollar una herramienta de software que ayude en la prueba de la ruta básica.

Una matriz de gráfica es una matriz cuadrada cuyo tamaño (es decir, el número de filas y columnas) es igual al número de nodos en la gráfica de flujo. Cada fila y columna corresponde a un nodo identificado, y las entradas de la matriz corresponden a las conexiones (una arista) entre nodos. En la figura 14.6 se muestra un ejemplo simple de una gráfica de flujo y su matriz de gráfica correspondiente [BEI90].

Tomando como referencia la figura, cada nodo en la gráfica está identificado con números, mientras que cada arista se identifica con letras. Una conexión entre dos nodos se indica creando una entrada de letra en la matriz. Por ejemplo, el nodo 3 se conecta al nodo 4 con la arista b.

¿Qué es una matriz de gráfica y cómo se extiende para usarla en la prueba?

Hasta este punto, la matriz de gráfica no es más que una representación tabular de una gráfica de flujo. Sin embargo, al agregar un *peso de enlace* a cada una de las entradas, la matriz de gráfica se convierte en una herramienta poderosa para evaluar la estructura de control del programa durante la prueba. El peso de enlace proporciona información adicional acerca del flujo de control. En su forma más simple, el peso de enlace es 1 (existe una conexión) o 0 (no existe una conexión). Pero a los pesos de enlace también se le asignan otras propiedades, más interesantes:

- La probabilidad de que se ejecute un enlace (arista).
- El tiempo de procesamiento gastado durante el recorrido a un enlace
- La memoria requerida durante el recorrido de un enlace.
- Los recursos requeridos durante el recorrido de un enlace.

Beizer [BEI90] proporciona un tratamiento completo de algoritmos matemáticos adicionales que son aplicables a una matriz de gráfica. El empleo de estas técnicas permite automatizar parcial o totalmente el análisis requerido para diseñar casos de prueba

> "Un error clásico es prestar más atención a la ejecución de las pruebas que a su diseño."
>
> Brian [illegible]

14.5 PRUEBAS DE LA ESTRUCTURA DE CONTROL

La técnica de prueba de la ruta básica descrita en la sección 14.4 es una de varias técnicas para la prueba de estructuras de control. Aunque la prueba de la ruta básica es simple y efectiva, no es suficiente por sí misma. En esta sección se analizan brevemente variaciones sobre la prueba de estructuras de control. Éstas amplían la cobertura de las pruebas y mejoran la calidad de la prueba de caja blanca

CONCEPTO CLAVE

s son mucho s en las s de las s lógicas que o mismo de nes de

14.5.1 Prueba de condición

La *prueba de condición* [TAI89] es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en un módulo del programa. Una *condición simple* es una variable booleana o una expresión relacional, tal vez precedida con un operador NOT (¬). Una expresión relacional toma la forma

$$E_1 \text{ <operador relacional> } E_2$$

donde E_1 y E_2 son expresiones aritméticas y <operador relacional> es uno de los siguientes: $<$, $\leq$, $=$, $\neq$ (desigual), $>$ o $\geq$. Una condición compuesta la integran dos o más condiciones simples, operadores booleanos y paréntesis. Se supone que entre los operadores booleanos permitidos en una condición compuesta se incluyen OR (|), AND (&) y NOT (¬). Una condición sin expresiones relacionales se considera una expresión booleana. Por tanto, los posibles tipos de elementos en una condición incluyen un operador booleano, una variable booleana, un par de paréntesis (que encierran una condición booleana simple o compuesta), un operador relacional o una expresión aritmética.

Si una condición es incorrecta, entonces por lo menos un componente de la condición es incorrecto. Por tanto, entre los tipos de errores en una condición se incluyen los presentes en el operador booleano (operadores booleanos incorrectos/faltantes/adicionales), en la variable booleana, en los paréntesis booleanos, en los operadores relacionales y en la expresión aritmética. El método de prueba de condición se concentra en la prueba de cada condición del programa para asegurar que no contiene errores.

14.5.2 Prueba del flujo de datos

El método de *prueba del flujo de datos* selecciona rutas de prueba en un programa de acuerdo con las ubicaciones de las definiciones y los usos de las variables en el programa. El enfoque de prueba del flujo de datos se ilustra suponiendo que a cada instrucción de un programa se le asigna un número de instrucción, y que ninguna función modifica sus parámetros o variables globales. En el caso de una instrucción con I como número de instrucción,

DEF(I) = {X | instrucción I contiene una definición de X}
USO(I) = {X | instrucción I contiene un uso de X}

Si la instrucción I es una instrucción *if* (*si*) o *loop* (bucle), su conjunto DEF está vacío y su conjunto USO se basa en la condición de la instrucción I. Se dice que la definición de la variable X en la instrucción I está viva en la instrucción I' si existe una ruta de la instrucción I a la I' que no contiene otra definición de X.

Una *cadena definición uso (DU)* de la variable X es de la forma [X, I, I'], donde I e I' son números de instrucción, X está en DEF(I) y USO(I'), y la definición de X en la instrucción I está viva en la I'.

CONSEJO

No es realista asegurar que la prueba del flujo de datos se usará de manera extensa cuando se pruebe un sistema grande. Sin embargo, puede usarse de una manera orientada a un blanco en áreas de software que están bajo sospecha.

Una estrategia simple de prueba de flujo de datos consiste en solicitar que cada cadena DU sea cubierta por lo menos una vez. Esta estrategia se denomina estrategia de prueba DU. Se ha mostrado que ésta no garantiza la cobertura de todas las ramas de un programa. Sin embargo, sólo en raras situaciones no se garantiza que cada rama esté cubierta por una prueba DU, como en las construcciones if-then-else (si-entonces-si_no) en que la parte *then* no tiene definición de alguna variable y la parte *else* no existe. En esta situación, la rama else de la instrucción *if* no está necesariamente cubierta por la prueba DU. Se han estudiado y comparado varias estrategias de prueba de flujo de datos (por ejemplo, [FRA88], [NTA88], [FRA93]). A los lectores interesados se les recomienda que consideren consultar esas referencias bibliográficas.

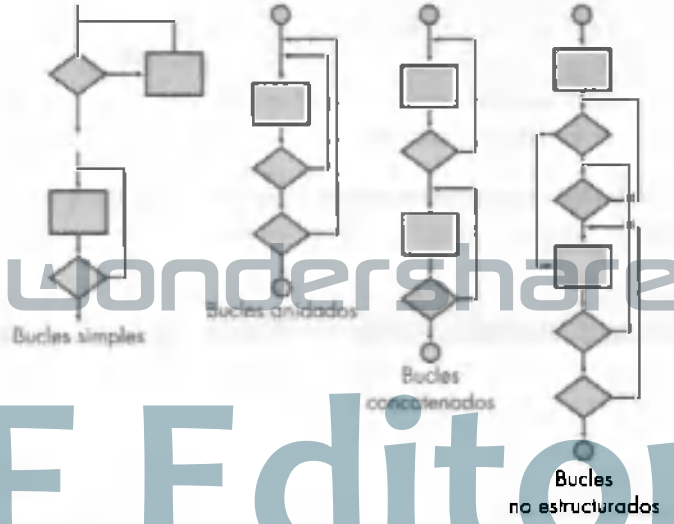
> "Los personas que destacan en la aplicación de pruebas son maestros en percibir que 'algo es gracioso' y actuar sobre ello."
>
> Brian Marick

14.5.3 Prueba de bucles

Los bucles son la piedra de toque para la gran mayoría de los algoritmos implementados en software. Y aun así, a menudo se les presta poca atención mientras se realizan pruebas de software.

La *prueba de bucles* es una técnica de prueba de caja blanca que se concentra exclusivamente en la validez de la construcción de bucles. Es posible definir cuatro diferentes clases de bucles [BEI90]: bucles simples, concatenados, anidados y no estructurados (figura 14.7).

FIGURA 14.7
Clases de bucles.



Bucles simples. El siguiente conjunto de pruebas se aplica a bucles simples, donde n es el número máximo de pasos que permite el bucle.

1. Omitir por completo el bucle
2. Sólo un paso por el bucle.
3. Dos pasos por el bucle.
4. m pasos por el bucle, donde $m < n$.
5. $n = 1, n, n + 1$ pasos por el bucle

Bucles anidados. Si se fuese a extender el enfoque de prueba de los bucles simples a los anidados, el número de pruebas posibles crecería geométricamente a medida que aumente el nivel de anidamiento. Esto generaría un número poco práctico de pruebas. Beizer [BEI90] sugiere un enfoque que ayudará a reducir el número de pruebas:

1. Iniciar en el bucle más interno. Asignar a todos los bucles los valores mínimos.
2. Aplicar pruebas de bucle simple al más interno mientras se mantienen los externos en los valores mínimos del parámetro de iteración (como el contador de bucles). Agregar otras pruebas para los valores fuera de rango o excluidos
3. Trabajar hacia fuera, conduciendo pruebas para el siguiente bucle, pero manteniendo todos los demás bucles externos en valores mínimos y otros bucles anidados en valores "típicos".
4. Seguir mientras no se hayan probado todos los bucles.

Bucles concatenados. Los bucles concatenados se prueban empleando el enfoque definido para los bucles simples, si cada uno de los bucles es independiente. Sin embargo, si dos bucles están concatenados y el contador del bucle 1 se emplea como valor inicial para el bucle 2, entonces los bucles no son independientes. Cuando los bucles no lo son, entonces se recomienda el enfoque aplicado a los bucles anidados.

Bucles no estructurados. Siempre que sea posible, esta clase de bucles debe diseñarse nuevamente para reflejar el uso de las construcciones de programación estructurada (capítulo 11).

CONSEJO

...e podrá probar ...eficacia los bucles ...estructurados, ...necesario volver a ...diseñarlos.

14.6 PRUEBA DE CAJA NEGRA

Las *pruebas de caja negra*, también denominadas *pruebas de comportamiento*, se concentran en los requisitos funcionales del software. Es decir, permiten al ingeniero de software derivar conjuntos de condiciones de entrada que ejercitarán por completo todos los requisitos funcionales de un programa. La prueba de caja negra no es una opción frente a las técnicas de caja blanca. Es, en cambio, un enfoque complementario que tiene probabilidades de descubrir una clase diferente errores de los que se descubrirían con los métodos de caja blanca

Las pruebas de caja negra tratan de encontrar errores en las siguientes categorías: 1) funciones incorrectas o faltantes, 2) errores de interfaz, 3) errores en estructuras de datos o en acceso a bases de datos externas, 4) errores de comportamiento o desempeño, y 5) errores de inicialización y término.

A diferencia de las pruebas de caja blanca, que se realizan al inicio del proceso de prueba, las de caja negra tienden a aplicarse durante las últimas etapas de la prueba (consúltese el capítulo 13). Debido a que éstas desatienden a propósito la estructura de control, la atención se concentra en el dominio de la información. Las pruebas están diseñadas para responder las siguientes preguntas:

¿Cuáles preguntas responden las pruebas de caja negra?

- ¿Cómo se prueba la validez funcional?
- ¿Cómo se prueban el comportamiento y el desempeño del sistema?
- ¿Cuáles clases de entrada serán buenos casos de prueba?
- ¿El sistema es particularmente sensible a ciertos valores de entrada?
- ¿Cómo se aíslan los límites de una clase de datos?
- ¿Cuáles tasas de datos y cuál volumen tolera el sistema?
- ¿Qué efecto tienen combinaciones específicas de datos sobre la operación del sistema?

Al aplicar técnicas de caja negra se deriva un conjunto de casos de prueba que satisfacen los siguientes criterios [MYE79]: 1) casos de prueba que reducen, mediante una cuenta mayor que uno, el número de casos de prueba adicionales que deben diseñarse para alcanzar una prueba razonable, y 2) casos de prueba que indican algo acerca de la presencia o ausencia de clases de errores, en lugar de un error asociado sólo con la prueba específica a la mano.

14.6.1 Métodos gráficos de prueba

PUNTO CLAVE

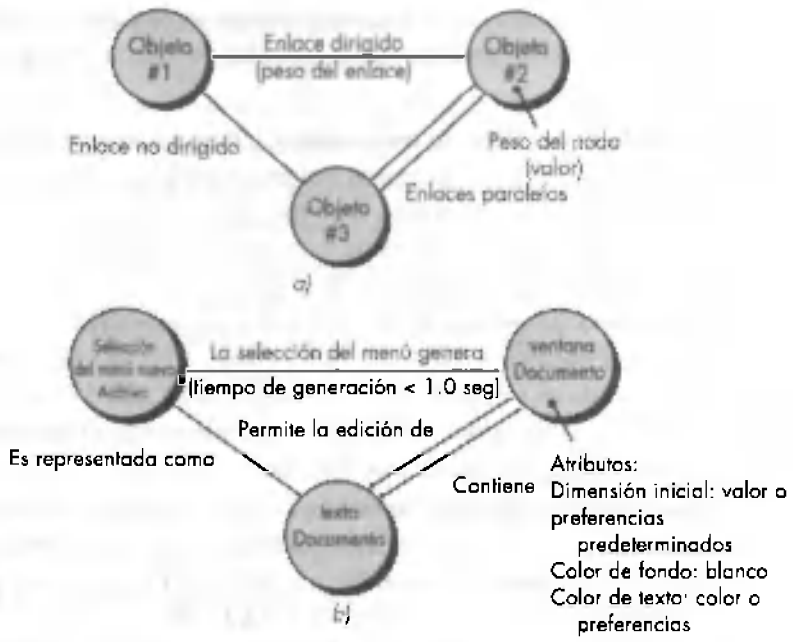
Una gráfica representa las relaciones entre los objetos de datos y los de programa, lo que permite derivar casos de prueba que busquen errores asociados con estas relaciones

El primer paso en la prueba de caja negra es comprender los objetos[5] modelados en el software y la relación entre ellos. Una vez que se ha logrado, el siguiente paso consiste en definir la serie de pruebas que verifican que "todos los objetos tienen la relación esperada entre sí" [BEI95]. Explicado de otra manera, la prueba de software empieza al crear una gráfica de objetos importantes y sus relaciones y luego diseñar una serie de pruebas que cubran la gráfica de tal manera que se ejercite cada objeto y relación y que se descubran los errores.

Para dar estos pasos, el ingeniero de software empieza creando *una gráfica*: una colección de *nodos* que representan objetos, enlaces que representan la relación entre objetos, *pesos de nodo* que describen las propiedades de un nodo (como un valor de datos o un comportamiento de estado específico) y *pesos de enlace* que describen algunas características de un enlace.

5 En este caso el término "objetos" se considera en el contexto más amplio posible. Abarca objetos, componentes (módulos) tradicionales y elementos orientados a objetos del software de co

FIGURA 14.8
a) Notación gráfica y *b)* ejemplo simple.



En la figura 14.8a se muestra una representación simbólica de una gráfica. Los nodos se representan como círculos conectados por enlaces que toman un número diferente de formas. Un *enlace directo* (representado por una flecha) indica que una relación se mueve en una sola dirección. Un *enlace bidireccional,* también denominado *enlace simétrico,* indica que la relación se aplica en ambas direcciones. Los *enlaces paralelos* se emplean cuando se establece un número diferente de relaciones entre los nodos de la gráfica.

Como ejemplo simple, considérese una parte de la gráfica para una aplicación de procesamiento de palabras (figura 14.8b) donde

Objeto #1 = **nuevoArchivo** (menú selección)
Objeto #2 = **ventanaDocumento**
Objeto #3 = **textoDocumento**

Si se toma como referencia la figura, una selección del menú en **nuevoArchivo** genera una ventana de documento. El peso del nodo de **ventanaDocumento** proporciona una lista de los atributos de la ventana que se esperaban cuando ésta se generó. El peso del enlace indica que la ventana debe generarse en menos de 1.0 segundos. Un enlace indirecto establece una relación simétrica entre la selección del menú **nuevoArchivo** y **textoDocumento**, y los enlaces paralelos indican las relaciones entre **ventanaDocumento** y **textoDocumento**. En realidad, tendría que generarse una gráfica mucho más detallada como precursora del diseño de casos de prueba. El ingeniero de software deriva entonces los casos de prueba al recorrer la

gráfica y cubrir cada una de las relaciones mostradas. Estos casos de prueba se diseñarán en un intento de encontrar errores en cualquiera de las relaciones.

Beizer [BEI95] describe varios métodos de prueba de comportamiento que usan gráficas:

Modelado del flujo de transacción. Los nodos representan pasos de alguna transacción (por ejemplo, los pasos requeridos para hacer una reservación en una aerolínea empleando un servicio en línea) y los enlaces representan la conexión lógica entre los pasos. El diagrama de flujo de datos (capítulo 8) se utiliza para ayudar en la creación de gráficas de este tipo

Modelado de estado finito. Los nodos representan los diferentes estados que el usuario observa en el software (por ejemplo, cada una de las "pantallas" que aparecen cuando un empleado toma un pedido por teléfono) y los enlaces representan las transiciones que ocurren para ir de un estado a otro. El diagrama de estado (capítulo 8) ayuda a crear gráficas de este tipo.

Modelado del flujo de datos. Los nodos son objetos de datos, y los enlaces son las transformaciones que ocurren para traducir un objeto de datos en otro. Por ejemplo, el nodo **impuesto retenido** por **FICA (IRF)** se calcula a partir de **salario neto (SN)** empleando la relación **IRF = 0.62 × SN**

Modelado relacionado con el tiempo. Los nodos son objetos de programa, y los enlaces son las conexiones secuenciales entre esos objetos. Con los pesos de enlace se especifican los tiempos de ejecución requeridos mientras el programa se ejecuta.

Un análisis detallado de cada uno de estos métodos gráficos de prueba se encuentra más allá del alcance de este libro. El lector interesado debe consultar [BEI95] para tener una cobertura completa.

14.6.2 Partición equivalente

La *partición equivalente* **es un método** de prueba de caja negra que divide el dominio de entrada de un programa en clases de datos a partir de las cuales pueden derivarse casos de prueba. Un caso de prueba ideal de manejo simple descubre una clase de errores (por ejemplo, procesamiento incorrecto de todos los datos de caracteres) que, de otra manera, requeriría la ejecución de muchos casos antes de que se observe el error general. La partición equivalente se esfuerza por definir un caso de prueba que descubra ciertas clases de errores, reduciendo así el número total de casos de prueba que deben desarrollarse.

CONSEJO

Las condiciones de entrada son conocidas en una etapa relativamente temprana del proceso de software. Por esto, debe empezarse a pensar en la partición equivalente mientras se [illegible] el diseño.

El diseño de casos de prueba para partición equivalente se basa en una evaluación de las clases de equivalencia para una condición de entrada. Con el uso de los conceptos introducidos en la sección anterior, si es posible enlazar un conjunto de objetos mediante relaciones simétricas, transitivas y reflexivas, entonces existe una clase de equivalencia [BEI95]. Una clase de equivalencia representa un conjunto de estados válidos y no válidos para las condiciones de entrada. Por lo general

condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición booleana. Las clases de equivalencia se definen de acuerdo con las siguientes directrices:

¿Cómo se definen las clases de equivalencia para las pruebas?

1. Si una condición de entrada especifica un rango, se definen una clase de equivalencia válida y dos no válidas.
2. Si una condición de entrada requiere un valor específico, se definen una clase de equivalencia válida y dos no válidas
3. Si una condición de entrada especifica un miembro de un conjunto, se definen una clase de equivalencia válida y otra no válida
4. Si una condición de entrada es booleana, se definen una clase de equivalencia válida y otra no válida.

Al aplicar estas directrices para la derivación de clases de equivalencia, se desarrollarán y ejecutarán los casos de prueba para cada objeto de los datos del dominio de entrada. Los casos de prueba se seleccionan de modo que el mayor número de atributos de clase de equivalencia se ejercita una vez.

14.6.3 Análisis de valores límite

Es mayor el número de errores que se presenta en los límites del dominio de entrada que en el "centro"; por ello se ha desarrollado el *análisis de valores límite* (AVL) como técnica de prueba. El AVL lleva a una selección de casos que prueba los valores límite.

El análisis de valores límite es una técnica de diseño de casos de prueba que complementa la partición equivalente. En lugar de seleccionar cualquier elemento de una clase de equivalencia, el AVL lleva a la selección de casos de prueba en las "aristas" de la clase. En lugar de concentrase exclusivamente en las condiciones de entrada, el AVL también deriva casos de prueba del dominio de salida [MYE79].

Las directrices para el AVL son muy similares a las proporcionadas para la partición equivalente:

PUNTO CLAVE
AVL extiende la partición equivalente al concentrarse en los casos de las "aristas" de una clase de equivalencia

1. Si una condición de entrada especifica un rango limitado por los valores a y b, los casos de prueba deben diseñarse con esos valores, además de los que se encuentran apenas arriba y abajo de ellos.
2. Si una condición de entrada especifica diversos valores, deben desarrollarse casos de prueba que ejerciten los números máximo y mínimo. También se prueban los valores ubicados apenas arriba y abajo de estos máximos y mínimos.
3. Aplicar las directrices 1 y 2 a las condiciones de salida. Por ejemplo, supóngase que una tabla que compara presión y temperatura se requiere como salida de un programa de análisis de ingeniería. Los casos de prueba deben diseñarse para crear un informe de salida que produzca el número máximo (y mínimo) permisible para las entradas de la tabla

¿Cómo puedo crear casos de prueba AVL?

4. Si la estructura interna de datos del programa tiene límites prescritos (por ejemplo, una matriz que tiene un límite definido de cien entradas) debe diseñarse un caso de prueba para ejercitar los límites de la estructura de datos

La mayoría de los ingenieros de software realizan intuitivamente el AVL, hasta cierto grado. Al aplicar estas directrices, la prueba de límites estará más completa y, por tanto, tendrá una mayor probabilidad de detectar errores.

> "El cohete Ariane 5 estalló al despegar debido a un simple defecto (error) del software que se relacionaba con la conversión de un valor de punto flotante de 64 bits en un entero de 16 bits. El cohete y sus cuatro satélites corecían no seguro y costaron 500 millones de dólares. Una prueba completa del sistema hubiera encontrado el error, pero fue omitida por razones de presupuesto."
>
> Informe noticia

14.6.4 Prueba de tabla ortogonal

Hay muchas aplicaciones en las cuales el dominio de entrada es relativamente limitado. Es decir, el número de parámetros es pequeño y los valores que cada parámetro puede tomar están claramente limitados. Cuando estos números son muy pequeños (por ejemplo, tres parámetros de entrada toman tres valores discretos cada uno), es posible considerar cada permutación de entrada y probar exhaustivamente el procesamiento del dominio de entrada. Sin embargo, a medida que crece el número de valores de entrada, junto con el número de valores discretos para cada elemento, la prueba exhaustiva se vuelve poco práctica o imposible.

PUNTO CLAVE

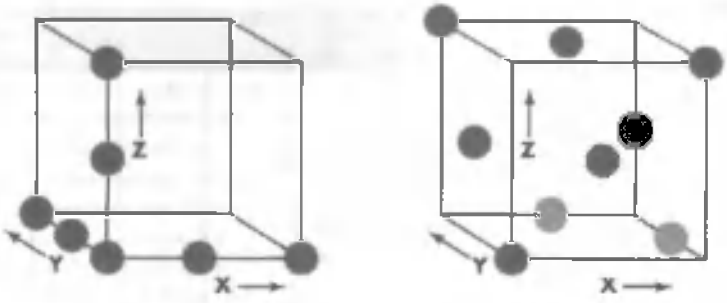
La prueba de tabla ortogonal permite diseñar casos de prueba que proporcionan la máxima cobertura de prueba con un número razonable de casos de prueba.

La *prueba de tabla ortogonal* se aplica en problemas en los cuales el dominio de entrada es relativamente pequeño, pero demasiado grande para una prueba exhaustiva. El método de prueba de tabla ortogonal resulta útil sobre todo para encontrar errores asociados con las *fallas de región* (una categoría de error asociada con los efectos de la lógica en un componente de software).

Ilustrar la diferencia entre la prueba de tabla ortogonal y los enfoques más convencionales de "un elemento de entrada a la vez" requiere imaginar un sistema con tres elementos de entrada, *X*, *Y* y *Z*. Cada uno de estos elementos tiene tres valores discretos asociados. Hay $3^3 = 27$ casos de prueba posibles. Phadke [PHA97] sugiere un concepto geométrico, que se ilustra en la figura 14.9, para los posibles casos de prueba asociados con *X*, *Y* y *Z*. Si toma como referencia la figura, sólo un elemento de entrada podría variar a un mismo tiempo en la secuencia que sigue cada caso de entrada. Esto da como resultado un cobertura relativamente limitada del dominio de entrada (que representa el cubo de la izquierda de la figura).

Cuando se presenta una prueba de tabla ortogonal se crea una *tabla ortogonal* de casos de prueba, la cual tiene una "propiedad de equilibrio" [PHA97]. Es decir, los casos de prueba (representados con puntos azules en la figura) están "uniformemente dispersos por todo el dominio de la prueba", como se ilustra en el cubo de la izquierda de la figura 14.9. La cobertura de prueba en el dominio de entrada es más completa

Figura 14.9 Guía geométrica de casos de prueba [PHA97].



Un elemento de entrada a la vez **Tabla ortogonal L9**

Para ilustrar el uso de la tabla ortogonal L9, considérese la función enviar de una aplicación de fax. Se pasan cuatro parámetros, P1, P2, P3 y P4 a la función enviar. Cada uno toma tres valores discretos. Por ejemplo, P1 toma los valores:

P1 = 1, enviarlo ahora

P1 = 2, enviarlo en una hora

P1 = 3, enviarlo a medianoche

P2, P3 y P4 también podrían tomar los valores 1, 2 y 3, representando otras funciones de enviar.

Si se eligiera una estrategia de prueba "un elemento de entrada a la vez", se especificarían las siguientes secuencias de prueba (P1, P2, P3, P4): (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2) y (1, 1, 1, 3) Phadke [PHA97] evalúa estos casos de prueba al afirmar:

> Estos casos de prueba sólo son útiles cuando se está seguro de que los parámetros de prueba no interactúan. Detectarán fallas de lógica donde un solo valor de parámetro hace que el software funcione mal. Se trata de *fallas de modalidad simple*. Este método no detecta fallas de lógica que provoquen un mal funcionamiento cuando dos o más parámetros toman ciertos valores simultáneamente; es decir, no detecta interacciones. Por tanto, su capacidad para detectar fallas está limitada.

Dado el número relativamente pequeño de parámetros de entrada y valores discretos es posible aplicar una prueba exhaustiva. El número de pruebas requeridas es $3^4 = 81$ (grande, pero manejable). Se encontrarían todas las fallas asociadas con permutación de elementos de datos, pero el esfuerzo requerido es relativamente alto.

El enfoque de prueba de tabla ortogonal permite proporcionar buena cobertura de prueba con un número considerablemente menor de casos de prueba que la estrategia exhaustiva. En la figura 14.10 se muestra una tabla ortogonal L9 para la función *enviar* de fax.

A continuación se presenta la evaluación de Phadke [PHA97] acerca de las pruebas aplicadas con la tabla ortogonal:

FIGURA 14.10

Tabla ortogonal L9.

Caso de prueba	Parámetros de prueba			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Detecta y aísla todas las fallas de modalidad simple. Una falla de modalidad simple es un problema consistente con cualquier nivel de cualquier parámetro simple. Por ejemplo, si todos los casos de prueba del factor P1 = 1 causan una condición de error, se trata de una falla de modalidad simple. En este ejemplo, las pruebas 1, 2 y 3 (figura 14.10) mostrarán errores. Al analizar la información acerca de cuáles pruebas muestran errores, se identifican cuáles valores de parámetros causan la falla. En este ejemplo, al notar que las pruebas 1, 2 y 3 causan errores, se aísla [el procesamiento lógico asociado con "elija lo ahora" (P1 = 1)] como la fuente del error. Este aislamiento de la falla es importante para corregirla.

Detecta todas las fallas de modalidad doble. Si existe un problema con cuando se presentan niveles específicos de dos parámetros al mismo tiempo, se le determina *falla de modalidad doble*. Por supuesto, se trata de una indicación de incompatibilidad de un par de valores o de interacciones dañinas entre dos parámetros de prueba.

Fallas multimodales. Las tablas ortogonales (del tipo mostrado) sólo aseguran la detección de fallas de modalidad simple y doble. Sin embargo, estas pruebas también detectan muchas fallas multimodales.

Un análisis detallado de las pruebas de tabla ortogonal se encontrará en [PHAD

HERRAMIENTAS DE SOFTWARE

Diseño de casos de prueba

Objetivo: Ayudar al equipo de software a desarrollar un conjunto completo de casos de prueba de caja blanca y negra.

Mecánica: Estas herramientas caen en dos amplias categorías: pruebas estáticas y dinámicas. Se emplean tres tipos diferentes de herramientas de prueba estática en la herramientas de prueba basadas en código, lenguajes de prueba especializados y herramientas de prueba basadas en requisitos. Las herramientas de prueba basadas en código aceptan código fuente como entrada y realizan varios análisis que llevan a la generación de casos de

...prueba. Los lenguajes de prueba especializados (como ...) permiten que un ingeniero de software escriba especificaciones detalladas que describen cada caso de prueba y la logística para su ejecución. Las herramientas de prueba basados en requisitos aíslan requisitos específicos del usuario y sugieren casos de prueba (o clases de prueba) que ejercitarán los requisitos. Las herramientas automáticas de prueba interactúan con un programa de ejecución, revisando la cobertura del camino, probando las condiciones relacionadas con el valor de variables específicas e instrumentando de otra manera el flujo de ejecución del programa.

Herramientas representativas[6]

McCabe Test, desarrollada por McCabe & Associates (www.mccabe.com), implementa diversas técnicas de prueba de la ruta derivadas de una evaluación de la complejidad ciclomática y de otras métricas de software.

Panorama, desarrollada por International Software Automation, Inc. (www.softwareautomation.com), abarca un juego completo de herramientas para desarrollar software orientado a objetos, incluidas herramientas que ayudan a diseñar casos de prueba y planear pruebas

TestWorks, desarrollada por Software Research Inc. (www.soft.com/Products), es un juego completo de herramientas automatizadas de prueba que ayudan al diseño de casos de prueba para software desarrollado en C/C++ y Java y proporcionan soporte a pruebas de regresión

T-Vec Test Generation System, desarrollado por T-VEC Technologies (www.t-vec.com), es un conjunto de herramientas que da soporte a pruebas de unidad, integración y validación al ayudar en el diseño de casos de prueba empleando información contenida en una especificación de requisitos orientada a objetos

14.7 MÉTODOS DE PRUEBAS ORIENTADAS A OBJETOS

La arquitectura del software orientado a objetos genera una serie de subsistemas separados en capas que encapsulan las clases que colaboran entre sí. Cada uno de estos elementos del sistema (subsistemas y clases) realiza funciones que ayudan a cumplir con los requisitos del sistema. Es necesario probar un sistema orientado a objetos en diferentes niveles para descubrir errores que podrían ocurrir a medida que las clases colaboran entre sí y los subsistemas se comunican entre las capas de la arquitectura

En el aspecto estratégico, la prueba orientada a objetos es similar a la de los sistemas convencionales, pero es diferente en el aspecto táctico. Debido a que los modelos de análisis y diseño orientados a objetos tienen una estructura y un contenido similares al programa orientado a objetos resultante, la "prueba" podría empezar con la revisión de estos modelos. Una vez que se ha generado el código, la prueba orientada a objetos real empieza por lo "pequeño", con una serie de pruebas diseñadas para ejercitar las operaciones de clase y examinar si existen errores a medida que una clase colabora con otra. Cuando las clases se integran para formar un subsistema, se aplica la prueba basada en el uso, junto con los enfoques basados en fallas, para ejercitar plenamente las clases que colaboran entre sí. Por último, se emplean los casos de uso para descubrir errores al nivel de validación del software.

El diseño convencional de casos de prueba lo determina el concepto entrada-proceso-salida del software o el detalle algorítmico de módulos individuales. La prueba

6 Las herramientas expuestas aquí sólo representan una muestra de esta categoría. En casi todos los casos los nombres de las mismas son marcas registradas de sus respectivos desarrolladores

orientada a objetos se concentra en el diseño de secuencias apropiadas de operaciones para ejercitar los estados de una clase.

Referencia Web

Una excelente colección de artículos y recursos sobre pruebas orientadas a objetos se encontrará en www.rbsc.com.

14.7.1 Implicaciones del concepto orientado a objetos en el diseño de casos de prueba

A medida que la clase evoluciona mediante los modelos de análisis y diseño se vuelve un destino para el diseño de casos de prueba. Debido a que los atributos y las operaciones están encapsulados, las operaciones de prueba fuera de la clase suelen ser improductivas. Aunque el encapsulamiento es un concepto de diseño esencial en la orientación a objetos, representa un obstáculo menor cuando se prueba, como indica Binder [BIN94]: "La prueba debe informar sobre el estado concreto y abstracto de un objeto". Sin embargo, el encapsulamiento llega a dificultar la adquisición de esta información. A menos que se proporcionen operaciones integradas para reportar los valores de los atributos de clase, será difícil obtener una instantánea del estado de un objeto.

La herencia también plantea desafíos adicionales para el diseñador de casos de prueba. Ya se ha observado que cada nuevo contexto de uso requiere una nueva serie de prueba, aunque se haya alcanzado la reutilización. Además, la herencia múltiple complica la prueba más allá de aumentar el número de contextos que requieren [7] prueba [BIN94]. Si las subclases que se convierten en instancias a partir de una superclase se usan dentro del mismo dominio del problema, es posible usar el conjunto de casos de prueba derivado de la superclase cuando se prueba la subclase. Sin embargo, si ésta se emplea en un contexto completamente nuevo, los casos de prueba de la superclase no serán aplicables y será necesario diseñar un nuevo conjunto de pruebas

14.7.2 Aplicabilidad de métodos convencionales de diseño de casos de prueba

Los métodos de prueba de caja blanca descritos en secciones anteriores pueden aplicarse a las operaciones definidas para una clase. Las técnicas de flujo de datos o de prueba de la ruta básica o de bucle ayudan a asegurar que se han probado todas las instrucciones de una operación. Sin embargo, la estructura concisa de muchas operaciones de clase hace que algunos argumenten que el esfuerzo aplicado a la prueba de caja blanca podría redirigirse mejor para probar un nivel de clase

Los métodos de prueba de caja negra son tan apropiados para los sistemas orientados a objetos como los sistemas que se desarrollan con métodos convencionales de ingeniería de software Como ya se indicó en este mismo capítulo, los casos de uso proporcionan información útil para el diseño de pruebas de caja negra y basadas en el estado [AMB95].

7 Un concepto de orientación a objetos que debe usarse con extremo cuidado.

14.7.3 Prueba basada en fallas[8]

El objetivo de la *prueba basada en fallas* dentro del sistema orientado a objetos es diseñar pruebas que tengan una alta probabilidad de descubrir posibles fallas. Debido a que el producto o sistema debe cumplir con los requisitos del cliente, la planeación preliminar requerida para realizar la prueba basada en fallas empieza con el modelo del análisis. La persona que aplica la prueba busca fallas posibles (es decir, aspectos de la implementación del sistema que generen defectos). Determinar si existen estas fallas requiere diseñar casos de prueba que revisen el diseño o el código.

Por supuesto, la efectividad de estas técnicas depende de la manera en que las personas que aplican las pruebas adviertan una falla posible. Si las fallas reales en un sistema orientado a objetos se consideran poco posibles, entonces este método en realidad no es mejor que cualquier técnica de prueba al azar. Sin embargo, si los modelos de análisis y diseño arrojan luz sobre lo que tal vez esté mal, entonces la prueba basada en fallas encontrará una cantidad importante de errores con gastos relativamente mínimos de esfuerzo.

La prueba de integración (cuando se aplica en un contexto orientado a objetos) busca fallas en llamadas a operación o en conexiones entre mensajes. Tres tipos de fallas se encuentran en este contexto: resultado inesperado, operación incorrecta/mensaje empleado, invocación incorrecta. Para determinar las fallas a medida que se invocan las funciones (operaciones), debe examinarse el comportamiento de la operación.

La prueba de integración se aplica a los atributos y a las operaciones. Los "comportamientos" de un objeto los definen los valores que se asignan a sus atributos. La prueba debe ejercitar los atributos para determinar si ocurren valores apropiados para distintos tipos de comportamiento de objeto.

Es importante observar que las prueba de integración tratan de encontrar errores en el objeto cliente, no en el servidor. Explicado en términos convencionales, el eje de la prueba de integración es determinar si existen errores en el código que llama, no en el código llamado. La llamada a la operación se usa como una pista, una manera de encontrar los requisitos de prueba que ejercitan el código que llama.

> "Si se quiere y espera que un programa funcione, lo más probable es que se vea un programa funcionando (y que se pasen por alto las fallas)."
>
> **Cem Kaner *et al.***

[8] De la sección 14.7.3 a la 14.7.6 se ha adaptado de un artículo de Brian Marick publicado en el grupo de noticias de Internet **componente.testing**. Esta adaptación se incluye con el permiso del autor. Para conocer mayor información sobre estos temas consúltese [MAR94]. Debe observarse que las técnicas analizadas de las secciones 14.7.3 a 14.7.6 son aplicables al software convencional.

14.7.4 Casos de prueba y jerarquía de clase

La herencia no obvia la necesidad de una prueba completa de todas las clases derivadas. En realidad, llega a complicar el proceso de prueba. Imagínese la siguiente situación. Una clase **Base** contiene las operaciones *heredado( )* y *redefinido( )*. Una clase **Derivado** define *redefinido( )* para que sirva en un contexto local. Hay pocas dudas de que debe probarse **Derivado::redefinido( )** porque representa un nuevo diseño y un nuevo código. Pero ¿debe probarse de nuevo **Derivado::heredado( )**?

PUNTO CLAVE

Aunque se haya probado por completo una clase de base, aún se tendrá que probar todas las clases derivadas de ella.

Si **Derivado::heredado( )** llama a *redefinido( )* y el comportamiento *redefinido( )* ha cambiado, es posible que **Derivado::heredado( )** maneje erróneamente el nuevo comportamiento. Por tanto, se necesitan nuevas pruebas aunque no hayan cambiado el diseño ni el código. Sin embargo, es importante observar que sólo es posible realizar un subconjunto de todas las pruebas de **Derivado::heredado( )**. Si parte del diseño o el código de *heredado( )* no depende de *redefinido( )* (es decir, no lo llama a él ni ningún código que lo llame indirectamente), es innecesario probar de nuevo ese código en la clase derivada.

Base::redefinido() y **Derivado::redefinido** son operaciones distintas con diferentes especificaciones e implementaciones. Cada una tendría un conjunto de requisitos de prueba derivados de la especificación y la implementación. Esos requisitos de prueba revisan fallas posibles: fallas de integración, de condición, de límites, etc. Sin embargo, es probable que las operaciones sean similares, sus conjuntos de requisitos de prueba se superpondrán. Mientras mejor sea el diseño orientado a objetos, mayor será la superposición. Es necesario derivar nuevas pruebas exclusivamente para los requisitos de **Derivado::redefinido( )** que no se satisfagan con las pruebas de **Base::redefinido( )**.

En resumen, las pruebas de **Base::redefinido( )** se aplican a objetos de la clase **Derivado**. Las entradas de prueba son apropiadas para las clases de base y derivadas, pero los resultados esperados podrían diferir en la clase derivada.

14.7.5 Prueba basada en escenarios

La prueba basada en fallas soslaya dos tipos importantes de errores: 1) especificaciones incorrectas y 2) interacciones entre subsistemas. Cuando ocurren errores asociados con especificaciones incorrectas, el producto no hace lo que el cliente quiere. Podría hacer lo incorrecto, u omitir funcionalidad importante. En cualquier caso se merma la calidad (el cumplimiento de los requisitos). Los errores asociados con las interacciones entre subsistemas ocurren cuando el comportamiento de un subsistema crea circunstancias (como eventos o flujo de datos) que causan la falla de otro subsistema.

PUNTO CLAVE

La prueba basada en escenarios descubrirá errores que ocurren [illegible] cualquier actor [illegible] con el [illegible].

La *prueba basada en escenarios* se concentra en lo que hace el usuario, no en lo que hace el producto. Esto significa que se deben capturar las tareas (mediante casos de uso) que el usuario tiene que realizar y luego aplicarlas, junto con sus variantes, como pruebas.

Los escenarios descubren los errores de interacción. Pero esto se logra si los casos de prueba son más complejos y realistas que las pruebas basadas en fallas.

pruebas basadas en escenarios tienden a ejercitar varios subsistemas en una sola prueba (los usuarios no se limitan al uso de un subsistema a la vez).

Como ejemplo, considérese el diseño de pruebas basadas en escenarios para un editor de texto mediante la revisión de los siguientes casos de uso informales:

Caso de uso: *Corregir el borrador final.*

Antecedentes: Es común que se imprima el borrador "final", se lea y se descubran algunos errores molestos y confusos en la imagen en pantalla. Este caso de uso describe la secuencia de eventos que se presenta cuando ocurre esto

1 Se imprime todo el documento.
2 Se recorre el documento, cambiando ciertas páginas.
3 A medida que se hacen cambios, se imprime página por página.
4 A veces se imprime una serie de páginas.

Este escenario describe dos cosas: una prueba y necesidades específicas del usuario. Las necesidades del usuario son obvias: 1) un método para imprimir páginas individuales, y 2) un método para imprimir un rango de páginas. A medida que se aplica la prueba, debe probarse la edición después de imprimir (y a la inversa). La persona que aplica la prueba espera descubrir que la función de impresión causa errores en la función de edición; es decir, que las dos funciones del software no tienen la independencia apropiada.

CONSEJO

...a la prueba ...o en escenarios ...méritos, se ...n mejores ...s por tiempo ...o si se revisan ...s de uso cuando ...esarrollan como ...del modelo de ...

Caso de uso: *Imprimir una nueva copia.*

Antecedentes: Alguien pide al usuario una nueva copia del documento. Debe imprimirse.

1. Se abre el documento.
2. Se imprime.
3. Se cierra el documento.

Una vez más, el enfoque de la prueba es relativamente obvio, con la excepción de que este documento no aparece de la nada. Se creó en una tarea inicial. ¿Aquella tarea afecta a ésta?

En muchos editores modernos, los documentos recuerdan cómo se imprimieron por última vez. Como opción predeterminada, se imprimen de la misma manera la siguiente ocasión. Después del escenario *Corregir el borrador final*, con sólo seleccionar "Imprimir" en el menú y hacer clic en el botón Imprimir del cuadro de diálogo se imprimirá de nuevo la última versión corregida. De modo que, de acuerdo con el editor, el escenario correcto debe tener este aspecto:

Caso de uso: *Imprimir una nueva copia*

1. Se abre el documento.
2. Se selecciona "Imprimir" en el menú

3. Se revisa si está imprimiendo un rango de páginas; si es así, se hace clic para imprimir todo el documento.
4. Se hace clic en el botón Imprimir.
5. Se cierra el documento.

Pero este escenario indica un posible error de especificación. El editor no hace lo que el usuario espera razonablemente que haga. Los clientes con frecuencia omiten la revisión indicada en el paso 3. Se sentirán molestos cuando vayan a la impresora y encuentren una página cuando querían 100. Los clientes molestos señalarán errores de especificación.

Un diseñador de casos de prueba podría pasar por alto esta dependencia al diseñar la prueba, pero es probable que el problema surja durante la prueba. El responsable de ésta tendría entonces que enfrentarse a la probable respuesta: "¡así se supone que debía funcionar!"

14.7.6 Estructuras de superficie y de fondo en pruebas

Estructura de superficie es la estructura observable externamente de un programa orientado a objetos. Es decir, la estructura que resulta inmediatamente obvia para el usuario final. En lugar de realizar funciones, se le dan objetos determinados al usuario de muchos sistemas orientados a objetos para que los manipule. Pero cualquiera que sea la interfaz, las pruebas aún se basan en tareas de usuario. La captura de estas tareas requiere comprensión, observación y comunicación con usuarios representativos (y con todos los usuarios que no lo son que valga la pena tomar en cuenta).

PUNTO CLAVE

La prueba de estructura de superficie es análoga a la prueba de caja negra. La de estructura de fondo es similar a la de caja blanca

Seguramente habrá algunas diferencias en los detalles. Por ejemplo, en un sistema convencional con una interfaz orientada a comandos, el usuario podría revisar todos los comandos como lista de verificación de una prueba. Si no existen escenarios de prueba para ejercitar un comando, es probable que la prueba soslaye algunas tareas del usuario (o que la interfaz tenga comandos inútiles). En una interfaz orientada a objetos el responsable de la prueba podría emplear todos los objetos como una lista de verificación de una prueba.

Las mejores pruebas se derivan cuando el diseñador observa el sistema de manera nueva o poco convencional. Por ejemplo, si el sistema o el producto tiene una interfaz basada en comandos, las pruebas más completas se derivarán cuando el diseñador del caso de prueba pretende que las operaciones sean independientes de los objetos. Deben plantearse preguntas como: "¿El usuario deseará usar esta operación (que sólo se aplica al objeto **escáner**) mientras trabaja con la impresora?" Sin importar cuál sea el estilo de la interfaz, el diseño de casos de prueba que ejercite la estructura de superficie debe usar objetos y operaciones como pistas que conduzcan a tareas omitidas.

La estructura a fondo representa los detalles técnicos internos de un programa orientado a objetos. Es decir, la estructura que se comprende al examinar el diseño o el código, o ambos. La prueba de estructura de fondo está diseñada para ejercitar dependencias, comportamientos y mecanismos de comunicación que se han

cido como parte del modelo de diseño (capítulos 9-12) para el software orientado a objetos.

Los modelos de análisis y diseño son la base de la prueba de estructura de fondo. Por ejemplo, el diagrama de colaboración UML o el modelo de despliegue describe las colaboraciones entre objetos y subsistemas que tal vez no sean visibles externamente. El diseñador de casos de prueba pregunta entonces: ¿hemos capturado (como prueba) algunas tareas que ejercitan la colaboración indicada en el diagrama de colaboración? Si no es así, ¿por qué?

> "No se avergüence de los errores ni los convierta en crímenes."
>
> Confucio

14.8 MÉTODOS DE PRUEBA APLICABLES AL NIVEL DE CLASE

En el capítulo 13 se indicó que la prueba del software empieza por lo "pequeño" y lentamente avanza hacia lo "grande". Se prueba en el pequeño entorno de una sola clase y los métodos que están encapsulados en la clase. La prueba aleatoria y la partición son métodos que se emplean para ejercitar una clase durante una prueba orientada a objetos [KIR94].

14.8.1 Prueba aleatoria para clases orientadas a objetos

CONSEJO

... de posibles ... en una ... aleatoria llega ... rse muy ... La eficiencia ... prueba mejorará ... alizara una ... similar a la ... de tabla ...

Para ilustrar brevemente estos métodos, imagínese una aplicación bancaria en que una clase **Cuenta** tiene las siguientes operaciones: *abrir( )*, *configurar( )*, *depositar( )*, *retirar( )*, *saldar( )*, *sumar( )*, *limiteCredito( )* y *cerrar( )* [KIR94]. Cada una de estas operaciones se aplica a **Cuenta**, pero hay ciertas restricciones (por ejemplo, la cuenta debe abrirse antes de aplicar otras operaciones, y debe cerrarse después de que se han completado todas las operaciones) relacionadas con la naturaleza del problema. Aun con estas restricciones, hay muchas permutas en las operaciones. El historial de comportamiento mínimo de una instancia de **Cuenta** incluye las siguientes operaciones:

abrir • configurar • depositar • retirar • cerrar

Esto representa la secuencia de prueba mínima para **Cuenta**. Sin embargo, podría presentarse una amplia variedad de comportamientos distintos dentro de esta secuencia.

abrir • configurar • depositar • [depositar | retirar | saldar | sumar | limiteCredito]^n • retirar • cerrar

Es posible generar al azar varias secuencias diferentes de operaciones. Por ejemplo:

Caso de prueba r_1: **abrir • configurar • depositar • depositar • saldar • sumar • retirar • cerrar**

Caso de prueba r_2: **abrir • configurar • depositar • retirar • depositar • saldar • limiteCredito • retirar • cerrar**

Éstas y otras pruebas de orden aleatorio se aplican para ejercitar diferentes historiales de instancias de clase

HOGARSEGURO

Prueba de clase

La escena: Cubículo de Shakira

Los actores: Jamie y Shakira, integrantes del equipo de ingeniería del software *HogarSeguro* que están trabajando en el diseño de casos de prueba para la función de seguridad.

La conversación:

Shakira: He desarrollado algunas pruebas para la clase **Detector** [figura 11.4], ya sabes, la que permite acceso a todos los objetos de **Sensor** para la función de seguridad. ¿Estás familiarizada con ella?

Jamie (sonriendo): Claro, es la que te permite agregar el sensor "antiperros"

Shakira: Esa misma. De cualquier manera, tiene una interfaz con cuatro operaciones. *leer()*, *activar()*, *desactivar()* y *probar()*. Antes de que un sensor tenga la posibilidad de leer, debe activarse. Una vez activado, puede leerse y probarse. Es posible desactivarlo en cualquier momento, a menos que se haya procesado una condición de alarma. De modo que definí una secuencia simple de prueba que ejercitará su historial de comportamiento.

(Muestra a Jamie la siguiente secuencia.)

#1: activar • probar • leer • desactivar

Jamie: Eso funcionará, ¡pero tienes que probar más que eso!

Shakira: Lo sé, lo sé. He aquí otras secuencias que he elaborado.

(Muestra a Jamie las siguientes secuencias.)

#2: activar • probar • [leer]* • desactivar

#3: [leer]*

#4: activar • desactivar • (probar | leer)

Jamie: Déjame ver si comprendo tu intención. #1 recorre el historial completo, lo que representa una especie de uso convencional. #2 repite la operación leer n veces, y ese es un escenario probable. #3 trata de leer el sensor antes de que se active —eso debe producir algún tipo de mensaje de error, ¿verdad? #4 activa y desactiva el sensor y luego trata de leerlo. ¿No es lo mismo que la prueba 3?

Shakira: En realidad no. En #4 el sensor se ha probado. Lo que realmente prueba #4 es si la operación desactivar funciona como debería. Si se presentan leer() o probar() después de desactivar(), debe generarse un mensaje de error. Si no lo hace, entonces tenemos un error en la operación desactivar.

Jamie: Estupendo. Sólo recuerda que las cuatro pruebas tienen que aplicarse a cada tipo de sensor, porque cada operación puede tener diferencias sutiles, dependiendo del tipo de sensor.

Shakira: No te preocupes. Ése es el plan.

14.8.2 Prueba de partición al nivel de clase

La prueba de partición reduce el número de casos de prueba requeridos para probar la clase de manera muy parecida a la partición equivalente (sección 14.6) para el software convencional. La entrada y la salida se ordenan en categorías y se diseñan casos de prueba para ejercitar cada categoría. ¿Cómo se derivan las categorías de partición?

? ¿Cuáles son opciones de prueba están disponibles al nivel de clase?

La *partición basada en el estado* ordena en categorías las operaciones de clase a partir de su capacidad para cambiar el estado de la clase. Si se piensa una vez más en la clase **Cuenta**, las operaciones de estado incluyen *depositar( )* y *retirar( )*, entre tras que las que no son de estado incluyen *saldar( )*, *sumar( )* y *limiteCredito( )*. Las pruebas están diseñadas de manera que ejercitan por separado las operaciones que cambian de estado y las que no lo hacen. Por tanto,

Caso de prueba p_1: abrir • configurar • depositar • depositar • retirar • retirar • cerrar

Caso de prueba p_2: abrir • configurar • depositar • sumar • limiteCredito • retirar • cerrar

El caso de prueba p_1 cambia de estado, mientras que el caso de prueba p_2 ejercita operaciones que no cambian de estado (aparte de las que se encuentran en la secuencia de prueba mínima)

La *partición basada en atributos* ordena en categorías las operaciones de clase basadas en los atributos que utilizan. En el caso de la clase **Cuenta**, los atributos **saldar** y **limiteCredito** se emplean para definir particiones. Las operaciones se dividen en tres particiones: 1) operaciones que usan **limiteCredito**, 2) operaciones que modifican **limiteCredito**, y 3) operaciones que no usan ni modifican **limiteCredito**. Entonces se diseñan secuencias de prueba para cada partición.

La *partición basada en categorías* ordena en categorías las operaciones de clase con base en la función genérica que cada una realiza. Por ejemplo, las operaciones de la clase **Cuenta** se organizan en operaciones de inicialización [*abrir( )*, *configurar( )*], operaciones computacionales [*depositar( )*, *retirar( )*], consultas [*saldar( )*, *sumar( )*, *limiteCredito( )*] y de terminación [*cerrar( )*]

14.9 Diseño de caso de prueba de interclase

El diseño de casos de prueba se vuelve más complicado cuando empieza la integración del sistema orientado a objetos. En esta etapa debe empezar la prueba de colaboración entre clases. Para ilustrar la "generación de casos de prueba de interclase" [KIR94], se expande el ejemplo del sistema bancario presentado en la sección 14.8 para que incluya las clases y colaboraciones indicadas en la figura 14.11. La dirección de las flechas en la figura indica la dirección de los mensajes. Y las etiquetas individuales indican las operaciones que se invocan como consecuencia de la colaboración indicada por los mensajes

Como en la prueba de clases individuales, la prueba de colaboración entre clases se lleva a cabo al aplicar métodos aleatorios y de partición, además de pruebas basadas en el escenario y de comportamiento.

14.9.1 Prueba de clases múltiples

Kirani y Tsai [KIR94] sugieren la siguiente secuencia de pasos para generar varios casos de prueba aleatorios de clases múltiples:

1. En cada clase cliente use la lista de operaciones de clase para generar una serie de secuencias de pruebas aleatorias. Las operaciones enviarán mensajes a otras clases del servidor.
2. En cada mensaje generado, determínese la clase colaboradora y la operación correspondiente en el objeto servidor
3. En cada operación del objeto servidor (invocada por los mensajes enviados desde el objeto cliente) determínense los mensajes que transmite.
4. En cada uno de los mensajes, determínese el siguiente nivel de operaciones invocadas e incorpórelos en la secuencia de prueba.

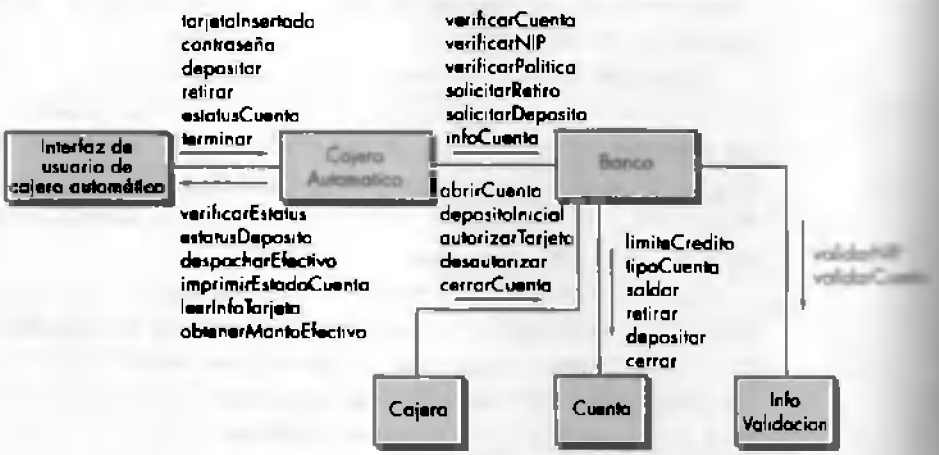


FIGURA 14.11
Diagrama de colaboración de clases para una aplicación bancaria (adaptado de [KIR94]).

Como ejemplo [KIR94], considérese una secuencia de operaciones para la clase **Banco** relacionada con una clase **CajeroAutomatico** (figura 14.11):

verificarCuenta • verificarNIP • [[verificarPolitica • solictudRetiro] | solicitarDeposito | solicitarCuenta]*

Un caso de prueba aleatoria para la clase **Banco** sería:

Caso de prueba r_3 = **verificarCuenta • verificarNIP • solicitarDeposito**

Considerar a los colaboradores que participan en la prueba requiere tomar en cuenta los mensajes asociados con cada una de las operaciones indicadas en el caso de prueba r_3. **Banco** debe colaborar con **infoValidacion** para ejecutar *verificarCuenta()* y *verificarNIP()*. **Banco** debe colaborar con **Cuenta** para ejecutar *solicitarDeposito()*. Por tanto, se tiene un nuevo caso de prueba que ejercita estas colaboraciones:

Caso de prueba r_4 = **verificarCuentaBanco[validarCuentaInfoValidacion]**
verificarNIPBanco • [validarNIPInfovalidacion] • solicitarDeposito • [depositarcuenta]

El enfoque para la prueba de partición de clases múltiples es similar al enfoque empleado para la de clases individuales. Como se analizó en la sección 14.8.2, se somete a partición una sola clase. Sin embargo, se expande la secuencia de prueba para incluir las operaciones invocadas mediante mensajes a las clases que colaboran. Un enfoque alterno lleva a cabo la partición de las pruebas con base en las interfaces de una clase determinada. Si se toma como referencia la figura 14.11, se **Banco** recibe mensajes de las clases **CajeroAutomatico** y **Cajero**. Por métodos dentro de **Banco** se prueban al particionarlas entre las que sirven a **roAutomatico** y a **Cajero**. La partición basada en el estado (sección 14.8.2) para refinar aún más las particiones.

14.9.2 Pruebas derivadas de modelos de comportamiento

En el capítulo 8 se analizó el uso del diagrama de estado como modelo para representar el comportamiento dinámico de una clase. El diagrama de estado de una clase ayuda a derivar una secuencia de pruebas que revisa el comportamiento dinámico de la clase (y las clases que colaboran con ella). En la figura 14.12 [KIR94] se muestra un diagrama de estado para la clase **Cuenta** que ya se analizó. Si se observa, las transiciones iniciales recorren los estados *vaciar Cuenta* y *configurar Cuenta*. Un retiro final y un cierre de la cuenta causan que la clase **Cuenta** haga transiciones a los estados *cuentaInactiva* y *cuentaMuerta*, respectivamente.

Las pruebas que se diseñen deben cubrir todos los estados [KIR94]. Es decir, las secuencias de operación deben lograr que la clase **Cuenta** haga una transición a todos los estados permisibles:

Caso de prueba s_1: **abrir • configurCuenta • depositar(inicial) • retirar(final) • cerrar**

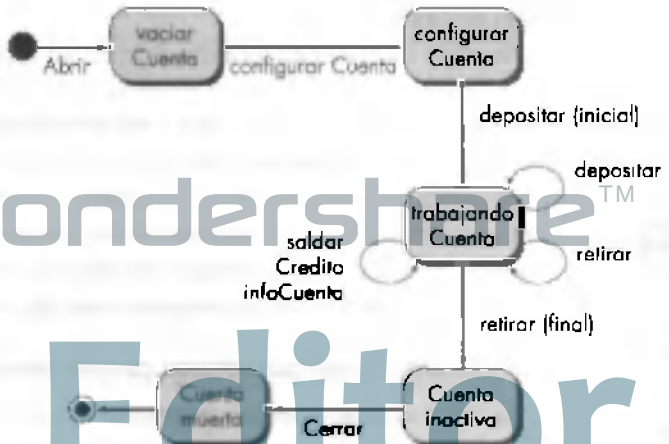
Debe notarse que esta secuencia es idéntica a la secuencia de la prueba mínima tratada en la sección 14.9.1. La secuencia de la prueba adicional se agrega a la sucesión mínima.

Caso de prueba s_1: **abrir • configurarCuenta • depositar(inicial) • depositar • saldar • acreditar • retirar(final) • cerrar**

Caso de prueba s_2: **abrir • configurarCuenta • depositar(inicial) • depositar • retirar • infoCuenta • retirar(final) • cerrar**

Es posible derivar aún más casos de prueba para asegurar que todos los comportamientos de la clase se hayan ejercitado adecuadamente. En situaciones en que el comportamiento de la clase da como resultado la colaboración con una o más clases, se utilizan varios diagramas de estado para registrar el flujo del comportamiento del sistema.

FIGURA 14.12
Diagrama de estado para la clase Cuenta (adaptado de [KIR94]).



El modelo de estado puede recorrerse de una manera "primero-en-anch[illegible]" [MGR94]. En este contexto, primero-en-anchura indica que un caso de prueba s[illegible] ejercita una transición. Cuando debe probarse una nueva transición sólo se u[illegible] transiciones probadas previamente.

Imagínese que el objeto **TarjetaCredito** es parte del sistema bancario. El [illegible] inicial de **TarjetaCredito** es *indefinido* (es decir, no se ha proporcionado un n[illegible]ro de tarjeta de crédito). Tras leer la tarjeta durante una venta, el objeto toma un [illegible]tado *definido*; es decir, se definen los atributos **numero, tarjeta** y **fecha vencimiento** [illegible]to con los identificadores específicos del banco. La tarjeta de crédito es *r*[illegible] cuando se le envía para autorización, y es *aprobada* cuando se recibe la au[illegible]ción La transición de **TarjetaCredito** de un estado a otro se prueba derivando [illegible]sos de prueba que causen la transición. Un método primero-en-anchura para este [illegible] de prueba no ejercitaría *remitida* antes de *indefinida* o *definida*. En este caso, usaría [illegible]siciones que no se han probado y, por tanto, violaría el criterio primero-en-anchura.

14.10 Prueba de entornos especializados: arquitecturas y aplicaciones

Los métodos de prueba analizados en secciones anteriores suelen aplicarse [illegible] dos los entornos, las arquitecturas y las aplicaciones, pero en ocasiones se [illegible]cionan directrices y enfoques únicos para las pruebas. En esta sección se a[illegible] las directrices para prueba de los entornos, las arquitecturas y las aplicaciones [illegible]cializadas que suelen encontrar los ingenieros de software.

CONSEJO

Úsese una estrategia de prueba similar a la aleatoria o de partición (sección 14.8) para diseñar pruebas de interfaz de usuario.

14.10.1 Pruebas de interfaces gráficas de usuario

Las *interfaces gráficas de usuario* (GUI, por sus siglas en inglés) plantean d[illegible] teresantes a los ingenieros de software Debido a los componentes reutiliza[illegible] porcionados como parte de los entornos de desarrollo de la GUI, la creación de [illegible]terfaz de usuario consume menos tiempo y es más precisa (capítulo 12). Pero a [illegible]mo tiempo ha crecido la complejidad de las GUI, lo que dificulta más el di[illegible] ejecución de casos de prueba.

Debido a que muchas GUI modernas tienen un aspecto y un modo de uso [illegible]res, es posible derivar una serie de pruebas estándar. Las gráficas de mod[illegible] estado finito se usan para derivar una serie de pruebas que ejerciten objetos [illegible]ficos de datos y programas que resultan relevantes para la GUI.

Debido al gran número de permutaciones asociadas con las operaciones [illegible] GUI, la prueba debe reducirse empleando herramientas automatizadas. En [illegible]mos años ha aparecido en el mercado una amplia serie de herramientas de [illegible] de GUI. Para un mayor análisis al respecto consulte el capítulo 12.

Referencia Web

En el siguiente sitio Web se encontrará información y recursos útiles [illegible] prueba de clientes/servidor: [illegible]w.cast-[illegible]

14.10.2 Prueba de arquitecturas cliente/servidor

La arquitectura cliente/servidor representa un importante desafío para [illegible] prueban el software. La naturaleza distribuida de los entornos cliente/ser[illegible]

aspectos de desempeño relacionados con el proceso de transacción, la posible presencia de varias plataformas de hardware diferentes, la complejidad de la comunicación en red, la necesidad de servir a varios clientes desde una base de datos centralizada (o, en algunos casos, distribuida) y los requisitos de coordinación impuestos al servidor se combinan para que la prueba de las arquitecturas de software cliente/servidor resulte considerablemente más difícil que la prueba de aplicaciones independientes. En realidad, estudios recientes de la industria indican un aumento importante en el tiempo y costo de la prueba cuando se desarrollan entornos cliente/servidor.

> "En el tema de las pruebas existe una buena dosis de similitud entre los sistemas tradicional y cliente/servidor."
>
> **Kelley Bourne**

En general, la prueba del software cliente/servidor se presenta en tres niveles diferentes: 1) aplicaciones de cliente individuales se prueban en una modalidad "desconectada"; la operación del servidor y la red no se toman en cuenta; 2) el software de cliente y las aplicaciones asociadas del servidor se prueban en conjunto, pero las operaciones de red no se ejercitan de manera explícita; 3) se prueba toda la arquitectura cliente/servidor, incluida la operación y el desempeño de la red.

Aunque muchos tipos diferentes de prueba se conducen en cada uno de estos niveles de detalle, los siguientes enfoques de prueba suelen encontrarse para aplicaciones cliente/servidor:

- **Pruebas de funcionalidad de la aplicación.** La funcionalidad de las aplicaciones de cliente se prueba empleando los métodos analizados en este capítulo. En esencia, la aplicación se prueba de manera independiente.
- **Pruebas de servidor.** Se prueban las funciones de coordinación y manejo de datos del servidor. También se considera el desempeño del servidor (tiempo de respuesta y procesamiento total de los datos).
- **Pruebas de base de datos.** Se prueba la exactitud e integridad de los datos almacenados en el servidor. Se examinan las transacciones que realizaron las aplicaciones de cliente para asegurar que los datos se almacenan, actualizan y recuperan apropiadamente. También se prueba la función de archivado.
- **Pruebas de transacción.** Se crea una serie de pruebas para asegurar que cada clase de transacciones se procesa de acuerdo con sus requisitos. Las pruebas se concentran en determinar si es correcto el procesamiento y en aspectos del desempeño (por ejemplo, tiempos de procesamiento de las transacciones y volumen de éstas).
- **Pruebas de comunicaciones de red.** Con estas pruebas se verifica que la comunicación entre los nodos de la red ocurre de manera correcta y que el paso de mensajes, las transacciones y el tráfico de la red relacionado se realiza sin errores. También es posible realizar pruebas de seguridad de la red como parte de estas pruebas.

Para completar estos enfoques de prueba, Musa [MUS93] recomienda el desarrollo de *perfiles operacionales* derivados de escenarios de uso de cliente/servidor.[9] Un perfil operacional indica la manera en que diferentes tipos de usuarios interoperan con el sistema cliente/servidor. Es decir, los perfiles proporcionan un "patrón de uso" aplicable al diseño y ejecución de las pruebas.

14.10.3 Prueba de la documentación y las funciones de ayuda

El término *prueba del software* evoca imágenes de grandes cantidades de casos de prueba preparados para ejercitar los programas de cómputo y los datos que manipulan. Si se recuerda la definición de software presentada en el primer capítulo de este libro, es importante observar que la prueba también debe extenderse al tercer elemento de la configuración del software: la documentación.

Los errores en la documentación son tan devastadores para la aceptación del programa como los errores en los datos o el código fuente. Nada es más frustrante que seguir una guía de usuario o una función de ayuda en línea con toda pulcritud y obtener resultados o comportamiento que no coinciden con los descritos en la documentación. Por eso la prueba de la documentación debe ser una parte significativa de cualquier plan de prueba del software.

La prueba de la documentación se aborda en dos fases. En la primera, revisión o inspección (capítulo 26), se examina la claridad editorial del documento. En la segunda fase, prueba en vivo, se emplea la documentación junto con el programa real.

INFORMACIÓN

Prueba de documentación

Las siguientes preguntas deben responderse durante la prueba de documentación, de función de ayuda, o ambas:

- ¿La documentación describe con exactitud la manera en que se realiza cada modalidad de uso?
- ¿Es exacta la descripción de cada secuencia de interacciones?
- ¿Los ejemplos son exactos?
- ¿La terminología, las descripciones de menú y las respuestas del sistema son consistentes con el programa real?
- ¿Es realmente fácil localizar una guía dentro de la documentación?
- ¿El uso de la documentación facilita la detección y resolución de errores?
- ¿El contenido y el índice del documento son exactos y completos?
- ¿El diseño del documento (formato, tipo de letra, sangrías, imágenes) es apropiado para comprender y asimilar rápidamente la información?
- ¿Todos los mensajes de error del software que se despliegan para el usuario están descritos con más detalle en el documento? ¿Las acciones que deben emprenderse como consecuencia de un mensaje de error están claramente delineadas?
- Si se usan los vínculos de hipertexto, ¿son exactos y completos?
- Si se usa el hipertexto, ¿el diseño de la navegación es apropiado para la información requerida?

La única manera viable de responder a estas preguntas es hacer que un tercero (por ejemplo, algunos usuarios seleccionados) pruebe la documentación en el contexto del uso del programa. Se habrán de indicar todas las discrepancias y definirse las áreas débiles o ambiguas del documento para una posible reescritura.

9 Debe observarse que los perfiles operacionales pueden aplicarse para probar todos los tipos de arquitecturas del sistema, no sólo la cliente/servidor.

14.10.4 Prueba de sistemas de tiempo real

La naturaleza asincrónica, dependiente del tiempo, de muchas aplicaciones en tiempo real agrega un elemento nuevo (y difícil en potencia) a la mezcla de pruebas: el tiempo. El diseñador del caso de prueba no sólo debe considerar los casos de prueba convencionales, sino también el manejo de eventos (es decir, el procesamiento de interrupciones), la temporización de los datos y el paralelismo entre las tareas (procesos) que manejan los datos. En muchas situaciones, los datos de prueba proporcionados cuando el sistema en tiempo real está en un estado producirán un procesamiento apropiado, mientras que los mismos datos proporcionados cuando el sistema esté en un estado diferente provocarán un error.

Por ejemplo, el software en tiempo real que controla una nueva fotocopiadora acepta interrupciones del operador (es decir, el operador del equipo oprime teclas como REINICIAR u OSCURECER) sin error cuando el equipo hace copias (en el estado *copiar*). Si estas mismas interrupciones del operador se ingresan cuando el equipo se encuentra en el estado *atasco*, se perderá la pantalla de visualización del código de diagnóstico (indicando la ubicación del atasco), lo que representa un error.

Además, la íntima relación entre el software en tiempo real y su entorno de hardware llegan a causar problemas en la prueba. Las pruebas del software deben considerar el impacto de las fallas del hardware en el procesamiento del software. Resulta extremadamente difícil simular estas fallas de manera realista.

¿Cuál es una estrategia efectiva para la prueba de un sistema en tiempo

Los métodos exhaustivos de diseño de casos de prueba para sistemas en tiempo real siguen evolucionando. Sin embargo, es posible proponer una estrategia de cuatro pasos:

- **Prueba de tareas.** El primer paso en la prueba del software en tiempo real consiste en probar cada tarea de manera independiente. Es decir, se diseñan y ejecutan pruebas convencionales para cada tarea. (Cada tarea se ejecuta de manera independiente durante estas pruebas.) La prueba de tareas descubre errores de lógica y funcionamiento, pero no de tiempo ni de comportamiento.
- **Prueba de comportamiento.** Con el empleo de modelos del sistema creados con herramientas automatizadas es posible simular el comportamiento de un sistema en tiempo real y examinarlo como una consecuencia de eventos externos. Estas actividades de análisis sirven como base para el diseño de casos de prueba que se realizan cuando se ha construido el software en tiempo real.
- **Prueba intertareas.** Una vez que se hayan aislado los errores en tareas individuales y en el comportamiento del sistema, la prueba se desplaza hacia los errores relacionados con el tiempo. Se prueban las tareas asincrónicas de las cuales se sabe que se comunican entre sí, empleando diferentes tasas de datos y cargas de procesamiento para determinar si ocurrirán errores de sincronización intertareas. Además, se prueban las tareas que se comunican por medio de la cola de mensajes o el almacén de datos para descubrir errores en el tamaño de estas áreas de almacenamiento de datos.

- **Prueba del sistema.** El software y el hardware están integrados, de mod que se aplica un rango completo de pruebas del sistema (capítulo 13) para tar de descubrir errores en la interfaz software/hardware. Casi todos los mas en tiempo real procesan interrupciones. Por tanto, resulta esencial la prueba del manejo de estos eventos booleanos. Por medio del diagrama de estado y la especificación de control (capítulo 8), el responsable de la pr desarrolla una lista de todas las interrupciones posibles y el procesamien que ocurre como consecuencia de las interrupciones. Entonces se diseña pruebas para evaluar las siguientes características del sistema.
  - —¿Se han asignado y manejado apropiadamente las propiedades de in ción?
  - —¿Se ha manejado correctamente el procesamiento de cada interrupci
  - —¿El desempeño de cada procedimiento de manejo de interrupciones ejemplo, el tiempo de procesamiento) cumple con los requisitos?
  - —¿Un elevado volumen de interrupciones que llega en momentos críticos problemas en la función o el desempeño?

Además, deben probarse áreas de datos globales que se emplean para t información como parte de un procesamiento de interrupciones, con el fin de luar la posibilidad de generación de efectos colaterales.

14.11 PATRONES DE PRUEBA

Referencia Web

Aportaciones a más de 70 patrones de prueba se encontrarán en www.rbsc.com.

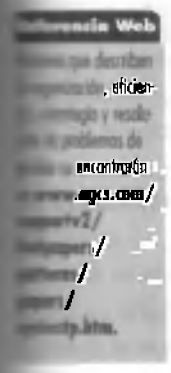
En capítulos anteriores se analizó el uso de patrones como mecanismo para d bir los bloques de construcción del software o situaciones de ingeniería del s Estos bloques de construcción o situaciones se repiten a medida que se co diferentes aplicaciones o que se realizan diferentes proyectos. Como sus co tes en el análisis y el diseño, los *patrones de prueba* describen bloques de c ción o situaciones frecuentes y que los responsables de probar el software reutilizar al afrontar la prueba de algún sistema nuevo o revisado.

¡PUNTO CLAVE

Los patrones de prueba ayudan a un equipo de software a comunicarse mejor sobre la prueba y facilitan a comprender mejor las fuerzas que llevan a un enfoque específico de prueba.

Los patrones de prueba no sólo proporcionan a los ingenieros del softwar directriz útil cuando empiezan las actividades de prueba, también proporci beneficios adicionales descritos por Marick [MAR02]:

1. Proporcionan una terminología a los encargados de la resolución de los probl "Hey, ¿sabes?, debemos usar un Objeto Nulo."
2. Concentran la atención en las fuerzas que se encuentran detrás del problema. Eso mite a los diseñadores [de casos de prueba] comprender mejor cuándo se aplica solución, y por qué.
3. Estimulan el razonamiento iterativo. Cada solución crea un nuevo contexto para ver nuevos problemas.



Aunque estos beneficios sean "leves", no deben perderse de vista. Buena parte de la prueba del software, incluso durante las últimas décadas, ha sido una actividad *ad hoc*. Si los patrones de prueba ayudan a un equipo de software a comunicarse de manera más efectiva, a comprender las fuerzas motivadoras que llevan a un enfoque específico de prueba y a considerar el diseño de los casos de prueba como una actividad en evolución, se habrá logrado mucho.

Los patrones de prueba se describen de manera muy parecida a los de análisis y diseño (capítulos 7 y 9) Se han propuesto docenas de patrones de prueba (por ejemplo, [BIN99], [MAR02]). Los siguientes tres patrones (presentados en forma resumida), proporcionan ejemplos representativos:

Nombre del patrón: **testigo par**

Resumen: Patrón orientado a procesos, **testigo par** describe una técnica análoga a la programación par (capítulo 4), en la cual dos responsables de una pruebas trabajan de manera conjunta para diseñar y ejecutar una serie de pruebas aplicables a actividades de prueba de unidad, integración o validación

Nombre del patrón **interfaz de prueba separada**

Resumen En sistemas orientados a objetos es necesario probar cada clase, incluidas las "clases internas" (es decir, las clases que no exponen ninguna interfaz fuera del componente que las utiliza). El patrón **interfaz de prueba separada** describe la manera de crear "una interfaz de prueba que permita describir pruebas específicas en clases que sólo son visibles internamente para un componente" [LAN01]

Nombre del patrón: **prueba de escenario**

Resumen: Una vez que se ha aplicado una prueba de unidad o de integración es necesario determinar si el software se comportará de manera tal que satisfaga al usuario El patrón **prueba de escenario** describe una técnica para ejercitar el software desde el punto de vista del usuario Una falla en este nivel indica que el software no satisface los requisitos de un usuario visible [KAN01]

Un análisis completo de los patrones de prueba está más allá del alcance de este libro. El lector interesado debe consultar [BIN99] y [MAR02] para conocer mayor información sobre este importante tema

14.12 RESUMEN

El objetivo principal del diseño de casos de prueba consiste en derivar un conjunto de pruebas que tenga la mayor probabilidad de descubrir errores en el software. Alcanzar este objetivo requiere emplear dos categorías diferentes de técnicas de diseño de casos de prueba (aplicables a sistemas convencionales y orientados a objetos): las pruebas de caja negra y de caja blanca

Las pruebas de caja blanca se concentran en la estructura de control del programa. Los casos de prueba se derivan para asegurar que todas las instrucciones del programa se ejecuten por lo menos una vez durante la prueba, y que todas las condiciones lógicas se ejerciten. La prueba de la ruta básica, una técnica de caja blan-

ca, aprovecha las gráficas del programa (o matrices de gráficas) para derivar un conjunto de pruebas linealmente independientes que aseguren una cobertura. La prueba de condición y de flujo de datos ejercitan aún más la lógica del programa y la prueba de bucle complementa otras técnicas de caja blanca al proporcionar un procedimiento para ejercitar bucles con grados diversos de complejidad.

Las pruebas de caja negra están diseñadas para validar requisitos funcionales sin importar el funcionamiento interno de un programa. Estas técnicas de prueba se concentran en el dominio de la información del software, derivando casos de prueba mediante partición de los dominios de entrada y salida de un programa en forma tal que proporcione cobertura completa. La partición equivalente divide el dominio de entrada en clases de datos que probablemente ejercitarán una función específica del software. El análisis de valores límite prueba la capacidad del programa para manejar datos en los límites de aceptabilidad. La prueba de tabla ortogonal proporciona un método eficiente y sistemático de probar sistemas con números pequeños de parámetros de entrada.

Aunque el objetivo general de la prueba orientada a objetos (encontrar el número máximo de errores con una cantidad mínima de esfuerzo) es idéntico al de la prueba del software convencional, la táctica para la prueba orientada a objetos difieren un poco. El concepto de prueba se ensancha para incluir la revisión del modelo de análisis y diseño. Además, el eje de la prueba se desplaza del componente procedimental (el módulo) hacia la clase. El diseño de pruebas para una clase aplica diversos métodos: prueba basada en fallas, aleatoria y de partición. Cada uno de estos métodos ejercita las operaciones encapsuladas por la clase. Las secuencias de prueba están diseñadas para asegurar que se ejerciten operaciones relevantes para examina el estado de la clase, representado por los valores de sus atributos, para determinar si existen errores.

La prueba de integración se realiza mediante una estrategia basada en el usuario. Este tipo de prueba construye el sistema en capas, empezando con las clases que no usan clase de servidor. Los métodos de diseño de casos de prueba de integración también pueden emplear pruebas aleatorias y de partición. Además, se utilizan pruebas basadas en el escenario y derivadas de modelos de comportamiento para probar una clase y sus colaboradores. Una secuencia de prueba da seguimiento al flujo de las operaciones entre las colaboraciones de clases.

Los métodos de prueba especializados abarcan una amplia serie de opciones de software y áreas de aplicación. La prueba para interfaces gráficas de usuario, de arquitecturas cliente/servidor, de la documentación y funciones de ayuda y de sistemas en tiempo real requieren directrices y técnicas especializadas.

Los desarrolladores de software con experiencia suelen decir: "La prueba nunca termina, sólo se transfiere del ingeniero del software al cliente. Cada vez que el cliente usa el programa, está realizando una prueba". Al aplicar el diseño de casos de prueba, el ingeniero de software logra pruebas más completas y, por tanto, descubre y corrige el mayor número de errores antes de que empiecen las "pruebas del cliente".

REFERENCIAS

[AMB95] Ambler, S., "Using Use Cases", en *Software Development*, julio de 1995, pp. 53-61

[BEI90] Beizer, B., *Software Testing Techniques*, segunda edición, Van Nostrand-Reinhold, 1990

[BEI95] Beizer, B., *Black-Box Testing*, Wiley, 1995

[BIN94] Binder, R. V., "Testing Object-Oriented Systems: A Status Report", en *American Programmer*; vol. 7, núm. 4, abril de 1994, pp. 23-28

[BIN99] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[DEU79] Deutsch, M., "Verification and Validation", en *Software Engineering* (R. Jensen y C. Tonies, eds.), Prentice-Hall, 1979, pp. 329-408

[FRA88] Frankl, P. G. y E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria", en *IEEE Trans. Software Engineering*, vol. SE-14, núm.10, octubre de 1988, pp. 1483-1498

[FRA93] Frankl, P. G. y S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow", en *IEEE Trans. Software Engineering*, vol. SE-19, núm.8, agosto de 1993, pp.770-787.

[KAN93] Kaner, C., J. Falk y H. Q. Nguyen, *Testing Computer Software*, segunda edición, Van Nostrand-Reinhold, 1993.

[KAN01] Kaner, C., "Pattern: Scenario Testing" (borrador), 2001, disponible en http://www.testing.com/test-patterns/patterns/pattern-scenario-testing-kaner.html.

[KIR94] Kirani, S. y W. T. Tsai, "Specification and Verification of Object-Oriented Programs", Technical Report TR 94-64, Computer Science Department, University of Minnesota, diciembre de 1994.

[LAN01] Lange, M., "It's Testing Time! Patterns for Testing Software", junio de 2001, disponible para descarga en http://www.testing.com/test-patterns/patterns/index.html

[LIN94] Lindland, O. I. *et al.*, "Understanding Quality in Conceptual Modeling", en *IEEE Software*, vol. 11, núm. 4, julio de 1994, pp. 42-49.

[MAR94] Marick, B., *The Craft of Software Testing*, Prentice-Hall, 1994.

[MAR02] Marick, B., "Software Testing Patterns", 2002, http://www.testing.com/test-patterns/index.html.

[MCC76] McCabe, T., "A Software Complexity Measure", en *IEEE Trans. Software Engineering*, vol. SE-2, diciembre de 1976, pp. 308-320.

[MGR94] McGregor, J. D. y T. D. Korson, "Integrated Object-Oriented Testing and Development Processes", *CACM*, vol. 37, núm. 9, septiembre de 1994, pp. 59-77.

[MUS93] Musa, J., "Operational Profiles in Software Reliability Engineering", en *IEEE Software*, marzo de 1993, pp. 14-32.

[MYE79] Myers, G., *The Art of Software Testing*, Wiley, 1979.

[NTA88] Ntafos, S. C., "A Comparison of Some Structural Testing Strategies", en *IEEE Trans. Software Engineering*, vol. SE-14, núm. 6, junio de 1988, pp. 868-874

[PHA89] Phadke, M. S., *Quality Engineering Using Robust Design*, Prentice-Hall, 1989

[PHA97] Phadke, M. S., "Planning Efficient Software Tests", *Crosstalk*, vol. 10, núm. 10, octubre de 1997, pp. 11-15

[TAI89] Tai, K. C., "What to Do Beyond Branch Testing", *ACM Software Engineering Notes*, vol. 14, núm. 2, abril de 1989, pp. 58-61.

PROBLEMAS Y PUNTOS A CONSIDERAR

14.1. Myers [MYE79] aplica el siguiente programa como autoevaluación de la capacidad propia para especificar pruebas adecuadas: un programa lee tres valores enteros. Los tres valores se interpretan como si representaran las longitudes de los lados de un triángulo. El programa imprime un mensaje que indica si el triángulo es escaleno, isósceles o equilátero. Desarróllese un conjunto de casos de prueba que se considere que prueben adecuadamente este programa.

14.2. Diseñar e implementar el programa (con manejo de errores cuando sea apropiado) especificado en el problema 14.1. Derivar una gráfica de flujo para el programa y aplicar la prueba de la ruta básica para desarrollar casos de prueba que garanticen que se han probado todas las instrucciones del programa. Ejecutar los casos y mostrar los resultados.

14.3. ¿Al lector le es posible pensar en alguna prueba adicional de características que n analizaron en la sección 14.7?

14.4. Seleccionar un componente de software que el lector haya diseñado e implementado cientemente. Diseñar un conjunto de casos de prueba que aseguren que todas las instrucc se hayan ejecutado con la prueba de la ruta básica.

14.5. Especificar, diseñar e implementar una herramienta de software que calcule la com dad ciclomática para el lenguaje de programación que se elija. Aplicar la matriz de gráfica mo estructura operativa de datos en el diseño.

14.6. Léase Beizer [BEI95] y determínese la manera en que el programa que se desarro el problema 14.5 puede extenderse para acomodar varios pesos de enlace. Extiéndase la h mienta para procesar probabilidades de ejecución o tiempos de procesamiento de enlaces

14.7. Diséñese una herramienta automatizada que reconozca bucles y los ordene en ca rías, como se indica en la sección 14.5.3.

14.8. Extiéndase la herramienta descrita en el problema 14.7 para generar casos de prueba r ra cada categoría de bucle, una vez encontrada. Será necesario desarrollar esta función de s nera interactiva con el encargado de la prueba

14.9. Ofrézcanse por lo menos tres ejemplos en que la prueba de caja negra daría la im de que "todo está bien", mientras que las pruebas de caja blanca descubrirían un error por lo menos tres ejemplos en que suceda lo contrario: la prueba de caja blanca daría la i sión de que "todo está bien", mientras que las pruebas de caja negra descubrirían un error

14.10. ¿La prueba exhaustiva (en caso de que sea posible en programas muy pequeños) tiza que el programa es totalmente correcto?

14.11. En palabras propias, descríbase por qué la clase es la menor unidad razonable prueba dentro de un sistema orientado a objetos

14.12. ¿Por qué se tienen que volver a probar subclases que crean instancias a partir clase existente si ésta ya se ha probado por completo? ¿Es posible usar los casos de pr señados para la clase existente?

14.13. Apliquense pruebas aleatorias y de partición a tres clases definidas en el diseño tema *HogarSeguro*. Produzcanse casos de prueba que indiquen las secuencias de opera se invocarán

14.14. Apliquense pruebas de clase múltiple y pruébense derivados del modelo de miento para el diseño *HogarSeguro*.

14.15. Pruébese un manual de usuario (o una función de ayuda) de una aplicación que con frecuencia. Encuéntrese por lo menos un error en la documentación

Otras lecturas y fuentes de información

Entre docenas de libros que presentan métodos de diseño de casos de prueba se Craig y Kaskiel (*Systematic Software Testing*, Artech House, 2002), Tamres (*Introducing Testing*, Addison-Wesley, 2002), Whittaker (*How to Break Software*, Addison-Wesley, gensen (*Software Testing. A Craftman's Approach*, CRC Press, 2002), Splaine y sus co *Web Testing Handbook*, Software Quality Engineering Publishing, 2001), Patton (*Softw* Sams Publishing, 2000), Kaner y sus colegas (*Testing Computer Software*, segunda ley, 1999). Además, Hutcheson (*Software Testing Methods and Metrics: The Most Imp* McGraw-Hill], 1997) y Marick (*The Craft of Software Testing: Subsystem Testing Including Based and Object-Oriented Testing*, Prentice-Hall, 1995) presentan tratamientos de mé trategias de prueba.

Myers [MYE79] sigue siendo un texto clásico, que analiza las técnicas de caja negra detalle. Beizer [BEI90] proporciona una amplia cobertura de las técnicas de caja bla

duce un nivel de rigor matemático que a menudo se omite en otros tratamientos de pruebas. Su último libro [BEI95] presenta un tratamiento conciso de métodos importantes. Perry (*Effective Methods for Software Testing*, Wiley-QED, 1995) y Friedman y Voas (*Software Assessment: Reliability, Safety, Testability*, Wiley, 1995) presentan buenas introducciones a las estrategias y tácticas de prueba. Mosley (*The Handbook of MIS Application Software Testing*, Prentice-Hall, 1993) analiza temas de prueba para sistemas de información grandes, y Marks (*Testing Very Big Systems*, McGraw-Hill, 1992) analiza los aspectos especiales que deben tomarse en cuenta cuando se prueban sistemas de programación importantes.

Sykes y McGregor (*Practical Guide for Testing Object-Oriented Software*, Addison-Wesley, 2001), Bashir y Goel (*Testing Object-Oriented Software*, Springer-Verlag, 2000), Binder (*Testing Object Oriented Systems*, Addison-Wesley, 1999), Kung y sus colegas (*Testing Object-Oriented Software*, IEEE Computer Society Press, 1998), Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) y Siegel y Muller (*Object-Oriented Software Testing: A Hierarchical Approach*, Wiley, 1996) presentan estrategias y métodos para probar sistemas orientados a objetos.

La prueba del software es una actividad que ocupa muchos recursos. Por ello, muchas organizaciones automatizan partes del proceso de prueba. Libros de Dustin, Rashka y Poston (*Automated Software Testing. Introduction, Management, and Performance*, Addison-Wesley, 1999), Graham y sus colegas (*Software Test Automation*, Addison-Wesley, 1999), y Poston (*Automating Specification-Based Software Testing*, IEEE Computer Society, 1996) analizan herramientas, estrategias y métodos para pruebas automatizadas.

Varios libros consideran los métodos y las estrategias de prueba en áreas de aplicación especializadas. Gardiner (*Testing Safety-Related Software: A Practical Handbook*, Springer-Verlag, 1999) ha editado un libro que aborda la prueba de sistemas de seguridad crítica. Mosley (*Client/Server Software Testing on the Desk Top and the Web*, Prentice-Hall, 1999) analiza el proceso de prueba para clientes, servidores y componentes de red. Rubin (*Handbook of Usability Testing*, Wiley, 1994) ha escrito una guía útil para quienes deben ejercitar interfaces humanas.

Binder [BIN99] describe casi 70 patrones de prueba que cubren métodos de prueba, clases/grupos, subsistemas, componentes reutilizables, marcos conceptuales y sistemas, además de automatización de pruebas y prueba de base de datos especializada. Una lista de estos patrones se encontrará en www.rbsc.com/pages/TestPatternList.htm.

Una amplia variedad de fuentes de información sobre los métodos de diseño de casos de prueba está disponible en Internet. Una lista actualizada de referencias en la World Wide Web que tienen relevancia para las técnicas de prueba se encuentra en el sitio Web SEPA **http://www.mhhe.com/pressman.**

CAPÍTULO

15 MÉTRICAS DEL PRODUCTO PARA EL SOFTWARE

CONCEPTOS CLAVE

La medición es un elemento clave en cualquier proceso de ingeniería. Las medidas se emplean para comprender mejor los atributos de los modelos que se crean y evaluar la calidad de los productos de la ingeniería o de los sistemas que se construyen. Pero a diferencia de otras disciplinas de la ingeniería, la del software no se basa en las leyes cuantitativas básicas de la física. Las medidas directas, como el voltaje, la masa, la velocidad o la temperatura no son comunes en el mundo del software. Debido a que las medidas y métricas del software suelen ser indirectas están expuestas al debate. Fenton [FEN91] analiza este tema cuando afirma:

> La medición es el proceso mediante el cual se asignan números o símbolos a los atributos de entidades reales para definirlas de acuerdo con reglas claramente establecidas. En las ciencias físicas, la medicina y, más recientemente, las ciencias sociales, ahora podemos medir atributos que se consideraban imposibles de medir. Por supuesto, estas mediciones no tienen el mismo refinamiento que casi todas las de las ciencias físicas... pero existen [y muchas decisiones importantes se toman con base en ellas]. Sentimos que la obligación de tratar de "medir lo inmedible" para mejorar nuestra comprensión de entidades particulares es tan importante en la ingeniería del software como en cualquier otra disciplina.

Pero algunos miembros de la comunidad de software siguen argumentando que el software "es inmedible", o que deben posponerse los intentos de medición hasta que se comprenda mejor el propio software y los atributos que deben usarse para describirlo. Esto es un error.

UN VISTAZO RÁPIDO

¿Qué es? Por su naturaleza, la ingeniería es una disciplina cuantitativa. Los ingenieros usan números como apoyo para el diseño y la evaluación del producto que construirán. Hasta hace poco, los ingenieros de software contaban con escasas guías cuantitativas en su trabajo, pero eso está cambiando. Las métricas del producto los ayudan a conocer mejor el diseño y la construcción del software que elaboran. A diferencia de las métricas del proceso y el proyecto que se aplican al proyecto (o al proceso) como un todo, las métricas del producto se concentran en atributos específicos de los productos del trabajo de la ingeniería del software y se recopilan a medida que se realizan las tareas técnicas del trabajo (análisis, diseño, codificación y prueba).

¿Quién lo hace? Los ingenieros de software usan las métricas del producto como apoyo para construir software de mayor calidad.

¿Por qué es importante? Siempre intervendrán elementos cualitativos en la creación del software. El problema es que no basta con la evaluación cualitativa. Un ingeniero de software

necesito criterios objetivos para orientar el diseño de los datos, la arquitectura, las interfaces y los componentes. El responsable de la prueba requiere una guía cuantitativa que le ayude a seleccionar los casos de prueba y sus objetivos. Las métricas del producto proporcionan una base para que el análisis, el diseño, la codificación y la prueba se realicen con mayor objetividad y se evalúen de manera más cuantitativa.

¿Cuáles son los pasos? El primer paso del proceso de medición consiste en derivar, a partir del software, las medidas y métricas apropiadas para la representación del software que se está considerando. A continuación se reúnen los datos para derivar las métricas formuladas. Una vez calculadas, se analizan las métricas apropiadas con base en directrices preestablecidas y en datos anteriores. Los resultados del análisis se interpretan para conocer más acerca del la calidad del software; los resultados desembocan en la modificación de los modelos de análisis y diseño, de código fuente o los casos de prueba. En algunos casos, tal vez se llegue a la modificación del propio proceso del software.

¿Cuál es el producto obtenido? Las métricas del producto que se calculan a partir de los datos recopilados de los modelos de análisis y diseño, de código fuente y casos de prueba.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Deben determinarse los objetivos de la medición antes de iniciar la recopilación de datos, definiendo cada métrica del producto sin ambigüedades. Definanse unas cuantas métricas y luego utilícense para reconocer la calidad de un producto de trabajo de la ingeniería del software.

Aunque las métricas del producto para el software de computadora no suelen ser absolutas, proporcionan una manera sistemática de evaluar la calidad a partir de un conjunto de reglas definidas con claridad. También proporcionan al ingeniero de software información inmediata y en el sitio; no posterior al hecho. Esto permite al ingeniero descubrir y corregir problemas potenciales antes de que se conviertan en defectos catastróficos.

En este capítulo se analizarán las mediciones con que se evalúa la calidad del producto mientras se diseña o construye. Estas medidas de atributos internos del producto proporcionan al ingeniero de software una indicación en tiempo real de la eficacia de los modelos de análisis, diseño y código; también aportan indicativos de la efectividad de los casos de prueba y la calidad general del software que habrá de construirse.

15.1 Calidad general

Hasta los desarrolladores de software exhaustos están de acuerdo en que es importante crear software de alta calidad. Pero, ¿cómo se define la calidad? En el sentido más amplio, calidad del software es el *cumplimiento de los requisitos de funcionalidad y desempeño explícitamente establecidos, de los estándares de desarrollo explícitamente documentados y de las características implícitas que se esperan de todo software desarrollado profesionalmente*.

Es indudable que esta definición podría modificarse o extenderse y debatirse interminablemente. En cuanto a los objetivos de este libro, la definición sirve para destacar tres puntos importantes:

1. Los requisitos del software son la base de las medidas de calidad. La falta de concordancia con estos requisitos es una falta de calidad.[1]
2. Los estándares especificados definen un conjunto de criterios de desarrollo que guían la ingeniería del software. Si no se siguen los criterios, el resultado será, casi seguramente, la falta de calidad
3. A menudo se soslaya un conjunto de requisitos implícitos (por ejemplo, el deseo de alcanzar la facilidad de uso). Si el software cumple con sus requisitos explícitos pero no con los implícitos, la calidad del software estará en duda.

La calidad del software es una compleja combinación de factores que variarán en las diferentes aplicaciones y los distintos clientes que las solicitan. En las siguientes secciones se identifican los factores que afectan la calidad del software y se describen las actividades humanas que deben desarrollarse para alcanzarla.

15.1.1 Factores de calidad de McCall

Los factores que afectan la calidad del software se dividen en dos grandes grupos: 1) los que se miden directamente (por ejemplo, defectos descubiertos durante la prueba), y 2) los que sólo se miden indirectamente (por ejemplo, facilidad de uso o mantenimiento). En cada caso debe presentarse una medición. Se debe comparar el software (programa, datos, documentos) con algún conjunto de datos y obtener así algún indicio sobre la calidad.

McCall, Richards y Walters [MCC77] propusieron una clasificación útil de los factores que afectan la calidad del software. Estos factores, que se muestran en la figura 15.1, se concentran en tres aspectos importantes de un producto de software: sus características operativas, su capacidad para experimentar cambios y su capacidad para adaptarse a nuevos entornos.

Si se toman como referencia los factores indicados en la figura 15.1, McCall y sus colegas proporcionan las siguientes descripciones:

FIGURA 15.1 Factores de la calidad del software de McCall.

1 Es importante indicar que la calidad se extiende a las características técnicas de los modelos de análisis y diseño, así como a la realización del código fuente de éstos. Modelos de alta calidad (en el sentido técnico) darán lugar a software de alta calidad, desde el punto de vista del cliente

PUNTO CLAVE

...e observar ... factores de ... e McCall son ... s hoy como ... en la década ... Por tanto, es ... afirmar que ... s que ... calidad del ... cambian ... tiempo.

Corrección. El grado en que el programa cumple con su especificación y satisface los objetivos que propuso el cliente

Confiabilidad. El grado en que se esperaría que un programa desempeñe su función con la precisión requerida. [Debe observarse que se han propuesto otras definiciones de confiabilidad más completas (consúltese el capítulo 26)]

Eficiencia. La cantidad de código y de recursos de cómputo necesarios para que un programa realice su función.

Integridad El grado de control sobre el acceso al software o los datos por parte de las personas no autorizadas.

Facilidad de uso. El esfuerzo necesario para aprender, operar y preparar los datos de entrada de un programa e interpretar la salida

Facilidad de mantenimiento El esfuerzo necesario para localizar y corregir un error en un programa (Una definición muy limitada.)

Flexibilidad El esfuerzo necesario para modificar un programa en operación

Facilidad de prueba El esfuerzo que demanda probar un programa con el fin de asegurar que realiza su función

Portabilidad El esfuerzo necesario para transferir el programa de un entorno de hardware o software a otro

Facilidad de reutilización. El grado en que un programa (o partes de él) puede reutilizarse en otras aplicaciones (en relación con el empaquetamiento y el alcance de las funciones que realiza el programa).

Interoperabilidad El esfuerzo necesario para acoplar un sistema con otro.

CONSEJO

...e una lista ... de verificación ... estos ... s. Primero ... a cada uno ... importancia ... para su ... Luego ... una gradua- ... para sus ... s de trabajo ... al fin de evaluar ... calidad del software ... se está constru-

> "La calidad de un producto es una función del bien que hace al mundo."
>
> **Tom DeMarco**

Es difícil, y en algunos casos imposible, desarrollar medidas directas[2] de estos factores de la calidad. En realidad, muchas de las métricas que definen McCall *et al.* sólo se miden subjetivamente. Es común que las métricas adquieran la forma de una lista de comprobación que se emplea para "asignar una graduación" a atributos específicos del software [CAV78]

15.1.2 Factores de calidad del estándar ISO 9126

El estándar ISO 9126 se desarrolló como un intento por identificar los atributos de calidad para el software de computadora. El estándar identifica seis atributos clave de la calidad:

2 Una *medida directa* indica que sólo es posible contar un valor que proporciona una indicación directa del atributo que se examina. Por ejemplo, el "tamaño" de un programa se mide directamente al contar el número de líneas de código.

Funcionalidad. El grado en que el software satisface las necesidades que in... los siguientes subatributos: idoneidad, exactitud, interoperabilidad, cumplimiento ... seguridad.

Confiabilidad. La cantidad de tiempo en que el software está disponible para ... lo según los siguientes subatributos: madurez, tolerancia a fallas y facilidad de re... peración.

Facilidad de uso. La facilidad con que se usa el software de acuerdo con los ... guientes subatributos: facilidad de comprensión, facilidad de aprendizaje y op... lidad.

Eficiencia. El grado en que el software emplea en forma óptima los recursos ... sistema, como lo indican los siguientes subatributos: comportamiento en el ti... comportamiento de los recursos.

Facilidad de mantenimiento. La facilidad con que se repara el software de a... con los siguientes subatributos: facilidad de análisis, facilidad de cambio, esta... y facilidad de prueba.

Portabilidad. La facilidad con que se lleva el software de un entorno a otro ... los siguientes subatributos: adaptabilidad, facilidad para instalarse, cumpli... facilidad para reemplazarse.

Al igual que otros factores de calidad del software analizados en el capítulo 9 ... sección 15.1.1, los factores ISO 9126 no necesariamente se prestan a la medición ... recta. Sin embargo, proporcionan una base valiosa para las medidas indirectas y ... lista de verificación excelente para evaluar la calidad de un sistema.

> "Cualquier actividad se vuelve creativo cuando la persona que hace las cosas las hace bien, o mejor."
>
> John Updike

15.1.3 La transición a un concepto cuantitativo

En las secciones anteriores se analizó un conjunto de factores cualitativos para ... "medición" de la calidad del software. El esfuerzo por desarrollar medidas preci... de la calidad del software en ocasiones se frustra por la naturaleza subjetiva ... actividad. Cavano y McCall [CAV78] analizan esta situación:

> La determinación de la calidad es un factor clave en todas las actividades diarias (concur... sos de cata de vinos, competencias deportivas [como la gimnasia], concursos de talen... etc.). En estas situaciones la calidad se juzga de la manera más básica y directa: compa... rando objetos que se encuentran juntos en condiciones idénticas y con conceptos predet... terminados. El vino se juzga de acuerdo con su claridad, color, bouquet, sabor, etc. Sin ... embargo, este tipo de juicios es muy subjetivo; para que tenga algún valor debe hacerlo ... un experto.
>
> La subjetividad y la especialización también se aplican para determinar la calidad de ... software. Se necesita una definición más precisa de la calidad del software para resolver ...

este problema, además de un medio para derivar medidas cuantitativas, a partir de la calidad del software, para realizar un análisis objetivo

En las secciones siguientes se examina un conjunto de métricas que se aplican en la evaluación cuantitativa de la calidad del software. En todos los casos las métricas representan medidas indirectas; es decir, nunca se mide realmente la calidad, sino alguna manifestación de ésta El factor que complica las cosas es la relación precisa entre la variable que se mide y la calidad del software.

> "Así como la medición de la temperatura empieza con un dedo índice... y da lugar a escalas, herramientas y técnicas sofisticadas, así sucede también con la madurez en la medición del software."
>
> **Shari Pfleeger**

15.2 UN MARCO CONCEPTUAL PARA LAS MÉTRICAS DEL PRODUCTO

Como se indicó en la introducción de este capítulo, la medición asigna números o símbolos a atributos de entidades reales. Esto requiere un modelo de medición que abarque un conjunto consistente de reglas. Aunque la teoría de la medición (por ejemplo, [KYB84]) y su aplicación al software de computadora (por ejemplo, [DEM81], [BRI96], [ZUS97]) son temas que rebasan el alcance de este libro, vale la pena establecer un marco conceptual y un conjunto de principios básicos para la medición de métricas para el producto de software.

15.2.1 Medidas, métricas e indicadores

Aunque *medida*, *medición* y *métrica* son términos que suelen utilizarse de manera intercambiable, es importante observar las sutiles diferencias entre ellos. En el contexto de la ingeniería del software una medida proporciona una indicación cuantitativa de la extensión, la cantidad, la dimensión, la capacidad o el tamaño de algún atributo de un producto o proceso. *Medición* es el acto de determinar una medida. El *Glosario de estándares del IEEE* [IEE93] define *métrica* como una "medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo determinado"

Cuando se ha recopilado un solo tipo de datos (por ejemplo, el número de errores descubiertos dentro de un solo componente del software) se ha establecido una medida La medición ocurre como resultado de la recopilación de uno o más puntos de datos (por ejemplo, se investigan varias revisiones de componentes y pruebas de unidad para reunir medidas del número de errores encontrados en cada uno) Una métrica de software relaciona de alguna manera las medidas individuales (por ejemplo, el número promedio de errores encontrados en cada revisión o prueba de unidad).

Un ingeniero de software recopila medidas y desarrolla métricas para obtener los indicadores. Un *indicador* es una métrica o una combinación de métricas que proporcionan conocimientos acerca del proceso del software, un proyecto de software o el propio producto. Un indicador proporciona conocimientos que permiten al jefe

de proyecto o los ingenieros de software ajustar el proceso, el proyecto o el prod... to para que las cosas mejoren.

> "Una ciencia tiene la misma madurez que sus herramientas de medición."
>
> Luis P...

15.2.2 El reto de las métricas del producto

En las últimas tres décadas, muchos investigadores han tratado de desarrolla... sola métrica que proporcione una medida completa de la complejidad del soft... Fenton [FEN94] caracteriza esta investigación como una búsqueda del "santo ... imposible". Aunque se han propuesto docenas de medidas de complejidad [ZUS... cada una tiene un concepto diferente de la complejidad y de los atributos de un ma que llevan a la complejidad. Por analogía, considérese una métrica para evalu... automóvil atractivo. Algunos observadores destacarían el diseño de la carrocería ... tomarían en cuenta características mecánicas y otros más podrían considerar el ... el desempeño, la economía de combustible o la capacidad de reciclarlo cuando ... to sea inservible. Como cualquiera de estas características estaría en desventaja ... otras, resulta difícil derivar un solo valor de "atractivo". El mismo problema ... con el software de computadora.

Referencia Web

Horst Zuse ha recopilado gran cantidad de información sobre métricas de producto en **irb.cs.tu-berlin.de/~zuse/**.

Pero existe la necesidad de medir y controlar la complejidad del software ... difícil derivar un solo valor de esta métrica de calidad, debe tenerse la posibi... desarrollar medidas de diferentes atributos internos del programa (por ejemplo: ... dularidad efectiva, independencia funcional y otros atributos analizados del capí... al 12). Estas medidas y las métricas derivadas de ellas se utilizan como indicad... dependientes de la calidad de los modelos de análisis y diseño. Pero aquí tamb... gen problemas. Fenton [FEN94] observa esto cuando afirma: "El peligro de tra... encontrar medidas que caractericen tantos atributos diferentes es que inevitab... te las medidas tienen que satisfacer objetivos que entran en conflicto entre sí ... opone a la teoría de que cada medición debe ser representativa." Aunque la ... ción de Fenton es correcta, muchas personas argumentan que la medición del ... realizada durante las primeras etapas del proceso del software proporciona a ... genieros un mecanismo consistente y objetivo para evaluar la calidad.

Sin embargo, vale la pena preguntarse sobre la validez de las métricas ... ducto. Es decir, ¿qué tanto concuerdan las métricas del producto, la confiab... largo plazo y la calidad de un sistema de cómputo? Fenton [FEN91] aborda e... quietud de la siguiente manera:

> A pesar de las conexiones intuitivas entre la estructura interna de los productos de sof... ware [métricas del producto] y los atributos de su producto y su proceso externos ... lidad se han realizado muy pocos intentos científicos por establecer relaciones especí... Esto es así por varias razones· la que se cita con más frecuencia es que resulta poco ... tico realizar experimentos relevantes.

Cada uno de los "retos" indicados aquí debe tomarse con cautela, pero no hay razón para restarle méritos a las métricas técnicas [3] La medición es esencial si se desea alcanzar la calidad.

15.2.3 Principios de medición

Antes de introducir una serie de métricas del producto que 1) ayuden a evaluar los modelos de análisis y diseño, 2) ofrezcan una indicación de la complejidad de los diseños procedimentales y el código fuente, y 3) faciliten el diseño de pruebas más efectivas, es importante comprender los principios básicos de la medición Roche [ROC94] sugiere un proceso de medición al que caracterizan cinco actividades:

¿Cuáles son los pasos de un proceso de medición efectivo?

- *Formulación*. La derivación de medidas y métricas apropiadas para la representación del software que se considera.
- *Recolección* El mecanismo con que se acumulan los datos necesarios para derivar las métricas formuladas.
- *Análisis* El cálculo de las métricas y la aplicación de herramientas matemáticas.
- *Interpretación*. La evaluación de las métricas en un esfuerzo por conocer mejor la calidad de la representación.
- *Retroalimentación*. Recomendaciones derivadas de la interpretación de las métricas del producto transmitidas al equipo del software.

Las métricas del software sólo serán útiles si están caracterizadas de manera efectiva y se validan para probar su valor. Los siguientes principios [LET03] son representativos de muchos otros que podrían proponerse para caracterizar y validar las métricas:

CONSEJO

Si bien la utilidad, muchas métricas de producto vigentes en la actualidad no cumplen con estos principios como deberían hacerlo. Para nosotros significa que carecen de valor; sólo debe tenerse cuidado al usarlas, y comprender que su utilidad es probablemente considerable, no una comprobación científica sólida.

- *Una métrica debe tener propiedades matemáticas deseables*. Es decir, el valor de la métrica debe estar en un rango significativo (por ejemplo, de cero a uno, donde cero realmente significa ausencia, uno indica el valor máximo y 0.5 representa el "punto medio"). Además, una métrica que pretende estar en una escala racional no debe contar con componentes que sólo se miden en una escala ordinal.
- *Cuando una métrica representa una característica de software que aumenta cuando se presentan rasgos positivos o que disminuye al encontrar rasgos indeseables, el valor de la métrica debe aumentar o disminuir en el mismo sentido.*
- *Cada métrica debe validarse empíricamente en una amplia variedad de contextos antes de publicarse o aplicarse en la toma de decisiones*. Una métrica debe medir el factor de interés, independientemente de otros factores. Debe "crecer" para aplicarse en sistemas grandes y funcionar en diversos lenguajes de programación y dominios de sistemas.

3 Aunque la crítica de métricas específicas es común en la bibliografía, muchas de esas críticas se concentran en aspectos esotéricos y pierden de vista el principal objetivo de las métricas en la realidad: ayudar al ingeniero de software a establecer una manera sistemática y objetiva de conocer a fondo su trabajo y, como resultado, mejorar la calidad del producto

Aunque la formulación, caracterización y validación son críticas, la recopilación y el análisis son las actividades que dirigen el proceso de medición. Roche [ROC94] sugiere las siguientes directrices para estas actividades: 1) siempre que sea posible deben automatizarse la recopilación de datos y su análisis; 2) deben aplicarse técnicas estadísticas válidas para establecer relaciones entre los atributos internos del producto y las características de calidad externas (por ejemplo, si el grado de complejidad de la arquitectura se correlaciona con el número de defectos informados al aplicarla en producción), y 3) para cada métrica deben establecerse directrices y recomendaciones para la interpretación.

15.2.4 Medición del software orientado a objetivos

Referencia Web

Información sobre OPM se encontrará en www.deadcs.com/GoldPractices/practices/gqm.html.

Basili y Weiss [BAS84] desarrollaron el paradigma *objetivo/pregunta/métrica* (OPM) como una técnica para identificar métricas significativas aplicables en cualquier parte del proceso del software. El OPM destaca la necesidad de 1) establecer un *objetivo* de medición explícito que sea específico para la actividad del proceso o las características del producto que se está evaluando, 2) definir un conjunto de *preguntas* que deben responderse con el fin de alcanzar el objeto, y 3) identificar *métricas* bien formuladas que ayuden a responder esas preguntas.

Es posible emplear una *plantilla de definición de objetivos* [BAS94] para definir cada objetivo de medición. La plantilla toma esta forma:

> **Analizar** [el nombre de la actividad o el atributo que se medirá] **con el propósito de** [el objetivo general del análisis[4]] **en relación con** [el aspecto de la actividad o atributo que se considera] **desde el punto de vista de** [la gente que tiene interés en la medición] **en el contexto de** [el entorno en que tiene lugar la medición]

Como ejemplo, considérese una plantilla de definición del objetivo para *HogarSeguro*:

> **Analizar** la arquitectura del software *HogarSeguro* **con el propósito de** evaluar los componentes arquitectónicos **en relación con** la capacidad para lograr que *HogarSeguro* sea más extensible **desde el punto de vista de** los ingenieros de software que realizan el trabajo **en el contexto de** la mejora del producto en los siguientes tres años.

Una vez definido explícitamente el objetivo de la medición, se desarrolla un conjunto de preguntas. Las respuestas a éstas ayudan al equipo de software (y a otros participantes) a determinar si se ha alcanzado el objetivo de medición. Algunas de las preguntas que se responden están las siguientes:

P_1: ¿Los componentes arquitectónicos están caracterizados de manera que aparecen por separado la función y los datos relacionados?

4 Van Solingen y Berghout [SOL99] sugieren que el objetivo es casi siempre "comprender y/o mejorar" la actividad del proceso o el atributo del producto.

P_2: ¿La complejidad de cada componente se encuentra dentro de los límites que facilitarán su modificación y extensión?

Cada una de estas preguntas debe responderse de manera cuantitativa, empleando una o más medidas y métricas. Por ejemplo, una métrica que proporciona una indicación de la cohesión (capítulo 9) de un componente arquitectónico sería útil para responder P_1. La complejidad ciclomática y las métricas analizadas en la sección 15.4.1 o 15.4.2 proporcionarían conocimientos a fondo para P_2.

En realidad, tal vez haya diversos objetivos de medición con preguntas y métricas relacionadas. En cualquier caso, las métricas elegidas (o derivadas) deben cumplir con los principios de medición analizados en la sección 15.2.3 y los atributos de medición analizados en la sección 15.2.5. Si se desea obtener mayor información sobre OPM, el lector interesado debe consultar [SHE98] o [SOL99].

15.2.5 Los atributos de las métricas efectivas del software

Se han propuesto cientos de métricas para el software de computadora, pero no todas proporcionan soporte práctico para el ingeniero de software. Algunas exigen mediciones demasiado complejas; otras son tan esotéricas que pocos profesionales podrían comprenderlas, y otras más violan las nociones intuitivas básicas de lo que es el software de alta calidad.

Ejiogu [EJI91] define un conjunto de atributos que toda métrica efectiva del software debe abarcar. La métrica derivada y las medidas que llevan a ella deben ser:

- *Simples y calculables*. Debe ser relativamente fácil aprender a derivar la métrica, y su cálculo no debe exigir cantidades anormales de tiempo o esfuerzo
- *Empírica e intuitivamente persuasivas*. La métrica debe satisfacer las nociones intuitivas del ingeniero acerca del atributo del producto que se está construyendo.
- *Consistentes y objetivas*. La métrica siempre debe arrojar resultados que no permitan ambigüedad alguna.
- *Consistentes en el uso de unidades y dimensiones*. El cálculo matemático de la métrica debe emplear medidas que no lleven a combinaciones extrañas de unidades
- *Independientes del lenguaje de programación*. Las métricas deben basarse en el modelo de análisis o diseño, o en la estructura del propio programa
- *Mecanismos efectivos para la retroalimentación de alta calidad*. Es decir, la métrica debe llevar a un producto final de la más alta calidad.

¿Cómo debemos evaluar la calidad de una métrica propuesta del software?

CONSEJO

La experiencia indica que sólo se usará una métrica del producto si es intuitiva y fácil calcular. Si deben hacerse docenas de "cuentas", es improbable que la métrica se adopte ampliamente

Aunque casi todas las métricas de software satisfacen estos atributos, algunas métricas de uso común no cumplen con una o dos de ellas. Un ejemplo es el punto de función (el cual se estudia en la sección 15.3.1), que es una medida de la entrega de "funcionalidad" por parte del software. Se puede argumentar que el atributo *consis-*

tente y objetiva falla porque tal vez un tercero, que sea independiente, no logre den el mismo valor del punto de función que un colega que utilice la misma información acerca del software. ¿Debemos rechazar entonces la medida de punto de función respuesta es ¡por supuesto que no! El punto de función proporciona conocimien útiles y, por tanto, valores distintos, aunque no satisfaga algún atributo a la perfecci

15.2.6 Panorama de las métricas del producto

Aunque se ha propuesto una amplia variedad de taxonomías métricas, el siguie esquema atiende las áreas más importantes de las métricas:

Métricas para el modelo de análisis. Estas métricas atienden varios aspectos modelo de análisis e incluyen:

Funcionalidad entregada: proporciona una medida indirecta de la funcion que se empaqueta con el software.

Tamaño del sistema: mide el tamaño general del sistema, definido desde el de vista de la información disponible como parte del modelo de análisis.

Calidad de la especificación: proporciona una indicación de la especificidad grado en que se ha completado la especificación de los requisitos.

Métricas para el modelo de diseño. Estas métricas cuantifican los atribut diseño de manera tal que le permiten al ingeniero de software evaluar la cal diseño. La métrica incluye:

Métricas arquitectónicas: proporcionan un indicio de la calidad del diseño a tónico.

Métricas al nivel de componente: miden la complejidad de los componer software y otras características que impactan la calidad.

Métricas de diseño de la interfaz: se concentran principalmente en la facilidad de

Métricas especializadas en diseño orientado a objetos: miden características ses, además de las correspondientes a comunicación y colaboración.

Métricas para el código fuente. Estas métricas miden el código fuente y se para evaluar su complejidad, además de la facilidad con que se mantiene y entre otras características:

Métricas de Halstead: controversiales pero fascinantes, estas métricas p nan medidas únicas de un programa de cómputo.

Métricas de complejidad: miden la complejidad lógica del código fuente se consideran métricas de diseño al nivel de componentes).

Métricas de longitud: proporcionan un indicio del tamaño del software.

6 Podría usarse un contraargumento igualmente vigoroso. Ésa es la naturaleza de las software

Métricas para pruebas. Estas métricas ayudan a diseñar casos de prueba efectivos y a evaluar la eficacia de las pruebas:

Métricas de cobertura de instrucciones y ramas: lleva al diseño de casos de prueba que proporcionan cobertura del programa

Métricas relacionadas con los defectos: se concentran en encontrar defectos y no en las propias pruebas.

Efectividad de la prueba: proporcionan un indicio en tiempo real de la efectividad de las pruebas aplicadas.

Métricas en el proceso: métricas relacionadas con el proceso que se determinan a medida que se aplican las pruebas

En muchos casos las métricas de un modelo pueden aplicarse en actividades posteriores de la ingeniería del software. Por ejemplo, las métricas de diseño se utilizan para estimar el esfuerzo requerido para generar código fuente. Además, las métricas de diseño se aprovechan para planear pruebas y el diseño de casos de prueba.

HogarSeguro

Debate sobre métricas del producto

El escenario: Cubículo de Vinod

Los actores: Vinod, Jamie y Ed, integrantes del equipo de ingeniería del software de *HogarSeguro*, que siguen trabajando con el diseño al nivel de componentes y de casos de prueba.

La conversación:

Vinod: Doug [Doug Miller, jefe de ingeniería del software] me dijo que todos deberíamos usar métricas del producto, pero lo hizo de manera vaga. También dijo que no presionaría .. su uso dependería de nosotros.

Jamie: Eso está bien, porque no tenemos tiempo para empezar a medir cosas. Estamos esforzándonos por cumplir con el calendario de trabajo

Ed: Estoy de acuerdo con Jamie. Estamos contra ellas, aquí . no hay tiempo

Vinod: Sí, lo sé, pero tal vez por alguna razón sea importante que las usemos

Jamie: No discuto eso, Vinod. Es cosa de tiempo... y yo no tengo tanto como para desperdiciarlo

Vinod: ¿Y si las mediciones nos ahorraran tiempo?

Ed: Estás equivocado. Requieren tiempo y, como dice Jamie...

Vinod: No, espera . ¿y si nos ahorran tiempo?

Jamie: ¿Cómo?

Vinod: Evitando retrabajar . Si una métrica nos ayuda a evitar un problema importante o incluso uno moderado, y eso nos evita retrabajar una parte del sistema, ahorraremos tiempo, ¿o no?

Ed: Es posible, supongo, pero ¿nos garantizas que alguna métrica del producto nos ayudará a encontrar un problema?

Vinod: ¿Y tú me garantizas que no lo hará?

Jamie: Bueno, ¿qué estás proponiendo?

Vinod: Creo que debemos seleccionar unas cuantas métricas de diseño, tal vez orientadas a clases, y aplicarlas como parte de nuestro proceso de revisión para todos los componentes que desarrollemos.

Ed: No estoy muy familiarizado con las métricas orientadas a objetos.

Vinod: Voy a dedicar un poco de tiempo a revisarlas y a hacer algunas recomendaciones ¿están de acuerdo?

(Ed y Jamie asienten sin mucho entusiasmo.)

15.3 MÉTRICAS PARA EL MODELO DE ANÁLISIS

Aunque existen en la bibliografía relativamente pocas métricas de análisis y especificación, es posible adaptar métricas que se utilizan en la estimación de proyectos y aplicarlas en este contexto. Estas métricas examinan el modelo de análisis con la intención de predecir el "tamaño" del sistema resultante. El tamaño es, en ocasiones (pero no siempre), un indicador de la complejidad del diseño y casi siempre resulta un indicador de un mayor esfuerzo de codificación, integración y prueba.

15.3.1 Métricas basadas en la función

Referencia Web

En los siguientes sitios Web se encontrará información abundante y útil acerca de los puntos de función: www.ifpug.org y www.functionpoints.com.

La métrica de punto de función (PF), propuesta inicialmente por Albretch [ALB79], se usa de manera efectiva como medio para medir la funcionalidad que entrega un sistema.[6] Empleando datos históricos, el PF se usa para 1) estimar el costo o el esfuerzo requerido para diseñar, codificar y probar el software; 2) predecir el número de errores que se encontrarán durante la prueba, y 3) pronosticar el número de componentes, de líneas de código proyectadas, o ambas, en el sistema implementado.

Los puntos de función se derivan empleando una relación empírica basada en medidas contables (directas) del dominio de la información del software y las evaluaciones de la complejidad de éste. Los valores del dominio de la información se definen de la siguiente manera [7]

Número de entradas externas (EE). Cada *entrada externa* se origina en el usuario o es transmitida desde otra aplicación y proporciona distintos datos orientados a la aplicación o información de control. Las entradas suelen emplearse para actualizar archivos lógicos internos (ALI). Las entradas deben distinguirse de las consultas, que se cuentan por separado.

Número de salidas externas (SE). Cada *salida externa* se deriva en el sistema de la aplicación y proporciona información al usuario. En este contexto, salida externa alude a informes, pantallas, mensajes de error, etc. Los elementos de datos individuales dentro de un informe no se cuentan por separado.

Número de consultas externas (CE). Una *consulta externa* se define como una entrada en línea que lleva a la generación de alguna respuesta inmediata por parte del software, en la forma de salida en línea (a menudo recuperada de un ALI).

Número de archivos lógicos internos (ALI). Cada *archivo lógico interno* es un agrupamiento lógico de datos que reside dentro de los límites de las aplicaciones y que se mantiene mediante entradas externas.

---

6 Desde que Albrecht dio a conocer su trabajo original, se han escrito cientos de libros, artículos sobre PF. En [IEP03] se encontrará una bibliografía muy valiosa.

7 En realidad, la definición de los valores del dominio de la información y la manera en que se cuentan son un poco más complejas. El lector interesado deberá consultar [IFP01] para conocer los detalles.

FIGURA 15.2
Cálculo de puntos de función.

Valor del dominio de información	Conteo		Factor de ponderación Simple	Promedio	Complejo	
Entradas externas (EE)	[ ]	×	3	4	6	= [ ]
Salidas externas (SE)	[ ]	×	4	5	7	= [ ]
Consultas externas (CE)	[ ]	×	3	4	6	= [ ]
Archivos de lógica interna (ALI)	[ ]	×	7	10	15	= [ ]
Archivos de interfaz externa (AIE)	[ ]	×	5	7	10	= [ ]
Total de conteos						[ ]

Número de archivos de interfaz externos (AIE). Cada *archivo de interfaz externo* es un agrupamiento lógico de datos externo a la aplicación pero que proporciona datos que podrían usarse en ésta.

Una vez que se han recolectado los datos, se completa la tabla de la figura 15.2 y se asocia un valor de complejidad con cada conteo. Las organizaciones que usan métodos de punto de función desarrollan criterios para determinar si una entrada determinada es simple, promedio o compleja. No obstante, la determinación de la complejidad es un poco subjetiva.

Para calcular los puntos de función (PF) se usa la siguiente relación:

$$PF = \text{conteo total} \times [0.65 + 0.01 \times \Sigma (F_i)] \qquad (15.1)$$

donde conteo total es la suma de todas las entradas de PF obtenidas de la figura 15.2.

F_i (i = 1 a 14) son *factores de ajuste de valor* basados en las respuestas a las siguientes preguntas [LON02]:

1. ¿El sistema requiere respaldo y recuperación confiables?
2. ¿Se requieren comunicaciones de datos especializadas para transferir información a la aplicación, u obtenerla de ella?
3. ¿Hay funciones distribuidas de procesamiento?
4. ¿El desempeño es crítico?
5. ¿El sistema se ejecutará en un entorno existente que tiene un uso pesado de operaciones?
6. ¿El sistema requiere entrada de datos en línea?
7. ¿La entrada de datos en línea requiere que la transacción de entrada se construya en varias pantallas u operaciones?
8. ¿Los ALI se actualizan en línea?
9. ¿Las entradas, las salidas, los archivos o las consultas son complejos?
10. ¿Es complejo el procesamiento interno?
11. ¿El código diseñado será reutilizable?

12. ¿Se incluyen la conversión e instalación en el diseño?
13. ¿Está diseñado el sistema para instalaciones múltiples en diferentes organizaciones?
14. ¿La aplicación está diseñada para facilitar el cambio y para que el usuario lo use fácilmente?

Cada una de estas preguntas se responde empleando una escala que va de 0 (no importante o aplicable) a 5 (absolutamente esencial). Los valores constantes de la ecuación (15.1) y los factores de peso que se aplican a los conteos del dominio de la información se determinan empíricamente.

Referencia Web
Una calculadora de PF en línea se encuentra en **irb.cs.uni-magdeburg.de/sw-eng/us/java/lp/.**

Para ilustrar el empleo de la métrica del PF en este contexto se ideó la representación simple del modelo de análisis, que se muestra en la figura 15.3. Ahí se representa un diagrama de flujo de datos (capítulo 8) dentro del software *HogarSeguro*. La función maneja la interacción con el usuario aceptando una contraseña de éste para activar o desactivar el sistema, y permite consultas sobre el estado de las zonas de seguridad y varios sensores de seguridad. La función despliega una serie de mensajes y envía señales de control apropiadas a varios componentes del sistema de seguridad.

Se evalúa el diagrama de flujo de datos para determinar un conjunto de medidas clave del dominio de información que se requieren para calcular la métrica del punto de función. En la figura se muestran tres entradas externas **(contraseña, botón de pánico y activar/desactivar)** junto con dos consultas externas **(consulta de zona y consulta de sensor)**. Se muestra un ALI **(archivo de configuración del sistema)**. También están presentes dos salidas de usuarios (**mensajes y estatus del sensor**) y cuatro AIE **(sensor de prueba, configuración de zona, activar/desactivar y alerta de alarma)**. En la figura 15.4 se muestran estos datos, junto con la complejidad apropiada.

El conteo total que se muestra en la figura 15.4 debe ajustarse empleando la ecuación (15.1):

$$PF = \text{conteo total} \times [0.65 + 0.01 \times \Sigma (F_i)]$$

FIGURA 15.3
Modelo de flujo de datos para el software *HogarSeguro*.

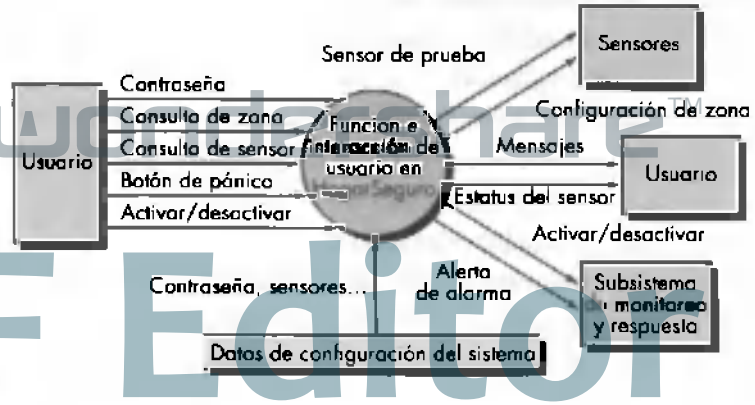


Figura 15.4 Cálculo de puntos de función.

Valor del dominio de información	Conteo		Factor de ponderación Simple	Promedio	Complejo		
Entradas externas (EE)	3	×	3	4	6	=	9
Salidas externas (SE)	2	×	4	5	7	=	8
Consultas externas (CE)	2	×	3	4	6	=	6
Archivos de lógica interna (ALI)	1	×	7	10	15	=	7
Archivos de interfaz externa (AIE)	4	×	5	7	10	=	20
Total de conteos							50

donde conteo total es la suma de todas las entradas de PF obtenidas de la figura 15.4, y F_i (i = 1 a 14) son factores de ajuste de valor. Para los objetivos de este ejemplo, supóngase que $\Sigma (F_i)$ es 46 (un producto moderadamente complejo). Por tanto:

$$PF = 50 \times [0.65 + (0.01 \times 46)] = 56$$

Con base en el valor proyectado del PF derivado del modelo de análisis, el equipo del proyecto puede estimar el tamaño implementado general de la función de interacción del usuario de *HogarSeguro*. Supóngase que los datos del pasado indican que un PF se traduce a 60 líneas de código (se va a usar un lenguaje orientado a objetos) y que se producen 12 PF por cada persona-mes de esfuerzo. Estos datos históricos proporcionan al jefe del proyecto información importante que sirve para la planeación y que se basa en el modelo de análisis más que en estimados preliminares. Supóngase, además, que los proyectos anteriores han encontrado un promedio de tres errores por punto de función durante las revisiones del análisis y el diseño, y de cuatro errores por punto de función durante las pruebas de unidad e integración. Estos datos ayudarán a los ingenieros de software a evaluar el grado en el que han completado sus actividades de revisión y prueba.

Uemura y sus colegas [UEM99] sugieren que los puntos de función también pueden calcularse a partir de diagramas UML de clase y secuencia (capítulos 8 y 10). El lector interesado debe consultar [UEM99] para conocer más detalles.

> "En lugar de sólo musitar acerca de cuál 'nueva métrica' podría aplicarse ... debemos plantearnos la pregunta básica: '¿qué haremos con las métricas?'"
>
> **Michael Mah y Larry Putnam**

15.3.2 Métricas para la calidad de la especificación

Davis y sus colegas [DAV93] proponen una lista de características con las cuales puede evaluarse la calidad del modelo de análisis y la correspondiente especificación

de requisitos: *especificidad* (falta de ambigüedad), *grado de avance*, *corrección*, *facilidad de comprensión*, *facilidad de verificación*, *consistencia interna* y *externa*, *facilidad para alcanzar los objetivos*, *concisión*, *facilidad para darle seguimiento*, *facilidad para modificarse*, *precisión* y *facilidad de reutilización*. Además, los autores [DAV93] observan que las especificaciones de alta calidad deben estar almacenadas electrónicamente, ser ejecutables o por lo menos interpretables, estar anotadas por importancia relativa, ser estables, tener indicada la versión, estar organizadas, incluir referencias cruzadas y especificarse con el grado de detalle correcto.

PUNTO CLAVE

Al medir las características de la especificación es posible obtener un conocimiento cuantitativo de la especificidad y el grado de avance

Aunque, al parecer, muchas de estas características tienen una naturaleza cualitativa, Davis *et al.* [DAV93] sugieren que cada una puede representarse empleando una o más métricas. Por ejemplo, supóngase que hay n_r requisitos en una especificación, de modo que

$$n_r = n_f + n_{nf}$$

donde n_f es el número de requisitos funcionales y n_{nf} el de no funcionales (como el desempeño).

Para determinar la especificidad (falta de ambigüedad) de los requisitos, Davis *et al.* sugieren una métrica basada en la consistencia de la interpretación de los revisores de cada requisito:

$$Q_1 = n_{ui}/n_r$$

donde n_{ui} es el número de requisitos que todos los revisores interpretaron de la misma manera. Cuanto más cercano esté el valor de Q a 1, menor será la ambigüedad de la especificación

El *grado de avance* de los requisitos funcionales se determina al calcular la relación

$$Q_2 = n_u/[n_i \times n_s]$$

donde n_u es el número de requisitos de función única; n_i, el número de entradas (estímulos) definidos o implícitos en la especificación, y n_s, el número de estados especificados. La relación Q_2 mide el porcentaje de funciones necesarias que se han especificado para un sistema. Sin embargo, no se atienden requisitos que no son funcionales. Para incorporarlos a una métrica general del grado de avance, se debe considerar el grado de validación de los requisitos:

$$Q_3 = n_c/[n_c + n_{nv}]$$

donde n_c es el número de requisitos que se han validado como correctos, y n_{nv} los requisitos que aún no se validan.

"Mida lo que sea mensurable, y lo que no lo sea, vuélvalo mensurable."

15.4 MÉTRICAS PARA EL MODELO DE DISEÑO

Sería inconcebible que el diseño de un nuevo avión, un nuevo chip de computadora o un nuevo edificio de oficinas se realizara sin definir las medidas del diseño, sin determinar las métricas de diversos aspectos de la calidad del diseño y sin usarlas para guiar la manera en que evoluciona el diseño. Sin embargo, a menudo el diseño de sistemas de software complejos suele avanzar casi sin medición. La ironía es que se dispone de métricas de diseño para el software, pero la gran mayoría de los desarrolladores siguen ignorando su existencia.

Las métricas de diseño para el software de computadora, como todas las demás métricas del software, no son perfectas. Sigue abierto el debate sobre su eficacia y la manera en que deben aplicarse. Muchos expertos argumentan que se necesita más experimentación antes de emplear las mediciones en el diseño. Sin embargo, un diseño sin medición es inaceptable.

15.4.1 Métricas del diseño arquitectónico

Las métricas de diseño arquitectónico se concentran en las características de la arquitectura del programa (capítulo 10), y se destacan la estructura arquitectónica y la efectividad de módulos o componentes dentro de la arquitectura. Estas métricas son de "caja negra", en el sentido de que no requieren ningún conocimiento del funcionamiento interno de un componente de software en particular.

Card y Glass [CAR90] definen tres medidas de la complejidad del diseño del software: estructural, de datos y del sistema.

En el caso de las arquitecturas jerárquicas (por ejemplo, las arquitecturas de llamada y retorno), la *complejidad estructural* de un módulo i se define de la siguiente manera:

$$S(i) = f^2_{out}(i) \tag{15.2}$$

donde $f_{out}(i)$ es la dependencia hacia fuera[8] del módulo i.

La *complejidad de datos* proporciona una indicación de la complejidad de la interfaze interna de un módulo i y se define como:

$$D(i) = v(i)/[f_{out}(i) + 1] \tag{15.3}$$

donde $v(i)$ es el número de variables de entrada y salida que se pasan al módulo i o se reciben de éste.

Por último, *complejidad del sistema* se define como la suma de las complejidades estructural y de datos, especificada como:

$$C(i) = S(i) + D(i) \tag{15.4}$$

CLAVE

...icos pueden ...discernimiento ...e los datos ...les y la ...d del ...o asociados con ...diseño ...nico.

8 Dependencia *hacia fuera* se define como el número de módulos inmediatamente subordinados al módulo i; es decir, el número de módulos invocados directamente por i. Lo contrario, dependencia hacia dentro, sería una variable f_{in} que indique el número de módulos que invocan directamente al módulo i.

A medida que aumentan estos valores, la complejidad arquitectónica general del sistema también lo hace. Esto lleva a una mayor probabilidad de que aumenten los esfuerzos de integración y prueba

Fenton [FEN91] sugiere varias métricas simples de morfología (es decir, de forma) que permiten la comparación entre diferentes arquitecturas de programas empleando un conjunto de dimensiones directas Si se toma como referencia la arquitectura de llamada y retorno de la figura 15.5, se definirán las siguientes métricas:

tamaño = $n + a$

donde n es el número de nodos y a, el de arcos. En el caso de la arquitectura mostrada en la figura 15.5,

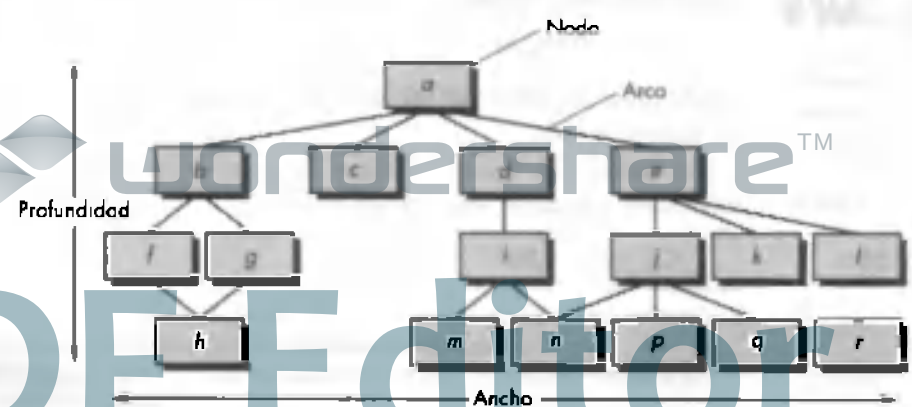
tamaño = 17 + 18 = 35
profundidad = 4, el camino más largo desde el nodo raíz (superior) a un nodo hoja.
anchura = 6, número máximo de nodos en cualquier nivel de la arquitectura
relación arco-a-nodo, $r = a/n$,

que mide la densidad de las conexiones y proporciona una simple indicación de acoplamiento de la arquitectura En el caso de la arquitectura mostrada en la figura 15.5, $r = 18/17 = 1.06$.

El Comando de Sistemas de la Fuerza Aérea de Estados Unidos [USA87] ha desarrollado varios indicadores de la calidad del software que se basan en las características de diseño que pueden medirse en un programa de computadora. Empleando conceptos similares a los propuestos en el IEEE Std. 982.1-1988 [IEE94], la Fuerza Aérea estadounidense emplea información obtenida del diseño de datos y arquitectónico para derivar un índice de calidad de la estructura de diseño (ICED) que va de 0 a 1. El cálculo del ICED requiere determinar los siguientes valores [CHA89].

S_1 = el número total de módulos definidos en la arquitectura del programa

FIGURA 15.5
Métricas de morfología.



S_2 = el número de módulos cuya función correcta depende de la fuente de entrada de datos o que produce datos que se usarán en otro lugar (en general, los módulos de control, entre otros, no se contarían como parte de S_2)

S_3 = el número de módulos cuya función correcta depende del procesamiento anterior

S_4 = el número de elementos de base de datos (incluye objetos de datos y todos los atributos que definen objetos)

S_5 = el número total de elementos únicos de base de datos

S_6 = el número de segmentos de base de datos (registros diferentes u objetos individuales)

S_7 = el número de módulos con una sola entrada y salida (con excepción del procesamiento, no se considera una salida múltiple)

Una vez que se han determinado los valores del S_1 al S_7 para un programa de computadora, es posible calcular los siguientes valores intermedios:

Estructura del programa. D_1, donde D_1 se define como sigue: si el diseño arquitectónico se desarrolló empleando un método distinto (por ejemplo, diseño orientado al flujo de datos u objetos), entonces $D_1 = 1$; de lo contrario, $D_1 = 0$.

Independencia del módulo: $D_2 = 1 - (S_2/S_1)$

Módulos no dependientes del procesamiento anterior: $D_3 = 1 - (S_3/S_1)$

Tamaño de la base de datos: $D_4 = 1 - (S_5/S_4)$

División en compartimientos de la base de datos: $D_5 = 1 - (S_6/S_4)$

Característica de entrada/salida del módulo. $D_6 = 1 - (S_7/S_1)$

Una vez determinados los valores intermedios, se calcula el ICED de la siguiente manera:

$$\text{ICED} = \sum w_i D_i \qquad (15.5)$$

donde $i = 1$ a 6, w_i es el peso relativo de la importancia de cada uno de los valores intermedios, y $\sum w_i = 1$ (si todo D_i tiene pesos iguales, entonces $w_i = 0.167$).

Se determina el valor de ICED para los diseños anteriores y se compara con un diseño que está en desarrollo. Si el ICED es significativamente menor que el promedio, lo indicado es realizar trabajo de diseño y revisión adicionales. De igual manera, si se van a realizar cambios importantes en un diseño existente, podrá calcularse el efecto de esos cambios sobre el ICED

> "Es posible considerar que la medición es un desvío. Un desvío necesario, porque la mayoría de los seres humanos no son capaces de tomar decisiones claras y objetivas (sin apoyo cuantitativo)."
>
> **Horst Zuse**

15.4.2 Métricas para el diseño orientado a objetos

Gran parte del diseño orientado a objetos es subjetivo (un diseñador experimentado "sabe" cómo caracterizar un sistema orientado a objetos para que implemente efec-

tivamente los requisitos del cliente). Pero, a medida que aumenta el tamaño y la complejidad del modelo de diseño orientado a objetos, un concepto más objetivo de las características del diseño beneficiaría al diseñador experimentado (que obtendría conocimientos adicionales) y al principiante (que obtendría una indicación de la calidad que de otra manera no estaría disponible).

En un tratamiento detallado de las métricas del software para sistemas orientados a objetos, Whitmire [WHI97] describe nueve características distintivas y mensurables de un diseño orientado a objetos:

¿Cuáles características pueden medirse cuando se evalúa un diseño orientado a objetos?

Tamaño. El tamaño se define a partir de cuatro conceptos: población, volumen, longitud y funcionalidad. *Población* se mide al tomar un conteo estático de entidades orientada a objetos como clases u operaciones. Las medidas de *volumen* son idénticas a las de la población, pero se recopilan dinámicamente (en un momento determinado). La *longitud* es una medida de una cadena de elementos de diseño interconectados (por ejemplo, la profundidad de un árbol de herencia es una medida de longitud). Las métricas de *funcionalidad* proporcionan una indicación indirecta del valor entregado al cliente en una aplicación orientada a objetos.

Complejidad. Como el tamaño, hay muchos conceptos diferentes de la complejidad del software [ZUS97]. Whitmire considera la complejidad desde el punto de vista de las características estructurales, al examinar la manera en que se interrelacionan las clases de un diseño orientado a objetos.

Acoplamiento. Las conexiones físicas entre los elementos de un diseño orientado a objetos (por ejemplo, el número de colaboraciones entre clases o el de mensajes pasados entre objetos) representan el acoplamiento dentro de un sistema orientado a objetos.

Suficiencia. Whitmire define suficiencia como "el grado en que una abstracción posee las características que se le piden, o el grado en que un componente de diseño posee características en su abstracción, desde el punto de vista de la aplicación actual". Expresado de otra manera, se pregunta: ¿Cuáles propiedades debe tener esta abstracción (clase) para que sea útil? [WHI97]. En esencia, un componente de diseño (por ejemplo, una clase) es suficiente si refleja plenamente todas las propiedades del objeto de dominio de la aplicación que se está modelando (es decir, que la abstracción, o clase, posee las características que debe tener).

> "Muchas de las decisiones para las que tenía que depender del folclore o los mitos puedo tomarlas ahora empleando datos cuantitativos."
>
> **Scott Whitmire**

Grado de avance. La única diferencia entre el grado de avance y la suficiencia es "el conjunto de características contra las que comparamos el componente de abstracción o diseño" [WHI97]. La suficiencia compara la abstracción desde el punto de vista de la aplicación actual. El grado de avance considera varios puntos de vista

planteando la pregunta: ¿Cuáles propiedades se requieren para representar plenamente el objeto del dominio del problema? Debido a que los criterios para el grado de avance consideran diferentes puntos de vista, indican indirectamente el grado en que puede reutilizarse el componente de abstracción o diseño.

Cohesión. Como su contraparte en el software convencional, un componente orientado a objetos debe diseñarse de manera que todas las operaciones trabajen en combinación para alcanzar un solo propósito, bien definido. El grado de cohesión de una clase se determina al examinar el grado en que "el conjunto de propiedades que posee es parte del dominio del problema o el diseño" [WHI97]

Primitivismo. Una característica similar a la simplicidad, el grado de primitivismo (aplicado a operaciones y clases) es el grado en que una operación es atómica (es decir, la operación no puede construirse a partir de una secuencia de otras operaciones contenidas dentro de una clase). Una clase que muestra un alto grado de primitivismo sólo encapsula operaciones primitivas.

Similitud. Esta medida indica el grado en que dos o más clases son similares en cuanto a su estructura, función, comportamiento o propósito.

Volatilidad. Como ya se ha visto en este libro, los cambios de diseño ocurren cuando los requisitos se modifican o cuando las modificaciones se presentan en otra parte de una aplicación, lo que produce una adaptación obligatoria del componente del diseño en cuestión. La volatilidad de un componente de diseño orientado a objetos mide la probabilidad de que ocurra un cambio

En realidad, las métricas del producto para sistemas orientados a objetos no sólo se aplican al modelo de diseño, sino también al de análisis. En las secciones que siguen se explorarán las métricas que proporcionan una indicación de la calidad al nivel de clase orientada a objetos y al nivel de operación.

15.4.3 Métricas orientadas a clases: la colección de métricas de CK

La clase es la unidad fundamental de un sistema orientado a objetos. Por tanto, las medidas y métricas de una clase individual, la jerarquía de clase y las colaboraciones de clase serán invaluables para un ingeniero de software que debe valorar la calidad del diseño. En capítulos anteriores se vio que la clase encapsula operaciones (procesamiento) y atributos (datos). La clase suele ser el "predecesor" de las subclases (a veces denominadas *descendientes*) que heredan sus atributos y operaciones. Con frecuencia, la clase colabora con otras clases. Cada una de estas características se utiliza como base de la medición.[9]

Chidamber y Kemerer [CH194] propusieron uno de los *conjuntos de métricas de software* orientado a objetos al que se hace referencia con mayor frecuencia. A me-

9 Debe observarse que aún se debate en la bibliografía técnica la validez de algunas de las métricas analizadas en este capítulo. Quienes defienden la teoría de la medición, exigen un grado de formalismo que algunas métricas orientadas a objetos no proporcionan. Sin embargo, es razonable determinar que las métricas indicadas proporcionan conocimientos útiles para el ingeniero de software

nudo denominadas colección de métricas de CK, los autores proponen seis métricas de diseño basado en clases para sistemas orientados a objetos.[10]

PUNTO CLAVE

El número de métodos y su complejidad están directamente correlacionados con el esfuerzo requerido para probar una clase.

Métodos ponderados por clase (MPC). Suponga que n métodos de complejidad $c_1, c_2, \ldots, c_n$ están definidos por la clase **C**. La métrica de complejidad especifica que se elija (por ejemplo, la complejidad ciclomática) debe normalizarse con el fin de que la complejidad nominal de un método tome un valor de 1.0.

$$\text{MPC} = \sum c_i$$

para $i = 1$ a n. El número de métodos y su complejidad son indicadores razonables de la cantidad de esfuerzo requerido para implementar y probar una clase. Además, cuanto mayor sea el número de métodos, más complejo será el árbol de herencia (todas las subclases heredan los métodos de sus predecesores). Por último, conforme crece el número de métodos de una clase determinada, es probable que se vuelva más y más específica de la aplicación, lo que limita sus posibilidades de reutilización. Por todas las razones mencionadas, MPC debe mantenerse lo más bajo que sea posible.

CONSEJO

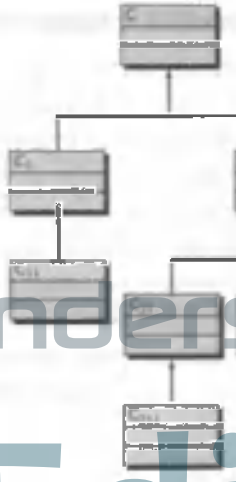
La herencia es una característica extremadamente poderosa que puede causar problemas si se emplea sin cuidado. Úsese el APH y otras métricas para obtener una lectura de la complejidad de las jerarquías de clase.

Aunque parezca relativamente simple desarrollar un conteo del número de métodos en la clase, en realidad el problema es más complejo de lo que parece. Debe desarrollarse un enfoque de conteo consistente para los métodos [CHU95].

Árbol de profundidad de la herencia (APH). Esta métrica es "la longitud máxima desde el nodo hasta la raíz del árbol" [CHI94]. Si toma como referencia la figura 15.6, el valor de APH para la jerarquía de clase mostrada es 4. A medida que crece la APH, es probable que las clases de nivel inferior heredarán muchos métodos. Es-

FIGURA 15.6

Jerarquía de una clase.



10 Chidamber y Kemerer usan el término *métodos* en lugar de *operaciones*. La forma en que emplean el término se refleja en esta sección.

to se presta a posibles dificultades cuando se trata de predecir el comportamiento de una clase. Una jerarquía de clase profunda (su APH es mayor) también se presta a una mayor complejidad de diseño. Por el lado positivo, valores grandes de APH indican que se podrían reutilizar muchos métodos.

Número de descendientes (NDD). Un *descendiente* es una subclase que se encuentra inmediatamente subordinada a otra en la jerarquía de clases. Si se toma como referencia la figura 15.6, la clase $\mathbf{C_2}$ tiene tres descendientes (las subclases $\mathbf{C_{21}}$, $\mathbf{C_{22}}$ y $\mathbf{C_{23}}$). A medida que crece el número de descendientes, se incrementa la reutilización, pero podría diluirse la abstracción que representa la clase predecesora si alguno de los descendientes no es un miembro apropiado de la clase predecesora. A medida que aumenta el NDD, también lo hace la cantidad de pruebas (requeridas para ejercitar cada descendiente en su contexto operacional).

Acoplamiento entre clases de objetos (AECO). El modelo de conjunto de respuesta de una clase (CRC), expuesto en el capítulo 8, se emplea para determinar el valor de AECO. En esencia, AECO es el número de colaboraciones enlistadas, para una clase, en su tarjeta de índice CRC.[11] A medida que AECO aumenta, es probable que disminuya la facilidad de reutilización de una clase. Valores elevados de AECO también complican las modificaciones y la prueba que asegura que esas modificaciones se han hecho En general, para cada clase deben mantenerse los valores de AECO en el valor más bajo que sea razonable. Esto es consistente con la directriz general para reducir el acoplamiento en el software convencional.

Respuesta para una clase (RPC). El conjunto de respuesta para una clase es un "conjunto de métodos que tiene la posibilidad de ejecutarse como respuesta a un mensaje que recibe un objeto de esa clase" [CHI94]. La RPC es el número de métodos en el conjunto de respuesta. A medida que la RPC aumenta, el esfuerzo requerido para probar también lo hace, debido a que crece la secuencia de prueba (capítulo 14). También se desprende que, a medida que la RPC aumenta, se incrementa la complejidad del diseño general de la clase

CONSEJO

...ceptos de ...nto y ...o se aplican al ...e convencional ...enfocado a ...s. Manténgase ...l acoplamiento ...la cohesión de ...s y operaciones.

Falta de cohesión en métodos (FCM). Cada método dentro de una clase, **C**, tiene acceso a uno o más atributos *(también denominados variables de instancia)*. La FCM es el número de métodos que acceden a uno o más de los mismos atributos.[12] Si ningún método accede a los mismos atributos, entonces FCM = 0. Para ilustrar el caso donde FCM ≠ 0, imagínese una clase de seis métodos. Cuatro de ellos tienen uno o más atributos en común (es decir, acceden a atributos comunes). Por tanto, FCM = 4. Si la FCM es alta, los métodos pueden acoplarse entre sí mediante atributos. Esto aumenta la complejidad del diseño de clase. Aunque hay casos en que re-

11 Si las tarjetas de índice CRC se desarrollan manualmente, el grado de avance y la consistencia deben evaluarse antes de determinar el AECO de manera confiable

12 La definición formal es un poco más compleja. Consúltese [CHI94] para conocer más detalles

sulta justificable un valor elevado para la FCM. Lo deseable es mantener alta la cohesión; es decir, conservar baja la FCM [13]

HOGARSEGURO

Aplicación de métricas de CK

El escenario: Cubículo de Vinod.

Los actores: Vinod, Shakira y Ed, integrantes del equipo del software *HogarSeguro*, que siguen trabajando en el diseño al nivel de componentes y de casos de prueba.

La conversación:

Vinod: ¿Tuvieron oportunidad de leer la descripción de la colección de métricas de CK que les envié el miércoles y de hacer esas mediciones?

Shakira: No fue muy complicado. Regresé a mis diagramas de clase y de secuencia UML, como sugeriste, y obtuve cuentas elementales de APH, FPC y FCM. No pude encontrar el modelo CRC, de modo que no conté AECO.

Jamie (sonriendo): No pudiste encontrar el modelo CRC porque yo lo tenía.

Shakira: Eso es lo que me encanta de este equipo, la gran comunicación.

Vinod: Yo hice mis conteos... ¿desarrollaron cifras para las métricas de CK?

(Jamie y Ed asienten.)

Jamie: Como tenía las tarjetas CRC, eché un vistazo al AECO y parecía muy uniforme en casi todas las clases. Hubo una excepción, y la anoté.

Ed: Hay unas cuantas clases donde la RPC es muy elevada, comparada con las asociaciones verdaderas... tal vez debemos echar un vistazo para simplificarlas.

Jamie: Tal vez sí, tal vez no. Todavía estoy preocupada por el tiempo, y no quiero corregir cosas que realmente no están mal.

Vinod: Estoy de acuerdo con eso. Tal vez debemos buscar clases que tengan malos números en al menos dos o más métricas de CK. Digamos que si le pasan dos *strikes*, hay que modificarlas.

Shakira (mirando la lista de clases de Ed con alta RPC): Mira, ¿ves esta clase? Tiene una FCM alta, además de una RPC alta. ¿Dos *strikes*?

Vinod: Sí, así lo creo... por lo mismo, será difícil de implementar debido a la complejidad y dificultad de probar. Tal vez valga la pena diseñar dos clases separadas para alcanzar el mismo comportamiento.

Jamie: ¿Crees que la modificación nos ahorrará tiempo?

Vinod: A la larga, sí.

15.4.4 Métricas orientadas a objetos: la colección de métricas para el diseño orientado a objetos

Harrison, Counsell y Nithi [HAR98] proponen un conjunto de métricas para diseño orientado a objetos que proporcionan indicadores cuantitativos para las características del diseño orientado a objetos. A continuación se presenta una pequeña muestra de estas métricas:

13 La métrica FCM proporciona conocimientos útiles en algunas situaciones, pero puede malinterpretarse en otras. Por ejemplo, mantener el acoplamiento encapsulado dentro de una clase aumenta la cohesión del sistema como un todo. Por tanto, por lo menos en un sentido importante, un FCM más elevado en realidad sugiere que una clase puede tener una mayor cohesión, no una menor.

Método del factor de herencia (MFH). El grado en que la arquitectura de clases de un sistema orientado a objetos usa la herencia para métodos (operaciones) y atributos se define como

$$\text{MFH} = \sum M_i(C_i) / \sum M_a(C_i)$$

donde la sumatoria se presenta desde $i = 1$ hasta T_c. T_c se define como el número total de clases en la arquitectura; C_i es una clase dentro de la arquitectura y

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

donde

$M_a(C_i)$ = el número de métodos que pueden invocarse en asociación con C_i.

$M_d(C_i)$ = el número de métodos declarados en la clase C_i.

$M_i(C_i)$ = el número de métodos heredados (y no redefinidos) en C_i.

El valor de MFH (el atributo de factor de herencia, AFH, se define de manera análoga) es un indicativo del impacto de la herencia en el software orientado a objetos.

> "El análisis del software orientado a objetos para evaluar su calidad se está volviendo cada vez más importante a medida que el paradigma [orientado a objetos] sigue ganando popularidad."
>
> **Rachel Harrison *et al.***

Factor de acoplamiento (FA). Al principio de este capítulo se indicó que el acoplamiento es una indicación de las conexiones entre elementos de un diseño orientado a objetos. El conjunto de métricas del diseño orientado a objetos define el acoplamiento de la siguiente manera:

$$\text{FA} = \sum_i \sum_j \textit{es\_cliente}\ (C_i, C_j) / (T_c^2 - T_c)$$

donde las sumatorias van desde $i = 1$ hasta T_c y desde $j = 1$ hasta T_c. La función

es_cliente = 1, si y sólo si existe una relación entre la clase cliente, C_c, y la clase servidor, C_s, y $C_c \neq C_s$
= 0, en cualquier otro caso

Aunque muchos factores afectan la complejidad, la facilidad de comprensión y el mantenimiento del software, resulta razonable concluir que, a medida que aumenta el valor de FA, también aumentará la complejidad del software orientado a objetos y, como consecuencia, es posible que resulten afectadas la facilidad de comprensión y mantenimiento, junto con la posibilidad de reutilización.

Harrison y sus colegas [HAR98] presentan un análisis detallado de MFH y FA, junto con otras métricas, y examinan su validez para emplearlos en la evaluación de la calidad del diseño.

15.4.5 Métricas orientadas a objetos propuestas por Lorenz y Kidd

En su libro sobre métricas orientadas a objetos, Lorenz y Kidd [LOR94] dividen las métricas basadas en clases en cuatro amplias categorías, cada una con un diseño al nivel de componentes: tamaño, herencia, valores internos y valores externos. Las métricas orientadas al tamaño aplicadas a una clase de diseño orientado a objetos se concentran en el conteo de atributos y operaciones de una clase individual, así como en valores promedio para el sistema orientado a objetos como un todo. Las métricas basadas en la herencia se concentran en la manera en que las operaciones se reutilizan en la jerarquía de clases. Las métricas para los valores internos buscan cohesión y aspectos orientados al código, y las métricas de valores externos examinan el acoplamiento y la reutilización. A continuación se presenta una muestra de las métricas propuestas por Lorenz y Kidd:

Durante la revisión del modelo de análisis, las tarjetas de índice CRC proporcionarán una indicación razonable de los valores esperados para el tamaño de la clase. Si se encuentra una clase con un número grande de responsabilidades, piénsese en dividirla.

Tamaño de la clase (TC). El tamaño general de una clase se determina con las siguientes medidas:

- El número total de operaciones (de instancia heredada y privada) que están encapsuladas dentro de la clase
- El número de atributos (de instancia heredada y privada) que están encapsulados por la clase

La métrica MPC que propusieron Chidamber y Kemerer (sección 15.4.3) también es una medida ponderada de tamaño de clase. Como ya se indicó, los valores grandes de TC indican que tal vez una clase tenga demasiada responsabilidad. Esto reducirá la posibilidad de reutilización de la clase y complicará la implementación y la prueba. En general, debe dársele más peso a las operaciones y los atributos heredados o públicos para determinar el tamaño de la clase [LOR94]. Las operaciones y los atributos privados permiten la especialización y están más focalizados en el diseño. También deben calcularse los promedios para el número de atributos y operaciones de clase. Cuanto menores sean los valores promedio para el TC, más probable será que las clases dentro del sistema puedan reutilizarse ampliamente.

Número de operaciones añadidas por una subclase (NOA). Las subclases se especializan al agregar operaciones y atributos. A medida que el valor de NOA aumenta, la subclase se aparta de la abstracción implícita en la superclase. En general, a medida que la profundidad de la jerarquía de clase aumenta (APH se vuelve mayor), debe caer el valor de NOA en los niveles inferiores de la jerarquía.

15.4.6 Métricas de diseño al nivel de componentes

Las métricas de diseño al nivel de componentes del software convencional se concentran en las características internas de un componente de software e incluyen medidas de cohesión, acoplamiento y complejidad del módulo. Estas medidas ayudan a un ingeniero de software a juzgar la calidad de un diseño al nivel de componentes.

Las métricas presentadas en esta sección son de "caja de cristal", en el sentido de que requieren conocimiento del funcionamiento interno del módulo que se está considerando. Las métricas de diseño al nivel de componentes se aplican una vez que se ha desarrollado el diseño procedimental. Como opción, pueden demorarse hasta que el código fuente esté disponible.

Métricas de cohesión. Bieman y Ott [BIE94] definen una colección de métricas que proporcionan una indicación del grado de cohesión (capítulo 9) de un módulo. Las métricas se definen a partir de cinco conceptos y medidas:

Porción de datos. Definido simplemente, una porción de datos es un recorrido hacia atrás por un módulo; busca valores de datos que afectan el estado del módulo cuando comienza el recorrido. Debe indicarse que es posible definir las porciones del programa (que se concentran en instrucciones y condiciones) y las porciones de datos.

Muestras de datos. Las variables definidas para un módulo se definen como muestras de datos para el módulo.

Señales de unión. Este conjunto de muestras de datos cae en una o más porciones de datos.

Señales de superunión. Estas muestras de datos son comunes a todas las porciones de datos de un módulo.

Capacidad de unión. La capacidad de unión relativa de una señal de unión es directamente proporcional al número de porciones de datos que une.

Bieman y Ott desarrollan métricas para *cohesión funcional fuerte*, *cohesión funcional débil* y *adhesividad* (que se relaciona con el grado en que las señales de unión integran las porciones de datos). Estas métricas se interpretan de la siguiente manera [BIE94]:

> Todas estas métricas de cohesión abarcan valores de 0 a 1. Tienen un valor de 0 cuando un procedimiento cuenta con más de una salida y no muestra atributo alguno de cohesión indicado por una métrica particular. Un procedimiento sin señales de superunión (es decir, sin muestras comunes a todas las porciones de datos), tiene 0 cohesión funcional fuerte (no hay muestras de datos que contribuyan a todas las salidas). Un procedimiento sin señales de unión (es decir, sin muestras comunes a más de una porción de datos, en procedimientos con más de una porción de datos) muestra 0 cohesión funcional débil y 0 adhesividad (no hay muestras de datos que contribuyan a más de una salida).

La cohesión funcional fuerte y la adhesividad se encuentran cuando las métricas de Bieman y Ott toman un valor máximo de 1.

Métricas de acoplamiento. El acoplamiento del módulo proporciona una indicación de la "conectividad" de un módulo con otros módulos, con datos globales y con el entorno exterior. En el capítulo 9 se analizó el acoplamiento desde el punto de vista cualitativo.

Dhama [DHA95] ha propuesto una métrica para el acoplamiento del módulo que abarca el acoplamiento de flujo de datos y de control, el global y el de entorno. Las medidas necesarias para calcular el acoplamiento del módulo se definen a partir de cada uno de los tres tipos de acoplamiento indicados antes. En el caso del acoplamiento de flujo de datos y de control,

d_e = número de parámetros de datos de entrada

c_e = número de parámetros de control de entrada

d_s = número de parámetros de datos de salida

c_s = número de parámetros de control de salida

En el caso del acoplamiento global:

g_d = número de variables globales usadas como datos

g_c = número de variables globales usadas como control

En el caso del acoplamiento de entorno:

w = número de módulos llamados (dependencia hacia fuera)

r = número de módulos que llaman al módulo en cuestión (dependencia hacia dentro)

Con estas medidas se define un indicador de acoplamiento del módulo, m_d, de la siguiente manera:

$$m_d = k/M$$

donde k es una constante de proporcionalidad y

$$M = d_e + (a \times c_e) + d_s + (b \times c_s) + g_d + (c \times g_c) + w + r \qquad (15$$

Los valores de k, a, b y c deben derivarse empíricamente.

A medida que el valor de m_d aumenta, disminuye el acoplamiento general del módulo. Para lograr que la métrica de acoplamiento suba a medida que aumenta el grado de acoplamiento, se define una métrica de acoplamiento revisada

$$C = 1 - m_d$$

donde el grado de acoplamiento aumenta a medida que lo hacen las medidas en la ecuación (15.6).

Métricas de complejidad. Es posible calcular diversas métricas del software para determinar la complejidad del flujo de control del programa. Muchas de ellas se basan en la gráfica de flujo. Como se analizó en el capítulo 14, una gráfica es una representación compuesta de nodos y enlaces (también denominados aristas). Cuando los enlaces (aristas) están dirigidos, la gráfica de flujo es una gráfica dirigida.

McCabe y Watson [MCC94] identifican varios usos importantes para las métricas de complejidad:

> Las métricas de complejidad se utilizan para predecir la información crítica acerca de la confiabilidad y la facilidad de mantenimiento de sistemas de software a partir del análisis automático del código fuente [o la información del diseño procedimental]. Las métricas de complejidad también ofrecen retroalimentación durante el proyecto de software para ayudar a controlar [la actividad del diseño]. Durante las pruebas y el mantenimiento, ofrecen una información detallada acerca de los módulos de software para ayudar a detectar áreas de posible inestabilidad

PUNTO CLAVE

La complejidad ... sólo es una ... gran ... de métricas ... complejidad

La métrica de complejidad cuyo uso es el más extendido (y debatido) para el software de computadora es la complejidad ciclomática, originalmente desarrollada por Thomas McCabe [MCC76], [MCC89], y que se analizó con todo detalle en el capítulo 14.

Zuse ([ZUS90], [ZUS97]) presenta un análisis enciclopédico de no menos de 18 categorías diferentes de las métricas de complejidad del software. El autor presenta las definiciones básicas de métricas en cada categoría (por ejemplo, hay distintas variaciones de la métrica de complejidad ciclomática) y luego analiza y critica cada una. El trabajo de Zuse es el más completo publicado a la fecha.

15.4.7 Métricas orientadas a la operación

Debido a que la clase es la unidad dominante en los sistemas orientados a objetos, se han propuesto pocas métricas para operaciones que residen dentro de la clase. Churcher y Shepperd [CHU95] analizan esto cuando afirman: "Los resultados de estudios recientes indican que los métodos tienden a ser pequeños en cuanto al número de instrucciones y a complejidad lógica [WIL93], lo que sugiere que la estructura de conectividad de un sistema es más importante que el contenido de los módulos individuales". Sin embargo, se apreciarán mejor las cosas si se examinan las consultas promedio de métodos (operaciones). Tres métricas simples, propuestas por Lorenz y Kidd [LOR94], resultan apropiadas:

Tamaño promedio de operación (TO_{prom}). Aunque las líneas de código podrían usarse como indicador del tamaño de operación, la medida de líneas de código adolece de una serie de problemas analizados en el capítulo 22. Por ello, el número de mensajes que envía la operación proporciona una opción al tamaño de operación. A medida que aumenta el número de mensajes enviados por una sola operación, es probable que las responsabilidades no se hayan asignado bien dentro de la clase.

Complejidad de la operación (CO). La complejidad de una operación se calcula empleando cualquier métrica de complejidad propuesta para el software convencional [ZUS90]. Debido a que las operaciones deben limitarse a una responsabilidad específica, el diseñador debe esforzarse por mantener la CO lo más baja posible.

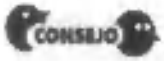
Número promedio de parámetros de la operación (NPO_{prom}). Mientras ma yor sea el número de parámetros de la operación, más compleja será la colaboración entre los objetos. En general, el NPO_{prom} debe mantenerse lo más bajo posible.

15.4.8 Métricas de diseño de la interfaz de usuario

Aunque hay obras importantes que tratan el diseño de interfaces ser humano/máquina (capítulo 12), se ha publicado relativamente poca información sobre métricas que proporcionen conocimientos profundos sobre la calidad y la facilidad de uso de la interfaz.

Sears [SEA93] sugiere que lo *apropiado del formato* (AF) es una métrica de diseño valiosa para interfaces ser humano/máquina. Una GUI común aplica entidades de formato (iconos gráficos, texto, menús, ventanas, etc.) para ayudar al usuario a completar tareas. Para realizar una tarea determinada con una GUI, el usuario debe pasar de una entidad de presentación a la siguiente. La posición absoluta y relativa de cada entidad de presentación, la frecuencia con que se emplea y el "costo" de la transición de una entidad de formato a la siguiente contribuirán a determinar lo apropiado de la interfaz.

> "Aprenderá por lo menos un principio del diseño de interfaces de usuario si echa la ropa en una lavadora. Si pone demasiada ropa, nada quedará limpio."
>
> Anónimo



Las métricas de diseño de la interfaz son adecuadas, pero sobre todo lo demás, es necesario asegurarse plenamente de que la interfaz le gusta a los usuarios finales y de que éstos se sienten cómodos con las interacciones requeridas

Kokol y sus colegas [KOK95] definen una métrica de cohesión para las pantallas de la interfaz de usuario que mide la conexión relativa entre el contenido de una pantalla y el de otra. Si los datos (o el contenido adicional) presentados en una pantalla pertenecen a un solo objeto importante de datos (como se definió dentro del modelo de análisis), la cohesión de la interfaz para esa pantalla será alta. Si se presentan muchos tipos diferentes de datos o contenidos y esos datos se relacionan con diferentes objetos de datos, la cohesión de la interfaz de usuario será baja. Los autores proporcionan modelos empíricos para la cohesión [KOK95].

Además, las medidas directas de la interacción con la interfaz de usuario se concentran en la medición del tiempo requerido para alcanzar un escenario o una operación específicos, el tiempo requerido para recuperarse de una condición de error, los conteos de operaciones o tareas específicas requeridas para alcanzar un caso de uso, el número de objetos de datos o contenido presentados en una pantalla, la densidad y el tamaño del texto y muchos otros. Sin embargo, estas medidas directas deben estar organizadas para crear métricas de interfaz de usuario que tengan un significado y que lleven a mejorar la calidad, la facilidad de uso, o ambos elementos de la interfaz de usuario.

Es importante observar que la selección de un diseño de interfaz gráfica de usuario puede determinarse a partir de métricas como AF o la cohesión de pantalla de la interfaz de usuario, pero el árbitro final debe ser la entrada del usuario basada en prototipos de interfaz gráfica de usuario. Nielsen y Levy [NIE94] reportan que "se tiene una probabilidad razonablemente grande de éxito si se elige entre las interfaces

[diseños] basadas exclusivamente en las opiniones de los usuarios. El desempeño promedio de las tareas de los usuarios y su satisfacción subjetiva con una interfaz gráfica de usuario tienen una elevada correlación"

15.5 MÉTRICAS PARA EL CÓDIGO FUENTE

La teoría de Halstead de la "ciencia del software" [HAL77] propuso las primeras "leyes" analíticas para el software de computadora.[14] Halstead asignó leyes cuantitativas al desarrollo de este software empleando un conjunto de medidas primitivas que se derivan después de que se ha generado el código, o se estiman una vez que el diseño esté completo. Las medidas son:

n_1 = el número de operadores distintos que aparecen en un programa
n_2 = el número de operandos distintos que aparecen en un programa
N_1 = el número total de veces que aparece el operador
N_2 = el número total de veces que aparece el operando.

CONSEJO

...de la interfaz ...s, pero ...todo lo demás, ...asegu... ...almente de ...interfaz le ...a los usuarios ...s y de que éstos ...cómodas ...s interacciones

Halstead aplica estas medidas primitivas para desarrollar expresiones relacionadas con la longitud global del programa, el volumen mínimo posible para un algoritmo, el volumen real (número de bits requeridos para especificar un programa), el nivel del programa (una medida de la complejidad del software), el nivel del lenguaje (una constante para un lenguaje determinado) y otras características como esfuerzo de desarrollo, tiempo de desarrollo y hasta el número proyectado de fallas en el software.

Halstead demuestra que la longitud N se puede estimar así:

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

y que el volumen de programa se puede definir como:

$$V = N \log_2 (n_1 + n_2)$$

Se debe observar que V variará de acuerdo con el lenguaje de programación y que representa el volumen de información (en bits) necesario para especificar un programa.

> "El conjunto de reglas que sigue el cerebro humano [para el desarrollo de algoritmos] es mas rígido de lo que suele pensarse."
>
> **Maurice Halstead**

En teoría, debe existir un volumen mínimo para un algoritmo determinado. Halstead define una relación de volumen, L, como la relación entre el volumen de la forma más compacta de un programa y el volumen real del programa. En realidad, L siempre debe ser menor que 1. Desde el punto de vista de las medidas primitivas, la relación de volumen se expresa como

14 Debe observarse que las "leyes" de Halstead han generado gran controversia, y muchos creen que la teoría tiene fallas. Sin embargo, se ha realizado la verificación experimental de lenguajes de programación seleccionados (por ejemplo, [FEL89]).

$$L = 2/n_1 \times n_2/N_2$$

El trabajo de Halstead es sensible a la verificación experimental, y se ha realizado una gran cantidad de investigación sobre la ciencia del software. Para obtener más información, consúltense [ZUS90], [FEN91] y [ZUS97].

15.6 MÉTRICAS PARA PRUEBAS

Aunque se ha escrito mucho sobre las métricas del software para pruebas (por ejemplo, [HET93]), casi todas las métricas propuestas se concentran en el proceso de prueba, no en las características técnicas de las propias pruebas. En general, quienes aplican las pruebas deben depender de las métricas de análisis, diseño y código como guía para el diseño y la ejecución de los casos de prueba.

Las métricas basadas en la función (sección 15.3.1) se aplican para predecir el esfuerzo general de la prueba. Es posible recopilar varias características al nivel de proyecto (como el esfuerzo y el tiempo para las pruebas, los errores descubiertos, el número de casos de prueba producidos) de proyectos anteriores y correlacionarlas con el número de puntos de función que produce un equipo de proyecto. Este equipo tiene la opción posterior de proyectar "valores esperados" de estas características para el proyecto actual.

PUNTO CLAVE

Las métricas de prueba se agrupan en dos amplias categorías: 1) las que tratan de predecir el número probable de pruebas que se requieren a varios niveles de prueba, y 2) las que se concentran en la cobertura de la prueba para un componente determinado.

Las métricas del diseño arquitectónico proporcionan información sobre la facilidad o la dificultad asociada con la prueba de integración (capítulo 13) y la necesidad de contar con software especializado en pruebas (por ejemplo, resguardos y controladores). La complejidad ciclomática (una métrica de diseño al nivel de componentes) cae en el eje de las pruebas de camino básico, un método de diseño de casos de prueba presentado en el capítulo 14. Además, la complejidad ciclomática se emplea para determinar los módulos que serán candidatos a pruebas de unidad más extensas. Los módulos con elevada complejidad ciclomática son más propensos a errores que los que tienen una menor complejidad. Por ello, la persona responsable de la prueba debe realizar un esfuerzo superior al promedio para descubrir errores en estos módulos antes de integrarlos en un sistema.

15.6.1 Métricas de Halstead aplicadas a las pruebas

También es posible estimar el esfuerzo que requieren las pruebas mediante métricas derivadas de las medidas de Halstead (sección 15.5). Si se aplican las definiciones del volumen, V, y el nivel de un programa, NP, el esfuerzo de Halstead, e, se calculan así:

$$NP = 1/[(n_1/2) \times (N_2/n_2)] \quad (15.7a)$$

$$e = V/NP \quad (15.7b)$$

El porcentaje del esfuerzo general de prueba que se debe asignar a un k se estimaría con la siguiente relación:

$$\text{porcentaje de esfuerzo de prueba } (k) = e(k)/\sum e(i) \quad (15.8$$

donde $e(k)$ se calcula para el módulo k empleando las ecuaciones (15.7), y donde

sumatoria en el denominador de la ecuación (15.8) es la suma del esfuerzo de Halstead en todos los módulos del sistema.

15.6.2 Métricas para pruebas orientadas a objetos

Las métricas del diseño orientado a objetos expuestas en la sección 15.4 proporcionan una indicación de la calidad del diseño. También proporcionan una indicación general de la cantidad de esfuerzo necesario en la prueba para ejercitar un sistema orientado a objetos.

Binder [BIN94] sugiere una amplia serie de métricas de diseño que tienen una influencia directa sobre la "facilidad de prueba" de un sistema orientado a objetos. Las métricas toman en cuenta aspectos de encapsulamiento y herencia. A continuación se presenta una muestra:

Falta de cohesión en métodos (FCM).[15] Mientras mayor sea el valor de FCM, deben probarse más estados para asegurar que los métodos no generen efectos colaterales.

Porcentaje público y protegido (PYP). Esta métrica indica el porcentaje de atributos de clase que son públicos o están protegidos. Valores elevados de PYP aumentan la probabilidad de efectos colaterales entre clases porque los atributos públicos o protegidos conllevan una alta probabilidad de acoplamiento (capítulo 9).[16] Deben diseñarse pruebas para asegurar el descubrimiento de estos efectos colaterales.

Integrantes de acceso público a datos (APD). Esta métrica indica el número de clases (o métodos) al que tienen acceso otros atributos de clase, lo que es una violación del encapsulamiento. Valores elevados de APD conllevan la posibilidad de efectos colaterales entre clases. Deben diseñarse pruebas para asegurar el descubrimiento de estos efectos colaterales.

Número de clases raíz (NCR). Esta métrica es un conteo de las distintas jerarquías de clase descritas en el modelo de diseño. Deben desarrollarse conjuntos de prueba para cada clase raíz y para la jerarquía de clases correspondiente. A medida que aumente el NCR, también aumentará el esfuerzo de la prueba.

Dependencia hacia dentro (FIN). Cuando se aplica en el contexto orientado a objetos, la dependencia hacia dentro para la jerarquía de herencia es un indicador de herencia múltiple. FIN > 1 indica que una clase hereda sus atributos y operaciones a partir de una clase raíz. Debe evitarse que FIN > 1 a toda costa.

Número de descendientes (NDD) y árbol de profundidad de herencia (APH).[17] Como se analizó en el capítulo 14, es necesario volver a probar los métodos de la superclase de cada subclase.

CONSEJO

...o orientado a ...ga a ser ...mplejo. Los ...s ayudarán a ... los recursos ... prueba a subpro... escenarios y ...s de clases ... "sospe... con base en ... características ... Es recomen... usarlas.

15 Consúltese la sección 15.4.3 para conocer una descripción de FCM.

16 Algunas personas promueven diseños en que ninguno de los atributos es público o privado; es decir, PYP = 0. Esto indica que todos los atributos deben accederse en otras clases por medio de métodos.

17 Consúltese la sección 15.4.3 para conocer una descripción de NDD y APH.

15.7 MÉTRICAS PARA EL MANTENIMIENTO

Todas las métricas del software presentadas en este capítulo se aplican también al desarrollo de nuevo software y al mantenimiento del existente. Sin embargo, se han propuesto métricas diseñadas explícitamente para actividades de mantenimiento.

El IEEE Std 982.1-1988 [IEE94] sugiere un índice de madurez del software (IMS) que proporciona una indicación de la estabilidad de un producto de software (basada en los cambios que ocurren con cada versión del producto). Se determina la siguiente información:

M_T = el número de módulos en la versión actual
F_c = el número de módulos cambiados en la versión actual
F_a = el número de módulos añadidos a la versión actual
F_d = el número de módulos de la versión anterior que se eliminaron en la actual

El índice de madurez del software se calcula de la siguiente manera:

$$\text{IMS} = [M_T - (F_a + F_c + F_d)]/ M_T$$

A medida que el IMS se acerca a 1.0, el producto empieza a estabilizarse. El IMS también se aplica como métrica para la planeación de actividades de mantenimiento de software. El tiempo medio para producir una versión de un producto de software puede correlacionarse con el IMS, y pueden desarrollarse modelos empíricos para el esfuerzo de mantenimiento.

HERRAMIENTAS DE SOFTWARE

Métricas del producto

Objetiva: Ayudar a los ingenieros de software en el desarrollo de métricas significativas que evalúen los productos del trabajo generados durante el modelado de análisis y diseño, la generación de código fuente y la prueba.

Mecánica: Las herramientas de esta categoría abarcan una amplia serie de métricas y se implementan como aplicaciones independientes o (con mayor frecuencia) como funcionalidad que existe dentro de las herramientas para análisis y diseño, codificación o prueba. En la mayor parte de los casos, la herramienta de métrica analiza una representación del software (por ejemplo, un modelo UML o el código fuente) y desarrolla una o más métricas.

Herramientas representativas[18]

Krakatau Metrics, desarrollada por Power Software (www.powersoftware.com/products), calcula métricas de complejidad, Halstead y otras relacionadas para C/C++ y Java.

Metrics4C, desarrollada por +1 Software Engineering (www.plus-one.com/Metrics4C-fact_sheet.html), calcula varias métricas arquitectónicas, de diseño y orientadas a código, además de otras orientadas a proyecto.

Rational Rose, desarrollada por Rational Corporation (www.rational.com), es un conjunto de herramientas completas para el modelado UML que incorpora varias características de análisis de métricas.

RSM, desarrollada por M-Squared Technologies (msquaredtechnologies.com/m2rsm/index.html), calcula una amplia variedad de métricas orientadas a configuración para C, C++ y Java.

Understand, desarrollada por Scientific Toolworks, Inc (www.scitools.com), calcula las métricas orientadas a código para diversos lenguajes de programación.

18 Las herramientas expuestas representan una muestra de esta categoría. En casi todos los casos los nombres de las mismas son marcas registradas de sus respectivos desarrolladores.

15.8 Resumen

Las métricas del software proporcionan una manera cuantitativa de evaluar la calidad de los atributos internos de un producto, lo que permite que un ingeniero de software evalúe la calidad antes de construirlo. Las métricas proporcionan los conocimientos necesarios para crear modelos efectivos de análisis y diseño, un código sólido y pruebas exhaustivas.

Para que resulte útil en la realidad, una métrica del software debe ser simple y calculable, persuasiva, consistente y objetiva. Debe ser independiente del lenguaje de programación y proporcionar retroalimentación efectiva al ingeniero del software.

Las métricas para el modelo de análisis se concentran en la función, los datos y el comportamiento (los tres componentes del modelo de análisis). Las métricas para el diseño consideran los aspectos del diseño de la arquitectura, al nivel de componentes y de la interfaz. Las métricas del diseño de la arquitectura consideran los aspectos estructurales del modelo de diseño. Las métricas de diseño al nivel de componentes indican la calidad del módulo al establecer medidas indirectas para la cohesión, el acoplamiento y la complejidad. Las métricas de diseño de la interfaz de usuario proporcionan un indicio de la facilidad con que se usa la interfaz gráfica del usuario.

Las métricas para los sistemas orientados a objetos se concentran en la medición que puede aplicarse a las características de clase y diseño (localización, encapsulamiento, ocultamiento de información, herencia y técnicas de abstracción de objetos) que convierten a la clase en única.

Halstead proporciona un conjunto interesante de métricas al nivel de código fuente. Empleando el número de operadores y operandos presentes en el código, se desarrolla una variedad de métricas para evaluar la calidad del programa.

Pocas métricas del producto se han propuesto para emplearlas directamente en las pruebas del software y en el mantenimiento. Sin embargo, muchas otras métricas del producto pueden aplicarse para guiar el proceso de prueba y como mecanismo para evaluar la facilidad de mantenimiento de un programa de cómputo. Una amplia variedad de métricas orientadas a objetos se ha propuesto para evaluar la facilidad de prueba de un sistema orientado a objetos.

Referencias

[ALB79] Albrecht, A. J., "Measuring Application Development Productivity", en *Proc. IBM Application Development Symposium*, Monterey, CA, octubre de 1979, pp. 83-92.

[ALB83] Albrecht, A. J. y J. E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation", en *IEEE Trans. Software Engineering*, noviembre de 1983, pp. 639-648.

[BAS84] Basili, V. R. y D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", en *IEEE Trans. Software Engineering*, vol. SE-10, 1984, pp. 728-738.

[BER95] Berard, E., "Metrics for Object-Oriented Software Engineering", publicación de Internet en comp.software-eng, 28 de enero, 1995.

[BIE94] Bieman, J. M. y L. M. Ott, "Measuring Functional Cohesion", en *IEEE Trans. Software Engineering*, vol. SE-20, núm. 8, agosto de 1994, pp. 308-320.

[BIN94] Binder, R. V., "Object-Oriented Software Testing", en *CACM*, vol. 37, num. 9, septiembre de 1994, p 29

[BRI96] Briand, L. C., S. Morasca y V. R. Basili, "Property-Based Software Engineering Measurement", en *IEEE Trans. Software Engineering*, vol SE-22, núm 1, enero de 1996, pp. 68-85

[CAR90] Card, D. N. y R. L. Glass, *Measuring Software Design Quality*, Prentice-Hall, 1990.

[CAV78] Cavano, J. P. y J. A. McCall, "A Framework for the Measurement of Software Quality" *Proc. ACM Software Quality Assurance Workshop*, noviembre de 1978, pp. 133-139.

[CHA89] Charette, R. N., *Software Engineering Risk Analysis and Management*, McGraw-Hill/Interlext, 1989

[CHI94] Chidamber, S. R. y C. F. Kemerer, "A Metrics Suite for Object-Oriented Design", en *IEEE Trans. Software Engineering*, vol. SE-20, núm 6, junio de 1994, pp 476-493

[CHI98] Chidamber, S. R., D. P. Darcy y C. F. Kemerer, "Management Use of Metrics for Object-Oriented Software: An Exploratory Analysis", en *IEEE Trans Software Engineering*, vol. SE-24, núm 8, agosto de 1998, pp. 629-639.

[CHU95] Churcher, N. I. y M. J. Shepperd, "Toward a Conceptual Framework for Object-Oriented Metrics", en *ACM Software Engineering Notes*, vol 20, núm 2, abril de 1995, pp 69-76.

[CUR80] Curtis, W., "Management and Experimentation in Software Engineering", en *Proc. IEEE*, vol 68, núm. 9, septiembre de 1980

[DAV93] Davis, A. *et al.*, "Identifying and Measuring Quality in a Software Requirements Specification", en *Proc First Intl Software Metrics Symposium*, IEEE, Baltimore, MD, mayo de 1993, pp. 141-152

[DEM81] DeMillo, R. A. y R. J. Lipton, "Software Project Forecasting", en *Software Metrics* (A. Perlis, F. G. Sayward y M. Shaw, eds.), MIT Press, 1981, pp. 77-89.

[DEM82] DeMarco, T., *Controlling Software Projects*, Yourdon Press, 1982

[DHA95] Dhama, H., "Quantitative Models of Cohesion and Coupling in Software", en *Journal of Systems and Software*, vol. 29, núm. 4, abril de 1995.

[EJI91] Ejiogu, L., *Software Engineering with Formal Metrics*, QED Publishing, 1991.

[FEL89] Felican, L. y G. Zalateu, "Validating Halstead's Theory for Pascal Programs", en *IEEE Trans Software Engineering*, vol SE-15, núm. 2, diciembre de 1989, pp 1630-1632.

[FEN91] Fenton, N., *Software Metrics*, Chapman and Hall, 1991

[FEN94] Fenton, N., "Software Measurement: A Necessary Scientific Basis", en *IEEE Trans Software Engineering*, vol SE-20, núm 3, marzo de 1994, pp. 199-206

[GRA87] Grady, R. B. y D. L. Caswell, *Software Metrics. Establishing a Company-Wide Program*, Prentice-Hall, 1987.

[HAL77] Halstead, M., *Elements of Software Science*, North-Holland, 1977.

[HAR98] Harrison, R., S. J. Counsell y R. V. Nithi, "An Evaluation of the MOOD set of Object-Oriented Software Metrics", en *IEEE Trans. Software Engineering*, vol. SE-24, núm. 6, junio de 1998, pp 491-496

[HET93] Hetzel, B., *Making Software Measurement Work*, QED Publishing, 1993

[IEE93] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1993

[IEE94] *Software Engineering Standards*, edición 1994, IEEE, 1994

[IFP01] *Function Point Counting Practices Manual*, versión 4.1.1, International Function Point Users Group, 2001, disponible en http://www.ifpug.org/publications/manual.htm

[IFP03] Function Point Bibliography/Reference Library, International Function Point Users Group, 2003, disponible en http://www.ifpug.org/about/bibliography.htm

[KOK95] Kokol, P., I. Rozman y V. Venuti, "User Interface Metrics", *ACM SIGPLAN Notices*, vol. 30, núm. 4, abril de 1995, puede descargarse de: http://portal.acm.org/

[KYB84] Kyburg, H. E., *Theory and Measurement*, Cambridge University Press, 1984.

[LET03] Lethbridge, T., comunicación privada sobre métricas de software, junio de 2003.

[LON02] Longstreet, D., "Fundamental of Function Point Analysis", Longstreet Consulting, Inc., 2002, disponible en http://www.ifpug.com/fpafund.htm

[LOR94] Lorenz, M. y J. Kidd, *Object-Oriented Software Metrics*, Prentice-Hall, 1994

[MCC76] McCabe, T. J., "A Software Complexity Measure", en *IEEE Trans. Software Engineering*, vol. SE-2, diciembre de 1976, pp. 308-320

[MCC77] McCall, J., P. Richards y G. Walters, "Factors in Software Quality", tres volúmenes, NTIS AD-A049-014, 015, 055, noviembre de 1977.

[MCC89] McCabe, T. J. y C. W. Butler, "Design Complexity Measurement and Testing", en *CACM*, vol. 32, núm. 12, diciembre de 1989, pp. 1415-1425

[MCC94] McCabe, T. J. y A. H. Watson, "Software en Complexity", en *Crosstalk*, vol. 7, núm. 12, diciembre de 1994, pp. 5-9

[NIE94] Nielsen, J. y J. Levy, "Measuring Usability: Preference vs. Performance", en *CACM*, vol. 37, núm. 4, abril de 1994, pp. 65-75

[ROC94] Roche, J. M., "Software Metrics and Measurement Principles", en *Software Engineering Notes*, ACM, vol. 19, núm. 1, enero de 1994, pp. 76-85.

[SEA93] Sears, A., "Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout", en *IEEE Trans. Software Engineering*, vol. SE-19, núm. 7, julio de 1993, pp. 707-719.

[SHE98] Sheppard, M., *Goal, Question, Metric*, 1998, disponible en http://dec.bournemouth.ac.uk/ESERG/mshepperd/SEMGQM.html

[SOL99] Van Solingen, R. y E. Berghout, *The Goal/Question/Metric Method*, McGraw-Hill, 1999.

[UEM99] Uemura, T., S. Kusumoto y K. Inoue, "A Function Point Measurement Tool for UML Design Specifications", en *Proc. Of Sixth International Symposium on Software Metrics*, IEEE, noviembre de 1999, pp. 62-69.

[USA87] *Management Quality Insight*, AFCSP 800-14 (U.S. Air Force), 20 de enero de 1987.

[WHI97] Whitmire, S., *Object-Oriented Design Measurement*, Wiley, 1997.

[WIL93] Wilde, N. y R. Huitt, "Maintaining Object-Oriented Software", en *IEEE Software*, enero de 1993, pp. 75-80.

[ZUS90] Zuse, H., *Software Complexity: Measures and Methods*, DeGruyter, 1990

[ZUS97] Zuse, H., *A Framework of Software Measurement*, DeGruyter, 1997.

PROBLEMAS Y PUNTOS A CONSIDERAR

15.1. La teoría de la medición es un tema avanzado que tiene una fuerte influencia sobre las métricas del software. Utilizando [ZUS97], [FEN91], [KYB84] u otras fuentes, escribir un breve ensayo que delinee los principales principios de la teoría de la medición. Proyecto individual: desarrollar una presentación sobre el tema y presentarla ante la clase.

15.2. Los factores de calidad de McCall se desarrollaron durante la década de 1970. Casi todos los aspectos de la computación han cambiado drásticamente desde que se desarrollaron; sin embargo, los factores de McCall siguen aplicándose al software moderno. ¿Podría llegarse a algunas conclusiones a partir de este hecho?

15.3. ¿Por qué no se puede desarrollar una sola métrica que lo abarque todo para la complejidad o la calidad de un programa?

15.4. Trátese de desarrollar una medida o una métrica tomada de la vida real que viole los atributos de las métricas efectivas del software que se definieron en la sección 15.2.5.

15.5. Un sistema tiene 12 entradas externas, 24 salidas externas, campos para 30 consultas externas diferentes, maneja cuatro archivos lógicos externos y tiene interfaces con seis sistemas heredados diferentes (6 AIE). Todos estos datos tienen una complejidad promedio, y el sistema general es relativamente simple. Calcúlese el punto de función para el sistema.

15.6. El software para el Sistema *X* tiene 24 requisitos funcionales individuales y 14 no funcionales. ¿Cuál es la especificidad de los requisitos? ¿En qué grado se ha completado?

15.7. Un importante sistema de información tiene 1140 módulos; 96 módulos realizan funciones de control y coordinación, y 490 dependen de un procesamiento anterior. El sistema procesa alrededor de 220 objetos de datos, cada uno con un promedio de tres atributos. Hay 140 elementos únicos de la base de datos y 90 segmentos diferentes de ésta. Por último, 600 módulos tienen puntos únicos de entrada y salida. Calcúlese el ICED del sistema.

15.8. Una clase, X, tiene 12 operaciones. Se ha calculado la complejidad ciclomática para todas las operaciones del sistema orientado a objetos, y el valor promedio de la complejidad del módulo es de 4. Para la clase X, la complejidad de la operación 1 a la 12 es 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4 y 4, respectivamente. Calcúlense los métodos ponderados por clase.

15.9. Desarróllese una herramienta de software que calcule la complejidad ciclomática de un módulo de lenguaje de programación. Elíjase el lenguaje que se desee.

15.10. Desarrolle una pequeña herramienta de software que realice un análisis de Halstead sobre código fuente en el lenguaje de programación que se desee.

15.11. Un sistema heredado tiene 940 módulos. La última versión requirió que 90 de esos módulos cambiaran. Además, se agregaron 40 nuevos módulos y se eliminaron 12 de esos módulos. Calcúlese el índice de madurez del software para el sistema.

Otras lecturas y fuentes de información

Hay un número sorprendentemente grande de libros dedicados a las métricas del software, aunque la mayor parte de ellos se concentra en las métricas del proceso y el proyecto, por lo que excluyen las métricas del producto. Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, segunda edición, 2002), Fenton y Pfleeger (*Software Metrics: A Rigourous and Practical Approach*, Brooks-Cole Publishing, 1998) y Zuse [ZUS97] han escrito tratamientos completos de las métricas del producto.

Libros de Card y Glass [CAR90], Zuse [ZUS90], Fenton [FEN91], Ejiogu [EJI91], Moeller y Paulish (*Software Metrics*, Chapman y Hall, 1993) y Hetzel [HET93] atienden las métricas del producto con algún detalle. Oman y Pfleeger (*Applyig Software Metrics*, IEEE Computer Society Press, 1997) han editado una antología de artículos importantes sobre las métricas del software. Además, vale la pena examinar los siguientes libros:

Conte, S. D., H. E. Dunsmore y V. Y. Shen, *Software Engineering Metrics and Models*, Benjamin-Cummings, 1984.

Grady, R. B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, 1992. Sheppard, M., *Software Engineering Metrics*, McGraw-Hill, 1992.

Denvir, Herman y Whitty presentan la teoría de la medición del software en una colección editada de artículos (*Proceedings of the International BCS-FACS Workshop: Formal Aspects of Measurement*, Springer-Verlag, 1992). Shepperd (*Foundations of Software Measurement*, Prentice Hall, 1996) también atiende con cierto detalle la teoría de la medición. El estado actual de la investigación se presenta en los *Proceedings of the Symposium on Software Metrics* (IEEE, publicados anualmente).

Un resumen muy completo de docenas de métricas de software útiles se presenta en [IEE93]. En general, un análisis de cada métrica se ha reducido a los "primitivos" (las medidas) esenciales necesarios para calcular la métrica y las relaciones apropiadas para realizar el cálculo. El apéndice proporciona un análisis y muchas referencias.

Whitmire [WHI97] presenta el tratamiento más completo y matemáticamente sofisticado de las métricas orientadas a objetos que se haya publicado a la fecha. Lorenz y Kidd [LOR94] y Henderson-Sellers (*Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, 1996) ofrecen un único libro adicional dedicado a las métricas orientadas a objetos. Hutcheson (*Software Testing Fundamentals: Methods and Metrics*, Wiley, 2003) presenta una guía útil para la aplicación y uso de métricas para la prueba del software.

Una amplia variedad de fuentes de información sobre métricas del software se encuentra disponible en Internet. Una lista actualizada de referencias en la World Wide Web relevantes para las métricas del software se encontrará en el sitio Web de SEPA:
http://www.mhhe.com/pressman.

PARTE

Tres

Aplicación de la ingeniería Web

En esta parte de *Ingeniería del software: un enfoque práctico*, se aprenderán los principios, conceptos y métodos con que se crean aplicaciones Web de alta calidad. Las siguientes preguntas se abordan en los capítulos posteriores:

- ¿Las aplicaciones Web (WebApps) son diferentes de otros tipos de software?
- ¿Qué es la ingeniería Web y qué elementos de la práctica de la ingeniería del software puede adoptar?
- ¿Cuáles son los elementos de un proceso de ingeniería Web?
- ¿Cómo se formula y planea un proyecto de ingeniería Web?
- ¿Cómo se analizan y modelan los requisitos de las WebApps?
- ¿Qué conceptos y principios guían la práctica en el diseño de las WebApps?
- ¿Cómo se dirigen la arquitectura, la interfase y el diseño de navegación de las WebApps?
- ¿Qué técnicas de construcción se pueden aplicar para implementar el modelo del diseño?
- ¿Qué conceptos, principios y métodos de prueba son aplicables a la ingeniería Web?

Una vez respondidas estas preguntas se estará mejor preparado para realizar la ingeniería de aplicaciones Web de alta calidad.

CAPÍTULO

16 INGENIERÍA WEB

CONCEPTOS CLAVE

La World Wide Web y la Internet que la alimentan son, posiblemente, los desarrollos más importantes en la historia de la computación. Estas tecnologías han llevado a todos (con cientos de millones más que eventualmente seguirán) a la era de la informática; además, se han convertido en parte integral de la vida diaria en la primera década del siglo XXI.

Para quienes pueden recordar un mundo sin la Web, el crecimiento caótico de la tecnología tiene su origen en otra era: los primeros días del software. Era una época de poca disciplina pero enorme entusiasmo y creatividad. Eran tiempos en que los programadores a menudo ingresaban a sistemas en conjunto, a veces para bien, a veces para mal. La actitud prevaleciente parecía ser: "hazlo rápido y entra en el campo; nosotros lo limpiaremos (y mejor entiende qué es lo que realmente se necesita construir) conforme avancemos". ¿Suena familiar?

En una mesa redonda virtual publicada en *IEEE Software* [PRE98], mantuve firme mi posición en relación con la ingeniería Web:

> Me parece que cualquier producto o sistema importante vale la pena una ingeniería. Antes de comenzar a construirla es mejor que entienda el problema, diseñe una solución factible, la implemente en una forma sólida y la ponga a prueba ampliamente. Tal vez también tenga que controlar los cambios conforme el trabajo avance y disponer de algún mecanismo para asegurar la calidad del resultado final. Muchos desarrolladores de Web no están de acuerdo con esto; ellos piensan que su mundo realmente es diferente y que los enfoques convencionales de ingeniería del software simplemente no se aplican.

UN VISTAZO RÁPIDO

¿Qué es? Los sistemas y aplicaciones basados en Web (WebApps) ofrecen un complejo arreglo de contenido y funcionalidad a una amplia población de usuarios finales. La ingeniería Web (IWeb) es el proceso con el que se crean WebApps de alta calidad. La IWeb no es un clon perfecto de la ingeniería del software, pero toma prestados muchos conceptos y principios fundamentales de ella. Además, el proceso IWeb acentúa actividades técnicas y administrativas similares. Existen sutiles diferencias en la manera como se dirigen dichas actividades, pero el método primordial dicta un enfoque disciplinado para el desarrollo de un sistema basado en la computadora.

¿Quién lo hace? Los ingenieros Web y los desarrolladores del contenido que no es técnico crean las WebApps.

¿Por qué es importante? Conforme las Web Apps se integran cada vez más en las estrategias de negocios para pequeñas y grandes empresas (por ejemplo, en el comercio electrónico), crece en importancia la necesidad de cons

...ruir sistemas confiables, prácticos y adaptables. Por tanto, es necesario un enfoque disciplinado en cuanto al desarrollo de WebApps

¿Cuáles son los pasos? Al igual que cualquier disciplina de ingeniería, la IWeb aplica un enfoque genérico que se suaviza mediante estrategias, tácticas y métodos especializados. El proceso IWeb comienza con una formulación del problema que se resolverá con la WebApp. Se planea el proyecto IWeb y se modelan los requisitos y el diseño de la WebApp. El sistema se construye con tecnologías y herramientas especializadas asociadas con la Web. Entonces se entrega a los usuarios finales y se evalúa mediante criterios tanto técnicos como empresariales. Dado que las WebApps evolucionan continuamente, se deben establecer mecanismos para el control de configuraciones, el aseguramiento de la calidad y el soporte continuo.

¿Cuál es el producto obtenido? Se producen muchos productos de trabajo IWeb. El resultado final es la WebApp operativa.

¿Cómo puedo estar seguro de que lo he hecho correctamente? En ocasiones es difícil estar seguro, hasta que los usuarios finales ejecutan la WebApp. Sin embargo, se aplican prácticas de aseguramiento de la calidad del software para valorar la calidad de los modelos IWeb, el contenido y la función globales del sistema, la facilidad de uso, el desempeño y la seguridad.

Esto conduce a una pregunta clave: *¿se pueden aplicar principios, conceptos y métodos de la ingeniería del software al desarrollo Web?* Es posible aprovechar muchos de ellos, pero su aplicación puede requerir un giro un tanto diferente.

¿Pero qué ocurre si persiste un enfoque sin disciplina respecto al desarrollo Web? En ausencia de un proceso disciplinado dirigido a desarrollar sistemas basados en Web, existe una creciente preocupación de que se enfrenten serios problemas en su desarrollo, despliegue y mantenimiento exitosos. En esencia, la infraestructura de aplicación que se está creando en la actualidad puede conducir a una "Web enmarañada" conforme se adentra más este nuevo siglo. Esta frase entraña un cúmulo de aplicaciones basadas en Web mal desarrolladas y que tienen muy altas probabilidades de fracaso. Peor aún, conforme los sistemas basados en Web crecen con mayor complejidad, una falla en uno puede propagar y propagará amplios problemas por medio de muchos. Cuando esto ocurra, la confianza en toda la Internet será sacudida. Peor aún, podría conducir a una regulación gubernamental innecesaria y mal concebida, lo que provocará un daño irreparable a estas tecnologías únicas.

Para evitar una Web enmarañada y lograr mayor éxito en el desarrollo y la aplicación de sistemas basados en Web complejos y a gran escala, existe una apremiante necesidad de enfoques disciplinados y nuevos métodos y herramientas con que desarrollar, desplegar y evaluar los sistemas y aplicaciones basados en Web. Tales enfoques y técnicas deben considerar las características especiales de los nuevos medios, los ambientes y escenarios operativos, y la multiplicidad de perfiles de usuario que colocan desafíos adicionales al desarrollo de aplicaciones basadas en Web.

La ingeniería Web (IWeb) aplica "sólidos principios científicos, de ingeniería y de administración, y enfoques disciplinados y sistemáticos para el desarrollo, despliegue y mantenimiento exitosos de sistemas y aplicaciones basados en Web de alta calidad" [MUR99]

16.1 Atributos de los sistemas y aplicaciones basados en Web

En los primeros días de la World Wide Web (*circa* 1990 a 1995) los "sitios Web" consistían en poco más de un conjunto de archivos de hipertexto ligados que presentaban información mediante texto y gráficos limitados. Conforme el tiempo pasó, el HTML aumentó al desarrollar herramientas (por ejemplo, XML, Java) que permitieron a los ingenieros Web ofrecer capacidades de cálculo junto con información. Nacieron los sistemas y aplicaciones[1] basados en Web (se les referirá de manera colectiva como *WebApps*). En la actualidad, las WebApps han evolucionado en sofisticadas herramientas de computación que no sólo proporcionan función por sí mismas al usuario final, sino que también se han integrado con bases de datos corporativas y aplicaciones de negocios.

> "En el momento en que veamos cierta especie de estabilización, la Web se habrá convertido en algo completamente diferente."
>
> **Louis Monier**

Existe poco debate en cuanto a que las WebApps *son* diferentes a las muchas otras categorías de software informático analizadas en el capítulo 1. Powell resume las diferencias principales cuando establece que los sistemas basados en Web "involucran una mezcla entre publicación impresa y desarrollo de software, entre marketing e informática, entre comunicaciones internas y relaciones externas, y entre arte y tecnología" [POW98]. En la gran mayoría de las WebApps se encuentran los siguientes atributos.

CONSEJO

Se puede argumentar que una aplicación tradicional dentro de cualquiera de los dominios de software tratados en el capítulo 1 puede mostrar esta lista de atributos. Sin embargo, las WebApps casi siempre lo hacen.

Intensidad de red. Una WebApp reside en una red y debe satisfacer las necesidades de una variada comunidad de clientes. Una WebApp puede residir en la Internet (y, en consecuencia, permitir una comunicación mundial abierta). Alternativamente, una aplicación puede colocarse en una Intranet (lo que implementa la comunicación en una organización) o en una Extranet (comunicación inter-red).

Concurrencia. Un gran número de usuarios puede tener acceso a la WebApp al mismo tiempo. En muchos casos, los patrones de uso entre los usuarios finales variarán enormemente.

Carga impredecible. El número de usuarios de la WebApp puede variar en órdenes de magnitud de día con día. El lunes pueden mostrarse 100 usuarios; el martes pueden usar el sistema 10 000.

1 En el contexto de este capítulo, el término "aplicación Web" (WebApp) abarca todo, desde una simple página Web que puede ayudar al consumidor a calcular el pago de arrendamiento de un automóvil, hasta un amplio sitio Web que proporcione servicios de viaje completos para gente de negocios y vacacionistas. Dentro de esta categoría se incluyen los sitios Web completos, la funcionalidad especializada dentro de los sitios Web y las aplicaciones de procesamiento de información que residen en la Internet o en una Intranet o Extranet.

Desempeño. Si un usuario de WebApp debe esperar demasiado (para ingresar, para procesamiento en el lado del servidor, para formateo y despliegue en el lado del cliente) puede decidir irse a cualquier otra parte

Disponibilidad. Aunque la expectativa de una disponibilidad del total es poco razonable, los usuarios de las WebApps populares con frecuencia demandan acceso sobre una base de "24/7/365". Los usuarios en Australia o Asia pueden demandar acceso durante momentos cuando las tradicionales aplicaciones de software doméstico en Norteamérica pueden estar fuera de línea por mantenimiento.

Gobernada por los datos. La función primordial de muchas WebApps es usar hipermedia para presentar contenido de texto, gráficos, audio y video al usuario final. Además, por lo general, las WebApps se utilizan para tener acceso a información que existe en bases de datos que originalmente no eran parte integral del ambiente basado en Web (por ejemplo, comercio electrónico o aplicaciones financieras).

Sensibilidad al contenido. La calidad y naturaleza estética del contenido sigue siendo un importante determinante de la calidad de una WebApp.

Evolución continua. A diferencia del software de aplicación convencional, que evoluciona a lo largo de una serie de planeadas liberaciones espaciadas cronológicamente, las aplicaciones Web evolucionan de manera continua. No es raro que algunas WebApps (específicamente, su contenido) se actualicen sobre una agenda minuto a minuto, o que el contenido sea calculado de manera independiente para cada solicitud Algunos argumentan que la evolución continua de las WebApps hace que el trabajo realizado sobre ellas sea análogo a la jardinería. Lowe [LOW99] comenta esto cuando escribe:

> La ingeniería trata de adoptar un enfoque consistente y científico, suavizado por un contexto práctico específico, para el desarrollo y comisionado de sistemas o aplicaciones Con frecuencia, el desarrollo de sitios Web se relaciona mucho con la creación de una infraestructura (sembrar el jardín) y luego con "cultivar" la información que crece y retoña dentro de este jardín. A lo largo del tiempo, el jardín (es decir, el sitio Web) continuará evolucionando, cambiando y creciendo Una buena arquitectura inicial debe permitir que este crecimiento ocurra en una forma controlada y consistente ..

El cuidado continuo y la alimentación permiten que un sitio Web crezca (en robustez e importancia) Pero, a diferencia del jardín, las aplicaciones Web deben satisfacer (y adaptarse a) las necesidades de alguien más que el jardinero.

Inmediatez. Aunque la *inmediatez* —la apremiante necesidad de poner software en el mercado rápidamente— es una característica de muchos dominios de aplicación, las WebApps con frecuencia muestran un tiempo para comercializar que puede ser cuestión de unos cuantos días o semanas.[2] Los ingenieros Web deben aplicar

2 Con las herramientas modernas se pueden producir elaboradas páginas Web en cuestión de unas cuantas horas

métodos de planeación, análisis, diseño, implementación y puesta a prueba que han sido adaptados a los apretados tiempos requeridos para el desarrollo de WebApps.

Seguridad. Puesto que las WebApps están disponibles mediante el acceso a la red, es difícil, si no imposible, limitar la población de usuarios finales que pueden tener acceso a la aplicación. Con la finalidad de proteger el contenido confidencial y ofrecer modos seguros de transmisión de datos, se deben implementar fuertes medidas de seguridad a lo largo de la infraestructura que sustenta una WebApp y dentro de la aplicación misma.

Estética. Una parte innegable de la apariencia de una WebApp es su presentación y la disposición de sus elementos. Cuando una aplicación se diseña para comercializar o vender productos o ideas, la estética puede tener tanto que ver con el éxito como el diseño técnico.

Estos atributos generales se aplican a todas las WebApps, pero con diferentes grados de influencia.

¿Pero qué hay de las WebApps por ellas mismas? ¿Qué problemas abordan? En el trabajo IWeb es usual encontrar las siguientes categorías de aplicaciones [DAR99]:

¿Qué categorías de WebApps se encuentran en el trabajo IWeb?

- *Informativo:* se proporciona contenido de sólo lectura con navegación y enlaces simples.
- *Descarga.* un usuario descarga información del servidor apropiado.
- *Personalizable:* el usuario personaliza el contenido según sus necesidades específicas.
- *Interacción:* la comunicación entre una comunidad de usuarios ocurre por medio de cuartos de charla, tableros de anuncios o mensajería instantánea.
- *Entrada del usuario:* la entrada con base en formularios es el principal mecanismo para las necesidades de comunicación.
- *Orientada a transacciones:* el usuario hace una solicitud (por ejemplo, realiza un pedido) que ejecuta la WebApp.
- *Orientada a servicios:* la aplicación proporciona un servicio al usuario; por ejemplo, lo asesora en la determinación del pago de una hipoteca.
- *Portal:* la aplicación canaliza al usuario hacia otro contenido o servicios Web fuera del dominio del portal de la aplicación.
- *Acceso a una base de datos:* el usuario consulta una gran base de datos y extrae información.
- *Almacén de datos:* el usuario consulta una colección de grandes bases de datos y extrae información.

Los atributos comentados en esta sección, y las categorías de aplicación destacadas líneas arriba, representan importantes hechos de vida para los ingenieros Web. La clave es vivir dentro de las restricciones que imponen dichos atributos y aun así producir una WebApp exitosa.

16.2 Estratos de la ingeniería de WebApp

El desarrollo de sistemas y aplicaciones basados en Web incorpora modelos de proceso especializados, métodos de ingeniería del software adaptados a las características del desarrollo de WebApps y un conjunto de importantes tecnologías habilitadoras. Los procesos, métodos y tecnologías (herramientas) proporcionan un enfoque en estratos de la IWeb que es conceptualmente idéntico a los estratos de la ingeniería del software descritos en la figura 2.1.

> "La ingeniería Web trata con enfoques disciplinados y sistemáticos para el desarrollo, despliegue y mantenimiento de los sistemas y aplicaciones basados en Web."
>
> Yogesh Deshpande

16.2.1 Proceso

CONSEJO

...Web con ... es ágil y ... es incre... Sin embargo, ... que el modelo ... puede elegirse ... mayoría de los ... de inge... Web.

Los modelos de procesos IWeb (que se tratan con detalle en la sección 16.3) adoptan la filosofía del desarrollo ágil (capítulo 4). El desarrollo ágil enfatiza un enfoque de desarrollo riguroso que incorpora rápidos ciclos de desarrollo. Aoyama [AOY98] describe la motivación para el enfoque ágil en la siguiente forma:

> Internet cambió la prioridad principal del desarrollo de software de *qué a cuándo*. El reducido tiempo para el mercado se ha convertido en el límite competitivo por el que luchan las compañías líderes. En consecuencia, reducir el ciclo de desarrollo es ahora una de las misiones más importantes de la ingeniería del software

Aun cuando rápidos ciclos de tiempo dominan la reflexión acerca del desarrollo, es importante reconocer que el problema todavía debe analizarse, debe desarrollarse un diseño, la implementación debe proceder en una forma incremental y se debe iniciar un enfoque organizado de prueba. Sin embargo, dichas actividades del marco de trabajo se deben definir dentro de un proceso que 1) adopte el cambio, 2) aliente la creatividad y la independencia del equipo de desarrollo y fortalezca la interacción con los accionistas de la WebApp, 3) construya sistemas que utilicen pequeños equipos de desarrollo, y 4) subraye el desarrollo evolutivo o incremental mediante el uso de cortos ciclos de desarrollo [MCD01].

16.2.2 Métodos

El panorama de los métodos de IWeb abarca un conjunto de labores técnicas que permiten al ingeniero Web comprender, caracterizar y luego construir una WebApp de alta calidad. Los métodos de IWeb (que se tratan con detalle en los capítulos 18 al 20) se pueden categorizar de la siguiente manera:

Métodos de comunicación: definen el enfoque con que se facilita la comunicación entre ingenieros Web y los demás participantes de la WebApp (por ejemplo, usuarios finales, clientes de negocios, expertos en problemas de dominio, diseñadores de contenido, líderes de equipo, gestores de proyecto). Las técnicas de comunicación son particularmente importantes durante la recolección de requisitos y siempre que sea evaluado un incremento en la WebApp

CONSEJO

Es importante notar que muchos métodos IWeb se han adoptado directamente de sus contrapartes de ingeniería del software. Otros están en sus etapas formativas. Algunos de estos sobrevivirán; otros serán descartados conforme se sugieran mejores enfoques.

Métodos de análisis de requisitos: proporcionan una base para comprender el contenido que entregará una WebApp, la función que proporcionará al usuario final y los modos de interacción que cada clase de usuario requerirá mientras ocurra la navegación por medio de la WebApp

Métodos de diseño: abarcan una serie de técnicas de diseño que abordan el contenido, la aplicación y la arquitectura de información, así como el diseño de interfase y la estructura de navegación de la WebApp.

Métodos de prueba: incorporan revisiones técnicas formales —tanto del contenido y el modelo de diseño como de una amplia variedad de técnicas de prueba que abordan conflictos al nivel de componente y arquitectónicos—, pruebas de la navegación, pruebas de facilidad de uso, pruebas de seguridad y pruebas de configuración.

Es importante señalar que, aunque los métodos IWeb adoptan muchos de los mismos conceptos y principios subyacentes a los métodos de ingeniería del software descritos en la parte 2 de este libro, los mecanismos de análisis, diseño y prueba deben adaptarse para acomodar las características especiales de las WebApps.

Además de los métodos técnicos que se han subrayado, es esencial una serie de actividades sombrilla (con métodos asociados) para la ingeniería Web exitosa. Ésta incluye técnicas de gestión de proyecto (por ejemplo, estimación, calendarización, análisis de riesgo), técnicas de gestión de configuración de software y de revisión

16.2.3 Herramientas y tecnología

Referencia Web

Se encuentran excelentes recursos para tecnología IWeb en webdeveloper.com y en www.abercom.com/webtracker.

A lo largo de la década pasada ha evolucionado un amplio conjunto de herramientas y tecnología conforme las WebApps se han vuelto más complejas y extendidas. Dichas tecnologías abarcan un amplio conjunto de descripción de contenido y lenguajes de modelación (por ejemplo, HTML, VRML, XML), lenguajes de programación (por ejemplo, Java), recursos de desarrollo basados en componentes (por ejemplo, CORBA, COM, ActiveX, .NET), navegadores, herramientas multimedia, herramientas de autoría de sitio, herramientas de conectividad de bases de datos, herramientas de seguridad, servidores y utilidades de servidor, y herramientas de administración y análisis de sitio.

Un tratamiento completo de las herramientas y tecnología para la ingeniería Web está más allá del ámbito de este libro. El lector interesado puede visitar uno o más de los siguientes sitios Web: *Web Developer's Virtual Encyclopedia* (www.wdlv.com), *WebDeveloper* (www.webdeveloper.com), *Developer Shed* (www.devshed.com), *Webknowhow.net* (www.webknowhow.net) o *WebReference* (www.webreference.com)

16.3 EL PROCESO DE INGENIERÍA WEB

Los atributos de los sistemas y aplicaciones basados en Web tienen una profunda influencia sobre el proceso de IWeb que se elija. En el capítulo 3 se hizo notar que un ingeniero de software elige un modelo de proceso basado en los atributos del soft

ware que habrá de desarrollarse. Esta premisa también es cierta para un ingeniero Web.

Si la inmediatez y la evolución continua son atributos principales de una WebApp, un equipo de ingeniería Web debe elegir un modelo de proceso ágil (capítulo 4) que produzca liberaciones de WebApp a un ritmo vertiginoso. Por otra parte, si una WebApp será desarrollada durante un largo periodo (por ejemplo, una gran aplicación de comercio electrónico) puede elegirse un modelo de proceso incremental (capítulo 3).

> "El desarrollo Web es un adolescente . . . al igual que la mayoría de los adolescentes, quiere ser aceptado como un adulto conforme intenta alejarse de sus padres. Si quiere alcanzar todo su potencial, debe tomar unas cuantas lecciones del más experimentado mundo del desarrollo de software."
>
> **Doug Wallace *et al.***

La intensa naturaleza de las aplicaciones de la red en este dominio sugiere una diversa población de usuarios (que, por lo tanto, realizan demandas especiales acerca de respuesta y modelado de requisitos) y una arquitectura de aplicación que puede ser altamente especializada (que en consecuencia realiza demandas acerca del diseño). Puesto que con frecuencia las WebApps son conductoras de contenido, con énfasis en la estética, es probable que se proyecten actividades de desarrollo paralelas dentro del proceso IWeb e involucren un equipo de personal tanto técnico como lego (por ejemplo, publicistas, diseñadores gráficos)

16.3.1 Definición del marco de trabajo

Cualquiera de los modelos de proceso ágil (por ejemplo, Programación Extrema, Desarrollo de Software Adaptativo, SCRUM) presentados en el capítulo 4 se pueden aplicar de manera exitosa como un proceso IWeb. El marco de trabajo del proceso que se presenta aquí es una amalgama de los principios e ideas tratados en dicho capítulo.

La efectividad de cualquier proceso de ingeniería depende de su adaptabilidad. Esto es, la organización del equipo de proyecto, los modos de comunicación entre miembros del equipo, las actividades de ingeniería y las tareas que deben realizarse, la información que se recolecte y cree, y los métodos empleados para producir un producto de alta calidad deben estar adaptados a la gente que realiza el trabajo, el plazo y las restricciones del proyecto, y al problema que se quiere resolver. Antes de definir un marco de trabajo de proceso para IWeb se debe reconocer que

1. *Las WebApps con frecuencia se entregan de manera incremental.* Esto es, las actividades del marco de trabajo ocurrirán de manera repetida conforme cada incremento se someta a ingeniería y se entregue.
2. *Los cambios ocurrirán frecuentemente.* Estos cambios pueden ocurrir como resultado de la evaluación de un incremento entregado o como consecuencia de cambiar las condiciones de los negocios.

CLAVE
de proceso
expone en
: entrega
, cambio
y plazos

3. *Los plazos son cortos.* Esto aminora la creación y revisión de voluminosa documentación de ingeniería, pero no excluye la simple realidad de que el análisis crítico, el diseño y la prueba deben registrarse en alguna forma.

Además, se deben aplicar los principios definidos como parte del "Manifiesto para el desarrollo de software ágil" (capítulo 4). Sin embargo, los principios no son los diez mandamientos. A veces es razonable adoptar el espíritu de dichos principios sin que sea necesario atenerse a la letra del manifiesto.

Con estos conflictos en mente se aborda el proceso de IWeb dentro del proceso genérico de marco de trabajo presentado en el capítulo 2.

PUNTO CLAVE

El modelo de proceso genérico (introducido en el capítulo 2) es aplicable a la ingeniería Web

Comunicación con el cliente. Dentro del proceso IWeb la comunicación con el cliente se caracteriza por medio de dos grandes tareas: el análisis del negocio y la formulación. El *análisis del negocio* define el contexto empresarial-organizativo para la WebApp. Además, se identifican los participantes, se predicen los potenciales cambios en el ambiente o los requisitos del negocio, y se define la integración entre la WebApp y otras aplicaciones de negocios, bases de datos y funciones. La *formulación* es una actividad de recopilación de requisitos que involucra a todos los participantes. El intento es describir el problema que la WebApp habrá de resolver (junto con los requisitos básicos para la WebApp) con el aprovechamiento de la mejor información disponible. Además, se intenta identificar áreas de incertidumbre y dónde ocurrirán cambios potenciales.

Planeación. Se crea el plan del proyecto para el incremento de la WebApp. El plan consiste de una definición de tareas y un calendario de plazos respecto al periodo (usualmente medido en semanas) proyectado para el desarrollo del incremento de la WebApp.

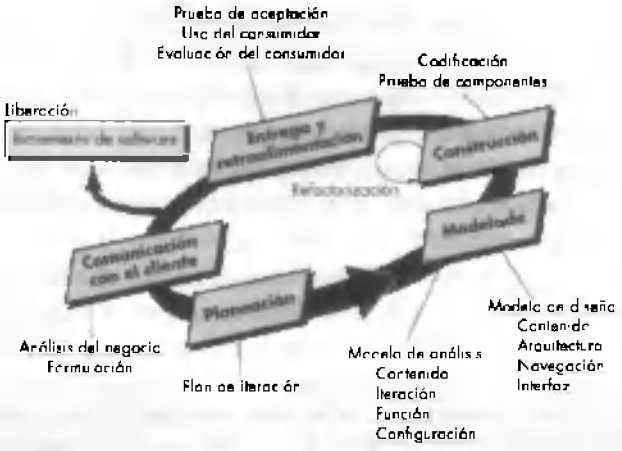
Modelado. Las labores convencionales de análisis y diseño de ingeniería del software se adaptan al desarrollo de la WebApp, se mezclan y luego se funden en la actividad de modelado IWeb (capítulos 18 y 19). El intento es desarrollar análisis "rápidos" y modelos de diseño que definan requisitos y al mismo tiempo representen una WebApp que los satisfará.

Construcción. Las herramientas y la tecnología IWeb se aplican para construir la WebApp que se ha modelado. Una vez que se construye el incremento de WebApp se dirige una serie de pruebas rápidas para asegurar que se descubran los errores en el diseño (es decir: contenido, arquitectura, interfase, navegación). Pruebas adicionales abordan otras características WebApp.

Despliegue. La WebApp se configura para su ambiente operativo, se entrega a los usuarios finales y luego comienza un periodo de evaluación. La retroalimentación acerca de la evaluación se presenta al equipo de IWeb y el incremento se modifica conforme se requiera.

Estas cinco actividades del marco de trabajo IWeb se aplican empleando un flujo de proceso incremental, como se muestra en la figura 16.1.

Figura 16.1



INFORMACIÓN

Ingeniería Web: preguntas básicas

La ingeniería de cualquier producto involucra sutilezas que no advierten inmediatamente quienes carecen de experiencia sustancial. Las características de las WebApps llevan a los ingenieros Web a responder una diversidad de preguntas que deben abordarse durante las primeras actividades del marco de trabajo. Las preguntas estratégicas relacionadas con las necesidades del negocio y los objetivos del producto, se tratan durante la formulación. Las preguntas acerca de los requisitos, relacionadas con las características y funciones, deben considerarse durante el análisis de modelado. Las preguntas específicas de diseño base, relacionadas con la arquitectura de la WebApp, las características de la interfaz y los conflictos de navegación, se consideran conforme evoluciona el modelo de diseño. Finalmente, un conjunto de conflictos humanos, relacionados con la forma en la que un usuario realmente interactúa con la WebApp, se abordan en forma continua.

Susan Weinshenk [WEI02] sugiere un conjunto de preguntas que se deben considerar conforme progresan el análisis y el diseño. Aquí se anota un pequeño subconjunto (adaptado):

- ¿Cuán importante es la página de inicio (*home page*) de un sitio Web? ¿Debe contener información útil o una simple lista de enlaces que conduzcan al usuario a mayores detalles en niveles inferiores?
- ¿Cuál es la plantilla de página más efectiva (por ejemplo, menú arriba, a la derecha, a la izquierda) y ésta variará según el tipo de WebApp que se desarrollará?
- ¿Qué opciones de medios audiovisuales tienen más impacto? ¿Los gráficos son más efectivos que el texto? ¿El video (o el audio) es una opción efectiva? ¿Cuándo se deben elegir varias opciones de medios audiovisuales?
- ¿Cuánto trabajo se puede esperar que realice un usuario cuando busca información? ¿Cuántos clics desea hacer la gente?
- ¿Cuán importantes son los auxiliares de navegación cuando las WebApps son complejas?
- ¿Cuán complejas pueden ser las entradas de formulario antes de que se vuelvan irritantes para el usuario? ¿Cómo se pueden expedir las entradas de formulario?
- ¿Cuán importantes son las capacidades de búsqueda? ¿Qué porcentaje de usuarios navega y qué porcentaje usa búsquedas específicas? ¿Cuán importante es estructurar cada página en una forma que suponga un enlace desde alguna fuente externa?
- ¿La WebApp se diseñará en una forma que sea accesible a quienes tengan discapacidades físicas o de algún otro tipo?

No existen respuestas absolutas a preguntas como éstas, e incluso<Ninguno> deben abordarse conforme avance la IWeb. En los capítulos 17 al 20 se considerarán respuestas potenciales.

16.3.2 Refinamiento del marco de trabajo

Ya se ha advertido que el modelo del proceso IWeb debe ser adaptable. Esto es, la definición de las tareas de ingeniería requeridas para refinar cada actividad del marco de trabajo se dejan a discrecional juicio del equipo de ingeniería Web. En algunos casos, una actividad del marco de trabajo se dirige de manera informal; en otros, se definirá una serie de distintas tareas y las dirigirán miembros del equipo. En todo caso, el equipo es responsable de producir un incremento WebApp de alta calidad dentro del periodo acordado.

Es importante destacar que las tareas asociadas con las actividades del marco de trabajo IWeb pueden modificarse, eliminarse o extenderse con base en las características del problema, el producto, el proyecto y la gente en el equipo de ingeniería Web

> "Existen algunos de nosotros que creen que las mejores prácticas para el desarrollo de software son prácticas y merecen implementación. Y luego existen algunos de nosotros que creen que las mejores prácticas son interesantes en cierta forma académica, mas no lo son para el mundo real, muchas gracias."
>
> **Warren Keuffel**

16.4 Mejores prácticas en ingeniería Web

¿Todo desarrollador de WebApp utilizará el marco de trabajo y el conjunto de tareas del proceso IWeb definido en la sección 16.3? Probablemente no. En ocasiones, los equipos de ingeniería Web están sometidos a enorme presión respecto del tiempo y tratarán de tomar atajos (incluso si éstos son imprudentes e implican *más* esfuerzo de desarrollo, en lugar de menos) Pero se debe aplicar un conjunto fundamental de mejores prácticas —adoptado de las prácticas de ingeniería del software tratadas a lo largo de la Parte 2 de este libro— si se han de construir WebApps con calidad industrial.

1. *Tomar tiempo para entender las necesidades del negocio y los objetivos del producto, incluso si los detalles de la WebApp son vagos.* Muchos desarrolladores de WebApps creen erróneamente que los requisitos vagos (que son bastante comunes) los liberan de la necesidad de asegurarse de que el sistema que están a punto de someter a ingeniería tenga un propósito empresarial legítimo El resultado final es (también con frecuencia) un buen trabajo técnico que conduce a la construcción del sistema equivocado por las razones equivocadas para el público equivocado. Si los accionistas no pueden enunciar una necesidad empresarial para la WebApp, debe procederse con extrema precaución. Si los accionistas luchan por identificar un conjunto de objetivos claros para el producto (WebApp), no debe procederse mientras ellos no concluyan.
2. *Describir cómo interactuarán los usuarios con la WebApp aplicando un enfoque basado en escenarios.* Se debe convencer a los accionistas para desarrollar casos de uso (tratados a lo largo de la Parte 2 de este libro) para reflejar cómo los diversos actores interactuarán con la WebApp. Entonces se pueden apro-

CONSEJO

Asegúrese de que alguien haya enunciado con claridad las necesidades del negocio para una WebApp. Si no es así, el proyecto de IWeb está en riesgo.

vechar dichos escenarios 1) para la planeación y el rastreo del proyecto, 2) para guiar el análisis y el modelado del diseño, y 3) como una entrada importante para el diseño de pruebas.

3. *Desarrollar un plan del proyecto, incluso si es muy breve.* El plan debe basarse en un proceso de marco de trabajo predefinido aceptable para todos los participantes. Puesto que los plazos del proyecto son muy cortos, la dosificación del programa debe ser exacta; es decir, en muchas instancias el proyecto debe planearse y rastrearse diariamente.
4. *Utilizar algún tiempo para modelar lo que se construirá.* Por lo general, el análisis total y los modelos de diseño no se desarrollan durante la ingeniería Web. Sin embargo, la clase UML y los diagramas de secuencia, junto con otra notación UML seleccionada (por ejemplo, diagramas de estado), pueden proporcionar una visión invaluable.
5. *Revisar la consistencia y calidad de los modelos.* Las revisiones técnicas formales (capítulo 26) se deben dirigir a lo largo del proyecto IWeb. El tiempo empleado en las revisiones paga importantes dividendos porque usualmente elimina reelaboraciones y resulta en una WebApp que exhibe alta calidad, lo que aumenta la satisfacción del cliente.
6. *Utilizar herramientas y tecnología que permitan construir el sistema con tantos componentes reutilizables como sea posible.* Un amplio conjunto de herramientas WebApp están a disposición virtualmente para cada aspecto de la construcción WebApp. Muchas de dichas herramientas permiten que un ingeniero Web construya porciones significativas de la aplicación empleando componentes reutilizables
7. *No apoyarse en usuarios anteriores para depurar la WebApp; diséñense pruebas amplias y ejecútense antes de liberar el sistema.* Los usuarios de una WebApp con frecuencia le dan una oportunidad Si falla en su ejecución se mueven a cualquiera otra parte: nunca regresan. Por esta razón, el "pruebe primero, después despliegue" debe ser un sistema primordial, incluso si los plazos se deben prolongar

INFORMACIÓN

Criterios de calidad/directrices para WebApps

La IWeb se esfuerza en la producción de WebApps de alta calidad Pero, en este contexto, ¿qué es calidad y qué directrices están disponibles para lograrla? Un artículo acerca de aseguramiento de la calidad en sitios Web, Quibeldey-Cirkel [QUI01] sugiere un amplio conjunto de recursos en línea que abordan estos conflictos:

W3C: guía de estilo para hipertexto en línea
www.w3.org/Provider/Style
La Guía Sevloid para el diseño Web
www.sev.com.au/webzone/design/guide.asp
Páginas Web que Apestan
www.webpagesthatsuck.com/index.html

Recursos acerca de estilo Web
www.westegg.com/unmaintained/badpages
Herramienta de evaluación Web de Gartner
www.gartner.com/ebusiness/website-ings
IBM Corp: directrices Web
www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/572
Facilidad de uso en la World Wide Web
ijhcs.open.ac.uk
Interfaz Salón de la Vergüenza
www.iarchitect.com/mshame.htm
El arte y el zen de los sitios Web
www.tlc-systems.com/webtips.shtml
Diseño para la Web. estudios empíricos
www.microsoft.com/usability/webconf.htm
useit.com de Nielsen
www.useit.com
Calidad de experiencia
www.qualityofexperience.org
Creación de sitios Web asesinos
www.killersites.com/core.html
Todas las cosas en la Web
www.pantos.org/atw
Nuevo diseño Web de SUN
www.sun.com/980113/sunonnet
Tognazzini, Bruce: homepage
www.asktog.com
Webmonkey
hotwired.lycos.com/webmonkey/design/?tw=design
Los mejores sitios Web del mundo
www.worldbestwebsites.com
Yale University. guía de estilo Web de Yale
info.med.yale.edu/caim/manual

16.5 RESUMEN

Es posible argumentar que el impacto de los sistemas y aplicaciones basados en Web es el suceso más significativo en la historia de la computación. Conforme la importancia de las WebApps crece ha comenzado a evolucionar un enfoque IWeb disciplinado (adaptado de los principios, conceptos, procesos y métodos de la ingeniería de software).

Las WebApps son diferentes de otras categorías de software informático; son eminentemente de red, las gobiernan los datos y se encuentran en evolución continua. La inmediatez que dirige su desarrollo, la necesidad apremiante de seguridad en su operación y la demanda de estética, así como la entrega de contenido funcional, son factores diferenciales adicionales. Al igual que otros tipos de software, las WebApps pueden valorarse mediante una diversidad de criterios de calidad que incluyen facilidad de uso, funcionalidad, confiabilidad, eficiencia, capacidad de mantenimiento, seguridad, disponibilidad, escalabilidad y tiempo para comercialización.

La IWeb se describe en tres estratos: proceso, métodos y herramientas/tecnología. El proceso IWeb adopta el enfoque de desarrollo ágil que subraya un punto de vista de ingeniería "magro", riguroso, que conduce a la entrega incremental del sistema que será construido. El proceso genérico del marco de trabajo —comunicación, planeación, modelado, construcción y despliegue— es aplicable a la IWeb. Dichas actividades del marco de trabajo se refinan en un conjunto de tareas IWeb que se adaptan a las necesidades de cada proyecto. A todos los proyectos IWeb se les aplica un conjunto de actividades sombrilla similar al aplicado durante el trabajo de ingeniería del software: SQA, SCM, gestión del proyecto.

REFERENCIAS

[AOY98] Aoyama, M., "Web-Based Agile Software Development", en *IEEE Computer*, noviembre-diciembre, 1998, pp. 56-65.

[DAR99] Dart, S., "Containing the Web Crisis Using Configuration Management", en *Proc. First ICSE Workshop on Web Engineering*, ACM, Los Ángeles, mayo de 1999. (*The Proceedings of the First ICSE Workshop on Web Engineering* se publican en línea en http://fistserv.macarthur.uws.edu.au/san/icse99-WebE/ICSE99-WebE-Proc/default.htm).

[FOW01] Fowler, M. y J. Highsmith, "The Agile Manifesto", en *Software Development Magazine*, agosto de 2001, http://www.sdmagazine.com/documents/s=844/sdm0108a/0108a.htm

[MCD01] McDonald, A. y R. Welland, *Agile Web Engineering (AWE) Process*, Department of Computer Science, University of Glasgow, Technical Report TR-2001-98, 2001, se puede descargar desde http://www.dcs.gla.ac.uk/~andrew/TR-2001-98.pdf.

[MUR99] Murugesan, S., *WebE Home Page*, http://fistserv.macarthur.uws.edu.au/san/WebEHome, julio de 1999.

[NOR99] Norton, K., "Applying Cross Functional Evolutionary Methodologies to Web Development", en *Proc. First ICSE Workshop on Web Engineering*, ACM, Los Ángeles, mayo de 1999.

[POW98] Powell, T. A., *Web Site Engineering*, Prentice-Hall, 1998.

[PRE98] Pressman, R. S. (moderador), "Can Internet-Based Applications Be Engineered?", *IEEE Software*, septiembre de 1998, pp. 104-110.

[QUI01] Quibeldey-Cirkel, K., "Checklist for Web Site Quality Assurance", en *Quality Week Europe*, 2001, se puede descargar desde www.fbi.fh-darmstadt.de/~quibeldey/Projekte/QWE-2001/Paper_Quibeldey_Cirkel.pdf.

[WEI02] Weinschenk, S., "Psychology and the Web: Designing for People", 2002, http://www.weinschenk.com/learn/facts.asp.

PROBLEMAS Y PUNTOS A CONSIDERAR

16.1 ¿Existen otros atributos genéricos que diferencien a las WebApps de las aplicaciones de software más convencionales? Inténtese mencionar dos o tres.

16.2 ¿Cómo juzga el lector la "calidad" de un sitio Web? Hágase una lista, en orden descendente de prioridad, de 10 atributos de calidad que consideren los más importantes.

16.3 Realizar un poco de investigación y escribir un artículo de dos a tres páginas que resuma una de las tecnologías anotadas en la sección 16.2.3.

16.4 Empleando un sitio Web real como ejemplo, ilustrar las diferentes manifestaciones del "contenido" de la WebApp.

16.5 Revisar los procesos de ingeniería del software descritos en los capítulos 3 y 4. ¿Existe(n) algún(os) otro(s) proceso(s) –distinto(s) al modelo de proceso ágil– que pueda(n) ser aplicable(s) a la ingeniería Web? Si la respuesta es afirmativa, indicar cuál(es) proceso(s) y por qué.

16.6 Revisar la exposición del "Manifiesto para desarrollo de software ágil" presentado en el capítulo 4. ¿Cuál de los 12 principios funcionaría bien para un proyecto de dos años (que involucra a docenas de personas) que construirá un gran sistema de comercio electrónico para una compañía automotriz? ¿Cuál de los 12 principios funcionaría bien para un proyecto de dos meses que construirá un sitio informativo para una pequeña firma de bienes raíces?

16.7 Elaborar una lista de "riesgos" que serían probables durante el desarrollo de una nueva aplicación de comercio electrónico que se diseña para vender teléfonos celulares y servicios directamente por medio de la Web.

Otras lecturas y fuentes de información

En años recientes se han publicado cientos de libros que analizan uno o más temas de ingeniería Web, aunque relativamente pocos abordan todos los aspectos de la IWeb. Sarukkai (*Foundations of Web Technology*, Kluwar Academic Publishers, 2002) presenta una valiosa compilación de tecnologías que se requieren para la IWeb. Murugusan y Deshpande (*Web Engineering: Managing Diversity and Complexity of Web Development*, Springer-Verlag, 2001) han editado una colección de útiles artículos acerca de IWeb. Las actas de conferencias internacionales acerca de ingeniería Web y de ingeniería de sistemas de información Web las publica anualmente el IEEE Computer Society Press.

Flor (*Web Business Engineering*, Addison-Wesley, 2000) analiza el análisis de negocios y las preocupaciones relacionadas que permiten al ingeniero Web comprender mejor las necesidades de los clientes. Bean (*Engineering Global E-commerce Sites*, Morgan Kaufman, 2003) presenta directrices para el desarrollo de WebApps globales. Lowe y Hall (*Hypermedia and the Web: An Engineering Approach*, Wiley, 1999) y Powell [POW98] ofrecen una cobertura razonablemente completa. Umar (*Application Re-engineering: Building Web-Based Applications and Dealing with Legacy Systems*, Prentice-Hall, 1997) aborda uno de los más difíciles conflictos en la IWeb: la reingeniería de los sistemas heredados para hacerlos compatibles con los sistemas basados en Web. IEEE Std 2001-1999 define prácticas básicas de IWeb.

En Internet hay disponible una gran variedad de fuentes de información acerca de ingeniería Web. En el sitio Web de SEPA se puede encontrar una lista actualizada de referencias en World Wide Web que son relevantes para la ingeniería Web: **http://www.mhhe.com/pressman.**

CAPÍTULO

17

FORMULACIÓN Y PLANEACIÓN PARA INGENIERÍA WEB

Durante la tempestuosa década de 1990, el *boom* de la Internet generó más arrogancia que cualquier otro evento en la historia de las computadoras. Los desarrolladores de WebApps en cientos de jóvenes compañías punto-com argumentaban que había surgido un nuevo paradigma para el desarrollo de software, que las viejas reglas ya no se aplicarían más, que el tiempo para el mercado dominaba todas las demás preocupaciones. Se rieron de la noción de que la formulación y la planeación cuidadosas debían ocurrir antes de que comenzara la construcción. ¿Y quién podía rebatirlos? El dinero estaba en todas partes, los jóvenes de 24 años se volvieron multimillonarios (al menos en el papel); tal vez las cosas realmente habían cambiado. Y entonces el suelo se vino abajo.

Conforme comenzaba el siglo xxi empezó a ser dolorosamente evidente que un enfoque de "construyelo y ellos vendrán" simplemente no funcionaba, que la formulación del problema es esencial para garantizar que una WebApp en verdad es necesaria, y que la planeación vale el esfuerzo, aun cuando el calendario de desarrollo sea apretado. Constantine y Lockwood [CON02] advierten esta situación cuando escriben:

> A pesar de las declaraciones radicales de que la Web representa un nuevo paradigma definido por reglas nuevas, los desarrolladores profesionales se están dando cuenta de que las lecciones acerca del desarrollo de software, aprendidas en los días previos al Internet, todavía se aplican. Las páginas Web son interfaces de usuario, la programación HTML es programación, y las aplicaciones desplegadas en el navegador son sistemas de software que pueden beneficiarse de los principios básicos de la ingeniería del software.

Entre los principios fundamentales de la ingeniería de software destaca el de: *comprender el problema antes de comenzar a resolverlo, y estar seguro de que la solución concebida es aquella que la gente realmente quiere.* Esta es la base de la formulación, la primera gran actividad en la ingeniería Web. Otro principio fundamental de la ingeniería de software es: *planear el trabajo antes de comenzar a realizarlo.* Este es el enfoque que subyace a la planeación de proyectos.

UN VISTAZO RÁPIDO

¿Qué es? Comunicación, planeación y modelado. Por una parte, existe una tendencia a diferir, a esperar hasta que toda *i* esté cruzada y toda *i* tenga punto antes de que comience el trabajo. Por otra parte, hay un deseo de saltar ya, de comenzar a construir incluso antes de que en realidad se conozca qué se necesita hacer. Ambos enfoques son inapropiados y por ello las dos primeras actividades del marco de trabajo de la ingeniería Web destacan la formulación y la planeación. La formulación valora la necesidad subyacente de la WebApp, las características y funciones globales que desean los usuarios

y el ámbito del esfuerzo de desarrollo. La planeación aborda los elementos que deben definirse para establecer un flujo de trabajo y un programa, y a rastrear el trabajo conforme avanza el proyecto.

¿Quién lo hace? Los ingenieros Web, sus administradores y los participantes sin funciones técnicas; todos participan en la formulación y la planeación.

¿Por qué es importante? Es difícil viajar a un lugar que nunca se ha visitado sin direcciones o un mapa. Eventualmente se llega (o tal vez no), pero con seguridad el viaje será frustrante y largo en forma innecesaria. La formulación y la planeación proporcionan un mapa para un equipo de ingeniería Web.

¿Cuáles son los pasos a seguir? La formulación comienza al establecer comunicación con el consumidor (accionista) que plantea las razones para la WebApp: ¿cuál es la necesidad del negocio, cuáles usuarios finales son el objetivo, qué características y funciones se desean, qué sistemas y bases de datos existentes van a tener acceso, el concepto es realizable, cómo se medirá el éxito? La planeación establece un plan de trabajo, desarrolla estimaciones para valorar la factibilidad de las fechas de entrega deseadas, considera riesgos, define un programa y establece mecanismos para rastreo y control.

¿Cuál es el producto obtenido? Puesto que el trabajo de ingeniería Web con frecuencia adopta una filosofía ágil, los productos obtenidos para la formulación y la planeación usualmente son parcos, pero existen y deben registrarse en forma escrita. La recopilación de información durante la formulación se registra en un documento escrito en el cual se basan la planeación y el modelado de análisis. El plan del proyecto extiende el programa de éste y presenta cualquier otra información que sea necesario comunicar a los miembros del equipo de ingeniería Web y al personal externo.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Es necesario desarrollar con suficiente detalle para establecer un mapa sólido, pero no tanto como para quedar empantanado. La información de la formulación y la planeación debe revisarse con los clientes para garantizar que las inconsistencias y las omisiones se identifiquen en una etapa temprana.

17.1 Formulación de sistemas basados en Web

La *formulación* de sistemas y aplicaciones basados en Web representa una secuencia de acciones de ingeniería Web que comienza con la identificación de las necesidades del negocio, se mueve hacia una descripción de los objetivos de la WebApp, define grandes características y funciones y realiza la recopilación de requisitos que conducen al desarrollo de un modelo de análisis. La formulación permite que los clientes y el equipo de ingeniería Web establezcan un conjunto común de metas y objetivos para la construcción de la WebApp. También identifica el ámbito de esfuerzo de desarrollo y proporciona un medio para determinar un resultado exitoso. El análisis —una actividad técnica que es una continuación de la formulación— identifica los requisitos funcionales, de comportamiento y de datos para la WebApp.

Antes de considerar la formulación con más detalle, es razonable preguntar dónde termina la formulación y dónde comienza el análisis de requisitos. No existe una respuesta sencilla para esta pregunta. La formulación se enfoca sobre el "gran cuadro": en las necesidades y objetivos del negocio y en la información relacionada. Sin embargo, virtualmente es imposible mantener este grado de abstracción. Los clientes y los ingenieros Web quieren definir el contenido requerido, discutir la fun-

cionalidad específica, enumerar características específicas e identificar la forma en que los usuarios finales interactuarán con la WebApp. ¿Esto es formulación o recopilación de requisitos? Ambos es la respuesta

17.1.1 Preguntas de formulación

Powell [POW98] sugiere un conjunto de preguntas que deben formularse y responderse al comienzo de la etapa de formulación.

- ¿Cuál es la principal motivación (necesidades del negocio) para la WebApp?
- ¿Cuáles son los objetivos que debe satisfacer la WebApp?
- ¿Quién usará la WebApp?

La respuesta a cada una de estas simples preguntas debe establecerse tan sucintamente como sea posible. Por ejemplo, supóngase que el fabricante de *HogarSeguro*[1] ha decidido establecer un sitio Web de comercio electrónico para vender sus productos directamente a los consumidores. Un enunciado que describa la motivación para la WebApp puede ser:

> HogarSeguroInc.com permitirá a los consumidores configurar y comprar todos los componentes requeridos para instalar un sistema de administración en su hogar o empresa

CONSEJO

...s comienza la ...n del ..., inténtese ...r la WebApp ...e pretende ...ar en un solo ...lo. Si no es ..., no se están ...o las ...s globales del ...

Es importante advertir que en este enunciado no se proporcionan detalles. El objetivo aquí es acotar la intención global de la WebApp y colocarla en un contexto empresarial legítimo.

Después de platicar con varios clientes se establece una respuesta a la segunda pregunta:

> HogarSeguroInc.com nos permitirá vender directamente a los consumidores, lo que eliminará los costos de intermediación y mejorará los márgenes de utilidad. También nos permitirá aumentar las ventas en un proyectado 25 por ciento sobre las ventas anuales actuales y penetrar en regiones geográficas donde en la actualidad no tenemos puntos de venta.

Finalmente, la compañía define la demografía para la WebApp: "Los usuarios proyectados de HogarSeguroInc com son los propietarios de viviendas y los dueños de pequeños negocios."

Las respuestas establecidas líneas arriba implican metas específicas para el sitio Web de HogarSeguroInc.com. En general, se identifican dos categorías de metas [GNA99]:

- *Metas informativas*: indican una intención de proporcionar contenido información específicos al usuario final
- *Metas aplicables*: indican la habilidad para realizar alguna tarea dentro de la WebApp

1 El producto *HogarSeguro* ya se usó como ejemplo a lo largo de las partes 1 y 2 de este libro

En el contexto de la WebApp HogarSeguroInc.com, una meta informativa puede ser

> El sitio proporcionará a los usuarios especificaciones de producto detalladas, que incluirán descripciones técnicas, instrucciones de instalación e información de precios.

El examen de las respuestas a las preguntas planteadas puede conducir al establecimiento de una meta aplicable:

> HogarSeguroInc.com consultará al usuario acerca de la instalación (es decir· casa, oficina, espacio de venta al menudeo) que será protegida y realizará recomendaciones personalizadas acerca del producto y la configuración que se utilizará.

Una vez identificadas todas las metas informativas y aplicables, se desarrolla un perfil de usuario. El perfil de usuario captura "características relevantes relacionadas con los usuarios potenciales, que incluye sus antecedentes, escolaridad, preferencias e incluso más" [GNA99]. En el caso de HogarSeguroInc.com, un perfil de usuario identificaría las características de un comprador típico de sistemas de seguridad (esta información la suministraría el departamento de mercadotecnia).

> "Si estás hackeando [WebApps], probablemente tu enfoque es 'preparen, fuego, apunten' Si estás comprometido con hacerlas funcionar, debe ser 'preparen, apunten, fuego'."
>
> **Autor desconocido**

Una vez que se han desarrollado las metas y perfiles de usuario, la actividad de formulación se enfoca sobre una afirmación del ámbito para la WebApp En muchos casos, las metas ya desarrolladas se integran en la afirmación del ámbito. Además es útil, no obstante, indicar el grado de integración esperado de la WebApp. Esto es con frecuencia es necesario integrar los sistemas de información existentes (por ejemplo, una aplicación existente de base de datos) con un planteamiento basado en Web Los temas de conectividad se consideran en esta etapa

17.1.2 Recopilación de requisitos para WebApps

Los métodos para la recopilación de requisitos se trataron en el capítulo 7. Aunque esta actividad puede abreviarse para la ingeniería Web, los objetivos globales de la recopilación de requisitos propuestos para la ingeniería de software permanecen inalterados. Adaptados para las WebApp, dichos objetivos se convierten en:

- Identificar requisitos de contenido
- Identificar requisitos funcionales.
- Definir escenarios de interacción para diferentes clases de usuarios.

Los siguientes pasos de la recopilación de requisitos se dirigen para lograr estos objetivos:

¿Qué pasos de recopilación de requisitos se aplican para las WebApps?

1. Pedir a los clientes que definan las categorías de usuario y describan cada categoría.
2. Comunicarse con los clientes para definir los requisitos básicos de la WebApp

3. Analizar la información recopilada y utilizar la información para realizar un seguimiento con los clientes.
4. Definir casos de uso (capítulo 8) que describan escenarios de interacción para cada clase de usuario.

Definición de las categorías de usuario. Se puede argumentar que la complejidad de la WebApp es directamente proporcional al número de categorías de usuario para el sistema. La definición de una categoría de usuario requiere formular un conjunto de preguntas fundamentales:

- *¿Cuál es el objetivo global del usuario cuando usa la WebApp?* Por ejemplo, un usuario del sitio de comercio electrónico de HogarSeguroInc.com puede estar interesado en recopilar información acerca de productos de administración del hogar. Otro usuario tal vez desee comparar precios. Un tercer usuario quiere comprar el producto *HogarSeguro*. Cada uno representa una clase o categoría diferente de usuario; cada uno tendrá diferentes necesidades y navegará a través de la WebApp de manera diferente. Un cuarto usuario ya posee *HogarSeguro* y busca soporte técnico o quiere comprar sensores o accesorios adicionales
- *¿Cuáles son los antecedentes y la pericia del usuario en relación con el contenido y la funcionalidad de la WebApp?* Si un usuario tiene un antecedente técnico y una pericia significativa el contenido o la funcionalidad elementales ofrecerán poco beneficio. De manera alternativa, un neófito demanda contenido y funcionalidad elementales y estaría confundido si se perdiese.
- *¿Cómo llegará el usuario a la WebApp?* ¿El arribo ocurrirá a través de un enlace desde otro sitio Web (probablemente hacia contenido o funcionalidad dentro de la WebApp), o llegará en una forma más controlada?
- *¿Qué características genéricas de la WebApps le gustan o disgustan al usuario?* Diferentes tipos de usuarios pueden tener distintos y predecibles gustos y aversiones. Vale la pena el intento de determinar si los tienen o no. En muchas situaciones la respuesta se puede averiguar preguntándoles cuáles son sus WebApps favorita y menos favorita.

Al aprovechar las respuestas a estas preguntas se debe definir el más pequeño conjunto razonable de clases de usuario. Conforme se avanza en la recopilación de requisitos, cada diferente clase de usuario debe encuestarse para obtener datos.

HOGARSEGURO

Recopilación de requisitos para WebApps

El escenario: la oficina de Doug

Los actores: Doug Miller, gerente del equipo de ingeniería del software; Vinod Raman,

equipo de ingeniería del software *HogarSegura*; más tarde, tres personas de mercadotecnia.

La conversación:

Doug: La gerencia ha decidido que construyamos un sitio de comercio electrónico para vender *HogarSegura*

Vinod: ¡Guau, Doug! No tenemos tiempo para hacer eso . estamos empantanados con el trabajo de software de producto

Doug: Lo sé, lo sé. subcontrataremos el desarrollo con una compañía especializada en la construcción de sitios de comercio electrónico. Ellos nos dirán que lo tendrán listo y corriendo en menos de un mes... muchos componentes reutilizables.

Vinod: Mmmmm Bien .. ¿entonces por qué estoy aquí?

Doug: Para facilitar las cosas: quieren que nos encarguemos de la recopilación de requisitos para el sitio Quiero que te entrevistes con los diversos clientes para comprender, aunque sea en forma mínima, los requisitos básicos.

Vinod (exasperado): Doug... no me estás escuchando.. estamos apretados de tiempo y esto .

Doug (interrumpe): Sólo dale un día de tu tiempo, Vinod. Entrevístate con los tipos de mercadotecnia y hazlos que especifiquen el contenido básico, la función; tú sabes, el procedimiento usual.

Vinod (resignado): Está bien. Los llamaré y concertaré algo para mañana, pero no me facilites la vida

Doug (sonríe): Por eso te llevas los billetes grandes.

Vinod: Cierto.

[Vinod se entrevista con tres personas de mercadotecnia al día siguiente.]

Vinod: Me decían acerca de los objetivos y antecedentes de los usuarios

Persona de mercadotecnia #1: Como dije, pretendemos que el usuario sea capaz de personalizar todo el sistema *HogarSegura*. Tú sabes: escoger sensores, paneles de control, características y funciones, luego obtener una "cuenta de materiales" generada automáticamente, obtener la cotización y luego comprar el sistema a través del sitio Web.

Persona de mercadotecnia #2: Suponemos que el usuario es propietario de una casa, no un técnico, así que necesitamos guiarlo a través del proceso paso a paso

Persona de mercadotecnia #3: Yo no soy técnico, pero me preocupan los elementos especializados que necesitamos elaborar además de los factores básicos de comercio electrónico

Vinod (enfrentando al #3): ¿Qué quieres decir?

Persona de mercadotecnia #3: La parte difícil será guiar al usuario a través del "proceso de personalización" en una forma simple y completa El asunto del comercio electrónico real es bastante directo.

Persona de mercadotecnia #1: Tendremos que ofrecer un número 800 para las personas que no estén dispuestas a realizar la personalización por sí mismas

Persona de mercadotecnia #2: Estoy de acuerdo

Vinos: Muy bien, tendremos que hablar acerca de cómo les gustaría exactamente hacer la personalización del producto como una actividad de preventas, pero dejemos eso por un momento Tengo otras cuantas preguntas fundamentales

Vinod (ve a la persona de mercadotecnia #2): Dijiste que querías guiar a los usuarios a través del proceso ¿Algún enfoque especial?

Persona de mercadotecnia #2: Quisiera ver un proceso paso a paso, con espacios en blanco para responder preguntas de requisitos básicos, menús desplegables, ese tipo de cosas. Cada paso es una ventana, y los datos de cada ventana se validan antes de moverse al paso siguiente

Vinod: ¿Has comprobado eso con usuarios representativos?

Persona de mercadotecnia #2: No, pero lo haré

Vinod: Una cosa más. . ¿cómo encontrará nuestro sitio un usuario?

Persona de mercadotecnia #1: Estamos trabajando en una campaña publicitaria que colocará www.HogarSeguroInc.com en anuncios de revistas, correo dirigido a objetivos, anuncios sensibles a contenido que aparecen en los motores de búsqueda, y tal vez incluso algunos spots de radio y televisión

Vinod: Lo que quiero decir es.. los usuarios siempre entran a través de la página inicial

Persona de mercadotecnia #3: Eso es lo que nos gustaría.

Vinod: Muy bien, ahora tenemos que ponernos a trabajar. Exploremos los detalles de cómo quieren personalizar los sistemas en línea.

Comunicación con los clientes y usuarios finales. La mayoría de las WebApps tiene una amplia población de usuarios finales. Aunque la creación de categorías o clases de usuario hace que una evaluación de los requisitos de usuario sea más manejable, no es recomendable emplear información recopilada sólo de una o dos personas como la base para la formulación o el análisis. Se deben considerar más personas (y más opiniones y puntos de vista).

La comunicación se puede lograr aprovechando uno o más de los mecanismos siguientes [FUC02a]

¿Qué mecanismos de comunicación es [illegible] emplear en el trabajo de [illegible]?

- *Grupo muestral tradicional.* Un moderador entrenado se reúne con un pequeño (usualmente menos de 10 personas) grupo representativo de usuarios finales (o participantes internos que los representan). El propósito es discutir la WebApp que se desarrollará y, fuera de la discusión, comprender mejor los requisitos del sistema
- *Grupo muestral electrónico.* Un debate electrónico moderado dirigido con un grupo de usuarios finales y participantes representativos. El número de participantes puede ser mayor. Puesto que todos los usuarios pueden participar al mismo tiempo, es posible recopilar más información en un periodo más corto. Dado que todo el debate se basa en texto es automático un registro contemporáneo
- *Entrevistas iterativas.* Una serie de entrevistas breves, dirigida a usuarios representativos y en la que se solicitan respuestas a preguntas específicas acerca de la WebApp, se dirige a través del sitio Web o mediante correo electrónico Las respuestas se analizan y aprovechan para afinar la entrevista siguiente.
- *Entrevistas de exploración.* Encuesta basada en Web y ligada a una o más WebApps con usuarios similares a los que usarán la WebApp que se desarrollará. Los usuarios se enlazan a la entrevista y responden una serie de preguntas (usualmente reciben alguna recompensa por participar).
- *Construcción de escenarios* A usuarios seleccionados se les pide crear casos de uso informales que describan interacciones específicas con la WebApp

Análisis de la información recopilada. Conforme se recopila información se categoriza en clase de usuario y tipo de transacción, y luego se valora según su relevancia. El objetivo es desarrollar listas de objetos de contenido, operaciones que se aplican a los objetos de contenido dentro de una transacción de usuario específica, funciones (por ejemplo, informativa, computacional, lógica y orientada a la ayuda) que la WebApp proporciona a los usuarios finales, y otros requisitos no funcionales que se advierten durante las actividades de comunicación.

[CONSEJO]

[illegible] demorar la [illegible] de los [illegible] y operaciones [illegible] mientras [illegible]nce el [illegible] del análisis. [illegible] punto es más [illegible]e la recopila- [illegible] información, [illegible] evaluación.

Fuccella y Pizzolato [FUC02b] sugieren un método simple (de baja tecnología: *low-tech*) para comprender cómo se deben organizar el contenido y la funcionalidad. Se crea un paquete de "cartas" para los objetos de contenido, las operaciones aplicadas a los objetos de contenido, la funciones WebApp y otros requisitos no funcio-

nales. Se barajan las cartas y luego se distribuyen a las personas representativas de cada categoría de usuario. Se pide a los usuarios que ordenen las cartas en grupos que reflejan cómo les gustaría que se organizara el contenido y la funcionalidad dentro de la WebApp. Luego se solicita a los usuarios que describan cada agrupamiento y las razones por las que son importantes para ellos. Una vez que cada usuario realiza este ejercicio, el equipo de ingeniería Web busca agrupamientos comunes entre diferentes clases de usuarios y otros agrupamientos que sean únicos para una clase de usuario específica.

El equipo IWeb desarrolla una lista de etiquetas que se usarán para apuntar la información dentro de cada uno de los agrupamientos derivados con el uso de los paquetes de cartas. Entonces, a los diferentes usuarios representativos se les dan los paquetes de cartas y se les pide ubicar el contenido y la funcionalidad a cada una de las etiquetas. Aquí el propósito es determinar cuándo las etiquetas (enlaces dentro de la WebApp real) implican de manera adecuada el acceso al contenido y las funciones que los usuarios esperan encontrar detrás de la etiqueta. Este paso se aplica de manera iterativa hasta que se alcanza el consenso.

> "Si lo que estás haciendo no lo puedes describir como un proceso, entonces no sabes lo que estás haciendo."
> **W. E. Deming**

CONSEJO

En la Parte 2 de este libro se trataron con detalle los casos de uso. Aunque muchos abogan por el desarrollo de casos de uso muy largos, incluso una narración informal proporciona algún beneficio. Convenza a los usuarios para que escriban casos de uso.

Desarrollo de casos de uso. Los casos de uso[2] describen cómo interactuará con la WebApp una categoría de usuario específica (llamada *actor*) para lograr una acción específica. La acción puede ser tan simple como adquirir contenido definido o tan compleja como que el usuario realice un análisis detallado de registros seleccionados que se mantienen en una base de datos en línea. Los casos de uso describen la interacción desde el punto de vista del usuario.

Aunque desarrollarlos y analizarlos toma tiempo, los casos de uso 1) ayudan al desarrollador a entender cómo perciben los usuarios su interacción con la WebApp; 2) proporcionan el detalle necesario para crear un modelo de análisis efectivo; 3) ayudan a dividir en compartimientos el trabajo de IWeb; y 4) ofrecen una guía importante para quienes deben probar la WebApp.

CONJUNTO DE TAREAS

La comunicación con el cliente (Análisis/Formulación)

1. *Identifíquense a los clientes del negocio.* ¿Exactamente quién es el "cliente" de la WebApp? ¿Qué personas de negocios pueden funcionar como expertos y usuarios finales representativos? ¿Quién participará como miembro activo del equipo?
2. *Formúlese el contexto del negocio.* ¿Cómo encaja la WebApp en una estrategia de negocios más amplia?
3. *Definanse las metas y objetivos clave del negocio para la WebApp.* ¿Cómo se medirá el éxito de la WebApp, tanto en términos cualitativos como cuantitativos?

2 En los capítulos 7 y 8 se presentaron con detalle las técnicas para desarrollar los casos de uso.

Definanse las metas informativos y aplicables. ¿Qué clase de contenido se proporcionará a los usuarios finales? ¿Qué funciones/tareas se lograrán cuando se use la WebApp?

Identifíquese el problema ¿Qué problema específico resuelve la WebApp?

Recopílense requisitos ¿Qué tareas del usuario se lograrán mediante el uso de la WebApp? ¿Qué contenido se desarrollará? ¿Qué metáfora de interacción se usará? ¿Qué funciones computacionales proporcionará la WebApp? ¿Cómo se configurará la WebApp para su utilización en red? ¿Qué esquema de navegación se desea?

CONSEJO

...de proyectos ...se puede ...una simple ...de datos de ..." (con la ...de una hoja ...en lugar ...UML. Esto ...a todos los ...del equipo ...los requi- ...el contenido ...entregado ...mejor el ...flujo de ...que ocurrirá.

17.1.3 El puente hacia el modelado de análisis

Como ya se ha señalado en este capítulo, las actividades que conducen a un equipo de ingeniería Web de la formulación al modelado de análisis representa un continuo. En esencia, el grado de abstracción considerado durante las primeras etapas de la formulación es la estrategia del negocio. Sin embargo, conforme la formulación se lleva a cabo, se analizan los detalles tácticos y se abordan los requisitos específicos de la WebApp. Finalmente, estos requisitos se modelan (con la utilización de casos de uso y notación UML).

Los conceptos y principios tratados para el análisis de requisitos de software (capítulos 7 y 8) se aplican sin revisión para la actividad de análisis de ingeniería Web. Durante el análisis se elabora el ámbito definido durante la actividad de formulación para crear un modelo de análisis completo para la WebApp. En la IWeb se realizan cuatro tipos diferentes de análisis: del contenido, de la interacción, de la función y de la configuración. En el capítulo 18 se exponen las tareas y técnicas de modelado asociadas con cada uno de estos análisis.

> "Al fracasar para preparar, se prepara para el fracaso."
>
> **Benjamin Franklin**

17.2 PLANEACIÓN DE PROYECTOS DE INGENIERÍA WEB

Dada la inmediatez de las WebApps es razonable preguntar: ¿en realidad se necesita gastar tiempo en la planeación y administración de un esfuerzo WebApp? ¿No sólo se debería dejar evolucionar naturalmente a la WebApp, con poca o ninguna gestión explícita? Más de un desarrollador Web optaría por poca o ninguna gestión, ¡pero eso no hace que estén en lo correcto!

Referencia Web

...que trata en la gestión de ...electrónicos.

La figura 17.1 presenta un cuadro adaptado de Kulik y Samuelson [KUL00] que indica cómo los "proyectos electrónicos" (e-projects, su término para los proyectos WebApp) se comparan con los proyectos de software tradicionales. Al consultar la figura, los proyectos de software tradicionales y los grandes proyectos electrónicos tienen similitudes sustanciales. Dado que la gestión del proyecto se indica para los proyectos tradicionales, parecería razonable argumentar que también estaría indicada para los grandes proyectos electrónicos. Los pequeños proyectos electrónicos tienen características especiales que los diferencian de los proyectos tradicionales. Sin embargo, incluso en el caso de los pequeños proyectos electrónicos, la planea-

FIGURA 17.1 Diferencias entre proyectos tradicionales y electrónicos (e-projects) [adaptado de KUL00]

	Proyectos tradicionales	Pequeños proyectos electrónicos	Grandes proyectos electrónicos
Recopilación de requisitos	Riguroso	Limitada	Riguroso
Especificaciones técnicas	Robustas: modelos, especificaciones	Panorama descriptivo	Robusta: modelos UML, especificaciones
Duración del proyecto	Medida en meses o años	Medida en días, semanas o meses	Medida en meses o años
Prueba y aseguramiento de la calidad	Enfocada en lograr blancos de calidad	Enfocada sobre control de riesgo	Aseguramiento de la calidad del software como se describe en el capítulo 26
Gestión de riesgos	Explícita	Inherente	Explícita
Vida media de los entregables	8 meses o más	De 3 a 6 meses o más corto	De 6 a 12 meses o más corto
Proceso de liberación	Riguroso	Expedito	Riguroso
Retroalimentación del cliente después de la liberación	Requiere esfuerzo proactivo	Se obtiene automáticamente de la interacción con el usuario	Se obtiene tanto de manera automática como por medio de solicitud de retroalimentación

ción se debe realizar, se deben considerar los riesgos, se debe establecer un prog- ma y se deben definir controles de modo que eviten la confusión, la frustración y fracaso.

17.3 EL EQUIPO DE INGENIERÍA WEB

Un equipo de ingeniería Web exitoso mezcla una amplia variedad de talentos deben trabajar como equipo en un ambiente de proyecto con alta presión. Los pl zos son cortos, los cambios son inexorables y la tecnología continúa cambiando creación de un equipo que se consolide (véase el capítulo 21) no es asunto senc.

> "En el mundo actual, alimentado por la Web y centrado en la red, uno necesita saber mucho de muchos temas."
>
> **Scot Tilley y Shihaog Huang**

17.3.1 Los actores

La creación de una aplicación Web exitosa demanda un amplio abanico de habil des. Tilley y Huang [TIL99] abordan este tema cuando afirman: "Existen tantos c rentes aspectos del software de aplicación [a la Web] que se ha dado el (re)surgimie del renacentista, aquel que se siente cómodo trabajando en varias disciplinas Aunque los autores están en lo correcto, los "renacentistas" son relativam pocos, y dadas las demandas asociadas con los grandes proyectos de desarroll WebApps, el conjunto de diversas habilidades requeridas puede ser mejor distri do entre un equipo de ingeniería Web.

Los equipos de ingeniería Web se pueden organizar, en gran medida, en la m forma que los equipos de software tradicionales (capítulo 21). Sin embargo, los a

res y sus papeles usualmente son bastante diferentes. Entre las muchas habilidades que se deben distribuir entre los miembros del equipo IWeb se encuentran: ingeniería del software basada en componentes, realización de redes, diseño arquitectónico y de navegación, lenguajes/estándares de Internet, diseño de interfase humana, diseño gráfico, disposición del contenido y pruebas de las WebApps.

Los siguientes papeles[3] se deben distribuir entre los miembros del equipo IWeb:

¿Qué papel desempeña el cliente en un equipo de IWeb?

Desarrolladores/proveedores de contenido. Dado que el contenido controla inherentemente las WebApps, una función del equipo IWeb se debe enfocar en la generación o recopilación del contenido. Recuérdese que el contenido abarca un amplio abanico de objetos de datos, por ello los desarrolladores/proveedores de contenido pueden provenir de diversos ámbitos (no de software).

Editores de web. El variado contenido que generan los respectivos desarrolladores/proveedores se debe organizar para incluirlo en la WebApp. Además, alguien debe actuar como conexión entre el equipo técnico que diseña la WebApp y los desarrolladores/proveedores de contenido sin conocimientos técnicos. Este papel lo satisface el *editor de Web*, quien debe entender tanto el contenido como la tecnología de la WebApp.

Ingeniero Web. El ingeniero Web se involucra en un amplio rango de actividades durante el desarrollo de una WebApp, que incluyen la obtención de requisitos, el modelado de análisis, el diseño arquitectónico, de navegación y de interfase, la implementación de la WebApp y las pruebas. El ingeniero Web también debe tener una sólida comprensión de las tecnologías de componentes, de las arquitecturas cliente/servidor, de HTML/XML y de tecnologías de bases de datos, y un conocimiento práctico de los conceptos multimedia, de las plataformas hardware/software, de la seguridad de redes y de cuestiones de apoyo a sitios Web.

Expertos en dominios empresariales. Un experto en dominio empresarial debe ser capaz de responder todas las preguntas relacionadas con metas, objetivos y requisitos empresariales relacionados con la WebApp.

Especialista de soporte. Este papel se asigna a la persona (personas) que es (son) responsable(s) del apoyo continuo a la WebApp. Puesto que las WebApps evolucionan continuamente, el especialista de soporte es responsable de las correcciones, adaptaciones y mejoras al sitio, que incluyen actualizaciones de contenido, implementación de nuevos procedimientos y formas, y cambios al patrón de navegación.

Administrador. Usualmente llamado "web master", esta persona tiene la responsabilidad de la operación diaria de la WebApp, lo que incluye: desarrollo e implementación de políticas para la operación de la WebApp, establecimiento de procedimientos de soporte y retroalimentación, implementación de seguridad y derechos de acceso, medición y análisis de tráfico del sitio Web, coordinación de los procedimientos de control de cambios (capítulo 27) y coordinación con el especialista de

3 Estos papeles se han adaptado de Hansen y sus colegas [HAN99].

soporte. El administrador también puede estar involucrado en las actividades técnicas que realizan los ingenieros Web y los especialistas de soporte.

17.3.2 Construcción del equipo

CONSEJO

Estas características son usuales en los equipos de colaboradores autoorganizados que han adoptado un enfoque ágil (capítulo 4). Mientras mejor sea el equipo, mejor será el producto de software que se produzca

En el capítulo 21 se tratarán con cierto detalle los lineamientos para la construcción exitosa de los equipos de ingeniería del software. Pero, ¿estos lineamientos se aplican en el apretujado mundo de los proyectos WebApp? La respuesta es sí.

Hace algún tiempo, en su éxito de librería acerca de la industria de la computación, Tracy Kidder [KID00] cuenta la historia del heroico intento de una compañía de computación por construir una computadora para enfrentar el reto de un nuevo producto que fabricó un competidor más grande.[4] La historia es una metáfora del equipo de trabajo, del liderazgo y del aplastante estrés que todos los tecnólogos encuentran cuando los proyectos críticos no avanzan tan suavemente como se planeó.

Un resumen del libro de Kidder difícilmente le hace justicia, pero los siguientes puntos clave [PIC01] tienen particular relevancia cuando una organización construye un equipo de ingeniería Web:

Se debe establecer un conjunto de directrices de equipo. Dichas directrices abarcan lo que se espera de cada persona, cómo se lidiará con los problemas y qué mecanismos existen para mejorar la efectividad del equipo conforme avanza el proyecto.

El liderazgo fuerte es una obligación. El líder del equipo debe guiar mediante el ejemplo y el contacto. Debe mostrar un grado de entusiasmo que impulse a los otros miembros del equipo "a endosarse" psicológicamente al trabajo que enfrentan.

El respeto hacia los talentos individuales es crucial. Nadie es bueno en todo. Los mejores equipos utilizan las fortalezas individuales. Los mejores líderes de equipo permiten que los individuos tengan libertad para seguir una buena idea.

Cada miembro del equipo se debe comprometer. El protagonista principal en el libro de Kidder le llama a esto "endoso".

Es fácil comenzar, lo difícil es mantener el ímpetu. Los mejores equipos nunca dejan que un problema "insuperable" los detenga. Los miembros del equipo desarrollan una solución "lo suficientemente buena" y proceden, con la esperanza de que el ímpetu del progreso pueda conducirlos a una solución todavía mejor en el largo plazo.

17.4 Conflictos de gestión de proyecto para ingeniería Web

Una vez realizada la formulación y que se han identificado los requisitos básicos de la WebApp, la empresa debe elegir una de dos opciones de ingeniería Web. La WebApp es *subcontratada* (outsourced): la ingeniería Web la realiza un tercer pro-

4 El libro de Kidder, *The Soul of a New Machine*, originalmente publicado en 1981, ¡es una lectura altamente recomendable para cualquiera que intente realizar una carrera en la computación y para quienes ya la tienen!

dor con experiencia, talento y recursos con los cuales no cuente la empresa; o 2) la WebApp la desarrollan *en casa* ingenieros Web que sean empleados de la empresa. Una tercera opción (hacer algún trabajo de ingeniería Web en casa y subcontratar otro trabajo) también es una posibilidad.

> "Como observó Thomas Hobbs en el siglo xvii, la vida bajo las reglas de las pandillas es solitaria, pobre, peligrosa, cruel y corta. La vida en un proyecto de software que corre pobremente; es solitaria, pobre, peligrosa, cruel y con dificultad alguna vez es lo suficientemente corta."
>
> **Steven McConnell**

El trabajo que debe realizarse sigue siendo el mismo sin importar si una WebApp es subcontratada, desarrollada en casa o distribuida entre un proveedor externo y el equipo de casa. No obstante, sí cambian los requisitos de comunicación, la distribución de actividades técnicas, el grado de interacción entre clientes y desarrolladores, y una diversidad de otros conflictos crucialmente importantes.

La figura 17.2 ilustra, respecto a las WebApps, la diferencia organizacional entre subcontratación y desarrollo en casa. Éste (figura 17.2a) integra directamente todos los miembros del equipo de ingeniería Web (el círculo punteado implica integración). La comunicación se establece mediante los caminos de la organización. En cuanto a la subcontratación (figura 17.2b), es impráctico y desaconsejable que cada elemento de casa (por ejemplo, desarrolladores de contenido, accionistas, ingenieros Web internos) tenga comunicación directa con el subcontratista sin que exista algún elemento de conexión para coordinar y controlar la comunicación. En las secciones que siguen se examinarán con más detalle las planeaciones para la subcontratación y el desarrollo en casa.

FIGURA 17.2 Relaciones organizativas para subcontratación y desarrollo en casa.

17.4.1 Planeación de WebApp: subcontratación

CONSEJO

No se suponga que, puesto que se ha subcontratado una WebApp, las responsabilidades son mínimas. De hecho, es probable que se requieran más, no menos, supervisión y gestión.

Un porcentaje sustancial de las WebApps se subcontrata con proveedores que (supuestamente) se especializan en el desarrollo de sistemas y aplicaciones basados en Web.[5] En tales casos, un negocio (el cliente) pide un precio fijo para desarrollar la WebApp de uno o más proveedores, evalúa los precios competitivos y luego elige un proveedor para efectuar el trabajo. Pero, ¿qué busca la organización contratante? ¿Cómo se determina la competencia de un proveedor de WebApps? ¿Cómo se sabe si una cotización es razonable? ¿Qué grado de planeación, programa de trabajo y valoración de riesgo se pueden esperar conforme una organización (y su subcontratista) se embarca en un esfuerzo por desarrollar una gran WebApp?

> "Muchas empresas de Fortune 500 han descubierto al software como un modelo de servicio [subcontratado] y están contratando modelos similares interna o externamente."
>
> **Nick Evans**

Estas preguntas no siempre son fáciles de contestar, pero vale la pena considerar algunos lineamientos.

Inicio del proyecto. Si la subcontratación se elegirá como la estrategia para desarrollar la WebApp, la organización debe realizar una serie de tareas antes de buscar una empresa subcontratista que haga el trabajo:

1. *Realizar, internamente, muchas de las labores de análisis tratadas en la sección 17.1.3 (y en el capítulo 18).* Se identifica el público para la WebApp; se hace una lista con los accionistas internos interesados en la WebApp; se definen y revisan las metas globales para la WebApp; se especifican la información y servicios que habrá de proporcionar la WebApp; se destacan los sitios Web competidores; y se identifican las "medidas" cualitativas y cuantitativas de una WebApp exitosa. Esta información deberá documentarse en una especificación de producto que se entregará al subcontratista.

CONSEJO

Algunas personas argumentan que "el diseño aproximado" es innecesario. Véase como una "primera oferta" que el proveedor subcontratista puede modificar y mejorar. Al menos está comunicando sus ideas acerca de a qué se debe parecer el resultado final.

2. *Desarrollar internamente un diseño aproximado de la WebApp.* Obviamente, un desarrollador Web experto creará un diseño completo, pero es posible ahorrar tiempo y costo si la visión y el sentido general de la WebApp se identifican para la empresa subcontratista (esto siempre puede modificarse durante las etapas preliminares del proyecto). El diseño debe incluir una indicación del tipo y volumen de contenido que se presentará en la WebApp y los tipos de procesamiento interactivo (por ejemplo, formatos, entrada de pedidos) que se llevarán a cabo. Esta información deberá agregarse a la especificación del producto.

5 Aunque es difícil encontrar datos industriales confiables, puede afirmarse que este porcentaje es considerablemente mayor que el que se observa en el trabajo de software convencional. En el capítulo 23 se ofrece una exposición adicional acerca de la subcontratación.

3. *Elaborar un programa aproximado que incluya no sólo las fechas finales de entrega, sino también fechas clave* Las fechas clave se deben adjuntar a las versiones de entrega (incrementos) de la WebApp conforme ésta evolucione

4. *Crear una lista de responsabilidades para la organización interna y el subcontratista* En esencia, esta tarea aborda qué información, contactos y otros recursos se requieren de ambas organizaciones.

5. *Identificar el grado de supervisión e interacción de la organización contratante con el subcontratista.* Esto debe incluir el nombre del contacto del vendedor y la identificación de las responsabilidades y autoridad del contacto, la definición de los puntos de revisión de calidad conforme avance el desarrollo, y las responsabilidades del subcontratista en relación con la comunicación entre las organizaciones

¿Qué directrices deben seguirse cuando se considera a varios subcontratistas?

Toda la información generada durante estos pasos deberá organizarse en una solicitud de presupuesto que se entrega las empresas candidatas.[6]

Selección entre los subcontratistas candidatos. En los últimos años han surgido miles de compañías de "diseño Web" dedicadas a ayudar a las empresas que desean establecer una presencia Web o aventurarse en el comercio electrónico Muchas se han vuelto adictas al proceso de lWeb, pero muchas otras son poco más que *hackers* (intrusos informáticos). Con la finalidad de elegir desarrolladores Web candidatos, el contratante debe realizar algunas diligencias obligadas: 1) entrevistar a los clientes antiguos para determinar el profesionalismo del vendedor Web, así como su habilidad para cumplir con compromisos de plazos y costos, y su destreza para comunicarse efectivamente; 2) determinar el nombre del ingeniero(s) Web jefe de la empresa subcontratista para buscar proyectos anteriores exitosos (y, después, asegurarse de que esta persona tenga la obligación contractual de estar involucrada en su proyecto); y 3) examinar cuidadosamente ejemplos del trabajo del subcontratista que sean similares en apariencia y sentido (y área de negocios) a la WebApp que será contratada. Incluso antes de que se ofrezca una solicitud de presupuesto, una entrevista personal puede ofrecer un discernimiento sustancial de la "conexión" entre el contratante y el subcontratista.

> "Si pagas cacahuates, obtienes monos."
>
> **George Peppard en el papel del coronel John "Hannibal" Smith en *The A-Team* (serie televisiva de los ochenta)**

Valoración de la validez de las cotizaciones y la confiabilidad de las estimaciones. Puesto que existen relativamente pocos datos históricos y que el ámbito de las WebApps es fluido en forma notoria, la estimación es inherentemente ries-

6 Si el trabajo de desarrollo de la WebApp lo dirigirá un grupo interno, ¡no cambia nada! El proyecto se inicia de la misma manera.

gosa. Por esta razón, algunos proveedores incorporarán márgenes de seguridad sustanciales en cotizaciones para un proyecto. Esto es comprensible y apropiado. La pregunta *no* es si se ha obtenido la mejor solución por la inversión. Más bien, las preguntas deben ser:

- ¿La cotización de la WebApp ofrece un rendimiento sobre la inversión, directo o indirecto, que justifique el proyecto?
- ¿La empresa emisora de la cotización tiene el profesionalismo y la experiencia que se requieren?

Si las respuestas a estas preguntas son afirmativas la cotización es justa.

Comprensión del grado de gestión del proyecto que puede esperar o realizar. La formalidad asociada con las labores de gestión del proyecto (que realizan el proveedor y la organización contratante) es directamente proporcional al tamaño, costo y complejidad de la WebApp. Respecto a proyectos complejos y grandes será necesario elaborar un programa del proyecto que defina las tareas del trabajo, los puntos de comprobación, el aseguramiento de la calidad del software, los productos de trabajo de ingeniería, los puntos de revisión del cliente y los hitos importantes. El proveedor y el contratante tendrán que valorar el riesgo conjuntamente y elaborar planes para mitigar, monitorear y manejar los riesgos considerados importantes. Los mecanismos para asegurar la calidad y el control de cambios se deberán definir explícitamente por escrito. Se deberán establecer métodos para la comunicación efectiva entre el contratante y el proveedor.

Evaluación del programa del proyecto. Dado que los programas de desarrollo de WebApps abarcan un periodo relativamente corto (por lo general menos de uno o dos meses para que se entregue el incremento), el programa para el desarrollo debe tener una dosificación muy precisa. Es decir: las tareas de trabajo y los hitos menores se deben programar en un cronograma diario. Esta dosificación precisa permite, tanto a la organización contratante como al subcontratista, reconocer la hoja suelta de la agenda antes de que amenace la fecha de finalización

Gestión del ámbito. Como es enormemente probable que el ámbito cambiará conforme avance el proyecto de la WebApp, el modelo de proceso IWeb es adaptable e incremental. Esto permite que el equipo de desarrollo del subcontratista "congele" el ámbito para un incremento, de modo que se pueda crear una liberación operativa de la WebApp. El siguiente incremento puede abordar cambios en el ámbito que haya sugerido una revisión del incremento precedente, pero una vez que comience el segundo incremento el ámbito nuevamente se "congela" de manera temporal. Este enfoque permite que el equipo de la WebApp trabaje sin tener que acomodar una corriente continua de cambios, pero al mismo tiempo reconoce la evolución continua característica de la mayoría de las WebApps.

¿PUNTO CLAVE

En la gestión del ámbito se congela el trabajo que vaya a realizarse en un incremento. Los cambios se demoran hasta el siguiente incremento de la WebApp.

Los lineamientos sugeridos líneas atrás no intentan ser un recetario a prueba de tontos para la producción a tiempo de WebApps de bajo costo. Sin embargo, ayudan

rán tanto a la organización contratante como al subcontratista a iniciar el trabajo de manera flexible con un mínimo de malas interpretaciones.

HogarSeguro

Preliminares para la subcontratación

El escenario: La oficina de Doug Miller

Los actores: Doug Miller (gerente del equipo de ingeniería del software *HogarSeguro*) y Sharon Woods, vicepresidente de e-CommerceSystems, el proveedor contratado para el sitio Web de comercio electrónico *HogarSeguro* y gerente del equipo de ingeniería Web contratado para el trabajo.

La conversación:

Doug: Sharon, qué bueno que por fin nos encontramos para algo de trabajo que realizar en el mes, más o menos

Sharon (sonríe): Tenemos, pero parece que ustedes se retrasaron debidamente. Vinod ya nos ha dado un conjunto de especificaciones para el sitio y también ha creado la mayor parte de los objetos de contenido para el sitio y de la funcionalidad del sitio.

Doug: Bien. ¿Qué más necesitan?

Sharon: La funcionalidad de comercio electrónica es estándar. Lo que me preocupa es la fachada... el trabajo que requiere para que los usuarios personalicen el producto antes de la compra.

Doug: Vinod le dio el procedimiento básico, ¿no es así?

Sharon: Sí, lo hizo; pero quiero validarlo con algunos usuarios finales. También necesitamos contactar a sus proveedores de contenido para obtener descripciones de productos, para cada sensor, dibujo, lista de precios, información de interfase/interconexión, ese tipo de cosas

Doug: ¿Vinod tiene tiempo para hacerlas un *storyboard* del proceso de personalización?

Sharon: Está trabajando en él mientras platicamos. Dijo que tenía que poner un *seguro* en el lado del producto. Sabe que es crucial... dijo que me lo enviaría por correo electrónico mañana en la mañana.

Doug: Muy bien... mira, me gustaría estar en el trayecto de este proyecto. Podemos establecer algunas reglas básicas para supervisar desde nuestra parte. No quiero entrometerme en tu camino, pero...

Sharon: No hay problema, nos gusta tener involucrados a nuestros clientes.

Doug: Yo trabajaré como contacto para este proyecto. Toda comunicación vendrá a través mío o de alguien como Vinod, a quien yo cite. Puesto que estamos en un calendario apretado, me gustaría establecer una agenda que tenga una dosificación diaria y hablar contigo o enviarte correos electrónicos todos los días acerca de los logros, los problemas, etcétera. Sé que es mucho, pero creo que así es lo adecuado.

Sharon: Está bien

Doug (toma algunas hojas de papel de su escritorio y las entrega a Sharon): Escribí una agenda aproximada con fechas límite... ¿qué opinas?

Sharon (luego de estudiar la agenda): Mmmmm, no estoy segura de que esto funcionará para nosotros. Déjame trabajar una alternativa y hoy en la tarde te la envío por correo electrónico.

Doug: Claro

17.4.2 Planeación de WebApp: ingeniería Web en casa

Conforme las WebApps se vuelven más extensas y estratégicas para los negocios, muchas compañías han optado por controlar el desarrollo en casa. No sorprende que la IWeb en casa se gestione de manera un poco diferente a un esfuerzo de subcontratación

> "¿Qué haces cuando necesitas tener listo un sitio Web para ayer?"
>
> **James Lewin**

La gestión de proyectos IWeb pequeños y de tamaño moderado (es decir: me de 3-5 meses de duración) requiere un enfoque ágil que quite el énfasis en la ges del proyecto pero *no* elimine la necesidad de planear. Todavía se aplican los pri pios básicos de gestión de proyectos (capítulo 21), pero el enfoque global es parco y menos formal. Sin embargo, conforme crece el tamaño del proy WebApp, la gestión del proyecto de ingeniería Web se vuelve más y más como gestión de proyectos de ingeniería del software (Parte 4 de este libro). Los siguientes se recomiendan para proyectos IWeb pequeños y de tamaño moderad

CONSEJO

Es importante reconocer que los pasos analizados en esta sección se pueden realizar rápidamente. En ningún caso la planeación IWeb para proyectos de este tamaño toma más del 5 por ciento del esfuerzo del proyecto global

Entender el ámbito, las dimensiones de cambio y las restricciones del proyecto. Ningún proyecto, sin importar cuán apretada sea la restricción del tie puede comenzar mientras el equipo del proyecto no entienda qué debe construir recopilación de requisitos y la comunicación con el cliente son precursores esenc les para la planeación efectiva de la WebApp.

Definir una estrategia de proyecto incremental. Ya se ha señalado que WebApps evolucionan con el tiempo. Si la evolución es descontrolada y caótica probabilidad de un resultado exitoso es pequeña. Sin embargo, si el equipo est ce una estrategia de proyecto que defina incrementos (liberaciones) de WebApp proporcionen contenido útil y funcionalidad a los usuarios finales, el esfuerzo ingeniería puede enfocarse con mayor facilidad

Realizar análisis de riesgo. En el capítulo 25 se presenta una exposición llada del análisis de riesgo para proyectos tradicionales de ingeniería del software Todas las labores de gestión de riesgo se realizan para proyectos IWeb, pero su en que se abrevia.

Los riesgos que entrañan el programa y la tecnología dominan la preocupac de la mayoría de los equipos de ingeniería Web. Entre las muchas preguntas rela nadas con el riesgo que el equipo debe formular y responder están: ¿Los increme WebApp planeados pueden entregarse en los plazos definidos? ¿Estos increme proporcionarán valor subsecuente para los usuarios finales mientras se realiza ingeniería de incrementos adicionales? ¿Cómo impactan las fechas de entrega solicitudes de cambios? ¿El equipo comprende los métodos, tecnologías y her mientas de ingeniería Web requeridos? ¿La tecnología disponible es adecuada para trabajo? ¿Los cambios probables requieren la introducción de nueva tecnología?

Desarrollar una estimación rápida. El eje de la estimación para la mayoría los proyectos de ingeniería Web lo representan los conflictos macroscópicos, m que los microscópicos. El equipo IWeb valora si los incrementos WebApp plan pueden desarrollarse con los recursos disponibles de acuerdo con las restriccio del programa definido. Esto se logra considerando el contenido y la función de

7 Aquellos lectores que no estén familiarizados con la terminología y las prácticas básicas de la tión de riesgos deberán consultar en este momento el capítulo 25.

incremento como un todo. Normalmente no se realizan rompimientos "microscópicos", funcionales o de trabajo, del incremento que sean seguidos por el cálculo de estimaciones puntuales de múltiples datos (véase el capítulo 23).

Elegir un conjunto de tareas (descripción del proceso). Empleando un marco de trabajo del proceso (capítulo 16) se elige un conjunto de tareas de ingeniería Web que sean adecuadas para las características del problema, el producto, el proyecto y la gente en el equipo de ingeniería Web. Reconózcase la posibilidad de adaptar el conjunto de tareas para que encaje en el desarrollo de cada incremento.

Establecer un programa. El programa de un proyecto IWeb tiene una dosificación relativamente precisa respecto de las tareas que se realizarán en el corto plazo, y luego una mucho más flexible durante periodos posteriores (cuando vayan a entregarse los incrementos adicionales). Esto es, las tareas de ingeniería Web se distribuyen a lo largo de la línea de tiempo del proyecto para el incremento que se desarrollará. La distribución de tareas para subsecuentes incrementos WebApp se demora hasta la entrega del incremento programado.

Definir mecanismos de rastreo del proyecto. En un ambiente de desarrollo ágil, la entrega de un incremento operativo de software con frecuencia es la medida primaria del progreso. Pero mucho antes de que el software liberable esté disponible, el ingeniero Web enfrentará inevitablemente la pregunta: ¿dónde estamos? En el trabajo convencional de ingeniería del software el progreso se mide determinando qué objetivos se han logrado (por ejemplo, la revisión exitosa de un producto de trabajo). Respecto a proyectos de ingeniería Web pequeños y de tamaño moderado, los objetivos pueden estar menos definidos, y las actividades formales de aseguramiento de la calidad pueden perder fuerza. En consecuencia, es posible derivar una respuesta si se entrevista al equipo de ingeniería Web para determinar qué actividades del marco de trabajo se han completado. No obstante, este enfoque puede ser poco fiable. Otro enfoque es determinar cuántos casos de uso se han implementado y cuántos (para un incremento dado) permanecen sin implementarse. Esto proporciona una indicación aproximada del grado relativo en que se ha completado el incremento del proyecto.

CLAVE

"El progreso se logra corrigiendo los errores resultantes de lograr el progreso".

Claude Gibb

Establecer un enfoque de gestión del cambio. La gestión del cambio se facilita mediante la estrategia de desarrollo incremental que se recomendó para las WebApps. Puesto que el tiempo de desarrollo para un incremento es corto, con frecuencia es posible demorar la introducción de un cambio hasta el siguiente incremento, con la consiguiente reducción de los efectos de demora asociados con los cambios que se deben implementar "al vuelo". En el capítulo 27 se presenta la gestión de la configuración y el contenido para las WebApps.

HERRAMIENTAS DE SOFTWARE

Gestión de proyectos IWeb

Objetivo: Auxiliar al equipo de ingeniería Web en la planeación, gestión, control y rastreo de proyectos de ingeniería Web.

Mecánica: Las herramientas de gestión de proyectos le permiten a un equipo IWeb establecer un conjunto de tareas de trabajo, asignar esfuerzo y especificar responsabilidad a cada tarea, establecer dependencia de tareas, definir un programa y rastrear y controlar las actividades del proyecto. Muchas herramientas en esta categoría están basadas en Web.

Herramientas representativas[8]

Business Engine, desarrollado por Business Engine (www.businessengine.com), es una suite de herramientas basadas en Web que ofrecen facilidades de gestión para proyectos completos de IWeb y proyectos de software convencionales.

iteamwork, desarrollado por iTeamwork.com (www.iteamwork.com), "es una aplicación de equipo de gestión de proyecto gratuita, en línea y basada en Web, que puede usar con su navegador web".

OurProject, desarrollado por Our Project (www.ourproject.com), es una suite de herramientas de gestión de proyecto que son aplicables a la IWeb y a los proyectos de software convencionales.

Proj-Net, desarrollado por Rational Concepts, Inc. (www.rationalconcepts.com), "implementa una oficina de proyecto virtual (VPO, virtual project office) para colaboración y comunicación".

StartWright (www.startwright.com/project1.htm) ha desarrollado uno de los recursos más completos de la Web para herramientas e información, tanto para IWeb como para gestión de proyectos de software convencional.

Es necesario observar que muchas de las herramientas de gestión de proyecto convencional (Parte 4 de este libro) también se pueden aprovechar de manera efectiva en los proyectos IWeb

17.5 MEDICIÓN PARA INGENIERÍA WEB Y WEBAPPS

Los ingenieros Web desarrollan sistemas complejos y, al igual que otros tecnólogos que realizan esta tarea, deben usar mediciones para mejorar el proceso de ingeniería Web y el producto. En el capítulo 15 se analizaron los usos estratégicos y tácticos para la medición de software en un contexto de ingeniería del software. Dichos usos también se aplican en la ingeniería Web.

En resumen, la medición de software ofrece una base para mejorar el proceso de software, lo que aumenta la precisión de las estimaciones del proyecto, incrementa el rastreo del proyecto y mejora la calidad del software. La medición de ingeniería Web, si se caracteriza de manera adecuada, podría lograr todos estos beneficios y también mejorar la facilidad de uso, el desempeño de la WebApp y la satisfacción de usuario.

CONSEJO

En general, el número de medidas IWeb que se debe recopilar, y su complejidad global, debe ser directamente proporcional al tamaño de la WebApp que se construirá.

En el contexto de la ingeniería Web, las mediciones tienen tres metas principales: 1) proporcionar un indicador de la calidad de la WebApp desde el punto de vista técnico, 2) proporcionar una base para la estimación del esfuerzo, y 3) proporcionar una indicación del éxito de la WebApp desde el punto de vista empresarial.

8 Las herramientas expuestas sólo representan una muestra de esta categoría. En casi todos los casos los nombres de las mismas son marcas registradas de sus respectivos desarrolladores

En esta sección se resume un conjunto de mediciones de esfuerzo común y complejidad[9] para las WebApps. Este conjunto puede destinarse al desarrollo de una base de datos histórica para la estimación del esfuerzo. Además, la medición de la complejidad puede conducir a final de cuentas a una incapacidad para valorar cuantitativamente uno o más atributos técnicos de las WebApps discutidos en el capítulo 16.

17.5.1 Mediciones para esfuerzo de ingeniería Web

Los ingenieros Web dedican esfuerzo humano al realizar una diversidad de tareas de trabajo conforme evoluciona una WebApp. Mendes y sus colegas [MEN01] sugieren algunas posibles medidas de esfuerzo para WebApps. Algunas de (o todas) ellas podría registrarlas un equipo de ingeniería Web y luego aprovecharse en una base de datos histórica con fines de estimación (capítulo 23).

Aplicación de las tareas de autoría y diseño

Medida sugerida	Descripción
esfuerzo de estructuración	tiempo para estructurar la WebApp y/o la arquitectura derivada
esfuerzo de intervinculación	tiempo para intervincular páginas y así construir la estructura de las WebApp
planeación de interfaz	tiempo en que se planea la interfaz de la WebApp
construcción de interfaz	tiempo en que se implementa la interfaz de la WebApp
esfuerzo de prueba de vínculos	tiempo en que se prueban todos los vínculos en la WebApp
esfuerzo de prueba de los medios audiovisuales	tiempo en que se prueban todos los medios audiovisuales en la WebApp
esfuerzo total	esfuerzo de estructuración + esfuerzo de intervinculación + planeación de interfase + construcción de interfase + esfuerzo de prueba de vínculos + esfuerzo de prueba de los medios audiovisuales

Esfuerzo de autoría

Medida sugerida	Descripción
esfuerzo de texto	tiempo en que se crea o reutiliza texto en una página
esfuerzo de vinculación de página	tiempo en que se crean vínculos en la página
esfuerzo de estructuración de página	tiempo en que se estructura la página
esfuerzo de página total	esfuerzo de texto + esfuerzo de vinculación de página + esfuerzo de estructuración de página

9 Es importante notar que las mediciones Web todavía están en su infancia.

Autoría de medios audiovisuales

Medida sugerida	Descripción
esfuerzo de medio audiovisual	tiempo en que se crean o reutilizan archivos de medios audiovisuales
esfuerzo de digitalización de medios audiovisuales	tiempo en que se digitalizan medios audiovisuales
esfuerzo total de medios audiovisuales	esfuerzo de medio audiovisual + esfuerzo de digitalización de medios audiovisuales

Autoría de programas

Medida sugerida	Descripción
esfuerzo de programación	tiempo en que se crean HTML, Java o implementaciones de lenguaje relacionado
esfuerzo de reutilización	tiempo para reutilizar/modificar la programación existente

17.5.2 Medición del valor de negocios

Referencia Web
Una excelente referencia acerca de muchos materiales relacionados con las ventajas en Internet es **www.internet.com.**

Es interesante advertir que la gente de negocios ha llegado considerablemente antes que la ingeniería Web al desarrollo, la recopilación y el empleo de la medición para las WebApps (por ejemplo, [STE02], [NOB01]). Al entender la demografía de los usuarios finales y sus patrones de uso, una compañía u organización puede desarrollar la entrada inmediata para más contenido WebApp significativo, ventas y esfuerzos de mercadotecnia más efectivos, y mejorar la rentabilidad de los negocios.

Los mecanismos requeridos con que se recopilan datos valiosos para la empresa usualmente los implementa el equipo de ingeniería Web, pero evaluarlos y las acciones resultantes las realizan otros participantes. Por ejemplo, supóngase la posibilidad de determinar el número de vistas de la página para cada visitante único. Con base en la medición recopilada, los visitantes que llegan desde el motor de búsqueda *X* promedian nueve vistas de página, mientras que los visitantes desde el portal *Y* sólo dos. Estos promedios los puede emplear el departamento de mercadotecnia para ubicar un anuncio publicitario (*banner*) donde promueva presupuestos (la publicidad en el motor de búsqueda *X* proporciona mayor exposición, con base en la medición recolectada, que la publicidad en el portal *Y*).

HERRAMIENTAS DE SOFTWARE

Mediciones Web

Objetivo: Valorar la forma en la que se utiliza una WebApp, las categorías de usuarios y la facilidad de uso de la WebApp.

Mecánica: La gran mayoría de las herramientas de medición Web captura la información de uso una vez que la WebApp está en línea. Dichas herramientas proporcionan una amplia variedad de datos con los cuales se valora qué elementos de la WebApp se utilizan más, cómo se utilizan y quién los utiliza.

...ientas representativas[10]

...s, desarrollada por clicktracks.com ...clicktracks.com), es una herramienta de análisis ... archivos de acceso (log) que muestra el ...amiento del visitante al sitio Web directamente ... páginas de éste.

...e, desarrollado por Coremetrics ...w.Coremetrics.com), es representativa de muchos ...amientas que recopilan datos con los cuales se valora el éxito de las WebApps de comercio electrónico.

Web Metrics Testbed, desarrollado por NIST (zing.ncsl.nist.gov/WebTools/), es una suite de herramientas basados en Web que valoran la facilidad de uso de una WebApp.

WebTrends, desarrollado por netIQ (www.NetIQ.com), recopila un amplio rango de datos de uso para WebApps de todos los tipos

Una revisión completa de la recopilación y el empleo de las mediciones con valor en los negocios (que incluya el debate actual acerca de la privacidad personal) está más allá del ámbito de este libro. El lector interesado deberá examinar [JNA02], [EIS02], [PAT02] o [RIG01].

17.6 Las "peores prácticas" para proyectos WebApp

En ocasiones, la mejor forma de aprender cómo hacer algo correctamente ¡es examinar cómo no hacerlo! Durante la década pasada, muchas WebApps fracasaron porque 1) un descuido del proyecto y el cambio en los principios de gestión (de cualquier manera informales) resultó en un equipo de ingeniería Web que "rebotó en las paredes"; 2) un enfoque *ad hoc* para el desarrollo de la WebApp falló y no produjo un sistema operable; 3) un enfoque desdeñoso hacia la recopilación y análisis de requisitos fracasó en producir un sistema que satisfaciera las necesidades del usuario; 4) un enfoque incompetente para el diseño fracasó al intentar producir un desarrollo de la WebApp que fuese utilizable, funcional, extensible (sustentable) y verificable; 5) un enfoque equivocado para las pruebas fracasó para producir un sistema que funcionase antes de su introducción

Con estas situaciones en mente, tal vez valga la pena considerar un conjunto de las "peores prácticas" en la ingeniería Web, adoptadas de un artículo de Tom Bragg [BRA00]. Si su proyecto electrónico muestra cualquiera de ellas, es necesaria una acción correctiva inmediata.

Peor práctica #1: *Se tiene una gran idea, así que se puede comenzar a construir la WebApp ahora.* No es necesario preocuparse en considerar si la WebApp está justificada por el negocio, si los usuarios realmente querrán usarla, si se comprenden los requisitos del negocio. El tiempo es corto, tiene que comenzarse.

Realidad: Tómense unas cuantas horas o días y elabórese un caso de negocios para la WebApp. Asegúrese de que la idea la apoyan quienes la financiarán y quienes la usarán.

10 Las herramientas expuestas el autor no las respalda; sólo representan una muestra de las herramientas incluidas en esta categoría. En casi todos los casos, los nombres de las herramientas son marcas registradas de sus respectivos desarrolladores

Peor práctica #2: *Las cosas cambiarán constantemente, así que no tiene caso tratar de comprender los requisitos de la WebApp* Nunca se escriba algo (pérdida de tiempo). El apoyo debe basarse exclusivamente en la palabra oral.

Realidad: Es cierto que los requisitos de la WebApp evolucionan conforme continúan las actividades de ingeniería Web También es más rápido y simple obtener información de manera verbal. Sin embargo, un enfoque desdeñoso respecto de la recopilación y el análisis de requisitos es un catalizador para más cambio (innecesario) todavía.

Peor práctica #3: *Los desarrolladores cuya experiencia dominante se relaciona con el desarrollo de software tradicional pueden desarrollar WebApps inmediatamente. No se requiere un nuevo entrenamiento.* Después de todo, el software es software, ¿o no?

Realidad: Las WebApps son diferentes. Se debe aplicar de manera experta un amplio abanico de métodos, tecnologías y herramientas. El entrenamiento y la experiencia con ellos es esencial.[1]

Peor práctica #4: *Burocratizarse.* Insista en modelos de proceso pesados, horarios, muchas e innecesarias reuniones "de progreso" y en líderes de proyecto que nunca han gestionado un proyecto WebApp

Realidad: Aliente un proceso ágil que resalte la competencia y la creatividad de un equipo de ingeniería Web experimentado. Luego salga de su camino y permita trabajar. Si se deben recopilar datos relacionados con el proyecto (por razones legales o el cálculo de la medición), el ingreso/recopilación de datos debe ser tan no obstructivo y simple como sea posible.

Peor práctica #5: *¿Pruebas? ¿Por qué molestarse?* Se le dará a unos cuantos usuarios finales y se dejará que ellos digan qué funciona y qué no

Realidad: Con el tiempo, los usuarios finales sí realizan "pruebas" exhaustivas pero están tan enojados por la falta de confiabilidad y el pobre desempeño que la dejan mucho antes de que los problemas sean corregidos (nunca regresan).

En los capítulos que siguen se considerarán los métodos de ingeniería Web que ayudarán a evitar estos errores.

17.7 Resumen

La formulación, una actividad de comunicación con el cliente, define el problema que resolverá una WebApp Se identifican las necesidades del negocio, la metas y objetivos del proyecto, las categorías de usuario final, las funciones y características principales y el grado de interoperabilidad con otras aplicaciones. Mientras más información detallada y técnica se adquiera, la formulación se convierte en análisis de requisitos

[1] Muchos grandes proyectos IWeb requieren integración con aplicaciones y bases de datos convencionales. En tales casos, los individuos sólo con experiencia convencional pueden y deben ser involucrados

El equipo IWeb lo integra un grupo de miembros técnicos y no técnicos organizados en una forma que les brinda considerable autonomía y flexibilidad. Durante la ingeniería Web se requiere gestión del proyecto, pero las tareas correspondientes están abreviadas y son considerablemente menos formales que las aplicadas en los proyectos convencionales de ingeniería del software. Muchos proyectos WebApp se subcontratan, pero existe una tendencia creciente hacia el desarrollo de WebApps en casa. La gestión del proyecto para cada enfoque difiere tanto en estrategia como en tácticas.

Las mediciones de la ingeniería Web están en desarrollo, pero tienen el potencial para ofrecer una indicación de la calidad de la WebApp, proporcionar una base para la estimación del esfuerzo y permitir vislumbrar el éxito de la WebApp desde el punto de vista de los negocios.

Referencias

[BRA00] Bragg, T., "Worst Practices for e-Business Projects: We Have Met the Enemy and He Is Us!", *Cutter IT Journal*, vol. 13, núm. 4, abril de 2000, pp. 35-39.

[CON02] Constantine, L. y L. Lockwood, "User-Centered Engineering for Web Applications", en *IEEE Software*, vol. 19, núm. 2, marzo-abril de 2002, pp. 42-50.

[EIS02] Eisenberg, B., "How to Interpret Web Metrics", en *ClickZ Today*, marzo de 2002, disponible en http://www.clickz.com/sales/traffic/article.php/992351.

[FUC02a] Fuccella, J., J. Pizzolato y J. Franks, "Finding Out What Users Want from your Web Site", IBM developerWorks, 2002, http://www-106.ibm.com/developerworks/library-/moderator-guide/requirements.html.

[FUC02b] Fuccella, J. y J. Pizzolato, "Giving People What They Want: How to Involve Users in Site Design", IBM developerWorks, 2002, http://www-106.ibm.com/developerworks/library/design-by-feedback/expectations.html.

[GNA99] Gnado, C. y F. Larcher, "A User-Centered Methodology for Complex and Customizable Web Applications Engineering", en *Proc. First ICSE Workshop in Web Engineering*, ACM, Los Ángeles, mayo de 1999.

[HAN99] Hansen, S., Y. Deshpande y S. Murugesan, "A Skills Hierarchy for Web Information System Development", en *Proc. First ICSE Workshop on Web Engineering*, ACM, Los Ángeles, mayo de 1999.

[INA02] Inan, H. y M. Kean, *Measuring the Success of Your Web Site*, Longman Publishing, 2002.

[KID00] Kidder, T., *The Soul of a New Machine*, Back Bay Books (edición reimpresa), 2000.

[KUL00] Kulik, P. y R. Samuelsen, "e-Project Management for a New e-Reality", Project Management Institute, diciembre de 2000, http://www.seeprojects.com/e-Projects/e-projects.html.

[LOW98] Lowe, D. y W. Hall (eds.), *Hypertext and the Web-An Engineering Approach*, Wiley, 1998.

[MEN01] Mendes, E., N. Mosley y S. Counsell, "Estimating Design and Authoring Effort", en *IEEE Multimedia*, enero-marzo de 2001, pp. 50-57.

[NOB01] Nobles, R. y K. Grady, *Web Site Analysis and Reporting*, Premier Press, 2001.

[PAT02] Patton, S., "Web Metrics That Matter", en *CIO*, 15 de noviembre de 2002, disponible en http://www.computerworld.com/developmenttopics/websitemgmt/story/0,10801,76002,00.html.

[PIC01] Pickering, C., "Building an Effective E-Project Team", en *E-Project Management Advisory Service*, Cutter Consortium, vol. 2, núm. 1, 2001, http://www.cutter.com/consortium.

[POW98] Powell, T. A., *Web Site Engineering*, Prentice Hall, 1998.

[RIG01] Riggins, F. y S. Mitra, "A Framework for Developing E-Business Metrics Through functionality Interaction", enero de 2001, se puede descargar de http://digitalenterprise.org/metrics/metrics.html.

[STE02] Sterne, J., *Web Metrics: Proven Methods for Measuring Web Site Success*, Wiley, 2002

[TIL99] Tilley, S. y S. Huang, "On the Emergence of the Renaissance Software Engineer", *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Ángeles, mayo de 1999

PROBLEMAS Y PUNTOS A CONSIDERAR

17.1. ¿En qué difiere la formulación de la recopilación de requisitos? ¿En qué difiere la formulación del análisis de requisitos y del modelado de análisis?

17.2. En la sección 17.1.1 se encuentran tres preguntas fundamentales acerca de la formulación. ¿Existen algunas otras preguntas que se considere posibles de plantear en este punto? Si es así, ¿cuáles son y por qué deberían hacerse?

17.3. En el contexto de la recopilación de requisitos, ¿qué es una "categoría de usuario"? Dense ejemplos de tres categorías de usuario para un vendedor de libros en línea.

17.4. Considérese el sitio de comercio electrónico de *HogarSeguro* tratado en este capítulo. ¿Qué mecanismo de comunicación con el usuario podría usarse para obtener requisitos del sistema y por qué?

17.5. Con palabras propias, exponga cómo se "analiza" la información recopilada durante la comunicación con el cliente y cuál es el resultado de esta actividad

17.6. ¿Qué beneficios se pueden derivar de requerir el desarrollo de casos de uso como parte de la actividad de recopilación de requisitos?

17.7. Revísese la tabla presentada en la figura 17.1. Agréguense tres hileras más que ulteriormente distinguirán los proyectos tradicionales de los electrónicos.

17.8. Con palabras propias, describa el papel del editor Web

17.9. Revísense las características de los equipos de desarrollo ágil analizados en el capítulo 4. ¿Se advierte que una organización en equipo ágil es apropiada para la IWeb? ¿El lector realizaría algún cambio a la organización para el desarrollo de la WebApp?

17.10. Describanse cinco riesgos asociados con la subcontratación del desarrollo de WebApps

17.11. Describanse cinco riesgos asociados con el desarrollo en casa de las WebApps

17.12. Considérense las mediciones para el esfuerzo de ingeniería Web tratadas en la sección 17.5.1. Inténtese desarrollar cinco o más mediciones adicionales para una o más categorías

17.13. La facilidad de navegación a través de un sitio Web es un indicador importante de la calidad de la WebApp. Desarróllense dos o tres mediciones con las cuales pudiera indicarse la facilidad de navegación

17.14. Aprovechando una de las referencias sugeridas en la sección 17.5.2, comente cómo se pueden aprovechar las mediciones con valor en los negocios para apoyar la toma de decisiones pragmática en éstos.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Los métodos para la formulación de WebApps y la recopilación de requisitos se pueden adaptar de análisis de métodos similares para el software de aplicación convencional. Las otras lecturas recomendadas en los capítulos 7 y 8 contienen mucha información útil para el ingeniero Web.

Flor (*Web Business Engineering*, Addison-Wesley, 2000) aborda el análisis de negocios y las preocupaciones relacionadas que permiten al ingeniero Web comprender mejor las necesidades del cliente. La facilidad de uso de la WebApp es un concepto que subyace a mucha de la información definida como parte de la formulación y la recopilación de requisitos. Krug y Black (*Don't Make me Think: A Common Sense Approach to Web Usability*, Que Publishing, 2000) contiene muchas directrices y ejemplos que pueden ayudar al ingeniero Web a traducir los requisitos del usuario en una WebApp efectiva.

La gestión de proyecto para los proyectos IWeb parte de muchos de los mismos principios y conceptos aplicables en proyectos de software convencional. Sin embargo, agilidad es un lema. Wallace (*Extreme Programming for Web Projects*, Addison-Wesley, 2003) describe cómo se puede aprovechar el desarrollo ágil para la IWeb y contiene análisis útiles de conflictos de gestión de proyectos. Shelford y Remillard (*Real Web Project Management*, Addison-Wesley, 2003), O'Connell (*How to Run Successful Projects in Web Time*, Arthec House, 2000), Freidlein (*Web Project Management*, Morgan Kaufman, 2000) y Gilbert (*90 Days to Launch: Internet Projects on Time and on Budget*, Wiley, 2000) tratan una amplia variedad de temas de gestión de proyectos para IWeb. Whitehead (*Leading a Software Development Team*, Addison-Wesley, 2001) presenta muchos lineamientos útiles que pueden adaptar los equipos de ingeniería Web.

Las técnicas para usar mediciones Web en la toma de decisiones empresariales se presentan en libros como los de Sterne [STE02], Inan [INA02], Nobles [NOB01] y Menasce y Almeida (*Capacity Planning for Web Services: Metrics, Models and Methods*, Prentice-Hall, 2001). Las "peores prácticas" son consideradas por Ferry y Ferry (*77 Sure Fire Ways to Kill a Software Project*, iUniverse.com, 2000).

En Internet está disponible una amplia variedad de fuentes de información acerca de formulación y planeación para ingeniería Web. Una lista actualizada de referencias en la World Wide Web, relevante para la formulación y la planeación, se encuentra en el sitio Web de SEPA: **http://www.mhhe.com/pressman**.

CAPÍTULO

18 MODELADO DE ANÁLISIS PARA APLICACIONES WEB

CONCEPTOS CLAVE

A primera vista, existe una aparente contradicción cuando se considera el modelado de análisis dentro del contexto de la ingeniería Web. Después de todo, se ha notado (capítulo 16) que las WebApps tienen una inmediatez y una volatilidad contraria al modelado detallado, ya sea en la etapa de análisis o en la del diseño. Y si se realiza algún tipo de modelado, la filosofía ágil (un modelo de proceso adecuado para muchos proyectos de ingeniería Web) sugiere que el modelado del análisis se minimice en favor del modelado de diseños limitados. Franklin [FRA02] advierte esta situación y escribe:

> Los sitios Web, por lo general, son complejos y enormemente dinámicos. Requieren fases de desarrollo cortas con la finalidad de tener listo el producto y ejecutarlo rápidamente. Con frecuencia, los desarrolladores van directo hacia la fase de codificación sin comprender qué están tratando de construir o cómo quieren construirlo. La codificación respecto del servidor con frecuencia se hace *ad hoc*, las tablas de bases de datos se agregan conforme se necesitan y la arquitectura evoluciona en una forma a veces no intencional. Pero alguna ingeniería de software modelada y disciplinada logrará que el proceso de desarrollo de software sea mucho más suave y asegurará que el sistema Web sea más sustentable en lo futuro.

¿Es posible tenerlo en las dos formas? ¿Se puede hacer "alguna ingeniería de software modelada y disciplinada" y todavía así trabajar efectivamente en un mundo donde reinan la inmediatez y la volatilidad? La respuesta es un calificado sí.

UN VISTAZO RÁPIDO

¿Qué es? El análisis de una potencial aplicación Web se enfoca en tres preguntas importantes: 1) ¿qué información o contenido se presentará o manipulará?; 2) ¿qué funciones realizará el usuario final?, y 3) ¿qué comportamientos exhibirá la WebApp conforme presente contenido y realice funciones? Las respuestas se representan como parte de un modelo de análisis que abarca una diversidad de representaciones UML.

¿Quién lo hace? Los ingenieros Web, los desarrolladores de contenido que no son técnicos y los clientes participan en la creación del modelo de análisis.

¿Por qué es importante? A lo largo de este libro se ha resaltado la necesidad de comprender el problema antes de comenzar a resolverlo. El modelado de análisis es importante no porque permita que un equipo de ingeniería Web desarrolle un modelo concreto de requisitos WebApp (las cosas cambian muy frecuentemente como para que esto sea una expectativa realista), sino que, más bien, permite que un ingeniero Web defina aspectos fundamentales del problema: elementos cuyo cambio no es probable (en un futuro cercano). El diseño y la construcción se facilitan cuando se comprenden el contenido, la función y el comportamiento fundamentales.

¿Cuáles son los pasos? El modelado de análisis se enfoca en los aspectos fundamentales del problema: contenido, interacción, función y configuración. El análisis de contenido identifica las clases y colaboraciones de contenido. El análisis de la interacción describe los elementos básicos de la interacción del usuario, la navegación y los comportamientos del sistema que ocurren. El análisis de las funciones define las funciones de WebApp que realizará el usuario y la secuencia de procesamiento que ocurre como consecuencia. El análisis de la configuración identifica el ambiente(s) operativo en el cual reside la WebApp.

¿Cuál es el producto obtenido? El modelado de análisis lo integran un conjunto de diagramas y textos UML que describen el contenido, la interacción, la función y la configuración.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Los productos obtenidos del modelado de análisis se deben revisar para corregirlos, completarlos y darles consistencia.

Un equipo de ingeniería Web debe emprender el modelado de análisis cuando se cumplen la mayoría o todas las condiciones siguientes:

- La WebApp que se construirá es grande o compleja.
- El número de clientes es grande.
- El número de ingenieros Web y otros colaboradores es grande.
- Las metas y los objetivos (determinados durante la formulación) para la WebbApp afectarán la línea de referencia del negocio.
- El éxito de la WebApp tendrá una fuerte conexión con el del negocio.

Si estas condiciones no están presentes, lo que le resta importancia al modelado de análisis, aprovechar la información obtenida durante la formulación y la recopilación de requisitos (capítulo 17) sirve como base para la creación de un modelo de diseño para la WebApp. En tales circunstancias, tal vez se obtenga un modelado de análisis limitado, pero que terminará incluido en el diseño.

18.1 Requisitos para el análisis de las WebApps

El *análisis de requisitos* para las WebApps abarca tres grandes tareas: formulación, recopilación de requisitos,[1] y modelado de análisis. Durante la formulación se identifican la motivación (metas) y los objetivos básicos para la WebApp, y también se definen las categorías de usuarios. Cuando comienza la recopilación de requisitos se intensifica la comunicación entre el equipo de ingeniería Web y los accionistas (por ejemplo, clientes, usuarios finales). Los requisitos de contenido y funcionales se enlistan y se desarrollan los escenarios de interacción (casos de uso) escritos desde el punto de vista del usuario final. La intención es establecer una comprensión básica de por qué se construirá la WebApp, quién la usará y qué problema) resolverá a sus usuarios.

1 En el capítulo 17 se abordan con detalle la formulación y la recopilación de requisitos.

"Los principios de ingeniería acerca de planear antes de diseñar y diseñar antes de construir han resistido cada transición tecnológica previa; también sobrevivirán a esta transición."

Watts Humphrey

18.1.1 La jerarquía de usuario

Las categorías de usuarios finales que interactuarán con la WebApp se identifican como parte de las tareas de formulación y de recopilación de requisitos. En la mayoría de los casos, las categorías de usuario son relativamente limitadas y no necesitan una representación UML. Sin embargo, cuando crece el número de categorías de usuario, a veces es aconsejable desarrollar una *jerarquía de usuarios*, como se muestra en la figura 18.1. La figura muestra a los usuarios del sitio de comercio electrónico de HogarSeguroInc.com tratada en los capítulos 16 y 17.

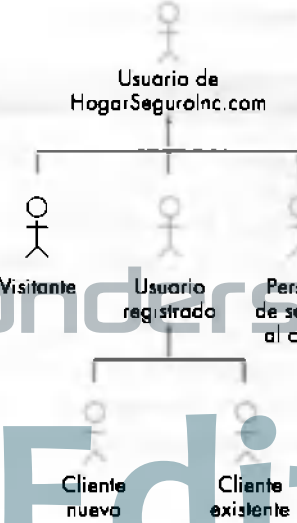
CONSEJO

Es buena idea construir una jerarquía de usuario. Ofrece una visión instantánea de la población de usuarios y una marca de verificación que ayudarán a asegurar que se han abordado las necesidades de cada usuario.

Las categorías de usuario (con frecuencia llamados *actores*) que se muestran en la figura 18.1 indican la funcionalidad que ofrecerá la WebApp; además, señalan la necesidad de que se desarrollen casos de uso para cada usuario final (actor) anotado en la jerarquía. En la misma figura, el **usuario de HogarSeguroInc.com** en la parte superior de la jerarquía representa la clase (categoría) de usuario más general y se refina niveles abajo. Un **visitante** es un usuario que visita el sitio pero no se registra. Tales usuarios usualmente buscan información general, comparan compras o de alguna otra forma están interesados en contenido o funcionalidad "gratuitos". Un **usuario registrado** dedica tiempo para ofrecer información y se le considera un contacto (junto con otros datos demográficos que solicitan las entradas de los formularios). Las subcategorías para los **usuarios registrados** incluyen:

FIGURA 18.1

Jerarquía de usuarios para HogarSeguroInc.com.



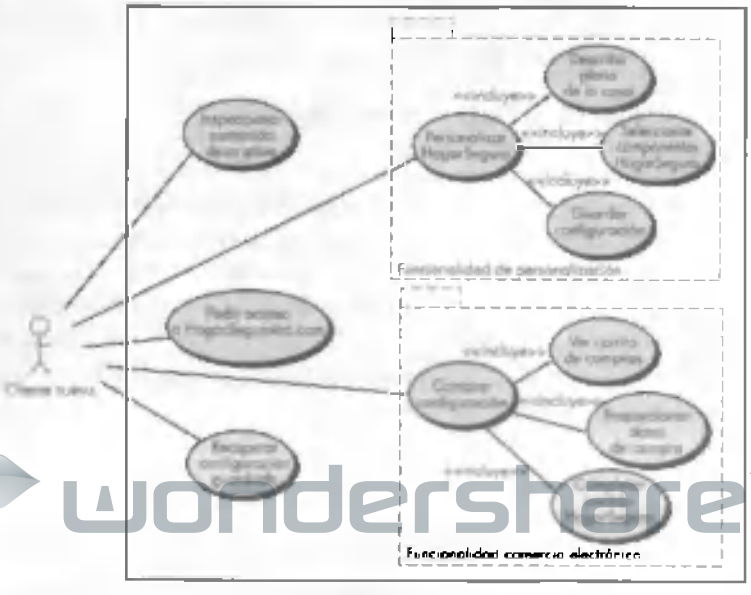
- **Cliente nuevo:** usuario registrado que quiere personalizar y luego comprar componentes de HogarSeguro (y, por tanto, debe interactuar con la WebApp de funcionalidad de comercio electrónico).
- **Cliente existente:** un usuario que ya posee componentes de HogarSeguro y usa la WebApp para 1) comprar componentes adicionales; 2) adquirir información de soporte técnico; o 3) contactar con el soporte al cliente.

Los miembros del **personal de servicio al cliente** son usuarios especiales que también pueden interactuar con el contenido y la funcionalidad de HogarSeguroInc.com conforme asisten a los clientes que han establecido contacto con el soporte al cliente de *HogarSeguro*.

18.1.2 Desarrollo de casos de uso

Franklin [FRA01] se refiere a los casos de uso como "haces de funcionalidad". Esta descripción captura la esencia de esta importante técnica de modelado de análisis.[2] Los casos de uso se desarrollan para cada categoría de usuario descrita en la jerarquía de usuario. En el contexto de la ingeniería Web, el caso de uso en sí mismo es relativamente informal: un párrafo narrativo que describe una interacción específica entre un usuario y la WebApp.[3]

18.2



2 Las técnicas para desarrollar casos de uso se analizaron con detalle en capítulos anteriores de este libro (véanse los capítulos 7 y 8).

3 Aunque es posible desarrollar descripciones más formales de casos de uso, la necesidad de agilidad para la IWeb con frecuencia excluye este enfoque.

La figura 18.2 representa un diagrama UML de caso de uso para la categoría de usuario **cliente nuevo** (figura 18.1). Cada óvalo en el diagrama representa un caso de uso que describe una interacción específica entre el **cliente nuevo** y la WebApp. Por ejemplo, la primera interacción se describe con el caso de uso *pedir acceso (log-in) a HogarSeguroInc.com*. No se requeriría más de un solo párrafo para describir esta interacción común.

La funcionalidad de las grandes WebApp (y los casos de uso relevantes para ella) se anotan adentro de recuadros con líneas punteadas en la figura 18.2. Tales recuadros se conocen como "paquetes" en UML y representan funcionalidad específica. Se advierten dos paquetes: *personalización* y *comercio electrónico*.

Como ejemplo, considérese el paquete *personalización* de casos de uso. Un nuevo cliente debe describir el ambiente doméstico en el cual se instalará *HogarSeguro*. Para lograrlo, **cliente nuevo** inicia los casos de uso *describir plano de la casa*, *seleccionar componentes HogarSeguro* y *guardar configuración*. Considérense los siguientes casos de uso preliminares escritos desde el punto de vista de un **cliente nuevo**.

Caso de uso: *describir plano de la casa*

CONSEJO

Conforme crece el tamaño de una WebApp, y el modelado de análisis se vuelve más riguroso, los casos de uso preliminares presentados aquí serán expandidos para ajustarse de manera más cercana al formato sugerido en la sección 8.5 del capítulo 8.

La WebApp formulará algunas preguntas generales acerca del ambiente en el cual se planea instalar *HogarSeguro*: número de habitaciones y su tamaño, tipo de habitación, número de pisos, número de puertas exteriores y ventanas. La WebApp permitirá construir un plano de la casa aproximado al conjuntar formas delineadas de las habitaciones para cada piso. El usuario será capaz de nombrar al plano de la casa y guardarlo para una referencia futura (véase caso de uso: *guardar configuración*).

Caso de uso: *seleccionar componentes HogarSeguro*

Entonces la WebApp recomendará componentes de producto (por ejemplo: paneles de control, sensores, cámaras) y otras características (por ejemplo, funcionalidad basada en PC implementada en software) para cada habitación y la entrada exterior. Si el usuario solicita opciones, la WebApp las proporcionará si existen. El usuario obtendrá información descriptiva y de precios para cada componente de producto. La WebApp creará y mostrará una factura de materiales conforme se seleccionen varios componentes. El usuario también podrá nombrar la factura de materiales y guardarla para referencia futura (véase caso de uso: *guardar configuración*).

Caso de uso: *guardar configuración*

La WebApp permitirá guardar los datos de personalización de modo que el usuario pueda regresar después. Podrá guardar el plano de la casa y la factura de materiales *HogarSeguro* que eligió para él. Lograr esto requiere que el usuario proporcione un identificador único para el plano de la casa y la factura de materiales. También proporcionará una contraseña (password) de configuración especial que debe validarse antes de que pueda tener acceso a la información guardada.

HOGARSEGURO

Refinar casos de uso para WebApps

El escenario: Oficina de Doug

Los actores: Doug Miller (gerente del grupo de ingeniería de software *HogarSegura*), Sharon Woods, gerente de ingeniería Web del proveedor externo para el sitio Web de comercio electrónico *HogarSeguro*, y Sam Chen, gerente de la organización de soporte al cliente de HogarSeguroInc.com.

La conversación:

Doug: Sharon, me da gusto escuchar que las cosas van bien. ¿El modelado de análisis está casi terminado?

Sharon (sonríe): Estamos progresando. El único conjunto de casos de uso que falta por desarrollar de la jerarquía de usuario [figura 18.1] es la categoría *personal de servicio al cliente*.

Doug (mirando a Sam): ¿Sam, ahora tú tienes esos casos de uso?

Sam: Sí, los tengo. Se los envié por correo electrónico a Doug, con copia para ti. Aquí está la versión impresa [le da a Doug y Sharon unas hojas de papel].

Doug: Como lo vemos, queremos usar el sitio Web de HogarSeguroInc.com como una herramienta de soporte cuando los clientes ordenen por teléfono. Nuestros representantes telefónicos completarán todos los formularios necesarios, etc., y procesarán el pedido por el cliente.

Sharon: ¿Por qué no sólo remitir al cliente al sitio Web?

Sam (sonríe): Los técnicos piensan que todos se sienten cómodos con la Web. ¡No es así! Hay mucha gente a la que todavía le gusta el teléfono, así que les tenemos que dar esa opción. Pero no queremos construir un sistema de procesamiento de solicitudes por separado cuando la mayoría de las piezas ya están en el lugar de la Web.

Sharon: Tiene sentido.

[Todas las partes leen los casos de uso [a continuación se presenta un ejemplo]]:

Caso de uso: *describir plano de la casa* [note que esto difiere del caso de uso del mismo nombre para la categoría *cliente nuevo*]

Sam: *Pediré al cliente [vía telefónica] que describa cada habitación de la casa e ingresaré las dimensiones de la habitación y otras características en un gran formulario diseñado específicamente para el personal de soporte al cliente. Una vez que se hayan ingresado los datos de la casa podré guardar los datos con el nombre o número de teléfono del cliente.*

Sharon: Sam, has sido un tanto lacónico en tus descripciones preliminares de caso de uso. Creo que tendremos que detallarlas un poco.

Doug (asintiendo con la cabeza): Estoy de acuerdo.

Sam (malhumorado): ¿Por qué?

Sharon: Bueno... mencionas "un gran formulario diseñado específicamente para el personal de soporte al cliente". Vamos a necesitar más detalles.

Sam: Lo que quise decir fue que no necesitamos llevar a nuestros representantes por todo el proceso como ustedes lo hacen para un cliente en línea. Un gran formulario resolvería el problema.

Sharon: Bosquejemos cómo se vería el formulario.

Las partes trabajan para proporcionar suficiente detalle que permita al equipo de Sharon emplear en forma efectiva el caso de uso.

Aunque es posible ofrecer considerablemente más detalle para cada uno de los casos de uso, la descripción textual informal ofrece una visión útil. Descripciones similares se desarrollarían para cada óvalo en la figura 18.2.

18.1.3 Afinación del modelo de caso de uso

A la par que se crean los diagramas de caso de uso para cada categoría de usuario, se desarrolla una vista superior de los requisitos de la WebApp observables de ma-

nera externa. Los casos de uso se organizan en paquetes funcionales, y cada paquete se valora [CON00] para garantizar que es:

¿Cómo se valoran los paquetes de casos de uso agrupados por la función usuario?

- Comprensible: todos los clientes entienden el propósito del paquete.
- Cohesivo: el paquete aborda funciones relacionadas cercanamente una con otra.
- Libremente acoplados: las funciones o clases dentro del paquete colaboran una con otra, pero la colaboración exterior del paquete se mantiene en un mínimo.
- Jerárquicamente superficial: las jerarquías funcionales profundas son difíciles de navegar y entender para los usuarios; en consecuencia, el número de niveles dentro de una jerarquía de casos de uso debe reducirse siempre que sea posible.

Puesto que el análisis de requisitos y el modelado son actividades iterativas, es probable que se sumen nuevos casos de uso a los paquetes que se han definido, que los casos de uso existentes sean refinados y que casos de uso específicos puedan reubicarse en paquetes diferentes.

18.2 El modelado de análisis para WebApps

El modelado de análisis para una WebApp se basa en la información que contienen los casos de uso desarrollados para la aplicación. Las descripciones de los casos de uso se analizan gramaticalmente para identificar potenciales clases de análisis y las operaciones y atributos asociados con cada clase. Se identifica el contenido que presentará la WebApp y se extraen las funciones que se desarrollarán a partir de las descripciones de caso de uso. Finalmente, los requisitos específicos de la implementación se deben desarrollar de modo que el ambiente y la infraestructura que apoyan la WebApps puedan construirse

Cuatro actividades de análisis, cada una con su aporte a la creación de un modelo de análisis completo, son:

¿Qué tipos de actividades de análisis ocurren durante el modelado de una WebApp?

- Análisis de contenido: identifica todo el espectro del contenido que ofrecerá la WebApp. El contenido incluye texto, gráficas e imágenes, así como datos de video y audio.
- Análisis de interacción: describe cómo interactúa el usuario con la WebApp.
- Análisis de funciones: define las operaciones que se aplicarán al contenido de la WebApp y describe otras funciones de procesamiento, independientes del contenido pero necesarias para el usuario final.
- Análisis de configuración: describe el ambiente y la infraestructura en la que reside la WebApp.

La información recopilada durante las tareas de estos cuatro análisis se debe revisar, modificar cuando se requiera y luego organizarse en un modelo que pueda pasarse a los diseñadores de WebApp

El modelo en sí mismo contiene elementos estructurales y dinámicos. Los *elementos estructurales* identifican las clases de análisis y los objetos de contenido que se requieren para crear una WebApp que satisfaga las necesidades de los clientes Los *elementos dinámicos* del modelo de análisis describen cómo interactúan los elementos estructurales, entre ellos y con los usuarios finales

> "[Las WebApps] exitosas permiten que los clientes satisfagan mejor sus necesidades, más rápido o más barato por sí mismos, que el trabajar a través del empleado [de una compañía] para los usuarios finales."
>
> **Mark McDonald**

18.3 EL MODELO DE CONTENIDO

El *modelo de contenido* contiene elementos estructurales que proporcionan una importante visión de los requisitos de contenido para una WebApp Dichos elementos estructurales incluyen objetos de contenido (por ejemplo: texto, imágenes gráficas, fotografías, imágenes de video, audio) que se presentan como parte de la WebApp Además, el modelo de contenido incluye todas las clases de análisis· entidades visibles para el usuario que se crean o manipulan conforme éste interactúa con la WebApp Una clase de análisis incluye atributos que la describen, operaciones que afectan el comportamiento requerido de la clase y colaboraciones que permiten la comunicación de la clase con otras clases.

Al igual que otros elementos del modelo de análisis, el modelo de contenido se deriva a partir de un examen cuidadoso de los casos de uso desarrollados para la WebApp. Los casos de uso se analizan gramaticalmente para extraer objetos de contenido y clases de análisis.

18.3.1 Definición de objetos de contenido

PUNTO CLAVE

objeto de contenido cualquier artículo de información cohesivo que se presentará a un usuario final. Usualmente, los objetos de contenido se extraen de los casos de uso.

Las aplicaciones Web presentan información preexistente —llamada *contenido*— a un usuario final. El tipo y forma del contenido abarca un amplio espectro de elaboración y complejidad. El contenido puede desarrollarse antes de la implementación de la WebApp, mientras ésta se construye o mucho después de que la WebApp se encuentra en operación En cada caso, se incorpora por medio de referencias de navegación en la estructura global de la WebApp. Un *objeto de contenido* puede ser una descripción textual de un producto, un artículo que describa un evento noticioso, una fotografía de acción tomada en un cotejo deportivo, una representación animada de un logotipo corporativo, un breve video de un discurso o un recubrimiento de audio para una colección de diapositivas Powerpoint

Los objetos de contenido se extraen de los casos de uso al examinar la descripción del escenario para referencias directas e indirectas al contenido. Por ejemplo, en el caso de uso *seleccionar componentes HogarSeguro*, se encuentra la oración:

Seré capaz de obtener información descriptiva y de precios para cada componente de producto.

Aunque no existe referencia directa al contenido, está implícita. El ingeniero Web podría reunirse con el autor del caso de uso y comprender en forma más detallada lo que significa "información descriptiva y de precios". En este caso, el autor del caso de uso puede indicar que "información descriptiva" incluye 1) una descripción general del componente en un párrafo; 2) una fotografía del componente; 3) una descripción técnica del componente en varios párrafos; 4) un diagrama esquemático de componente que muestre cómo encaja en un sistema *HogarSeguro* típico; y 5) un breve video que muestre cómo instalar el componente en una configuración doméstica típica.

Es importante advertir que cada uno de estos objetos de contenido debe desarrollarse (con frecuencia a través de desarrolladores de contenido que *no* son ingenieros Web) o adquirirse para integrarlo en la arquitectura de la WebApp (analizada en el capítulo 19)

> "La Web: tanto contenido, tan poco tiempo."
>
> **Anónimo**

18.3.2 Relaciones y jerarquía de contenido

PUNTO CLAVE

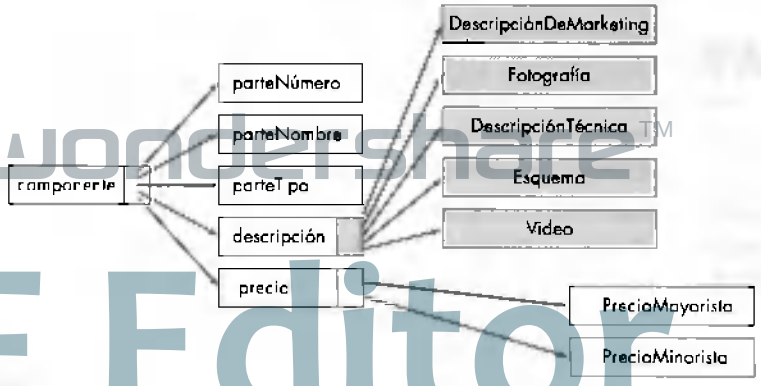
Un árbol de datos representa una jerarquía de objetos de contenido

En muchas instancias, una simple lista de objetos de contenido, pareadas con una breve descripción de cada objeto, es suficiente para definir los requisitos para el contenido que deben diseñarse e implementarse. Sin embargo, en algunos casos, el modelo de contenido puede contener diagramas de relación de entidades (capítulo 8) o *árboles de datos* [SRI01] que bosquejan las relaciones entre los objetos de contenido o la jerarquía de éste que mantiene una WebApp.

Considérese el árbol de datos creado para el componente *HogarSeguro* que se muestra en la figura 18.3. El árbol representa una jerarquía de información con que

FIGURA 18.3

Árbol de datos para un componente *HogarSeguro*



se describe el componente (más adelante se verá que, en realidad, un componente *HogarSeguro* es una clase de análisis para esta aplicación). Los artículos de datos simples o compuestos (uno o más valores de datos) se representan como rectángulos sin sombreado. Los objetos de contenido se representan como rectángulos sombreados. En la figura, **descripción** se define por medio de cinco objetos de contenido (los rectángulos sombreados). En algunos casos, uno o más de dichos objetos se refinará más todavía conforme se expanda el árbol de datos.

18.3.3 Clases[4] de análisis para WebApps

Como ya se ha señalado, las clases de análisis se derivan al examinar cada caso de uso. Por ejemplo, considérese el caso de uso preliminar: *seleccionar componentes HogarSeguro* que se presentó en la sección 18.1.2

> **Caso de uso:** *seleccionar componentes HogarSeguro*
>
> Entonces la WebApp recomendará componentes de producto (por ejemplo: paneles de control, sensores, cámaras) y otras características (por ejemplo, funcionalidad basada en PC implementada en software) para cada habitación y la entrada exterior. Si el usuario solicita opciones, la WebApp las proporcionará si existen. El usuario obtendrá información descriptiva y de precios para cada componente de producto. La WebApp creará y mostrará una factura de materiales conforme se seleccionen varios componentes. El usuario también podrá nombrar la factura de materiales y guardarla para referencia futura (véase caso de uso: *guardar configuración*).

Un rápido análisis gramatical del caso de uso identifica dos clases candidatas (subrayadas): **ComponenteDeProducto** y **FacturaDeMateriales**. En la figura 18.4 se muestra una primera descripción de cada clase.

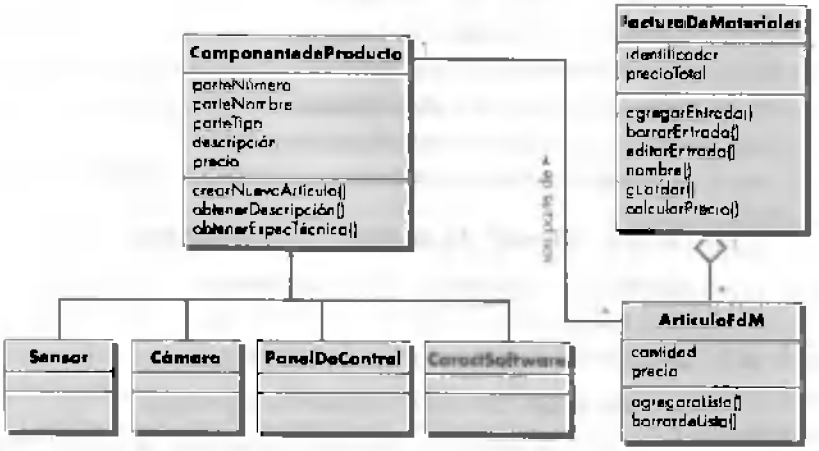
La clase **ComponenteDeProducto** abarca todos los componentes de *HogarSeguro* que se pueden comprar para personalizar el producto destinado a una instalación particular. Es una representación generalizada de **Sensor**, **Cámara**, **PaneldeControl** y **CaracterísticadeSoftware**. Cada objeto de **ComponenteDeProducto** contiene información que corresponde al árbol de datos que se muestra en la figura 18.3 para la clase. Algunos de estos atributos de clase son artículos de datos sencillos o compuestos, y otros son objetos de contenido (véase la figura 18.3). También se muestran las operaciones relevantes para la clase.

La clase **FacturaDeMateriales** abarca una lista de componentes que **cliente nuevo** ha seleccionado. **FacturaDeMateriales** es en realidad un agregado de **ArticuloFdM** (muchas instancias de **ArticuloFdM** comprenden una **FacturaDeMateriales**); una clase que construye una lista compuesta de cada componente que se comprará y de atributos específicos acerca del componente, como se muestra en la figura 18.4.

4 En el capítulo 8 se presentaron en forma detallada los mecanismos para identificar y representar las clases de análisis. Si todavía no se ha hecho, el capítulo 8 debe revisarse en este momento.

FIGURA 18.4

Clases de análisis para el caso de uso: seleccionar componentes *HogarSeguro*.



Cada caso de uso identificado para HogarSeguroInc com se analiza gramaticalmente para objetos de análisis Respecto a cada caso se uso se desarrollan mod de clase similares al descrito en esta sección.

18.4 EL MODELO DE INTERACCIÓN

La gran mayoría de las WebApps permite una "conversación" entre un usuario final y la funcionalidad, el contenido y el comportamiento de una aplicación Este *modelo de interacción* lo componen cuatro elementos: 1) casos de uso, 2) diagramas de secuencia, 3) diagramas de estado,[5] y 4) un prototipo de interfaz de usuario. Además de estas representaciones, la interacción también se representa dentro del contexto del modelo de navegación (sección 18.7).

Es posible utilizar las técnicas asociadas con el análisis de tareas (capítulo 12) para ayudarse a definir los modos de interacción del usuario

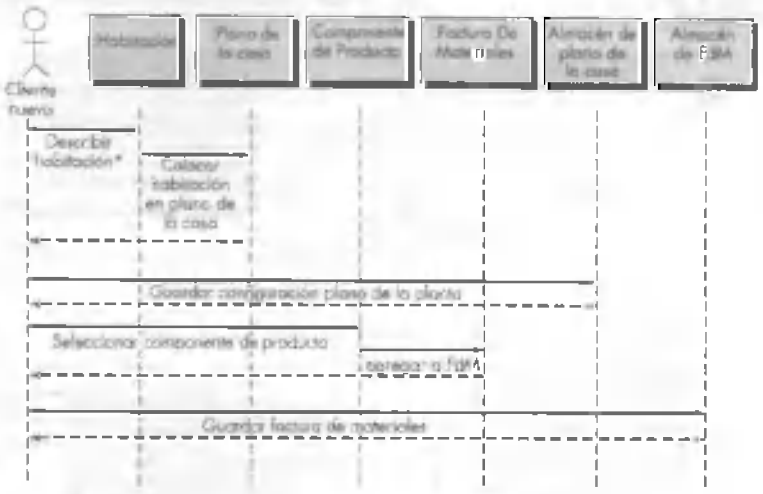
Casos de uso. Los casos de uso son el elemento dominante del modelo de interacción para las WebApps No es raro describir 100 o más casos de uso cuando se analizan, diseñan y construyen grandes y complejas WebApps. Sin embargo, un porcentaje relativamente bajo de estos casos de uso describe las principales interacciones entre las categorías de usuario final (actores) y el sistema. Otros casos de uso refinan las interacciones y proporcionan el detalle de análisis necesario para guiar el diseño y la construcción

Diagramas de secuencia. Los diagramas de secuencia UML ofrecen una representación abreviada de la forma en la cual las acciones del usuario (los elementos dinámicos de un sistema que definen los casos de uso) colaboran con las clases

5 Cada uno de éstos es una importante notación UML que se describió en el capítulo 8

18.5

...de secuencia ...el caso ...seleccionar componentes HogarSeguro.



análisis (los elementos estructurales de un sistema que definen los diagramas de clase). Dado que las clases de análisis se extraen de las descripciones de caso de uso, existe la necesidad de garantizar que hay una forma de realizar un seguimiento entre las clases definidas y los casos de uso que describen la interacción del sistema

En capítulos anteriores se apreció que los diagramas de secuencia proporcionan un vínculo entre las acciones descritas en el caso de uso y las clases de análisis (entidades estructurales). Conallen [CON00] señala esto cuando escribe: "La mezcla de elementos dinámicos y estructurales del modelo [de análisis] es el vínculo clave en la capacidad de seguimiento del modelo y se debe considerar muy seriamente."

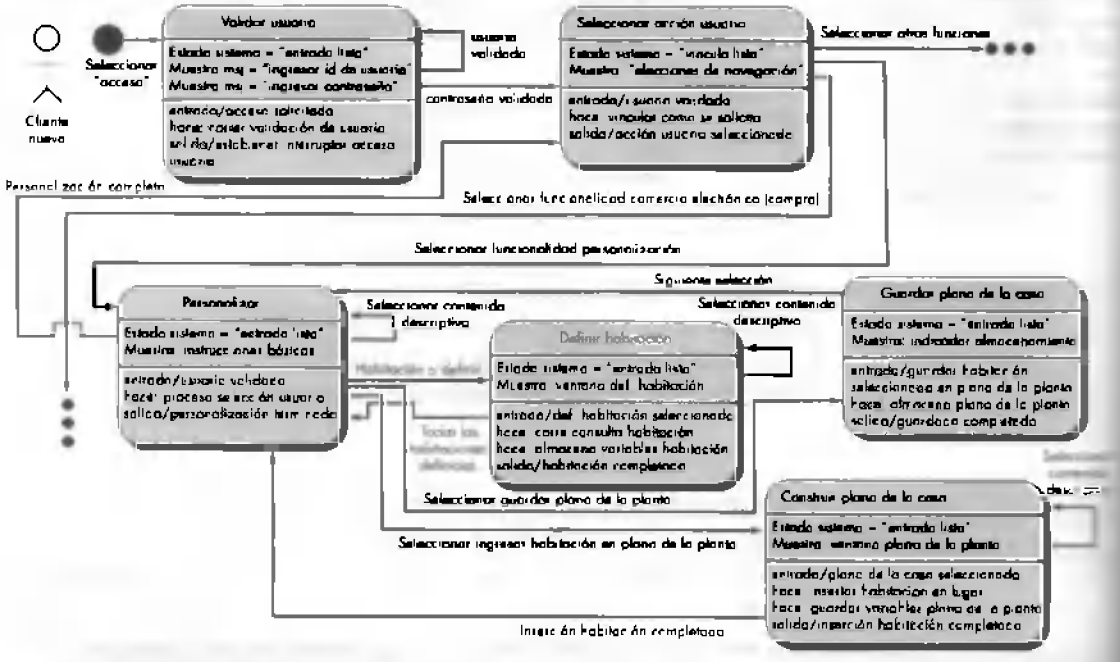
CONSEJO

...casos, en ...izquierda ...usuario), ...reproducir ...texto ...caso de uso ...representar una ...seguimiento directo.

En la figura 18.5 se muestra un diagrama de secuencia para el caso de uso *seleccionar componentes HogarSeguro*. El eje vertical del diagrama muestra las acciones que se definen dentro del caso de uso. El eje horizontal identifica las clases de análisis que se usan conforme procede el caso de uso. Por ejemplo, un cliente nuevo primero debe describir cada habitación de la casa (el asterisco a continuación de "describir habitación" indica que la acción es iterativa). Para lograr esto, el cliente nuevo responde preguntas acerca del tamaño de la habitación, puertas y ventanas, etcétera. Una vez definida una habitación, se coloca en un plano de la casa. Entonces el cliente nuevo describe la siguiente habitación o procede a la siguiente acción (guardar la configuración del plano de la planta). El movimiento a través y hacia abajo del diagrama de secuencia enlaza cada clase de análisis con las acciones del caso de uso. Si en el diagrama se pierde una acción de caso de uso, el ingeniero Web debe reevaluar la descripción de las clases de análisis para determinar si una o más clases se han perdido. Es posible crear diagramas de secuencia para cada caso de uso una vez que se definen las clases de análisis para el caso de uso.

Diagramas de estado. El diagrama de estado UML (capítulo 8) ofrece otra representación del comportamiento dinámico de la WebApp conforme sucede una inter-

FIGURA 18.6 Diagrama de estado parcial para interacción con cliente nuevo.



acción. Al igual que la mayoría de las representaciones de modelado utilizadas en la ingeniería Web (o en la ingeniería del software), el diagrama de estado puede representarse en diferentes grados de abstracción. La figura 18.6 muestra la vista superior (mayor grado de abstracción) de un diagrama de estado parcial para la interacción entre un cliente nuevo y la WebApp de HogarSeguroInc.com.

En el diagrama de estado que se muestra se identifican seis estados observables externamente: *validar usuario*, *seleccionar acción del usuario*, *personalizar*, *definir habitación*, *construir plano de la casa* y *guardar plano de la casa*. El diagrama de estado indica las acciones que se requieren para mover al cliente nuevo de un estado a otro, la información que se muestra conforme se ingresa un estado, el procesamiento que ocurre dentro de un estado y la condición de salida que provoca una transición de un estado a otro.

Puesto que los casos de uso, los diagramas de secuencia y los diagramas de estado muestran información relacionada, es razonable preguntar por qué son necesarios los tres. En algunos casos no lo son. Los casos de uso tal vez sean suficientes en algunas situaciones. Sin embargo, los casos de uso proporcionan una visión bien unidimensional de la interacción. Los diagramas de secuencia presentan una segunda dimensión que en esencia es más de procedimiento (dinámica). Los diagramas de estado proporcionan una tercera dimensión que se refiere más al comportamiento y contiene información acerca de los patrones de navegación pote-

que no proporcionan los casos de uso o el diagrama de secuencia. Cuando se usan las tres dimensiones, las omisiones o inconsistencias que pueden escapar en una dimensión se vuelven obvias cuando se examina una segunda (o tercera) dimensión. Por esta razón, los grandes WebApps complejas pueden beneficiarse de un modelo de interacción que abarque las tres representaciones.

Prototipo de la interfaz de usuario. La plantilla de la interfaz de usuario, el contenido que presenta, los mecanismos de interacción que implementa y la estética global de las conexiones usuario-WebApp, tienen mucho que ver con la satisfacción del usuario y la aceptación global de la WebApp. Aunque se puede argumentar que la creación de un prototipo de interfaz de usuario es una actividad de diseño, es una buena idea realizarla durante la creación del modelo de análisis. Mientras más rápido se pueda revisar la representación física de una interfaz de usuario, mayor será la probabilidad de que los usuarios finales obtengan lo que quieren. En el capítulo 12 se aborda el análisis de la interfaz de usuario y su diseño.

Puesto que las herramientas de desarrollo de la WebApp son abundantes, relativamente baratas y funcionalmente poderosas, es mejor crear el prototipo de la interfaz mediante tales herramientas. El prototipo debe implementar los principales vínculos de navegación y representar la plantilla de pantalla global en gran parte como será construida.

18.5 EL MODELO FUNCIONAL

El *modelo funcional* aborda dos elementos de procesamiento de la WebApp y cada uno representa un grado diferente de la abstracción de procedimiento: 1) funcionalidad observable respecto al usuario y que entrega al usuario final la WebApp, y 2) las operaciones dentro de las clases de análisis que implementan comportamientos asociados con la clase.

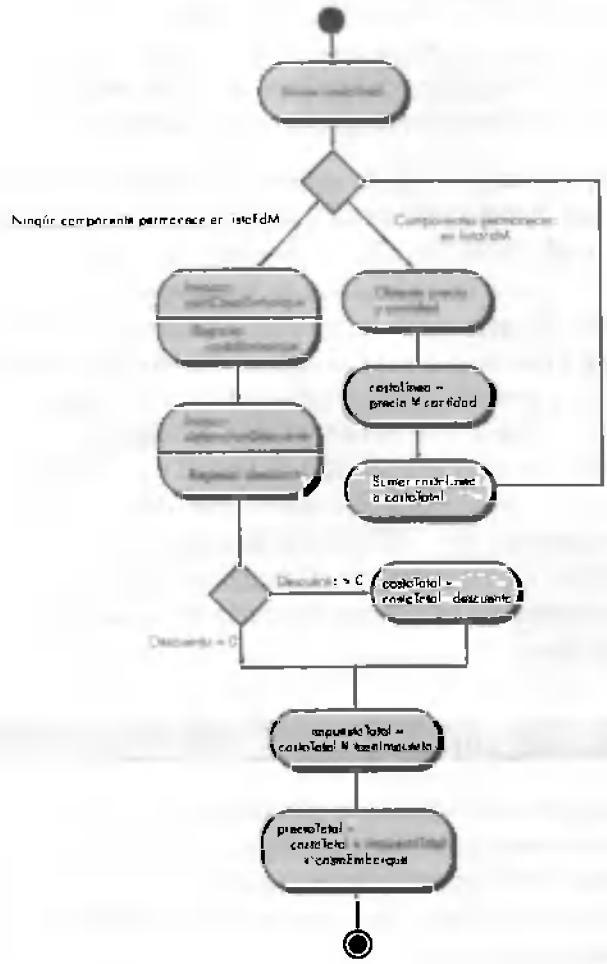
La funcionalidad observable para el usuario comprende cualesquiera funciones de procesamiento que éste inicia directamente. Por ejemplo, un sitio Web financiero puede implementar una variedad de funciones financieras (como una calculadora para fondo de matrícula universitaria o una calculadora para fondo de retiro). Dichas funciones en realidad pueden implementarse mediante operaciones dentro de las clases de análisis, pero, desde el punto de vista del usuario final, la función (más precisamente, los datos que proporciona la función) es el resultado visible.

En un grado inferior de abstracción procedimental, el modelo de análisis describe el procesamiento que realizarán las operaciones de la clase de análisis. Dichas operaciones manipulan atributos de la clase y están involucradas como clases que colaboran entre sí para lograr algún comportamiento requerido.

Sin importar el grado de abstracción procedimental, con el diagrama de actividad UML se representan detalles de procesamiento. La figura 18.7 muestra un diagrama de actividad para la operación *calcularPrecio()* que forma parte de la clase de análi-

FIGURA 18.7

Diagrama de actividad para la operación *calcularPrecio( )*.



Como alternativa, también es posible escribir una simple narración del procesamiento o representación en lenguaje de programación de diseño (capítulo 11). Sin embargo, muchas personas prefieren una representación gráfica

sis **FacturaDeMateriales** [6] Como se anotó en el capítulo 8, el diagrama de actividades es similar al diagrama de flujo, el cual ilustra el flujo de procesamiento y las decisiones lógicas del flujo. Debe notarse que, dentro del flujo procedimental, se invocan dos operaciones adicionales: *calcCostoEmbarque( )*, que calcula el costo de embarque dependiendo del método que haya elegido el cliente, y *determinarDescuento( )*, que determina cualquier descuento especial para los componentes *HogarSeguro* elegidos para comprar. Los detalles de construcción que indican cómo se solicitan estas operaciones y los detalles de la interfaz para cada operación no se consideran sino hasta que comienza el diseño WebApp.

6 Una revisión de la clase de análisis **FacturaDeMateriales** puede determinar que, con la intención de cohesionar, la operación *calcularPrecio( )* puede colocarse mejor en una clase **Facturas.** Esta sugerencia tiene mérito. Sin embargo, permanece dentro de la clase de análisis **FacturaDeMateriales** para los propósitos de este ejemplo.

18.6 EL MODELO DE CONFIGURACIÓN

Las WebApps se deben diseñar e implementar de forma que se acomoden a una diversidad de ambientes, tanto en lado del servidor como en el del cliente.[7] La WebApp puede residir en un servidor que proporcione acceso vía Internet, una Intranet o una Extranet. Se deben especificar el hardware del servidor y el ambiente del sistema operativo. Además, se deben considerar aspectos de interoperabilidad en el lado del servidor. Si la WebApp debe tener acceso a una gran base de datos o interoperar con las aplicaciones corporativas existentes en el lado del servidor, se deben especificar las interfaces apropiadas, los protocolos de comunicación y la información complementaria necesaria.

y importar todos
modelado
utilizados,
una
some-
o para
a sus usua-
no a lo
de un no-
particular.

El software del lado del cliente proporciona la infraestructura que permite el acceso a la WebApp desde la ubicación del usuario. En general, el software de navegación se utiliza para entregar el contenido y la funcionalidad de la WebApp que se descargan del servidor. Aunque existen estándares, cada navegador tiene sus propias peculiaridades. Por esta razón, la WebApp debe someterse a una amplia prueba en cada configuración de navegador que se especifique como parte del *modelo de configuración*.

En algunos casos, el modelo de configuración no es más que una lista de atributos tanto del lado del servidor como del cliente. Sin embargo, para WebApps más elaboradas, varias complejidades de configuración (por ejemplo: distribución de carga entre múltiples servidores, arquitecturas de caché, bases de datos remotas, múltiples servidores que sirven a varios objetos en la misma página Web) pueden impactar el análisis y el diseño. Es factible aprovechar el diagrama de despliegue UML (capítulo 10) en situaciones en las cuales se deban considerar arquitecturas de configuración complejas.

18.7 ANÁLISIS RELACIÓN-NAVEGACIÓN

Los elementos del modelo de análisis descritos en las secciones previas identifican los elementos de contenido y funcionalidad, junto con la forma en que se utilizan para implementar la interacción con el usuario. Conforme el análisis evoluciona en diseño, dichos elementos se vuelven parte de la arquitectura de la WebApp. En el contexto de las aplicaciones Web, cada elemento arquitectónico tiene el potencial de vincularse con todos los otros elementos arquitectónicos. Pero, conforme aumenta el número de vínculos, la complejidad de navegación a través de la WebApp también crece. Entonces, la pregunta es cómo establecer los vínculos apropiados entre los objetos de contenido y las funciones que proporcionan las capacidades que requiere el usuario.

---

7 El *lado del servidor* hospeda la WebApp y todas las características de sistema relacionadas que permiten a múltiples usuarios tener acceso a la WebApp vía una red. El *lado del cliente* proporciona un ambiente de software (por ejemplo, navegadores) que permite a los usuarios finales interactuar con la WebApp en el escritorio del usuario.

"[La navegación] no sólo es la acción de saltar de página a página, sino la idea de moverse a través de un espacio de información."

A. Reina y J. Torres

El *análisis relación-navegación* (ARN) proporciona una serie de pasos de análisis que luchan por identificar relaciones entre los elementos descubiertos como parte de la creación del modelo de análisis.[8] Yoo y Bieber [YOO00] describen un ARN del modo siguiente:

> El ARN proporciona a los analistas de sistemas una técnica sistemática para determinar la estructura de relación de una aplicación, lo que les ayuda a descubrir las relaciones potencialmente útiles en los dominios de la aplicación y que se pueden implementar como vínculos más adelante. El ARN también ayuda a determinar las estructuras de navegación apropiadas sobre estos vínculos. El ARN fomenta la comprensión de los desarrolladores de sistemas en torno a los dominios de la aplicación al ampliar y profundizar su modelo conceptual del dominio. Entonces los desarrolladores pueden mejorar su implementación al incluir vínculos, metainformación y navegación adicionales.

El enfoque ARN se organiza en cinco pasos:

- Análisis de los participantes: identifica las diversas categorías de usuario (como se describe en la sección 18.1) y establece una apropiada jerarquía de participantes.
- Análisis de elementos: identifica los objetos de contenido y los elementos funcionales de interés para los usuarios finales (como se describe en las secciones 18.3 y 18.5).
- Análisis de relaciones: describe las relaciones entre los elementos WebApp.
- Análisis de navegación: examina cómo los usuarios pueden acceder a elementos individuales o grupos de elementos.
- Análisis de evaluación: considera temas pragmáticos (por ejemplo: costo/beneficio) asociados con la implementación de las relaciones definidas con anterioridad.

Los primeros dos pasos en el enfoque ARN se trataron en párrafos anteriores de este capítulo. En las siguientes secciones se consideran métodos para establecer las relaciones entre los objetos de contenido y las funciones.

18.7.1 Análisis de relaciones: preguntas clave

Yoo y Bieber [YOO00] sugieren una lista de preguntas que un ingeniero Web o analista de sistemas deben responder acerca de cada elemento (objeto de contenido

8 Se debe señalar que el ARN es aplicable a cualquier sistema de información y originalmente se desarrolló para los sistemas hipermedia en general. Sin embargo, es posible adaptarlo muy bien a la ingeniería Web.

función) identificado dentro del modelo de análisis. La siguiente lista, adaptada para WebApps, es representativa [YOO00]:

¿Cómo se minera los elementos del modelo de análisis para comprender las relaciones entre ellos?

- ¿El elemento es miembro de una categoría de elementos más amplia?
- ¿Qué atributos o parámetros se han identificado para el elemento?
- ¿Ya existe información descriptiva acerca del elemento? Si es así, ¿dónde está?
- ¿El elemento aparece en diferentes ubicaciones dentro de la WebApp? Si es así, ¿dónde?
- ¿El elemento lo componen otros pequeños elementos? Si es así, ¿cuáles son?
- ¿El elemento es miembro de una colección de elementos mayor? Si es así, ¿cuál es y cuál es su estructura?
- ¿Al elemento lo describe una clase de análisis?
- ¿Otros elementos son similares al elemento considerado? Si es así, ¿es posible que pudieran combinarse en un elemento?
- ¿El elemento se usa en un ordenamiento específico de otros elementos? ¿Su aparición depende de otros elementos?
- ¿Otro elemento siempre sigue a la aparición del elemento considerado?
- ¿Qué condiciones previas y posteriores se deben satisfacer para utilizar el elemento?
- ¿Categorías de usuario específicas aprovechan al elemento? ¿Las diferentes categorías de usuario emplean de manera diferente al elemento? Si es así, ¿cómo?
- ¿El elemento puede estar asociado con una meta u objetivo de formulación específico? ¿Con un requisito WebApp específico?
- ¿Este elemento siempre aparece al mismo tiempo que aparecen otros elementos? Si es así, ¿cuáles son los otros elementos?
- ¿Este elemento siempre aparece en el mismo lugar (por ejemplo, misma ubicación de la pantalla o página) que otros elementos? Si es así, ¿cuáles son los otros elementos?

Las respuestas a éstas y otras preguntas ayudan al ingeniero Web a posicionar el elemento en cuestión dentro de la WebApp y a establecer relaciones entre elementos. Es posible desarrollar una relación taxonómica y categorizar cada relación identificada debido a las preguntas anotadas. El lector interesado debe remitirse a [YOO00] para más detalles.

18.7.2 Análisis de navegación

Una vez que entre los elementos se han desarrollado relaciones definidas dentro del modelo de análisis, el ingeniero Web debe considerar los requisitos que dictan cómo navegará cada categoría de usuario de un elemento (por ejemplo, objeto de conte-

nido) a otro. Los mecanismos de navegación se definen como parte del diseño. En esta etapa, los desarrolladores deben considerar requisitos de navegación globales. Las siguientes preguntas se deben plantear y responder:

¿Qué preguntas se deben plantear para comprender mejor los requisitos de navegación?

- ¿Ciertos elementos deben ser más fáciles de alcanzar (es decir, requieren menos pasos de navegación) que otros? ¿Cuál es la prioridad de presentación?
- ¿Ciertos elementos deben resaltarse para forzar a los usuarios a navegar en su dirección?
- ¿Cómo se manejarán los errores de navegación?
- ¿La navegación hacia grupos de elementos relacionados debe ser prioritaria sobre la navegación hacia un elemento específico?
- ¿La navegación se debe lograr por medio de vínculos, de acceso basado en búsqueda o por otros medios?
- ¿Ciertos elementos se deben presentar a los usuarios con base en el contexto de acciones de navegación previas?
- ¿El acceso a la navegación debe mantenerse para los usuarios?
- ¿En cada punto de la interacción del usuario debe estar disponible un mapa o menú de navegación completo (en oposición a un simple vínculo de "retroceso" o puntero dirigido)?
- ¿El diseño de la navegación debe nutrirse de los comportamientos de usuario más comúnmente esperados o mediante la importancia percibida de los elementos WebApp definidos?
- ¿Un usuario puede "almacenar" su navegación previa a través de la WebApp para un uso futuro expedito?
- ¿Para qué categoría de usuario se debe diseñar una navegación óptima?
- ¿Cómo se manejarán los vínculos externos a la WebApp? ¿Superponiendo la ventana de navegador existente? ¿Cómo una nueva ventana de navegador? ¿Cómo un marco separado?

CONSEJO

Mientras se analizan los requisitos de navegación, recuérdese que el usuario siempre debe saber dónde está y a dónde va. Para lograrlo el usuario necesita un "mapa".

Éstas y muchas otras preguntas se deben plantear y responder como parte del análisis de navegación.

El equipo de ingeniería Web y sus participantes también deben determinar los requisitos globales para la navegación. Por ejemplo, ¿se proporcionará un "mapa del sitio" para brindar a los usuarios un panorama integral de la estructura de la WebApp? ¿El usuario puede realizar un "recorrido" que subraye los elementos más importantes (objetos de contenido y funciones) disponibles? ¿Un usuario tendrá la capacidad de acceder a los objetos de contenido o funciones con base en los atributos definidos de dichos elementos (por ejemplo, un usuario tal vez desee acceder a todas las fotografías de una construcción específica o a todas las funciones que permitan el cálculo del peso)?

18.8 RESUMEN

La formulación, la recopilación de requisitos y el modelado de análisis se llevan a cabo como parte del análisis de requisitos para las WebApps. El propósito de dichas actividades es 1) describir la motivación básica (metas) y objetivos para la WebApp; 2) definir las categorías de usuarios; 3) señalar los requisitos de contenido y de función para la WebApp; y 4) establecer una comprensión básica de por qué se construirá la WebApp, quien la usará y qué problema(s) les resolverá a los usuarios.

Los casos de uso son los catalizadores para todos los análisis de requisitos y actividades de modelado. Además, pueden organizarse en paquetes funcionales, y cada paquete se valora para garantizar que es comprensible, cohesivo, libremente acoplado y jerárquicamente superficial.

Cuatro actividades de análisis contribuyen a la creación de un modelo de análisis completo: el análisis de contenido identifica todo el espectro de contenido que proporcionará la WebApp; el análisis de interacción describe la forma en la que el usuario interactúa con la WebApp; el análisis de funciones define las operaciones que se aplicarán al contenido de la WebApp y describe otras funciones de procesamiento independientes del contenido, pero necesarias para el usuario final; y el análisis de la configuración describe el ambiente de la infraestructura en la que reside la WebApp

El modelo de contenido describe el espectro de los objetos correspondientes que serán incorporados en una WebApp. Dichos objetos de contenido se deben desarrollar o adquirir para integrarlos en la arquitectura de la WebApp. Es factible utilizar un árbol de datos para representar la jerarquía de un objeto de contenido. Las clases de análisis (derivadas de los casos de uso) proporcionan otros medios para representar los objetos clave que manipulará la WebApp.

El modelo de interacción se construye con casos de uso, diagramas de secuencia UML y diagramas de estado UML para describir la "conversación" entre el usuario y la WebApp. Además, se construye un prototipo de la interfaz que auxilie en el desarrollo de la plantilla y los requisitos de navegación.

El modelo funcional describe las funciones observables para el usuario y las operaciones de clase que emplean el diagrama de actividad UML. El modelo de configuración describe el ambiente que requerirá la WebApp, tanto en el lado del servidor como en el del cliente del sistema.

El análisis de relación-navegación identifica las relaciones entre el contenido y los elementos funcionales, definidos en el modelo de análisis, y establece requisitos para definir vínculos de navegación apropiados a través del sistema. Una serie de preguntas ayudan a establecer relaciones e identificar características que influirán sobre el diseño de navegación.

REFERENCIAS

[CON00] Conallen, J., *Building Web Applications with UML*, Addison-Wesley, 2000

[FRA01] Franklin, S., "Planning Your Web Site with UML", *webReview*, disponible en http://www.webreview.com/2001/05_18/developers/index01.shtml

[SRI01] Sridhar, M. y N. Mandyam, "Effective Use of Data Models in Building Web Applications" 2001, disponible en http //www2002.org/CDROM/alternate/698/.

[YOO99] Yoo, J. y M. Bieber, "A Systematic Relationship Analysis for Modeling Information Domains", 1999, se puede descargar de http://citeseer.nj.nec.com/312025.html

[YOO00] Yoo, J y M. Bieber, "Toward a Relationship Navigation Analysis", en *Proc. 33rd Hawaii Conf. On System Sciences*, vol 6, IEEE, enero de 2000, se puede descargar de www.cs.njit.edu/~bieber/pub/hicss00/INWEB02.pdf.

Problemas y puntos a considerar

18.1 Con base en el gran abanico de recursos acerca del desarrollo de software ágil disponible en la Web, investíguese un poco y establézcase un razonamiento en contra del modelado de análisis para las WebApps ¿Se considera que la argumentación resultante se aplica en todos los casos?

18.2 Si fuese forzoso a llevar a cabo un "modelado de análisis ligero" (es decir, modelado de análisis mínimo), ¿qué representaciones, diagramas e información se definirían durante esta actividad de ingeniería Web?

18.3 Mediante un diagrama similar al mostrado en la figura 18 1, establézcase una jerarquía de usuario para (a) un sitio Web de servicios financieros o (b) un sitio Web de venta de libros.

18.4 ¿Qué representa un paquete de caso de uso?

18.5 Los casos de uso o los paquetes de caso de uso se valoran para garantizar que son *comprensibles, cohesivos, libremente acoplados y jerárquicamente superficiales* Describase con palabras propias qué significan estos términos

Elíjase una WebApp que se visite regularmente de una de las siguientes categorías· (a) noticias o deportes, (b) entretenimiento, (c) comercio electrónico, (d) juegos, (e) relacionados con la computación, (f) una WebApp que recomienden los profesores. Realícense las actividades indicadas en los problemas del 18.6 al 18.12·

18.6 Desarróllense uno o más casos de uso que describan un comportamiento de usuario específico para la WebApp.

18.7 Represéntese una jerarquía de contenido parcial y definanse al menos tres clases de análisis para la WebApp.

18.8 Desarróllese un diagrama de secuencia UML y un diagrama de estado UML que describa una interacción específica con la WebApp.

18.9 Considérese la interfaz existente de la WebApp Hágase un prototipo de cambio a la interfaz que se considere susceptible de mejorar

18.10 Elíjase una función observable para el usuario que ofrezca la WebApp y modélese mediante un diagrama de actividad UML.

18.11 Elíjase un objeto de contenido o función que sea parte de la arquitectura de la WebApp y respóndanse las preguntas relación-navegación mencionadas en la sección 18.7.1.

18.12 Considérese la WebApp existente y respóndanse las preguntas relación-navegación mencionadas en la Sección 18 7.2.

Otras lecturas y fuentes de información

Muchos libros dedicados al modelado de análisis para software convencional (con particular énfasis en los casos de uso y la notación UML) contienen mucha información útil susceptible de adaptarse fácilmente a la ingeniería Web. Los casos de uso forman los cimientos del modelado

de análisis para las WebApps. Los libros de Kulak y sus colegas (*Use Cases Requirements in Context*, segunda edición, Addison-Wesley, 2004), Bittner y Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn (*Writing Effective Use Cases*, Addison-Wesley, 2001), Armour y Miller (*Advanced Use-Case Modeling: Software Systems*, Addison-Wesley, 2000), Rosenberg y Scott (*Use Case Driven Object Modeling with UML: A Practical Approcah*, Addison-Wesley, 1999) y Schneider, Winters y Jacobson (*Applying use Cases. A practical Guide*, Addison-Wesley, 1998) ofrecen una guía valiosa en la creación y empleo de este importante mecanismo de representación de requisitos. Valiosas discusiones de UML han escrito Arlow y Neustadt (*UML and the Unified Process*, Addison-Wesley, 2002), Schmuller (*Teach Yourself UML*, Sams Publishing, 2002), Booch y sus colegas (*The UML User Guide*, Addison-Wesley, 1998), y Rumbaugh y sus colegas (*The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998)

Los libros dedicados al diseño de sitios Web con frecuencia contienen uno o dos capítulos que abordan temas de análisis (aunque usualmente son discusiones superficiales). Los siguientes libros contienen uno o más aspectos del análisis dentro del contexto de la ingeniería Web: Van Duyne y sus colegas (*The Design of Sites*, Addison-Wesley, 2002), Rosenfeld y Morville (*Information Architecture for the World Wide Web*, O'Reilly & Associates, 2002), Wodtke (*Information Architecture*, New Riders Publishing, 2002), Garret (*The Elements of User Experience: User Centered Design for the Web*, New Riders Publishing, 2002), Niederst (*Web Design in a Nutshell*, O'Reilly & Associates, 2001), Lowe y Hall (*Hypertext and the Web: An Engineering Approach*, Wiley 1999), y Powell (*Web Site Engineering*, Prentice-Hall, 1998) ofrecen una cobertura razonablemente completa Norris, West y Watson (*Media Engineering. A Guide to Developing Information Products*, Wiley, 1997), Navarro y Khan (*Effective Web Design: Master the Essentials*, Sybex, 1998) y Fleming y Koman (*Web Navigation: Designing the User Experience*, O'Reilly & Associates, 1998) proporcionan guía adicional para análisis y diseño.

En Internet hay disponible una gran variedad de fuentes de información acerca del modelado de análisis para ingeniería Web Una lista actualizada de referencias en la World Wide Web se encuentra bajo "software engineering resources" en el sitio Web de SEPA: **http://www.mhhe.com/pressman.**

CAPÍTULO

19 MODELADO DE DISEÑO PARA APLICACIONES WEB

CONCEPTOS CLAVE

En su autorizado libro acerca del diseño Web, Jakob Nielsen [NIE00] afirma: "En esencia, existen dos enfoques básicos del diseño: el ideal artístico de expresarse uno mismo y el ideal de ingeniería de resolver un problema para un cliente". Durante la primera década del desarrollo Web, la idea artística fue el enfoque que eligieron muchos desarrolladores. El diseño ocurrió en una forma *ad hoc* y usualmente era dirigido conforme se generaba el HTML. El diseño evolucionó de una visión artística que en sí misma evolucionó conforme ocurría la construcción de la WebApp.

Incluso en la actualidad, los defensores "radicales" del desarrollo de software ágil (capítulo 4) utilizan las aplicaciones Web como cartel de niños para "diseño limitado". Argumentan que la inmediatez y la volatilidad de las WebApps desalientan el diseño formal, que el diseño evoluciona conforme se construye (codifica) una aplicación y que se debe gastar relativamente poco tiempo en la creación de un modelo de diseño detallado. Este argumento tiene mérito, pero sólo para WebApps relativamente simples. Cuando el contenido y la función son complejos, cuando el tamaño de la WebApp abarca cientos de objetos de contenido, funciones y clases de análisis, cuando el éxito de la WebApp tendrá un impacto directo sobre el éxito del negocio, el diseño no puede ni debe ser tomado a la ligera.

Esta realidad conduce al segundo enfoque de Nielsen: "el ideal de ingeniería de resolver un problema para un cliente". La ingeniería Web adopta esta filosofía, y un enfoque más riguroso del diseño WebApp permite a los desarrolladores lograrlo.

UN VISTAZO RÁPIDO

¿Qué es? El diseño de WebApps abarca actividades técnicas y otras que no lo son. La visión y el sentido del contenido se desarrollan como parte del diseño gráfico; la plantilla estética de la interfaz de usuario se crea como parte del diseño de la interfaz; y la estructura técnica de la WebApp se modela como parte del diseño arquitectónico y de navegación. En toda instancia se debe crear un modelo de diseño antes de que comience la construcción, pero un buen ingeniero Web reconoce que el diseño evolucionará mientras más se conozca acerca de los requisitos de los participantes conforme se construya la WebApp.

¿Quién lo hace? Los ingenieros Web, diseñadores gráficos, desarrolladores de contenido y otros participantes colaboran en la creación de un modelo de diseño para la ingeniería Web.

¿Por qué es importante? El diseño permite a un ingeniero Web crear un modelo que puede valorarse en calidad y mejorarse antes de que se generen el contenido y el código, se realicen pruebas y se involucren muchos usuarios finales. El diseño es el lugar donde se establece la calidad de la WebApp.

¿Cuáles son los pasos? El diseño Web abarca seis grandes pasos a los cuales alimenta la información obtenida durante el modelado del análisis.

El diseño de contenido utiliza información contenida dentro del modelo de análisis como una base para establecer el diseño de los objetos de contenido y sus relaciones. El diseño estético (también llamado diseño gráfico) establece la apariencia y el sentimiento que observa el usuario final. El diseño arquitectónico se enfoca sobre la estructura hipermedia global de todos los objetos de contenido y funciones. El diseño de la interfaz establece la plantilla global y los mecanismos de interacción que definen la interfaz del usuario. El diseño de navegación define cómo navega el usuario final a través de la estructura hipermedia, y el diseño de componentes representa la estructura interna detallada de los elementos funcionales de la WebApp.

¿Cuál es el producto obtenido? Un modelo de diseño que abarque temas de diseño de contenido, estética, arquitectura, interfaz, navegación y al nivel de componente es el producto de trabajo primario del diseño de ingeniería Web.

¿Cómo puedo estar seguro de que lo he hecho correctamente? El equipo de ingeniería Web (y algunos participantes seleccionados) revisa cada elemento del modelo de diseño con la finalidad de descubrir errores, inconsistencias u omisiones. Además, se consideran soluciones alternativas, y también se valora el grado en el que el modelo de diseño actual conducirá a una implementación efectiva.

19.1 Temas de diseño para ingeniería Web

Cuando se aplica el diseño dentro del contexto de la ingeniería Web, se deben considerar cuestiones tanto genéricas como específicas. Desde un punto de vista genérico, el diseño resulta en un modelo que guía la construcción de la WebApp. El modelo de diseño, sin importar su forma, debe contener suficiente información para reflejar cómo habrán de traducirse los requisitos de los participantes (definidos en un modelo de análisis) en contenido y código ejecutable. Pero el diseño también debe ser específico. Debe abordar atributos clave de una WebApp en una forma que permita al ingeniero Web construir y ponerla a prueba de manera efectiva.

19.1.1 Diseño y calidad de una WebApp

En capítulos anteriores se señaló que el diseño es la actividad de ingeniería que conduce a un producto de gran calidad. Esto conduce a una pregunta recurrente que se presenta en toda las disciplinas de ingeniería: ¿qué es calidad? En esta sección se examinará la respuesta en el contexto de la ingeniería Web.

Toda persona que haya navegado en la Web o usado una Intranet corporativa tiene una opinión acerca de lo que hace una "buena" WebApp. Los puntos de vista individuales varían enormemente. Algunos usuarios disfrutan los gráficos que bailan, otros quieren texto simple. Algunos solicitan información copiosa, otros desean una presentación abreviada. A algunos les gustan las herramientas analíticas sofisticadas o los accesos a las bases de datos, a otros les gustan las cosas simples. De hecho, la percepción del usuario de lo que es "bueno" (y la resultante aceptación o rechazo de la WebApp como consecuencia) puede ser más importante que cualquier discusión técnica de la calidad de la WebApp.

¿Pero cómo se aprecia la calidad de la WebApp? ¿Qué atributos debe exhibir para lograr ser buena a los ojos de los usuarios finales y al mismo tiempo mostrar las características técnicas de calidad que permitirán a un ingeniero Web corregir, adaptar, mejorar y apoyar la aplicación a largo plazo?

En realidad, todas las características generales de la calidad de software tratadas en los capítulos 9, 15 y 26 se aplican a la WebApps. Sin embargo, las más relevantes de dichas características —facilidad de uso, funcionalidad, confiabilidad, eficiencia y facilidad de mantenimiento— proporcionan una base útil para valorar la calidad de los sistemas basados en Web.

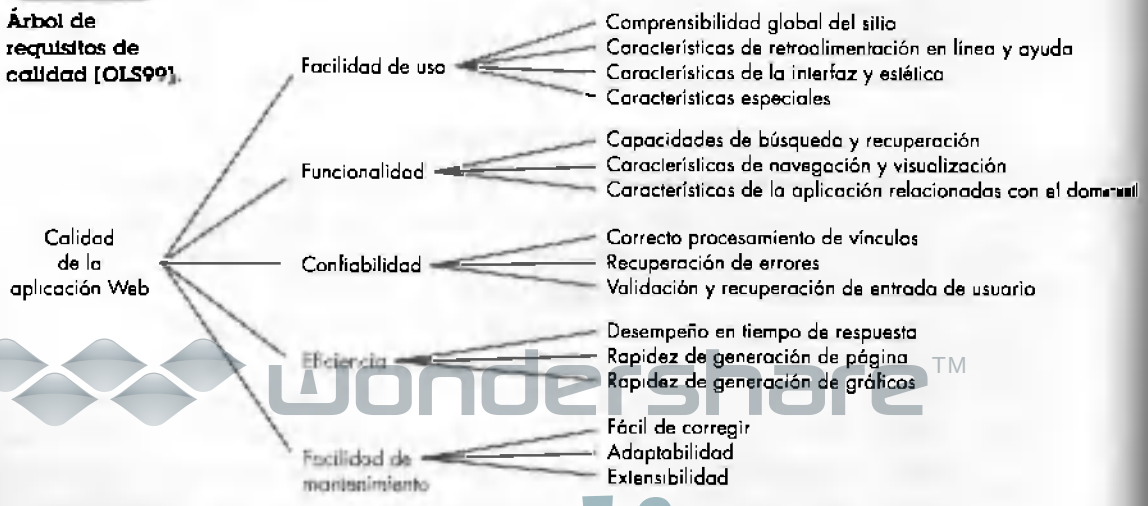
> "Si los productos se diseñan para encajar mejor en las tendencias naturales del comportamiento humano, entonces la gente estará más satisfecha, más completa y será más productiva."
>
> **Susan Weinschenk**

Olsina y sus colegas [OLS99] han preparado un "árbol de requisitos de calidad" que identifica un conjunto de atributos técnicos —facilidad de uso, funcionalidad, confiabilidad, eficiencia y facilidad de mantenimiento— que conducen a WebApps de gran calidad.[1] La figura 19.1 resume su trabajo. Los criterios anotados en la figura son de particular interés para los ingenieros Web que deben diseñar, construir y mantener las WebApps a largo plazo.

Offutt [OFF02] extiende los cinco principales atributos de calidad anotados en la figura 19.1 al agregar los atributos siguientes:

FIGURA 19.1

Árbol de requisitos de calidad [OLS99].



1 Estos atributos de calidad son muy similares a los que se presentan en los capítulos 9, 15 y 26 lo tanto, se deduce que las características de calidad son universales para todo el software.

¿Cuáles son los principales atributos de calidad para las WebApps?

Seguridad. Las WebApps se han convertido en una parte integral de las bases de datos cruciales del gobierno y las empresas. Las aplicaciones de comercio electrónico extraen y luego almacenan información confidencial de los clientes. Por éstas y muchas otras razones, la seguridad de las WebApps es primordial en muchas situaciones. La medida clave de la seguridad es la habilidad de la WebApp y su ambiente de servidor de rechazar el acceso no autorizado e impedir un franco ataque malévolo. Un análisis detallado de la seguridad WebApp está más allá del alcance de este libro. El lector interesado debe consultar [MCC01], [NOR02] o [KAL03].

Disponibilidad. Incluso la mejor WebApp no satisfará las necesidades de los usuarios si no está disponible. En un sentido técnico, la disponibilidad es la medida del porcentaje del tiempo que una WebApps está disponible para usarla. El usuario final común espera que las WebApps estén disponibles las 24 horas de todos los días del año. Algo menos es considerado inaceptable.[2] Pero "a tiempo" no es el único indicador de disponibilidad. Offutt [OFF02] sugiere que "usar características disponibles sólo en un navegador o una plataforma" hace que la WebApp no esté disponible para quienes tengan una configuración de navegador y plataforma diferente. El usuario invariablemente se irá a otra parte.

Escalabilidad. ¿La WebApp y su ambiente de servidor pueden escalarse para manejar 100, 1 000, 10 000 o 100 000 usuarios? ¿La WebApp y los sistemas con los cuales está conectada manejan variaciones significativas en el volumen o la capacidad de respuesta caerá catastróficamente (o cesará por completo)? No es suficiente construir una WebApp exitosa. Es igualmente importante construir una WebApp que pueda acomodar el peso del éxito (significativamente más usuarios finales) y volverse todavía más exitosa.

Tiempo en el mercado. Aunque en sentido técnico el tiempo en el mercado no es un verdadero atributo de calidad, es una medida de calidad desde un punto de vista de los negocios. La primera WebApp en el mercado usualmente captura un número desproporcionado de usuarios finales.

Cientos de miles de páginas Web están disponibles para quienes busquen información en la World Wide Web. Incluso las búsquedas Web mejor dirigidas resultan en una avalancha de contenido. Con tantas fuentes de información de las cuales elegir, ¿cómo valora el usuario la calidad (por ejemplo, veracidad, precisión, integridad, oportunidad) del contenido que se presenta dentro de una WebApp? Tillman [TIL00] sugiere un conjunto de criterios útil para valorar la calidad del contenido:

- ¿El ámbito y la profundidad del contenido se pueden determinar con facilidad para asegurar que satisfacen las necesidades del usuario?

2 Desde luego, esta expectativa es irreal. Las grandes WebApps deben planificar el "periodo de inactividad" para reparaciones y actualizaciones.

INFORMACIÓN

Lista de verificación de la calidad del diseño de la WebApp

La siguiente lista de verificación, adaptada de la información presentada en Webreference com, proporciona un conjunto de preguntas que ayudarán tanto a los ingenieros Web como a los usuarios finales a valorar la calidad global de una WebApp:

- ¿El contenido, la función o las opciones de navegación pueden ajustarse a las preferencias del usuario?
- ¿El contenido o la funcionalidad se pueden personalizar al ancho de banda en el cual se comunica el usuario?
- ¿Las gráficas y los otros medios que no son textuales se han usado de manera apropiada? ¿El tamaño de los archivos gráficos está optimizado para que se despliegue con eficiencia?
- ¿Las tablas están organizadas y dimensionadas en una forma que las hace comprensibles y que se despliegue eficientemente?
- ¿El HTML está optimizado para eliminar ineficiencias?
- ¿El diseño global de la página facilita la lectura y la navegación?
- ¿Todos los punteros (vínculos) proporcionan vínculos con información interesante para los usuarios?
- ¿Es probable que la mayoría de los vínculos persistan en la Web?
- ¿La WebApp está instrumentada con utilidades de gestión del sitio que incluyan herramientas para rastrear su uso, prueba de vínculos, búsqueda local y seguridad?

¿Qué se debería considerar cuando se valore el contenido de calidad?

- ¿Los antecedentes y la jerarquía de los autores del contenido se pueden identificar fácilmente?
- ¿Es posible determinar la precisión del contenido, la última actualización y lo que fue actualizado?
- ¿El contenido y su ubicación son estables (es decir, permanecerán en la URL de referencia)?

Además de estas preguntas relacionadas con el contenido, se pueden añadir las siguientes:

- ¿El contenido es creíble?
- ¿El contenido es único? Esto es, ¿la WebApp proporciona algún beneficio único a quienes la usen?
- ¿El contenido es valioso para la comunidad de usuarios a que se dirige?
- ¿El contenido está bien organizado? ¿Está en un índice? ¿Es fácilmente accesible?

La lista de verificación anotada en esta sección sólo representa una pequeña muestra de los asuntos que deben abordarse conforme evolucione el diseño de una WebApp. Una meta importante de la ingeniería Web es desarrollar sistemas en los que se proporcionen respuestas afirmativas a todas las preguntas relacionadas con la cali-

"Sólo porque puedes, no significa que debes."

Jean Kaiser

19.1.2 Metas de diseño

En su columna regular acerca del diseño Web, Jean Kaiser [KAI02] sugiere las siguientes metas de diseño, que son aplicables virtualmente a toda WebApp sin importar el dominio, tamaño o complejidad de la aplicación:

Simplicidad. Aunque pueda parecer pasada de moda, la expresión "todo con moderación" se aplica a las WebApps. Existe una tendencia entre algunos diseñadores a proporcionar al usuario final "demasiado": contenido exhaustivo, efectos visuales extremos, animación entrometida, enormes páginas Web, la lista es larga Es mejor luchar por la moderación y la simplicidad

Consistencia. Esta meta de diseño se aplica virtualmente a cada elemento del modelo de diseño. El contenido se debe construir de manera consistente (por ejemplo, el formato del texto y los estilos de fuente deben ser los mismos a lo largo de todos los documentos de texto; el arte gráfico debe tener una apariencia consistente, esquema de color y estilo). El diseño gráfico (estética) debe presentar una apariencia consistente en todas las partes de la WebApp. El diseño arquitectónico debe establecer plantillas que conduzcan a una estructura hipermedia consistente. El diseño de interfaz debe definir modos consistentes de interacción, navegación y despliegue de contenido. Los mecanismos de navegación deben usarse de manera consistente a través de todos los elementos de la WebApp

Identidad. La estética, la interfaz y el diseño de navegación de una WebApp deben ser consistentes con el dominio de la aplicación para la cual se va a construir. Un sitio Web para un grupo hip-hop indudablemente tendrá una apariencia y un sentido diferente al de una WebApp diseñada para una compañía de servicios financieros. La arquitectura de WebApp será completamente diferente, las interfases se construirán para acomodar diferentes categorías de usuarios, la navegación estará organizada para lograr diferentes objetivos. Un ingeniero Web (y otros contribuyentes de diseño) deberá trabajar para establecer una identidad para la WebApp por medio del diseño.

Robustez. Con base en la identidad establecida, usualmente una WebApp hace una "promesa" implícita al usuario El usuario espera contenido y funciones robustas que sean relevantes para sus necesidades. Si dichos elementos están perdidos o son insuficientes es probable que la WebApp fracase.

Navegabilidad. Ya se ha señalado que la navegación debe ser simple y consistente También debe estar diseñada de modo que sea intuitiva y predecible Esto es, el usuario debe entender cómo moverse por la WebApp sin tener que buscar vínculos o instrucciones de navegación

Apariencia visual. De todas las categorías de software, las aplicaciones Web son incuestionablemente las más visuales, las más dinámicas y sin duda las más estéticas Es indudable que la belleza (apariencia visual) está en el ojo del observador, pero muchas características de diseño (por ejemplo, la apariencia y sentido del conteni-

do, la plantilla de la interfaz, la coordinación del color, el equilibrio del texto, los gra ficos y otros medios audiovisuales, los mecanismos de navegación) si contribuyen a aspecto visual

Compatibilidad. Una WebApp se utilizará en una diversidad de ambientes (por ejemplo, diferentes equipos, tipos de conexión a Internet, sistemas operativos, nave gadores) y se debe diseñar para que sea compatible con cada uno.

> "Para algunos, el diseño Web se enfoca en la apariencia y sentido visuales... para otros, el diseño Web trata de la es tructuración de información y la navegación a través del espacio del documento. Otros incluso pueden considerar que el diseño Web trata acerca de la tecnología empleada para construir aplicaciones Web interactivas. En realidad, el dise ño incluye todos estos factores y araso más."
>
> **Thomas Powell**

19.2 PIRÁMIDE DEL DISEÑO IWEB

¿Qué es diseño en el contexto de la ingeniería Web? Esta pregunta simple es más fícil de responder de lo que uno puede creer. El diseño conduce a un modelo que contiene la mezcla adecuada de estética, contenido y tecnología. La mezcla varía dependiendo de la naturaleza de la WebApp, y, como consecuencia, las actividades de diseño también variarán.

La figura 19.2 muestra una pirámide de diseño para la ingeniería Web. Cada nivel de la pirámide representa una de las siguientes actividades de diseño:

PUNTO CLAVE

La IWeb abarca seis diferentes tipos de diseño. Cada uno contribuye a la calidad global de la WebApp.

- Diseño de la interfaz: describe la estructura y organización de la interfaz de usuario. Incluye una representación de la plantilla de pantalla, una definición de los modos de interacción y una descripción de los mecanismos de navegación.
- Diseño estético: también llamado diseño gráfico, describe la "apariencia y sentimiento" de la WebApp. Incluye esquemas de color, plantilla geométrica, tamaño de texto, fuente y ubicación, uso de gráficos y decisiones estéticas relacionadas.
- Diseño de contenido: define la plantilla, la estructura y el bosquejo de todos los contenido que se presenta como parte de la WebApp. Establece las relaciones entre los objetos de contenido.
- Diseño de navegación: representa el flujo de navegación entre los objetos de contenido y para todas las funciones de la WebApp.
- Diseño arquitectónico: identifica la estructura hipermedia global para la WebApp.
- Diseño de componentes: desarrolla la lógica de procesamiento detallado que se requiere para implementar componentes funcionales.

En las secciones que siguen se consideran con mayor detalle cada una de las actividades de diseño.

19.3 Diseño de la interfaz de la WebApp[3]

Toda interfaz del usuario —ya sea diseñada para una WebApp, una aplicación de software tradicional, un producto de consumo o un dispositivo industrial— debe presentar las siguientes características: fácil de usar, fácil de aprender, fácil de navegar, intuitiva, consistente, eficiente, libre de errores y funcional. Debe ofrecer al usuario final una experiencia satisfactoria y gratificante. Los conceptos, principios y métodos de diseño de la interfaz brindan al ingeniero Web las herramientas requeridas para lograr esta lista de atributos.

En el capítulo 12 se observó que el diseño de la interfaz comienza no con una consideración de la tecnología, sino más bien con un cuidadoso examen del usuario final. Durante el modelado de análisis para la ingeniería Web (Capítulo 18), se desarrolla una jerarquía de usuario. Cada categoría de usuario puede tener necesidades sutilmente diferentes, tal vez quiera interactuar con la WebApp en diferentes formas y quizá requiera funcionalidad y contenido únicos. Esta información se deriva durante el análisis de requisitos, pero se revisa como el primer paso en el diseño de la interfaz.

> "Si un sitio es perfectamente utilizable pero carece de un estilo de diseño elegante y adecuado, fracasará."
>
> **Curt Cloninger**

Dix [DIX99] argumenta que un ingeniero Web debe diseñar una interfaz de modo que responda tres preguntas primarias para el usuario final:

3 La mayoría de, si no es que todas, las directrices presentadas en el capítulo 12 se aplican igualmente al diseño de interfases WebApp. Si todavía no lee el capítulo 12, hágalo en este momento.

Si es probable que los usuarios puedan entrar a su WebApp en varias ubicaciones y niveles en la jerarquía de contenido, asegúrese de diseñar cada página con características de navegación que conduzcan al usuario a los otros puntos de interés.

¿Dónde estoy? La interfaz debe 1) ofrecer una indicación de que se ha tenido acceso a la WebApp[4] y 2) informar al usuario de su ubicación en la jerarquía de contenido.

¿Qué puedo hacer ahora? La interfaz siempre debe ayudar al usuario a entender sus opciones actuales: qué funciones están disponibles, qué vínculos están vivos, qué contenido es relevante.

¿Dónde he estado, a dónde voy? La interfaz debe facilitar la navegación. En consecuencia, debe proporcionar un "mapa" (implementado en una forma fácil de entender) de dónde ha estado el usuario y qué rutas puede tomar para moverse a cualquier parte dentro de la WebApp.

La interfaz de una WebApp efectiva debe proporcionar respuestas a cada una de estas preguntas conforme el usuario final navega a través del contenido y la funcionalidad.

19.3.1 Principios y directrices del diseño de la interfaz

Bruce Tognozzi [TOG01] define un conjunto de características fundamentales que deben presentar todas las interfaces y, al hacerlo, establece una filosofía que debe seguir todo diseñador de interfaz de WebApp:

Una buena interfaz WebApp es comprensible e indulgente, y ofrece al usuario una sensación de control.

> Las interfaces efectivas son visualmente aparentes e indulgentes, e implantan en sus usuarios una sensación de control. Los usuarios ven rápidamente la envergadura de sus opciones, comprenden cómo lograr sus metas y hacen su trabajo
>
> Las interfaces efectivas no preocupan al usuario con los trabajos internos del sistema. El trabajo se guarda de manera cuidadosa y continua, con la opción total de que el usuario deshaga cualquier actividad en cualquier tiempo.
>
> Las aplicaciones y servicios efectivos realizan un máximo de trabajo mientras demandan un mínimo de información a los usuarios

Con la finalidad de diseñar interfaces que muestren dichas características, Tognozzi [TOG01] identifica un conjunto de principios de diseño primordiales:[5]

Anticipación: una WebApp se debe diseñar de modo que anticipe el siguiente movimiento del usuario. Por ejemplo, considere una WebApp de soporte al cliente desarrollada para un fabricante de impresoras para computadora. Un usuario ha solicitado un objeto de contenido que presenta información acerca de un controlador de impresora para un sistema operativo lanzado recientemente. El diseñador de la WebApp debe anticipar que el usuario pueda solicitar una descarga del controlador.

[4] Todas las personas han marcado alguna página de un sitio Web, sólo para volver a visitarla mas tarde y no tener que dar indicaciones del sitio Web o del contexto de la página (así como para evitar moverse hacia otra ubicación dentro del sitio).

[5] Los principios originales de Tognozzi se han adaptado y extendido con el fin de aprovechar este libro. Véase [TOG01] para mayores detalles acerca de estos principios.

y debe proporcionar facilidades de navegación que permitan hacerlo sin solicitarle al usuario una búsqueda de esta capacidad

Comunicación: la interfaz debe comunicar el estado de cualquier actividad que haya iniciado el usuario. La comunicación puede ser obvia (por ejemplo, un mensaje de texto) o sutil (por ejemplo, una hoja de papel que se mueva a través de una impresora para indicar que la impresión está en camino). La interfaz también debe comunicar el estado del usuario (por ejemplo, la identificación del usuario) y la ubicación dentro de la jerarquía de contenido de la WebApp

Consistencia: el uso de los controles de navegación, menús, íconos y estética (por ejemplo, color, forma, plantilla) deben ser consistentes a través de toda la WebApp. Por ejemplo, si el texto subrayado de azul implica un vínculo de navegación, el contenido nunca debe incorporar texto subrayado en azul que no implique un vínculo. Toda característica de la interfaz debe responder en una forma que sea consistente con las expectativas del usuario [6]

Autonomía controlada: la interfaz debe facilitarle al usuario el movimiento a través de toda la WebApp, pero lo debe hacer en una forma que refuerce las convenciones de navegación establecidas para la aplicación. Por ejemplo, la navegación hacia porciones seguras de la WebApp se deben controlar con la identificación del usuario y su contraseña, y no debe existir mecanismo de navegación que permita al usuario dar la vuelta a dichos controles.

Eficiencia: el diseño de la WebApp y su interfaz deben optimizar la eficiencia laboral del usuario, no la eficiencia del ingeniero Web que la diseña y la construye o el ambiente cliente-servidor que la ejecuta. Tognozzi [TOG01] señala esto cuando escribe: "Esta simple verdad es por lo que es importante para todos los involucrados en un proyecto de software el apreciar la importancia de hacer propia la meta de productividad del usuario y entender la diferencia vital entre construir un sistema eficiente y fortalecer a un usuario eficiente."

Flexibilidad: la interfaz debe ser lo suficientemente flexible como para permitir que algunos usuarios realicen tareas directamente y otros exploren la WebApp en una forma un tanto aleatoria. En todo caso, debe permitirle al usuario entender dónde está y ofrecerle la funcionalidad para que pueda deshacer los errores y volver a trazar las rutas de navegación mal elegidas.

Enfoque: la interfaz de la WebApp (y el contenido que presenta) debe enfocarse en la(s) tarea(s) importante(s) para el usuario. En toda hipermedia existe una tendencia para dirigir al usuario hacia contenido mal relacionado. ¿Por qué? ¡Porque es muy fácil hacerlo! El problema es que el usuario rápidamente se puede perder en muchas capas de información de apoyo y perder el sitio del contenido original que quería en primer lugar.

6 Tognozzi [TOG01] señala que la única forma de garantizar que las expectativas del usuario se comprendan adecuadamente es mediante una amplia prueba por parte del usuario (capítulo 20)

Ley de Fitt: "El tiempo para adquirir un objetivo es una función de la distancia a la que se halla y de su tamaño" [TOG01] Con base en un estudio realizado en la década de 1950 [FIT54], la ley de Fitt "es un método efectivo de modelar rápidos movimientos dirigidos, donde un apéndice (como una mano) parte del reposo en una posición de inicio específica y se mueve hacia el reposo dentro de una área establecida como objetivo" [ZHA02] Si una tarea del usuario define una secuencia de selecciones o entradas estandarizadas (con muchas opciones diferentes dentro de la secuencia), la primera selección (por ejemplo, selección de ratón) debe estar físicamente cerca de la siguiente selección Por ejemplo, considere la interfaz de la página de inicio de una WebApp en un sitio de comercio electrónico que vende aparatos electrodomésticos

Cada opción del usuario implica un conjunto de elecciones o acciones de seguimiento del usuario. Por ejemplo, una opción "comprar un producto" requiere que el usuario ingrese una categoría de producto seguida por el nombre de éste. La categoría del producto (por ejemplo, equipo de audio, televisores, reproductores de DVD) aparece como un menú desplegable tan pronto como se selecciona "comprar un producto". En consecuencia, la siguiente elección es inmediatamente obvia (está cerca) y el tiempo para adquirir es despreciable. Si, por otra parte, la elección aparece en un menú ubicado en el otro lado de la pantalla, el tiempo para que el usuario lo adquiera (y luego realice la elección) será demasiado.

Referencia Web

Una búsqueda en la Web descubrirá muchas librerías disponibles, por ejemplo, paquetes, interfaces y clases Java API en **java.sun.com** o COM, DCOM y clases tipo en **msdn.Microsoft.com.**

Objetos de interfaz humana: Se ha desarrollado una gran librería de objetos de interfaz humana reutilizables para WebApps. Úselas. Es posible adquirir, de varias librerías de objetos, cualquier objeto de interfaz que pueda ser "visto, escuchado, tocado o de algún otro modo percibido" [TOG01] por un usuario final.

Reducción de latencia: Más que obligar al usuario a esperar el fin de alguna operación interna (por ejemplo, descargar una imagen gráfica compleja), la WebApp debe usar la multitarea en una forma que permita al usuario proceder con el trabajo como si la operación hubiese sido completada. Además de reducir la latencia, las demoras deben reconocerse de modo que el usuario comprenda lo que está ocurriendo. Esto incluye 1) proporcionar retroalimentación de audio (por ejemplo, un "clic" o campanazo cuando una selección no genera una acción inmediata de la WebApp; 2) desplegar un reloj animado o barra de progreso para indicar que el procesamiento está en marcha; 3) ofrecer algún entretenimiento (por ejemplo, una animación o presentación de texto) mientras ocurra un procesamiento largo.

> "El mejor viaje es el que tiene el menor número de pasos. Acorta la distancia entre el usuario y su meta."
>
> Anónimo

Facilidad de aprendizaje: La interfaz de una WebApp se debe diseñar para minimizar el tiempo de aprendizaje y, una vez aprendido, reducir el reaprendizaje requerido cuando se vuelve a visitar la WebApp. En general, la interfaz debe acentuar un diseño simple e intuitivo que organice el contenido y la funcionalidad en categorías obvias para el usuario.

Metáforas: una interfaz que utilice una metáfora de interacción es más fácil de aprender y de usar, en tanto la metáfora sea apropiada para la aplicación y el usuario Una metáfora debe llamar imágenes y conceptos de la experiencia del usuario, pero no necesita ser una reproducción exacta de una experiencia del mundo real. Por ejemplo, un sitio de comercio electrónico que implementa el pago de cuentas automatizado para una institución financiera usa una metáfora de lista de verificación (no de manera sorprendente) para asistir al usuario en la especificación y la calendarización de los pagos de cuentas. Sin embargo, cuando un usuario "escribe" un cheque, no necesita ingresar el nombre completo del pagador sino que puede elegir de una lista de pagadores o hacer que el sistema seleccione con base en las primeras letras escritas. La metáfora permanece intacta, pero el usuario obtiene asistencia de la WebApp.

Mantener la integridad del producto de trabajo. Un producto de trabajo (por ejemplo, una forma completada por el usuario, una lista especificada por el usuario) debe guardarse de manera automática de modo que no se perderá si ocurriese un error. Todo mundo ha experimentado la frustración asociada con el hecho de completar un gran formulario WebApp sólo para que el contenido se pierda debido a un error (que comete el usuario, la WebApp o la transmisión de cliente a servidor). Para evitar esto la WebApp se debe diseñar para autoguardar todos los datos especificados por el usuario.

...s son una ... porque ... ingeniería ... Sólo ... que ... que se eli... conocido ... usuarios

Legibilidad: toda la información presentada a través de la interfaz debe ser legible para jóvenes y viejos. El diseñador de la interfaz debe enfatizar los estilos de letra legible, tamaños de fuente y opciones de fondo de color que mejoren el contraste.

Estado de rastro: Cuando sea adecuado, el estado de la interacción del usuario debe rastrearse y almacenarse de modo que un usuario pueda salir y regresar más tarde al lugar de donde salió. En general, las *cookies* se pueden diseñar para almacenar información de estado. Sin embargo, las *cookies* son una tecnología controvertida, y otras soluciones de diseño pueden ser más aceptables para algunos usuarios.

Navegación visible: una interfaz de WebApp bien diseñada proporciona "la ilusión de que los usuarios están en el mismo lugar, y que se les lleva el trabajo hasta sus lugares" [TOG01]. Cuando se usa este enfoque la navegación no es preocupación del usuario. En lugar de eso, el usuario recupera objetos de contenido y selecciona funciones que se despliegan y ejecutan por medio de la interfaz.

HOGARSEGURO

Repaso del diseño de la interfaz

El escenario: Oficina de Doug Mi...

Los actores: Doug Miller (gerente del grupo de ingeniería del software de *HogarSeguro*) y Vinod Raman, miembro del equipo de ingeniería del software del producto *HogarSeguro*.

La conversación:

Doug: Vinod, ¿el equipo y tú tuvieron oportunidad de revisar el prototipo de la interfaz de comercio electrónico de HogarSeguroInc.com?

Vinod: Sí ... todos lo experimentamos desde un punto

de vista técnico y yo tengo un montón de notas. Ayer se las envié por correo electrónico a Sharon [gerente del equipo de ingeniería Web de la empresa subcontratista para el sitio Web de comercio electrónico de *HogarSeguro*].

Doug: Sharon y tú se pueden reunir y discutir los pequeños detalles... dame un resumen de los conflictos importantes.

Vinod: En general, hicieron un buen trabajo, nada de labor profunda, pero es una típica interfaz de comercio electrónico, estética decente, plantilla razonable. Han considerado todas las funciones importantes.

Doug (sonríe tristemente): ¿Pero?

Vinod: Bueno, existen algunas cositas.

Doug: ¿Como cuáles?

Vinod (muestra a Doug una secuencia de bosquejos para el prototipo de la interfaz): Aquí está el menú de funciones principales que se despliega en la página de inicio:

Aprenda acerca de *HogarSeguro*
Describa su casa
Obtenga recomendaciones de componentes de *HogarSeguro*
Compre un sistema *HogarSeguro*
Obtenga soporte técnico

El problema no es con estas funciones, todas están bien, pero el nivel de abstracción no es el correcto.

Doug: Todas son funciones principales, ¿no es así?

Vinod: Lo son, pero este es el punto. Tú puedes comprar un sistema al ingresar una lista de componentes en realidad no necesitas describir la casa, si no quieres. Yo sugerí sólo cuatro opciones de menú en la página de inicio:

Aprenda acerca de *HogarSeguro*
Especifique el sistema *HogarSeguro* que necesita
Compre un sistema *HogarSeguro*
Obtenga soporte técnico

Cuando selecciones **especifique el sistema *HogarSeguro* que necesita**, entonces tendrás las siguientes opciones:

Seleccione componentes *HogarSeguro*
Obtenga recomendaciones de componentes de *HogarSeguro*

Si eres un usuario avanzado, seleccionarás componentes de un conjunto de menús desplegables categorizados para sensores, cámaras, paneles de control, etcétera. Si necesitas ayuda, pedirás una recomendación y ésta requerirá que describas tu casa. Creo que es un poco más lógico.

Doug: Estoy de acuerdo. ¿Ya hablaste con Sharon acerca de esto?

Vinod: No, primero quiero discutir esto con los de mercadotecnia; luego le hablaré por teléfono.

Nielsen y Wagner [NIE96] sugieren unas cuantas directrices pragmáticas en el diseño de interfases (basados en su rediseño de una gran WebApp) que proporcionan un buen complemento a los principios sugeridos párrafos atrás en esta sección:

- La rapidez de lectura en un monitor de computadora es aproximadamente 25 por ciento más lenta respecto de la lectura en impresos. En consecuencia, no fuerce al usuario a leer voluminosas cantidades de texto, en particular cuando se explica la operación de la WebApp o se ofrece ayuda en la navegación.
- Evite los signos de "en construcción", crean expectativas y provocan un vínculo innecesario que es seguro para la decepción.
- Los usuarios prefieren no desplazarse. La información importante debe estar dentro de las dimensiones de una ventana típica de navegador.
- Los menús de navegación y los encabezados deben estar diseñados de manera consistente y deben estar disponibles en todas las páginas que estén dispo-

nibles para el usuario. El diseño no debe descansar en las funciones del navegador para asistir en la navegación.

- La estética nunca debe sustituir la funcionalidad. Por ejemplo, un simple botón puede ser una mejor opción de navegación que una imagen o un icono estéticamente placenteros pero vagos, cuya intención no es clara.
- Las opciones de navegación deben ser obvias, incluso para el usuario casual. El usuario no debe tener que buscar en la pantalla para determinar cómo vincularse con otro contenido o servicio.

Una interfaz bien diseñada mejora la percepción del usuario del contenido o servicios que proporciona el sitio. No necesariamente tiene que ser ostentosa, sino que siempre debe estar bien estructurada y ergonómicamente saludable.

> "La gente tiene muy poca paciencia con los sitios WWW pobremente diseñados."
>
> **Jakob Nielsen y Annette Wagner**

19.3.2 Mecanismos de control de la interfaz

Los objetivos de la interfaz de una WebApp son 1) establecer una ventana consistente con el contenido y la funcionalidad que proporciona, 2) guiar al usuario a través de una serie de interacciones con la WebApp, y 3) organizar las opciones de navegación y el contenido disponible para el usuario. Lograr una interfaz consistente requiere que el diseñador use primero el diseño estético (sección 19.4) con el fin de establecer una "apariencia" coherente para la interfaz. Esto abarca muchas características, pero debe subrayar la plantilla y la forma de los mecanismos de navegación. Para guiar la interacción del usuario, el diseñador de la interfaz puede emplear una metáfora apropiada[7] que permita al usuario adquirir una comprensión intuitiva de la interfaz. Las opciones de navegación las implementa el diseñador seleccionando de entre varios mecanismos de interacción:

¿Qué mecanismos de interacción están disponibles para los diseñadores de WebApps?

- *Menús de navegación:* menús clave (organizados vertical u horizontalmente) que mencionan contenido o funcionalidad clave. Dichos menús se pueden implementar de modo que el usuario pueda elegir de una jerarquía de subtemas que se despliegan cuando se selecciona la opción de menú primario.
- *Iconos gráficos:* botón, interruptores e imágenes gráficas similares que permiten al usuario seleccionar alguna propiedad o especificar una decisión.
- *Imágenes gráficas:* alguna representación gráfica que el usuario pueda seleccionar y que implemente un vínculo hacia un objeto de contenido o funcionalidad de la WebApp.

7 En este contexto, una *metáfora* es una representación (extraída de la experiencia del mundo real del usuario) que puede modelarse dentro del contexto de la interfaz. Un ejemplo simple puede ser un interruptor deslizable con que se controla el volumen auditivo de un archivo .mp3.

Es importante anotar que uno o más de dichos mecanismos de control debe proporcionarse en cada nivel de la jerarquía de contenido

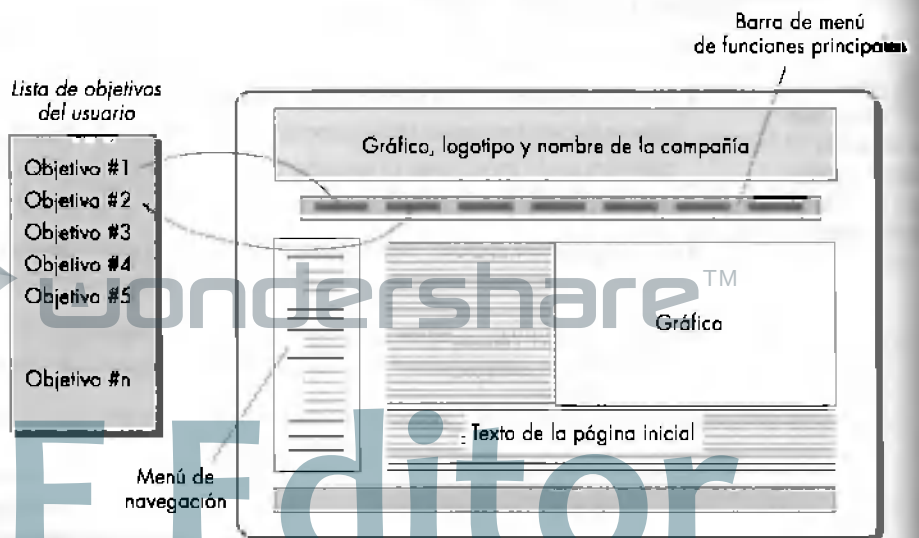
19.3.3 Flujo de trabajo en el diseño de la interfaz

Aunque un análisis detallado del diseño de la interfaz para WebApps es mejor dejarlo a libros de texto que se dedican a la materia (por ejemplo, [GAL02], [RAS00], [NIE00]), vale la pena echar un vistazo a las tareas de diseño clave. En el capítulo 12 se señaló que el diseño de la interfaz del usuario comienza con la identificación de éste, la tarea y los requisitos ambientales. Una vez que se han identificado las tareas del usuario, se crean y analizan sus escenarios (casos de uso) para definir un conjunto de objetos y acciones de interfaz. Este trabajo se representa como parte del modelo de análisis de la WebApp tratado en el capítulo 18.

Las siguientes tareas representan un flujo de trabajo rudimentario para el diseño de la interfaz WebApp:

1. **Revisar la información contenida en el modelo de análisis y refinarla conforme se requiera.**
2. **Desarrollar un bosquejo aproximado de la plantilla de la interfaz de la WebApp**. Como parte de la actividad de modelado del análisis se pudo haber desarrollado un prototipo de la interfaz (que incluya la plantilla). Si ya existe la plantilla, debe revisarse y refinarse conforme se requiera. Si no se ha desarrollado la plantilla de la interfaz, el equipo de ingeniería Web debe trabajar con los participantes para desarrollarla en este momento. En la figura 19.3 se muestra una primera versión de un bosquejo de plantilla

FIGURA 19.3 Correlación de los objetivos del usuario en las acciones de la interfaz.



3. **Correlacionar los objetivos del usuario con acciones específicas de la interfaz.** Para la gran mayoría de las WebApps, el usuario tendrá un conjunto relativamente primario de objetivos primarios (usualmente entre cuatro y siete) Éstos deben correlacionarse con acciones específicas de la interfaz, como se muestra en la figura 19.3.
4. **Definir un conjunto de tareas de usuario que estén asociadas con cada acción.** Cada acción de la interfaz (por ejemplo, "comprar un producto") está asociada con un conjunto de tareas de usuario. Dichas tareas se identificaron durante el modelado de análisis Durante el diseño deben correlacionarse interacciones específicas que abarquen asuntos de navegación, objetos de contenido y funciones WebApp.
5. **Elaborar bosquejos con imágenes de la pantalla para cada acción de la interfaz.** Conforme se considera cada acción, se debe crear una secuencia de imágenes en bosquejo (imágenes de pantallas) para esbozar cómo responde la interfaz a la interacción del usuario. Se deben identificar los objetos de contenido (incluso si todavía no se diseñan ni desarrollan), se debe mostrar la funcionalidad de la WebApp y se deben indicar los vínculos de navegación.
6. **Refinar la plantilla de la interfaz y los bosquejos con el uso de entradas desde el diseño estético.** La plantilla aproximada y los bosquejos los completan los ingenieros Web, pero la apariencia y la percepción estética para un gran sitio comercial con frecuencia los desarrolla un artista, en lugar de profesionales técnicos.
7. **Identificar los objetos de la interfaz del usuario que se requieran para implementarla.** Esta tarea puede requerir una búsqueda en una librería de objetos existente para encontrar aquellos objetos reutilizables (clases) apropiados para la interfaz de la WebApp. Además, en este momento se especifican cualesquiera clases de personalización.
8. **Desarrollar una representación de procedimiento de la interacción del usuario con la interfaz.** Esta labor opcional usa diagramas de secuencia UML o diagramas de actividad (estudiados en el capítulo 18) para esbozar el flujo de actividades (y decisiones) que ocurren conforme el usuario interactúa con la WebApp.
9. **Desarrollar una representación del comportamiento de la interfaz.** Esta tarea opcional utiliza diagramas de estado UML (estudiados en el capítulo 18) para representar las transiciones de estado y los eventos que las causan Se definen los mecanismos de control (es decir, los objetos y acciones disponibles con que el usuario altera el estado de una WebApp).
10. **Describir la plantilla de la interfaz para cada estado.** Con el uso de la información de diseño desarrollada en las tareas 2 y 5, se asocia una plantilla específica o imagen de pantalla con cada estado de la WebApp descrito en la tarea 9.

11. **Refinar y revisar el modelo de diseño de la interfaz.** La revisión de la interfaz se debe enfocar en la facilidad de uso (capítulo 12).

Es importante notar que el conjunto de tareas finales que haya elegido un equipo de ingeniería Web se debe adaptar a los requisitos especiales de la aplicación que se va a construir.

19.4 Diseño estético

CONSEJO

No todo ingeniero Web (o ingeniero de software) tiene talento artístico (estético). Si se está en esta categoría, contrátese un diseñador gráfico experimentado para el trabajo de diseño estético.

El diseño estético, también llamado *diseño gráfico*, es un esfuerzo artístico que complementa los aspectos técnicos de la ingeniería Web. Sin él, una WebApp puede ser funcional, pero sin atractivo. Con él, una WebApp lleva a sus usuarios a un mundo que los incluye en un ámbito tanto emocional como intelectual.

Pero, ¿qué es estética? Existe un viejo dicho: "la belleza existe en los ojos de quien la ve". Esto es particularmente apropiado cuando se considera el diseño estético para las WebApps. Para realizar un diseño estético efectivo, de nuevo se regresa a la jerarquía de usuarios desarrollada como parte del modelo de análisis (capítulo 18) y se pregunta quiénes son los usuarios de la WebApp y qué "apariencia" desean.

> "Encontramos que la gente rápidamente evalúa un sitio sólo por su diseño visual."
>
> **Directrices Stanford para la credibilidad en la Web**

19.4.1 Cuestiones de la plantilla

Toda página Web tiene una cantidad limitada de "bien inmueble" que puede usarse para dar soporte a la estética no funcional, características de navegación, contenido de información y funcionalidad dirigida al usuario. El "desarrollo" de este bien inmueble se planea durante el diseño estético.

Al igual que las cuestiones estéticas, no existen reglas absolutas cuando se diseña una plantilla de pantalla. Sin embargo, vale la pena considerar algunos lineamientos generales de plantilla:

No temerle al espacio vacío. No es aconsejable rellenar cada centímetro cuadrado de una página Web con información. El amontonamiento resultante dificulta que el usuario identifique la información o características necesarias y crea un caos visual desagradable.

Resaltar el contenido. Después de todo, ésta es la razón por la cual el usuario está aquí. Nielsen [NIE00] sugiere que la típica página Web debe ser 80 por ciento contenido con el resto del bien inmueble dedicado a navegación y otras características.

Organizar los elementos de plantilla de arriba a la izquierda hacia abajo a la derecha. La gran mayoría de los usuarios explorarán una página Web en gran parte de la misma forma en que exploran las páginas de un libro: de arriba a la izquierda hacia aba-

jo a la derecha.[8] Si los elementos de plantilla tienen prioridades específicas, los elementos de mayor prioridad deben colocarse en la porción superior izquierda de la página bien inmueble.

Agrupar navegación, contenido y función geográficamente dentro de la página. Los humanos buscan patrones virtualmente en todas las cosas. Si no existen patrones discernibles dentro de una página Web, es probable que aumente la frustración del usuario (debido a la búsqueda innecesaria de la información requerida).

No extender el bien inmueble con la barra de desplazamiento Aunque con frecuencia el desplazamiento es necesario, la mayoría de los estudios indican que los usuarios preferirían no desplazarse Es mejor reducir el contenido de la página o presentar el contenido necesario en varias páginas.

Considerar la resolución y el tamaño de la ventana de navegador cuando diseñe plantillas. En vez de definir tamaños fijos dentro de una plantilla, el diseño debe especificar todos los artículos de la plantilla como un porcentaje del espacio disponible [NIE00]

19.4.2 Cuestiones de diseño gráfico

El *diseño gráfico* considera cada aspecto de la presentación y percepción de una WebApp El proceso de diseño gráfico comienza con la plantilla (sección 19.4.1) y procede hacia la consideración de esquemas de color globales, tipos de fuentes, tamaños y estilos, el uso de medios audiovisuales complementarios (por ejemplo, audio, video, animación) y todos los demás elementos estéticos de una aplicación. El lector interesado puede obtener sugerencias y directrices de diseño en muchos sitios Web que se dedican al tema (por ejemplo, www.graphic-design.com, www grantasticdesigns.com, www.wpdfd com) o de una o más fuentes impresas (por ejemplo, [BAC01], [CLO01] o [HEI02]).

INFORMACIÓN

Sitios Web bien diseñados

En ocasiones, la mejor forma de comprender el buen diseño de las WebApps es observar unos ejemplos. En su artículo "The Top Twenty Web De- Tips" (Las mejores 20 sugerencias para el diseño Marcelle Toor (http://www.graphic-design.com/ ure/tips html) sugiere los siguientes sitios Web ejemplos de buen diseño gráfico:

prima com: firma de diseño encabezado por Primo Angeli

workbook.com: este sitio sirve como aparador para el trabajo de ilustradores y diseñadores.

www.pbs.org/riverofsong: serie de televisión para la radio y la televisión públicas acerca de la música estadounidense

www.RKDINC.com: firma de diseño con un portafolios en línea y buenas sugerencias de diseño.

www commarts.com/career/index.html: revista *Communication Arts*, una publicación periódica para diseñadoras gráficos. Una buena fuente para otros sitios bien diseñados

www.bldnyc.com: firma de diseño encabezado por Beth Toudreau.

8 Existen excepciones basadas en la cultura y el idioma, pero esta regla se aplica a la mayoría de los usuarios

19.5 DISEÑO DEL CONTENIDO

El *diseño del contenido* se enfoca en dos asuntos de diseño diferentes, cada uno de los cuales se aborda individuos con distintos conjuntos de habilidades. El diseño del contenido desarrolla una representación de diseño para los objetos de contenido y representa los mecanismos que se requieren para que establezcan sus relaciones uno con otro. Esta actividad de diseño la dirigen los ingenieros Web.

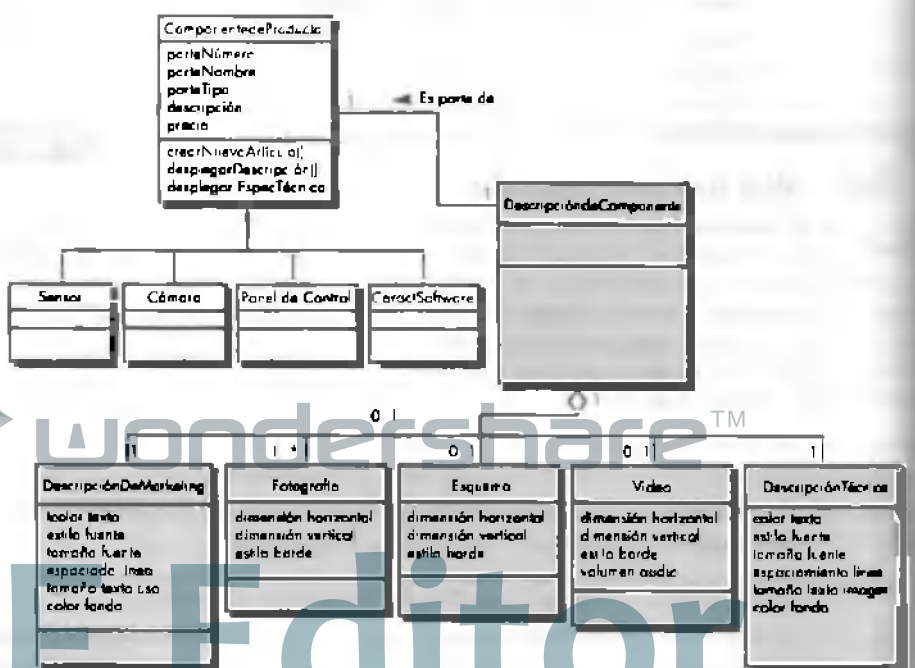
Además, el diseño de contenido se ocupa de la representación de la información dentro de un objeto de contenido específico actividad de diseño que dirigen los publicistas, los diseñadores gráficos y otros que generan el contenido de una WebApp.

> "Los buenos diseñadores pueden crear normalidad a partir del caos; pueden comunicar las ideas con claridad por medio de la organización y el manejo de las palabras y los dibujos."
>
> **Jeffery Veen**

19.5.1 Objetos de contenido

La relación entre objetos de contenido, definida como parte del modelo de análisis WebApp (por ejemplo, figura 18.3), y los objetos de diseño que representan contenido es análoga a la relación entre las clases de análisis y los componentes de diseño descritos en el capítulo 11. En el contexto de la ingeniería Web un *objeto de contenido* está alineado de manera más cercana con un objeto de dato para software con-

FIGURA 19.4
Representación del diseño de los objetos de contenido.



vencional. Un objeto de contenido tiene atributos que incluyen información específica de contenido (normalmente definida durante el modelado de análisis WebApp) y atributos específicos de implementación que se especifican como parte del diseño.

Como ejemplo, considérese la clase de análisis que se desarrolló para el sistema de comercio electrónico *HogarSeguro* llamado **ComponentedeProducto** que se desarrolló en el capítulo 18 y se representa como aparece en la figura 19.4. En el capítulo 18 se mencionó un atributo **descripción** que aquí se representa como una clase de diseño llamada **DescripcióndeComponente,** compuesta de cinco objetos de contenido· **DescripciónDeMarketing, Fotografía, DescripciónTécnica, Esquema** y **Video**, que se muestran como objetos sombreados en la figura La información que contiene el objeto de contenido se registra como atributos Por ejemplo, **Fotografía** (una imagen .jpg) tiene los atributos **dimensión horizontal, dimensión vertical** y **estilo de borde**

Mediante una asociación UML y un agregado[9] se pueden representar relaciones entre los objetos de contenido. Por ejemplo, la asociación UML que se muestra en la figura 19.4 indica que se emplea una **DescripcióndeComponente** para cada instancia de la clase **ComponentedeProducto DescripcióndeComponente** está integrado por los cinco objetos de contenido mostrados Sin embargo, la multiplicidad de notación que se muestra indica que **Esquema** y **video** son opcionales (es posible que se presenten cero ocurrencias), se requieren una **DescripciónDeMarketing** y **DescripciónTécnica**, y se aplican una o más instancias de **Fotografía**

19.5.2 Cuestiones del diseño de contenido

Una vez modelados todos los objetos de contenido, la información que cada objeto entregará debe crearse y luego formatearse para satisfacer mejor las necesidades del cliente. La creación del contenido es el trabajo de los especialistas que diseñan el objeto de contenido al proporcionar un esbozo de la información que se entregará y una indicación de los tipos de los objetos de contenido genéricos (por ejemplo, texto descriptivo, imágenes gráficas, fotografías) mediante los cuales se entregará la información También se puede aplicar el diseño estético (sección 19.4) para representar la apariencia y percepción adecuados para el contenido.

CONSEJO

s tienden a
el desplaza-
vertical más
e que el
miento hori-
. Evite los
de página

Conforme se diseñan, los objetos de contenido se "despedazan" [POW00] para formar páginas de la WebApp El número de objetos de contenido que se incorporan en una sola página es en función de las necesidades del usuario, de las restricciones impuestas por la rapidez de descarga de las conexiones de Internet y debido a las restricciones que impone la cantidad de desplazamiento que el usuario tolerará.

19.6 DISEÑO ARQUITECTÓNICO

El *diseño arquitectónico* está enlazado con las metas establecidas para la WebApp, el contenido que se presentará, los usuarios que la visitarán y la filosofía de navega-

9 Ambas representaciones se discuten en el capítulo 8

ción que se establezca. El diseñador arquitectónico debe identificar la arquitectura de contenido y la arquitectura de la WebApp. La *arquitectura de contenido*[10] se centra en la forma en la que los objetos de contenido (u objetos compuestos como las páginas Web) se estructuran para su presentación y navegación. La *arquitectura de WebApp* aborda la forma en la que la aplicación se estructura para gestionar la interacción del usuario, manejar las tareas de procesamiento internas, efectuar la navegación y presentar el contenido.

> "[L]a estructura arquitectónica de un sitio bien diseñado no siempre es aparente para el usuario... ni lo debe ser."
> **Thomas Powell**

En la mayoría de los casos, el diseño arquitectónico se dirige en paralelo con el diseño de la interfaz, el estético y el de contenido. Puesto que la arquitectura WebApp puede tener una fuerte influencia sobre la navegación, las decisiones tomadas durante esta actividad de diseño influirán en el trabajo dirigido durante el diseño de navegación

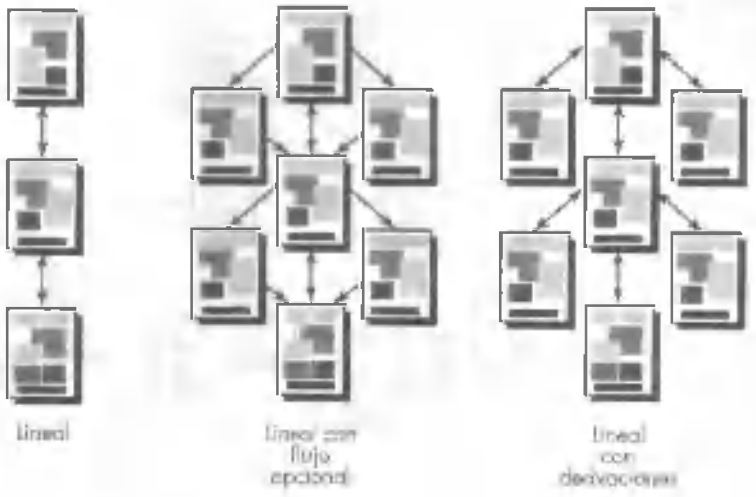
19.6.1 Arquitectura de contenido

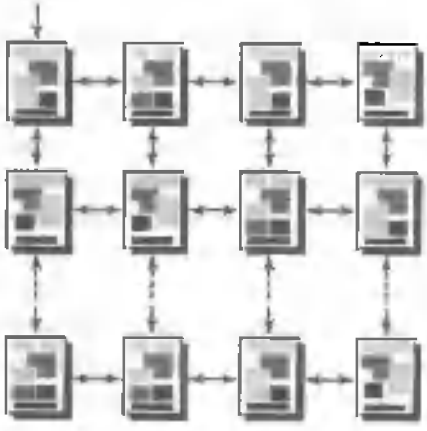
El diseño de la *arquitectura de contenido* se centra en la definición de la estructura hipermedia global de la WebApp. El diseño se puede elegir de cuatro diferentes estructuras de contenido [POW00]:

Las *estructuras lineales* (figura 19.5) se encuentran cuando es común una secuencia predecible de interacciones (con alguna variación o desviación). Un ejemplo clásico puede ser una presentación tutorial en la que las páginas de información junto con gráficos relacionados, videos cortos o audio se presentan sólo después de que se ha presentado información de prerrequisitos. La secuencia de la presentación de contenido está predefinida y, por lo general, es lineal. Otro ejemplo puede ser una secuencia de entradas para comprar un producto, en la cual se debe detallar información específica en un orden específico. En tales casos, son apropiadas las estructuras mostradas en la figura 19.5. Conforme el contenido y el procesamiento se vuelven más complejos, el flujo meramente lineal mostrado a la izquierda de la figura da paso a estructuras lineales más complejas en las que se puede llamar contenido alternativo u ocurre una desviación para adquirir contenido complementario (estructura mostrada a la derecha de la figura 19.5).

Las *estructuras en retícula* (figura 19.6) son una opción arquitectónica aplicable cuando el contenido de la WebApp está organizado categóricamente en dos (o más) dimensiones. Por ejemplo, considérese una situación en la cual un sitio de comercio electrónico vende palos de golf. La dimensión horizontal de la retícula representa el tipo de palo que se vende (por ejemplo, madera, hierro, *wedges*, *putters*). La dimen-

10 El término *arquitectura de información* también se utiliza para sugerir estructuras que conducen a una mejor organización, etiquetado, navegación y búsqueda de objetos de contenido

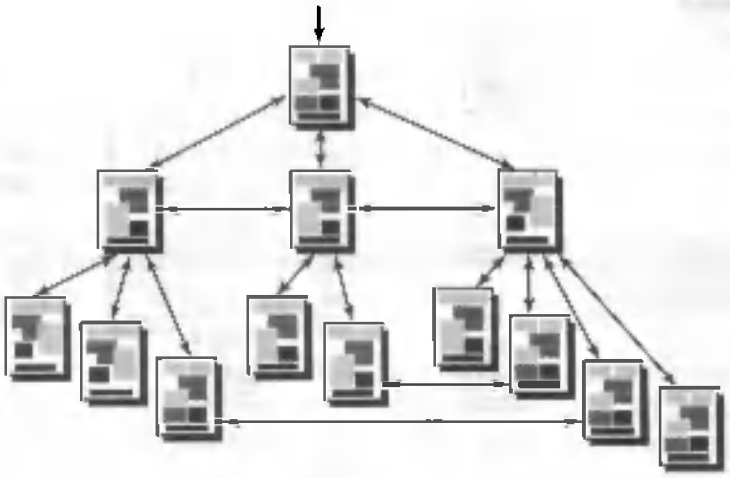




sión vertical representa las ofertas de varios fabricantes de palos de golf. En consecuencia, un usuario puede navegar la retícula horizontalmente para encontrar la columna *putters* y luego verticalmente para examinar las ofertas de aquellos fabricantes que venden *putters*. Esta arquitectura de WebApp sólo es útil cuando se encuentra contenido altamente regular [POW00].

Las *estructuras jerárquicas* (figura 19.7) son indudablemente las arquitecturas WebApp más comunes. A diferencia de las jerarquías de software factorizadas —que se estudiaron en el capítulo 10—, que alinean el flujo de control sólo a lo largo de las ramas verticales de la jerarquía, una estructura jerárquica WebApp se puede diseñar en una forma que permita (vía ramificaciones de hipertexto) el flujo de control horizontalmente, a través de las ramas verticales de la estructura. Por lo tanto, el contenido presentado en la rama de la extrema izquierda de la jerarquía puede tener vínculos

FIGURA 19.7
Estructura jerárquica.



de hipertexto que conduzcan a contenido que existe en la rama de en medio o a la derecha de la estructura. Sin embargo, se debe señalar que, aunque tales ramificaciones permiten la navegación rápida a través del contenido de la WebApp, pueden conducir a confusión en la parte del usuario.

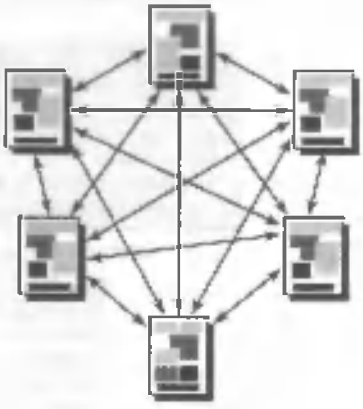
Una *estructura en red* o *"Web pura"* (figura 19.8) es similar en muchos sentidos a la arquitectura que evoluciona para los sistemas orientados a objetos. Los componentes arquitectónicos (en este caso, páginas Web) están diseñados de modo que pueden pasar el control (vía vínculos de hipertexto) virtualmente a cualquier otro componente en el sistema. Este enfoque permite una considerable flexibilidad en la navegación, pero al mismo tiempo puede ser confusa para el usuario.

Las estructuras arquitectónicas comentadas en los párrafos precedentes se puede combinar para formar *estructuras compuestas*. La arquitectura global de una WebApp puede ser jerárquica, pero parte de la estructura puede mostrar características lineales, mientras que otra parte puede estar en red. La meta para el diseñador arquitectónico es emparejar la estructura WebApp con el contenido que se presentará y el procesamiento que se llevará a cabo.

19.6.2 Arquitectura de WebApp

La *arquitectura de WebApp* describe una infraestructura que permite a un sistema o aplicación basados en Web lograr sus objetivos de negocios. Jacyntho y sus colegas [JAC02] describen las características básicas de esta infraestructura en la forma siguiente:

> Las aplicaciones deben construirse con el uso de capas en las que se tomen en cuenta las diferentes preocupaciones; en particular, los datos de la aplicación se deben separar de los contenidos de la página (nodos de navegación), y dichos contenidos, a su vez, deben estar claramente separados de la apariencia y la percepción de la interfaz (páginas)



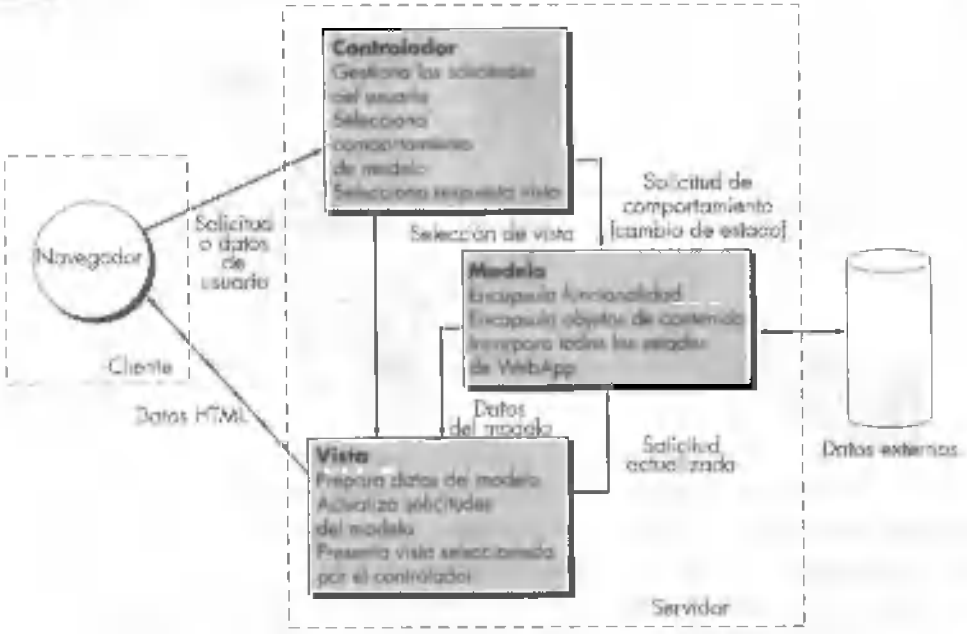
Los autores sugieren una arquitectura de diseño en tres capas que desacople la interfaz de la navegación y del comportamiento de la aplicación, y argumentan que mantener la separación de la interfaz, aplicación y navegación simplifica la implementación y mejora la reutilización.

La arquitectura de *modelo-vista-controlador* (MVC) [KRA88][11] es uno de varios modelos de infraestructura WebApp sugeridos para desacoplar la interfaz del usuario de la funcionalidad y el contenido de información de la WebApp. El *modelo* (a veces llamado "objeto modelo") contiene todo el contenido específico de la aplicación y la lógica de procesamiento, e incluye todos los objetos de contenido, el acceso a fuentes de datos/información externas y toda la funcionalidad de procesamiento que son específicos de la aplicación. La *vista* contiene todas las funciones específicas de la interfaz y habilita la presentación del contenido y la lógica de procesamiento, e incluye todos los objetos de contenido, acceso a fuentes de datos/información externas y a toda la funcionalidad de procesamiento requerida por el usuario final. El *controlador* gestiona el acceso al *modelo* y a la *vista* y coordina el flujo de datos entre ellos. En una WebApp, "la vista la actualiza el controlador con datos provenientes del modelo con base en la entrada del usuario" [WMT02]. En la figura 19.9 se muestra una representación esquemática de la arquitectura MVC.

En referencia a la figura, las solicitudes o datos del usuario se manejan mediante el controlador. Éste también selecciona el objeto vista que es aplicable con base en la solicitud del usuario. Una vez que se determina el tipo de solicitud, se transmite una solicitud de comportamiento al modelo, que implementa la funcionalidad o recupera el contenido requerido para acomodar la solicitud. El objeto modelo puede tener acceso a datos almacenados en una base de datos corporativa, como parte de

11 Se debe destacar que MVC es en realidad un patrón de diseño arquitectónico desarrollado por el ambiente Smalltalk (véase http://www.cetus-links.org/oo_smalltalk.html) y se puede usar para cualquier aplicación interactiva.

FIGURA 19.7 La arquitectura MVC (adaptada de [JAC02]).



un almacén de datos local o como una colección de archivos independientes datos que desarrolla el modelo debe formatearlos y organizarlos el objeto vista ad cuado y luego transmitirlo del servidor de la aplicación de vuelta al navegador bas do en el cliente para que se despliegue en la máquina de éste.

En muchos casos, la arquitectura de WebApp se define dentro del contexto ambiente de desarrollo en el que la aplicación habrá de implementarse (por ejemp ASPnet, JWAA o J2EE). El lector interesado debe ver [FOW03] para una exposici ulterior acerca de los ambientes de desarrollo modernos y de su papel en el dise de las arquitecturas de aplicaciones Web.

19.7 DISEÑO DE NAVEGACIÓN

Una vez establecida la arquitectura de WebApp y la identificación de los compon tes (páginas, guiones, *applets* y otras funciones de procesamiento), el diseñador be definir las rutas de navegación que habiliten para los usuarios el acceso al c nido y las funciones de la WebApp. Para lograr esto el diseñador debe 1) identi la semántica de navegación para diferentes usuarios del sitio y 2) definir la m ca (sintaxis) que logra la navegación

> "Sólo espera, Gretel, hasta que la luna se eleve, entonces veremos los trozos de pan que he desparramado, ellos nos mostrarán de nuevo el camino a casa."
>
> Tomado de Hansel y

19.7.1 Semántica de navegación

Al igual que muchas actividades de ingeniería Web, el diseño de navegación comienza con una consideración de la jerarquía de usuario y los casos de uso relacionados (capítulo 18) desarrollados para cada categoría de usuario (actor). Cada actor puede usar la WebApp de manera un poco diferente y, por tanto, tener diferentes requisitos de navegación. Además, los casos de uso desarrollados para cada actor definirán un conjunto de clases que abarcan uno o más objetos de contenido o funciones de la WebApp. Conforme cada usuario interactúa con la WebApp, encuentra una serie de *unidades semánticas de navegación* (USN), "un conjunto de estructuras de información y navegación relacionadas que colaboran en el cumplimiento de un subconjunto de requisitos de usuario relacionados" [CAC02].

Gnaho y Larcher [GNA99] describen la USN en la forma siguiente:

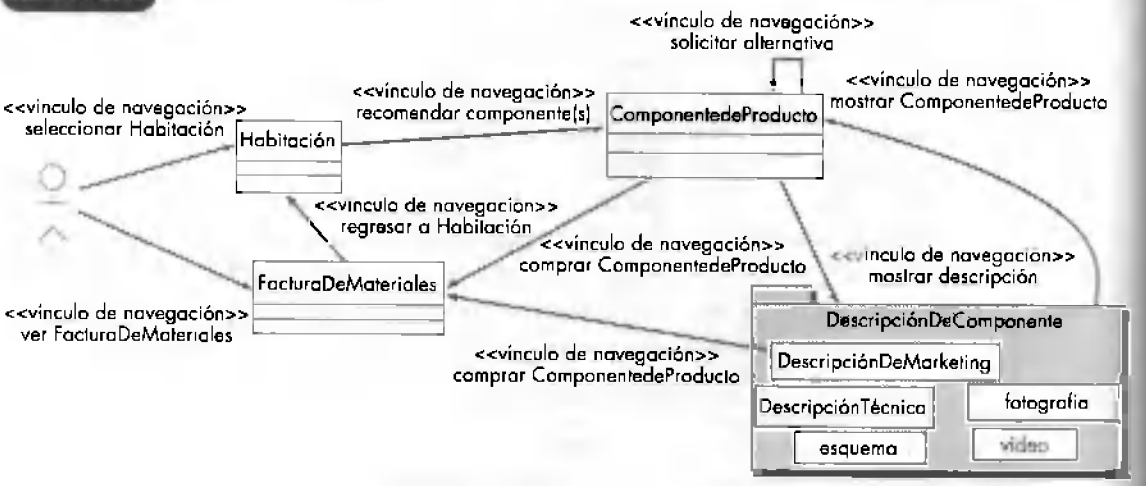
> La estructura de una USN está compuesta de un conjunto de subestructuras de navegación que se llamarán *formas de navegación* (FdN). Una FdN representa la mejor forma o ruta de navegación para los usuarios con ciertos perfiles para lograr su meta o submeta deseada. En consecuencia, el concepto de FdN está asociado con el concepto de Perfil de Usuario.
>
> La estructura de una FdN está integrada con un conjunto de *nodos de navegación* (NN) relevantes conectados por *vínculos de navegación*, que en ocasiones incluyen otras FdN. Esto significa que las FdN pueden, en sí mismas, ser agregadas para formar una FdN de nivel superior, o pueden anidarse en cualquier profundidad.
>
> Para ilustrar el desarrollo de una FdN, considérese el caso de uso *seleccionar componentes HogarSeguro* descrito en la sección 18.1.2 y que se reproduce a continuación:
>
> **Caso de uso:** *seleccionar componentes HogarSeguro*
>
> Entonces la WebApp recomendará componentes de_producto (por ejemplo, paneles de control, sensores, cámaras) y otras características (por ejemplo, funcionalidad basada en PC implementada en software) para cada habitación y la entrada exterior. Si el usuario solicita opciones, la WebApp las proporcionará si existen. El usuario obtendrá información descriptiva y de precios para cada componente de producto. La WebApp creará y mostrará una factura de materiales conforme se seleccionen varios componentes. El usuario también podrá nombrar la factura de materiales y guardarla para referencia futura (véase caso de uso: *guardar configuración*).

PUNTO CLAVE
describe los
de
para cada
uso. En
la USN
como un actor
entre los
de contenido o
de la

Los artículos subrayados en la descripción del caso de uso representan clases y objetos de contenido que serán incorporados en una o más FdN que permitirán a un nuevo cliente realizar el escenario descrito en el caso de uso *seleccionar componentes HogarSeguro*.

La figura 19.10 bosqueja un análisis semántico parcial de la navegación que implica el caso de uso *seleccionar componentes HogarSeguro*. Con la aplicación de la terminología introducida anteriormente, la figura también representa una FdN para la WebApp HogarSeguroInc.com. Se muestran importantes problemas en las clases de dominio junto con objetos de contenido seleccionados (en este caso, el paquete

FIGURA 19.10 Creación de una USN.



de objetos de contenido llamado **DescripciónComp**, un atributo de la clase **ComponentedeProducto**). Dichos artículos son nodos de navegación. Cada una de las flechas representa un vínculo de navegación[12] y está etiquetado con la acción que inicia el uso que causa que el vínculo ocurra.

El diseñador de la WebApp crea una *unidad semántica de navegación* (USN) para cada caso de uso asociado con cada papel de usuario [GNA99]. Por ejemplo, el **cliente nuevo** (figura 18.1) puede tener tres diferentes casos de uso, y todos resultan en acceso a diferente información y funciones de la WebApp. Para cada meta se crea una USN.

Durante las etapas iniciales del diseño de navegación se valora la arquitectura de contenido de la WebApp para determinar una o más FdN para cada caso de uso. Como se anotó anteriormente, una FdN identifica los nodos de navegación (por ejemplo, contenido) y los vínculos que permiten la navegación entre ellos. Entonces las FdN se organizan en USN.

> "El problema de la navegación en el sitio Web es conceptual, técnico, espacial, filosófico y logístico. Consecuentemente las soluciones tienden a pedir combinaciones de arte, ciencia y psicología organizacional improvisadas y complejas."
>
> **Tim Horgan**

19.7.2 Sintaxis de navegación

Conforme el diseño se lleva a cabo se define la mecánica de navegación. Entre muchas posibles opciones están:

- *Vínculo de navegación individual:* vínculos basados en texto, iconos, botones e interruptores, y metáforas gráficas.

12 En ocasiones, a éstos se les conoce como *vínculos semánticos de navegación* (VSN) [CAC02].

- *Barra de navegación horizontal:* lista de las principales categorías de contenido o funcionales en una barra que contiene vínculos adecuados. En general, se mencionan entre cuatro y siete categorías.
- *Columna de navegación vertical:* 1) lista de las principales categorías de contenido o funcionales, o 2) lista de virtualmente todos los principales objetos de contenido dentro de la WebApp. Si se elige la segunda opción, tales columnas de navegación se pueden "expandir" para presentar objetos de contenido como parte de una jerarquía.
- *Pestañas:* una metáfora que no es más que una variación de la barra o columna de navegación, que representa las categorías de contenido o funcionales como marcas que se seleccionan cuando se requiere un vínculo.
- *Mapas de sitio:* proporcionan una tabla de contenido incluyente para la navegación hacia todos los objetos de contenido y funcionalidad contenidos en la WebApp.

Además de elegir los mecanismos de navegación, el diseñador también debe establecer convenciones y auxiliares de navegación adecuados. Por ejemplo, íconos y vínculos gráficos que deben parecer "oprimibles" mediante el biselado de los bordes para que la imagen tenga una apariencia tridimensional. Debe diseñar retroalimentación visual o de audio para ofrecer al usuario un indicador de que ha elegido una opción de navegación. En la navegación basada en texto debe usarse color para indicar los vínculos de navegación y proporcionar un indicador de los vínculos ya recorridos. Éstas son sólo algunas de las docenas de convenciones de diseño que hacen la navegación amigable al usuario.

19.8 DISEÑO AL NIVEL DE COMPONENTES

Las modernas aplicaciones Web entregan funciones de procesamiento cada vez más elaboradas que 1) realizan procesamiento localizado para generar capacidad de contenido y navegación en una forma dinámica; 2) ofrecen capacidades de computación o procesamiento de datos que son adecuadas para el dominio de negocios de la WebApp; 3) proporcionan cuestionamientos y acceso sofisticados a bases de datos, 4) establecen interfases de datos con sistemas corporativos externos. Para lograr estas (y muchas otras) capacidades, el ingeniero Web debe diseñar y construir componentes de programa que sean idénticos en forma a los componentes de software para el software convencional.

En el capítulo 11 se considera con cierto detalle el diseño al nivel de componentes. Los métodos de diseño estudiados en el capítulo 11 se aplican a los componentes WebApp con poca, si acaso, modificación. El ambiente de implementación, los lenguajes de programación y los patrones de reutilización, marcos de trabajo y software pueden variar un poco, pero el enfoque de diseño global permanece igual.

19.9 PATRONES DE DISEÑO HIPERMEDIA

Los patrones de diseño aplicados en la ingeniería Web abarcan dos grandes clases: 1) *patrones de diseño genérico* que son aplicables a todos los tipos de software (por ejemplo, [BUS96] y [GAM95]) y 2) *patrones de diseño hipermedia* que son específicos de las WebApp. En el capítulo 9 se trataron los patrones de diseño genérico. A través de Internet se puede tener acceso a varios catálogos y almacenes de patrones de hipermedia.[13]

> "Cada patrón es una regla de tres partes que expresa una relación entre cierto contexto, un problema y una solución."
> **Christopher Alexander**

Como se apuntó antes en este libro, los patrones de diseño son un enfoque genérico para resolver algún pequeño problema de diseño que se puede adaptar a una variedad mucho más amplia de problemas específicos. En el contexto de los sistemas basados en Web, German y Cowan [GER00] sugieren las siguientes categorías de patrones:

Patrones arquitectónicos. Estos patrones auxilian en el diseño del contenido y la arquitectura de la WebApp. Las secciones 19.6.1 y 19.6.2 presentan patrones arquitectónicos para el contenido y la arquitectura de la WebApp. Además, están disponibles muchos patrones arquitectónicos relacionados (por ejemplo, Java Blueprints en java sun.com/blueprints/) para los ingenieros Web que deben diseñar WebApps en una diversidad de dominios de negocios.

Patrones de construcción de componentes. Estos patrones recomiendan métodos para combinar componentes WebApp (por ejemplo, objetos de contenido, funciones) en componentes compuestos. Cuando se requiere la funcionalidad de procesamiento de datos en una WebApp, son aplicables los patrones de diseño arquitectónico y al nivel de componente que proponen [BUS96], [GAM95] y otros.

Patrones de navegación. Estos patrones auxilian en el diseño de USN, vínculos de navegación y el flujo global de navegación de la WebApp.

Patrones de presentación. Estos patrones auxilian en la presentación del contenido como se presenta al usuario vía la interfaz correspondiente. Los patrones en esta categoría abordan cómo organizar las funciones de control de la interfaz del usuario para una mejor facilidad de uso; cómo mostrar la relación entre una acción de la interfaz y los objetos de contenido que afecta; cómo establecer jerarquías de contenido efectivas; y muchas otras.

Patrones de interacción comportamiento/usuario. Estos patrones auxilian en el diseño de la interacción usuario-máquina. Los patrones en esta categoría abordan

13 Véase la barra lateral al final de esta sección.

cómo la interfaz informa al usuario de las consecuencias de una acción específica; cómo un usuario expande el contenido con base en el contexto de uso y sus deseos; cómo describir mejor el destino que implica un vínculo; cómo informar al usuario acerca del estado de una interacción en marcha y otros.

Las fuentes de información acerca de los patrones de diseño hipermedia se han expandido en forma sustancial en años recientes. Los lectores interesados deben consultar [GAR97], [PER99] y [GER00]

HERRAMIENTAS DE SOFTWARE

Almacenes de patrones de diseño hipermedia

El sitio Web IAWiki (http://iawiki.net/WebsitePatterns) es un espacio de discusión conjunto [illegible] información de los arquitectos y que contiene muchos [illegible] útiles. Entre ellas están vínculos a varias catálogos [illegible]es de patrones hipermedia útiles. Están representa[illegible] cientos de patrones de diseño:

[illegible]cán de patrones de diseño hipermedia
[illegible]'www.designpattern.lu.unisi.ch/
[illegible]tionPatterns de Tom Erickson
[illegible]//www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html
[illegible]es de diseño Web de Martijn vanWelie
[illegible]'www.welie.com/patterns/
Mejora de los sistemas de información Web con patrones de navegación
http://www8.org/w8-papers/5b-hypertext-media/improving/improving.html
Un patrón de lenguaje HTML 2.0
http://www.anamorph.com/docs/patterns/default.html
Terreno común
http://www.mit.edu/~jtidwell/interaction_patterns.html
Patrones para sitios Web personales
http://www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html
Índice de lenguajes patrón
http://www.cs.brown.edu/~rms/InformationStructures/Indexing/Overview.html

19.10 MÉTODO DE DISEÑO HIPERMEDIA ORIENTADO A OBJETOS (MDHOO)

Durante las pasadas décadas se propusieron varios métodos de diseño para aplicaciones Web. A la fecha, ningún método es el dominante. En esta sección se presenta un breve panorama de uno de los métodos de diseño WebApp más ampliamente analizados MDHOO.[14]

El *método de diseño hipermedia orientado a objetos* (MDHOO) lo propusieron originalmente Daniel Schwabe y sus colegas [SCH95, SCH98]. El MDHOO está compuesto de cuatro diferentes actividades de diseño: diseño conceptual, diseño de navegación, diseño abstracto de la interfaz e implementación. En la figura 19.11 se muestra un resumen de estas actividades de diseño, y en las secciones que siguen se discuten brevemente

19.10.1 Diseño conceptual por el MDHOO

El *diseño conceptual* mediante el MDHOO crea una representación de los subsistemas, clases y relaciones que definen el dominio de aplicación para la WebApp Se

14 Koch [KOC99] ha desarrollado una amplia comparación de los diez métodos de diseño hipermedia

FIGURA 19.11 Resumen del método MDHOO (adaptado de [SCH95]).

	 Diseño conceptual	 Diseño de navegación	 Diseño abstracto de la interfaz	 Implementación
Productos de trabajo	Clases, subsistemas, relaciones, atributos	Nodos, vínculos, estructuras de acceso, contextos de navegación, transformaciones de navegación	Objetos abstractos de la interfaz, respuestas a eventos externos, transformaciones	WebApp ejecutable
Mecanismos de diseño	Clasificación, composición, agregación, generalización, especialización	Correlación entre objetos conceptuales y de navegación	Correlación entre objetos de navegación y perceptibles	Recurso proporcionado por ambiente objetivo
Preocupaciones de diseño	Modelado de la semántica del dominio de aplicación	Toma en cuenta el perfil del usuario y la tarea. Resalta los aspectos cognitivos.	Modelado de los objetos perceptibles, implementación de las metáforas elegidas. Descripción de la interfaz para los objetos de navegación	Exactitud, desempeño de la aplicación, integridad

puede usar[15] UML para crear diagramas de clase adecuados, agregados y representaciones de clase compuestas, diagramas de colaboración y otra información que describe el dominio de la aplicación (véase la Parte 2 de este libro para más detalles).

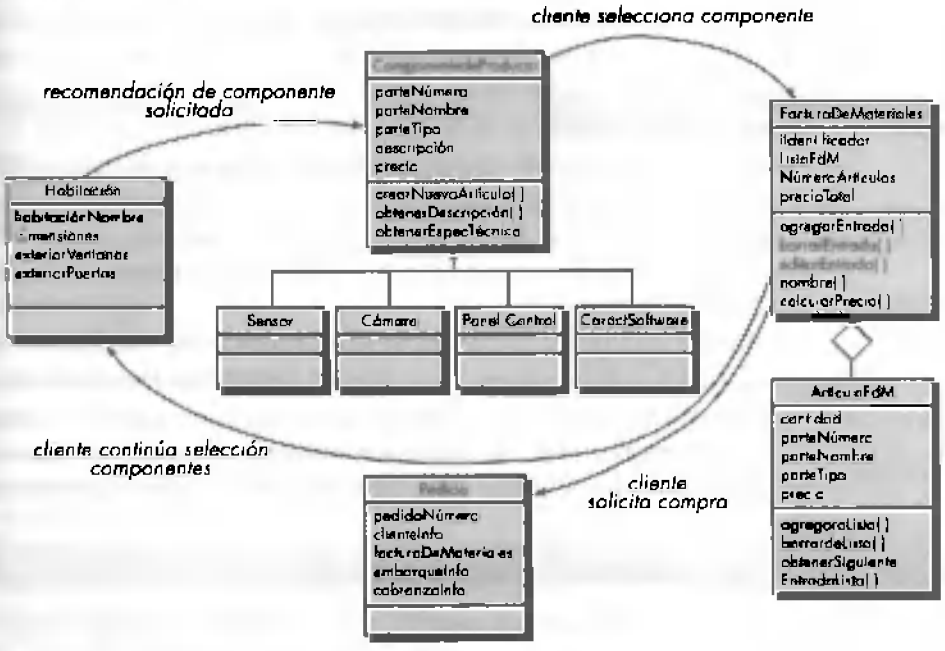
Como un ejemplo simple de diseño conceptual del MDHOO, considérese de nuevo la aplicación de comercio electrónico de HogarSeguroInc.com. En la figura 19.12 se muestra un "esquema conceptual" parcial para HogarSeguroInc.com. Los diagramas de clase, agregados e información relacionada desarrollados como parte del análisis de la WebApp se reutilizan durante el diseño conceptual para representar relaciones entre clases.

19.10.2 Diseño de navegación mediante el MDHOO

El *diseño de navegación* identifica un conjunto de "objetos" que se derivan de las clases definidas en el diseño conceptual. Se define una serie de "clases de navegación" o "nodos" para encapsular dichos objetos. Se puede usar UML para crear casos de uso adecuados, gráficos de estado y diagramas de secuencia, todos ellos auxilian al diseñador a comprender mejor los requisitos de navegación. Además, es posible aplicar los patrones de diseño para el diseño de navegación conforme el diseño se desarrolle. El MDHOO utiliza un conjunto predefinido de clases de navegación: no-

15 El MDHOO no prescribe una notación específica, sin embargo, el uso de UML es común cuando se aplica este método.

19.12 Esquema conceptual parcial para HogarSeguroInc.com.



dos, vínculos, anclas y estructuras de acceso [SCH98]. Las estructuras de acceso son más elaboradas e incluyen mecanismos como un índice de la WebApp, un mapa de sitio o un paseo guiado

Una vez definidas las clases de navegación, el MDHOO "estructura el espacio de navegación mediante el agrupamiento de los objetos de navegación en conjuntos llamados contextos" [SCH98]. Schwabe describe un *contexto* en los términos siguientes:

> Cada definición de contexto incluye, además de los elementos que están incluidos en él, la especificación de su estructura de navegación interna, un punto de entrada, restricciones de acceso en términos de clases de usuario y operaciones, y una estructura de acceso asociada.
>
> Se desarrolla una plantilla de contexto (análoga a las tarjetas CRC estudiadas en el capítulo 8) y se emplea para rastrear los requisitos de navegación de cada categoría de usuario a través de varios contextos definidos en el MDHOO. Al hacer esto surgen rutas específicas de navegación (que se llamaron FdN en la sección 19.7.1).

19.10.3 Diseño abstracto de la interfaz e implementación

La actividad de *diseño abstracto de la interfaz* especifica los objetos de la interfaz que el usuario ve conforme interactúa con la WebApp. Un modelo formal de objetos de la interfaz, llamado *visión abstracta de datos* (VAD) se utiliza para representar la re-

lación entre objetos de la interfaz y objetos de navegación, y las características de comportamiento de los objetos de la interfaz.

El modelo VAD define una "plantilla estática" [SCH98] que representa la metáfora de la interfaz e incluye una representación de los objetos de navegación dentro de la interfaz y la especificación de los objetos de la interfaz (por ejemplo, menús, botones, iconos) que auxilian en la navegación y la interacción. Además, el modelo VAD contiene un componente relacionado con el comportamiento (similar al diagrama de estado UML) que indica cómo los eventos externos "disparan la navegación y qué transformaciones de la interfaz ocurren cuando el usuario interactúa con la aplicación" [SCH01] Una exposición detallada del VAD el lector interesado puede hallarla en [SCH98] y [SCH01].

La actividad *implementación* del MDHOO representa una interacción de diseño que es específica al ambiente en el que operará la WebApp. Las clases, la navegación y la interfaz son caracterizadas en una forma que puede construirse para el ambiente cliente/servidor, sistemas operativos, software de soporte, lenguajes de programación y otras características del entorno relevantes respecto del problema

19.11 MÉTRICAS DE DISEÑO PARA WEBAPPS

Las métricas de diseño se deben caracterizar en una forma que proporcione a los ingenieros Web un indicador de calidad en tiempo real. En esencia, un conjunto útil de medidas y métricas ofrece respuestas cuantitativas a las siguientes preguntas:

- ¿La interfaz del usuario promueve la facilidad de uso?
- ¿La estética de la WebApp es apropiada para el dominio de la aplicación y confortable para el usuario?
- ¿El contenido está diseñado en una forma que proporciona la mayor información con el menor esfuerzo?
- ¿La navegación es eficiente y directa?
- ¿La arquitectura de la WebApp se ha diseñado para acomodar las metas y objetivos especiales de los usuarios de la WebApp, la estructura de contenido y funcionalidad, y el flujo de navegación requerido para usar el sistema de manera efectiva?
- ¿Los componentes están diseñados en una forma que reduce la complejidad de procedimientos y aumenta la exactitud, la confiabilidad y el desempeño?

En la actualidad, cada una de estas preguntas se puede abordar de manera cualitativa,[16] pero todavía no existe un conjunto validado de métricas que ofrezcan respuestas cuantitativas.

16 Véase el capítulo 16 (sección 16.4) y la sección 19.11 para una exposición cualitativa de la calidad de una WebApp

Las métricas para el diseño de WebApps están en desarrollo y pocas se han validado ampliamente. El lector interesado debería consultar [IVO01] y [MEN01] para una muestra de las métricas propuestas para el diseño de WebApps

HERRAMIENTAS DE SOFTWARE

Métricas técnicas para WebApps

Objetivo: Apoyar a los ingenieros Web en el desarrollo de métricas WebApp significativas ofrezcan una visión acerca de la calidad global de aplicación

Mecánica: Las herramientas mecánicas varian

Herramientas representativas[17]

Mechanic Tools, desarrollada por Netmechanic (www.netmechanic.com), es una colección de herramientas que ayudan a mejorar el desempeño de un sitio Web; se enfoca sobre los temas específicos de la implementación.

NIST Web Metrics Testbed, desarrollado por The National Institute of Standards and Technology (zing ncsl nist.gov/WebTools/), abarca la siguiente colección de herramientas útiles que están disponibles para descargar:

Web Static Analyzer Tool (WebSAT): verifica el HTML de la página web contra los lineamientos de facilidad de uso típicos

Web Category Analysis Tool (WebCAT): permite al ingeniero de facilidad de uso construir y dirigir un análisis de categoría Web.

Web Variable Instrumenter Program (WebVIP): instrumenta un sitio Web para capturar un registro de interacción de usuario.

Framework for Logging Usability Data (FLUD): implementa un formateador y analizador gramatical de archivos para representar los registros de interacción de usuario.

VisVIP Tool: produce una visualización tridimensional de las rutas de navegación del usuario a través de un sitio Web

TreeDec: agrega auxiliares de navegación a las páginas de un sitio Web.

19.12 RESUMEN

La calidad de una WebApp —definida en términos de facilidad de uso, funcionalidad, confiabilidad, eficiencia, facilidad de mantenimiento, seguridad, escalabilidad y tiempo en el mercado— se introduce durante el diseño. Para lograr dichos atributos de calidad, un buen diseño WebApp debe posser simplicidad, consistencia, identidad, robustez, navegabilidad y apariencia visual.

El diseño de la interfaz describe la estructura y organización de la interfaz del usuario. Incluye una representación de la plantilla de pantalla, una definición de los modos de interacción y una descripción de los mecanismos de navegación.

El diseño estético, también llamado diseño gráfico, describe la "apariencia y la percepción" de la WebApp e incluye esquemas de color, plantilla geométrica, tamaño de texto, fuente y ubicación, el uso de gráficos y decisiones estéticas relacionadas. Un conjunto de lineamientos de diseño gráfico proporciona la base para un enfoque de diseño.

17 Las herramientas anotadas son una muestra de esta categoría

El diseño de contenido define la plantilla, la estructura y el subrayado de todo el contenido que se presenta como parte de la WebApp; además, establece las relaciones entre objetos de contenido. El diseño de contenido comienza con la representación de los objetos de contenido, sus asociaciones y relaciones. Un conjunto de consideraciones elementales establece las bases para el diseño de navegación.

El diseño de arquitectura identifica la estructura hipermedia global para la WebApp y abarca tanto la arquitectura de contenido como la de WebApp. Los estilos arquitectónicos para el contenido incluyen estructuras lineal, en retícula, jerárquica y en red. La arquitectura de la WebApp describe una infraestructura que permite a un sistema o aplicación basado en Web lograr sus objetivos de negocios.

El diseño de navegación representa el flujo de navegación entre los objetos de contenido y para todas las funciones de la WebApp. La navegación se define al describir un conjunto de unidades semánticas de navegación. Cada unidad está compuesta de formas de navegación y de vínculos y nodos de navegación. Los mecanismos de sintaxis de navegación se aplican para afectar la navegación descrita como parte de la semántica.

El diseño de componentes desarrolla la lógica de procesamiento detallada que se requiere para implementar los componentes funcionales de la WebApp. Las técnicas de diseño descritas en el capítulo 11 se aplican a la ingeniería de componentes WebApp.

Los patrones para el diseño de WebApps abarcan patrones de diseño genérico que se aplican a todos los tipos de software y patrones hipermedia especialmente relevantes para las WebApp. Se han propuesto patrones de diseño arquitectónico, de navegación, de componentes, de presentación y de comportamiento/usuario.

El método de diseño hipermedia orientado a objetos (MDHOO) es uno de varios métodos propuestos para el diseño WebApp. El MDHOO sugiere un proceso de diseño que incluye diseño conceptual, diseño de navegación, diseño abstracto de la interfaz e implementación.

Las métricas de diseño para ingeniería Web están en desarrollo y todavía tienen que validarse por completo. Sin embargo, se han propuesto varias medidas y métricas para abordar cada una de las actividades de diseño reanalizadas en este capítulo.

REFERENCIAS

[AME96] Amento, B. *et al.*, "Fitt's Law", *CS 5724: Models and Theories of Human-Computer Interactions*, Virginia Tech, 1996, disponible en http://ei.cs.vt.edu/~cs5724/g1/.

[BAG01] Baggerman, L., y S. Bowman, *Web Design That Works*, Rockport Publishers, 2001.

[BUS96] Buschmann, F. *et al*., *Pattern-Oriented Software Architecture*, Wiley, 1996.

[CAC02] Cachero, C. *et al*., "Conceptual Navigation Analysis: a Device and Platform Independent Navigation Specification", *Proc. 2nd Intl. Workshop on Web-Oriented Technology*, junio de 2002, se puede descargar de www.dsic.upv.es/~west/iwwost02/papers/cachero.pdf.

[CLO01] Cloninger, C., *Fresh Styles for Web Designers*, New Riders Publishing, 2001.

[DIX99] Dix, A., "Design of User Interfaces for the Web", *Proc. Of User Interfaces to Data Systems Conference*, septiembre de 1999, se puede descargar de http://www.comp.lancs.ac.uk/computing/users/dixa/topics/webarch/.

[FIT54] Fitts, P., "The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement", en *Journal of Experimental Psychology*, vol 47, 1954, pp. 381-391
[FOW03] Fowler, M *et al*, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003
[GAL02] Galitz, W., *The Essential Guide to User Interface Design*, Wiley, 2002.
[GAM95] Gamma, E. *et al.*, *Design Patterns*, Addison-Wesley, 1995.
[GAR97] Garrido, A., G Rossi y D. Schwabe, "Patterns Systems for Hypermedia", 1997, se puede descargar de www inf.puc-rio.br/~schwabe/papers/PloP97 pdf
[GER00] German, D y D Cowan, "Toward a Unified Catalog of Hypermedia Design Patterns", *Proc. 33rd Hawaii Intl Conf. on System Sciences*, IEEE, vol. 6, Maui, Hawaii, junio de 2000, se puede descargar de www.turingmachine.org/~dmg/research/papers/dmg_hicss2000 pdf
[GNA99] Gnaho, C. y F. Larcher, "A User-Centered Methodology for Complex and Customizable Web Engineering", *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Ángeles, mayo de 1999
[HEI02] Heinicke, E., *Layout: Fast Solutions for Hands-On Design*, Rockport Publishers, 2002.
[IVO01] Ivory, M., R. Sinha y M Hearst, "Empirically Validated Web Page Design Metrics", ACM SIGCHI '01, Seattle, WA, abril de 2001, disponible en http://www rashmisinha.com/articles/WebTangoCHI01.html.
[JAC02] Jacyntho, D., D. Schwabe y G. Rossi, "An Architecture for Structuring Complex Web Applications", 2002, disponible en http://www2002.org/CDROM/alternate/478/.
[KAI02] Kaiser, J., "Elements of Effective Web Design", About, Inc , 2002, disponible en http://webdesign about.com/library/weekly/aa091998.htm
[KAL03] Kalman, S., *Web Security Field Guide*, Cisco Press, 2003
[KOC99] Koch, N., "A Comparative Study of Methods for Hypermedia Development", Technical Report 9905, Ludwig-Maximilians Universitat, Munich, Alemania, 1999, se puede descargar de http://www dsic upv.es/~west2001/iwwost01/files/contributions/NoraKoch/hypdev.pdf.
[KRA88] Krasner, G. y S. Pope, "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, vol 1, núm 3, agosto-septiembre de 1988, pp.26-49.
[LOW98] Lowe, D., y W Hall (eds.), *Hypertext and the Web- -An Engineering Approach*, John Wiley & Sons, 1998.
[MCC01] McClure, S J. Scambray y G Kurtz, *Hacking Exposed*, McGraw-Hill/Osborne, 2001
[MEN01] Mendes, E., N Mosley y S. Counsell, "Estimating Design and Authoring Effort", en *IEEE Multimedia*, enero-marzo de 2001, pp. 50-57.
[MIL00] Miller, E., "The Website Quality Challenge", Software Research, Inc., 2000, http://www soft com/eValid/Technology/White Papers/website.quality.challenge html.
[NIE96] Nielsen, J y A Wagner, "User Interface Design for the WWW", *Proc CHI '96 Conf. On Human Factors in Computing Systems*, ACM Press, 1996, pp 330-331.
[NIE00] Nielsen, J., *Designing Web Usability*, New Riders Publishing, 2000
[NOR02] Northcutt, S. y J. Novak, *Network Intrusion Detection*, New Riders Publishing, 2002
[OFF02] Offutt, J., "Quality Attributes of Web Software Applications", en *IEEE Software*, marzo-abril de 2002, pp. 25-32.
[OLS98] Olsina, L., "Building a Web-Based Information System Applying the Hypermedia Flexible Process Modeling Strategy", *Proc 1st Intl Workshop on Hypermedia Development*, 1998
[OLS99] Olsina, L. *et al.*, "Specifying Quality Characteristics and Attributes for Web Sites", *Proc 1st ICSE Workshop on Web Engineering*, ACM, Los Ángeles, mayo de 1999.
[PER99] Perzel, K. y D. Kane, "Usability Patterns for Applications on the World Wide Web", 1999, se puede descargar de http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/Kane/perzel_kane pdf
[POW00] Powell, T., *Web Design*, McGraw-Hill/Osborne, 2000.
[RAS00] Raskin, J., *The Humane Interface*, Addison-Wesley, 2000
[RHO98] Rho, Y y T Gedeon, "Surface Structures in Browsing the Web", *Proc. Australasian Computer Human Interaction Conference*, IEEE, diciembre de 1998.
[SCH95] Schwabe, D. y G Rossi, "The Object-Oriented Hypermedia Design Model", en *CACM*, vol 38, núm. 8, agosto de 1995, PP. 45-46

[SCH98] Schwabe, D. y G. Rossi, "Developing Hypermedia Applications Using OOHDM", *Proc Workshop on Hypermedia Development Process, Methods and Models, Hypertext '98*, 1998, se puede descargar de http://citeseer.nj.nec.com/schwabe98developing.html

[SCH01] Schwabe, D., G. Rossi y S. Barbosa, "Systematic Hypermedia Application Design Using OOHDM", 2001, disponible en http://www-di.inf.puc-rio.br/~schwabe/HT96WWW/section1.html.

[TIL00] Tillman, H. N., "Evaluating Quality On the Net", Babson College, 30 de mayo de 2000, disponible en http://www.hopetillman.com/findqual.html#2

[TOG01] Tognozzi, B., "First Principles", *askTOG*, 2001, disponible en http://www.asktog.com/basics/firstPrinciples.html

[WMT02] Web Mapping Testbed Tutorial, 2002, disponible en http://www.webmapping.org/vcgdocuments/vcgTutorial/

[ZHA02] Zhao, H., "Fitt's Law: Modeling Movement Time in HCI", *Theories in Computer Human Interaction*, University of Maryland, octubre de 2002, disponible en http://www.cs.umd.edu/class/fall2002/cmsc838s/tichi/fitts.html.

PROBLEMAS Y PUNTOS A CONSIDERAR

19.1. ¿Por qué el "ideal artístico" es una filosofía de diseño insuficiente cuando se construyen las WebApps modernas? ¿Existe un caso en el que el ideal artístico sea la filosofía que debe seguirse?

19.2. En este capítulo se analizó una amplia variedad de atributos de calidad para las WebApps. Elíjanse las tres que se considere como las más importantes y elabórese un argumento que explique por qué cada uno debe resaltarse en el trabajo de diseño de ingeniería Web.

19.3. Agréguense al menos cinco preguntas adicionales a la "Lista de verificación de la calidad del diseño de la WebApp" presentada en una barra lateral en la sección 19.1.1.

19.4. Revisar los principios de diseño de la interfaz de Tognozzi tratados en la sección 19.3.1. Considerar cada principio para una WebApp operativa con la cual se esté familiarizado. Calificar la WebApp (úsense calificaciones A, B, C, D o F) en relación con el grado en el cual ha logrado el principio. Explicar la razón para cada calificación.

19.5. Diseñar una interfaz prototipo para la WebApp de HogarSeguroInc.com. Inténtese ser innovador pero, al mismo tiempo, se debe asegurar que la interfaz se ajusta a los principios para el buen diseño de la interfaz.

19.6. ¿Se han encontrado mecanismos de control de la interfaz que sean diferentes a los anotados en la Sección 19.3.2? Si es así, descríbanse brevemente.

19.7. El lector es el diseñador WebApp para una compañía de enseñanza a larga distancia. Su intención es implementar un "motor de aprendizaje" basado en Internet que le permitirá entregar contenido del curso a los estudiantes. El motor de aprendizaje ofrece la infraestructura básica para entregar contenido de aprendizaje de cualquier materia (diseñadores de contenido prepararán el contenido adecuado). Desarróllese un diseño de interfaz prototipo para el motor de aprendizaje.

19.8. ¿Cuál es el sitio Web estéticamente más agradable que el lector haya ha visitado y por qué?

19.9. Considerar el objeto de contenido **pedido**, generado una vez que un usuario de HogarSeguroInc.com ha completado la selección de todos los componentes y está listo para finalizar su compra. Desarrollar una descripción UML de **pedido** junto con todas las representaciones de diseño apropiadas.

19.10. ¿Cuál es la diferencia entre arquitectura de contenido y arquitectura de WebApp?

19.11. Reconsidérese el "motor de aprendizaje" descrito en el problema 19.7, selecciónese una arquitectura de contenido que sería apropiada para la WebApp. Coméntese por qué se hizo dicha elección.

19.12. Con UML desarróllense tres o cuatro representaciones de diseño para objetos de contenido que podrían encontrarse conforme se diseña el "motor de aprendizaje" descrito en el problema 19 7

19.13. Hacer un poco de investigación adicional acerca de la arquitectura MVC y decidir si sería una arquitectura WebApp apropiada para el "motor de aprendizaje" mencionado en el problema 19 7

19.14. ¿Cuál es la diferencia entre sintaxis de navegación y semántica de navegación?

19.15. Definir dos o tres USN para la WebApp de HogarSeguroInc.com Describir cada una con cierto detalle

19.16. Hacer alguna investigación y presentar a su clase dos o tres patrones de diseño hipermedia completos.

Otras lecturas y fuentes de información

Aunque se han escrito cientos de libros acerca del "diseño Web", muy pocos abordan algunos métodos técnicos significativos para realizar el trabajo de diseño. Cuando mucho, se presentan varios lineamientos útiles para el diseño de la WebApp, ejemplos valiosos de páginas Web y se muestra programación Java, y se analizan los detalles técnicos importantes para implementar WebApps modernas. Entre los muchos que se ofrecen en esta categoría, vale la pena considerar la discusión enciclopédica de Powell [POW00]. Además, los libros de Galitz [GAL02], Heinicke [HEI02], Schmitt (*Designing CSS Web Pages*, New Riders Publishing, 2002), Donnelly (*Designing Easy-to Use Websites*, Addison-Wesley, 2001) y Nielsen [NIE00] proporcionan una guía útil.

La visión ágil del diseño (y otros tópicos) para WebApps la presentan Wallace y sus colegas (*Extreme Programming for Web Projects*, Addison-Wesley, 2003) Conallen (*Building Web Applications with UML*, segunda edición, Addison-Wesley, 2002) y Rosenberg y Scott (*Applying Use-Case Driven Object Modeling with UML*, Addison-Wesley, 2001) presentan ejemplos detallados de WebApps modeladas con la aplicación de UML.

Van Duyne y sus colegas (*The Design of Sites: Patterns, Principles and Processes*, Addison-Wesley, 2002) escribieron un libro excelente que cubre los aspectos más importantes del proceso de diseño en la ingeniería Web. Se cubren en detalle los modelos de proceso de diseño y los patrones de diseño. Wodtke (*Information Architecture*, New Riders Publishing, 2003), Rosenfeld y Morville (*Information Architecture for the World Wide Web*, O'Reilly & Associates, 2002), y Reiss (*Practical Information Architecture*, Addison-Wesley, 2000) abordan la arquitectura de contenido y otros tópicos.

Las técnicas de diseño también se mencionan en libros escritos acerca de ambientes de desarrollo específicos. Los lectores interesados deben examinar libros acerca de J2EE, Java, ASP.NET, CSS, XML, Perl y una diversidad de aplicaciones de creación de WebApps (*Dreamweaver, Home Page, Frontpage, GoLive, MacroMedia Flash*, etc.) para comentarios de diseño útiles.

En Internet está disponible una gran variedad de fuentes de información acerca de diseño para ingeniería Web Una lista actualizada de referencias en la World Wide Web se encuentra en el sitio Web de SEPA:
http://www.mhhe.com/pressman.

CAPÍTULO

20 Cómo probar aplicaciones Web

Conceptos clave

Existe una urgencia que siempre permea el proceso de ingeniería Web. Conforme se dirigen la formulación, la planeación, el análisis, el diseño y la construcción, los participantes —preocupados acerca de la competencia de otras WebApps, fustigados por las demandas de los clientes e intranquilos porque perderán una ventana en el mercado— presionan para poner la WebApp en línea. Como consecuencia, las actividades técnicas que usualmente ocurren en las últimas etapas del proceso de ingeniería Web, como la prueba de la WebApp, en ocasiones reciben poca atención. Esto puede ser un error catastrófico. Para evitarlo, el equipo de ingeniería Web debe asegurarse de que cada producto de trabajo de IWeb muestre alta calidad. Wallace y sus colegas [WAL03] advierten esto cuando afirman:

> Llevar a cabo la prueba no debe esperar hasta que termine el proyecto. Comience a probar antes de escribir una línea de código. Pruebe constante y efectivamente y desarrollará un sitio Web mucho más durable

Dado que los modelos de análisis y diseño no pueden ponerse a prueba en el sentido clásico, el equipo de ingeniería Web debe dirigir revisiones técnicas formales (capítulo 26), así como pruebas ejecutables. El objetivo es descubrir y corregir errores antes de que la WebApp se ponga a disposición de sus usuarios finales.

Un vistazo rápido

¿Qué es? El proceso de someter a prueba la WebApp es una suma de actividades relacionadas con una sola meta: descubrir errores en el contenido, la función, la facilidad de uso, la navegabilidad, el desempeño, la capacidad y la seguridad de la WebApp. Esto se logra a lo largo de todo el proceso de ingeniería Web mediante la aplicación de una estrategia de prueba que abarca tanto revisiones como pruebas ejecutables.

¿Quién lo hace? Los ingenieros Web y otros participantes del proyecto (gerentes, clientes, usuarios finales) toman parte en el proceso de probar la WebApp.

¿Por qué es importante? Si los usuarios finales encuentran errores que afecten su confianza en la WebApp, se irán a cualquier otra parte por el contenido y la función que necesitan, y la WebApp fracasará. Por esta razón, los ingenieros Web deben trabajar para eliminar tantos errores como sea posible antes de que la WebApp esté en línea.

¿Cuáles son los pasos? El proceso de prueba de la WebApp comienza al enfocarse sobre aquellos aspectos de ésta que son visibles para el usuario y procede a probar dicha tecnología e infraestructura. La prueba consta de siete etapas: contenido, interfaz, navegación, componente, configuración, desempeño y prueba de seguridad.

¿Cuál es el producto obtenido? En algunos casos se produce un plan de prueba de la WebApp. En todos los casos se desarrolla un conjunto de casos prueba para cada etapa de la prue-

ca y se conserva un archivo de resultados de pruebas para uso futuro.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Aunque nunca se puede estar seguro de que han llevado a cabo todas las pruebas que se necesitan, puede tenerse la seguridad de que la puesta en prueba ha descubierto errores (y que éstos se han corregido). Además, si se ha establecido un plan de prueba, puede verificarse para asegurar que se han realizado todas las pruebas planeadas

20.1 PRUEBA DE CONCEPTOS PARA WEBAPPS

En el capítulo 13 se señaló que la prueba es el proceso de ejercitar al software con la finalidad de encontrar (y a final de cuentas corregir) errores. Esta filosofía fundamental no cambia para las WebApps. De hecho, puesto que los sistemas y aplicaciones basados en Web residen en una red e interoperan con muchos sistemas operativos diferentes, navegadores (u otros dispositivos de interfaz como PDA o teléfonos celulares), plataformas de hardware, protocolos de comunicaciones y aplicaciones "de cuarto trasero", la búsqueda de errores representa un desafío significativo para los ingenieros Web.

La comprensión de los objetivos de las pruebas dentro de un contexto de ingeniería Web requiere considerar las diversas dimensiones de la calidad WebApp [1]. En el contexto de esta exposición se consideran las dimensiones de calidad que son particularmente relevantes en cualquier debate de las pruebas para el trabajo de ingeniería Web. También se considera la naturaleza de los errores que se encuentran como consecuencia de las pruebas, y la estrategia de poner a prueba aplicable para descubrir dichos errores.

20.1.1 Dimensiones de calidad

La calidad se incorpora en una aplicación Web como consecuencia de un buen diseño. Se evalúa al aplicar una serie de revisiones técnicas que valoran varios elementos del modelo de diseño y al aplicar un proceso de prueba que se trata a lo largo de este capítulo. Tanto las revisiones como las pruebas examinan una o más de las siguientes dimensiones de calidad [MIL00]:

- El *contenido* se evalúa tanto en el ámbito sintáctico como semántico. En el ámbito sintáctico, la ortografía, la puntuación y la gramática se valoran para los documentos basados en texto. En el ámbito semántico se valoran la exactitud (de la información presentada), la consistencia (a través de todos los objetos de contenido y objetos relacionados) y la falta de ambigüedad
- La *función* se prueba para descubrir errores que indiquen que no hay concordancia con los requisitos del cliente. Cada función de la WebApp se valora en

1 En el capítulo 19 también se consideró la calidad de la WebApp

cuanto a exactitud, inestabilidad y concordancia general con los estándares de implementación apropiados (por ejemplo, estándares de lenguaje Java o XML)

- La *estructura* se valora para asegurarse de que entrega adecuadamente contenido y función de la WebApp, que es extensible y puede sostenerse conforme se añade nuevo contenido o funcionalidad
- La *facilidad* de uso se prueba para garantizar que a cada categoría de usuario la soporta la interfaz; puede aprender y aplicar toda la sintaxis y semántica de navegación requerida.
- La *navegabilidad* se pone a prueba para garantizar que toda la sintaxis y semántica de navegación se ejercen para descubrir cualquier error de navegación (por ejemplo, vínculos rotos, vínculos inadecuados, vínculos erróneos).
- El *desempeño* se prueba en una diversidad de condiciones operativas, configuraciones y cargas para asegurar que el sistema responde a la interacción del usuario y maneja cargas extremas sin que haya una degradación operativa inaceptable.
- La *compatibilidad* se prueba al ejecutar la WebApp en varias configuraciones huésped, en los lados tanto del cliente como del servidor. El objetivo es encontrar errores específicos respecto a sólo una configuración huésped.
- La *interoperabilidad* se prueba para asegurar que la WebApp realiza interfaces adecuadas con otras aplicaciones o bases de datos.
- La *seguridad* se prueba al valorar las vulnerabilidades potenciales e intentar explotar cada una de ellas. Cualquier intento de penetración exitoso se considera una falla en la seguridad

En este capítulo se estudian, más adelante, una estrategia y algunas tácticas que se han desarrollado para poner a prueba cada una de las anteriores características de calidad de una WebApp

> "La innovación es un asunto agridulce para quienes ponen a prueba el software. Justo cuando parece que se sabe cómo probar una tecnología particular, llega una nueva [WebApp] y todas las apuestas se pierden."
>
> **James Bach**

20.1.2 Errores dentro de un ambiente WebApp

Ya se ha señalado que el intento primario de realizar pruebas en cualquier contexto de software es descubrir errores (y corregirlos). Los errores encontrados como consecuencia de la prueba exitosa de la WebApp tienen varias características únicas [NGU00]:

1. Puesto que muchos tipos de pruebas de WebApp descubren problemas que se evidencian primero en el lado del cliente (es decir, a través de una interfaz implementada en un navegador específico, una PDA o un teléfono celular), el ingeniero Web ve un síntoma del error, no el error en sí.

2. Puesto que una WebApp se implementa en varias configuraciones diferentes y dentro de distintos ambientes, puede ser difícil o imposible reproducir un error afuera del ambiente en el que el error se encontró originalmente.
3. Aunque algunos errores son resultado de un diseño incorrecto o una codificación HTML impropia (o algún otro lenguaje de programación), muchos errores pueden rastrearse hacia la configuración de la WebApp
4. Puesto que las WebApp residen dentro de una arquitectura cliente/servidor, el rastreo de los errores puede ser difícil a través de las tres capas arquitectónicas: el cliente, el servidor o la red en sí.
5. Algunos errores se deben al ambiente operativo estático (es decir, la configuración específica en la que se desarrolla la prueba), mientras que otros son atribuibles al ambiente operativo dinámico (es decir, la carga instantánea de recursos o los errores relacionados con el tiempo)

Estos cinco atributos de error sugieren que el ambiente desempeña un importante papel en el diagnóstico de todos los errores descubiertos durante el proceso de ingeniería Web. En algunas situaciones (por ejemplo, prueba de contenido), el sitio del error es obvio, pero en muchos otros tipos de pruebas de WebApp (por ejemplo, pruebas de navegación, de desempeño, de seguridad) la causa subyacente del error tal vez sea considerablemente más difícil de determinar.

20.1.3 Estrategias de pruebas

La estrategia para probar una WebApp adopta los principios básicos para todas las pruebas de software (capítulo 13) y aplica una estrategia y las tácticas que se recomendaron respecto de los sistemas orientados a objetos (capítulo 14). Los siguientes pasos resumen el enfoque:

1. Se revisa el modelo de contenido de la WebApp para descubrir errores
2. Se revisa el modelo de la interfaz para asegurarse que todos los casos de uso pueden acomodarse.
3. Se revisa el modelo de diseño de la WebApp para descubrir errores de navegación.
4. Se prueba la interfaz del usuario para descubrir errores en la presentación o los mecanismos de navegación
5. Componentes funcionales seleccionados se prueban en forma individual
6. Se prueba la navegación a través de toda la arquitectura.
7. La WebApp se implementa en diversas configuraciones ambientales y se prueba su compatibilidad con cada configuración
8. Se realizan pruebas de seguridad con el objetivo de explotar vulnerabilidades en la WebApp o dentro de su ambiente.

Referencia Web

Se pueden encontrar excelentes artículos acerca de las pruebas de WebApps en **www.stickyminds.com/testing.asp.**

9. Se llevan a cabo pruebas de desempeño.
10. La WebApp se prueba en una población controlada y monitoreada de usuarios finales; los resultados de su interacción con el sistema se evalúan para buscar errores de contenido y navegación, relacionados con la facilidad de uso, con la compatibilidad y con la confiabilidad y el desempeño de la WebApp.

Puesto que muchas WebApps evolucionan continuamente, la prueba de la WebApp es una actividad de seguimiento que dirige el equipo de soporte Web, que utiliza pruebas de regresión derivadas de las pruebas desarrolladas cuando la WebApp se sometió a ingeniería por primera ocasión.

20.1.4 Planeación de las pruebas

El empleo de la palabra planeación (en cualquier contexto) es anatema para algunos desarrolladores Web. Como se anotó en capítulos anteriores, dichos profesionales sólo comienzan, pues temen el surgimiento de algún saboteador de WebApps. Un ingeniero Web reconoce que la planeación establece un mapa vial para todo el trabajo que sigue. Vale la pena el esfuerzo.

En su libro acerca de las pruebas de las WebApps, Splaine y Jaskiel [SPL01] afirman:

> Excepto por el más simple de los sitios Web, rápidamente se vuelve aparente que es necesaria cierta especie de planeación de pruebas. Con demasiada frecuencia, el número inicial de errores que se encuentran a partir de una prueba adecuada es lo suficientemente grande como para que no todos se fijen la primera vez que se detectan. Esto pone una presión adicional sobre la gente que prueba los sitios y aplicaciones Web. No sólo deben conjurar nuevas pruebas imaginativas, también deben recordar cómo se ejecutaron las pruebas anteriores con la finalidad de volver a probar con confiabilidad el sitio/la aplicación Web, y asegurarse de que los errores conocidos se han removido y que no se han introducido otros nuevos.

PUNTO CLAVE

El plan de prueba identifica un conjunto de tareas de prueba, los productos de trabajo que se desarrollarán y la forma en la cual los resultados se evalúan, registran y reutilizan.

La pregunta para todo ingeniero Web es: ¿cómo "conjuro nuevas pruebas imaginativas" y en qué se deben enfocar dichas pruebas? La respuesta se encuentra dentro de un plan de pruebas.

Un plan de pruebas WebApp identifica 1) un conjunto de tareas[2] que se aplicarán cuando comience la prueba, 2) los productos de trabajo que se generarán conforme se ejecute cada tarea de prueba, y 3) la forma en la que los resultados de las pruebas se evalúan, registran y reutilizan cuando se realicen pruebas de regresión. En algunos casos, el plan de pruebas se integra con el plan del proyecto; en otros, el plan de pruebas es un documento separado.

2 Los conjuntos de tareas se estudian en el capítulo 2. En este libro también se ha empleado un término relacionado —flujo de trabajo— para describir la serie de tareas necesaria para completar una actividad de ingeniería del software.

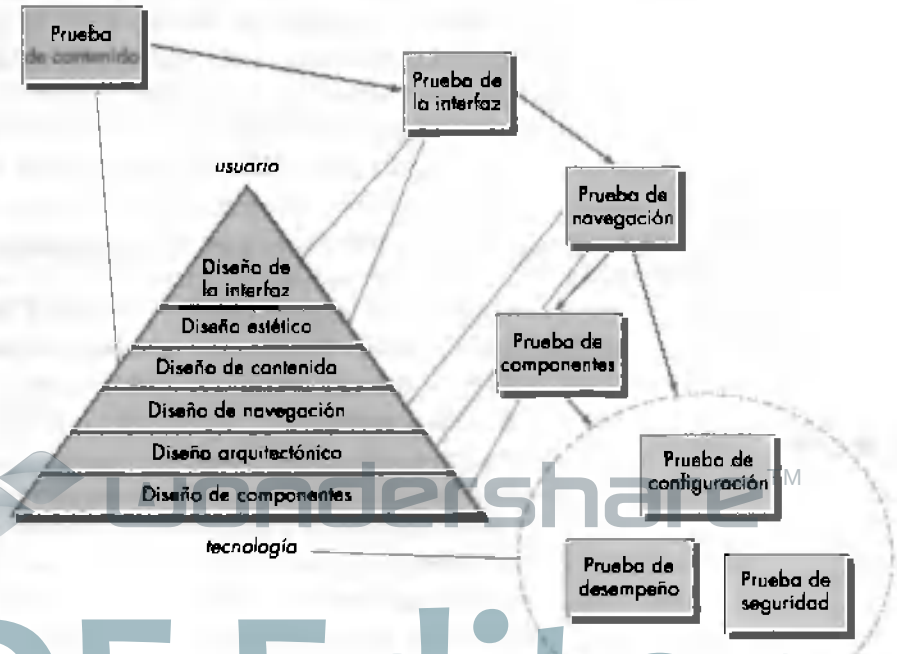
20.2 EL PROCESO DE PRUEBA: UN PANORAMA

Los procesos de prueba para ingeniería Web comienzan con pruebas que ejercitan el contenido y la funcionalidad de la interfaz que es inmediatamente visible para los usuarios finales. Conforme se realizan las pruebas, se ejercitan los aspectos de la arquitectura de diseño y la navegación. El usuario puede o no conocer estos elementos de la WebApp. Finalmente, el foco se cambia a las pruebas que ejercitan las capacidades tecnológicas que no siempre son aparentes para los usuarios finales: la infraestructura de la WebApp y cuestiones de instalación/implementación.

> "En general, las técnicas de prueba de software [capítulos 13 y 14] que se utilizan con otras aplicaciones son los mismos que los empleados en las aplicaciones basadas en Web... La diferencia entre los dos tipos de pruebas es que se multiplican las variables de tecnología en el ambiente Web."
>
> Hung Nguyen

La figura 20.1 yuxtapone el proceso de prueba WebApp con la pirámide de diseño examinada en el capítulo 19. Nótese que, conforme se desarrolló el flujo de pruebas, de izquierda a derecha y de arriba abajo, los elementos del diseño WebApp visibles para el usuario (elementos superiores de la pirámide) se prueban primero, seguidos por los elementos de diseño de infraestructura.

Figura 20.1 Proceso de prueba.



La *prueba de contenido* (y las revisiones) intentan descubrir errores en el conte-
do. Esta actividad de prueba es similar en muchos aspectos a la copia-edición de
documento escrito. De hecho, un gran sitio Web puede reclutar los servicios de un co-
rrector de estilo profesional para descubrir errores tipográficos, equívocos grama-
cales, errores en la consistencia del contenido, inexactitudes en las representacion
gráficas y fallas en las referencias cruzadas. Además de examinar el contenido está-
tico en busca de errores, esta etapa de las pruebas también considera el conteni
dinámico derivado de los datos conservados como parte de un sistema de base de
datos integrado a la WebApp

La *prueba de la interfaz* ejercita los mecanismos de interacción y valida los aspec-
tos estéticos de la interfaz del usuario. El objetivo es descubrir los errores que re-
tan de mecanismos con una pobre implementación de interacción, u omisiones, in-
sistencias o ambigüedades que se han introducido a la interfaz en forma inadvertida

La *prueba de navegación* aplica casos de uso, derivados como parte de la actu-
dad de análisis, en el diseño de casos de prueba que ejerciten cada escenario de uso
contra el diseño de navegación

Los mecanismos de navegación (por ejemplo, barras de menú) implementa
dentro de la plantilla de la interfaz se prueban contra casos de uso y USN (capí
19) para garantizar que los errores que impiden completar un caso de uso se ide-
fiquen y corrijan.

La *prueba de componentes* ejercita el contenido y las unidades funcionales de
de la WebApp. Cuando se consideran las WebApps, cambia el concepto de uni
(introducido en el capítulo 13) La "unidad" de elección dentro de la arquitectura de
contenido (capítulo 19) es la página Web. Cada página Web encapsula conter
vínculos de navegación y elementos de procesamiento (formatos, guiones, *appl*
Una "unidad" dentro de la arquitectura WebApp puede ser un componente funci
definido que proporciona servicio directamente a un usuario final o un compon
de infraestructura que posibilita que la WebApp desarrolle todas sus capacida
Cada componente funcional se prueba, en gran parte, en la misma forma que se
prueba un módulo individual en el software convencional. En la mayoría de los ca-
sos, las pruebas están orientadas a las cajas negras. Sin embargo, si el procesam
to es complejo, también se pueden usar pruebas de cajas blancas.[3] Además de la
prueba funcional, también se ejercitan las capacidades de bases de datos

PUNTO CLAVE

La estrategia para la prueba de integración depende de la arquitectura de la WebApp elegida durante el diseño

Conforme se construye la arquitectura de la WebApp, las pruebas de la nave-
ción y los componentes se utilizan como *pruebas de integración*. La estrategia para
la prueba de integración depende del contenido y la arquitectura WebApp que se ha-
ya elegido (capítulo 19). Si la arquitectura de contenido se diseña con estructura li-
neal, retícula o jerárquica simple, es posible integrar páginas Web en gran parte de
la misma forma como se integran módulos para el software convencional. Sin em-
bargo, si se usa una estructura de jerarquía mixta o de red (Web), la prueba de in-

3 Las técnicas de pruebas de caja negra y caja blanca se examinan en el capítulo 14

gración es similar al enfoque usado para los sistemas OO. Las pruebas basadas en ligas (capítulo 14) se pueden aprovechar para integrar el conjunto de páginas Web (se puede usar una USN para definir el conjunto apropiado) requerido para responder a un evento de usuario. Cada liga se integra y pone a prueba individualmente Mediante las pruebas de regresión se asegura que no ocurran efectos colaterales Las pruebas de agrupamiento integran un conjunto de páginas asociadas (determinadas mediante el examen de los casos de uso y la USN) Los casos de prueba se derivan para descubrir los errores en las colaboraciones.

Cada elemento de la arquitectura WebApp se prueba de manera unitaria en la medida de lo posible. Por ejemplo, en una arquitectura MVC (capítulo 19), los componentes *modelo, vista* y *controlador* se prueban cada uno de manera individual. Después de la integración, el flujo de control y datos a través de cada uno de estos elementos se valora en detalle

Las *pruebas de configuración* intentan descubrir los errores que son específicos respecto de un cliente o ambiente de servidor particulares Se crea una matriz de referencia cruzada que define todos los probables sistemas operativos, navegadores,[4] plataformas de hardware y protocolos de comunicación. Entonces las pruebas se encaminan a descubrir los errores asociados con cada posible configuración.

La *prueba de seguridad* incorpora una serie de pruebas diseñadas para explotar las vulnerabilidades en la WebApp y su ambiente. El objetivo es demostrar la posibilidad de una brecha en la seguridad.

La *prueba de desempeño* abarca una serie de pruebas diseñadas para valorar 1) cómo afecta el aumento del tráfico de usuarios la respuesta en tiempo y confiabilidad de la Web, 2) cuáles componentes de la WebApp son responsables de la degradación del desempeño y qué características de uso provocan que ocurra la degradación, y 3) cómo la degradación del desempeño impacta los objetivos y requisitos globales de la WebApp.

CONJUNTO DE TAREAS

Prueba de la WebApp

1 Revisar los requisitos de los participantes. Identificar los metas y objetivos de los usuarios. Revisar los casos de uso respecto de cada categoría de usuario.
Establecer prioridades para asegurar que cada meta y objetivo de usuario se probarán de manera adecuada.

3 Definir la estrategia de prueba de la WebApp al describir los tipos de pruebas (sección 20.2) que se realizarán

4 Desarrollar un plan de prueba
Definir un calendario de pruebas y asignar responsabilidades a cada prueba.
Especificar herramientas automatizadas para realizar las pruebas.

4 Los navegadores son notables porque implementan sus propios "estándares", sutilmente diferentes en las interpretaciones de HTML y Javascript.

Definir criterios de aceptación para cada clase de prueba.
Especificar mecanismos de rastreo de defectos.
Definir mecanismos de reporte de problemas
5. Realizar pruebas "unitarias".
Revisar el contenido para errores de sintaxis y semántica.
Revisar el contenido para clarificaciones y permisos adecuados.
Probar los mecanismos de la interfaz para una operación correcta
Probar cada componente (por ejemplo, guión) para asegurar el funcionamiento adecuado.
6. Realizar pruebas de "integración"
Probar la semántica de la interfaz respecto de los casos de uso.
Dirigir pruebas de navegación.
7. Realizar pruebas de configuración.
Valorar la compatibilidad de configuración en el lado del cliente.
Valorar configuraciones en el lado del servidor.
8. Dirigir pruebas de desempeño.
9 Dirigir pruebas de seguridad

20.3 PRUEBA DEL CONTENIDO

CONSEJO

Aunque las revisiones técnicas formales no son parte de una prueba, se deben llevar a cabo revisiones de contenido para garantizar la calidad del contenido.

Los errores en el contenido de la WebApp pueden ser tan triviales como errores tipográficos menores o tan significativos como información incorrecta, organización impropia o violación de las leyes de propiedad intelectual. La prueba del contenido intenta descubrir éstos y muchos otros problemas antes de que el usuario los encuentre.

La prueba del contenido combina tanto revisiones como la generación de casos de prueba ejecutables. La revisión se aplica para descubrir errores semánticos en el contenido (examinados en la sección 20.3.1). La prueba ejecutable se aprovecha para descubrir errores de contenido susceptibles de rastrear hacia contenido dinámicamente derivado que hayan suministrado los datos adquiridos de una o más bases de datos.

20.3.1 Objetivos de la prueba de contenido

PUNTO CLAVE

Los objetivos de la prueba de contenido son 1) descubrir errores sintácticos en el contenido, 2) descubrir errores semánticos y 3) encontrar errores estructurales.

La prueba del contenido tiene tres objetivos importantes: 1) descubrir errores sintácticos (por ejemplo, errores tipográficos, equívocos gramaticales) en los documentos basados en texto, representaciones gráficas y otros medios audiovisuales, 2) descubrir errores semánticos (es decir, errores en la precisión de la información o que ésta sea incompleta) en cualquier objeto de contenido presentado conforme ocurra la navegación, y 3) hallar errores en la organización o estructura del contenido que se presenta al usuario final

El primer objetivo se logra empleando verificadores de ortografía y gramática automatizados. Sin embargo, muchos errores sintácticos evaden la detección mediante tales herramientas y debe descubrirlos un revisor humano (examinador). Como se anotó en la sección anterior, la corrección de estilo es el mejor enfoque para encontrar errores sintácticos.

La prueba semántica se centra en la información presentada dentro de cada objeto de contenido. El revisor (examinador) debe responder las siguientes preguntas:

- ¿La información realmente es precisa?
- ¿La información es concisa y exacta?

¿Qué preguntas se deben plantear y responder para descubrir errores semánticos en el contenido?

- ¿La plantilla del objeto de contenido es fácil de entender para el usuario?
- ¿La información anidada en un objeto de contenido se encuentra con facilidad?
- ¿Se han ofrecido referencias adecuadas para toda la información derivada de otras fuentes?
- ¿La información presentada es consistente internamente y con la información que presentan otros objetos de contenido?
- ¿El contenido es ofensivo, engañoso o abre la puerta a litigios?
- ¿El contenido infringe derechos de autor o marcas registradas existentes?
- ¿El contenido contiene vínculos internos que complementan el contenido existente? ¿Los vínculos son correctos?
- ¿El estilo estético del contenido entra en conflicto con el estilo estético de la interfaz?

La obtención de respuestas a cada una de estas preguntas en una WebApp grande (que contiene cientos de objetos de contenido) puede ser una labor atemorizante. Sin embargo, el fracaso para descubrir errores semánticos alterará la fe del usuario en la WebApp y puede conducir a fallas de la aplicación basada en Web.

Los objetos de contenido existen dentro de una arquitectura que tiene un estilo específico (capítulo 19). Durante la prueba del contenido, la estructura y organización de la arquitectura del contenido se prueba para garantizar que el contenido requerido se presenta al usuario final en el orden y las relaciones adecuados Por ejemplo, la WebApp HogarSeguroInc.com[5] presenta una variedad de información acerca de sensores que se utilizan como parte de productos de seguridad y vigilancia. Los objetos de contenido proporcionan información descriptiva, especificaciones técnicas, una representación fotográfica e información relacionada. Las pruebas de la arquitectura de contenido de HogarSeguroInc.com se esfuerzan por descubrir errores en la presentación de esta información (por ejemplo, una descripción del Sensor *X* se presenta con una fotografía del Sensor *Y*).

20.3.2 Prueba de las bases de datos

Las modernas aplicaciones Web hacen mucho más que presentar objetos de contenido estáticos. En muchos dominios de aplicación, la interfaz de las WebApps con bases de datos sofisticados gestionan sistemas y construyen objetos de contenido dinámicos que se crean en tiempo real aprovechando datos adquiridos de una base de datos.

Por ejemplo, una WebApp de servicios financieros puede producir compleja información basada en texto, tabular y gráfica acerca de una participación accionaria específica (por ejemplo, fondos de acciones o de inversión colectiva). El objeto de contenido compuesto que presenta esta información se crea dinámicamente después de

5 La WebApp HogarSeguroInc.com se ha utilizado como ejemplo a lo largo de la Parte 3 de este libro

que el usuario ha consultado información acerca de una participación accionaria específica. Esto se logra mediante los siguientes pasos: 1) se consulta una gran base de datos de participaciones accionarias, 2) se extraen datos relevantes de la base de datos, 3) los datos extraídos se deben organizar como un objeto de contenido, y 4) este objeto de contenido (que representa información personalizada solicitada por un usuario final) se transmite al ambiente del cliente para su despliegue. Los errores pueden ocurrir, y de hecho lo hacen, como consecuencia de cada uno de estas etapas. El objetivo de probar la base de datos es descubrir dichos errores

La prueba de la base de datos para las WebApps es complicada por varios factores

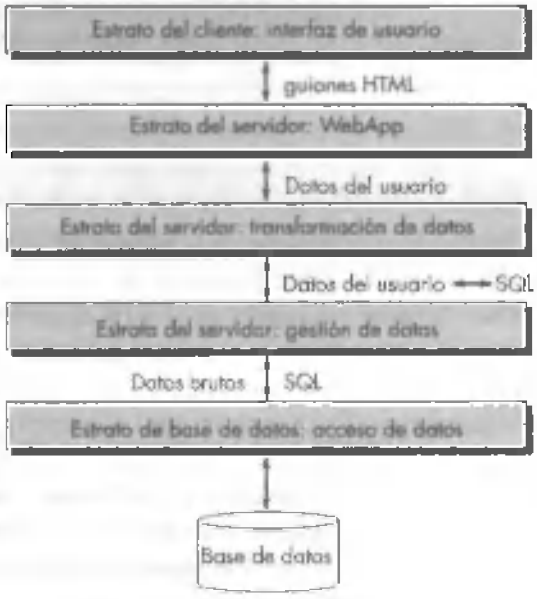
¿Qué conflictos complican la prueba de bases de datos para WebApps?

1. *La solicitud original de información en el lado del cliente rara vez se presenta en la forma (por ejemplo, lenguaje de consultas estructurado, [SQL, por sus siglas en inglés])* que pueda introducirse en un sistema gestor de bases de datos (DBMS por sus siglas en inglés). En consecuencia, las pruebas se deben diseñar para descubrir errores cometidos al traducir la solicitud del usuario en una forma que puedan procesar dichos DBMS.
2. *La base de datos quizá sea remota al servidor que hospeda la WebApp.* Por lo tanto, se deben desarrollar las pruebas que descubran los errores en la comunicación entre la WebApp y la base de datos remota.[6]
3. *Los datos brutos adquiridos de la base de datos se deben transmitir al servidor de la WebApp y formatearse adecuadamente para su transmisión subsecuente al cliente.* En consecuencia, se deben desarrollar pruebas que demuestren la validez de los datos brutos que recibe el servidor WebApp, y también se deben crear pruebas adicionales que demuestren la validez de las transformaciones aplicadas a los datos brutos para crear objetos de contenido válidos.
4. *Los objetos de contenido dinámico se deben transmitir al cliente en una forma que se pueda desplegar al usuario final.* Por lo tanto, se debe diseñar una serie de pruebas para *a)* descubrir errores en el formato de objeto de contenido, y *b)* probar la compatibilidad con diferentes configuraciones de ambiente de cliente.

Al considerar estos cuatro factores, se deben aplicar los métodos de diseño de casos de prueba para cada uno de los "estratos de interacción" [NGU01] anotados en la figura 20.2. Las pruebas deben asegurar que 1) información válida pasa entre el cliente y el servidor desde el estrato de la interfaz; 2) la WebApp procese los guiones correctamente y extraiga o formatee adecuadamente datos del usuario; 3) los datos del usuario pasen correctamente a una función de transformación de datos en el lado del servidor para formatear consultas apropiadas (por ejemplo, SQL); 4) las con-

6 Dichas pruebas se vuelven complejas cuando se encuentran bases de datos distribuidas o cuando se requiere el acceso a un almacén de datos (capítulo 10).

Figura 20.2
Niveles de interacción.



sultas pasen a un estrato de gestión de datos[7] que se comunique con rutinas de acceso a bases de datos (potencialmente ubicadas en otra máquina).

Los estratos de transformación de datos, gestión de datos y acceso a bases de datos, que se muestran en la figura 20.2, usualmente se construyen con componentes reutilizables que se han validado por separado y como paquete. Si éste es el caso, las pruebas de la WebApp se centran en el diseño de casos de prueba para ejercitar las interacciones entre el estrato del cliente y los primeros dos estratos del servidor (WebApp y transformación de datos) mostrados en la figura.

El estrato de la interfaz del usuario se prueba para garantizar que los guiones HTML están construidos de manera adecuada para cada solicitud de usuario y se transmiten adecuadamente al lado del servidor. La capa WebApp en el lado del servidor se prueba para asegurar que los datos del usuario se extraen adecuadamente de guiones HTML y se transmiten de manera adecuada al estrato de transformación de datos en el lado del servidor.

Las funciones de transformación de datos se prueban para asegurar que se crea el SQL correcto y que pasa a componentes apropiados de gestión de datos.

Una exposición detallada de la tecnología subyacente que se debe entender para diseñar apropiadamente dichas pruebas de bases de datos está más allá del alcance de este libro. El lector interesado debe consultar [SCE02], [NGU01] y [BRO01].

7 La capa de gestión de datos por lo general incorpora una interfase SQL al nivel de llamada (SQL CLI) como puede ser Microsoft OLE/ADO o la Conectividad de Bases de Datos Java (Java Database Connectivity, JDBC).

"Como clientes electrónicos (ya sea de negocios a consumo) es improbable que tengamos confianza en un sitio Web que sufra de frecuentes periodos de inactividad, cuelga a la mitad de una transacción o tiene un mal sentido de la facilidad de uso. Las pruebas, por lo tanto, tienen un papel importantísimo en el proceso de desarrollo global."

Wing Lam

20.4 PRUEBA DE LA INTERFAZ DEL USUARIO

La verificación y validación de la interfaz del usuario de una WebApp ocurre en tres puntos distintos durante el proceso de ingeniería Web. Durante la formulación y el análisis de requisitos (capítulos 17 y 18) se revisa el modelo de la interfaz para garantizar que se ajusta a los requisitos del cliente y a otros elementos del modelo de análisis. Durante el diseño (capítulo 19) se revisa el modelo de diseño de la interfaz para garantizar que se han alcanzado los criterios genéricos de calidad establecidos para todas las interfaces de usuario, y que los conflictos en el diseño de la interfaz específicos de la aplicación se han abordado adecuadamente. Durante las pruebas, el enfoque se cambia a la ejecución de los aspectos específicos de la aplicación de la interacción del usuario según se manifiestan mediante la sintaxis y la semántica de la interfaz. Además, las pruebas proporcionan una valoración final de la facilidad de uso.

20.4.1 Estrategia de prueba de la interfaz

La estrategia global para la prueba de la interfaz es 1) descubrir los errores relacionados con mecanismos específicos de la interfaz (por ejemplo, errores en la ejecución adecuada de un vínculo de menú o la forma en que los datos ingresan en un formulario), y 2) descubrir los errores en la forma en que la interfaz implementa la semántica de navegación, la funcionalidad de la WebApp o el despliegue de contenido. El cumplimiento de esta estrategia requiere lograr varios objetivos:

CONSEJO

Con excepción de especificidades orientados a la WebApp, la estrategia de la interfaz anotada aquí es aplicable a todos los tipos de software cliente/servidor.

- *Las características de la interfaz se prueban para asegurar que las reglas del diseño, la estética y el contenido visual relacionado están a disposición del usuario sin error alguno.* Las características incluyen tipo de fuentes, uso de color, marcos, imágenes, bordes, tablas y elementos relacionados que se generan conforme procede la ejecución de la WebApp.
- *Los mecanismos individuales de la interfaz se prueban en una forma análoga a la prueba unitaria.* Por ejemplo, las pruebas están diseñadas para ejercitar todas las formas, creación de guiones en el lado del cliente, HTML dinámico, guiones CGI, clasificación por niveles de contenido y mecanismos de interfaz específicos de la aplicación (por ejemplo, un carrito de compras para una aplicación de comercio electrónico). En muchos casos, las pruebas se pueden enfocar exclusivamente en uno de dichos mecanismos (la "unidad") para excluir otras características y funciones de la interfaz.
- *Cada mecanismo de la interfaz se prueba dentro del contexto de un caso de uso o USN (capítulo 19) para una categoría de usuario específica.* Este enfoque de

pruebas es análogo a las pruebas de integración (capítulo 13) en que las pruebas se llevan a cabo conforme los mecanismos de la interfaz se integran para permitir la ejecución de un caso de uso o una USN

- *La interfaz completa se prueba frente a los casos de uso y las USN seleccionadas para descubrir errores en su semántica.* Este enfoque de pruebas es análogo a las pruebas de validación (capítulo 13), ya que el propósito es demostrar conformidad con la semántica específica del caso de uso o la USN. En el curso de esta etapa se lleva a cabo una serie de pruebas de facilidad de uso.
- *La interfaz se prueba dentro de una diversidad de ambientes (por ejemplo, navegadores) para asegurar que será compatible.* En la actualidad, esta serie de pruebas también se puede considerar como parte de las pruebas de configuración.

20.4.2 Prueba de mecanismos de la interfaz

Cuando un usuario interactúa con una WebApp, la interacción ocurre por medio de uno o más mecanismos de la interfaz. En el párrafo siguiente se presenta un breve panorama de las consideraciones de prueba para cada mecanismo de la interfaz [SPL01].

CONSEJO

...o de los ...s externos ...nfiarse a lo ...a toda la vida ...WebApp. Parte ...una estrategia de ...e debe ser lo ...regular y ...o de los ...

Vínculos. Cada vínculo de navegación se prueba para asegurar que se alcance el objeto de contenido o función adecuado.[8] El ingeniero Web construye una lista de todos los vínculos asociados con la plantilla de la interfaz (por ejemplo, barras de menú, artículos índice) y luego ejecuta cada uno de manera individual. Además, se deben ejercitar los vínculos dentro de cada objeto de contenido para descubrir malas URL o vínculos hacia objetos de contenido o funciones impropios. Finalmente, se deben probar los vínculos con WebApps externas para tener precisión y también deben evaluarse para determinar el riesgo de que se volverán inválidos con el tiempo.

Formatos. En el ámbito microscópico las pruebas se realizan para garantizar que 1) las etiquetas identifican correctamente los campos dentro del formato y que los campos obligatorios están identificados visualmente para el usuario; 2) el servidor recibe toda la información contenida en el formato y ningún dato se pierde en la transmisión entre cliente y servidor; 3) se usan los valores por defecto adecuados cuando el usuario no selecciona de un menú desplegable o conjunto de botones, 4) las funciones del navegador (por ejemplo, la flecha "retroceso") no corrompen los datos ingresados en un formato; y 5) los guiones que realizan verificación de error en los datos ingresados funcionan de manera adecuada y ofrecen mensajes de error significativos.

En un nivel más dirigido, las pruebas deben garantizar que 1) los campos del formato tienen ancho y tipos de datos adecuados; 2) el formato establece salvaguardas apropiadas para evitar que el usuario ingrese cadenas de texto más largas que cier-

8 Dichas pruebas se pueden llevar a cabo como parte de pruebas o de la interfaz o de navegación

to máximo predefinido; 3) todas las opciones apropiadas para menús desplegables están especificadas y ordenadas en una forma significativa para el usuario final; 4) las características de "autollenado" del navegador no conducen a errores en la entrada de datos; 5) la tecla de tabulador (o alguna otra) inicia el movimiento adecuado entre campos de formato.

Las pruebas de creación de guiones en el lado del cliente y las pruebas asociadas con el HTML dinámico se deben repetir siempre que se libere una nueva versión de un navegador popular.

Creación de guiones en el lado del cliente. Las pruebas de caja negra se llevan a cabo para descubrir los errores en el procesamiento conforme se ejecuta el guión (por ejemplo, Javascript). Dichas pruebas usualmente se acoplan con pruebas de formatos, ya que la entrada del guión por lo general se deriva de los datos proporcionados como parte del procesamiento de los formatos. Se debe realizar una prueba de compatibilidad para garantizar que el lenguaje de guión elegido funcionará adecuadamente en la configuración ambiental que soporta la WebApp. Además de poner a prueba el guión mismo, Splaine y Jaskiel [SPL01] sugieren que "debe asegurarse que los estándares de su compañía [WebApp] establecen el lenguaje preferido y la versión del lenguaje de creación de guiones que se usará para la creación de guiones en el lado del cliente (y en el lado del servidor)".

HTML dinámico. Cada página Web que contenga HTML dinámico se ejecuta para garantizar que el despliegue dinámico es correcto. Además, se debe llevar a cabo una prueba de compatibilidad para garantizar que el HTML dinámico funciona adecuadamente en la(s) configuración(es) ambiental(es) que soporta la WebApp.

Ventanas pop-up.[9] Una serie de pruebas garantizan que 1) la *pop-up* está ubicada de manera adecuada y tiene un tamaño apropiado; 2) la *pop-up* no cubre la ventana original de la WebApp; 3) el diseño estético de la *pop-up* es consistente con el diseño estético de la interfaz; y 4) las barras de desplazamiento y otros mecanismos de control agregados a la *pop-up* funcionan, están ubicados adecuadamente y trabajan como se requiere.

Guiones CGI. Las pruebas de caja negra se dirigen centrándose en la integridad de los datos (conforme los datos pasan al guión CGI) y en el procesamiento del guión una vez que los datos validados se han recibido. Además, se pueden llevar a cabo pruebas de desempeño para asegurarse de que la configuración del lado del servidor se puede ajustar a las demandas de procesamiento de invocaciones múltiples de los guiones CGI [SPL01].

Clasificación por niveles del contenido. Las pruebas deben demostrar que la clasificación por niveles de los datos está actualizada, se despliega adecuadamente y se puede suspender sin error y volver a comenzar sin dificultad.

Cookies. Se requieren pruebas tanto del lado del servidor como del lado del cliente. En el lado del servidor, las pruebas deben garantizar que una *cookie* está construida de manera adecuada (contiene datos correctos) y se transmite de modo apropiado al

9 La utilización de las pop-up se ha extendido mucho y son uno de los principales motivos de irritación para muchos usuarios. Deben usarse juiciosamente o evitarlas por completo.

lado del cliente cuando se solicita contenido o funcionalidad específicos. Además, se prueba la propia persistencia de la cookie para garantizar que su fecha de expiración es correcta. En el lado del cliente, las pruebas determinan si la WebApp une adecuadamente las *cookies* existentes a una solicitud específica (enviada al servidor).

Mecanismos de la interfaz específicos a la aplicación. Las pruebas conforman una lista de verificación de funcionalidad y características que se definen mediante el mecanismo de la interfaz. Por ejemplo, Splaine y Jaskiel [SPL01] sugieren la siguiente lista de verificación para la funcionalidad carrito de compras definido para una aplicación de comercio electrónico:

- La frontera (capítulo 14) prueba los números mínimo y máximo de artículos que pueden colocarse en el carrito.
- Probar una solicitud de "verificación" para un carrito vacío
- Probar el borrado adecuado de un artículo del carrito.
- Determinar mediante prueba si el contenido del carrito se vacía con una compra.
- Determinar mediante prueba la persistencia del contenido del carrito de compras (esto se debe especificar como parte de los requisitos del cliente)
- Determinar mediante prueba si la WebApp puede recuperar el contenido del carrito en alguna fecha futura (suponiendo que no se ha realizado compra alguna) si el usuario solicita que el contenido se guarde

20.4.3 Prueba de la semántica de la interfaz

Una vez que cada mecanismo de la interfaz se ha probado de manera "unitaria", el enfoque de la prueba de la interfaz cambia para considerar la semántica de ésta. La prueba de la semántica de la interfaz "evalúa cuán bien el diseño se ocupa de los usuarios, ofrece dirección clara, entrega retroalimentación y mantiene consistencia de lenguaje y enfoque" [NGU01].

Una revisión exhaustiva del modelo de diseño de la interfaz puede ofrecer respuestas parciales a las preguntas implícitas en el párrafo anterior. Sin embargo, se debe probar cada escenario de caso de uso (para cada categoría de usuario) una vez implementada la WebApp. En esencia, un caso de uso se convierte en la entrada para el diseño de una secuencia de pruebas. La finalidad de la secuencia de pruebas es descubrir los errores que le impedirán al usuario lograr el objetivo asociado con el caso de uso.

Conforme se prueba cada caso de uso, el equipo de ingeniería Web mantiene una lista de verificación para asegurarse de que todo artículo del menú se ha ejercido al menos una vez, y que todo vínculo anidado dentro de un objeto de contenido ha sido empleado. El objetivo es determinar si la WebApp ofrece un efectivo manejo del error y recuperación.

Referencia Web

Una valiosa guía para probar la facilidad de uso se encuentra en www.nbrel.com/guides/design/199806/0615jef.html.

20.4.4 Prueba de la facilidad de uso

La *prueba de la facilidad de uso* es similar a la prueba de la semántica de la interfaz (sección 20.4.3) en el sentido de que también evalúa el grado en el cual los usuarios pueden interactuar efectivamente con la WebApp, así como el grado en el cual la WebApp guía las acciones de los usuarios, proporciona retroalimentación significativa y fortalece un enfoque de interacción consistente. Más que enfocarse fijamente en la semántica de algún objetivo interactivo, las revisiones y pruebas de la facilidad de uso se diseñan para determinar el grado en el cual la interfaz de la WebApp facilita la vida del usuario.[10]

Las pruebas de facilidad de uso puede diseñarlas un equipo de ingeniería Web-App, pero las pruebas mismas las llevan a cabo los usuarios finales. Se aplica la siguiente secuencia de pasos [SPL01].

1. Definir un conjunto de categorías de prueba de facilidad de uso e identificar las metas para cada una.
2. Diseñar pruebas que permitirán evaluar cada meta.
3. Seleccionar los participantes que dirigirán las pruebas.
4. Instrumentar la interacción de los participantes con la WebApp mientras se lleva a cabo la prueba.
5. Desarrollar un mecanismo para valorar la facilidad de uso de la WebApp

La prueba de la facilidad de uso puede llevarse a cabo en varios grados de abstracción: 1) se puede valorar la facilidad de uso de un mecanismo de la interfaz específico (por ejemplo, un formulario); 2) se puede evaluar la facilidad de uso de una página Web completa (abarcando mecanismos de la interfaz, objetos de datos y funciones relacionadas); o 3) se puede considerar la facilidad de uso de la WebApp completa.

El primer paso en la prueba de la facilidad de uso es identificar un conjunto de categorías de facilidad de uso y establecer objetivos de prueba para cada categoría. Los siguientes objetivos y categorías de prueba (escritos en forma de pregunta) ilustran este enfoque.[11]

¿Qué características de facilidad de uso se vuelven el foco de las pruebas, y qué objetivos específicos se abordan?

Interactividad: ¿los mecanismos de interacción (por ejemplo, menús desplegables, botones, punteros) son fáciles de entender y usar?

Plantilla: ¿los mecanismos de navegación, contenido y funciones están colocados en una forma que permiten al usuario encontrarlos rápidamente?

10 En este contexto se ha usado el término "amigable para el usuario". Desde luego, el problema es que la percepción de un usuario de lo que es una interfaz "amigable" puede ser radicalmente diferente de la de otras.

11 Para preguntas adicionales acerca de la facilidad de uso, véase "Facilidad de uso" en el capí

Legibilidad: ¿el texto está bien escrito y es comprensible?[12] ¿Las representaciones gráficas son fáciles de entender?

Estética: ¿la plantilla, el color, los caracteres y las características relacionadas conducen a un uso más sencillo? ¿Los usuarios "se sienten cómodos" con la apariencia y la percepción de la WebApp?

Características de despliegue: ¿la WebApp utiliza en forma óptima el tamaño y la resolución de la pantalla?

Sensibilidad del tiempo: ¿las características, funciones y contenido importantes pueden utilizarse o adquirirse de manera oportuna?

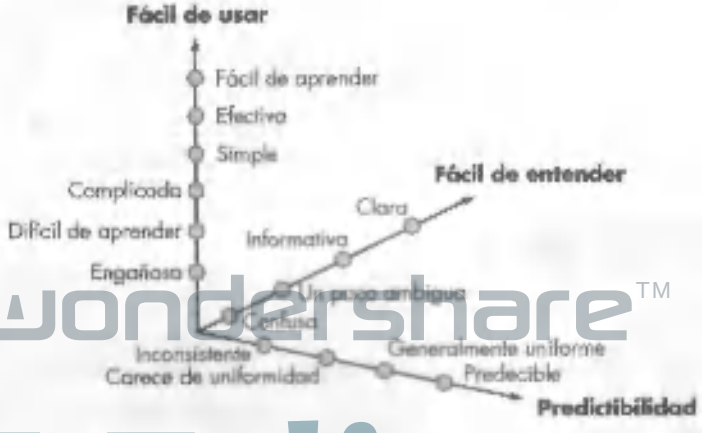
Personalización: ¿la WebApp se ajusta por sí misma a las necesidades específicas de las diferentes categorías de usuario o usuarios individuales?

Accesibilidad: ¿la WebApp es accesible a las personas con discapacidades?

Dentro de cada una de estas categorías se diseña una serie de pruebas. En algunos casos, la "prueba" puede ser una revisión visual de una página Web. En otros, se puede ejecutar de nuevo la prueba de semántica de la interfaz, pero en esta ocasión son más importantes las preocupaciones por la facilidad de uso.

Como ejemplo, considérese la valoración de la facilidad de uso para la interacción y los mecanismos de la interfaz. Constantine y Lockwood [CON03] sugieren que se revise y pruebe la facilidad de uso de la siguiente lista de características de la interfaz: animación, botones, color, control, diálogos, campos, formularios, marcos, gráficos, etiquetas, vínculos, menús, mensajes, navegación, páginas, selectores, texto y barras de herramientas. Conforme se valora cada característica, los usuarios que realizan la prueba la califican en una escala cualitativa. La figura 20.3 muestra un

...ura 20.3
...ación ...alitativa de la ...idad de uso.



12 Se pueden aprovechar el índice de Legibilidad FOG y otros para proporcionar una valoración cuantitativa de la legibilidad. Véase para más detalles http://developer.gnome.org/documents/usability/usability-readability.html

posible conjunto de "calificaciones" de valoración que pueden seleccionar los usuarios. Estas calificaciones se aplican a cada característica individualmente, a una página Web completa o a la WebApp como un todo.

20.4.5 Pruebas de compatibilidad

PUNTO CLAVE

Los WebApps se ejecutan dentro de una variedad de ambientes en el lado del cliente. El objetivo de las pruebas de compatibilidad es descubrir errores asociados con un ambiente específico (por ejemplo, navegador).

Las WebApps deben operar dentro de ambientes que difieren uno de otro. Diferentes computadoras, dispositivos de despliegue, sistemas operativos, navegadores y velocidad en las conexiones de red tienen una influencia significativa en las velocidades de procesamiento en el lado del cliente, la resolución de despliegue y las velocidad de conexión. Las variaciones en el sistema operativo pueden provocar conflictos en el procesamiento de la WebApp. En ocasiones, los diferentes navegadores producen resultados ligeramente diferentes, sin importar el grado de estandarización HTML dentro de la WebApp. Los *plug-in* requeridos pueden o no ser fácilmente accesibles para una configuración particular.

En algunos casos, los pequeños conflictos de compatibilidad no representan problemas significativos, pero en otros se pueden encontrar serios errores. Por ejemplo, las velocidades de descarga pueden volverse inaceptables; la falta de un *plug-in* requerido puede hacer que el contenido no sea disponible; la diferencias en cuanto al navegador pueden cambiar drásticamente la plantilla de página, los estilos de fuentes se pueden alterar y volverse ilegibles, o los formatos pueden estar organizados de manera inadecuada. La prueba de compatibilidad se esfuerza para descubrir estos problemas antes de que la WebApp esté en línea.

El primer paso en la prueba de compatibilidad es definir un conjunto de configuraciones de computadoras "encontrado comúnmente" en el lado del cliente y sus variantes. En esencia, se crea una estructura de árbol que identifica cada plataforma de computadora, los dispositivos de despliegue típicos, los sistemas operativos soportados en la plataforma, los navegadores disponibles, las probables velocidades de conexión a Internet e información similar. A continuación, el equipo de ingeniería Web produce una serie de pruebas de validación de compatibilidad, derivadas de las pruebas de la interfaz existentes, pruebas de navegación, pruebas de desempeño y pruebas de seguridad. La finalidad de estas pruebas es descubrir errores o problemas de ejecución que se pueden rastrear hasta las diferencias de configuración

HOGARSEGURO

Prueba de la WebApp

El escenario: Oficina de Doug

Los actores: Doug Miller (gerente del grupo de ingeniería del software de HogarSeguro) y Vinod Raman, miembro del equipo de ingeniería del software del producto.

La conversación:

Doug: ¿Qué piensas de la WebApp V0.0 de [illegible] electrónico de HogarSeguroInc.com?

...contratista hizo un buen trabajo. Sharon ...desarrollo de la empresa] me dicen que la ... mientras tú y yo conversamos.

...gustaría que tú y el resto del equipo hicieran ... prueba informal del sitio de comercio

... **muecas):** Creo que tendríamos ... una compañía de prueba para validar la ...odavía nos estamos matando al intentar que ...producto salga a la calle.

...aremos una empresa de prueba para las ...desempeño y seguridad, y nuestro ... ya está haciendo pruebas. Sólo creo que ...sta sería útil y, además, no gusta ...os costos en línea, así que..

...ira): ¿Qué buscas?

... estar seguro de que la interfaz y todo la ... son sólidos.

... que podemos comenzar con los casos ... cada una de las principales funciones de la

... **cerca de HogarSeguro**
... **el sistema HogarSeguro que**

Compre un sistema HogarSeguro
Obtenga soporte técnico

Doug: Bien. Pero sigue las rutas de navegación hasta su conclusión.

Vinod (observa los casos de uso en una computadora portátil): Sí, cuando eliges **Especifique el sistema HogarSeguro que necesita**, eso te llevará hasta

Seleccione componentes HogarSeguro

Obtenga recomendaciones de componentes de HogarSeguro

Podemos ejercitar la semántica de cada ruta.

Doug: Mientras estás en eso, verifica el contenido que aparece en cada nodo de navegación.

Vinod: Desde luego... y los elementos funcionales también. ¿Quién está probando la facilidad de uso?

Doug: Oh... la empresa de pruebas coordinará las pruebas de facilidad de uso. Hemos contratado una firma de investigación de mercado que reclutará 20 usuarios típicos para el estudio de facilidad de uso, pero si ustedes descubren algún conflicto al respecto...

Vinod: Ya sé: dénselos a ellos.

Doug: Gracias, Vinod.

20.5 PRUEBA AL NIVEL DE COMPONENTES

Las *pruebas al nivel de componentes*, también llamadas pruebas de función, se enfocan sobre un conjunto de pruebas que intentan descubrir errores en las funciones de la WebApp. Cada función WebApp es un módulo de software (implementado en algún lenguaje de programación o guiones) y se puede probar empleando las técnicas de caja negra (y, en algunos casos, de caja blanca) examinadas en el capítulo 14.

Los casos de prueba al nivel de componentes con frecuencia se alimentan con entrada al nivel de formularios. Una vez definidos los datos de los formularios, el usuario selecciona un botón u otro mecanismo de control para iniciar la ejecución. Son comunes los siguientes métodos de diseño de casos de prueba (capítulo 14):

- *Partición de equivalencia.* El dominio de entrada de la función se divide en categorías o clases de entrada a partir de las cuales se derivan los casos de prueba. La forma de entrada se valora para determinar qué clases de datos son relevantes para la función. Los casos de prueba para cada clase de entrada se derivan y ejecutan mientras otras clases de entrada se mantienen constantes. Por ejemplo, una aplicación de comercio electrónico puede implementar una función que calcule los cargos de embarque. Entre la diver-

sidad de información de embarque proporcionada mediante un formulario, está el código postal del usuario. Los casos de prueba se diseñan con la finalidad de descubrir errores en el procesamiento del código postal al especificar valores de código postal que puedan descubrir diferentes clases de errores (por ejemplo, un código postal incompleto, un código postal correcto, un código postal inexistente, un formato de código postal erróneo).

- *Análisis de valores límite*. Los datos de los formularios se prueban en sus límites. Por ejemplo, la función de cálculo de embarque señalada anteriormente solicita el número máximo de días requerido para la entrega del producto. En el formulario se anotan un mínimo de 2 días y un máximo de 14. Sin embargo, las pruebas de valor de límite pueden ingresar valores de 0, 1, 2, 13, 14 y 15 para determinar cómo reacciona la función frente a los datos en y fuera de los límites de las entradas válidas.[13]
- *Pruebas de ruta*. Si la complejidad lógica de la función es alta,[14] se puede emplear la prueba de ruta (método caja blanca de diseño de caso de prueba) para garantizar que se ha ejercitado toda ruta independiente en el programa

Además de estos métodos de diseño de casos de prueba, se utiliza una técnica llamada prueba de error forzado [NGU01] para producir casos de prueba que deliberadamente conducen los componentes de la WebApp hacia una condición de error. El propósito es descubrir los errores que ocurren durante el manejo de los errores (por ejemplo, mensajes de errores incorrectos o inexistentes, falla de la WebApp como consecuencia del error, salida errónea producida por entrada errónea, efectos colaterales relacionados con el procesamiento del componente).

Cada caso de prueba al nivel de componentes especifica todos los valores de entrada y la salida esperada que proporcionará el componente. La salida real producida como consecuencia de la prueba se registra para referencia futura durante el soporte y el mantenimiento.

En muchas situaciones la ejecución correcta de la función de una WebApp está ligada a una interfaz adecuada con una base de datos que puede ser externa a la WebApp. En consecuencia, la prueba de la base de datos se vuelve una parte integral del régimen de prueba de componentes. Hower [HOW97] examina esto cuando escribe:

> Los sitios Web alimentados con bases de datos pueden involucrar una interacción compleja entre navegadores Web, sistemas operativos, aplicaciones *plug-in*, protocolos de co-

13 En este caso, un mejor diseño de entrada puede eliminar errores potenciales. El máximo número de días se podría seleccionar de un menú desplegable, con lo que se evita que el usuario especifique una entrada fuera de los límites.

14 La complejidad lógica se puede determinar al calcular la complejidad ciclomática del algoritmo. Véase el capítulo 14 para detalles adicionales.

municación, servidores Web, bases de datos, programas [lenguaje de guión] ..., mejoras a la seguridad y cortafuegos.

Tal complejidad imposibilita probar todas las posibles dependencias y todo lo que podría ir mal con un sitio. El típico proyecto de desarrollo de un sitio Web también estará en un calendario agresivo, de modo que el mejor enfoque de prueba empleará análisis de riesgo para determinar dónde enfocar los esfuerzos de prueba. Los análisis de riesgo debe incluir consideración de cuánto coincidirá el ambiente de prueba con el ambiente de producción real... Otras consideraciones típicas en el análisis de riesgo incluyen.

- ¿Cuál funcionalidad en el sitio Web es más crucial para su propósito?
- ¿Cuáles áreas del sitio requieren la más dura interacción con la base de datos?
- ¿Cuáles aspectos de los CGI, applets, componentes ActiveX, etc., del sitio son los más complejos?
- ¿Qué tipos de problemas causaría la mayoría de las quejas o la peor publicidad?
- ¿Qué áreas del sitio serán las más populares?
- ¿Qué aspectos del sitio tienen los mayores riesgos de seguridad?

Cada uno de los asuntos relacionados con el riesgo que examina Hower deben considerarse cuando se diseñen casos de prueba para componentes WebApp y funciones de bases de datos relacionadas.

20.6 PRUEBAS DE NAVEGACIÓN

Un usuario viaja a través de una WebApp en gran medida como lo hace un visitante al caminar por una tienda o un museo. Existen muchas rutas que se pueden tomar, muchas paradas que se pueden realizar, muchas cosas que aprender y ver, actividades por iniciar y decisiones por tomar. Como ya se ha comentado, este proceso de navegación es predecible en el sentido en que todo visitante tiene un conjunto de objetivos cuando llega. Al mismo tiempo, el proceso de navegación puede ser impredecible porque el visitante, influido por algo que ve o aprende, puede elegir una ruta o iniciar una acción que no es típica para el objetivo original. El trabajo de probar la navegación es 1) garantizar que todos los mecanismos que permiten al usuario de la WebApp viajar a través de ella son funcionales, y 2) validar que cada unidad semántica de navegación (USN) pueda ser alcanzada por la categoría de usuario adecuada.

"No estamos perdidos. Enfrentamos un reto de ubicación."

John M. Ford

20.6.1 Prueba de la sintaxis de navegación

La primera fase de la prueba de navegación en realidad comienza durante la prueba de la interfaz. Los mecanismos de navegación se prueban para asegurar que cada

uno realiza la función que se busca. Splaine y Jaskiel [SPL01] sugieren que se debe probar cada uno de los mecanismos de navegación siguientes:

- *Vínculos de navegación*. Se deben probar los vínculos internos dentro de la WebApp, los vínculos externos hacia otras WebApps y las anclas dentro de una página Web específica para garantizar que se alcanzarán el contenido o la funcionalidad adecuados cuando el vínculo se elija.
- *Redirigir*. Dichos vínculos entran en juego cuando un usuario solicita una URL inexistente o selecciona un vínculo cuyo destino se ha removido o cuyo nombre ha cambiado. Se despliega un mensaje al usuario y la navegación se redirige hacia otra página (por ejemplo, la página de inicio). La redirección se debe probar al solicitar vínculos internos incorrectos o URL externas y valorar cómo maneja la WebApp dichas solicitudes.
- *Bookmarks*. Aunque los bookmarks son función del navegador, se debe probar la WebApp para asegurar que se puede extraer un título de página significativo cuando se cree el bookmark
- *Marcos y conjuntos de marcos*. Cada marco contiene el contenido de una página Web específica, un conjunto de marcos contiene múltiples marcos y permite el despliegue de múltiples páginas Web al mismo tiempo. Puesto que es posible anidar marcos y conjuntos de marcos uno dentro de otro, se deben probar dichos mecanismos de navegación y despliegue para un contenido correcto, plantilla y tamaño adecuados, desempeño de descarga y compatibilidad de navegador.
- *Mapas de sitio*. Las entradas se deben probar para garantizar que el vínculo lleva al usuario hacia el contenido o la funcionalidad adecuados.
- *Motores de búsqueda internos*. Las WebApps complejas usualmente contienen cientos o incluso miles de objetos de contenido. Un motor de búsqueda interno permite al usuario realizar una búsqueda por palabra clave dentro de la WebApp para encontrar el contenido necesario. La prueba del motor de búsqueda valida la precisión y qué tan completa es la búsqueda, las propiedades de manejo de errores del motor de búsqueda y las características de búsqueda avanzada (por ejemplo, el uso de operadores booleanos en el campo de búsqueda).

Algunas de estas pruebas pueden desarrollarse mediante herramientas automatizadas (por ejemplo, verificador de vínculos), mientras que otras se diseñan y ejecutan de manera manual. La finalidad de todo esto es asegurar que los errores en las mecánicas de navegación se encuentren antes de que la WebApp esté en línea.

20.6.2 Prueba de la semántica de navegación

En el capítulo 19 se definió una unidad semántica de navegación (USN) como "un conjunto de estructuras de información y navegación relacionadas que colaboran en

el cumplimiento de un subconjunto de requisitos de usuario relacionados" [CAC02]. Cada USN se define mediante un conjunto de rutas de navegación (llamadas "formas de navegar") que conectan nodos de navegación (por ejemplo, páginas Web, objetos de contenido o funcionalidad). Tomada como un todo, cada USN permite al usuario lograr requisitos específicos definidos por medio de uno o más casos de uso para una categoría de usuario. La prueba de navegación ejercita cada USN para asegurar que dichos requisitos son asequibles.

Conforme se prueba cada USN, el equipo de ingeniería Web debe responder las siguientes preguntas:

- ¿La USN se logra en su totalidad sin error?
- ¿Todo nodo de navegación (definido para una USN) es asequible dentro del contexto de las rutas de navegación que define la USN?
- Si la USN puede alcanzarse empleando más de una ruta de navegación, ¿se ha probado cada ruta relevante?
- Si la interfaz del usuario proporciona una guía para ayudar en la navegación, ¿las direcciones son correctas y comprensibles conforme se realiza la navegación?
- ¿Existe un mecanismo (distinto a la flecha "retroceso" del navegador) para regresar al nodo de navegación precedente y hacia el comienzo de la ruta de navegación?
- ¿Los mecanismos de navegación dentro de un gran nodo de navegación (es decir, una gran página Web) funcionan adecuadamente?
- Si una función se ha de ejecutar en un nodo y el usuario elige no proporcionar entrada, ¿se puede completar el resto de la USN?
- Si una función se ejecuta en un nodo y ocurre un error en el procesamiento de la función, ¿se puede completar la USN?
- ¿Existe una forma para descontinuar la navegación antes de que se hayan alcanzado todos los nodos, pero entonces regresar a donde se descontinuó la navegación y proceder desde ahí?
- ¿Todo nodo se alcanza desde el mapa de sitio? ¿Los nombres de los nodos son significativos para los usuarios finales?
- Si un nodo dentro de una USN se alcanza desde alguna fuente externa, ¿es posible proceder hacia el siguiente nodo en la ruta de navegación? ¿Es posible regresar al nodo previo en la ruta de navegación?
- ¿El usuario entiende su ubicación dentro de la arquitectura de contenido conforme se ejecuta la USN?

La prueba de navegación, como las pruebas de la interfaz y facilidad de uso, se debe dirigir en tantos participantes diferentes como sea posible. Las primeras etapas

de las pruebas las dirigen los ingenieros Web, pero etapas ulteriores las deben dirigir otros participantes del proyecto, un equipo de prueba independiente y, a final de cuentas, usuarios sin calificación técnica. La finalidad es ejercitar ampliamente la navegación de la WebApp.

20.7 PRUEBA DE LA CONFIGURACIÓN

La variabilidad y la inestabilidad de la configuración son factores importantes que hacen de la ingeniería Web un desafío. Hardware, sistemas operativos, navegadores, capacidad de almacenamiento, rapidez de comunicación de la red y una diversidad de otros factores del lado del cliente son difíciles de predecir para cada usuario. Además, la configuración para un usuario dado puede cambiar (por ejemplo, actualizaciones de sistema operativo, nuevo ISP y rapidez de conexión) con regularidad. El resultado puede ser, en el lado del cliente, un ambiente proclive a errores, tanto sutiles como significativos. La impresión que un usuario tenga de la WebApp y la forma en la que interactúa con ella puede diferir significativamente de la experiencia de otro usuario, si ambos no trabajan dentro de la misma configuración del lado del cliente.

La labor de probar la configuración no es ejercitar toda posible configuración del lado del cliente. Más bien, es probar un conjunto de probables configuraciones de los lados del cliente y del servidor para garantizar que la experiencia del usuario será la misma en todos ellos y para aislar errores que puedan ser específicos de una configuración particular.

20.7.1 Conflictos en el lado del servidor

En el lado del servidor, los casos de prueba de configuración se diseñan para verificar que la configuración de servidor proyectada (es decir: servidor WebApp, servidor de base de datos, sistemas operativos, software cortafuegos, aplicaciones concurrentes) pueden soportar la WebApp sin error. En esencia, la WebApp se instala dentro del ambiente del lado del servidor y se prueba con la intención de encontrar errores relacionados con la configuración.

Conforme se diseñan la pruebas de configuración del lado del servidor, el ingeniero Web debe considerar cada componente de la configuración del servidor. Entre las preguntas que es preciso plantear y responder durante la prueba de configuración del lado del servidor se encuentran:

¿Qué preguntas se deben plantear y responder conforme se lleva a cabo la prueba de configuración en el lado del servidor?

- ¿La WebApp es totalmente compatible con el sistema operativo del servidor?
- ¿Los archivos de sistema, directorios y datos de sistema relacionados se crean correctamente cuando la WebApp es operativa?
- ¿Las medidas de seguridad del sistema (por ejemplo, cortafuegos o encriptado) permiten a la WebApp ejecutarse y dar servicio a los usuarios sin interferencia o menoscabo del desempeño?

- ¿La WebApp se ha probado con la configuración de servidor distribuido[15] (si existe uno) que se haya elegido?
- ¿La WebApp está adecuadamente integrada con software de base de datos? ¿La WebApp es sensible a diferentes versiones del software de base de datos?
- ¿Los guiones de la WebApp en el lado del servidor se ejecutan adecuadamente?
- ¿Los errores del administrador del sistema se han examinado para ver su efecto sobre las operaciones de la WebApp?
- Si se usan servidores proxy, ¿las diferencias en sus configuraciones se han abordado con pruebas en el sitio?

20.7.2 Conflictos en el lado del cliente

En el lado del cliente las pruebas de configuración se centran principalmente en la compatibilidad de la WebApp con las configuraciones que contienen una o más permutaciones de los siguientes componentes [NGU01]:

- *Hardware*. CPU, memoria, almacenamiento y dispositivos de impresión
- *Sistemas operativos*. Linux, Macintosh, Microsoft Windows, un sistema operativo con base móvil
- *Software de navegación*. Internet Explorer, Mozilla/Netscape, Opera, Safari y otros.
- *Componentes de la interfaz del usuario*. ActiveX, applets Java y otros.
- *Plug-ins*. QuickTime, RealPlayer y muchos otros
- *Conectividad*. Cable, DSL, módem regular, T1.

Además de estos componentes, otras variables incluyen el software de red, las variaciones del ISP y aplicaciones que corren al mismo tiempo.

Al diseñar pruebas de configuración en el lado del cliente el equipo de ingeniería Web debe reducir el número de variables de configuración hacia un número manejable.[16] Por tanto, se valora cada categoría de usuario para determinar las probables configuraciones que se encontrarán dentro de la categoría. Además, se pueden utilizar los datos compartidos en el mercado industrial para predecir las combinaciones más probables de componentes. Entonces la WebApp se prueba dentro de estos ambientes.

15 Por ejemplo, se puede usar un servidor de aplicación y uno de base de datos por separado. La comunicación entre las dos máquinas ocurre por medio de una conexión de red.

16 Ejecutar pruebas en toda posible combinación de componentes de configuración consume demasiado tiempo.

20.8 PRUEBAS DE SEGURIDAD

La seguridad de las WebApps es una materia compleja que debe entenderse por completo antes de que se pueda lograr una prueba de seguridad efectiva.[17] Las WebApps y los ambientes en el lado del cliente y en el lado del servidor en los que son hospedadas representan un blanco atractivo para *hackers*, empleados descontentos, competidores deshonestos y cualquiera otro que desee robar información sensible, modificar contenido maliciosamente, disminuir el desempeño, deshabilitar la funcionalidad o poner en apuros a una persona, organización o negocio.

> "El Internet es un lugar riesgoso para llevar a cabo negocios o almacenar activos. Hackers, crackers, snoops, spoofers, ... ladrones, vándalos, lanzadores de virus y proveedores de programas delictivos se pasean libremente."
>
> **Dorothy y Peter Denning**

Las pruebas de seguridad están diseñadas para probar las vulnerabilidades en el ambiente del lado del cliente, las comunicaciones de red que ocurren mientras los datos pasan del cliente al servidor y de vuelta, y el ambiente del lado del servidor. Cada uno de estos dominios puede recibir ataques, y es labor de quien prueba la seguridad descubrir las debilidades que pueden explotar quienes tengan la intención de hacerlo.

En el lado del cliente, las vulnerabilidades con frecuencia se pueden rastrear hasta errores preexistentes en los navegadores, programas de correo electrónico o software de comunicación. Nguyen [NGU01] describe un hoyo de seguridad típico:

> Uno de los errores mencionados comúnmente es el desbordamiento del *buffer*. Esto permite que un código malicioso se ejecute en la máquina cliente. Por ejemplo, al ingresar una URL en un navegador que es mucho mayor que el tamaño del *buffer* alocado para la URL provocará un error de sobreescritura de memoria (desbordamiento de buffer) si el navegador no tiene código de detección de error para validar la longitud de la URL ingresada. Un hacker estacional puede explotar astutamente este bug al escribir una URL grande con código ejecutable que puede provocar que el navegador reviente o altere las opciones de seguridad (de mayor a menor) o, peor aún, corrompa los datos del usuario.

CONSEJO

Si la WebApp es crucial en el negocio, conserva datos sensibles o es un probable blanco de hackers, es buena idea subcontratar las pruebas de seguridad con una empresa especializada en esta labor.

Otra vulnerabilidad potencial en el lado del cliente es el acceso no autorizado a cookies colocadas dentro del navegador. Los sitios Web creados con intenciones maliciosas pueden adquirir información contenida dentro de cookies legítimas y usar esta información en formas que ponen en juego la privacidad del usuario, o peor, establecen el escenario para el robo de identidad.

Los datos comunicados entre el cliente y el servidor son vulnerables a la simulación (*spoofing*). Ésta ocurre cuando un extremo de la ruta de comunicación lo sub-

17 Los libros de Trivedi [TRE03], McClure y sus colegas [MCC03] y Garfinkel y Spafford [GAR02] ofrecen información útil acerca del tema.

vierte una entidad con intenciones maliciosas. Por ejemplo, a un usuario puede engañarlo un sitio Web malicioso que actúa como si fuese el legítimo servidor de la WebApp (idénticos en apariencia y percepción). La intención es robar contraseñas, información de propiedad o datos de crédito.

En el lado del servidor, las vulnerabilidades incluyen ataques de negación de servicio y guiones maliciosos que pueden pasar al lado del cliente o empleados para deshabilitar las operaciones del servidor. Además, se puede tener acceso, sin autorización, a bases de datos en el lado del servidor (robo de datos).

La protección contra éstas (y muchas otras) vulnerabilidades requiere implementar uno o más de los siguientes elementos de seguridad [NGU01]:

- Cortafuegos: mecanismo de filtrado —combinación de hardware y software— que examina cada paquete de información entrante para garantizar que llega de una fuente legítima y bloquea cualquier dato sospechoso.
- Autentificación: mecanismo de verificación que valida la identidad de todos los clientes y servidores, y permite que la comunicación ocurra sólo cuando ambos lados son verificados.
- Encriptado: mecanismo de codificación que protege los datos sensibles mediante su modificación en una forma que imposibilita la lectura de quienes tengan intenciones maliciosas. El encriptado se fortalece empleando certificados digitales que permiten al cliente verificar el destino al cual se transmiten los datos.
- Autorización: mecanismo de filtrado que permite el acceso al ambiente del cliente o el servidor sólo a aquellos individuos con códigos de autorización apropiados (por ejemplo, identificación del usuario y contraseña).

La finalidad de las pruebas de seguridad es exponer los hoyos en dichos elementos de seguridad que podrían explotar quienes tengan intenciones maliciosas. El diseño actual de las pruebas de seguridad requiere un conocimiento profundo de los trabajos internos de cada elemento de seguridad, así como una extensa comprensión de un amplio rango de tecnologías de red. En muchos casos, las pruebas de seguridad se subcontratan con firmas que se especializan en dichas tecnologías.

20.9 PRUEBAS DEL DESEMPEÑO

Nada es más frustrante que una WebApp que tarda minutos en cargar contenido cuando los sitios de la competencia descargan contenido similar en segundos. Nada es más agraviante que el intento de entrar en una WebApp y recibir un mensaje de "servidor ocupado", que se acompaña con la sugerencia de intentarlo más tarde. Nada es más desconcertante que una WebApp que responde instantáneamente en algunas situaciones, y luego, en otras ocasiones, parece irse a un estado de espera infinita. Todos estos hechos suceden en la Web todos los días, y todos están relacionados con el desempeño.

Las pruebas del desempeño se aplican para descubrir problemas de desempeño que se presentan debido a falta de recursos en el lado del servidor, ancho de banda de red inapropiado, capacidades inadecuadas de base de datos, defectuosas o débiles capacidades del sistema operativo, funcionalidad WebApp mal diseñada y otros conflictos de hardware o software que pueden conducir a un pobre desempeño cliente-servidor. La finalidad es doble: 1) comprender cómo responde el sistema a la carga (es decir, número de usuarios, número de transacciones o volumen de datos global), y 2) recolectar métricas que conducirán a modificaciones de diseño para mejorar el desempeño.

20.9.1 Objetivos de las pruebas del desempeño

Las pruebas del desempeño se diseñan con el fin de simular situaciones de carga del mundo real. Conforme crece el número de usuarios simultáneos de la WebApp, o aumenta el número de transacciones en línea, o se incrementa la cantidad de datos (descargados o cargados), las pruebas de desempeño ayudarán a responder las siguientes preguntas:

Algunos aspectos del desempeño de la WebApp, al menos como los perciben los usuarios finales, son difíciles de probar, incluso la carga de red, las variaciones del hardware que establece una interfaz con la red y tópicos similares.

- ¿El tiempo de respuesta del servidor se reduce hasta un punto donde es apreciable e inaceptable?
- ¿En qué punto (en términos de usuarios, transacciones o carga de datos) el desempeño se vuelve inaceptable?
- ¿Qué componentes del sistema son responsables de la reducción del desempeño?
- ¿Cuál es el tiempo de respuesta promedio para los usuarios en una variedad de condiciones de carga?
- ¿La reducción del desempeño tiene impacto sobre la seguridad del sistema?
- ¿La confiabilidad o la precisión de la WebApp se afectan conforme crece la carga del sistema?
- ¿Qué ocurre cuando se aplican cargas que rebasan la capacidad máxima del servidor?

Para desarrollar respuestas a estas preguntas, se dirigen dos pruebas de desempeño diferentes:

- Prueba de carga: la carga en el mundo real se prueba en una diversidad de niveles de carga y en una variedad de combinaciones.
- Prueba de tensión: la carga se aumenta hasta el punto de ruptura para determinar cuánta capacidad puede manejar el ambiente de la WebApp.

Cada una de estas estrategias de prueba se considera en las secciones siguientes

20.9.2 Pruebas de carga

El objetivo de las pruebas de carga es determinar cómo la WebApp y su ambiente del lado del servidor responderán a varias condiciones de carga. Conforme proceden las pruebas, las permutaciones a las siguientes variables definen un conjunto de condiciones de prueba:

N, el número de usuarios concurrentes
T, el número de transacciones en línea por usuario por unidad de tiempo
D, la carga de datos procesada por el servidor por transacción

En cada caso, estas variables se definen dentro de los límites operativos normales del sistema. Conforme se corre cada condición de prueba, se recopilan una o más de las siguientes medidas: la respuesta de usuario promedio, el tiempo promedio para descargar una unidad de datos estandarizada o el tiempo promedio para procesar una transacción. El equipo de ingeniería Web examina estas medidas para determinar si una disminución precipitada en el desempeño se puede rastrear hasta una combinación específica de *N*, *T* y *D*.

La prueba de carga también se aplica para valorar la velocidad de conexión recomendada para los usuarios de la WebApp. La cantidad de información global procesada en una unidad de tiempo, P, se calcula de la forma siguiente:

$$P = N \times T \times D$$

Como ejemplo, considérese un popular sitio de noticias deportivas. En un momento dado, 20 000 usuarios concurrentes realizan una solicitud (una transacción, T) una vez cada dos minutos en promedio. Cada transacción requiere que la WebApp descargue un nuevo artículo que promedia 3 Kbytes de longitud. En consecuencia, la cantidad de información procesada en una unidad de tiempo se puede calcular como:

$$P = [20\,000 \times 0.5 \times 3\ Kb]/60 = 500 \text{ Kbytes/seg}$$
$$= 4 \text{ megabits por segundo}$$

Por lo tanto, la conexión de red para el servidor tendría que soportar esta tasa de datos y se debería probar para garantizar que la tiene.

20.9.3 Pruebas de tensión

La prueba de tensión (capítulo 13) es una continuación de la prueba de carga, pero en esta instancia las variables *N*, *T* y *D* se fuerzan para alcanzar y luego superar los límites operativos. La finalidad de estas pruebas es responder cada una de las preguntas siguientes:

- ¿El sistema se degrada "gentilmente" o el servidor se desconecta cuando se rebasa su capacidad?
- ¿El software del servidor genera mensajes de "servidor no disponible"? De manera más general: ¿a los usuarios se les advierte que no pueden alcanzar el servidor?

- ¿El servidor pone en cola las solicitudes de recursos y vacía la cola una vez que la capacidad demanda disminución?
- ¿Las transacciones se pierden conforme se rebasa la capacidad?
- ¿La integridad de los datos se afecta cuando se rebasa la capacidad?
- ¿Qué valores de N, T y D fuerzan el fallo del ambiente del servidor? ¿Cómo se manifiesta la falla en sí misma? ¿Las notificaciones automáticas se envían al equipo de soporte técnico en el sitio del servidor?
- Si el sistema falla, ¿cuánto tardará en estar en línea de nuevo?
- ¿Ciertas funciones de la WebApp (por ejemplo, calcular funcionalidad intensa capacidades de flujo de datos) se descontinúan cuando la capacidad alcanza el nivel de 80 o 90 por ciento?

A una variación de la prueba de tensión a veces se le refiere como prueba pico/*rebote* [SPL01]. En este régimen de pruebas la carga se lleva hasta su punto máximo de capacidad, luego se baja rápidamente hasta condiciones de operación normales y luego se sube de nuevo al pico. Al rebotar la carga del sistema un examinador puede determinar cuán bien el servidor puede poner en orden los recursos para satisfacer la demanda muy elevada y luego liberarlos cuando reaparecen las condiciones normales (de modo que estén listas para el siguiente pico).

HERRAMIENTAS DE SOFTWARE

Taxonomía de herramientas para pruebas de WebApp

En su artículo acerca de la prueba de los sistemas de comercio electrónico, Lam [LAM01] presenta una útil taxonomía de las herramientas automatizadas que tienen aplicabilidad directa en la realización de pruebas en un contexto de ingeniería Web. En cada categoría se indican herramientas representativas.[18]

Las herramientas de configuración y de gestión del contenido gestionan la versión y cambian el control de los objetos de contenido y de los componentes funcionales de la WebApp.

Herramienta(s) representativa(s).

En www.daveeaton.com/scm/CMTools.html se encuentra una extensa lista.

Las herramientas de desempeño de bases de datos miden, por ejemplo, el tiempo para realizar consultas a bases de datos seleccionadas. Estas herramientas facilitan el mejoramiento de la base de datos.

Herramienta(s) representativas(s): BMC Software (www.bmc.com)

Los programas depuradores *(debuggers)* son herramientas de programación comunes que encuentran y resuelven defectos de software en el código. Forman parte de la mayoría de los ambientes modernos de desarrollo de aplicaciones.

Herramienta(s) representativa(s):
Accelerated Technology (www.acceleratedtechnology.com)
IBM *VisualAge* Environment (www.ibm.com)
JDebug Tool (www.debugtools.com)

Las sistemas de gestión de defectos registran los defectos y rastrean su estatus y resolución. Algunos incluyen herramientas de reporte para ofrecer información de gestión acerca del defecto esporádico y las tasas de resolución de defectos.

Herramienta(s) representativa(s):
EXCEL *Quickbugs* (www.excelsoftware.com)

18 Las herramientas expuestas aquí sólo representan una muestra de esta categoría. Además, los nombres de las mismas son marcas registradas de las compañías mencionadas.

TRUETrack (www.mccabe com)
ClearQuest (www.rational.com)

...amientas de supervisión de red vigilan el ...tráfico de la red. Son útiles para identificar los ... de botella" de la red y probar los vínculos entre ... de entrada y de salida

...ienta(s) representativa(s):
...slac stanford.edu/xorg/nmtf/nmtf-tools.html se ...uentra una extensa lista.

...amientas de pruebas de regresión al... casos y datos de prueba y se pueden volver a ... para casos de prueba después de cambios de soft... sucesivos.

...nienta(s) representativa(s):
...are *QARun* (www compuware. ... products/qacenter/qarun)
VisualTest (www.rational.com)
... Software (www seque.com)

...erramientas de supervisión de sitio vigilan ...eño de un sitio, usualmente desde una perspecti... usuario. Úsense para compilar estadísticas como el ... de respuesta de extremo a extremo y la cantidad ...ación procesado en una unidad de tiempo, así ... para verificar periódicamente la disponibilidad de ...

...nienta(s) representativa(s):
... Systems (www keynote.com)

... herramientas de tensión ayudan a los desarro... a explorar el comportamiento del sistema en con... de niveles elevados de uso operativo y a encon... puntos de quiebre de un sistemo.

...nienta(s) representativa(s):
... Interactive (www.merc-int.com)
... Technologis (www scapatech.com)

... sistemas de supervisión de recursos son par... la mayoría de los sistemos operativos de los servido... res y del software de servidores Web; vigilan los recursos ta... les como el espacio de disco, el uso del CPU y la memoria

Herramienta(s) representativa(s):
Successful Hosting.com (www successfulhosting.com)
Quest Software Foglight (www quest.com)

Las herramientas de generación de datos de prueba auxilian a los usuarios en dicha tarea.

Herramienta(s) representativa(s):
En www.softwareqatest.com/qatweb1.html se encuentra una extensa lista.

Los comparadores de resultados de pruebas ayudan a comparar los resultados de un conjunto de pruebas con los de otro conjunto. Úsense para verificar que los cambios de código no han introducido cambios adversos en el comportamiento del sistema.

Herramienta(s) representativa(s):
En www aptest com/resources.html se encuentra una lista útil.

Los monitores de transacción miden el desempeño de los sistemas de procesamiento de transacciones de alto volumen

Herramienta(s) representativa(s):
QuotiumPro (www.quotium.com)
Software Research eValid (www soft com/eValid/index html)

Los herramientas de seguridad de sitios Web ayudan a detectar potenciales problemas de seguridad Con frecuencia pueden configurar herramientas de prueba y supervisión de la seguridad para que corran sobre una base calendarizada.

Herramienta(s) representativa(s):
En www timberlinetechnologies com/products/www.html se encuentra una extensa lista.

20.10 RESUMEN

La meta de probar las WebApp es ejercitar cada una de las muchas dimensiones de la calidad WebApp con la finalidad de encontrar errores o descubrir conflictos que pudieran conducir a fallas en la calidad. Las pruebas se centran en contenido, función, estructura, facilidad de uso, navegabilidad, desempeño, compatibilidad, interoperabilidad, capacidad y seguridad. Las pruebas también incorporan revisiones que ocurren conforme se diseña la WebApp.

La estrategia de prueba de la WebApp ejercita cada una de las dimensiones de calidad al examinar inicialmente "unidades" de contenido, funcionalidad o navegación. Una vez que las unidades individuales han sido validadas, el enfoque se cambia a

pruebas que ejerciten la WebApp como un todo. Esto se logra derivando muchas pruebas de las perspectivas de los usuarios y se alimentan con la información contenida en los casos de uso. Se desarrolla un plan de prueba de ingeniería Web y se identifican los pasos de prueba, los productos de trabajo (por ejemplo, casos de prueba) y los mecanismos para la evaluación de los resultados de prueba. El proceso de prueba abarca siete tipos diferentes de pruebas.

La prueba del contenido (y las revisiones) se centra en varias categorías de contenido. La finalidad es descubrir errores tanto semánticos como sintácticos que afecten la precisión del contenido o la forma en la que se presenta al usuario final. La prueba de la interfaz ejercita los mecanismos de interacción que permiten que un usuario se comunique con la WebApp y valida los aspectos estéticos de la interfaz. El objetivo es descubrir errores que resulten de mecanismos de interacción mal implementados, u omisiones, inconsistencias o ambigüedades en la semántica de la interfaz.

La prueba de navegación aplica casos de uso, derivados como parte de la actividad de análisis, en el diseño de casos de prueba que ejercitan cada uno de los escenarios de uso frente al diseño de navegación. Los mecanismos de navegación se prueban para garantizar que se identifican y corrigen los errores que impiden e completar un caso de uso. La prueba de componentes ejercita las unidades de contenido y funcionales dentro de la WebApp. Cada página Web encapsula contenido vínculos de navegación y elementos de procesamiento que forman una "unidad" dentro de la arquitectura de la WebApp. Se deben probar dichas unidades.

La prueba de configuración intenta descubrir los errores o los problemas de compatibilidad específicos de un ambiente particular de cliente o servidor. Entonces se llevan a cabo pruebas para descubrir los errores asociados con cada posible configuración. La prueba de la seguridad incorpora una serie de pruebas diseñadas para explotar las vulnerabilidades en la WebApp y en su ambiente. La finalidad es encontrar hoyos de seguridad. La prueba de desempeño abarca una serie de pruebas que se diseñan para valorar el tiempo de respuesta de la WebApp y la confiabilidad conforme aumenta la demanda en la capacidad de recursos en el lado del servidor.

Referencias

[BRO01] Brown, B., *Oracle9i Web Development*, McGraw-Hill, 2a. ed., 2001

[CAC02] Cachero, C. *et al* , "Conceptual Navigation Analysis: A Device and Platform Independent Navigation Specification", *Proc. 2nd Intl. Workshop on web-oriented technology*, junio de 2002, se puede descargar de www.dsic.upv.es/~west/iwwost02/papers cachero.pdf

[CON03] Constantine, L. y L. Lockwood, *Software for Use*, Addison-Wesley, 1999; véase también http://www foruse.com/.

[GAR02] Garfinkel, S. y G. Spafford, *Web Security, Privacy and Commerce*, O'Reilly & Associates, 2002.

[HOW97] Hower, Rick, "Beyond Broken Links", *Internet Systems*, 1997, disponible en http://www.dbmsmag.com/9707i03.html.

[LAM01] Lam, W., "Testing E-Commerce Systems: A Practical Guide", en *IEEE IT Pro*, marzo-abril de 2001, pp. 19-28.

[MCC03] McClure, S., S. Shah y S. Shah, *Web Hacking: Attacks and Defense*, Addison-Wesley, 2003.
[MIL00] Miller, E., "WebSite Testing", 2000, disponible en http://www.soft.com/eValid/Technology/White.Papers/website.testing.html
[NGU00] Nguyen, H., "Testing Web-Based Applications", en *Software Testing and Quality Engineering*, mayo-junio de 2000, disponible en http://www.stqemagazine.com.
[NGU01] Nguyen, H., *Testing Applications on the Web*, Wiley, 2001.
[SCE02] Sceppa, D., *Microsoft ADO.NET*, Microsoft Press, 2002.
[SPL01] Splaine, S. y S. Jaskiel, *The Web Testing Handbook*, STQE Publishing, 2001
[TRE03] Trivedi, R., *Professional Web Services Security*, Wrox Press, 2003
[WAL03] Wallace, D., I. Raggett y J. Aufgang, *Extreme Programming for Web Projects*, Addison-Wesley, 2003.

PROBLEMAS Y PUNTOS A CONSIDERAR

20.1. ¿Existen algunas situaciones en las cuales las pruebas de la WebApp deban ignorarse por completo?

20.2. Con argumentos propios, coméntense los objetivos de la realización de pruebas en el contexto de la ingeniería Web.

20.3. La compatibilidad es una importante dimensión de calidad. ¿Qué debe ponerse a prueba para garantizar que existe compatibilidad para una WebApp?

20.4. ¿Qué errores tienden a ser más serios: los errores del lado del cliente o los errores del lado del servidor? ¿Por qué?

20.5. ¿Qué elementos de la WebApp pueden "probarse de manera unitaria"? ¿Qué tipos de pruebas se deben llevar a cabo sólo después de que están integrados los elementos de la WebApp?

20.6. ¿Siempre es necesario desarrollar un plan de prueba escrito de manera formal? Explíquese la respuesta.

20.7. ¿Es justo decir que la estrategia global de pruebas de la WebApp comienza con los elementos visibles para el usuario y se mueve hacia los elementos de tecnología? ¿Existen excepciones a esta estrategia?

20.8. ¿La prueba del contenido *realmente* es una prueba en el sentido convencional del término? Explicar la respuesta.

20.9. Describir los pasos asociados con la prueba de las bases de datos para una WebApp. ¿La prueba de las bases de datos es una actividad predominantemente del lado del cliente o del lado del servidor?

20.10. ¿Cuál es la diferencia entre las pruebas que están asociadas con los mecanismos de la interfaz y las pruebas que abordan la semántica de la interfaz?

20.11. Suponer que se está en el desarrollo de una farmacia en línea (FarmaciadelaEsquina.com) que intenta satisfacer los deseos de pacientes geriátricos. La farmacia proporciona las funciones usuales, pero también mantiene una base de datos para cada cliente de modo que puede ofrecer información de medicamentos y advertencias de potenciales interacciones respecto de medicamentos. Comentar algunas pruebas especiales de facilidad de uso para esta WebApp.

20.12. Suponer que se ha implementado una función de verificación de interacción de medicamentos para FarmaciadelaEsquina.com (problema 20.11). Examine los tipos pruebas al nivel de componente que tendrían que llevarse a cabo para garantizar que esta función trabaja de manera adecuada. [Nota: Se tendría que usar una base de datos para implementar esta función.]

20.13. ¿Cuál es la diferencia entre probar la sintaxis de navegación y probar la semántica de navegación?

20.14. ¿Es posible probar cada configuración que probablemente encuentre la WebApp en el lado del servidor? ¿En el lado del cliente? Si no, ¿cómo un ingeniero Web selecciona un conjunto significativo de pruebas de configuración?

20.15. ¿Cuál es el objetivo de las pruebas de seguridad? ¿Quién lleva a cabo esta actividad de prueba?

20.16. FarmaciadelaEsquina.com (problema 20.11) se ha vuelto exitosa y el número de usuarios ha aumentado de manera significativa en los primeros dos meses de operación. Dibuje una gráfica que muestre el tiempo de respuesta probable como función del número de usuarios para un conjunto fijo de recursos en el lado del servidor. Rotule la gráfica para indicar puntos de interés en la "curva de respuesta".

20.17. En respuesta a su éxito, FarmaciadelaEquina.com (problema 20.11) ha implementado un servicio especial exclusivamente para manejar el resurtido de recetas. En promedio, 1 000 usuarios concurrentes giran una solicitud de resurtido una vez cada dos minutos. La WebApp descarga un bloque de datos de 500 bytes como respuesta. ¿Cuál es la cantidad de información aproximada que se necesita procesar por una unidad de tiempo para este servidor, en megabytes por segundo?

20.18. ¿Cuál es la diferencia entre prueba de carga y prueba de tensión?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La literatura para las pruebas de WebApps todavía está en evolución. Los libros de Ash (*The Web Testing Companion*, Wiley, 2003), Dustin y sus colegas (*Quality Web Systems*, Addison-Wesley, 2002), Nguyen [NGU01] y Splaine y Jaskiel [SPL01] se encuentran entre los que, publicados hasta la fecha, presentan los tratamientos más completos del tema. Mosley (*Client-Server Software Testing on the Desktop and the Web*, Prentice-Hall, 1999) aborda asuntos de prueba tanto en el lado del cliente como en el lado del servidor.

Stottlemeyer (*Automated Web Testing Toolkit*, Wiley, 2001) presenta información útil acerca de los métodos y estrategias en la prueba de las WebApps, así como un valioso análisis de las herramientas de prueba automatizadas. Graham y sus colegas (*Software Test Automation*, Addison-Wesley, 1999) presentan material adicional acerca de las herramientas automatizadas.

Nguyen y sus colegas (*Testing Applications for the Web*, segunda edición, Wiley, 2003) desarrollaron una gran actualización [NGU01] y ofrecen una guía única para probar aplicaciones móviles. Aunque Microsoft (*Performance Testing Microsoft .NET Web Applications*, Microsoft Press, 2002) se enfoca predominantemente en su ambiente .NET, sus comentarios acerca de las pruebas de desempeño pueden ser útiles para cualquier interesado en la materia.

Splaine (*Testing Web Security*, Wiley, 2002), Klevinsky y sus colegas (*Hack I.T.: Security Through Penetration Testing*, Addison-Wesley, 2002), Chirillo (*Hack Attacks Revealed*, segunda edición, Wiley, 2003) y Skoudis (*Counter Hack*, Prentice-Hall, 2001) ofrecen mucha información útil para quienes deben diseñar pruebas de seguridad.

En Internet está disponible una gran variedad de fuentes de información acerca de las pruebas para ingeniería Web. En el sitio Web de SEPA se encuentra una lista actualizada de referencias en la World Wide Web:
http://www.mhhe.com/pressman

PARTE

Cuatro

GESTIÓN DE PROYECTOS DE SOFTWARE

En esta parte de *Ingeniería del software: un enfoque práctico* se consideran las técnicas de gestión necesarias para planear, organizar, supervisar y controlar los proyectos de software. En los capítulos que siguen se abordan las siguientes preguntas:

- ¿Cómo se deben gestionar el personal, el proceso y los problemas durante un proyecto de software?
- ¿Cómo pueden emplearse las métricas de software para gestionar un proyecto de software y el respectivo proceso?
- ¿Cómo se estiman el esfuerzo, el costo y la duración del proyecto?
- ¿Qué técnicas pueden aplicarse para evaluar formalmente los riesgos que pueden incidir en el éxito del proyecto?
- ¿Cómo selecciona un gestor de proyectos un conjunto de tareas de trabajo de ingeniería del software?
- ¿Cómo se crea el programa cronológico de un proyecto?
- ¿Qué es la gestión de calidad?
- ¿Por qué son tan importantes las revisiones técnicas formales?
- ¿Cómo se gestionan los cambios durante el desarrollo del software de computadora y después de entregarlo al cliente?

Una vez que se respondan estas preguntas se estará mejor preparado para gestionar proyectos de software en una forma que conducirá a la entrega puntual de un producto de alta calidad.

CAPÍTULO

21

CONCEPTOS DE GESTIÓN DE PROYECTOS

CONCEPTOS CLAVE

En el prefacio de su libro acerca de la gestión de proyectos de software, Meiler Page-Jones [PAG85] hace una afirmación a la que pueden sumarse muchos asesores en ingeniería del software:

> He visitado docenas de tiendas comerciales, tanto buenas como malas, y he observado registros de gerentes de procesamiento de datos, de nuevo tanto buenos como malos. Con mucha frecuencia he visto con horror como dichos gerentes luchaban inútilmente con proyectos de pesadilla, sufriendo por fechas límites imposibles o sistemas entregados que indignaron a sus usuarios y devoraron enormes cantidades de tiempo de mantenimiento

Lo que Page-Jones describe son síntomas que resultan de una serie de problemas de gestión y técnicos. Sin embargo, si se realizara una autopsia de cualquier proyecto, es muy probable que se encontrara un tema consistente: la gestión del proyecto fue débil.

En este capítulo, y en los seis siguientes, se consideran los conceptos clave que conducen a una gestión efectiva de proyectos de software. Este capítulo considera conceptos y principios básicos de gestión de proyectos de software. El capítulo 22 presenta métricas del proceso y del proyecto, la base para una efectiva toma de decisiones de gestión. En los capítulos 23 y 24 se analizan las técnicas con que se estiman, se define un calendario realista y se establece un plan efectivo para el proyecto. En el capítulo 25 se presentan las actividades de gestión que conducen a una efectiva supervisión, reducción y gestión del riesgo. Finalmente, en los capítulos 26 y 27 se consideran técnicas para asegurar la calidad conforme un proyecto se lleva a cabo y se gestionan los cambios a lo largo de la vida de una aplicación.

UN VISTAZO RÁPIDO

¿Qué es? Aunque muchos (en sus momentos más oscuros) toman la visión de "gestión" de Dilbert, sigue siendo una actividad muy necesaria cuando se construyen sistemas y productos basados en computadoras. La gestión de proyectos involucra la planificación, supervisión y control del personal, el proceso y los eventos que ocurren mientras el software evoluciona desde un concepto preliminar hasta una implementación operativa.

¿Quién lo hace? Todos "gestionan" en cierta medida, pero el ámbito de las actividades de gestión varía entre las personas involucradas en un proyecto de software. Un ingeniero de software gestiona sus actividades diarias, y planifica, supervisa y controla las labores técnicas. Los gestores de proyecto planifican, supervisan y controlan el trabajo de un equipo de ingenieros de software. Los gestores ejecutivos coordinan la relación entre el negocio y los profesionales del software.

¿Por qué es importante? La construcción de software de computadora es una empresa compleja, en particular si involucra a mucha gente que trabaja durante un tiempo relativamente

...argo. Por esto los proyectos de software nece...itan ser gestionados.

¿Cuáles son los pasos? Comprender las cua...ro P: personal, producto, proceso y proyecto. El personal debe estar organizado para reali...zar el trabajo de software con efectividad. La ...omunicación con el cliente y otros participan...tes debe ocurrir de modo que sean comprensi...bles el ámbito y los requisitos del producto. Se ...ebe seleccionar un proceso adecuado para el personal y el producto. El proyecto se debe planificar para estimar el esfuerzo y el tiempo para cumplir las labores del trabajo: definir productos de trabajo, establecer puntos de control de calidad e identificar mecanismos para supervisar y controlar el trabajo definido al planificar.

¿Cuál es el producto obtenido? Cuando comienzan las actividades de gestión se produce un plan del proyecto. El plan define el proceso y las labores que se llevarán a cabo, el personal que hará el trabajo y los mecanismos para valorar los riesgos, controlar los cambios y evaluar la calidad.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Nunca se está completamente seguro de que el plan del proyecto es correcto hasta que haya entregado un producto de alta calidad a tiempo y dentro del presupuesto. Sin embargo, un gestor de proyecto hace lo correcto cuando alienta al personal de software a trabajar en conjunto como un equipo efectivo, y enfoca su atención tanto en las necesidades del cliente como en la calidad del producto.

21.1 El espectro de la gestión

La gestión eficaz de la gestión de proyectos de software se enfoca sobre las cuatro P: *personal*, *producto*, *proceso* y *proyecto*. El orden no es arbitrario. El gestor que olvida que el trabajo de ingeniería del software es una empresa intensamente humana nunca tendrá éxito en la gestión de proyectos. Un gestor que fracasa en alentar la comunicación amplia con los participantes en etapas tempranas de la evolución de un proyecto se arriesga a construir una solución elegante para el problema equivocado. El gestor que presta poca atención al proceso corre el riesgo de colocar métodos y herramientas técnicos competentes en el vacío. El gestor que se embarca sin un plan de proyecto sólido arriesga el éxito del producto.

21.1.1 El personal

La formación de personal de software motivado y altamente calificado se ha debatido desde los años 60 del siglo pasado (por ejemplo, [COU80], [WIT94], [DEM98]). De hecho, el "factor humano" es tan importante que el Software Engineering Institute ha desarrollado un modelo de madurez de la capacidad de gestión de personal (MMCGP) para "aumentar la rapidez con la cual las organizaciones de software acometen las aplicaciones cada vez más complejas al ayudar a atraer, aumentar, motivar, desplegar y retener el talento necesario para mejorar su capacidad de desarrollo de software" [CUR94].

El modelo de madurez de gestión de personal define las siguientes áreas clave prácticas para el personal de software: reclutamiento, selección, gestión del desem-

peño, entrenamiento, retribución, desarrollo de la carrera, diseño de la organización y el trabajo, y desarrollo de la cultura de equipo. Las organizaciones que logran altos niveles de madurez en el área de gestión de personal tienen una mayor probabilidad de implementar efectivas prácticas de ingeniería del software.

El MMCGP es compañero de la Integración del Modelo de Madurez de la Capacidad de Software (capítulo 2) que guía a las organizaciones en la creación de un proceso de software maduro. Más adelante en este capítulo se consideran tópicos asociados con la gestión del personal y la estructura de los proyectos de software

21.1.2 El producto

CONSEJO

En este contexto, el término producto se emplea para abarcar cualquier software que se construye o solicitud de otros. No sólo incluye productos de software estandarizado, sino también sistemas basados en computadoras, software anidado, Web-Apps y software de resolución de problemas (por ejemplo, programas para resolución de problemas de ingeniería y científicos).

Antes de planear un proyecto se deberían establecer los objetivos y el ámbito de producto, considerar soluciones alternativas e identificar las restricciones técnicas y de gestión. Sin esta información es imposible definir estimaciones razonables (y precisas) del costo, una valoración efectiva del riesgo, una división realista de las tareas del proyecto o un calendario de proyecto manejable que ofrezca una indicación fiable del progreso.

El desarrollador del software y el cliente se deben reunir para definir los objetivos y el ámbito del producto. En muchos casos, esta actividad comienza como parte de la ingeniería del sistema o de la ingeniería del proceso de negocio (capítulo 6) y continúa como el primer paso en la ingeniería de requisitos de software (capítulo 7). Los objetivos identifican las metas globales del producto (desde el punto de vista de cliente) sin considerar cómo se lograrán. El ámbito identifica los datos primarios, las funciones y los comportamientos que caracterizan al producto y, más importante, los intentos por enlazar tales características en una forma cuantitativa.

Una vez entendidos los objetivos y el ámbito del producto se consideran soluciones alternativas. Aunque se trata relativamente poco detalle, las alternativas posibilitan que los gestores y practicantes seleccionen un "mejor" enfoque, cumplan las restricciones que imponen las fechas límites de entrega, las restricciones presupuestarias, la disponibilidad del personal, las interfases técnicas y miles de factores más

21.1.3 El proceso

CONSEJO

Quienes se adhieren a la filosofía del proceso ágil (capítulo 4) argumentan que sus procesos son más magros que otros. Esto puede ser cierto, pero ellos todavía tienen un proceso, y la ingeniería de software ágil todavía requiere disciplina.

Un proceso de software (capítulos 2, 3 y 4) proporciona el marco de trabajo desde el cual se puede establecer un plan detallado para el desarrollo del software. Un pequeño número de actividades del marco de trabajo es aplicable a todos los proyectos de software, sin importar su tamaño o complejidad. Algunos conjuntos de tareas diferentes (tareas, hitos, productos de trabajo y puntos de control de calidad) permiten que las actividades del marco de trabajo se adapten a las características del proyecto de software, así como a los requisitos del equipo del proyecto. Finalmente, las actividades protectoras (como el control de calidad del software, la gestión de configuración del software y la medición) cubren el modelo del proceso. Las actividades protectoras son independientes de cualquier actividad del marco de trabajo y ocurren durante todo el proceso.

21.1.4 El proyecto

Los proyectos de software se realizan de manera planificada y controlada por una razón principal: es la única forma conocida de gestionar la complejidad. Incluso los esfuerzos continuarán. En 1998, los datos industriales indicaron que el 26 por ciento de los proyectos de software fracasaron por completo, y que el 46 por ciento rebasaron sus costos y y tiempos de entrega [REE99]. Aunque la tasa de éxito para los proyectos de software ha mejorado un poco, la tasa de fracaso de proyectos permanece más elevada de lo que debiera.[1]

> "Un proyecto es como un viaje por carretera. Algunos proyectos son simples y rutinarios, como conducir hacia la tienda a plena luz del día. Pero la mayoría de los proyectos que vale la pena realizar son más parecidos a conducir un camión en la carretera, en la montaña, de noche."
>
> **Cem Kaner, James Bach y Brat Pettichord**

Para evitar el fracaso del proyecto, un gestor de proyecto de software y los ingenieros de software que construyen el producto deben eludir un conjunto de señales de advertencia comunes, comprender los factores de éxito críticos que conducen a una buena gestión del proyecto y desarrollar un enfoque de sentido común para planificar, supervisar y controlar el proyecto. Cada uno de estos tópicos se tratan en la sección 21.5 y en los capítulos siguientes.

21.2 PERSONAL

En un estudio que publicó el IEEE [CUR88] se les preguntó a los vicepresidentes de ingeniería de tres grandes compañías tecnológicas cuál era el contribuyente más importante para un proyecto de software exitoso. Respondieron de la siguiente forma:

VP 1: Supongo que si tienes que escoger alguna cosa que sea la más importante en nuestro ambiente, yo diría que no son las herramientas que utilizamos, es la gente.

VP 2: El ingrediente más importante que fue exitoso en este proyecto fue el tener gente inteligente... en mi opinión, muy pocas cosas importan más. . La cosa más importante que puedes hacer por un proyecto es seleccionar al equipo.. El éxito de la organización de desarrollo de software está muy asociado con la habilidad de reclutar buen personal.

VP 3: La única regla que tengo en la gestión es asegurarme que tengo buen personal, verdadero buen personal, y que hago crecer al buen personal, y que proporciono un ambiente en el que el buen personal puede producir.

1 Dadas estas estadísticas, es razonable preguntar cómo el impacto de las computadoras continúa creciendo exponencialmente. El autor considera que parte de la respuesta es que un número sustancial de estos proyectos "fallidos" estuvo, en primer lugar, mal concebido. Los clientes pierden interés rápidamente (porque el producto solicitado no fue en realidad tan importante como lo que pensaron primero) y los proyectos se cancelan.

De hecho, éste es un testimonio convincente acerca de la importancia del personal en los procesos de ingeniería del software. Y sin embargo, todos, desde los vicepresidentes de ingeniería hasta el profesional más modesto, usualmente no prestan atención al personal. Los gestores argumentan (como lo hace el grupo anterior) que las personas son lo principal, pero sus acciones con frecuencia contradicen sus palabras. En esta sección se examinan los participantes en el proceso de software y la forma en que se les organiza para realizar una ingeniería de software efectiva.

21.2.1 Los participantes

El proceso de software (y cualquier proyecto de software) lo integran participantes que pueden clasificarse dentro en una de cinco categorías:

1. *Gestores ejecutivos*, que definen los aspectos del negocio que usualmente tienen una influencia significativa en el proyecto
2. *Gestores (técnicos) del proyecto*, quienes planifican, motivan, organizan y controlan a los profesionales que realizan el trabajo de software
3. *Profesionales*, quienes proporcionan las habilidades técnicas necesarias para realizar la ingeniería de un producto o aplicación.
4. *Clientes*, quienes especifican los requisitos para la ingeniería del software y otros elementos que tienen un interés mínimo en el resultado
5. *Usuarios finales*, quienes interactúan con el software una vez que se libera para su uso productivo.

Todo proyecto de software lo integran personas que se clasifican en esta taxonomía.[2] Para ser eficaz, el equipo de proyecto debe estar organizado en una forma que maximice las capacidades y habilidades de cada persona. Y esta es la labor del líder del equipo.

21.2.2 Líderes de equipo

La gestión del proyecto es una actividad intensamente humana; por tanto, los profesionales competentes con frecuencia no son buenos líderes de equipo. Simplemente no tienen la mezcla correcta de habilidades con el personal. Además, como Edgemon afirma: "Desafortunadamente, y con demasiada frecuencia, los individuos simplemente caen en un papel de gestor de proyecto y se vuelven gestores de proyecto accidentales" [EDG95].

En un excelente libro acerca del liderazgo técnico, Jerry Weinberg [WEI86] sugiere un modelo MOI de liderazgo:

Motivación. La habilidad para alentar (mediante "empuje o jalón") al personal técnico para que produzca según su mejor capacidad.

2 Cuando se desarrollan las aplicaciones Web (Parte 3 de este libro), en la creación de contenido pueden involucrarse otras personas no técnicas

Organización. La habilidad para adecuar los procesos existentes (o inventar unos nuevos) que permitirán que el concepto inicial sea traducido en un producto final.

Ideas o innovación. La habilidad para alentar a la gente a crear y sentir creativamente, incluso cuando deben trabajar dentro de los límites establecidos por un producto o aplicación de software particular.

Weinberg sugiere que los líderes de proyecto exitosos aplican un estilo de gestión de resolución de problemas. Esto es: un gestor de proyecto de software debe concentrarse en entender el problema que será resuelto, gestionar el flujo de ideas y, al mismo tiempo, hacer que todos los que forman el equipo conozcan (con palabras y, mucho más importante, con acciones) que la calidad es relevante y que no será comprometida.

> "En los términos más simples, un líder es aquel que sabe adónde quiere ir, y se levanta y va."
>
> John Erskine

Otra visión [EDG95] de las características que definen un gestor de proyecto eficiente resalta cuatro rasgos clave:

Resolución de problemas. Un gestor de software eficiente puede diagnosticar los conflictos técnicos y organizativos más relevantes, estructurar de manera sistemática una solución o motivar adecuadamente a otros profesionales para desarrollar la solución, aplicar en las nuevas situaciones las lecciones aprendidas en proyectos pasados, y mantenerse lo suficientemente flexible como para cambiar de dirección si los intentos iniciales en la solución del problema son infructuosos

Dotes de gestión. Un buen gestor de proyecto debe encabezarlo y dirigirlo. Debe tener la confianza de asumir el control cuando es necesario y la seguridad para permitir que los buenos profesionales técnicos sigan sus instintos.

Incentivos. Para optimizar la productividad de un equipo de proyecto, un gestor debe recompensar la iniciativa y los logros; además, demostrar con sus propias acciones que la toma de riesgos controlada no será penalizada.

Influencia y fomento de la cultura de equipo. Un gestor de proyecto eficaz debe ser capaz de "leer" a la gente; de entender las señales verbales y no verbales y reaccionar a las necesidades de la gente que las envía. El gestor debe mantener el control en situaciones de alta tensión emocional

21.2.3 El equipo de software

Existen casi tantas estructuras organizacionales de profesionales para el desarrollo de software como organizaciones que tiene el mismo fin. Para bien o para mal, la estructura organizacional no puede ser fácilmente modificada. Las preocupaciones acerca de las consecuencias prácticas y políticas del cambio organizacional no es-

tán dentro del ámbito de responsabilidad del gestor del proyecto de software. Sin embargo, la organización de la gente directamente involucrada en un proyecto de software está dentro del ámbito del gestor del proyecto.

> "No todo grupo es un equipo, y no todo equipo es eficiente."
>
> Glenn Parker

La "mejor" estructura de equipo depende del estilo de gestión de cada organización, del número de personas que integrarán el equipo y de sus grados de habilidad, así como de la dificultad global del problema. Mantei [MAN81] describe siete factores de proyecto que deberían considerarse cuando se planifica la estructura de los equipos de ingeniería del software:

¿Qué factores se deben considerar cuando se elige la estructura de un equipo de software?

- La dificultad del problema que se resolverá
- El "tamaño" del programa(s) resultante(s) en líneas de código o puntos de función (capítulo 22).
- El tiempo que el equipo estará junto (vida del equipo).
- El grado en el que el problema puede separarse en módulos
- La calidad y confiabilidad requeridos del sistema que se construirá.
- La rigidez de la fecha de entrega.
- El grado de sociabilidad (comunicación) que requiere el proyecto.

> "Si quiere ser cada vez mejor: sea competitivo. Si quiere ser exponencialmente mejor: sea cooperativo."
>
> Anónimo

Constantine [CON93] sugiere cuatro "paradigmas organizacionales" para los equipos de ingeniería del software:

¿Qué opciones se tienen cuando se define la estructura de un equipo de software?

1. Un *paradigma cerrado* estructura un equipo a lo largo de una jerarquía tradicional de autoridad. Estos equipos pueden trabajar mejor cuando producen software muy similar a los proyectos anteriores, pero será menos probable que sean innovadores cuando trabajen dentro del paradigma cerrado.
2. Un *paradigma aleatorio* estructura un equipo libremente y depende de la iniciativa individual de los miembros del equipo. Cuando se requieren innovación o adelantos tecnológicos, los equipos que siguen el paradigma aleatorio serán excelentes. Pero estos equipos pueden luchar cuando se requiere "desempeño ordenado".
3. Un *paradigma abierto* intenta estructurar un equipo en una forma que logre algunos de los controles asociados con el paradigma cerrado, pero también mucha de la innovación que ocurre cuando se aplica el paradigma aleatorio.

El trabajo se desarrolla en colaboración. La sólida comunicación y la toma de decisiones basada en el consenso son las marcas características de los equipos de paradigma abierto. Las estructuras de equipo de paradigma abierto se adecuan bien a la solución de problemas complejos, pero no pueden desempeñarse de manera tan eficiente como otros equipos.

4. Un *paradigma sincrónico* se apoya en la compartimentalización natural de un problema y organiza a los miembros del equipo para trabajar en partes del problema con poca comunicación activa entre ellos.

> "Trabajar con la gente es difícil, mas no imposible."
>
> **Peter Drucker**

Como nota histórica, una de las primeras organizaciones de los equipos de software fue una estructura de paradigma cerrado originalmente llamada *equipo programador jefe*. Esta estructura la propuso originalmente Harlan Mills y la describió Baker [BAK72]. El núcleo del equipo lo compone un *ingeniero ejecutivo* (el programador jefe), quien planifica, coordina y revisa todas las actividades técnicas del equipo; *personal técnico* (por lo general, de dos a cinco personas), quienes dirigen las actividades de análisis y desarrollo; y un *ingeniero de respaldo*, quien apoya al ingeniero ejecutivo en sus actividades y puede reemplazarlo con mínima pérdida en la continuidad del proyecto.

Al programador jefe pueden asistirlo uno o más *especialistas* (por ejemplo, expertos en telecomunicaciones, diseñador de bases de datos), *personal de apoyo* (por ejemplo, escritores técnicos, personal administrativo) y un *bibliotecario de software*.

Como contraparte a la estructura de equipo programador jefe, el paradigma aleatorio de Constantine [CON93] sugiere un equipo de software con independencia creativa cuyo enfoque para trabajar puede denominarse mejor como *anarquía innovadora*. Aunque se ha apelado al enfoque de libre espíritu para el trabajo de software, la canalización de la energía creativa hacia un equipo de alto rendimiento debe ser una meta central en una organización de ingeniería del software. Para lograr un equipo de alto rendimiento:

- Los miembros del equipo deben tenerse mutua confianza.
- La distribución de las habilidades debe adecuarse al problema.
- Tal vez los disidentes deban ser excluidos del equipo si ha de conservarse su cohesión.

Sin importar la organización del equipo, el objetivo de cualquier gerente de proyecto es apoyar la creación de un equipo que muestre cohesión. En su libro, *Peopleware*, DeMarco y Lister [DEM98] examinan este tópico:

¿Qué es un equipo "real"?

Tendemos a usar la palabra *equipo* con bastante libertad en el mundo de los negocios, y llamamos "equipo" a cualquier grupo de personas asignadas a trabajar juntas. Pero muchos de estos grupos no parecen equipos. No tienen una definición común de éxito o al-

> gún espíritu de equipo identificable. Lo que se ha perdido es un fenómeno que llamamos *cuajar (jell)*.
>
> Un equipo cuajado es un grupo de personas tan fuertemente unido que el todo es mayor que la suma de las partes...
>
> Una vez que un equipo comienza a cuajar, la probabilidad de éxito aumenta. El equipo puede volverse imparable, un monstruo destructivo para lograr el éxito. No necesita ser gestionado en la forma tradicional, y ciertamente no necesita ser motivado. Tiene ímpetu

DeMarco y Lister sostienen que los miembros de los equipos cuajados son significativamente más productivos y están más motivados que el promedio. Comparten una meta común, una cultura común y, en muchos casos, un "sentimiento elitista" que los hace únicos.

¿Por qué fracasan los equipos que buscan cuajar?

Pero no todos los equipos cuajan. De hecho, muchos equipos sufren de lo que Jackman [JAC98] denomina "toxicidad de equipo". Ella define cinco factores que "fomentan un ambiente de equipo potencialmente tóxico": 1) una atmósfera de trabajo frenética, 2) alta frustración que provoca fricción entre los miembros del equipo, 3) un proceso de software "fragmentado o pobremente coordinado", 4) una definición poco clara de los papeles del equipo de software, y 5) "continuas y repetidas exposiciones al fracaso".

Para evitar un ambiente de trabajo frenético, el gestor del proyecto debe tener la certeza de que el equipo tiene acceso a toda la información requerida para realizar el trabajo y que las metas y objetivos, una vez definidos, no deben modificarse a menos que sea absolutamente necesario. Un equipo de software puede evitar la frustración (y el estrés) si se le da tanta responsabilidad en la toma de decisiones como sea posible. Un proceso inadecuado (por ejemplo, tareas de trabajo innecesarias o abrumadoras o productos de trabajo mal elegidos) se puede evitar si se comprende el producto que se construirá y al personal que realiza el trabajo, y al permitir al equipo seleccionar su propio modelo de proceso. El equipo debe establecer por sí mismo mecanismos para su responsabilidad (revisiones técnicas formales y la programación por pares son excelentes formas de lograrlo) y definir una serie de enfoques correctivos cuando un miembro del equipo falle en su desempeño. Y, finalmente, la clave para evitar una atmósfera de fracaso es establecer técnicas basadas en el equipo para la realimentación y resolución de problemas.

> "Hacer o no hacer. No existe intentar."
>
> **Yoda, de *Star Wars***

Además de las cinco toxinas que describe Jackman, un equipo de software usualmente enfrenta los diferentes rasgos humanos de sus miembros. Algunos miembros del equipo son extrovertidos; otros, introvertidos. Algunas personas recopilan información intuitivamente; separan los conceptos amplios de los hechos disparatados

Otros procesan la información linealmente, reúnen y organizan detalles minuciosos de los datos proporcionados. Algunos miembros del equipo se sienten cómodos al tomar decisiones sólo cuando se presenta un ordenado argumento lógico. Otros son intuitivos, por lo que desean tomar decisiones con base en el "sentimiento". Algunos profesionales prefieren una planificación detallada que incluya tareas organizadas que les permitan lograr el cierre de algún elemento del proyecto. Otros prefieren un ambiente más espontáneo en el que los temas abiertos son bien vistos Algunos trabajan duro para hacer que las cosas estén listas mucho antes de una fecha límite, y por consiguiente evitan la tensión conforme la fecha se aproxime, mientras que otros se sienten vigorizados por la prisa de lograrlo en el último minuto del plazo Un examen detallado de la psicología de estos rasgos y las formas en las cuales el líder experimentado del equipo puede ayudar a la gente con rasgos opuestos para trabajar en conjunto está más allá del ámbito de este libro.[3] Sin embargo, es importante destacar que el reconocimiento de las diferencias humanas es el primer paso hacia la creación de equipos que cuajan.

21.2.4 Equipos ágiles

En años recientes se ha propuesto el desarrollo del software ágil (capítulo 4) como antídoto para muchos de los problemas que perjudican el trabajo de los proyectos de software. En síntesis, la filosofía ágil alienta la satisfacción del cliente y la temprana entrega incremental de software; pequeños equipos de trabajo enormemente motivados, métodos informales; mínimos productos de trabajo de ingeniería del software; y simplicidad global de desarrollo.

El pequeño equipo de trabajo enormemente motivado, también llamado *equipo ágil*, adopta muchas características de los equipos de proyecto de software exitosos tratados en la sección precedente y evitan muchas de las toxinas que crean problemas Sin embargo, el enfoque ágil subraya la competencia individual (miembros del equipo) en conjunción con la colaboración del grupo como factores de éxito cruciales para el equipo. Cockburn y Highsmith [COC01] destacan esto cuando escriben:

> Si el personal en el proyecto es lo suficientemente bueno, pueden usar casi cualquier proceso y lograr su cometido. Si no es lo suficientemente bueno, ningún proceso reparará su incapacidad· "persona mata proceso" es una forma de decir esto Sin embargo, la falta de apoyo del usuario y el ejecutivo pueden aniquilar un proyecto: "política mata persona". El apoyo inadecuado puede incluso evitar que el buen personal logre la tarea.

Para aprovechar en forma eficiente las competencias de cada miembro del equipo y fomentar la colaboración eficaz a lo largo de un proyecto de software, los equipos ágiles son *autoorganizados*. Un equipo autoorganizado no necesariamente man-

3 Una excelente introducción a estos temas, relatados por equipos de proyecto de software, se puede encontrar en [FER98].

tiene una sola estructura de equipo, sino que más bien aprovecha elementos de los paradigmas aleatorio, abierto y sincrónico de Constantine tratados en la sección 21.2.3.

> "La propiedad colectiva no es más que una particularización de la idea de que los productos deben ser atribuibles al equipo [ágil], no a los individuos que integraron el equipo."
>
> Jim Highsmith

PUNTO CLAVE

Un equipo ágil es un equipo autoorganizado que tiene autonomía para planear y tomar decisiones técnicas

Muchos modelos de proceso ágil (por ejemplo, Scrum) brindan al equipo ágil una autonomía significativa para realizar la gestión del proyecto y tomar las decisiones técnicas requeridas para cumplir el trabajo. La planificación se mantiene en el mínimo, y al equipo se le permite seleccionar su propio enfoque (por ejemplo, proceso, métodos, herramientas), sólo restringido por los requisitos del negocio y los estándares organizacionales. Conforme el proyecto avanza el equipo se autoorganiza para enfocar la competencia individual en una forma que sea más benéfica para el proyecto en un punto dado en el tiempo. Para lograrlo, un equipo ágil puede dirigir breves reuniones de equipo diarias para coordinar y sincronizar el trabajo que se debe lograr ese día.

Con base en la información obtenida durante estas reuniones, el equipo adapta su enfoque de forma tal que logra un incremento de trabajo. Conforme pasa cada día, la autoorganización continua y la colaboración mueven al equipo hacia la conclusión de un incremento de software.

21.2.5 Conflictos de coordinación y comunicación

Existen muchas razones por las cuales los proyectos de software se vuelven problemáticos. La escala de muchos esfuerzos de desarrollo es grande, lo que conduce a complejidad, confusión y dificultades significativas en la coordinación de los miembros del equipo. La incertidumbre es común, lo que genera una corriente continua de cambios que mueve gradualmente en una sola dirección al equipo del proyecto. La interoperabilidad se ha convertido en una característica clave de muchos sistemas. El nuevo software debe comunicarse con el anterior y adecuarse a las restricciones predefinidas que impone el sistema o producto.

Estas características del software moderno —escala, incertidumbre e interoperabilidad— son aspectos de la vida. Para lidiar con ellos en forma eficaz, un equipo de ingeniería de software debe establecer métodos eficientes para coordinar al personal que realiza el trabajo. Para lograrlo se deben establecer mecanismos para la comunicación formal e informal entre los miembros del equipo y entre múltiples equipos. La comunicación formal se logra por medio de "escritos, reuniones estructuradas y otros canales de comunicación relativamente no interactivos e impersonales [KRA95]. La comunicación informal es más personal. Los miembros de un equipo de software comparten ideas sobre una base *ad hoc*, piden ayuda cuando surgen problemas e interactúan unos con otros diariamente.

HOGARSEGURO

Estructura de equipo

El escenario: La oficina de Doug [illegible] el inicio del proyecto de software *HogarSe-*

[illegible]ras: Doug Miller (gerente del equipo de ingeniería del software *HogarSeguro*) y Vinod Raman, Jamie [illegible] y otros miembros del equipo de ingeniería de software del producto

[illegible]versación:

[illegible]: ¿Ustedes han tenido oportunidad de echar un [illegible] a la información preliminar de *HogarSeguro* que [illegible] para mercadotecnia?

[illegible] (afirma con la cabeza y mira a sus [illegible]ferros de equipo): Sí. Pero tenemos muchas [illegible]

[illegible]: Dejemos eso por el momento. Quiero hablar [illegible] de cómo vamos a estructurar el equipo, quién es [illegible]sable de-qué

[illegible]: Yo realmente estoy con la filosofía ágil, Doug [illegible] debemos ser un equipo autoorganizado

[illegible]: Estoy de acuerdo. Dados el ceñido periodo de [illegible] y algo de incertidumbre, y al hecho de que todos [illegible] somos competentes [risas], me parece que esa [illegible] es el que debemos ir.

Doug: Está bien por mí, pero ustedes conocen el procedimiento.

Jamie (sonríe y habla como si recitara algo): Tomamos decisiones tácticas, acerca de quién hace qué y cuándo, pero es nuestra responsabilidad tener el producto listo a tiempo.

Vinod: Y con calidad.

Doug: Exactamente. Pero recuerda que existen restricciones. Mercadotecnia define los incrementos de software que se producirán . . . con nuestra asesoría, desde luego.

Jamie: ¿Y?

Doug: Y, vamos a usar UML como nuestro enfoque de modelado.

Vinod: Pero mantén la documentación extraña en un mínimo absoluto.

Doug: ¿Quién va a ser mi contacto?

Jamie: Decidimos que Vinod será el líder técnico. Él tiene más experiencia, así que Vinod es tu contacto, pero siéntete en libertad de hablar con cualquiera de nosotros.

Doug (ríe): No te preocupes, lo haré

21.3 EL PRODUCTO

El gestor de un proyecto de software se enfrenta con un dilema desde el principio mismo de un proyecto de ingeniería del software. Se requieren estimaciones cuantitativas y un plan organizado, pero no se dispone de información sólida. Un análisis detallado de los requisitos de software proporcionaría la información necesaria para las estimaciones, pero, con frecuencia, el análisis toma semanas o meses en completarse. Peor aún, los requisitos pueden ser fluidos, y cambian regularmente conforme el proyecto avanza. Más todavía, ¡se necesita un plan "ahora"!

En consecuencia, se deben examinar el producto y el problema que se intenta resolver al inicio del proyecto. Como mínimo, se debe establecer y acotar el ámbito del producto.

21.3.1 Ámbito del software

La primera actividad de gestión de un proyecto de software es la determinación del *ámbito del software*. El ámbito se define al responder las siguientes preguntas:

Si no puede acotar una característica del software que intenta construir, anote la característica como un riesgo del proyecto (capítulo 25).

Contexto. ¿Cómo encaja el software que se desarrollará en un sistema más grande, producto o contexto de negocios, y qué restricciones se imponen como resultado del contexto?

Objetivos de información. ¿Qué objetos de datos visibles al usuario (capítulo 8) se producen como resultado del software? ¿Qué objetos de datos se requieren de entrada?

Función y desempeño. ¿Qué funciones realiza el software para transformar los datos de entrada en salida? ¿Existen algunas características de desempeño especiales que deban abordarse?

El ámbito del proyecto de software no debe ser ambiguo ni incomprensible a nivel de gestión y técnico. Se debe acotar un enunciado del ámbito del software. Esto es, se establecen de manera explícita los datos cuantitativos (por ejemplo, número de usuarios simultáneos, tamaño de la lista de correo, tiempo de respuesta máximo permitido); se anotan las restricciones o limitaciones (por ejemplo, el costo del producto restringe el tamaño de la memoria) y se describen los factores que reducen riesgos (por ejemplo, los algoritmos deseados se comprenden bien y están disponibles en C++).

21.3.2 Descomposición del problema

La descomposición del problema, a veces llamada *partición* o *elaboración del problema*, es una actividad que se asienta en el núcleo del análisis de requisitos de software (capítulos 7 y 8). Durante la actividad de fijación del ámbito no se realiza intento alguno por descomponer completamente el problema. Más bien, la descomposición se aplica en dos grandes áreas: 1) la funcionalidad que debe entregarse y 2) el proceso que se empleará para entregarla.

El desarrollo de un plan de proyecto razonable requiere descomponer el problema. Esto se puede lograr empleando una lista de funciones o de casos de uso o, en el trabajo ágil, historias de usuario.

Los seres humanos tienden a aplicar una estrategia de divide y vencerás cuando enfrentan un problema complejo. Dicho con simplicidad, un problema complejo se divide en problemas menores que resultan más manejables. Ésta es la estrategia que se aplica cuando comienza la planificación del proyecto. Las funciones de software descritas al enunciar el ámbito, se evalúan y refinan para proporcionar más detalles antes del comienzo de la estimación (capítulo 23). Puesto que las estimaciones de costo y planificación temporal están funcionalmente orientadas, con frecuencia es útil cierto grado de descomposición.

Por ejemplo, considérese un proyecto que construirá un nuevo procesador de textos. Entre las características únicas del producto están la entrada continua mediante voz, así como por teclado, funciones muy sofisticadas de "edición automática y copia", capacidad de diseño de página, índice y tabla de contenido automáticos y otras. El gestor del proyecto primero debe establecer un enunciado del ámbito que acote estas características (así como otras funciones más usuales como la edición, la gestión de archivos, la producción de documentos y otras parecidas). Por ejemplo, ¿la entrada continua de voz requiere que el usuario del producto lo "entrene"? Esto

cíficamente, ¿qué capacidades proporcionará la característica de edición de copia? ¿Cuán sofisticada será la capacidad de diseño de página?

Conforme evoluciona el enunciado del ámbito ocurre naturalmente un primer nivel de partición. El equipo del proyecto aprende que el departamento de mercadotecnia ha hablado con los clientes potenciales y encontró que las siguientes funciones deben integrarse a la edición automática de copia: 1) comprobación ortográfica, 2) comprobación gramatical, 3) comprobación de referencias para documentos grandes (por ejemplo, ¿la referencia a una entrada bibliográfica se encuentra en la lista de entradas en la bibliografía?) y 4) validación de referencias de sección y capítulo para documentos grandes. Cada una de estas características representa una subfunción que debe implementarse en el software. Cada una todavía puede refinarse más si la descomposición simplifica la planificación.

21.4 El proceso

Las actividades del marco de trabajo (capítulo 2) que caracterizan al proceso de software son aplicables a todos los proyectos de software. El problema es seleccionar el modelo de proceso apropiado para que un equipo de proyecto someta al software a ingeniería.

El gestor del proyecto debe decidir cuál modelo de proceso es más adecuado para 1) los clientes que han solicitado el producto y el personal que hará el trabajo; 2) las características del producto mismo, y 3) el ambiente del proyecto en el que trabaja el equipo de software. Cuando se ha seleccionado un modelo de proceso, entonces el equipo define un plan de proyecto preliminar con base en el conjunto de actividades del marco de trabajo del proceso. Una vez que se establece el plan preliminar, comienza la descomposición del proceso. Esto es, se debe crear un plan completo, que refleje las tareas de trabajo requeridas para cubrir las actividades del marco de trabajo. Estas actividades se exploran brevemente en las secciones siguientes, y en el capítulo 24 se presenta una visión más detallada.

...n de una ...tareas" puede ...a herramien... ...icación de ...s automatiza... ...o 24). La ...ligo con esti... ...s de requisi... ...recursos, ...de inicio/fin y ...s pertinen... ...ces puede ...r esta red car... ...recursos para ...ento y ...r el proyecto.

21.4.1 Combinación del producto y el proceso

La planeación del proyecto comienza con la combinación del producto y el proceso. Cada función que el equipo de software someterá a ingeniería debe pasar a través del conjunto de actividades del marco de trabajo definidas para una organización de software. Supóngase que la organización ha adoptado el siguiente conjunto de actividades del marco de trabajo (capítulo 2): comunicaciones, planificación, modelado, construcción y despliegue.

Los miembros del equipo que trabajan en una función de producto le aplicarán cada una de las actividades del marco de trabajo. En esencia, se crea una matriz similar a la mostrada en la figura 21.1. Cada función de producto principal (en la figura se anotan funciones para el software del procesador de textos comentado anteriormente) se menciona en la columna izquierda. Las actividades del marco de tra-

FIGURA 21.1 Combinación del problema y el proceso.

PUNTO CLAVE

El marco de trabajo del proceso establece un esqueleto para la planificación del proyecto. Se adapta al ubicar un conjunto de tareas adecuadas para el proyecto

bajo se mencionan en la hilera superior. Las labores de trabajo de ingeniería del software (para cada actividad del marco de trabajo) se ingresarían en la hilera siguiente.[4] El trabajo del gestor del proyecto (y de otros miembros del equipo) consiste en estimar los requisitos de recursos para cada celda de la matriz, fechas de inicio y final para las tareas asociadas con cada celda, y los productos de trabajo que producirá cada tarea. Dichas actividades se consideran en el capítulo 24.

21.4.2 Descomposición del proceso

Un equipo de software debe tener un grado significativo de flexibilidad al elegir el modelo de proceso de software que sea mejor para el proyecto y las tareas de ingeniería del software que integren el modelo de proceso una vez elegido. Un proyecto relativamente pequeño similar a otros que se hayan realizado puede lograrse mejor si se utiliza el enfoque secuencial lineal. Si se imponen restricciones de tiempo muy ceñidas y el problema se puede compartimentalizar mucho, tal vez el modelo de desarrollo rápido de aplicaciones (DRA) sea la opción correcta. Si la fecha límite es tan ceñida que la funcionalidad completa no pueda alcanzarse, tal vez sea mejor una estrategia incremental. De igual modo, los proyectos con otras características (por ejemplo, requisitos inciertos, avances en la tecnología, clientes difíciles, significativo potencial de reutilización) conducirán a la selección de otros modelos de proceso.[5]

Una vez elegido el modelo de proceso, el marco de trabajo respectivo se adapta a él. En cualquier caso se puede aplicar el marco de trabajo genérico comentado

4 Se debe destacar que las tareas de trabajo tienen que adaptarse a las necesidades específicas del proyecto.

5 Recuérdese que las características del proyecto también tienen una fuerte influencia en la estructura del equipo de software (sección 21.2.3).

vamente: comunicación, planificación, modelado, construcción y despliegue. Funcionará para modelos lineales, iterativos e incrementales, así como evolutivos e incluso para modelos concurrentes o de ensamble de componentes. El marco de trabajo del proceso es invariable y sirve como base para todo el trabajo de software que realiza una organización de software.

Pero las tareas de trabajo real varían. La descomposición del proceso comienza cuando el gerente de proyecto pregunta: "¿Cómo logramos esta actividad del marco de trabajo?" Por ejemplo, un proyecto pequeño y relativamente simple puede requerir las siguientes tareas de trabajo para la actividad de comunicación:

1. Desarrollar una lista de conflictos que deben clarificarse.
2. Reunirse con los clientes para abordar los conflictos que deben clarificarse
3. Desarrollar en conjunto un enunciado del ámbito.
4. Revisar el enunciado del ámbito con todos los implicados.
5. Modificar el enunciado del ámbito según se requiera.

Estos sucesos pueden ocurrir en un periodo menor a 48 horas. Representan un proceso de descomposición adecuado para el proyecto pequeño y relativamente simple.

Ahora, considérese un proyecto más complejo, el cual tiene un ámbito más amplio y un impacto comercial más significativo Este proyecto puede requerir las siguientes tareas de trabajo para la actividad de comunicación:

1. Revisar la petición del cliente.
2. Planificar y programar una reunión formal con el cliente
3. Llevar a cabo investigaciones para especificar la solución propuesta y los enfoques existentes.
4. Preparar un "documento de trabajo" y una agenda para la reunión formal
5. Celebrar la reunión.
6. Desarrollar en conjunto miniprospectos que reflejen los datos, función y características de comportamiento del software. Alternativamente, se desarrollan casos de uso que describen al software desde el punto de vista del usuario.
7. Revisar cada miniprospecto o caso de uso para valorar su corrección, consistencia y falta de ambigüedad.
8. Ensamblar los miniprospectos en un documento más amplio
9. Revisar el documento más amplio o colección de casos de uso con todos los implicados.
10. Modificar el documento más amplio o casos de uso según se requiera

Ambos proyectos realizan la actividad del marco de trabajo que se llama "comunicación", pero el primer equipo de proyecto efectuó la mitad de las tareas de trabajo de ingeniería de software que realizó el segundo.

21.5 EL PROYECTO

La gestión de un proyecto de software exitoso requiere entender qué puede salir mal (de modo que sea factible evitar los problemas). En un excelente artículo acerca de proyectos de software, John Reel [REE99] define 10 señales que indican que un proyecto de sistemas de información está en peligro:

¿Cuáles son las señales de que un proyecto de software está en peligro?

1. El personal de software no entiende las necesidades de sus clientes.
2. El ámbito del producto está mal definido.
3. Los cambios se gestionan mal.
4. La tecnología elegida cambia
5. Las necesidades comerciales cambian [o están mal definidas].
6. Los plazos de entrega no son realistas
7. Los usuarios se resisten
8. Se pierde el patrocinio [o nunca se obtuvo de manera adecuada].
9. El equipo de proyecto carece de personal con las habilidades apropiadas.
10. Los gestores [y los profesionales] evitan las mejores prácticas y las lecciones aprendidas

Los profesionales industriales muy experimentados con frecuencia se refieren (medio frívolamente) a la regla 90-90 cuando estudian proyectos de software particularmente difíciles. El primer 90 por ciento de un sistema absorbe el 90 por ciento del esfuerzo y el tiempo asignados. El último 10 por ciento toma el otro 90 por ciento del esfuerzo y el tiempo asignados [ZAH94]. Las causas que conducen a la regla del 90-90 están contenidas en las señales anotadas en la lista precedente

> "No tenemos tiempo para detenernos por combustible, ya vamos tarde."
>
> M. Clara

¡Pero basta de negatividad! ¿Cómo actúa un gestor para evitar los problemas recién señalados? Reel [REE99] sugiere un enfoque de sentido común de cinco partes para proyectos de software:

1. *Comience con el pie derecho.* Esto se logra trabajando duro (muy duro) para entender el problema que será resuelto y entonces establecer objetivos y expectativas realistas para todos los que estarán involucrados en el proyecto. Esto se refuerza mediante la construcción del equipo correcto (sección 21.2) y al darle al equipo la autonomía, autoridad y tecnología necesarios para hacer el trabajo.
2. *Mantenga el ímpetu.* Muchos proyectos tienen un buen comienzo y luego lentamente se desintegran. Para mantener el ímpetu, el gestor del proyecto debe proporcionar incentivos para conservar los reveses del personal en un mini-

mo absoluto; el equipo debe resaltar la calidad en cada tarea que realiza, y los gestores ejecutivos debe hacer todo lo posible por mantenerse fuera del camino del equipo.[6]

3. *Rastree el progreso.* En un proyecto de software el progreso se rastrea conforme se elaboran los productos de trabajo (por ejemplo, modelos, código fuente, conjuntos de casos de prueba) y se aprueban (mediante revisiones técnicas formales) como parte de una actividad de aseguramiento de la calidad. Además, se pueden recopilar y aplicar procesos del software y medidas del proyecto (capítulo 22) para valorar el progreso contra los promedios establecidos por la organización que desarrolla software.
4. *Tome decisiones inteligentes.* En esencia, las decisiones del gestor del proyecto y del equipo de software deben encaminarse a "mantenerlo simple". Siempre que sea posible, decídase a emplear software comercial ya desarrollado o componentes de software existentes, decídase a evitar interfases personalizadas cuando estén disponibles enfoques estándar, decídase a identificar y luego evitar riesgos obvios, y decídase a asignar más tiempo que el que considere necesario a las tareas complejas o riesgosas (necesitara cada minuto).
5. *Realice un análisis de resultados.* Establezca un mecanismo consistente para extraer lecciones aprendidas por cada proyecto. Evalúe la planificación real y la prevista, recolecte y analice métricas de proyecto de software, obtenga realimentación de los miembros del equipo y de los clientes, y registre los hallazgos en forma escrita.

21.6 EL PRINCIPIO W^{5}HH

En un excelente artículo acerca de los procesos y proyectos de software, Barry Boehm [BOE96] establece: "Usted necesita un principio organizador que escale hacia abajo para proporcionar planes [de proyecto] simples para proyectos simples". Boehm sugiere un enfoque que aborde los objetivos del proyecto, los hitos y planificación, responsabilidades, gestión y enfoques técnicos y recursos requeridos. Boehm lo llama el principio W^{5}HH, en honor a una serie de preguntas que conducen a una definición de las características claves del proyecto y al plan de proyecto resultante:

¿Cómo se definen las características clave del proyecto?

¿Por qué se desarrolla el sistema? La respuesta a esta pregunta permite a todas las partes evaluar la validez de las razones del negocio para el trabajo de software. Dicho de otra forma: ¿el propósito del negocio justifica el gasto de personal, tiempo y dinero?

6 La implicación de este enunciado es que la burocracia se reduce al mínimo, las reuniones extrañas se eliminan y la adherencia dogmática a los procesos y reglas del proyecto se eliminan. El equipo debe ser autoorganizado y autónomo.

¿Qué se hará? La respuesta a esta pregunta establece el conjunto de tareas que se requerirá para el proyecto.

¿Cuándo se hará? La respuesta a esta pregunta ayuda al equipo a establecer una planificación del proyecto al identificar cuándo se realizarán las tareas del proyecto y cuándo se alcanzarán los objetivos.

¿Quién es el responsable de una función? Párrafos atrás se anotó que el papel y la responsabilidad de cada miembro del equipo de software deben estar definidos. La respuesta a esta pregunta ayuda a lograrlo.

¿Dónde están ubicados en la organización? No todos los papeles y responsabilidades residen en el equipo de software. El cliente, los usuarios y otros participantes también tienen responsabilidades.

¿Cómo se hará el trabajo desde los puntos de vista técnico y de gestión? Una vez establecido el ámbito del producto se debe definir una estrategia de gestión y técnica para el proyecto.

¿Cuánto de cada recurso se necesita? La respuesta a esta pregunta se deriva al desarrollar estimaciones (capítulo 23) con base en las respuestas a las preguntas anteriores.

El principio W[5]HH de Boehm se aplica sin importar el tamaño o la complejidad de un proyecto de software. Las preguntas anotadas proporcionan un excelente lineamiento de planificación para el gestor del proyecto y el equipo de software.

21.7 PRÁCTICAS CRÍTICAS

El Airlie Council[7] ha elaborado una lista de "prácticas críticas de software para la gestión basada en el desempeño". Dichas prácticas son "empleadas consistentemente por, y consideradas críticas por, proyectos de software muy exitosos y por organizaciones cuya 'línea base' de desempeño es mucho mejor que los promedios de la industria" [AIR99].

Las prácticas críticas[8] incluyen: gestión de proyecto basado en métricas (capítulo 22), costo empírico y estimación de la planificación (capítulos 23 y 24), seguimiento del valor ganado (capítulo 24), gestión del riesgo formal (capítulo 25), seguimiento de defectos frente a objetivos de calidad (capítulo 26) y gestión al tanto del personal (sección 21.2). Cada una de estas prácticas críticas se aborda a lo largo de la parte 4 de este libro.

7 El Airlie Council es un equipo de expertos en ingeniería de software que reclutó el Departamento de Defensa de los Estados Unidos de América para ayudar a desarrollar lineamientos para mejores prácticas en la gestión de proyectos de software y de ingeniería del software.

8 Aquí sólo se anotan las prácticas críticas asociadas con la "integridad del proyecto"

HERRAMIENTAS DE SOFTWARE

Herramientas de software para gestores de proyectos

Las "herramientas" mencionadas aquí son genéricas y se aplican a un amplio rango de actividades que realizan los gestores de proyecto. En los siguientes capítulos se consideran herramientas de gestión de proyectos específicas (por ejemplo, herramientas de planificación, estimación, de análisis de riesgo).

Herramientas representativas[9]

Software Program Manager's Network (www.spmn.com) ha desarrollado una herramienta simple llamada *Project Control Panel* que ofrece a los gestores de proyectos un aviso directo del estado del proyecto. La herramienta tiene "calibradores" muy parecidos a un tablero y está implementada con Microsoft Excel. Está disponible para descarga en http://www.spmn.com/products_software.html.

Gantthead.com ha desarrollado un conjunto de útiles *listas de verificación para gestores de proyecto* que se puede descargar de http://www.gantthead.com/.

Ittoolkit.com (www.ittoolkit.com) ofrece "una colección de guías de planificación, plantillas de proceso y hojas de cálculo inteligentes" disponibles en CD-ROM.

21.8 RESUMEN

La gestión de proyectos de software es una actividad protectora dentro de la ingeniería del software. Comienza antes de iniciar cualquier actividad técnica y continúa a lo largo de la definición, el desarrollo y el soporte del software de computadora.

Las cuatro P que tienen una influencia sustancial en la gestión de proyectos de software: personal, producto, proceso y proyecto. El personal debe estar organizado en equipos eficientes, motivados para hacer un trabajo de software de alta calidad y coordinados para lograr una comunicación eficaz. Los requisitos del producto se deben comunicar del cliente al desarrollador, ser divididos (descompuestos) en sus partes constitutivas y distribuirse para que trabaje el equipo de software. El proceso debe adaptarse al personal y al problema. Se selecciona un marco de trabajo de proceso común, se aplica un paradigma de ingeniería de software adecuado y se elige un conjunto de tareas de trabajo para llevar a cabo el trabajo. Finalmente, el proyecto debe estar organizado en una forma que permita triunfar al equipo de software.

El elemento central en todos los proyectos de software es el personal. Los ingenieros de software pueden organizarse en diferentes estructuras de equipo, que van desde las jerarquías de control tradicionales hasta los equipos de "paradigma abierto". Se pueden aplicar varias técnicas de coordinación y comunicación para apoyar el trabajo del equipo. En general, las revisiones formales y la comunicación informal de persona a persona son las más valiosas para los profesionales.

La actividad de gestión del proyecto abarca medidas y métricas, estimación y planificación, análisis de riesgos, seguimiento y control. Cada uno de estos tópicos se considera en los capítulos siguientes.

9 Las herramientas registradas aquí son una muestra de esta categoría. En la mayoría de los casos los nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

REFERENCIAS

[AIR99] Airlie Council, "Performance Based Management: The Program Manager's Guide Bas on the 16-Point Plan and Related Metrics", Draft Report, 8 de marzo, 1999.
[BAK72] Baker, F. T., "Chief Programmer Team Management of Production Programming", *IBM Systems Journal*, vol. 11, núm. 1, 1972, pp. 56-73.
[BOE96] Boehm, B , "Anchoring the Software Process", en *IEEE Software*, vol. 13, núm. 4, de 1996, pp 73-82
[COC01] Cockburn, A. y J. Highsmith, "Agile Software Development: The People Factor", en *IE Computer*, vol. 34, núm 11, noviembre de 2001, pp. 131-133.
[CON93] Constantine, L., "Work Organization: Paradigms for Project Management and Orga zation", en *CACM*, vol. 36, núm. 10, octubre de 1993, pp 34-43
[COU80] Cougar, J y R Zawacki, *Managing and Motivating Computer Personnel*, Wiley, 1980
[CUR88] Curtis, B *et al* , "A Field Study of the Software Design Process for Large Systems", *IEEE Trans. Software Engineering*, vol SE-31, núm 11, noviembre de 1988, pp. 1268-1287
[CUR94] Curtis, B. *et al.*, *People Management Capability Maturity Model*, Software Engineering I titute, 1994.
[DEM98] DeMarco, T. y T. Lister, *Peopleware*, 2a. ed., Dorset House, 1998.
[EDG95] Edgemon, J , "Right Stuff: How to Recognize It When Selecting a Project Manager", *Application Development Trends*, vol 2, núm. 5, mayo de 1995, pp. 37-42.
[FER98] Ferdinandi, P. L., "Facilitating Communication", en *IEEE Software*, septiembre de 19 pp. 92-96
[JAC98] Jackman, M., "Homeopathic Remedies for Team Toxicity", en *IEEE Software*, julio 1998, pp 43-45
[KRA95] Kraul, R y L Streeter, "Coordination in Software Development", en *CACM*, vol. 38, nú 3, marzo de 1995, pp. 69-81.
[MAN81] Mantei, M., "The Effect of Programming Team Structures on Programming Tasks", *CACM*, vol. 24, núm. 3, marzo de 1981, pp. 106-113.
[PAG85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, p vii
[REE99] Reel, J. S , "Critical Success Factors in Software Projects", en *IEEE Software*, mayo 1999, pp. 18-23.
[WEI86] Weinberg, G., *On Becoming a Technical Leader*, Dorset House, 1986.
[WIT94] Whitaker, K., *Managing Software Maniacs*, Wiley, 1994.
[ZAH94] Zahniser, R., "Timeboxing for Top Team Performance", en *Software Development*, zo de 1994, pp 35-38.

PROBLEMAS Y PUNTOS A CONSIDERAR

21.1. Con base en la información contenida en este capítulo y la experiencia propia, desar llar "10 mandamientos" para alentar el potencial de los ingenieros de software. Esto es, ela rar una lista de 10 lineamientos que conducirán al personal que desarrolla software a ejercer s potencial completo

21.2. El modelo de madurez de la capacidad de gestión de personal (MMCGP) del Software I gineering Institute realiza un estudio organizado de las "áreas prácticas clave" (APC) que va el buen personal de software. El instructor asignará una APC para analizar y resumir

21.3. Describir tres situaciones de la vida real en las cuales el cliente y el usuario final son mismo. Describir tres situaciones en las cuales son diferentes.

21.4. Las decisiones que toman los gestores ejecutivos pueden tener un impacto significat en la eficacia de un equipo de ingeniería del software. Proporcionar cinco ejemplos que ilust que esto es cierto.

21.5. Repasar el libro de Weinberg [WEI86] y escribir un resumen de dos o tres páginas de tópicos que deben considerarse al aplicar el modelo MOI

21.6. Usted ha sido nombrado gestor de proyecto dentro de una organización de sistemas de información. Su labor es construir una aplicación que sea bastante similar a otras que ha construido su equipo, aunque ésta es mayor y más compleja. El cliente ha documentado ampliamente los requisitos. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo(s) de proceso de software elegiría y por qué?

21.7. Usted ha sido nombrado gestor de proyecto en una pequeña compañía de productos de software. Su labor es construir un producto de avanzada que combine hardware de realidad virtual con software de última generación. Puesto que la competencia en el mercado del entretenimiento casero es intensa, existe una presión significativa para completar el trabajo. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo(s) de proceso de software elegiría y por qué?

21.8. Usted ha sido nombrado gestor de proyecto en una gran compañía de productos de software. Su labor es gestionar el desarrollo de la versión de siguiente generación de su software de procesamiento de textos ampliamente utilizado. Puesto que se deben generar nuevos ingresos, se han establecido y anunciado fechas límite precisas. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo(s) de proceso de software elegiría y por qué?

21.9. Usted ha sido nombrado gestor de proyecto de software para una compañía que atiende al mundo de la ingeniería genética. Su labor es gestionar el desarrollo de un nuevo producto de software que acelerará el ritmo de la clasificación de genes. El trabajo está orientado I+D, pero la meta es elaborar un producto dentro del siguiente año. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo(s) de proceso de software elegiría y por qué?

21.10. A usted se le pide desarrollar una pequeña aplicación que analice los curso que ofrece una universidad y reporte la calificación promedio obtenida en el curso (para un periodo determinado). Escriba un enunciado del ámbito que abarca este problema.

21.11. Realice una descomposición funcional de primer nivel de la función plantilla de página tratada brevemente en la sección 21.3.2.

Otras lecturas y fuentes de información

El Project Management Institute (*Guide to the Project Management Body of Knowledge*, PMI, 2001) cubre todos los aspectos importantes de la gestión de proyectos. Murch (*Project Management Best Practices for IT Professionals*, Prentice-Hall, 2000) enseña habilidades básicas y proporciona una guía detallada para todas las fases de un proyecto de TI. Lewis (*Project Managers Desk Reference*, McGraw-Hill, 1999) presenta un proceso de 16 pasos para planificar, supervisar y controlar cualquier tipo de proyecto. McConnell (*Professional Software Development*, Addison-Wesley, 2004) ofrece consejos pragmáticos para "lograr planes más cortos, productos de mayor calidad y proyectos más exitosos".

Una excelente serie de cuatro volúmenes escritos por Weinberg (*Quality Software Management*, Dorset House, 1992, 1993, 1994, 1996) introduce conceptos básicos de pensamiento y gestión de sistemas; explica cómo usar mediciones efectivamente; y aborda la "acción congruente", la habilidad de establecer "acoplamiento" entre las necesidades del gestor, las necesidades del equipo técnico y las necesidades del negocio. El libro brindará información útil a los gestores tanto nuevos como experimentados. Futrell y sus colegas (*Quality Software Project Management*, Prentice-Hall, 2002) presentan un voluminoso tratamiento de la gestión de proyectos.

Phillips (*IT Project Management: On Track from Start to Finish*, McGraw-Hill/Osborne, 2002), Charvat (*Project Management Nation*, Wiley, 2002), Schwalbe (*Information Technology Project Management*, 2a. ed., Course Technology, 2001) y Holtsnider y Jaffe (*IT Manager's Handbook*, Morgan Kaufmann Publishers, 2000) son representativos de los muchos libros que se han escrito acerca de la gestión de proyectos de software. Brown y sus colegas (*AntiPatterns in Project Management*, Wiley, 2000) examinan qué no hacer durante la gestión de un proyecto de software.

Brooks (*The Mythical Man Month*, Anniversary Edition, Addison-Wesley, 1995) ha actualizado su libro clásico para ofrecer una nueva visión en los temas de proyecto de software y gestión. McConnell (*Software Project Survival Guide*, Microsoft Press, 1997) presenta una excelente guía pragmática para quienes deben gestionar proyectos de software. Purba y Shah (*How to Ma-*

nage a Successful Software Project, 2a. ed., Wiley, 2000) presentan varios casos de estudio que indican por qué algunos proyectos tienen éxito y otros fracasan. Bennatan (*On Time Within Budget*, 3a. ed., Wiley, 2000) presenta sugerencias y lineamientos útiles para gestores de proyectos de software.

Se puede argumentar que el aspecto más importante de la gestión de proyectos de software es la gestión de personal. Cockburn (*Agile Software Development*, Addison-Wesley, 2002) presenta uno de los mejores análisis del personal de software escrito hasta la fecha. DeMarco y Lister [DEM98] han escrito el libro definitivo acerca del personal de software y los proyectos respectivos. Además, en años recientes se han publicado los siguientes libros en esta materia y vale la pena examinarlos:

Beaudouin-Lafon, M., *Computer Supported Cooperative Work*, Wiley-Liss, 1999.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones*, Prentice Hall, 1999.

Constantine, L., *Peopleware Papers: Notes on the Human Side of Software*, Prentice-Hall, 2001.

Humphrey, W. S., *Managing Technical People: Innovation, Teamwork and the Software Process*, Addison-Wesley, 1997.

Humphrey, W. S., *Introduction to the Team Software Process*, Addison-Wesley, 1999.

Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development*, McGraw-Hill, 1997.

Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments*, IEEE Computer Society, 1998.

Ensworth (*The Accidental Project Manager*, Wiley, 2001) ofrece profusamente guías útiles a quienes deben sobrevivir "la transición de técnico a gestor de proyecto". Otro excelente libro de Weinberg [WEI86] es una lectura obligada para todo gestor de proyecto y todo líder de equipo. Este libro le brindará el conocimiento y la guía que le permitirán hacer su trabajo de manera más eficiente.

Aun cuando no se relacionan específicamente con el mundo del software, y en ocasiones sobresimplifican y generalizan en extremo, los libros de "gestión" más vendidos de Bossidy (*Execution: The Discipline of Getting Things Done*, Crown Publishing, 2002), Drucker (*Management Challenges for the 21st Century*, Harper Business, 1999), Buckingham y Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently*, Simon and Schuster, 1999) y Christensen (*The Innovator's Dilemma*, Harvard Business School Press, 1997) resaltan "nuevas reglas" que define una economía rápidamente cambiante. Los títulos viejos como *Who Moved My Cheese?*, *The One-Minute Manager* e *In Search of Excellence* continúan ofreciendo valiosa información que pueden ayudarle a gestionar personas y proyectos de manera más eficaz.

En Internet está disponible una amplia variedad de fuentes de información acerca de la gestión de proyectos de software. Una lista actualizada de referencias en la World Wide Web se encuentra en el sitio Web de SEPA:
http://www.mhhe.com/pressman.

CAPÍTULO

22

MÉTRICAS DE PROCESO Y PROYECTO

La medición permite obtener una visión del proceso y el proyecto pues proporciona un mecanismo para lograr una evaluación objetiva. Lord Kelvin dijo una vez:

> Cuando puede medir aquello de lo que está hablando y expresarlo en números sabe algo acerca de ello; pero cuando no puede medir, cuando no puede expresarlo en números, su conocimiento es escaso, deficiente: puede ser el comienzo del conocimiento, pero, en sus pensamientos, apenas está avanzado al ámbito de la ciencia.

La comunidad de la ingeniería del software ha tomado en serio las palabras de lord Kelvin. ¡Mas no sin frustración y algo más que un poco de controversia!

La medición se aplica al proceso de software con la finalidad de mejorarlo de manera continua. La medición se utiliza a lo largo de un proyecto de software como apoyo en la estimación, el control de calidad, la valoración de la productividad y el control del proyecto. Finalmente, la medición la aplican los ingenieros de software como auxiliar en la evaluación de la calidad de los productos de trabajo y para apoyar la toma de decisiones táctica conforme avanza un proyecto (capítulo 15).

En su guía acerca de la medición de software, Park, Goethert y Florac [PAR96] apuntan las razones por las que se mide: 1) para *caracterizar* en un esfuerzo por comprender acerca "de los procesos, productos, recursos y entornos, y para establecer líneas base para comparaciones con evaluaciones futuras"; 2) para *evaluar* "determinando el estado con respecto a los planes"; 3) para *predecir* mediante "la comprensión de relaciones entre procesos y productos y construir mo-

UN VISTAZO RÁPIDO

¿Qué es? El proceso de software y las métricas del proyecto son medidas cuantitativas que permiten a los ingenieros de software obtener una visión de la eficacia del proceso de software y los proyectos que llevan a cabo utilizando el proceso como marco de trabajo. Se recopilan datos básicos de calidad y productividad. Luego dichos datos se analizan, comparan con promedios pasados y valoran para determinar si han ocurrido mejoras en la calidad y la productividad. Las métricas también se emplean para marcar las áreas problema de modo que se puedan desarrollar remedios y mejorar el proceso de software.

¿Quién lo hace? Los gestores de software analizan y evalúan las métricas del software. Con frecuencia, los ingenieros de software recopilan las medidas.

¿Por qué es importante? Si no se realizan mediciones el juicio sólo se basa en evaluación subjetiva. La medición permite destacar las tendencias (ya sean buenas o malas) y hacer mejores estimaciones, y con el tiempo se puede lograr una verdadera mejoría.

¿Cuáles son los pasos? Se comienza definiendo un conjunto limitado de medidas del proceso y del proyecto que puedan recopilarse con facilidad. Dichas medidas por lo general se normalizan empleando métricas orientadas al tamaño

o la función. El resultado se analiza y compara con promedios pasados para proyectos similares realizados dentro de la organización. Se valoran las tendencias y se generan conclusiones.

¿Cuál es el producto obtenido? Un conjunto de métricas del software que proporcionan amplia visión del proceso y un conocimiento detallado acerca del proyecto.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Al aplicar un esquema de medición consistente pero simple con el cual no se valora, recompensa o castiga el desempeño individual.

delos de dichas relaciones"; y 4) para *mejorar* al "identificar barricadas, causas raíz, ineficiencias y otras oportunidades para mejorar la calidad del producto y el desempeño del proceso" .

La medición es una herramienta de gestoría. Si se lleva a cabo adecuadamente, proporciona visión al gestor del proyecto. Y, como resultado, apoya al gestor del proyecto y al equipo de software a tomar decisiones que conducirán a un proyecto exitoso

22.1 MÉTRICAS EN LOS DOMINIOS DEL PROCESO Y EL PROYECTO

¡PUNTO CLAVE

Las métricas del proceso tienen impacto a largo plazo. Su objetivo es mejorar el proceso en sí. Con frecuencia, las métricas del proyecto contribuyen al desarrollo de métricas del proceso.

Las *métricas del proceso* se recopilan en el curso de todos los proyectos y durante largos periodos. Su objetivo es proporcionar un conjunto de indicadores de proceso que conduzcan a la mejora de los procesos de software de largo plazo. Las *métricas del proyecto* permiten que un gestor de proyecto de software 1) valore el estado de un proyecto en curso, 2) rastree los riesgos potenciales; 3) descubra las áreas problema antes de que se vuelvan "críticas"; 4) ajuste el flujo de trabajo o las tareas, y 5) evalúe la habilidad del equipo del proyecto para controlar la calidad de los productos del trabajo del software.

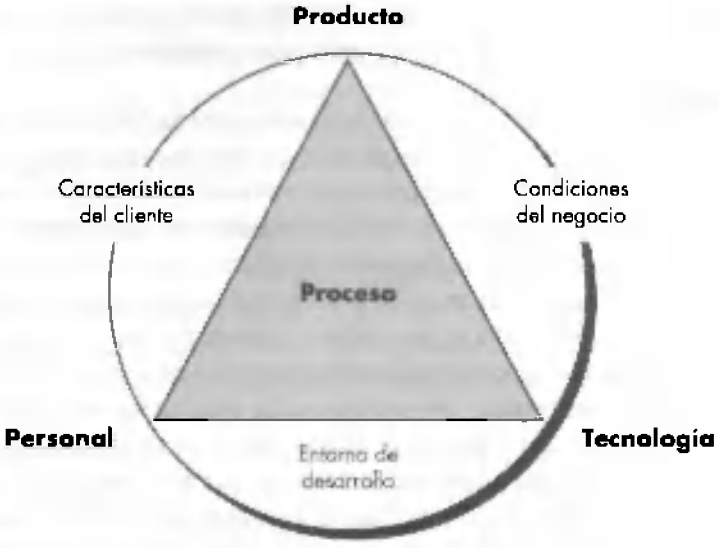
Las medidas que recopila un equipo de proyecto y las que convierte en métricas para emplearlas durante un proyecto también se pueden transmitir a quienes tienen la responsabilidad de mejorar el proceso de software. Por esta razón, muchas de las mismas métricas se usan tanto en el dominio del proceso como en el del proyecto.

22.1.1 Métricas del proceso y mejora del proceso de software

La única forma racional de mejorar cualquier proceso es medir sus atributos específicos, desarrollar un conjunto de métricas significativas con base en dichos atributos y luego emplear las métricas para ofrecer indicadores que conducirán a una estrategia de mejora. Pero antes de estudiar las métricas de software y su impacto en la mejora del proceso de software es importante destacar que el proceso sólo es uno de varios "factores controlables en la mejora de la calidad del software y el desempeño organizacional" [PAU94].

En la figura 22.1 el proceso se asienta en el centro de un triángulo que conecta tres factores con una profunda influencia en la calidad del software y el desempeño organizacional. La destreza y la motivación del personal [BOE81] se muestran como

FIGURA 22.1
...minantes ... la calidad ... software y la ...ia ...acional ...ado de ...D.



el factor individual más influyente en la calidad y el desempeño. La complejidad del producto puede tener un impacto sustancial sobre la calidad y el desempeño del equipo. La tecnología (es decir, los métodos y herramientas de la ingeniería del software) que reside en el proceso también tiene un impacto.

Además, el triángulo de proceso existe dentro de un círculo de condiciones ambientales que incluyen el entorno de desarrollo (por ejemplo, herramientas CASE), condiciones del negocio (por ejemplo, fechas límite, reglas comerciales) y características del cliente (por ejemplo, facilidad de comunicación y colaboración).

PUNTO CLAVE
...lidad y ...ción del ... de software ... el trabajo ... factores más ...s que ... en la calidad

La eficacia de un proceso de software se mide indirectamente. Esto es, se deduce un conjunto de métricas basadas en los resultados que se derivan del proceso. Los resultados incluyen medidas de errores descubiertos antes de liberar el software, los defectos que detectan y reportan los usuarios finales, los productos de trabajo entregados (productividad), el esfuerzo humano gastado, el tiempo de la planificación consumido, concordancia con la planificación y otras medidas. También se deducen métricas de proceso al medir las características de tareas específicas de ingeniería del software. Por ejemplo, se mide el esfuerzo y el tiempo utilizados al realizar las actividades genéricas de la ingeniería del software descritas en el capítulo 2

> "Las métricas de software le permiten conocer cuándo reír y cuándo llorar."
>
> Tom Gilb

Grady [GRA92] argumenta que existen usos "privados y públicos" para diferentes tipos de datos de proceso. Como es natural que los ingenieros de software especiales sean sensibles al uso de las métricas recopiladas sobre una base particular, di-

¿Cuál es la diferencia entre usos privados y públicas de las métricas de software?

chos datos deben ser privados para el individuo y funcionar como un indicador sólo para él. Los ejemplos de *métricas privadas* incluyen índices de defecto por individuo, índices de defecto por componente de software y errores encontrados durante el desarrollo.

La filosofía de "datos de proceso privados" se ajusta bien al enfoque de proceso personal del software (capítulo 2) que propone Humphrey [HUM95]. Humphrey reconoce que la mejora en el proceso de software puede y debe comenzar en el nivel individual. Los datos de proceso privados pueden funcionar como un importante promotor para que el trabajo individual del ingeniero de software mejore.

Algunas métricas de proceso son privadas para el equipo del proyecto de software, pero públicas para todos los miembros del equipo. Los ejemplos incluyen defectos que reportan las grandes funciones de software (las cuales desarrollaron varios profesionales), errores detectados durante las revisiones técnicas formales y líneas de código o puntos de función por módulo o función [1] Dichos datos los revisa el equipo para descubrir indicadores que mejoren su desempeño.

Las métricas públicas por lo general asimilan información que originalmente era privada para los individuos y equipos. Los índices de defecto al nivel del proyecto (que no se atribuyen por ningún motivo a un individuo), esfuerzo, planificación y datos relacionados se recopilan y evalúan con la finalidad de descubrir indicadores que pueden mejorar el desempeño del proceso organizacional.

Las métricas del proceso de software ofrecen beneficios significativos conforme una organización trabaja en mejorar su grado global de madurez del proceso. Sin embargo, como todas las métricas, éstas pueden emplearse mal y crear más problemas de los que solucionan. Grady [GRA92] sugiere un "conjunto de reglas de etiqueta para las métricas de software", adecuado tanto para gestores como para profesionales conforme instituyen un programa de métricas del proceso:

¿Qué lineamientos se deben aplicar cuando se recopilan métricas de software?

- Aplique sentido común y sensibilidad organizativa cuando interprete datos métricos
- Ofrezca retroalimentación regular a los individuos y equipos que recopilan medidas y métricas.
- No utilice las métricas para evaluar a los individuos
- Trabaje con los profesionales y equipos para establecer metas claras y las métricas que se emplearán para conseguirlas.
- Nunca use métricas para amenazar a los individuos o equipos
- Los datos métricos que indican una área problema no deben considerarse "negativos" Dichos datos sólo son un indicador de la mejora del proceso
- No se obsesione con una sola métrica y excluya otras métricas importantes

1 Las métricas de líneas de código y punto de función se estudian en las secciones 22.2.1 y 22.2.2

Conforme una organización se siente más cómoda con la recopilación y el empleo de las métricas de proceso, la deducción de indicadores simples da la pauta para un enfoque más riguroso llamado *mejora estadística del proceso de software* (MEPS). En esencia, el MEPS aplica el análisis de falla de software para recopilar información acerca de todos los errores y defectos[2] que se encuentran al desarrollar y utilizar una aplicación, sistema o producto

22.1.2 Métricas del proyecto

A diferencia de las métricas del proceso de software que se utilizan con propósitos estratégicos, las métricas del proyecto de software son tácticas. Es decir, un gerente de proyecto y un equipo de software emplean las métricas del proyecto y los indicadores que se deducen de ellas para adaptar el flujo de trabajo del proyecto y las actividades técnicas.

La primera aplicación de las métricas del proyecto en la mayoría de los proyectos de software ocurre durante la estimación. Las métricas recopiladas de los proyectos previos se aprovechan como base desde la cual se realizan estimaciones de esfuerzo y tiempo para el trabajo de software actual. Conforme el proyecto avanza, la medidas de esfuerzo y tiempo utilizados se comparan con las estimaciones originales (y la planificación del proyecto). El gestor del proyecto emplea dichos datos para supervisar y controlar el progreso.

Mientras comienza el trabajo técnico, las otras métricas del proyecto comienzan a tener significado. Se miden los índices de producción representados en términos de modelos creados, horas de revisión, puntos de función y líneas fuente entregadas Además, se les da seguimiento a los errores descubiertos durante cada tarea de ingeniería del software. Conforme el software evoluciona desde los requisitos hasta el diseño, se recopilan métricas técnicas para valorar la calidad del diseño y mejorar los indicadores que influirán en el enfoque que se adopte para la generación y prueba del código.

¿Cómo se emplean las métricas durante el proyecto?

La finalidad de las métricas del proyecto es doble. Primero, se emplean para minimizar el tiempo de desarrollo haciendo los ajustes necesarios para evitar demoras y reducir los problemas y riesgos potenciales. Segundo, se utilizan para valorar la calidad del producto sobre una base actual y, cuando es necesario, modificar el enfoque técnico para mejorar la calidad

Conforme la calidad mejora los defectos se minimizan, y mientras esto sucede también se reduce la cantidad de reelaboración requerida durante el proyecto. Esto conduce a una reducción en el costo global del proyecto.

2 En este libro, un *error* se define como algún fallo en un producto de trabajo de ingeniería del software que se descubre antes de que el software se entregue al usuario final. Un *defecto* es un fallo que se descubre *después* de la entrega al usuario final. Se debe advertir que otros no hacen esta distinción. En el capítulo 26 se presenta un mayor análisis

HogarSeguro

Establecimiento de un enfoque de métricas

El escenario: La oficina de Doug Miller cuando se está a punto de iniciar el proyecto de software *HogarSeguro*

Los actores: Doug Miller (gerente del equipo de ingeniería del software *HogarSeguro*) y Vinod Raman y Jamie Lazar, miembros del equipo de ingeniería de software del producto

La conversación:

Doug: Antes de comenzar a trabajar en este proyecto me gustaría que definieran y reunieran un conjunto de métricas simples. Para comenzar, tendrán que definir sus metas.

Vinod (ceñudo): Nunca antes hemos hecho eso, y

Jamie (interrumpe): Y con base en la administración del tiempo de la que se ha estado hablando, nunca tendremos el tiempo. Además, ¿qué tan buenas son las métricas?

Doug (alza las manos para detener el ataque): Calma... relájense, chicos. El hecho de que nunca antes lo hayamos hecho es la principal razón para comenzar ahora, y el trabajo de las métricas de las que estoy hablando no debería tomar mucho tiempo... de hecho, puede ahorrarnos tiempo

Vinod: ¿Cómo?

Doug: Mira, haremos mucha ingeniería de software en cosa conforme nuestro producto se vuelve más inteligente, llegue a estar habilitado en la Web, todo eso... y necesitamos entender el proceso que utilizamos para construir software... y mejorarlo de modo que construyamos mejor software. La única forma de hacer esto es midiendo.

Jamie: Pero tenemos presión de tiempo, Doug. No estoy en favor de más presión de papeleo... necesitamos el tiempo para hacer nuestro trabajo, no recopilar datos

Doug (calmadamente): Jamie, el trabajo de un ingeniero involucra recopilar datos, evaluarlos y aprovechar los resultados para mejorar el producto y el proceso ¿Me equivoco?

Jamie: No, pero...

Doug: ¿Qué tal si dejamos el número de medidas que deben reunirse en no más de cinco o seis y nos enfocamos en la calidad?

Vinod: Nadie puede estar en contra de la alta calidad...

Jamie: Cierto... pero, no sé, todavía creo que esto no es necesario

Doug: Esta vez les pediré que me complazcan ¿Cuánto saben acerca de las métricas de software?

Jamie (voltea y ve a Vinod): No mucho

Doug: Aquí tienen algunas referencias de la Web... pasen algunas horas organizándose para aumentar la velocidad.

Jamie (sonríe): Pensé que dijiste que esto no nos llevaría tiempo

Doug: El tiempo que pasan aprendiendo nunca es tiempo perdido... vayan a hacerlo y luego estableceremos algunas metas, plantearemos algunas preguntas y definiremos las métricas que necesitamos reunir

22.2 Medición del software

En el capítulo 15 se indicó que la medición de software se clasifica en dos categorías: 1) *medidas directas* del proceso de software (por ejemplo, costo y esfuerzo aplicados) y del producto (por ejemplo, líneas de código [LDC] producidas, rapidez de ejecución y defectos reportados a lo largo de cierto periodo establecido) y 2) *medidas indirectas* del producto que incluyen funcionalidad, calidad, complejidad, eficiencia, confiabilidad, facilidad de mantenimiento y muchas otras "habilidades" tratadas en el capítulo 15.

"No todo lo que puede ser contado cuenta, y no todo lo que cuenta puede ser contado."

Albert Einstein

Las métricas del proyecto se consolidan con el fin de crear métricas del proceso que sean públicas para la organización de software como un todo. Pero, ¿cómo combina una organización las métricas provenientes de diferentes individuos o proyectos?

Con fines ilustrativos, considérese un ejemplo simple. Los integrantes de dos diferentes equipos de proyecto registran y categorizan los errores que encuentran durante el proceso del software. Luego, las mediciones individuales se combinan para desarrollar medidas de equipo. El equipo A encontró 342 errores durante el proceso del software previo al lanzamiento. El equipo B encontró 184 errores. Si todas las demás cosas se mantienen iguales, ¿qué equipo es más eficiente al descubrir errores a lo largo del proceso? Puesto que no se conocen ni el tamaño ni la complejidad de los proyectos, no se puede responder esta pregunta. Sin embargo, si las mediciones se normalizan, es posible crear métricas de software que posibiliten la comparación a promedios organizacionales más amplios. De esta forma, las métricas orientadas tanto al tamaño como a la función están normalizadas.

22.2.1 Métricas orientadas al tamaño

Las métricas del software orientadas al tamaño preceden de la normalización de las medidas de calidad o productividad considerando el tamaño del software que se ha producido. Si una organización de software mantiene registros simples es factible crear una tabla de medidas orientadas al tamaño, como la que se muestra en la figura 22.2. En la tabla se menciona cada proyecto de desarrollo de software que se ha completado en años pasados, así como las medidas correspondientes para dichos proyectos. Como se advierte en la entrada de tabla (figura 22.2) para el proyecto alfa: 12 100 líneas de código se desarrollaron con 24 personas-mes de esfuerzo a un costo de 168 000 dólares. Se debe notar que el esfuerzo y el costo registrados en la tabla representan todas las actividades de ingeniería del software (análisis, diseño, código y prueba), no sólo codificación. Información adicional del proyecto alfa indica que se desarrollaron 365 páginas de documentación, se registraron 134 errores antes de que el software fuese liberado y se encontraron 29 defectos después de la liberación al cliente dentro del primer año de operación. Tres personas trabajaron en el desarrollo del software para el proyecto alfa.

El desarrollo de métricas que se asimilen con métricas similares procedentes de otros proyectos requiere elegir *líneas de código* como valor de normalización. A partir de los datos rudimentarios de la tabla se desarrolla un conjunto de métricas simples orientadas al tamaño para cada proyecto: errores por KLDC (miles de líneas de código), defectos por KLDC, costo por KLDC, páginas de documentación por KLDC. Además, se pueden calcular otras métricas interesantes: errores por persona-mes, KLDC por persona-mes, costo por página de documentación.

FIGURA 22.2

Métricas orientadas al tamaño.

Proyecto	LDC	Esfuerzo	$(000)	Pag. Doc.	Errores	Defectos	Personal
alfa	12 100	24	168	365	134	29	3
beta	27 200	62	440	1 224	321	86	5
gamma	20 200	43	314	1 050	256	64	6
...	...	...	...	...	...	...	...

PUNTO CLAVE

Las métricas orientadas al tamaño son ampliamente utilizadas, pero continúa el debate acerca de su validez y aplicabilidad.

Las métricas orientadas al tamaño no se aceptan universalmente como la mejor forma de medir el proceso del software [JON86]. La mayor parte de la controversia gira en torno al uso de líneas de código como medida clave. Los partidarios de la medida LDC afirman que éstas son un "artefacto" de todos los proyectos de desarrollo de software que pueden fácilmente contarse, que muchos modelos de estimación de software existentes usan LDC o KLDC como entrada clave, y que ya existe un gran cuerpo de bibliografía y datos publicados para LDC. Por otra parte, los detractores argumentan que las medidas LDC dependen del lenguaje de programación, que, cuando se considera la productividad, castigan los programas bien diseñados, pero más cortos, que no pueden amoldar con facilidad lenguajes que no provienen del proceso y cuyo empleo en la estimación requiere un nivel de detalle que sería difícil de lograr (es decir, el planificador debe estimar que las LDC se producirán mucho antes de que el análisis y el diseño se hayan completado).

22.2.2 Métricas orientadas a la función

Las métricas de software orientadas a la función emplean como un valor de normalización una medida de la funcionalidad que entrega la aplicación. La métrica orientada a la función utilizada con mayor amplitud es el *punto de función* (PF). El cálculo del punto de función se basa en características del dominio de información y la complejidad del software. La mecánica del cálculo del PF se trató en el Capítulo 15.

El punto de función, al igual que la medida LDC, es controversial. Los partidarios afirman que el PF es independiente del lenguaje de programación, característica que lo hace ideal para aplicaciones que utilizan lenguajes convencionales y no procedimentales, y que se basa en datos que es más probable que se conozcan tempranamente en la evolución de un proyecto, lo que hace al PF más atractivo como enfoque de es-

3 Véase la sección 15.3.1 para una detallada exposición del cálculo de PF.

timación. Los detractores afirman que el método requiere cierta "prestidigitación" en cuanto a que el cálculo se basa en datos subjetivos más que objetivos, que el conteo del dominio de información (y otras dimensiones) puede ser difícil de recopilar después del hecho, y que el PF no tiene significado físico directo: es sólo un número.

22.2.3 Reconciliación de las métricas LDC y PF

La relación entre líneas de código y puntos de función depende del lenguaje de programación en que se implementan el software y la calidad del diseño. Varios estudios han intentado relacionar las medidas de PF y LDC. Por ejemplo Albrecth y Gaffney [ALB83]:

> La tesis de este trabajo es que la cantidad de función que se ofrecerá por medio de la aplicación (programa) se puede estimar a partir de pormenorizar los grandes componentes[4] de datos que se emplearán o proporcionarán. Más aún, esta estimación de la función debe estar correlacionada tanto con la cantidad de LDC que se desarrollará como con el esfuerzo de desarrollo necesario

La tabla[5] siguiente [QSM02] ofrece estimaciones burdas del número promedio de líneas de código que se requieren para construir un punto de función en varios lenguajes de programación:

Una revisión de estos datos indica que una LDC de C++ proporciona aproximadamente 2 4 veces la "funcionalidad" (en promedio) de una LDC de C Más aún, una LDC de Smalltalk proporciona al menos cuatro veces la funcionalidad de una LDC de un lenguaje de programación convencional como Ada, COBOL o C La utilización de la información contenida en la tabla permite "tomar como contrafuego" [JON98] el software existente para estimar el número de puntos de función, una vez que se conozca el número total de enunciados del lenguaje de programación.

Se ha encontrado que las métricas basadas en puntos de función y LDC son indicadoras relativamente precisos del esfuerzo y el costo del desarrollo de software. Sin embargo, emplear LDC y PF en la estimación (capítulo 23) requiere establecer una línea de referencia histórica de información.

En contexto del proceso y las métricas del proyecto, la preocupación principal la generan la productividad y la calidad· medidas de la "salida" de desarrollo de software como función del esfuerzo y el tiempo aplicados y medidas de la "aptitud para el uso" de los productos de trabajo obtenidos. Respecto a propósitos de mejora del proceso y planeación del proyecto, el interés es histórico. ¿Cuál fue la productividad

4 Es importante notar que "pormenorizar los grandes componentes" se puede interpretar en varias formas. Los ingenieros de software que trabajan en un entorno de desarrollo orientado a objetos usan el número de clases u objetos como el tamaño de métrica dominante Una organización de mantenimiento puede considerar el tamaño del proyecto en términos del número de pedidos de cambios de ingeniería (capítulo 27). Una organización de sistemas de información quizá vea el número de procesos comerciales que afecta una aplicación

5 Utilizado con permiso de Quantitative Software Management (www.qsm.com), copyright 2002

de desarrollo del software en los proyectos pasados? ¿Cuál fue la calidad del software que se produjo? ¿Cómo se pueden extrapolar al presente la productividad y la calidad pasadas? ¿Cómo puede ayudar a mejorar el proceso y planificar nuevos proyectos con mayor precisión?

Lenguaje de programación	*LDC por punto de función*			
	Promedio	**Mediana**	**Baja**	**Alto**
Access	35	38	15	47
Ada	154	—	104	205
APS	86	83	20	184
ASP 69	62	—	32	127
Ensamblador	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
Clipper	38	39	27	70
COBOL	77	77	14	400
Cool·Gen/IEF	38	31	10	180
Culprit	51	—	—	—
DBase IV	52	—	—	—
Easytrieve1	33	34	25	41
Excel47	46	—	31	63
Focus	43	42	32	56
FORTRAN	—	—	—	—
FoxPro	32	35	25	35
Ideal	66	52	34	203
IEF/Cool·Gen	38	31	10	180
Informix	42	31	24	57
Java	63	53	77	—
JavaScript	58	63	42	75
JCL	91	123	26	150
JSP	59	—	—	—
Lotus Notes	21	22	15	25
Mantis	71	27	22	250
Mapper	118	81	16	245
Natural	60	52	22	141
Oracle	30	35	4	217
PeopleSoft	33	32	30	40
Perl	60	—	—	—
PL/1	78	67	22	263
Powerbuilder	32	31	11	105
REXX	67	—	—	—
RPG II/III	61	49	24	155
SAS	40	41	33	49
Smalltalk	26	19	10	55
SQL	40	37	7	110
VBScript36	34	27	50	—
Visual Basic	47	42	16	158

22.2.4 Métricas orientadas a objetos

Las métricas de proyectos de software convencionales (LDC o PF) se aplican en la estimación de proyectos de software orientados a objetos. Sin embargo, estas métricas no proporcionan suficiente granularidad para la planificación y los ajustes de esfuerzo que se requieren conforme se itera a lo largo de un proceso evolutivo o incremental. Lorenz y Kidd [LOR94] sugieren el siguiente conjunto de métricas para proyectos OO:

Número de guiones de escenario. Un guión de escenario (análogo a los casos de uso estudiados a través de las partes 2 y 3 de este libro) es una secuencia detallada de pasos que describen la interacción entre el usuario y la aplicación. El número de guiones de escenario está directamente correlacionado con el tamaño de la aplicación y con el número de casos de prueba que se deben desarrollar para ejercitar el sistema una vez que se construya.

Número de clases clave. Las *clases clave* son los "componentes enormemente independientes" [LOR94] definidos con antelación en el análisis orientado a objetos (capítulo 8).[6] Puesto que las clases clave son centrales respecto del dominio del problema, su número indica la cantidad de esfuerzo necesario para desarrollar el software. También indican la potencial cantidad de reutilización que se aplicará durante el desarrollo del sistema.

Número de clases de apoyo. Las *clases de apoyo* son necesarias en la implementación del sistema, pero no están inmediatamente relacionadas con el dominio del problema. Los ejemplos pueden ser las clases IU, los accesos a bases de datos y las clases de manipulación y las de cálculo. Además, las clases de apoyo se pueden desarrollar para cada una de las clases clave. El número de clases de apoyo es un indicio de la cantidad de esfuerzo indispensable para desarrollar el software, así como un indicio de la potencial cantidad de reutilización que se aplicará durante el desarrollo del sistema.

Número promedio de clases de apoyo por clase clave. En general, las clases clave se conocen en etapas iniciales del proyecto. Las clases de apoyo se definen a lo largo del curso de éste. Si se conociese el número promedio de clases de apoyo por clase clave respecto de un dominio de problema dado, la estimación (con base en el número total de clases) se simplificaría mucho. Lorenz y Kidd sugieren que las aplicaciones con una GUI tienen entre dos y tres veces el número de clases de apoyo que las clases clave. Las aplicaciones sin GUI tienen entre una y dos veces el número de clases de apoyo que las clases clave.

Número de subsistemas. Un *subsistema* es un agregado de clases que apoyan una función visible para el usuario final de un sistema. Una vez identificados los sub-

6 En la parte 2 de este libro a las clases clave se les refino como *clases de análisis*.

sistemas es más fácil extender una planificación razonable en la cual se haya hecho la partición del trabajo entre el equipo del proyecto.

La utilización eficiente en un entorno de ingeniería de software orientada a objetos requiere recopilar métricas similares a las anotadas líneas arriba, junto con medidas del proyecto tales como esfuerzo gastado, errores y defectos descubiertos y modelos o páginas de documentación producidos. Conforme la base de datos crece (después de completados varios proyectos), las relaciones entre medidas orientadas a objetos y medidas de proyecto proporcionarán métricas que auxilien en la estimación del proyecto.

22.2.5 Métricas orientadas a casos de uso

Parecería razonable aplicar el caso de uso[7] como una medida de normalización similar a la LDC o PF. Como el PF, el caso de uso se define en etapas tempranas del proceso de software, lo que permite emplearlo en la estimación antes de iniciar las actividades significativas de modelado y construcción. Los casos de uso describen (al menos indirectamente) funciones y características visibles al usuario que son requisitos básicos para un sistema. El caso de uso es independiente del lenguaje de programación. Además, el número de casos de uso es directamente proporcional al tamaño de la aplicación en LDC, así como al número de casos de prueba que tendrán que diseñarse para ejercitar completamente la aplicación.

Puesto que los casos de uso pueden crearse con grados de abstracción ampliamente diferentes, no existe tamaño estándar para ellos. Sin una medida estándar, la aplicación como medida de normalización (por ejemplo, esfuerzo empleado por caso de uso) es sospechosa. Aunque varios investigadores (por ejemplo, [SMI99]) han intentado obtener métricas de caso de uso, todavía queda mucho trabajo por hacer.

22.2.6 Métricas de proyectos de ingeniería Web

El objetivo de todos los proyectos de ingeniería Web (parte 3 de este libro) es construir una aplicación Web (WebApp) que propocione una combinación de contenido y funcionalidad al usuario final. Las medidas y métricas que se emplean en los proyectos de ingeniería de software tradicionales son difíciles de traducir directamente a la WebApp. Incluso, una organización de ingeniería Web debe desarrollar una base de datos que le permita valorar su productividad y calidad internas a lo largo de varios proyectos. Entre las medidas que se pueden recopilar están:

Número de páginas Web estáticas. Las páginas Web de contenido estático (es decir, el usuario final no controla el contenido desplegado en la página) son las más comunes de todas las características WebApp. Estas páginas representan una complejidad relativa baja y por lo general requieren menos esfuerzo al construirlas que

7 Los casos de uso se estudian a través de las partes 2 y 3 de este libro.

las páginas dinámicas. Esta medida proporciona un indicio del tamaño global de la aplicación y el esfuerzo que se requiere para desarrollarla

Número de páginas Web dinámicas. Las páginas Web de contenido dinámico (es decir, las acciones del usuario final generan contenido personalizado que se despliega en la página) son esenciales en todas las aplicaciones de comercio electrónico, motores de búsqueda, aplicaciones financieras y muchas otras categorías de WebApp. Estas páginas representan una mayor complejidad relativa y requieren más esfuerzo al construirlas que las páginas estáticas. Esta medida proporciona un indicio del tamaño global de la aplicación y el esfuerzo requerido para desarrollarla

Número de vínculos internos de página. Los vínculos internos de página son punteros que ofrecen un hipervínculo hacia alguna otra página Web dentro de la WebApp. Esta medida proporciona un indicio del grado de acoplamiento arquitectónico dentro de la WebApp. Conforme aumenta el número de vínculos de la página, también lo hace el esfuerzo empleado en el diseño y construcción de la navegación

Número de objetos de datos persistentes. Una WebApp puede tener acceso a uno o más objetos de datos persistentes (por ejemplo, una base de datos o archivo de datos) Conforme el número de objetos de datos persistentes crece, también lo hace la complejidad de la WebApp y el esfuerzo para implementarla aumenta proporcionalmente.

Número de sistemas externos en interfaz. Con frecuencia las WebApps deben hacer interfaz con aplicaciones comerciales "de cuarto trasero". Conforme crece el requisito para hacer interfaz, la complejidad del sistema y el esfuerzo de desarrollo también aumentan.

Número de objetos de contenido estático. Los objetos de contenido estático abarcan información estática basada en texto, gráfica, video, animación y audio que se incorporan dentro de la WebApp. En una página Web sencilla pueden aparecer múltiples objetos de contenido estático.

Número de objetos de contenido dinámico. Los objetos de contenido dinámico se generan con base en las acciones del usuario final y abarcan información generada internamente basada en texto, gráfica, video, animación y audio que se incorporan dentro de la WebApp. En una página Web sencilla pueden aparecer múltiples objetos de contenido dinámico

Número de funciones ejecutables. Una función ejecutable (por ejemplo, un guión o *applet*) ofrece cierto servicio computacional al usuario final. Conforme aumenta el número de funciones ejecutables, también aumentan los esfuerzos de modelado y construcción.

Cada una de estas medidas se puede determinar en una etapa relativamente temprana del proceso de ingeniería Web.

Por ejemplo, es posible definir una métrica que refleje el grado de personalización de usuario final que se requiere para la WebApp y correlacionarla con el esfuerzo

empleado en el proyecto de IWeb o los errores descubiertos conforme se llevan a cabo revisiones y pruebas. Para lograr esto, se define

N_{sp} = número de páginas Web estáticas
N_{dp} = número de páginas Web dinámicas

Entonces,

Índice de personalización, $C = N_{dp}/(N_{dp} + N_{sp})$

El valor de C varía de 0 a 1. Conforme C crece, el nivel de personalización de la WebApp se vuelve un conflicto técnico significativo.

Es posible calcular y correlacionar métricas similares de aplicaciones Web con medidas del proyecto, tales como el esfuerzo empleado, los errores y defectos descubiertos y los modelos o páginas de documentación producidos. Conforme la base de datos crece (después de que varios proyectos se han completado), las relaciones entre las medidas WebApp y las medidas del proyecto proporcionarán indicadores que auxilien en la estimación del proyecto.

HERRAMIENTAS DE SOFTWARE

Métricas del proyecto y el proceso

Objetivo: Ayudar en la definición, recopilación, evaluación y reporte de medidas y métricas de software.

Mecánica: Cada herramienta varía en cuanto a su aplicación, pero todas ofrecen mecanismos para recopilar y evaluar datos que conduzcan al cálculo de métricas de software.

Herramientas representativas[8]

Function Point WORKBENCH, desarrollada por Charismatek (www.charismatek.com.au), ofrece una amplia variedad de métricas orientadas a PF.

MetricCenter, desarrollada por Distributive Software (www.distributive.com), soporta recopilación automatizada de datos, análisis, formateo de gráficos, generación de reportes y otras tareas de medición.

PSM Insight, desarrollada por Practical Software and Systems Measurement (www.psmsc.com), auxilia en la creación y subsiguiente análisis de una base de datos de medición del proyecto.

SLIM tool set, desarrollado por QSM (www.qsm.com), proporciona un completo conjunto de métricas y herramientas de estimación.

SPR tool set, desarrollado por Software Productivity Research (www.spr.com), ofrece una colección detallada de herramientas orientadas a PF.

TychoMetrics, desarrollado por Predicate Logic, Inc. (www.predicate.com), es una suite de herramientas para gestionar recopilación de métricas y reportes.

22.3 MÉTRICAS PARA CALIDAD DEL SOFTWARE

La meta primordial de la ingeniería del software es producir un sistema, aplicación o producto de alta calidad dentro de un marco temporal que satisfaga una necesidad

8 Las herramientas anotadas aquí son una muestra de esta categoría. En la mayoría de los casos los nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

del mercado. El logro de esta meta requiere que los ingenieros de software apliquen métodos eficaces acoplados con herramientas modernas dentro del contexto de un proceso de software maduro. Además, un buen ingeniero de software (y los buenos gestores de ingeniería del software) debe medir si se logrará la alta calidad.

Las métricas privadas reunidas por los ingenieros de software individuales se asimilan con los resultados ofrecidos en el ámbito del proyecto. Aunque se pueden reunir muchas medidas de calidad, el impulso primario en el ámbito del proyecto es medir los errores y defectos. Las métricas derivadas de estas medidas proporcionan un indicio de la efectividad de la garantía de la calidad del software y de las actividades de control tanto de los individuos como del grupo.

Las métricas como los errores en el producto de trabajo (por ejemplo, requisitos o diseño) por punto de función, errores descubiertos por hora de revisión, y los errores descubiertos por hora de prueba ofrecen una visión de la eficacia de cada una de las actividades implicadas en la métrica. Los datos de error también se pueden emplear en el cálculo de la *eficacia en la eliminación de defectos* (EED) para cada actividad del marco de trabajo del proceso. La EED se estudia en la sección 22.3.2.

22.3.1 Medición de la calidad

Aunque existen muchas medidas de la calidad del software,[9] la corrección, la facilidad de mantenimiento, la integridad y la facilidad de uso ofrecen indicadores útiles para el equipo del proyecto. Gilb [GIL88] sugiere definiciones y mediciones para cada una de ellas.

Referencia Web

Una excelente fuente [illegible] acerca [illegible] calidad de [illegible] y kilobytes [illegible] gratuitos [illegible] se encuentra en www.qualityworld.com.

Corrección. Un programa debe operar correctamente o proporcionará poco valor para sus usuarios. La corrección es el grado en que el software desempeña la función para la que fue creado. La medida más común para la corrección es defectos por KLDC, donde un defecto se define como una falta comprobada de concordancia con los requisitos. Cuando se considera la calidad global de un producto de software, los defectos son los problemas que reporta un usuario del programa después de que éste se liberó para el uso general. Para los propósitos de la evaluación de la calidad, los defectos se cuentan sobre un periodo estándar, usualmente un año.

Facilidad de mantenimiento. El mantenimiento del software justifica más esfuerzos que cualquier otra actividad de la ingeniería del software. La facilidad de mantenimiento es la sencillez con la que un programa puede corregirse si se encuentra un error, adaptarse si su entorno cambia, o mejorar si el cliente desea un cambio en los requisitos. No existe forma de medir directamente la facilidad de mantenimiento; en consecuencia, se deben emplear medidas indirectas. Una simple medida orientada al tiempo es el *tiempo medio de cambio* (TMC), el tiempo que toma analizar el cambio solicitado, diseñar una modificación apropiada, implementar el

9 En el capítulo 15 se presentó una discusión detallada de los factores que influyen en la calidad del software y las métricas que se pueden usar para valorar la calidad del software.

cambio, probarlo y distribuir el cambio a todos los usuarios. En promedio, los programas susceptibles de mantenimiento tendrán un TMC bajo (para tipos de cambios equivalentes) que los programas que no lo son

Integridad. La integridad del software se ha vuelto cada vez más importante en la edad de los ciberterroristas y *hackers*. Este atributo mide la habilidad de un sistema para resistir ataques (tanto accidentales como intencionales) a su seguridad. Los ataques se pueden realizar en los tres componentes del software: programas, datos y documentos.

La medición de la integridad requiere definir dos atributos adicionales: amenaza y seguridad. *Amenaza* es la probabilidad (que puede estimarse o deducirse de evidencia empírica) de que un ataque de un tipo específico ocurrirá dentro de un tiempo dado. *Seguridad* es la probabilidad (que puede estimarse o deducirse de evidencia empírica) de que se repela el ataque de un tipo específico. Entonces, la integridad de un sistema se puede definir como:

$$\text{integridad} = 1 - (\text{amenaza} \times (1 - \text{seguridad}))$$

Por ejemplo, si la amenaza (la probabilidad de que un ataque ocurrirá) es 0.25 y la seguridad (la posibilidad de repeler un ataque) es 0.95, la integridad del sistema es 0.99 (muy elevada). Si, por otra parte, la probabilidad de amenaza es 0.50 y la posibilidad de repeler un ataque es sólo 0.25, la integridad del sistema es 0.63 (inaceptablemente baja).

Facilidad de uso. Con frecuencia, un programa que no es fácil de usar está condenado al fracaso, incluso si las funciones que realiza son valiosas. La facilidad de uso es un intento por cuantificar la sencillez de la aplicación al utilizarla y se puede medir en términos de las características presentadas en el capítulo 12.

Los cuatro factores apenas descritos sólo representan una muestra de los que se han propuesto como medidas para la calidad del software. El capítulo 15 considera este tópico con mayor detalle.

22.3.2 Eficacia en la eliminación de defectos

Una métrica de calidad que ofrece beneficios tanto en el ámbito del proyecto como en el del proceso es la *eficacia en la eliminación de defectos* (EED). En esencia, la EED es una medida de la habilidad de filtrar las actividades de la garantía de cualidad y de control conforme se aplica a través de todas las actividades del marco de trabajo del proceso.

Cuando se considera un proyecto como un todo, la EED se define de la manera siguiente:

$$\text{EED} = E/(E + D)$$

donde E es el número de errores encontrados antes de entregar el software al usuario final, y D es el número de defectos encontrados después de la entrega

El valor ideal de la EED es 1. Esto es: no se encuentra defecto alguno en el software. En realidad, D será mayor que 0, pero el valor de EED todavía puede acercarse a 1. Conforme E aumenta (para un valor dado de D), el valor global de EED comienza a acercarse a 1. De hecho, conforme E aumenta, es probable que el valor final de D disminuya (los errores se filtran antes de que se conviertan en defectos). Si se utiliza como una métrica que proporciona un indicador de la habilidad de filtrado de las actividades de control y aseguramiento de la calidad, EED alienta a un equipo de proyecto de software a instituir técnicas para encontrar tantos errores como sea posible antes de la entrega.

La EED también se puede aplicar en el proyecto para valorar la habilidad de un equipo de encontrar errores antes de que pasen a la siguiente actividad del marco de trabajo o a la siguiente tarea en la ingeniería del software. Por ejemplo, la tarea de análisis de requisitos produce un modelo de análisis que se revisa para encontrar y corregir errores. Aquellos errores que no se encuentran durante la revisión del modelo de análisis pasan al diseño (donde pueden o no encontrarse). Cuando se aplica en este contexto la EED se redefine como

$$EED_i = E_i/(E_i + E_{i+1})$$

donde E_i es el número de errores encontrado durante la actividad i de ingeniería de software y E_{i+1} es el número de errores encontrado durante la actividad $i + 1$ de ingeniería del software que se puede seguir para llegar a errores que no fueron descubiertos en la actividad i de ingeniería del software.

Un objetivo de calidad para un equipo de software (o un ingeniero de software individual) es lograr una EED, que se acerque a 1. Esto es: los errores deben filtrarse antes de que pasen a la siguiente actividad.

HOGARSEGURO

Establecimiento de un enfoque de métricas

El escenario: Oficina de Doug [illegible] días después de la reunión inicial acerca de [illegible]cas de software.

Los actores: Doug Miller (gerente del equipo de ingeniería de software de *HogarSeguro*) y Vinod Raman y Jamie Lazar, miembros del equipo de ingeniería de software del producto.

La conversación:

Doug: ¿Ustedes dos tienen oportunidad de aprender un poco acerca de métricas del proceso y el proyecto?

Vinod y Jamie: [Ambos asienten con la cabeza.]

Doug: Siempre es una buena idea establecer metas cuando adoptan alguna métricas. ¿Cuáles son las suyas?

Vinod: Nuestras métricas se deben enfocar en la calidad. De hecho, nuestra meta global es mantener en un mínimo absoluto el número de errores que pasamos de una actividad de ingeniería del software a la siguiente.

Doug: Y asegúrense muy bien de que el número de defectos liberados con el producto se mantenga tan cerca de cero como sea posible.

Vinod (asiente con la cabeza): Desde luego.

Jamie: Me gusta la EED como métrica, y creo que podemos emplearla en todo el proyecto. Además, podemos aplicarla conforme nos movamos de una actividad del marco de trabajo a la siguiente. Eso nos alentará para encontrar errores en cada paso.

Vinod: También me gustaría reunir el número de horas que pasamos en las revisiones

Jamie: Y el esfuerzo global que pasamos en cada tarea de ingeniería del software.

Doug: Tú puedes calcular una razón de revisión a desarrollo... podría ser interesante.

Jamie: Me gustaría seguir también algunos datos de caso de uso. Como la cantidad de esfuerzo requerido para desarrollar un caso de uso, la cantidad de esfuerzo requerido para construir software para implementar un caso de uso y. .

Doug (sonríe): Creo que tendremos que conservar esto simple.

Vinod: Deberíamos, pero una vez que te metes en el asunto de las métricas, existen muchas cosas interesantes que observar

Doug: Estoy de acuerdo, pero caminemos antes de correr, y apeguémonos a nuestra meta. Limiten los datos que recopilen a cinco o seis elementos, y estamos listos para despegar.

22.4 INTEGRACIÓN DE LAS MÉTRICAS DENTRO DEL PROCESO DE SOFTWARE

La mayoría de los desarrolladores de software todavía no miden y, por desgracia, muchos tienen poco deseo de comenzar. Como se ha señalado en este capítulo, el problema es cultural. El intento de recopilar medidas donde nadie lo ha hecho en el pasado con frecuencia genera resistencia. "¿Por qué tenemos que hacer esto?" pregunta un gestor de proyecto acosado. "No le veo el caso", se queja un profesional con exceso de trabajo.

En esta sección se consideran algunos argumentos para las métricas de software y se presenta un enfoque para instituir un programa de recopilación de métricas en una organización de ingeniería del software. Pero antes de comenzar, conviene considerar [GRA87] algunas palabras de cordura de Grady y Caswell:

> Algunas de las cosas que describimos aquí sonarán bastante sencillas. En realidad, sin embargo, el establecimiento de un programa de métricas de software exitoso en el ámbito de la compañía es un trabajo duro. Cuando decimos que se debe esperar al menos tres años antes de que estén disponibles tendencias organizacionales amplias, se obtiene alguna idea del ámbito de tal esfuerzo.

Vale la pena prestar atención a la advertencia que sugieren estos autores, pero los beneficios de la medición son tan convincentes que el trabajo duro vale la pena

22.4.1 Argumentos para las métricas del software

¿Por qué es importante medir el proceso de ingeniería del software y el producto (software) que elabora? La respuesta es relativamente obvia. Si no se mide, no existe una forma real de determinar si se está mejorando. Y si no se mejora, se está perdido.

Al cuestionar y evaluar la productividad y las medidas de calidad, un equipo de software (y su gestión) puede establecer metas significativas para mejorar el proceso del software. En el capítulo 1 se apuntó que el software es un tema comercial e-

tratégico para muchas compañías. Si el proceso con el cual se desarrolla puede mejorarse, se producirá un impacto directo en lo sustancial. Pero para establecer objetivos de mejora es preciso comprender el estado actual del desarrollo de software. Por lo tanto, la medición se emplea para establecer una línea base de proceso a partir de la cual se evalúan las mejoras.

> "Gestionamos las cosas mediante los números en muchos aspectos de nuestras vidas... Estos números nos brindan capacidad de juicio y nos ayudan a dirigir nuestras acciones."
>
> **Michael Mah y Larry Putnam**

Los rigores cotidianos del trabajo del proyecto de software dejan poco tiempo para ejercitar el pensamiento estratégico. Los gestores de proyectos de software están preocupados con temas más concretos (aunque igualmente importantes): desarrollar estimaciones de proyecto significativas, producir sistemas de alta calidad, tener el producto en circulación a tiempo. Si se emplea la medición para establecer una línea base del proyecto, cada uno de dichos temas se vuelve más manejable. Ya se ha mencionado que la línea base sirve como fundamento para la estimación. Adicionalmente, la recopilación de métricas de calidad permite que una organización "sintonice" su proceso de software para remover las causas "poco vitales" de los defectos que tienen el mayor impacto sobre el desarrollo del software [10]

22.4.2 Establecimiento de una línea base

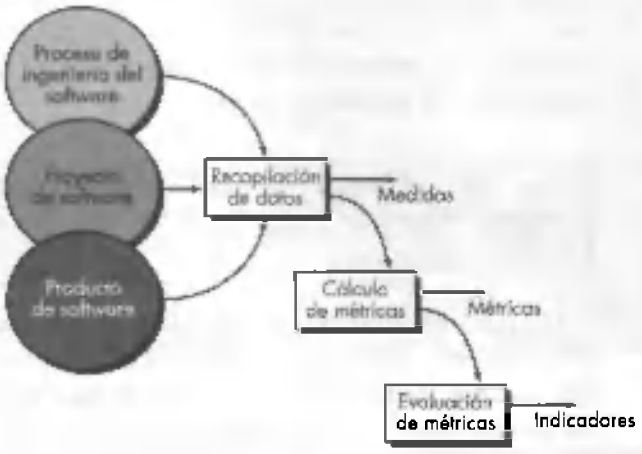
¿Qué es una línea base de métricas y qué beneficios le proporciona a un ingeniero de software?

Con el establecimiento de una línea base de métricas se obtienen beneficios en los ámbitos del proceso, del proyecto y del producto (técnico). Incluso la información que se recopila no necesita ser fundamentalmente diferente. Las mismas métricas pueden servir a muchos maestros. La línea base de métricas consiste de datos recopilados en proyectos previos de desarrollo de software y pueden ser tan simples como la tabla presentada en la figura 22.2 o tan complejos como una base de datos detallada que contiene docenas de medidas de proyectos y las métricas derivadas de éstos

Para ser un auxiliar eficaz en el proceso de mejora o de costo y esfuerzo, los datos de la línea base deben tener los siguientes atributos: 1) los datos deben ser razonablemente precisos: se deben evitar las "conjeturas" acerca de los proyectos pasados 2) los datos deben recopilarse para tantos proyectos como sea posible; 3) las medidas deben ser consistentes, por ejemplo: una línea de código debe interpretarse consistentemente a través de todos los proyectos para los que se recopilan los datos, 4) las aplicaciones deben ser similares al trabajo que se estimará: tiene poco sentido emplear una línea base en un trabajo de sistemas de información en bloque para estimar una aplicación anidada en tiempo real.

[10] En el capítulo 26 se tratan con detalle estas ideas, formalizadas en un enfoque denominado *garantía estadística de calidad del software*

FIGURA 22.3
Proceso de recopilación de métricas de software.



PUNTO CLAVE

Los datos de métricas de línea base deben recopilarse de una gran muestra representativa de proyectos de software previos.

22.4.3 Recopilación, cálculo y evaluación de métricas

En la figura 22.3 se ilustra el proceso con el que se establece una línea base de métricas. De manera ideal, los datos necesarios para hacerlo se han recopilado conforme se avanza. Por desgracia, esto rara vez es el caso. En consecuencia, la recopilación de datos requiere una investigación histórica de los proyectos previos para reconstruir los datos requeridos. Una vez que se han recopilado las medidas (incuestionablemente el paso más difícil) es posible calcular las métricas. Dependiendo de la amplitud de las medidas recopiladas, las métricas pueden abarcar una amplia gama de métricas orientadas a la aplicación (por ejemplo, LDC, PF, orientada a objetos, WebApp), así como otras orientadas a la calidad y al proyecto. Finalmente, las métricas deben evaluarse y aplicarse durante la estimación del trabajo técnico, el control del proyecto y la mejora del proceso. La evaluación de las métricas se centra en las razones subyacentes de los resultados obtenidos y produce un conjunto de indicadores que guían el proyecto o proceso.

22.5 MÉTRICAS PARA ORGANIZACIONES PEQUEÑAS

CONSEJO

Si se está comenzando a recopilar datos de métricas, recuérdese mantenerlos simples. Si los datos abruman los esfuerzos de métricas fracasarán.

La gran mayoría de las organizaciones de desarrollo de software tiene menos de 20 empleados. Es irracional, y en la mayoría de los casos irreal, esperar que tales organizaciones desarrollarán programas detallados de métricas de software. Sin embargo, es razonable sugerir que las organizaciones de software de todos los tamaños miden y luego emplean las métricas resultantes para ayudar a mejorar sus procesos de software locales y la calidad y la puntualidad de los productos que elaboran.

Un enfoque de sentido común respecto de la implementación de cualquier actividad relacionada con el proceso de software es mantenerlo simple, personalizarlo para satisfacer las necesidades locales y asegurarse de que agrega valor. En los párrafos

¿Cómo se obtiene un conjunto de métricas de software "simple"?

que siguen se examina cómo estos lineamientos se relacionan con las métricas para pequeños negocios.[11]

"Mantenerlo simple" es una directriz que funciona razonablemente bien en muchas actividades. Pero, ¿cómo se deduce un conjunto "simple" de métricas de software que aun así proporcione valor, y cómo se garantiza que dichas métricas simples satisfarán las necesidades de una organización de software particular? Un buen comienzo consiste en enfocarse no sobre las mediciones sino más bien sobre los resultados. Al grupo de software se le entrevista para definir un objetivo sencillo que requiere mejora. Por ejemplo, "reducir el tiempo para evaluar e implementar los cambios solicitados". Una organización pequeña puede seleccionar el siguiente conjunto de medidas que se recopilan con facilidad:

- Tiempo (horas o días) transcurrido desde el momento en que se hizo una solicitud hasta que la evaluación está completa, t_{cola}.
- Esfuerzo (persona-horas) para realizar la evaluación, T_{eval}.
- Tiempo (horas o días) transcurrido desde que se completa la evaluación hasta la asignación del pedido de cambio al personal, t_{eval}.
- Esfuerzo (persona-horas) requerido para hacer el cambio, T_{cambio}.
- Tiempo requerido (horas o días) para hacer el cambio, t_{cambio}.
- Errores descubiertos durante el trabajo para hacer el cambio, E_{cambio}.
- Defectos descubiertos después de que el cambio es liberado a la base de clientes, D_{cambio}.

Una vez que se han recopilado dichas medidas para varios cambios solicitados es posible calcular el tiempo transcurrido total promedio desde que el cambio se solicitó hasta la implementación del cambio y el porcentaje del tiempo transcurrido absorbido por la cola de espera inicial, la evaluación y la asignación y la implementación del cambio. De igual modo, es posible determinar el porcentaje del esfuerzo que requieren la evaluación y la implementación. Estas métricas se evalúan en el contexto de los datos de calidad, E_{cambio} y D_{cambio}. Los porcentajes proporcionan conocimiento detallado de dónde se vuelve lento el proceso de solicitud de cambio y conduce a pasos de mejora del proceso para reducir t_{cola}, T_{eval}, t_{eval}, T_{cambio} o E_{cambio}. Además, la eficacia en la de eliminación de defectos se puede calcular como

$$\text{EED} = E_{cambio}/(E_{cambio} + D_{cambio})$$

EED se compara con el tiempo transcurrido y el esfuerzo total para determinar el impacto de las actividades de aseguramiento de la calidad en el tiempo y el esfuerzo requeridos para realizar un cambio.

11 Esta exposición es igualmente relevante para los equipos de software que han adoptado un proceso de desarrollo de software ágil (capítulo 4).

22.6 ESTABLECIMIENTO DE UN PROGRAMA DE MÉTRICAS DE SOFTWARE

El Software Engineering Institute (SEI) ha elaborado una guía detallada [PAR96] para establecer un programa de métricas de software "dirigido por metas". La guía sugiere los siguientes pasos:

Referencia Web
A Guide for Goal Driven Software Measurement se puede descargar de: www.sei.cmu.edu.

1. Identificar los objetivos de la empresa.
2. Identificar lo que se quiere conocer o aprender
3. Identificar los subobjetivos.
4. Identificar las entidades y atributos relacionados con los objetivos secundarios
5. Formalizar los objetivos de la medición.
6. Identificar preguntas cuantificables y los indicadores relacionados que se emplearán como apoyo para lograr los objetivos de sus mediciones.
7. Identificar los elementos de datos que se recopilarán para construir los indicadores que ayudarán a responder las preguntas.
8. Definir las medidas que se emplearán y hacer que estas definiciones sean operativas.
9. Identificar las acciones que se tomarán para implementar las medidas
10. Preparar un plan para implementar las medidas.

Una exposición detallada de estos pasos mejor se deja para el libro del SEI. Sin embargo, bien vale la pena dar un breve vistazo a los puntos clave.

PUNTO CLAVE
Las métricas de software que se elijan deben estar basadas en las metas de negocios y técnicas que se desean lograr

Puesto que las funciones comerciales de apoyo del software diferencian los sistemas o productos basados en computadora, o actúan como un producto en sí mismo, las metas definidas para las empresas casi siempre pueden seguirse hacia abajo hasta metas específicas al nivel de la ingeniería del software. Por ejemplo, considérese una compañía que fabrica avanzados sistemas de seguridad para el hogar que tiene contenido de software sustancial. Al trabajar como equipo, la ingeniería del software y los gestores del negocio pueden confeccionar una lista de metas priorizadas del negocio:

1. Mejorar la satisfacción de los clientes con los productos.
2. Hacer que los productos sean más fáciles de usar.
3. Reducir el tiempo que toma poner un producto en el mercado
4. Simplificar el soporte para los productos.
5. Mejorar la obtención global de utilidades.

La organización de software examina cada meta de negocios y pregunta: ¿Qué actividades se gestionan o ejecutan y qué se quiere mejorar de dichas actividades? Para responder estas preguntas el SEI recomienda la creación de una "lista entidad-pregunta" en la que se anoten todas las cosas (entidades) dentro del proceso de softwa-

re que se gestionan o en las que influye la organización de software. Los ejemplos de entidades incluyen recursos de desarrollo, productos de trabajo, código fuente, casos de prueba, solicitudes de cambio, tareas de ingeniería del software y planificaciones. Para cada entidad en la lista el personal de software desarrolla un conjunto de preguntas que evalúan características cuantitativas de la entidad (por ejemplo, tamaño, costo, tiempo de desarrollo). Las preguntas que se derivan de la creación de una lista entidad-pregunta conducen a la derivación de un conjunto de subobjetivos que se relacionan directamente con las entidades creadas y las actividades realizadas como parte del proceso del software.

Considérese la cuarta meta: "Simplificar el soporte para los productos". Para esta meta se puede derivar la siguiente lista de preguntas [PAR96]:

- ¿La solicitud de cambio del cliente contiene la información requerida para evaluar adecuadamente el cambio y luego implementarlo en una forma oportuna?
- ¿Cuán grande es el registro de petición de cambio?
- ¿El tiempo de respuesta para fijar los *bugs* es aceptable con base en las necesidades del cliente?
- ¿Se sigue el proceso de control de cambios (capítulo 27)?
- ¿Los cambios de alta prioridad se implementan en forma oportuna?

Con base en estas preguntas la organización de software puede deducir el siguiente subobjetivo: *mejorar el desempeño del proceso de gestión de cambio*. Se identifican las entidades y los atributos del proceso de software relevantes respecto del subobjetivo, y se delinean las metas de medición asociadas con ambos elementos

El SEI [PAR96] proporciona una guía detallada para los pasos 6 al 10 de su enfoque de medición orientado a objetivos. En esencia, se aplica un proceso de refinamiento paso a paso en el que los objetivos se refinan en preguntas que posteriormente se refinan en entidades y atributos que entonces se refinan en métricas.

INFORMACIÓN

Establecimiento de un programa de métricas

El Software Productivity Center (www.spc.ca) sugiere un enfoque de ocho pasos para establecer un programa de métricas dentro de una organización de software y que se puede emplear como alternativa al enfoque del SEI descrito en la sección 22.6 enfoque se resume a continuación.

1. Entender el proceso de software existente
   - Se definen las actividades del marco de trabajo (capítulo 2)
   - Se describe la información de entrada para cada actividad
   - Se definen las tareas asociadas con cada actividad
   - Se anotan las funciones de aseguramiento de la calidad
   - Se hace una lista con los productos de trabajo producidos.
2. Definir los objetivos que se lograrán mediante el establecimiento de un programa de métricas.
   - Ejemplos: mejorar la precisión de la estimación, mejorar la calidad del producto

3. Identificar las métricas requeridas para lograr los objetivos
   Se definen las preguntas que deben responderse; por ejemplo, ¿cuántos errores encontrados en una actividad de marco de trabajo pueden seguirse hasta la actividad del marco de trabajo precedente?
   Crear medidas y métricas que serán recopiladas y calculadas.
4. Identificar las medidas y métricas que serán recopiladas y calculadas.
5. Establecer un proceso de recopilación de medidas respondiendo las siguientes preguntas:
   ¿Cuál es la fuente de las mediciones?
   ¿Las herramientas se pueden emplear en la recopilación de los datos?
   ¿Quién es responsable de la recopilación de datos?
   ¿Cuándo se recopilan y registran los datos?
   ¿Cómo se almacenan los datos?
   ¿Qué mecanismos de validación se aplican para garantizar que los datos sean correctos?
6. Adquirir herramientas adecuadas para apoyar la recopilación y evaluación.
7. Establecer una base de datos de métricas.
   Se establece la complejidad relativa de la base de datos.
   Se explora el empleo de herramientas relacionadas (por ejemplo, un almacén SCM, capítulo 27).
   Se evalúan los productos de base de datos existentes.
8. Definir mecanismos de realimentación adecuados
   ¿Quién requiere información de métricas en marcha?
   ¿Cómo se entregará la información?
   ¿Cuál es el formato de la información?

Una descripción considerablemente más detallada de estos ochos pasos se puede descargar de http://www.spc.ca/resources/metrics/.

22.7 RESUMEN

Las mediciones permiten que los gestores y profesionales mejoren el proceso del software; auxilien en la planificación, seguimiento y control de un proyecto de software; y evalúen la calidad del producto (software) que se produce. Las medidas de atributos específicos del proceso, proyecto y producto se utilizan para calcular métricas de software. Dichas métricas se pueden analizar para ofrecer indicadores que guíen las acciones de gestión y técnica.

Las métricas del proceso permiten que una organización adopte una visión estratégica al proporcionar información detallada de la eficacia de un proceso de software. Las métricas de proyecto son tácticas; permiten que un gestor de proyecto adapte el flujo de trabajo del proyecto y un enfoque técnico realista en términos de tiempo.

Las métricas orientadas tanto al tamaño como a la función se aplican a lo largo de toda la industria. Las métricas orientadas al tamaño emplean la línea de código como un factor de normalización para otras medidas como persona-mes o defectos. El punto de función se deduce de las medidas del dominio de la información y de una valoración subjetiva de la complejidad del problema. Además, se pueden utilizar las métricas orientadas a objetos y las métricas de aplicación Web.

Las métricas de calidad del software, como las métricas de productividad, se enfocan sobre el proceso, el proyecto y el producto. Al desarrollar y analizar una línea

base de métricas para la calidad, una organización puede corregir aquellas áreas del proceso de software que causan defectos de software

La medición resulta en cambios culturales La recopilación de datos, el cálculo y el análisis de métricas son los tres pasos que preciso implementar para comenzar un programa de métricas. En general, un enfoque orientado a objetivos ayuda a que una organización se enfoque en las métricas correctas para su negocio Al crear una línea base de métricas —una base de datos que contiene mediciones de proceso y producto— los ingenieros de software y sus gestores pueden comprender mejor el trabajo que hacen y el producto que elaboran

REFERENCIAS

[ALB83] Albrecht, A. J. y J E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction. A Software Science Validation", en *IEEE Trans Software Engineering*, noviembre de 1983, pp. 639-648

[BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.

[GRA87] Grady, R B y D L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987

[GRA92] Grady, R. G , *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, 1992.

[GIL88] Gilb, T., *Principles of Software Project Management*, Addison-Wesley, 1988.

[HET93] Hetzel, W., *Making Software Measurement Work*, QED Publishing Group, 1993

[HUM95] Humphrey, W , *A Discipline for Software Engineering*, Addison-Wesley, 1995.

[IEE93] *IEEE Software Engineering Standards*, Standard 610.12-1990, pp. 47-48

[JON86] Jones, C., *Programming Productivity*, McGraw-Hill, 1986

[JON91] Jones, C., *Applied Software Measurement*, McGraw-Hill, 1991.

[JON98] Jones, C., *Estimating Software Costs*, McGraw-Hill, 1998

[LOR94] Lorenz, M. y J. Kidd, *Object-Oriented Software Metrics*, Prentice-Hall, 1994.

[PAR96] Park, R. E., W. B. Goethert y W A Florac, *Goal Driven Software Measurement- -A Guidebook*, CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, agosto de 1996

[PAU94] Paulish, D. y A Carleton, "Case Studies of Software Process Improvement Measurement", en *Computer*, vol. 27, núm 9, septiembre de 1994, pp. 50-57

[QSM02] "QSM Function Point Language Gearing Factors", Versión 2.0, Quantitative Software Management, 2002, http://www.qsm.com/FPGearing.html.

[RAG95] Ragland, B., "Measure, Metric or Indicator. What's the Difference?", en *Crosstalk*, vol 8. núm. 3, marzo de 1995, pp. 29-30

[SMI99] Smith, J., "The Estimation of Effort Based on Use-Cases", un artículo de Rational Corporation, 1999, se puede descargar de http://www.rational.com/products/rup/whitepapers.jsp

PROBLEMAS Y PUNTOS A CONSIDERAR

22.1. Describir con palabras propias la diferencia entre métricas del proceso y del proyecto.

22.2. ¿Por qué algunas métricas de software deben conservarse "privadas"? Ofrecer ejemplos de tres métricas que deban ser privadas Ofrecer ejemplos de tres métricas que deban ser públicas.

22.3. ¿Qué es una medida indirecta y por qué tales medidas son comunes en el trabajo de métricas del software?

22.4. Grady sugiere un conjunto de reglas de etiqueta para las métricas de software. ¿El lector puede agregar tres reglas más a las mencionadas en la sección 22 1 1?

22.5 El equipo A encontró 342 errores durante el proceso de ingeniería del software previo a la liberación. El equipo B encontró 184 errores. ¿Qué medidas adicionales tendrían que realizar los proyectos A y B para determinar cuál de los equipos eliminó los errores de manera más eficiente? ¿Qué métricas podrían proponerse para ayudar a realizar la determinación? ¿Qué datos históricos pueden ser útiles?

22.6. Presentar un argumento contra las líneas de código como medida para la productividad de software. ¿El caso se sostiene cuando se consideran docenas o cientos de proyectos?

22.7. Calcular el valor de punto de función para un proyecto con las siguientes características de dominio de información:

Número de entradas externas: 32
Número de salidas externas: 60
Número de consultas externas: 24
Número de archivos lógicos internos: 8
Número de archivos de interfaz externos: 2

Suponer que todos los valores de ajuste de complejidad son promedios. Utilizar el algoritmo anotado en el capítulo 15.

22.8. Emplear la tabla presentada en la sección 22.2.3 para elaborar un argumento contra la utilización del lenguaje ensamblador basado en la funcionalidad entregada por enunciado de código. Véase de nuevo la tabla y comentar por qué C++ presentaría una mejor alternativa que C.

22.9. El software utilizado para controlar una fotocopiadora requiere 32 000 LDC de C y 4 200 líneas de Smalltalk. Estimar el número de puntos de función para el software dentro de la copiadora.

22.10. Un equipo de ingeniería Web ha construido una WebApp de comercio electrónico que contiene 145 páginas individuales. De estas páginas 65 son dinámicas; esto es, se generan de manera interna con base en la entrada del usuario final. ¿Cuál es el índice de personalización para esta aplicación?

22.11. Una WebApp y su entorno de soporte no han sido completamente reforzados contra ataques. Los ingenieros Web estiman que la probabilidad de repeler un ataque sólo es del 30 por ciento. El sistema no contiene información sensible o controversial, así que la probabilidad de amenaza es de 25 por ciento. ¿Cuál es la integridad de la WebApp?

22.12. En la conclusión de un proyecto que utilizó el Proceso Unificado (Capítulo 3) se determinó que 30 errores se encontraron durante la fase de elaboración, y 12 errores hallados durante la fase de construcción podían seguirse hasta errores que no fueron descubiertos en la fase de elaboración. ¿Cuál es la EED para estas dos fases?

22.13. Un equipo de software entrega un incremento de software a los usuarios finales. Éstos descubren ocho defectos durante el primer mes de uso. Antes de la entrega, el equipo de software encontró 242 errores durante las revisiones técnicas formales y todas las tareas de prueba. ¿Cuál es la EED global para el proyecto?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La mejora del proceso de software (MPS) ha recibido una cantidad significativa de atención durante las dos décadas pasadas. Dado que la medición y las métricas de software son clave para mejorar exitosamente el proceso de software, muchos libros acerca de la MPS también tratan las métricas. Las fuentes valiosas de información acerca de las métricas de procesos incluyen:

Burr, A. y M. Owen, *Statistical Methods for Software Quality*, International Thomson Publishing, 1996.

El Emam, K. y N. Madhavji (eds.), *Elements of Software Process Assessment and Improvement*, IEEE Computer Society, 1999.

Florac, W. A. y A. D. Carleton, *Measuring the Software Process: Statistical Process Control for Software Process Improvement*, Addison-Wesley, 1999.

Garmus, D. y D. Herron, *Measuring the Software Process: A Practical Guide to Functional Measurements*, Prentice-Hall, 1996.

Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley/Longman, 2000

Kan, S. H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995

McGarry y sus colegas (*Practical Software Measurement*, Addison-Wesley, 2001) presentan consejos a profundidad para valorar el proceso de software. Haug y sus colegas (*Software Process Improvement: Metrics, Measurement, and Process Modeling*, Springer-Verlag, 2001) han editado una colección de artículos valiosos. Florac y Carlton (*Measuring the Software Process*, Addison-Wesley, 1999) y Fenton y Pfleeger (*Software Metrics: A Rigorous and Practical Approach*, revisado, Brooks/Cole Publishers, 1998) tratan cómo se pueden emplear las métricas de software para proporcionar los indicadores necesarios para mejorar el proceso de software.

Putnam y Myers (*Five Core Metrics*, Dorset House, 2003) estudian una base de datos de más de 6 000 proyectos de software para demostrar cómo cinco métricas centrales —tiempo, esfuerzo, tamaño, confiabilidad y productividad de proceso— se pueden emplear para controlar los proyectos de software. Maxwell (*Applied Statistics for Software Managers*, Prentice-Hall, 2003) presenta técnicas para analizar datos del proyecto de software. Munson (*Software Engineering Measurement*, Auerbach, 2003) estudia un amplio abanico de temas de medición de ingeniería del software. Jones (*Software Assessments, Benchmarks and Best Practices*, Addison-Wesley, 2000) describe tanto medidas cuantitativas como factores cualitativos que ayudan a una organización a valorar sus procesos y prácticas de software. Garmus y Herron (*Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesley, 2000) examinan las métricas de proceso con énfasis en el análisis de los puntos de función.

Lorenze y Kidd [LOR94] y DeChampeax (*Object-Oriented Development Process and Metrics*, Prentice-Hall, 1996) consideran el proceso OO y describen un conjunto de métricas para valorarlo. Whitmire (*Object-Oriented Design Measurement*, Wiley, 1997) y Henderson-Sellers (*Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, 1995) se enfocan en las métricas técnicas para el trabajo OO, pero también consideran medidas y métricas que se pueden aplicar en el ámbito del proceso y del producto.

Se ha publicado relativamente poco acerca de las métricas para el trabajo de ingeniería Web. Sin embargo, Stern (*Web Metrics: Proven Methods for Measuring Web Site Success*, Wiley, 2002), Inan y Kean (*Measuring the Success of Your Website*, Longman, 2002) y Nobles y Grady (*Web Site Analysis and Reporting*, Premier Press, 2001) abordan las métricas Web desde una perspectiva de negocios y de marketing.

Lo más reciente en la investigación en el área de métricas lo resume el IEEE (*Symposium on Software Metrics*, publicado anualmente). En Internet está disponible una amplia variedad de fuentes de información acerca de las métricas de proceso y proyecto. Una lista actualizada de referencias en la World Wide Web se encuentra en el sitio Web SEPA: **http://www.mhhe.com/pressman**.

CAPÍTULO

23 ESTIMACIÓN PARA PROYECTOS DE SOFTWARE

CONCEPTOS CLAVE

La gestión del proyecto de software comienza con un conjunto de activid- des que en grupo se denominan *planificación del proyecto*. Antes de que el proyecto comience el gestor del proyecto y el equipo de software deben e- timar el trabajo que habrá de realizarse, los recursos que se requerirán y el tiemp que transcurrirá desde el principio hasta el final. Una vez que se completen es actividades, el equipo de software debe establecer un plan del proyecto que de fina las tareas y fechas clave de la ingeniería del software, que identifique qui- es responsable de dirigir cada tarea y especifique las dependencias entre tare- que pueden ser determinantes en el progreso.

En una excelente guía para "sobrevivir el proyecto de software", Steve McConn [MCC98] presenta una visión del mundo real de la planificación del proyecto·

> Muchos trabajadores técnicos preferirán realizar el trabajo técnico en lugar de pa sar el tiempo en la planificación. Muchos gestores técnicos no tienen suficiente en- trenamiento en la gestión técnica para sentirse seguros de que su planificación mejorará el resultado de un proyecto. Puesto que ninguna parte quiere hacer la pla nificación, con frecuencia no se realiza.
>
> Pero las fallas para planificar es uno de los mayores errores que un proyecto pueda cometer... se necesita la planificación eficaz para resolver los problemas corriente arriba [temprano en el proyecto] a bajo costo, más que corriente abajo [tarde en el proyecto] a alto costo. El proyecto promedio gasta *80 por ciento* de su tiempo en re- laboración· corrigiendo errores que se cometieron en etapas tempranas del proyecto.

UN VISTAZO RÁPIDO

¿Qué es? Se ha establecido una necesidad real para el software; los participantes están a bordo; los inge- nieros de software están listos para comenzar; y el proyecto está a punto de arrancar. Pero, ¿cómo se procederá? La pla- nificación del proyecto de software abarca cinco grandes actividades: estimación, programa de trabajo, análisis de riesgos, planificación de la gestión de la calidad y planificación de la ges- tión del cambio. En el contexto de este capítulo sólo se considera la estimación: su intento por determinar cuánto dinero, esfuerzo, recursos y tiempo tomará construir un sistema o producto específico basado en software.

¿Quién lo hace? Los gestores de proyecto del software, con base en la información solicitada de los participantes e ingenieros de software , los datos de las métricas de software recopilados en proyectos previos.

¿Por qué es importante? ¿Se construiría una casa sin saber cuánto dinero está a punto de gastarse, las tareas que se deben realizar y el tiempo para que el trabajo se haga? Desde luego que no, y puesto que la mayoría de los siste- mas y productos basados en computadora son con- siderablemente más caros que construir una gran casa, parecería razonable desarrollar una estimación antes de comenzar a crear el software.

¿Cuáles son los pasos? La estimación comien za con una descripción del ámbito del producto. Entonces el problema se descompone en un con- junto de problemas más pequeños, y cada uno de éstos se estima empleando datos históricos y

...periencia como guías. La complejidad y el ...esgo del problema se consideran antes de rea... ...lizar una estimación final.

¿Cuál es el producto obtenido? Se genera una simple tabla en la que se delinean las tareas que deben realizarse, las funciones que habrán de implementarse y el costo, esfuerzo y tiempo involucrados para cada uno.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Eso es difícil, porque en realidad no se sabrá sino hasta que el proyecto se haya completado. Sin embargo, si se tiene experiencia y se sigue un enfoque sistemático, se generan estimaciones empleando datos históricos sólidos, se crean puntos de datos de estimación mediante al menos dos métodos diferentes, se establece un calendario realista y continuamente se adapta conforme el proyecto avanza, se puede estar seguro de que se está haciendo lo mejor.

McConell argumenta que cualquier proyecto puede encontrar el tiempo para planificar (y adaptar el plan a lo largo del proyecto) simplemente tomando un pequeño porcentaje del tiempo que se habría gastado en la reelaboración que ocurre debido a que no se planificó.

23.1 Observaciones acerca de la estimación

La planificación requiere que los gestores técnicos y los miembros del equipo de software establezcan un compromiso inicial, aun cuando sea probable que este "compromiso" pruebe estar equivocado. Siempre que se realizan estimaciones se atisba al futuro y se acepta automáticamente algún grado de incertidumbre. Para citar a Frederick Brooks [BRO75]:

> [N]uestras técnicas de estimación están pobremente desarrolladas. Más seriamente, reflejan una suposición no expresada que es bastante incierta, es decir: que todo irá bien. Puesto que no estamos seguros de nuestras estimaciones, con frecuencia los gestores de software carecen de la cortés testarudez para hacer que la gente espere un buen producto.

Aunque la estimación es tanto un arte como una ciencia, esta importante actividad no necesita realizarse en una forma improvisada. Existen técnicas útiles para la estimación de tiempo y esfuerzo. Las métricas del proceso y del proyecto ofrecen la perspectiva histórica y la energía para la generación de estimaciones cuantitativas. La experiencia (de toda la gente involucrada) puede auxiliar enormemente conforme se desarrollan y revisan las estimaciones. Puesto que la estimación coloca los cimientos para las demás actividades de planificación del proyecto, y ésta proporciona la ruta para la ingeniería del software exitosa, se estaría mal aconsejado si se embarcara sin ella.

> "Los buenos enfoques de estimación y los datos históricos sólidos ofrecen la mejor esperanza de que en realidad se triunfará sobre demandas imposibles."
>
> **Capers Jones**

La estimación de recursos, costo y programa de trabajo para una tarea de ingeniería de software requiere experiencia, acceso a buena información histórica (métricas) y el valor para comprometerse con predicciones cuantitativas cuando la información cualitativa es todo lo que existe. La estimación implica riesgo inherente,[1] y éste conduce a la incertidumbre.

La disponibilidad de información histórica tiene una fuerte influencia en el riesgo de la estimación. Al mirar en retrospectiva, se pueden emular las cosas que funcionaron y mejorar las áreas donde surgieron problemas. Cuando hay disponibles amplias métricas de software (capítulo 22) de proyectos previos, las estimaciones se hacen con mayor seguridad, los programas de trabajo se pueden establecer para evitar dificultades pasadas y el riesgo global se reduce.

> "La característica de una mente instruida es descansar satisfecha con el grado de precisión que la naturaleza del asunto admite, y no buscar la exactitud cuando sólo es posible una aproximación de la verdad."
>
> **Aristóteles**

El riesgo de la estimación se mide por el grado de incertidumbre en las estimaciones cuantitativas establecidas para recursos, costos y programa de trabajo. Si el ámbito del proyecto se comprende en forma deficiente o los requisitos del proyecto están sujetos a eventuales cambios, la incertidumbre y el riesgo de la estimación se incrementan peligrosamente. El planificador y, en forma más importante, el cliente deben reconocer que la variabilidad en los requisitos del software significa inestabilidad en costo y programa de trabajo.

Sin embargo, un gestor de proyecto no debe obsesionarse con las estimaciones. Los modernos enfoques de ingeniería del software (por ejemplo, modelos de proceso incremental) asumen una visión iterativa del desarrollo. En tales enfoques es posible, aunque no siempre aceptable políticamente, reexaminar las estimaciones (cuando se conozca más información) y modificarlas cuando el cliente cambia los requisitos.

23.2 EL PROCESO DE PLANIFICACIÓN DEL PROYECTO

CONSEJO

Mientras más conozca, mejor estimará. En consecuencia, actualice sus estimaciones conforme avance el proyecto.

El objetivo de la planificación del proyecto de software es proporcionar un marco de trabajo que permita al gestor estimar razonablemente recursos, costo y programa de trabajo. Además, las estimaciones deben intentar definir los escenarios de mejor y peor caso de modo que los resultados del proyecto se puedan acotar. Aunque existe un grado inherente de incertidumbre, el equipo de software se embarca en un plan establecido como consecuencia de las tareas de planificación. Por lo tanto, el plan se debe adaptar y actualizar conforme avance el proyecto. En las secciones siguientes se estudiará cada una de las actividades asociadas con la planificación del proyecto de software.

1 En el capítulo 25 se presentan técnicas sistemáticas para el análisis del riesgo.

CONJUNTO DE TAREAS

Conjunto de tareas para la planificación del proyecto

1. Establecer el ámbito del proyecto.
2. Determinar la factibilidad
3. Analizar los riesgos (capítulo 25).
4. Definir los recursos requeridos.
   a. Determinar los recursos humanos requeridos.
   b. Definir los recursos de software reutilizables.
   c. Identificar los recursos del entorno.
5. Estimar costo y esfuerzo.
   a. Descomponer el problema.
   b. Desarrollar dos o más estimaciones empleando tamaño, puntos de función, tareas de proceso o casos de uso.
   c. Reconciliar las estimaciones.
6. Desarrollar un plan del proyecto (capítulo 24).
   a. Establecer un conjunto de tareas significativo.
   b. Definir una red de tareas
   c. Usar herramientas de planificación para desarrollar un cronograma
   d. Definir mecanismos de seguimiento del programa de trabajo.

23.3 ÁMBITO DEL SOFTWARE Y FACTIBILIDAD

El *ámbito del software* describe las funciones y características que se entregarán a los usuarios finales, los datos que son entrada y salida, el "contenido" que se presenta a los usuarios como consecuencia de emplear el software, así como el desempeño, las restricciones, las interfases y la confiabilidad que *acotan* el sistema. El ámbito se define al usar una de las dos técnicas siguientes:

1. Después de una comunicación con todos los participantes se desarrolla una descripción narrativa del ámbito del software.
2. Los usuarios finales desarrollan un conjunto de casos de uso.[2]

Las funciones descritas en el enunciado del ámbito (o dentro de los casos de uso) se evalúan y en algunos casos se refinan para proporcionar más detalles antes de comenzar la estimación. Puesto que las estimaciones de costo y programa de trabajo están funcionalmente orientadas, con frecuencia es útil cierto grado de descomposición Las consideraciones del desempeño abarcan los requisitos de procesamiento y tiempo de respuesta Las restricciones identifican los límites colocados en el software por el hardware externo, la memoria disponible u otros sistemas existentes

Una vez identificado el ámbito (con la participación del cliente) es razonable preguntar: ¿es posible construir software para satisfacer este ámbito? ¿El proyecto es factible? Con mucha frecuencia los ingenieros de software soslayan estas preguntas (o gestores o clientes impacientes los presionan para hacerlo), sólo para verse enredados en un proyecto condenado al fracaso. Putnam y Myers [PUT97a] abordan este conflicto cuando escriben:

2 Los casos de uso se discutieron con detalle en las partes 2 y 3 de este libro Un caso de uso es una descripción basada en escenario de la interacción del usuario con el software, desde el punto de vista del usuario

CONSEJO

La factibilidad del proyecto es importante, pero una consideración de las necesidades del negocio es incluso más importante. No es bueno construir un sistema o producto de alta tecnología que nadie quiere.

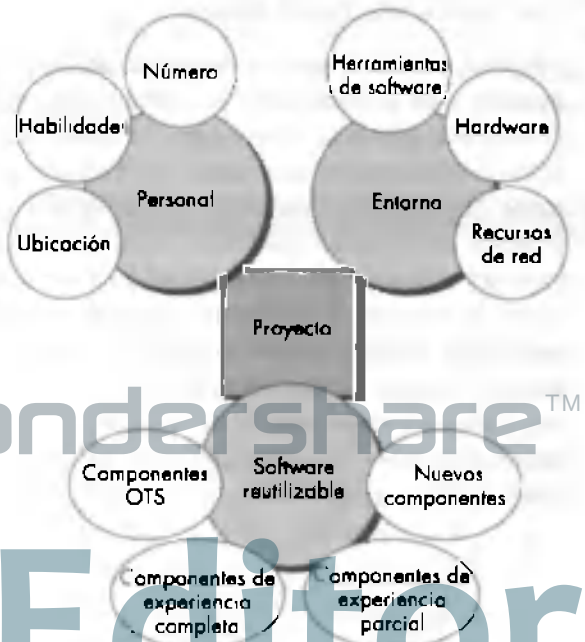
[N]o todo lo imaginable es factible, incluso ni en software, tan evanescente como puede parecer a los extraños. Por el contrario, la factibilidad del software tiene cuatro dimensiones sólidas: *Tecnología*: ¿el proyecto es técnicamente factible? ¿Está dentro del terreno de la disciplina? ¿Los defectos se pueden reducir a tal grado que se emparejen con las necesidades de la aplicación? *Finanzas*: ¿es financieramente factible? ¿Se puede completar el desarrollo a un costo que la organización de software, su cliente o el mercado puedan enfrentar? *Tiempo*: ¿el proyecto llegará al mercado antes y vencerá a la competencia? *Recursos*: ¿la organización tiene los recursos necesarios para triunfar?

Putnam y Myers sugieren, acertadamente, que el ámbito no es suficiente. Una vez que el ámbito se comprende, el equipo de software y otros deben trabajar para determinar si se puede hacer dentro de las dimensiones anotadas líneas arriba. Ésta es una parte crucial, aunque con frecuencia ignorada, del proceso de estimación.

23.4 RECURSOS

La segunda tarea de la planificación es la estimación de los recursos necesarios para completar el esfuerzo de desarrollo del software. La figura 23.1 muestra las tres grandes categorías de los recursos de ingeniería del software: personal, componentes de software reutilizables y el entorno de desarrollo (hardware y herramientas de software). Cada recurso se especifica con cuatro características: descripción de recurso, un informe de disponibilidad; cuándo se requerirá el recurso; tiempo durante el cual el recurso se aplicará. Las últimas dos características se pueden ver como

FIGURA 23.1
Recursos del proyecto.



una *ventana de tiempo*. La disponibilidad del recurso para una ventana específica debe establecerse lo más pronto posible

23.4.1 Recursos humanos

El planificador comienza evaluando el ámbito del software y seleccionando las habilidades requeridas para completar el desarrollo. Se especifican tanto la posición organizacional (por ejemplo, gestor, ingeniero de software ejecutivo) como la especialidad (por ejemplo, telecomunicaciones, base de datos, cliente/servidor). En proyectos relativamente pequeños (unos pocos persona-meses) un solo individuo puede realizar todas las tareas de ingeniería del software y consultar con especialistas conforme se requiera. En proyectos mayores el equipo de software puede estar geográficamente disperso en varios sitios. Aquí se especifica la ubicación de cada recurso humano.

El número de personas que requiere un proyecto de software sólo se determina después de que se ha hecho una estimación del esfuerzo de desarrollo (por ejemplo, personas-mes). Las técnicas para estimar el esfuerzo se tratarán más adelante en este capítulo.

23.4.2 Recursos de software reutilizables

La ingeniería del software basada en componentes (capítulo 30) enfatiza la reutilización; es decir, la creación y reutilización de bloques de construcción de software [HOO91]. Tales bloques, usualmente llamados *componentes*, deben catalogarse para consultarlos con facilidad, estandarizarse para facilitar su aplicación y validarse para integrarlos fácilmente.

Bennatan [BEN92] sugiere cuatro categorías de recursos de software que deben considerarse conforme avanza la planificación:

CONSEJO

...a olvide que ...r varios compo... ...s de reutilización ...e ser un reto ...nte. El ...o de la inte... ...con frecuencia ...e conforme se ...r varios ...ponentes.

Componentes ya desarrollados. El software existente se puede adquirir de un tercero o se desarrolló internamente para un proyecto previo. Los CCYD (componentes comerciales ya desarrollados) se compran de un tercero, están listos para emplearlos en el proyecto actual y han sido ampliamente validados.

Componentes experimentados. Especificaciones, diseños, código o datos de prueba existentes que se desarrollaron para proyectos previos son similares al software que se construirá para el proyecto actual. Los miembros del equipo de software actual ya tienen experiencia en el área de aplicación que representan dichos componentes. En consecuencia, las modificaciones que requieran los componentes experimentados serán relativamente de bajo riesgo.

Componentes de experiencia parcial. Especificaciones, diseños, código o datos de prueba existentes que se desarrollaron para proyectos previos están relacionados con el software que se construirá para el proyecto actual pero requerirán modificaciones sustanciales. Los miembros del equipo de software actual sólo tienen experiencia limitada en el área de aplicación que representan dichos componentes. Por

lo tanto, las modificaciones que requieren los componentes de experiencia parcial tienen un grado considerable de riesgo.

Componentes nuevos. El equipo de software debe construir los componentes de software especificamente para las necesidades del proyecto actual.

Irónicamente, con frecuencia los componentes de software reutilizables son despreciados durante la planificación, sólo para convertirse en una preocupación importante durante la fase de desarrollo del proceso de software. Es mejor especificar cuanto antes los requisitos de recursos de software. De esta forma se puede llevar a cabo la evaluación técnica de las alternativas y puede ocurrir la adquisición oportuna.

23.4.3 Recursos del entorno

El entorno que soporta un proyecto de software, con frecuencia denominado *entorno de ingeniería del software* (EIS), incorpora hardware y software. El hardware proporciona una plataforma que soporta las herramientas (software) con que se producen los productos de trabajo basados en una buena práctica de la ingeniería del software.[3] Puesto que la mayor parte de las organizaciones de software tienen múltiples constituyentes que requieren acceso al EIS, el planificador de proyecto debe prescribir la ventana de tiempo requerida por el hardware y el software, y verificar que estos recursos estarán disponibles.

Cuando un sistema basado en computadora (que incorpora hardware y software especializados) se someterá a ingeniería, el equipo de software quizá requiera acceso a elementos de hardware que están desarrollando otros equipos de ingeniería. Por ejemplo, el software de un contador numérico (CN) utilizado en un tipo de máquinas-herramienta tal vez requiera una máquina-herramienta específica (por ejemplo, un CN de torno) como parte del paso de prueba de validación; un proyecto de software para plantilla de página avanzada quizá necesite una impresora de alta calidad en algún punto durante el desarrollo. El planificador del proyecto de software debe especificar cada elemento de hardware.

23.5 Estimación de proyectos de software

El software es el elemento más caro de virtualmente todos los sistemas basados en computadora. En sistemas complejos, personalizados, un gran error en la estimación del costo puede hacer la diferencia entre beneficio y pérdida. Rebasar el costo puede ser desastroso para el desarrollador.

> "En una era de subcontratación y competencia creciente, la habilidad para estimar con mayor precisión... ha surgido como un factor de éxito crucial para muchos grupos de TI."
>
> **Rob Thomsett**

3 Otro hardware —el entorno objetivo— es la computadora en la que el software se ejecutará cuando haya sido liberado al usuario final.

La estimación del costo y el esfuerzo nunca será una ciencia exacta.[4] Demasiadas variables —humanas, técnicas, ambientales, políticas— pueden afectar el costo final del software y el esfuerzo aplicado a desarrollarlo. Sin embargo, la estimación del proyecto de software se puede transformar de una práctica oscura en una serie de pasos sistemáticos que proporcionan estimaciones con riesgo aceptable. Para lograr estimaciones confiables de costo y esfuerzo se tienen varias opciones:

1. Demorar la estimación hasta más tarde en el proyecto (obviamente, ¡se puede lograr 100 por ciento de estimaciones precisas después de que el proyecto esté terminado!)
2. Basar las estimaciones en proyectos similares que ya hayan sido completados.
3. Emplear técnicas de descomposición relativamente simples para generar estimaciones de costo y esfuerzo del proyecto.
4. Utilizar uno o más modelos empíricos en la estimación de costo y esfuerzo.

Desgraciadamente, la primera opción, aunque atractiva, no es práctica. Las estimaciones de costos se tienen que proporcionar "por adelantado". No obstante, se debe reconocer que, mientras más se espere más se conocerá, y mientras más se conozca menos probable es que se cometan serios errores en las estimaciones.

La segunda opción puede funcionar razonablemente bien si el proyecto en curso es muy similar a los previos y a otras influencias del proyecto (por ejemplo, el equipo de software, el cliente, las condiciones del mercado, el EIS, las fechas límite) son aproximadamente equivalentes. Por desgracia, la experiencia no siempre ha sido un buen indicador de los resultados futuros

Las opciones restantes son enfoques viables para la estimación del proyecto de software. Idealmente, las técnicas mencionadas para cada opción deben aplicarse juntas, cada una empleada como una marca de verificación para la otra. Las técnicas de descomposición asumen un enfoque de "divide y vencerás" respecto de la estimación del proyecto de software. Al descomponer un proyecto en funciones principales y actividades de ingeniería del software relacionadas, las estimaciones de costo y esfuerzo se pueden realizar en forma escalonada. Los modelos de estimación empírica son útiles para complementar las técnicas de descomposición y ofrecer un enfoque de estimación potencialmente valioso por su propio derecho. Estos modelos se estudian en la sección 23.7.

Cada una de las opciones viables de estimación de costo del software será tan buena como lo sean los datos históricos en que se basa la estimación. Si no existen datos históricos, la estimación del costo se basará en un fundamento muy tambaleante. En el capítulo 22 se examinan las características de algunas de las métricas de software que proporcionan la base para los datos históricos de estimación

4 Bennatan [BEN03] reporta que 40 por ciento de los desarrolladores de software continúan luchando con las estimaciones y que el tamaño del software y el tiempo de desarrollo son muy difíciles de estimar con precisión.

23.6 TÉCNICAS DE DESCOMPOSICIÓN

La estimación del proyecto de software es una forma de resolver problemas; en la mayoría de los casos, el problema que debe resolverse (es decir, el desarrollo de una estimación de costo y esfuerzo para un proyecto de software) es muy complejo como para considerarlo una sola pieza. Por esta razón se descompone el problema, recaracterizándolo como un conjunto de problemas más pequeños (y, esperanzadoramente, más manejable)

En el capítulo 21 se examinó el enfoque de descomposición desde dos diferentes puntos de vista: descomposición del problema y descomposición del proceso. La estimación emplea una o ambas formas de partición. Pero antes de que pueda realizarse una estimación, el planificador del proyecto debe entender el ámbito del software que se construirá y generar una estimación de su "tamaño".

23.6.1 Tamaño del software

PUNTO CLAVE

El "tamaño" del software que se construirá puede estimarse empleando una medida directa, LDC, o una indirecta, PF.

La precisión de la estimación de un proyecto de software se manifiesta en varios factores: 1) el grado con el cual el planificador ha estimado adecuadamente el tamaño del producto que se construirá; 2) la habilidad para traducir la estimación del tamaño en esfuerzo humano, programa de trabajo y dinero (una función de la disponibilidad de las métricas de software confiables a partir de proyectos previos), 3) el grado en el cual el plan del proyecto refleja las habilidades del equipo de software, y 4) la estabilidad de los requisitos del producto y el entorno que soporta el esfuerzo de ingeniería del software

En esta sección se considera el problema del *tamaño del software*. Puesto que la estimación de un proyecto sólo es tan buena como la estimación del tamaño del trabajo para realizarlo, el tamaño representa el primer gran desafío del planificador del proyecto. En el contexto de la planificación del proyecto, *tamaño* se refiere a un resultado cuantificable del proyecto de software. Si se asume un enfoque directo, el tamaño se puede medir en líneas de código (LDC). Si se elige un enfoque indirecto, el tamaño se representa como puntos de función (PF).

Putnam y Myers [PUT92] sugieren cuatro diferentes enfoques al problema del tamaño:

¿Cómo se mide el tamaño del software que se planea construir?

- *Tamaño de "lógica difusa".* La aplicación de este enfoque requiere que el planificador identifique el tipo de aplicación, establezca su magnitud en una escala cualitativa y luego refine la magnitud dentro del rango original
- *Tamaño de punto de función.* El planificador desarrolla estimaciones de las características del dominio de la información tratadas en el capítulo 15.
- *Tamaño de componentes estándar.* El software se compone de varios "componentes estándar", los cuales son diferentes y genéricos de una área particular de la aplicación. Por ejemplo, los componentes estándar de un sistema de información son subsistemas, módulos, pantallas, reportes, programas inter-

activos, programas por lotes, archivos, LDC e instrucciones al nivel de objeto. El planificador del proyecto estima el número de ocurrencias de cada componente estándar y luego aplica datos de proyectos históricos para determinar el tamaño de entrega por componente estándar.

- *Tamaño del cambio.* Este enfoque se aplica cuando un proyecto incluye la utilización de software existente que debe modificarse en cierta forma como parte de un proyecto. El planificador estima el número y tipo (por ejemplo, reutilización, código de adición, código de cambio, código de borrado) de las modificaciones que se deben lograr.

Putnam y Myers sugieren que los resultados de cada uno de estos enfoques de tamaño se combinen estadísticamente para crear una estimación de *tres puntos* o del *valor esperado*. Esto se logra desarrollando valores optimistas (bajos), más probables y pesimistas (altos) para el tamaño y combinándolos con la ecuación (23-1) descrita en la siguiente sección.

23.6.2 Estimación basada en el problema

En el capítulo 22, las líneas de código y los puntos de función se describieron como medidas a partir de las cuales se calculan las métricas de productividad. Los datos de las LDC y los PF se utilizan en dos formas al estimar el proyecto de software: 1) como una variable de la estimación para el "tamaño" de cada elemento del software, y 2) como métricas de línea base recolectadas a partir de proyectos previos y utilizados en conjunción con variables de estimación para desarrollar proyecciones de costo y esfuerzo.

¿Qué tienen en común las estimaciones basadas en LDC y PF?

Las estimaciones de LDC y PF son distintas técnicas de estimación, aunque ambas tienen varias características en común. El planificador del proyecto comienza con un enfoque acotado del ámbito del software y a partir de ahí intenta descomponer el software en funciones problema que puedan estimarse individualmente. Entonces se estiman las LDC o PF (las variables de estimación) para cada función. De manera alternativa, el planificador puede elegir otro componente para tamaño, como clases u objetos, cambios o procesos de negocios afectados.

CONSEJO

...o recopile ...icas de producti... ...l para proyectos, ...e esti... ...r una taxonomía ...de los tipos de ...proyecto. Esto le ...permitirá calcular ...métricas específicas ...de dominio, lo que ...le permitirá realizar esti... ...maciones más ...precisas.

Entonces se aplican las métricas de la línea base de productividad (por ejemplo, LDC/pm o PF/pm[5]) a la variable de estimación apropiada, y se deriva el costo o esfuerzo de la función. Las estimaciones de función se combinan para producir una estimación global del proyecto completo.

Sin embargo, es importante notar que con frecuencia existe una dispersión sustancial en las métricas de productividad de una organización, lo que hace sospechoso el uso de una sola métrica de línea base de productividad. En general, el dominio del proyecto debe calcular los promedios de LDC/pm o PF/pm. Es decir, los proyectos deben agruparse por tamaño de equipo, área de aplicación, complejidad y otros

5 Las siglas *pm* significan persona-mes de esfuerzo.

parámetros relevantes. Luego se tienen que calcular los promedios de dominio local. Cuando se estima un nuevo proyecto primero debe ubicarse en un dominio, y luego aplicar el promedio del dominio apropiado para la productividad y así generar la estimación.

Las técnicas de estimación LDC y PF difieren en en cuanto al detalle requerido para descomposición y el objetivo de la partición. Al emplear LDC como variable de estimación la descomposición es absolutamente esencial y con frecuencia se lleva a grados considerables de detalle. Mientras mayor sea el grado de partición es más probable que se desarrolle una estimación razonablemente precisa de LDC.

PUNTO CLAVE

En las estimaciones PF la descomposición se enfoca sobre las características del dominio de información.

En las estimaciones de PF la descomposición funciona de manera diferente. Más que enfocarse sobre la función, se estima cada una de las cinco características de dominio de información, así como los 14 valores de ajuste de complejidad estudiados en el capítulo 15. Entonces se pueden utilizar las estimaciones resultantes para derivar un valor de PF que se pueda ligar a datos previos y empleados para generar una estimación.

Sin importar la variable de estimación que se utilice, el planificador del proyecto comienza estimando una gama de valores para cada función o valor de dominio de información. Al emplear datos históricos o (cuando todo lo demás falla) intuición, el planificador estima un valor de tamaño optimista, más probable, y pesimista para cada función o cuenta para cada valor de dominio de información. Cuando se especifica un rango de valores se proporciona un indicio implícito del grado de incertidumbre.

¿Cómo se calcula el "valor esperado" para el tamaño de software?

Entonces se puede calcular un valor de tres puntos o uno esperado. El *valor esperado* para la variable de estimación (tamaño), S, se calcula como un promedio ponderado de las estimaciones optimista (S_{opt}), más probable (S_m) y pesimista (S_{pes}). Por ejemplo,

$$S = (S_{opt} + 4S_m + S_{pes})/6 \qquad (23-1)$$

brinda la credibilidad más fuerte a la estimación "más probable" y sigue una distribución de probabilidad beta. Se supone que existe una probabilidad muy pequeña de que el tamaño real resultante se ubique fuera de los valores optimista y pesimista.

Una vez determinado el valor esperado para la variable de estimación se aplican los datos de productividad histórica de LDC o PF. ¿Son correctas las estimaciones? La única respuesta razonable a esta pregunta es: no se puede estar seguro. Cualquier técnica de estimación, no importa cuán sofisticada sea, debe contrastarse con otro enfoque. Incluso entonces deben prevalecer el sentido común y la experiencia.

23.6.3 Un ejemplo de estimación basada en LDC

Como ejemplo de técnicas de estimación de LDC y PF basadas en el problema, considérese un paquete de software que será entregado para una aplicación de diseño asistido por computadora (CAD, por sus siglas en inglés) destinado a componentes mecánicos. El software se ejecutará en una estación de trabajo de ingeniería y debe

CONSEJO

Muchas aplicaciones modernas residen en red o son parte de una arquitectura cliente-servidor. Por lo tanto, asegúrese de que sus estimaciones incluyan el esfuerzo necesario para desarrollar software de "infraestructura".

tener interfaz con varios periféricos que incluyen ratón, digitalizador, monitor de color de alta resolución e impresora láser. Se puede elaborar una descripción preliminar del ámbito del software:

> El software CAD mecánico aceptará del ingeniero datos geométricos de dos y tres dimensiones. El ingeniero interactuará y controlará el sistema CAD a través de una interfaz del usuario que exhibirá características de buen diseño de interfaz humano-máquina. Todos los datos geométricos y otra información de soporte se conservarán en una base de datos CAD. Se desarrollarán módulos de análisis de diseño para producir la salida requerida, la cual se desplegará en una diversidad de dispositivos gráficos. El software se diseñará para controlar e interactuar con dispositivos periféricos que incluyen ratón, digitalizador, impresora láser y plotter.

Esta descripción del ámbito es preliminar, no está acotada. Se tendría que expandir cada oración para ofrecer detalle concreto y acotación cuantitativa. Por ejemplo, antes de que comience la estimación, el planificador debe determinar qué significa "características de buen diseño de interfaz humano-máquina" o cuáles serán el tamaño y la complejidad de la "base de datos CAD".

En cuanto a los propósitos actuales, se supone que se ha llevado a cabo más refinamiento y que están identificadas las grandes funciones de software mencionadas en la figura 23.2. Al continuar con la técnica de descomposición para LDC se elabora una tabla de estimación, la cual se muestra en la figura 23.2. En cada función se desarrolla un rango de estimaciones LDC. Por ejemplo, el rango de las estimaciones LDC para la función de análisis geométrico 3D es: optimista, 4 600 LDC; más probable, 6 900 LDC; y pesimista, 8 600 LDC. Al aplicar la ecuación 23-1 el valor esperado para la función de análisis geométrico 3D es 6 800 LDC. Otras estimaciones se derivan en forma similar. Al sumar verticalmente en la columna de LDC estimadas, se establece una estimación de 33 200 líneas de código para el sistema CAD

CONSEJO

No sucumba a la tentación de utilizar este resultado como la estimación de su proyecto. Usted debe derivar otro resultado utilizando un enfoque diferente.

FIGURA 23.2

Tabla de estimación para métodos LDC.

Función	LDC estimadas
Facilidades de interfaz del usuario y control (FIUC)	2 300
Análisis geométrico bidimensional (AG2D)	5 300
Análisis geométrico tridimensional (AG3D)	6 800
Gestión de base de datos (GBD)	3 350
Facilidades de presentación gráfica de computadora (FPGC)	4 950
Función de control periférico (FCP)	2 100
Módulos de análisis de diseño (MAD)	8 400
Líneas de código estimadas	33 200

Una revisión de los datos históricos indica que el promedio organizacional de productividad para sistemas de este tipo es de 620 LDC/pm. Con base en una tarifa laboral de 8 000 dólares por mes, el costo por línea de código es aproximadamente de 13 dólares. Con base en la estimación de LDC y los datos históricos de productividad, el costo total estimado del proyecto es de 431 000 dólares y el esfuerzo estimado es de 54 personas-mes.[6]

HogarSeguro

Estimación

El escenario: La oficina de Doug Miller cuando comienza la planificación del proyecto.

Los actores: Doug Miller (gerente del equipo de ingeniería del software de *HogarSeguro*) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

La conversación:

Doug: Necesitamos desarrollar una estimación del esfuerzo para el proyecto, y luego tenemos que definir un microprograma de trabajo para el primer incremento y un macroprograma de trabajo para los incrementos restantes.

Vinod (asiente con la cabeza): Muy bien, pero no hemos definido ningún incremento todavía.

Doug: Cierto, pero es por eso por lo que necesitamos estimar.

Jamie (malhumorada): ¿Quieres saber cuánto nos tomará?

Doug: He aquí lo que necesito. Primero, necesitamos descomponer funcionalmente el software de *HogarSeguro*... a un nivel elevado. Luego tenemos que estimar el número de líneas de código que tomará cada función. Luego...

Jamie: ¡Vaya! ¿Cómo se supone que lo haremos?

Vinod: Ya lo he hecho en proyectos previos. Utilizas casos de uso, determinas la funcionalidad requerida para implementar cada uno, supones el conteo de LDC para cada pieza de la función. La mejor aproximación es que cada uno lo haga de manera independiente y luego comparamos los resultados.

Doug: O puedes hacer una descomposición funcional de todo el proyecto.

Jamie: Pero eso tomará toda la vida, y ya tenemos que comenzar.

Vinod: No... se puede hacer en pocas horas... esta mañana, de hecho.

Doug: Estoy de acuerdo... no podemos esperar exactitud, sólo una idea aproximada de cuál será el tamaño de *HogarSeguro*.

Jamie: Creo que sólo debemos estimar el esfuerzo... eso es todo.

Doug: También haremos eso. Luego usaremos ambas estimaciones como comprobación mutua.

Vinod: Vamos a hacerlo...

23.6.4 Un ejemplo de estimación basada en PF

La descomposición de la estimación basada en PF se centra en los valores de dominio de información más que en las funciones de software. Al consultar la tabla presentada en la figura 23.3 el planificador del proyecto estima entradas externas, salidas externas, consultas externas, archivos lógicos internos y archivos de interfaz externos para el software CAD. Los PF se calculan usando la técnica estudiada en el capí-

6 Las estimaciones están redondeadas a 1 000 dólares y a la persona-mes más cercanos. Mayor precisión es innecesaria e irreal, dadas las limitaciones de precisión de la estimación.

FIGURA 23.3 Estimación de valores de dominio de información.

Valor de dominio de información	Optim.	Probable	Pesim.	Conteo estim.	Peso	Conteo PF
Número de entradas externas	20	24	30	24	4	97
Número de salidas externas	12	15	22	16	5	78
Número de preguntas externas	16	22	28	22	5	88
Número de archivos lógicos internos	4	4	5	4	10	42
Número de archivos de interfase externos	2	2	3	2	7	15
Conteo total						320

tulo 15. En cuanto a los propósitos de esta estimación se supone que el factor ponderado de complejidad es el promedio. La figura 23.3 presenta los resultados de esta estimación.

Se estima cada uno de los factores ponderados de complejidad y el valor del factor ajustado se calculan como se describe en el capítulo 15:

Factor		Valor
1	Respaldo y recuperación	4
2	Comunicaciones de datos	2
3	Procesamiento distribuido	0
4	Desempeño crítico	4
5.	Entorno operativo existente	3
6	Entrada de datos en línea	4
7	Transacción de entrada sobre pantallas múltiples	5
8.	ILF actualizado en línea	3
9	Complejo de valores de dominio de información	5
10.	Complejo de procesamiento interno	5
11.	Código diseñado para reutilización	4
12	Conversión/instalación en diseño	3
13	Instalaciones múltiples	5
14	Aplicación diseñada para cambio	5
	Factor de ajuste de valor	**1.17**

Finalmente, se deriva el número estimado de PF:

$$PF_{estimado} = \text{conteo total} \times [0.65 + 0.01 \times \Sigma(F_i)]$$

$$PF_{estimado} = 375$$

La productividad organizacional promedio para sistemas de este tipo es 6.5 PF/pm. Con base en una escala salarial de 8 000 dólares por mes, el costo por PF es aproximadamente 1 230 dólares. Con base en la estimación de PF y los datos de producti-

vidad históricos, el costo total estimado del proyecto es de 461 000 dólares, y el esfuerzo estimado es de 58 personas-mes.

CONSEJO

Si el tiempo lo permite, use la granularidad fina cuando especifique las tareas en la figura 23.4. Por ejemplo, divida el análisis en sus tareas principales y estime cada una por separado

23.6.5 Estimación basada en el proceso

La técnica más común para estimar un proyecto es basar la estimación en el proceso que se empleará. Esto es, el proceso se descompone en un conjunto relativamente pequeño de tareas y se estima el esfuerzo requerido para lograr cada tarea

Al igual que las técnicas basadas en el problema, la estimación basada en el proceso comienza con un bosquejo de las funciones del software obtenidas a partir de ámbito del proyecto. Cada función requiere realizar una serie de actividades del marco de trabajo Las funciones y actividades[7] del marco de trabajo relacionadas se presentan como parte de una tabla similar a la presentada en la figura 23.4.

FIGURA 23.4

Tabla de estimación basada en procesos

Actividad →	CC	Planificación	Análisis de riesgos	Ingeniería		Liberación de construcción		EC	Totales
Tarea →				Análisis	Diseño	Código	Prueba		
Función ↓									
FIUC				0.50	2.50	0.40	5.00	n/a	8.40
AG2D				0.75	4.00	0.60	2.00	n/a	7.35
AG3D				0.50	4.00	1.00	3.00	n/a	8.50
FPGC				0.50	3.00	1.00	1.50	n/a	6.00
GRD				0.50	3.00	0.75	1.50	n/a	5.75
FCP				0.25	2.00	0.50	1.50	n/a	4.25
MAD				0.50	2.00	0.50	2.00	n/a	5.00
Totales	0.25	0.25	0.25	3.50	20.50	4.75	16.50		46.00
% esfuerzo	0.5%	0.5%	0.5%	8%	45%	10%	36%		

CC = comunicación del cliente EC = evaluación del cliente

Una vez que se combinan las funciones del problema y las actividades del proceso, el planificador estima el esfuerzo (por ejemplo, personas-mes) que se requerirá para lograr cada actividad del proceso de software en cada función. Estos datos constituyen la matriz central de la tabla en la figura 23.4. Entonces se aplican las tasas de trabajo promedio (es decir, costo/unidad de esfuerzo) al esfuerzo estimado para cada actividad del proceso Es muy probable que la tasa de trabajo varíe en cada tarea El equipo veterano está enormemente involucrado en las actividades

7 Las actividades del marco de trabajo que se eligen para este proyecto difieren un poco de la actividades genéricas tratadas en el capítulo 2, que son la comunicación con el cliente (CC), la planificación, el análisis de riesgos, la ingeniería y la construcción-liberación

tempranas del marco de trabajo y generalmente es más costoso que el equipo menos experimentado involucrado en la construcción y liberación.

Los costos y el esfuerzo para cada función y actividad del marco de trabajo se calculan como el último paso. Si la estimación basada en el proceso se realiza independientemente de la estimación de LDC o PF, ahora se tienen dos o tres estimaciones para costo y esfuerzo que se pueden comparar y armonizar. Si ambos conjuntos de estimaciones muestran una concordancia razonable, existe una buena razón para creer que las estimaciones son confiables. Si, por otra parte, los resultados de dichas técnicas de descomposición muestran poca concordancia, se debe llevar a cabo mayor investigación y análisis.

> "Es mejor comprender el trasfondo de una estimación antes de utilizarlo."
>
> **Barry Boehm y Richard Fairley**

23.6.6 Un ejemplo de estimación basada en el proceso

Para ilustrar el uso de la estimación basada en el proceso considérese de nuevo el software CAD introducido en la sección 23.6.3. La configuración del sistema y las funciones del software permanecen invariables y las indica el ámbito del proyecto

En la tabla basada en el proceso que se muestra en la figura 23.4 las estimaciones del esfuerzo (en personas-mes) para cada actividad de ingeniería del software se proporcionan para cada función del software CAD (abreviadas para mayor rapidez). Las actividades de ingeniería y de liberación de construcción se subdividen en las principales tareas de ingeniería del software que se muestran. Las primeras estimaciones de esfuerzo corresponden a comunicación con el cliente, planificación y análisis de riesgos, las cuales se registran en la hilera total al final de la tabla. Los totales horizontal y vertical ofrecen un indicio del esfuerzo estimado que se requiere para análisis, diseño, código y prueba. Se debe señalar que 53 por ciento del esfuerzo se emplea en las tareas de ingeniería del sistema de salida (análisis de requisitos y diseño), lo que indica la importancia relativa de este trabajo.

Con base en la escala salarial promedio de 8 000 dólares por mes, el costo total estimado del proyecto es de 368 000 dólares , y el esfuerzo estimado es de 46 personas-mes. Si se desea, los promedios de trabajo pueden asociarse con cada actividad del marco de trabajo o tarea de ingeniería del software y calcularse por separado

23.6.7 Estimación con casos de uso

Como se ha señalado a lo largo de las partes 2 y 3 de este libro, los casos de uso permiten que un equipo de software comprenda el ámbito del software y los requisitos. Sin embargo, desarrollar un enfoque de estimación con casos de uso es problemático por las siguientes razones [SMI99]

¿Por qué es difícil desarrollar una técnica de estimación con casos de uso?

- Los casos de uso se describen empleando muchos formatos y estilos diferentes; no existe un formato estándar

- Los casos de uso representan una visión externa (la visión del usuario) del software y con frecuencia están escritos con diferentes grados de abstracción.
- Los casos de uso no abordan la complejidad de las funciones ni de las características que se describen.
- Los casos de uso no describen el comportamiento complejo (por ejemplo, interacciones) que involucran muchas funciones y características.

A diferencia de las LDC o un punto de función, el "caso de uso" de una persona tal vez requiera meses de esfuerzo mientras que el de otra quizá se implemente en un día o dos.

Aunque varios investigadores han considerado los casos de uso como una entrada a la estimación, a la fecha no ha surgido ningún método de estimación probado. Smith [SMI99] sugiere el empleo de los casos de uso en la estimación, pero sólo si se consideran dentro del contexto de la "jerarquía estructural" que los casos de uso describen.

Smith argumenta que cualquier nivel de esta jerarquía estructural se describe con no más de 10 casos de uso. Cada uno de éstos abarcaría no más de 30 escenarios distintos. Obviamente, los casos de uso que describen un sistema grande están escritos con un grado mucho mayor de abstracción (y representan considerablemente más esfuerzo de desarrollo) que aquellos que describen un solo subsistema. En consecuencia, antes de que los casos de uso se empleen en la estimación, se establece el nivel en la estructura jerárquica, se determina la longitud promedio (en páginas) de cada caso de uso, se define el tipo de software (por ejemplo, tiempo real, de negocios, de ingeniería/científico, anidado) y se considera una arquitectura común del sistema. Una vez establecidas dichas características, los datos empíricos se aprovechan para establecer el número estimado de LDC o de PF por caso de uso (para cada nivel de la jerarquía). Entonces se utilizan los datos históricos para calcular el esfuerzo necesario para desarrollar el sistema.

Enseguida se ilustra cómo realizar este cálculo; por tanto, considérese la siguiente relación:[8]

$$\text{LDC estimada} = N \times \text{LDC}_{\text{prom}} + [(S_a/S_h - 1) + (P_a/P_h - 1)] \times \text{LDC}_{\text{ajuste}} \qquad (23\text{-}2)$$

donde

N = número real de casos de uso

LDC_{prom} = promedio histórico de LDC por caso de uso para este tipo de subsistema

8 Es importante señalar que la expresión (23-2) se emplea sólo con propósitos ilustrativos. Al igual que todos los modelos de estimación, debe validarse localmente antes de que se le pueda utilizar con seguridad.

LDC_{ajuste} = representa un ajuste con base en n por ciento de LDC_{prom} donde n se define localmente y representa la diferencia entre este proyecto y los proyectos "promedio"
S_a = escenarios reales por caso de uso
S_h = escenarios promedio por caso de uso para este tipo de subsistema
P_a = páginas reales por caso de uso
P_h = página promedio por caso de uso para este tipo de subsistema

Con la expresión (23-2) se desarrolla una estimación común del número de LDC con base en el número real de casos de uso ajustado mediante el número de escenarios y la longitud de la página de los casos de uso. El ajuste representa hasta n por ciento del promedio histórico de las LDC por caso de uso.

23.6.8 Un ejemplo de estimación basada en casos de uso

El software CAD presentado en la sección 23.6.3 está compuesto de tres grupos de subsistemas:

- Subsistema de interfaz del usuario (incluye FIUC).
- Grupo de subsistema de ingeniería (incluye el subsistema AG2D, subsistema AG3D y el subsistema MAD)
- Grupo de subsistema de infraestructura (incluye el subsistema FPCG y el subsistema FCP).

Seis casos de uso describen el subsistema de interfaz del usuario. Cada caso de uso lo describen no más de 10 escenarios y tiene una longitud promedio de seis páginas. El grupo de subsistema de ingeniería lo describen 10 casos de uso (considerados en un nivel más alto de la jerarquía estructural). Cada uno de los casos de uso no tiene más de 20 escenarios asociados y tiene una longitud promedio de ocho páginas. Finalmente, el grupo de subsistema de infraestructura lo describen cinco casos de uso con un promedio de sólo seis escenarios y una longitud promedio de cinco páginas.

Con la relación anotada en la expresión (23-2) y n = 30 por ciento se elabora la tabla de la figura 23.5. Si se considera la primera hilera de la tabla, los datos históricos indican que el software IU requiere un promedio de 800 LDC por caso de uso cuando éste no tiene más de 12 escenarios y está descrito en menos de cinco pági-

FIGURA 23.5 Estimación de caso de uso.

	casos de uso	escenarios	páginas	escenarios	páginas	LDC	LDC estimadas
Subsistema de interfaz del usuario	6	10	6	12	5	560	3 366
Grupo de subsistema de ingeniería	10	20	8	14	8	3100	31 233
Grupo de subsistema de infraestructura	5	6	5	10	6	1650	7 970
Total LDC estimadas							42 568

nas. Estos datos se ajustan razonablemente bien para el sistema CAD. Así pues, la estimación de LDC para el subsistema de interfaz del usuario se calcula mediante la expresión (23-2). Si se aplica el mismo enfoque, se realizan estimaciones para los grupos de subsistemas de ingeniería e infraestructura. La figura 23.5 resume las estimaciones e indica que el tamaño global del software CAD se estima en 42 500 LDC.

Si se utilizan 620 LDC/pm como la productividad promedio en los sistemas de este tipo y una escala salarial de 8 000 dólares por mes, el costo por línea de código es aproximadamente de 13 dólares. Con base en la estimación de caso de uso y los datos históricos de productividad, el costo total estimado del proyecto es de 552 000 dólares, y el esfuerzo estimado es de 68 personas-mes.

23.6.9 Reconciliación de estimaciones

Las técnicas de estimación estudiadas en las secciones precedentes dan por resultado múltiples estimaciones que deben reconciliarse para producir una sola estimación de esfuerzo, duración del proyecto o costo. Este procedimiento de reconciliación se ilustra considerando de nuevo el software CAD presentado en la sección 23.6.3.

> "Tal vez los métodos complicados no produzcan una estimación más precisa, particularmente cuando los desarrolladores pueden incorporar su propia intuición en la estimación."
>
> **Phillip Johnson *et al.***

El esfuerzo estimado total para el software CAD varía desde un bajo de 46 personas-mes (obtenido empleando el enfoque de la estimación basada en el proceso) hasta un alto de 68 personas-mes (derivado con la estimación de caso de uso). La estimación promedio (que utiliza los cuatro enfoques) es de 56 personas-mes. La variación de la estimación promedio es aproximadamente 18 por ciento en el lado bajo y de 21 por ciento en el lado alto.

¿Qué ocurre cuando la concordancia entre las estimaciones es insuficiente? Responder esta pregunta requiere reevaluar la información con que se hicieron las estimaciones. Las estimaciones ampliamente divergentes con frecuencia pueden rastrearse hasta una de dos causas:

1. El planificador no ha comprendido adecuadamente o ha malinterpretado el ámbito del proyecto.
2. Los datos de productividad que utilizan las técnicas de estimación basadas en el problema son inapropiados para la aplicación, obsoletos (pues ya no reflejan con precisión la organización de ingeniería de software) o se han aplicado mal.

El planificador debe determinar la causa de la divergencia y luego reconciliar las estimaciones.

INFORMACIÓN

Técnicas de estimación automatizada para proyectos de software

Las herramientas de estimación automatizadas permiten que planificador estime costo y esfuerzo y realice análisis "si. . entonces" respecto de importantes variables del proyecto, como la fecha de entrega o la plantilla de personal. Aunque existen muchas herramientas de estimación automatizada (véase el recuadro más adelante en este capítulo), todas presentan las mismas características generales y realizan las siguientes seis funciones genéricas [JON96]:

1. *Tamaño de los entregables del proyecto* Se estima el "tamaño" de uno o más productos de trabajo del software. Los productos de trabajo incluyen la representación externa del software (por ejemplo, pantallas, reportes), el software en sí mismo (por ejemplo, KLDC), funcionalidad entregada (por ejemplo, puntos de función) e información descriptiva (por ejemplo, documentos).
2. *Selección de las actividades del proyecto.* Se selecciona el marco de trabajo del proceso adecuado y se especifica el conjunto de tareas de ingeniería del software.
3. *Predicción de los niveles del personal.* Se especifica el número de personas que estará disponible para realizar el trabajo. Puesto que la relación entre el personal disponible y el trabajo (esfuerzo predicho) es enormemente no lineal, ésta es una entrada importante.
4. *Predicción del esfuerzo de software* Las herramientas de estimación emplean uno o más modelos (sección 23.7) que relacionan el tamaño de los entregables del proyecto con el esfuerzo requerido para producirlos.
5. *Predicción del costo del software* Dados los resultados del paso 4, los costos se calculan relacionando los índices de trabajo con las actividades del proyecto anotadas en el paso 2.
6. *Predicción del programa de trabajo del software* Cuando se conocen el esfuerzo, el nivel del personal y las actividades del proyecto es posible producir un anteproyecto de programa al ubicar el trabajo a través de las actividades de ingeniería del software con base en los modelos recomendados para la distribución del esfuerzo estudiados en el capítulo 24.

La aplicación de diferentes herramientas de estimación a los mismos datos de proyecto permite encontrar una variación relativamente grande en los resultados estimados. Más importante todavía, en ocasiones los valores predichos son significativamente diferentes de los valores reales. Esto refuerza la noción de que las salidas de las herramientas de estimación se deben emplear como un "punto de datos" a partir del cual se derivan las estimaciones, no como la única fuente para una estimación.

23.7 MODELOS EMPÍRICOS DE ESTIMACIÓN

PUNTO CLAVE

Un modelo de estimación refleja la población de proyectos desde la que se ha derivado. Por lo tanto, el modelo es sensible al dominio

Un modelo de estimación para software de computadora utiliza fórmulas obtenidas empíricamente para predecir el esfuerzo como una función de LDC o PF.[9] Los valores de LDC o PF se estiman mediante el enfoque descrito en las secciones 23.6.3 y 23.6.4. Pero, en lugar de emplear las tablas descritas en dichas secciones, los valores resultantes para LDC o PF se colocan en el modelo de estimación.

Los datos empíricos que apoyan la mayoría de los modelos de estimación proceden de una muestra limitada de proyectos. Por esta razón, ningún modelo de estimación es apropiado para todas las clases de software ni en todos los entornos de desarrollo. En consecuencia, los resultados obtenidos a partir de tales modelos se deben emplear juiciosamente.

9 En la sección 23.6.7 se sugiere un modelo empírico que utiliza casos de uso como la variable independiente. Sin embargo, a la fecha han aparecido relativamente pocos en la respectiva bibliografía.

Un modelo de estimación debe calibrarse para reflejar las condiciones locales. El modelo debe probarse mediante la aplicación de los datos recopilados a partir de proyectos completados, colocar los datos en el modelo y luego comparar los resultados reales con los predichos. Si la concordancia es insuficiente, el modelo debe afinarse y ponerse a prueba nuevamente antes de que se pueda utilizar.

23.7.1 La estructura de los modelos de estimación

Un modelo de estimación típico se deriva mediante el análisis de regresión de los datos recopilados de proyectos de software previos. La estructura global de tales modelos toma la forma [MAT94]

$$E = A + B \times (e_v)^C \qquad (23\text{-}3)$$

donde A, B y C son constantes obtenidas empíricamente, E es el esfuerzo en persona-mes y e_v es la variable de estimación (ya sea LDC o PF). Además de la relación anotada en la ecuación (23-3), la mayoría de los modelos de estimación tiene alguna forma de componente de ajuste del proyecto que permite ajustar E por otras características del proyecto (por ejemplo, complejidad del problema, experiencia del personal, entorno de desarrollo). Entre los muchos modelos de estimación orientados a LDC propuestos en la bibliografía se encuentran:

CONSEJO

Ninguno de estos modelos se debe emplear sin una calibración cuidadosa a su entorno.

$E = 5.2 \times (KLDC)^{0.91}$	modelo Walston-Felix
$E = 5.5 + 0.73 \times (LDC)^{1.16}$	modelo Bailey-Basili
$E = 3.2 \times (KLDC)^{1.05}$	modelo simple de Boehm
$E = 5.288 \times (KLDC)^{1.0457}$	modelo Doty para KLDC > 9

También se han propuesto modelos orientados a PF. Entre éstos se incluyen:

$E = -91.4 + 0.355\ PF$	modelo de Albrecht y Gaffney
$E = -37 + 0.96\ PF$	modelo de Kemerer
$E = -12.88 + 0.405\ PF$	modelo de regresión para proyecto pequeño

Un rápido examen de estos modelos indica que cada uno producirá un resultado diferente para los mismos valores de LDC o PF. La implicación es clara. ¡Los modelos de estimación deben calibrarse para las necesidades locales!

23.7.2 El modelo COCOMO II

Referencia Web

En sunset.usc.edu/research/COCOMOII/cocomo-main.html se puede obtener información detallada acerca de COCOMO II, la última versión

En su libro clásico acerca de "economía de la ingeniería del software", Barry Boehm [BOE81] introduce una jerarquía de modelos de estimación de software que lleva el nombre de COCOMO, por *COnstructive COst MOdel* (Modelo Constructivo de Costos). El modelo COCOMO original se volvió uno de los más ampliamente utilizados y estudiados modelos de estimación de costo de software en la industria. Además, ha evolucionado a un modelo de estimación más amplio, llamado COCOMO II [BOE96, BOE00]. Al igual que su predecesor, COCOMO II es en realidad una jerarquía de modelos de estimación que aborda las áreas siguientes:

- *Modelo de composición de la aplicación*. Se emplea durante las primeras etapas de la ingeniería del software, cuando son primordiales la elaboración de prototipos de las interfases de usuario, la consideración de la interacción del software y el sistema, la valoración del desempeño y la evaluación de la madurez de la tecnología.
- *Modelo de etapa de diseño temprano*. Se utiliza una vez que se han estabilizado los requisitos y se ha establecido la arquitectura básica del software
- *Modelo de etapa posterior a la arquitectura*. Se emplea durante la construcción del software.

Al igual que los modelos de estimación del software, los modelos COCOMO II requieren información de tamaño. Como parte de la jerarquía del modelo hay disponibles tres opciones diferentes de tamaño: puntos objeto, puntos de función y líneas de código fuente.

El modelo COCOMO II de composición de la aplicación emplea puntos objeto, una medida indirecta de software que se calcula mediante conteos del número de 1) pantallas (en la interfaz del usuario), 2) reportes y 3) componentes que probablemente se requieran para construir la aplicación. Cada instancia de objeto (por ejemplo, una pantalla o reporte) se clasifica en uno de tres niveles de complejidad (es decir, simple, medio o difícil) aplicando criterios sugeridos por Boehm [BOE96]. En esencia, la complejidad es una función del número y origen de las tablas de datos del cliente y el servidor que se requieren para generar la pantalla o reporte, y el número de vistas o secciones presentadas como parte de la pantalla o el informe.

¿Qué es un punto objeto?

Una vez que se ha determinado la complejidad, el número de pantallas, informes y componentes se pondera de acuerdo con la tabla ilustrada en la figura 23.6. Entonces se determina la cuenta de puntos objeto al multiplicar el número original de instancias de objeto por el factor de ponderación en la figura y se suma para obtener una cuenta total de puntos objeto. Al aplicar el desarrollo basado en componentes o la reutilización general de software se estima el porcentaje de reutilización (%reut) y se ajusta la cuenta de puntos objeto:

$$\text{NPO} = (\text{puntos objeto}) \times [(100 - \%\text{reut})/100]$$

donde NPO se define como nuevos puntos objeto.

Para obtener una estimación del esfuerzo con base en el valor NPO calculado, se debe calcular una "tasa de productividad". La figura 23.7 presenta la tasa de productividad

$$\text{PROD} = \text{NPO}/\text{persona-mes}$$

para diferentes niveles de experiencia del desarrollador y madurez del entorno de desarrollo. Una vez determinada la tasa de productividad se puede obtener una estimación del esfuerzo del proyecto del modo siguiente:

$$\text{esfuerzo estimado} = \text{NPO}/\text{PROD}$$

FIGURA 23.6

Ponderación de complejidad para tipos de objeto [BOE96].

Tipo de objeto	Peso de complejidad		
	Simple	Medio	Difícil
Pantalla	1	2	3
Reporte	2	5	8
Componente 3GL			10

FIGURA 23.7 **Tasa de productividad por puntos objeto [BOE96].**

Experiencia/capacidad del desarrollador	Muy baja	Baja	Nominal	Alta	Muy alta
Madurez/capacidad del entorno	Muy baja	Baja	Nominal	Alta	Muy alta
TASA PRODUCTIVIDAD (PROD)	4	7	13	25	50

En modelos COCOMO II más avanzados[10] se requieren varios factores de escala, controladores de costo y procedimientos de ajuste. El lector interesado debe leer [BOE00] o visitar el sitio Web de COCOMO II.

23.7.3 La ecuación del software

Referencia Web

www.qsm.com

Información acerca de herramientas de estimación de costo de software que han evolucionado a partir de la ecuación del software

La ecuación de software [PUT92] es un modelo multivariable que supone una distribución específica del esfuerzo a lo largo de la vida de un proyecto de desarrollo de software. El modelo procede de datos de productividad recopilados de casi 4 000 proyectos de software contemporáneos. Con base en estos datos, un modelo de estimación de la forma

$$E = [\text{LDC} \times B^{0.333}/P]^3 \times (1/t^4) \qquad (23\text{-}4)$$

donde

E = esfuerzo en personas-mes o personas-año
t = duración del proyecto en meses o años
B = "factor especial de habilidades"[11]

10 Como se anotó anteriormente, estos modelos utilizan conteos de PF y KLDC para la variable tamaño.

11 B aumenta lentamente conforme "crecen la necesidad de integración, las pruebas, la garantía de calidad, la documentación y las habilidades de gestión" [PUT92]. Para programas pequeños (KLDC = 5 a 15), $B = 0.16$. Para programas más grandes que 70 KLDC, $B = 0.39$.

P = "parámetro de productividad" que refleja: madurez global del proceso y prácticas de gestión, la medida en la que se emplean las buenas prácticas de ingeniería del software, el nivel de los lenguajes de programación utilizados, el estado del entorno del software, las habilidades y experiencias del equipo de software, y la complejidad de la aplicación.

Los valores típicos pueden ser $P = 2\,000$ para desarrollo del software anidado de tiempo real; $P = 10\,000$ para software de telecomunicaciones y sistemas; $P = 28\,000$ para aplicaciones de sistemas de negocios. El parámetro de productividad se puede calcular para condiciones locales si se emplean datos históricos recopilados de esfuerzos de desarrollo previos.

Es importante notar que la ecuación del software tiene dos parámetros independientes: 1) una estimación del tamaño (en LDC) y 2) una estimación de la duración del proyecto en meses o años calendario.

Putnam y Myers [PUT92] sugieren un conjunto de ecuaciones derivadas de la ecuación del software para simplificar el proceso de estimación y emplear una forma más común para su modelo de estimación. El tiempo mínimo de desarrollo se define como

$$t_{min} = 8.14(\text{LDC}/P)^{0.43} \text{ en meses para } t_{min} > 6 \text{ meses} \quad (23\text{-}5a)$$

$$E = 180Bt^3 \text{ en personas-mes para } E \geq 20 \text{ personas-mes} \quad (23\text{-}5b)$$

Nótese que t se representa en años en la ecuación (23-5b).

Al utilizar las ecuaciones (23-5) con P = 12 000 (el valor recomendado para software científico) para el software CAD estudiado previamente en este capítulo,

$$t_{min} = 8.14(33\,200/12\,000)^{0.43}$$
$$t_{min} = 12.6 \text{ meses calendario}$$
$$E = 180 \times 0.28 \times (1.05)^3$$
$$E = 58 \text{ personas-mes}$$

Los resultados de la ecuación del software corresponden favorablemente con las estimaciones desarrolladas en la sección 23.6. Al igual que el modelo COCOMO señalado en la sección precedente, la ecuación del software ha evolucionado durante la década pasada. En [PUT97b] se puede encontrar una detallada exposición de una versión extendida de este enfoque de estimación.

23.8 ESTIMACIÓN PARA PROYECTOS ORIENTADOS A OBJETOS

Vale la pena complementar los métodos convencionales de estimación de costo del software con un enfoque diseñado explícitamente para software OO. Lorenz y Kidd [LOR94] sugieren el enfoque siguiente

1. Desarrollar estimaciones aplicando la descomposición de esfuerzo, análisis de PF y cualquier otro método que sea aplicable en aplicaciones convencionales.

2. Aplicar el modelado de análisis orientado a objetos (capítulo 8), desarrollar casos de uso y determinar un conteo. Reconocer que el número de casos de uso puede cambiar conforme avance el proyecto.
3. A partir del modelo de análisis, determinar el número de clases clave (llamadas *clases de análisis* en el capítulo 8).
4. Categorizar el tipo de interfaz para la aplicación y desarrollar un multiplicador para las clases de soporte:

Tipo de interfaz	Multiplicador
Sin IUG	2.0
Interfaz del usuario basada en texto	2.25
IUG	2.25
IUG compleja	3.0

Multiplicar el número de clases clave (paso 3) por el multiplicador para obtener una estimación del número de clases de soporte.

5. Multiplicar el número total de clases (clave + soporte) por el número promedio de unidades de trabajo por clase. Lorenz y Kidd sugieren de 15 a 20 personas-día por clase.
6. Comprobar de manera cruzada la estimación basada en clase al multiplicar el número promedio de unidades de trabajo por caso de uso.

23.9 TÉCNICAS DE ESTIMACIÓN ESPECIALIZADAS

Las técnicas de estimación estudiadas en las secciones 23.6, 23.7 y 23.8 se pueden aplicar en cualquier proyecto de software. Sin embargo, cuando un equipo de software encuentra una duración de proyecto extremadamente corta (semanas más que meses) —que probablemente tengan una continua corriente de cambios— la planificación del proyecto en general y la estimación en particular se deben abreviar [12]. En las secciones siguientes se examinan dos técnicas de estimación especializadas.

23.9.1 Estimación para desarrollo ágil

Puesto que los requerimientos para un proyecto ágil (capítulo 4) se definen como un conjunto de escenarios de usuario (por ejemplo, "historias" en programación extrema) es posible desarrollar un enfoque de estimación informal, aunque razonablemente disciplinado y significativo dentro del contexto de la planificación del proyecto para cada incremento de software.

La estimación para los proyectos ágiles aplica un enfoque de descomposición que abarca los pasos siguientes:

12 "Abreviar" no significa eliminar. Incluso los proyectos de corta duración deben planificarse, y la estimación es el fundamento de la planificación sólida.

¿Cómo se desarrollan las estimaciones cuando se aplica un proceso ágil?

1. Cada escenario de usuario (el equivalente de un minicaso de uso creado en el comienzo mismo de un proyecto por los usuarios finales u otros participantes) se considera por separado respecto de propósitos de estimación.
2. El escenario se descompone en el conjunto de funciones y las tareas de ingeniería del software que se requerirán para desarrollarlo.

3a. Cada tarea se estima por separado. Nota: la estimación se puede basar en datos históricos, en un modelo empírico o en la "experiencia".

3b. Alternativamente, el "volumen" (tamaño) del escenario se puede estimar en LDC, PF o alguna otra medida orientada a volumen (por ejemplo, puntos objeto).

CONSEJO

El contexto de la estimación para un proyecto ágil, donde "es una descripción del alcance global de un incremento de usuario [HIS] o PF.

4a. Las estimaciones de cada tarea se suman para crear una estimación para el escenario.

4b. Alternativamente, el volumen estimado para el escenario se traduce en esfuerzo mediante la aplicación de datos históricos.

5. Las estimaciones de esfuerzo acerca de todos los escenarios que se implementarán para un incremento de software dado se suman con el fin de desarrollar el esfuerzo estimado para el incremento.

Puesto que la duración del proyecto requerido para el desarrollo de un incremento de software es bastante corta (usualmente de 3-6 semanas), este enfoque de estimación tiene dos propósitos: 1) asegurar que el número de escenarios que se incluirán en el incremento se integra con los recursos disponibles, y 2) establecer una base para ubicar el esfuerzo conforme se desarrolla el incremento.

23.9.2 Estimación para proyectos de ingeniería Web

Como se asentó en el capítulo 16, los proyectos de ingeniería Web con frecuencia adoptan el modelo de proceso ágil. Es factible emplear una medición de punto de función modificada, en conjunto con los pasos subrayados en la sección 23.9.1, con el fin de desarrollar una estimación para la WebApp.

Roetzheim [ROE00] sugiere los siguientes valores de dominio de información cuando se adaptan puntos de función (capítulos 15 y 22) para la estimación WebApp:

- *Entradas* son cada pantalla o formato de entrada (por ejemplo, CGI o Java), cada pantalla de mantenimiento y, si en alguna parte utiliza una etiqueta de metáfora de cuaderno, cada etiqueta.
- *Salidas* son cada página Web estática, cada guión de página Web dinámica (por ejemplo, ASP, ISAPI u otro guión DHTML) y cada reporte (ya sea que esté basado en Web o sea del todo administrativo).
- *Tablas* son cada tabla lógica en la base de datos más, si emplea XML para almacenar datos en un archivo, cada objeto XML (o colección de atributos XML).

- Las *interfaces* retienen su definición como archivos lógicos (por ejemplo, formatos de registro únicos) en las fronteras exteriores del sistema.
- *Consultas* son cada interfaz publicada externamente o utiliza una interfaz orientada al mensaje. Un ejemplo típico son las referencias externas DCOM o COM

Los puntos de función (calculados empleando los valores de dominio de información anotados) son un indicador razonable del volumen para una WebApp.

Mendes y sus colegas [MEN01] sugieren que el volumen de una WebApp se determina mejor mediante la recopilación de medidas (llamadas "variables predictoras") asociadas con la aplicación (por ejemplo, conteo de página, conteo de medios audiovisuales, conteo de función), las características de su página Web (por ejemplo complejidad de página, complejidad de vinculación, complejidad gráfica), sus características de medios audiovisuales (por ejemplo, duración de los clips) y características funcionales (por ejemplo, longitud de código, longitud de código reutilizado). Dichas medidas se pueden emplear en el desarrollo de modelos de estimación empí-

HERRAMIENTAS DE SOFTWARE

Estimación de esfuerzo y costo

Objetivo: El objetivo de las herramientas de estimación de esfuerzo y costo es proporcionar al equipo del proyecto una estimación del esfuerzo requerido, de la duración del proyecto y del costo en una forma que aborde las características específicas del proyecto inmediato y el entorno en el que se construirá.

Mecánica: En general, las herramientas de estimación de costo utilizan una base de datos histórica procedente de proyectos locales, datos recopilados a través de la industria y un modelo empírico (por ejemplo, COCOMO II) que se emplea para calcular estimaciones de esfuerzo, duración y costo. Las características del proyecto y el entorno de desarrollo son entradas, y la herramienta proporciona un rango de estimación de salidas.

Herramientas representativas[13]

Costar, desarrollado por Softstar Systems (www.softstarsystems.com), emplea el modelo COCOMO II para desarrollar estimaciones de software.

Cost Xpert, desarrollado por Cost Xpert Group, Inc. (www.costxpert.com), integra modelos de estimación múltiples y una base de datos histórica de proyectos.

Estimate Professional, desarrollado por el Software Productivity Center, Inc. (www.spc.com), está basado en COCOMO II y en el Modelo SLIM.

Knowledge Plan, desarrollado por Software Productivity Research (www.spr.com), utiliza la entrada de punto de función como el controlador primario para un paquete de estimación completo.

Price S, desarrollado por Price Systems (www.pricesystems.com), es una de las herramientas más viejas y ampliamente utilizadas en proyectos de desarrollo de software a gran escala.

SEER/SEM, desarrollado por Galorath Inc. (www.galorath.com), proporciona una capacidad de estimación completa, análisis de sensibilidad, valoración de riesgo y otras características.

SLIM-Estimate, desarrollado por QSM (www.qsm.com), echa mano de "bases de conocimiento industrial" detalladas para ofrecer una "verificación sanitaria" de las estimaciones obtenidas empleando datos locales.

13 Las herramientas anotadas aquí son una muestra de esta categoría. En la mayoría de los casos nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

rica para esfuerzos de proyecto total, de creación de página, de creación de medios audiovisuales y de creación de guiones. Sin embargo, todavía falta realizar más trabajo antes de que tales modelos puedan emplearse con confianza.

23.10 LA DECISIÓN DESARROLLAR-COMPRAR

A menudo, en muchas áreas de aplicación de software es más rentable adquirir que desarrollar software de computadora. Los gestores de ingeniería del software enfrentan una decisión de desarrollar-comprar que tal vez se complique aún más por varias opciones de adquisición: 1) el software se puede comprar (o adquirir la licencia) ya desarrollado, 2) los componentes de software de "experiencia completa" o "experiencia parcial" (véase la sección 23.4.2) se pueden adquirir y luego modificar e integrar para satisfacer necesidades específicas, o 3) el software se puede construir de manera personalizada por medio de un contratista externo para satisfacer las necesidades del comprador.

Los pasos involucrados en la adquisición los definen lo crucial del software que se comprará y el costo final. En el análisis final, la decisión desarrollar-comprar se realiza basándose en las siguientes condiciones: 1) ¿El producto de software estará disponible antes que el software desarrollado de manera interna? 2) ¿El costo de adquisición más el costo de personalización será menor que el costo de desarrollar el software de manera interna? 3) ¿El costo del soporte externo (por ejemplo, un contrato de mantenimiento) será menor que el costo del soporte interno? Estas condiciones se aplican a cada una de las opciones de adquisición.

23.10.1 Creación de un árbol de decisión

¿Existe una forma [illegible] de [illegible] los [illegible] [illegible] con la [illegible] desarrollar-comprar?

Los pasos recién descritos se pueden aumentar mediante técnicas estadísticas tales como el *análisis del árbol de decisión* [BOE89]. Por ejemplo, la figura 23.8 bosqueja un árbol de decisión para un sistema basado en software, *X*. En este caso, la organización de ingeniería del software puede 1) construir el sistema *X* desde cero, 2) reutilizar componentes existentes de "experiencia parcial" para construir el sistema, 3) comprar un producto de software disponible y modificarlo para satisfacer las necesidades locales, o 4) contratar el desarrollo de software con una empresa externa.

Si el sistema se construirá desde cero existe un 70 por ciento de probabilidad de que el trabajo sea difícil. Al emplear las técnicas de estimación estudiadas antes en este capítulo, el planificador del proyecto estima que un esfuerzo de desarrollo difícil costará 450 000 dólares. Un esfuerzo de desarrollo "simple" se estima que costará 380 000 dólares. El *valor esperado* para el costo, calculado a lo largo de cualquier rama del árbol de decisión, es

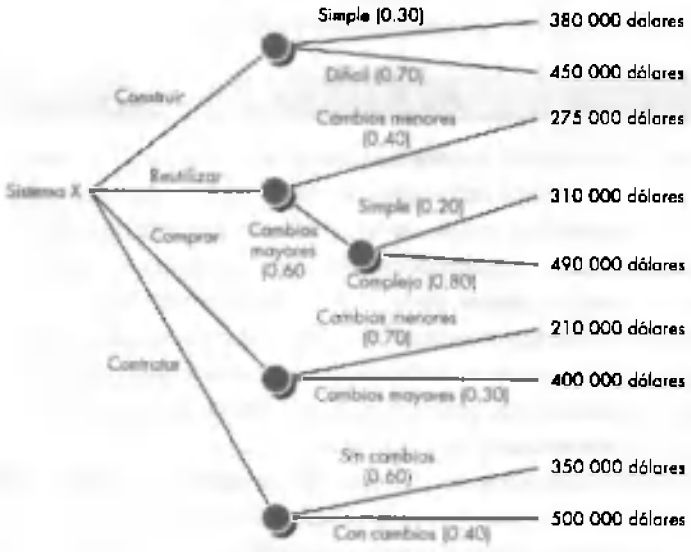
$$\text{costo esperado} = \sum (\text{probabilidad de la ruta})_i \times (\text{costo estimado de la ruta})_i$$

donde i es la trayectoria del árbol de decisión. Para la trayectoria de construcción,

$$\text{costo esperado}_{\text{construir}} = 0.30(380\text{K dólares}) + 0.70(450\text{K dólares}) = 429\text{K dólares}$$

FIGURA 23.8

Árbol de decisión para apoyar la decisión desarrollar-comprar.



Al seguir otras trayectorias del árbol de decisión también se muestran, en una diversidad de circunstancias, los costos proyectados para reutilización, compra y contratación. Los costos esperados para dichas trayectorias son

$$\text{costo esperado}_{\text{reutilizar}} = 0.40(275\text{K dólares}) = 0.60[0.20(310\text{K dólares}) + 0.80(490\text{K dólares})] = 382\text{K dólares}$$

$$\text{costo esperado}_{\text{comprar}} = 0.70(210\text{K dólares}) + 0.30(400\text{K dólares}) = 267\text{K dólares}$$

$$\text{costo esperado}_{\text{contratar}} = 0.60(350\text{K dólares}) + 0.40(500\text{K dólares}) = 410\text{K dólares}$$

Con base en la probabilidad y los costos proyectos que se han anotado en la figura 23.8, el costo esperado más bajo es la opción "comprar".

Sin embargo, es importante señalar que se deben considerar muchos criterios —no sólo costo— durante el proceso de toma de decisión. Disponibilidad, experiencia del desarrollador-vendedor-contratista, concordancia con los requisitos, "políticas" locales y la probabilidad de cambiar son sólo algunos de los criterios que pueden incidir en la decisión final de construir, reutilizar, comprar o contratar.

23.10.2 Subcontratación

Tarde o temprano, cualquier compañía que desarrolla software de computadora se plantea una pregunta fundamental: ¿existe una forma de conseguir los sistemas y software necesarios a un precio más bajo? La respuesta no es tan simple, y los debates emocionales que genera la pregunta siempre conducen a una sola palabra: *subcontratación*.

Como concepto, la subcontratación es extremadamente simple. Las actividades de ingeniería del software se contratan con una tercera parte que realiza el trabajo

a un costo más bajo y, así se espera, mayor calidad. El trabajo de software llevado a cabo dentro de una compañía se reduce a una actividad de gestión de contratos.[14]

> "Como regla, la subcontratación requiere incluso más gestión experta que el desarrollo en casa."
>
> **Steve McConnell**

La decisión de subcontratar puede ser estratégica o táctica. En el ámbito estratégico, los gestores comerciales consideran si una porción significativa de todo el trabajo de software se puede contratar con otros. En el plano táctico, un gestor de proyecto determina si parte o todo un proyecto puede lograrse mejor al subcontratar el trabajo de software.

Sin importar la amplitud de la visión, la decisión de subcontratar usualmente es financiera. Una exposición detallada del análisis financiero de la subcontratación está más allá del ámbito de este libro y mejor se deja a otros (por ejemplo, [MIN95]) Sin embargo, vale la pena una breve revisión de los pros y contras de la decisión.

En el lado positivo, usualmente es factible ahorrar en el costo reduciendo el número de personal de software y las instalaciones (por ejemplo, computadoras, infraestructura) que los soportan En el lado negativo, una compañía pierde cierto control sobre el software que necesita. Dado que el software es una tecnología que diferencia sus sistemas, servicios y productos, una compañía corre el riesgo de poner el destino de su competitividad en las manos de una tercera parte.

HogarSeguro

Subcontratación

El escenario: Sala de juntas de

actores: Mal Golden, gerente ejecutivo, desarrollo producto; Lee Warren, gerente de ingeniería; Joe , VP ejecutivo, desarrollo comercial; Doug , gerente de proyecto, ingeniería de software

conversación:

Joe: Estamos considerando subcontratar la porción de del software del producto *HogarSeguro*

(Impresionada): ¿Cuándo sucedió esto?

Lee: Obtuvimos un presupuesto de un desarrollador externo. Presupuesta un 30 por ciento por debajo de lo que tu grupo parece creer que costará. Aquí. [Extiende el presupuesto a Doug, quien lo lee.]

Mal: Como sabes, Doug, estamos tratando de mantener los costos bajos, y 30 por ciento es 30 por ciento. Además, estas personas vienen muy bien recomendadas.

Doug (toma un respiro e intenta recuperar la calma): Me tomaron por sorpresa, pero antes de que tomen una decisión final, ¿algunos comentarios?

Joe (asiente con la cabeza): Seguro, adelante

Doug: No hemos trabajado con esta empresa subcontratista antes, ¿cierto?

Mal: Cierto, pero

14 La subcontratación se puede considerar, de manera más general, como cualquier actividad que conduce a la adquisición de software o algunos de sus componentes con una fuente externa a la organización de ingeniería del software

Doug: Y ellos anotan que cualquier cambio a las especificaciones será cobrado a una tasa adicional, ¿cierto?

Joe (frunce el entrecejo): Cierto, pero esperamos que las cosas serán razonablemente estables

Doug: Una mala suposición, Joe.

Joe: Bueno ..

Doug: Es probable que liberemos nuevas versiones de este producto durante algunos años Y es razonable suponer que el software proporcionará muchas de las nuevas características, ¿cierto?

[Todos afirman con la cabeza.]

Doug: ¿Alguna vez hemos coordinado un proyecto internacional?

Lee (se ve preocupado): No, pero me dijeron ..

Doug (intenta suprimir su enojo): Así que lo que me están diciendo es 1) estamos a punto de trabajar con una empresa desconocida, 2) los costos para hacer esto no son tan bajos como parecen, 3) de facto nos estamos comprometiendo a trabajar con ellos durante muchas liberaciones de producto, sin importar qué hagan en la primera, y 4) aprenderemos en el camino lo relativo a un proyecto internacional.

[Todos guardan silencio]

Doug: Muchachos... creo que esto es un error, y me gustaría que tomaran un día para reconsiderar Tendremos más control si hacemos el trabajo en casa Tenemos la experiencia y puedo garantizar que no nos costará mucho más... el riesgo será más bajo y yo sé que todos tienen aversión por el riesgo, como yo.

Joe (cañudo): Has anotado buenos puntos, pero tú tienes un interés personal en mantener este proyecto en casa.

Doug: Es cierto, pero eso no cambia los hechos

Joe (con un suspiro): Muy bien, pospongamos esto un día o dos, pensemos en ello un poco más y reunámonos de nuevo para una decisión final Doug, ¿puedo hablar contigo en privado?

Doug: Claro... Realmente quiero estar seguro de que hacemos las cosas correctas

23.11 RESUMEN

El planificador del proyecto de software debe estimar tres factores antes de que un proyecto comience: cuánto tiempo tomará, cuánto esfuerzo requerirá y cuánto personal estará involucrado. Además, el planificador debe predecir los recursos (hardware y software) que se requerirán y el riesgo involucrado.

La descripción del ámbito ayuda al planificador a desarrollar estimaciones empleando una o más técnicas que se clasifican en dos amplias categorías: descomposición y modelado empírico.

Las técnicas de descomposición requieren un bosquejo de las principales funciones del software, seguido por estimaciones de 1) el número de LDC, 2) valores seleccionados dentro del dominio de información, 3) el número de casos de uso, 4) el número de personas-mes requerido para implementar cada función, o 5) el número de personas-mes requerido para cada actividad de ingeniería del software. Las técnicas empíricas usan expresiones para esfuerzo y tiempo obtenidas empíricamente para predecir estas cantidades del proyecto. Se pueden utilizar herramientas automatizadas para implementar un modelo empírico específico.

Por lo general, las estimaciones precisas de proyecto emplean al menos dos de las tres técnicas anotadas. Al comparar y reconciliar las estimaciones obtenidas con la aplicación de diferentes técnicas, el planificador tiene más probabilidad de calcu-

lar una estimación precisa. La estimación del proyecto de software nunca será una ciencia exacta, pero una combinación de buenos datos históricos y técnicas sistemáticas puede mejorar la precisión de la estimación.

REFERENCIAS

[BEN92] Bennatan, E. M., *Software Project Management: A Practitioner's Approach*, McGraw-Hill, 1992.

[BEN03] Bennatan, E. M., "So What is the State of Software Estimation?" en *The Cutter Edge* (hoja informativa en línea), 11 de febrero de 2002, disponible en http://www.cutter.com

[BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981

[BOE89] Boehm, B., *Risk Management*, IEEE Computer Society Press, 1989.

[BOE96] Boehm, B., "Anchoring the Software Process", en *IEEE Software*, vol. 13, núm. 4, julio de 1996, pp. 73-82

[BOE00] Boehm, B. *et al*, *Software Cost Estimation in COCOMO II*, Prentice-Hall, 2000.

[BRO75] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975

[GAU89] Gause, D. C. y G. M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.

[HOO91] Hooper, J. y R. O. Chester, *Software Reuse Guidelines and Methods*, Plenum Press, 1991

[JON96] Jones, C., "How Software Estimation Tools Work", en *American Programmer*, vol. 9, núm. 7, julio de 1996, pp. 19-27.

[LOR94] Lorenz, M. y J. Kidd, *Object-Oriented Software Metrics*, Prentice-Hall, 1994.

[MAT94] Matson, J., B. Barrett y J. Mellichamp, "Software Development Cost Estimation Using Function Points", en *IEEE Trans. Software Engineering*, vol. SE-20, núm. 4, abril de 1994, pp. 275-287.

[MCC98] McConnell, S., *Software Project Survival Guide*, Microsoft Press, 1998.

[MEN01] Mendes, E., N. Mosley y S. Counsell, "Web Metrics-Estimating Design and Authoring Effort", *IEEE Multimedia*, enero-marzo de 2001, pp. 50-57.

[MIN95] Minoli, D., *Analyzing Outsourcing*, McGraw-Hill, 1995.

[PHI98] Phillips, D., *The Software Project Manager's Handbook*, IEEE Computer Society Press, 1998.

[PUT78] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimation Problem", en *IEEE Trans. Software Engineering*, vol. SE-4, núm. 4, julio de 1978, pp. 345-361.

[PUT92] Putnam, L. y W. Myers, *Measures for Excellence*, Yourdon Press, 1992.

[PUT97a] Putnam, L. y W. Myers, "How Solved Is the Cost Estimation Problem?", en *IEEE Software*, noviembre de 1997, pp. 105-107

[PUT97b] Putnam, L. y W. Myers, *Industrial Strength Software: Effective Management Using Measurement*, en IEEE Computer Society Press, 1997

[ROE00] Roetzheim, W., "Estimating Internet Development", en *Software Development*, agosto de 2000, disponible en http://www.sdmagazine.com/documents/s=741/sdm0008d/0008d.htm

[SMI99] Smith, J., "The Estimation of Effort Based on Use Cases", Rational Software Corp., 1999, se puede descargar de http://www.rational.com/media/whitepapers/finalTP171.PDF.

PROBLEMAS Y PUNTOS A CONSIDERAR

23.1. Suponga que es el gestor de proyecto de una compañía que construye software para robots caseros. Se le ha contratado para construir el software destinado a un robot que corta el pasto. Describa por escrito el ámbito del software. Asegúrese de que la descripción del ámbito esté acotada. Si no está familiarizado con los robots, investigue un poco antes de comenzar a escribir. Además, establezca sus suposiciones acerca del hardware que se requerirá. Alternativa: sustituya el robot que corta el pasto por otro problema robótico que le interese.

23.2. La complejidad del proyecto de software influye en la precisión de la estimación. Desarrollar una lista de características de software (por ejemplo, operación concurrente, salida gráfica) que afecten la complejidad de un proyecto. Establecer prioridades en la lista

23.3. El desempeño es una consideración importante durante la planificación. Comentar c se puede interpretar de manera diferente el desempeño, dependiendo del área de aplicación software

23.4. Haga una descomposición funcional del software robótico que describió en el probl 23.1. Estime el tamaño de cada función en LDC. Suponga que su organización produce LDC/pm con una escala salarial de 7 000 dólares por persona-mes. Estime el esfuerzo y c requeridos para construir el software empleando la técnica de estimación basada en LDC crita en este capítulo.

23.5. Emplear el modelo COCOMO II en la estimación del esfuerzo que requiere la constru del software para una simple ATM que produce 12 pantallas, 10 reportes y requerirá aproxim mente 80 componentes de software. Suponer complejidad promedio y madurez desarroll entorno promedio. Emplear el modelo de composición de aplicación con puntos objeto.

23.6. Utilizar la ecuación del software para estimar el software del robot que corta pasto problema 23.1. Suponer que las ecuaciones (23-5) son aplicables y que P = 8000.

23.7. Comparar las estimaciones de esfuerzo obtenidas en los problemas 23.4 y 23.6. ¿Cuál es la desviación estándar y cómo afecta el grado de certidumbre acerca de la estimación?

23.8. Utilizar los resultados obtenidos en el problema 23.7 para determinar si es razonable esperar que el software se construya dentro de los siguientes seis meses y cuánto personal se tendría que emplear para realizar el trabajo.

23.9. Desarrollar un modelo de hojas de cálculo que implemente una o más de las técnicas estimación descritas en este capítulo. Alternativamente, adquirir de fuentes basadas en W uno o más modelos en línea para la estimación de proyectos de software.

23.10. Para un equipo de proyecto, desarrollar una herramienta de software que implemente cada una de las técnicas de estimación desarrolladas en este capítulo.

23.11. Parece extraño que las estimaciones de costo y programa de trabajo se desarrollen durante la planificación del proyecto de software, antes de que se haya llevado a cabo un diseño o un análisis detallado de los requisitos de software. ¿Por qué cree que se haga esto? ¿Existen circunstancias en las cuales no se deba hacer?

23.12. Vuelva a calcular los valores anotados para el árbol de decisión en la figura 23. Suponga que cada rama tiene una probabilidad de 50-50. ¿Esto cambiaría su decisión final?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La mayoría de los libros de gestión de proyectos de software incluyen análisis de la estimación de proyectos. El Project Management Institute (*PMBOK Guide*, PMI, 2001), Wysoki y sus colegas (*Effective Project Management*, Wiley, 2000), Lewis (*Project Planning Scheduling and Control*, tercera edición, McGraw-Hill, 2000), Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques*, tercera edición, Wiley, 2000) y Phillips [PHI98] ofrecen útiles directrices de estimación.

Jones (*Estimating Software Costs*, McGraw-Hill, 1998) ha escrito uno de los tratamientos más completos de la materia publicados a la fecha. Su libro contiene modelos y datos aplicables a estimaciones de software en cualquier dominio de aplicación. Coombs (*IT Project Estimation*, Cambridge University Press, 2002), Roetzheim y Beasley (*Software Project Cost and Schedule Estimating: Best Practices*, Prentice-Hall, 1997) y Wellman (*Software Costing*, Prentice-Hall, 1992) presentan muchos modelos útiles y sugieren directrices paso a paso para generar las mejores estimaciones posibles.

El detallado tratamiento de Putnam y Myers de la estimación de costo del software ([PUT92] y [PUT97b]) y los libros de Boehm acerca de economía de ingeniería del software ([BOE81] y COCOMO II [BOE00]) describen modelos de estimación empíricos. Estos libros proporcionan un análisis detallado de datos derivados de cientos de proyectos de software. Un libro excelente de

DeMarco (*Controlling Software Projects*, Yourdon Press, 1982) ofrece un valiosa visión de gestión, medición y estimación de proyectos de software. Lorenz y Kidd (*Object-Oriented Software Metrics*, Prentice-Hall, 1994) y Cockburn (*Surviving Object-Oriented Projects*, Addison-Wesley, 1998) consideran la estimación para sistemas orientados a objetos.

En Internet hay disponible una amplia variedad de fuentes de información acerca de estimación de software. Una lista actualizada de referencias en la World Wide Web se encuentra en el sitio Web de SEPA:
http://www.mhhe.com/pressman.

CAPÍTULO

24 CALENDARIZACIÓN DE PROYECTOS DE SOFTWARE

CONCEPTOS CLAVE

A finales de los de los años 60, un joven y brillante ingeniero fue elegido para "escribir" un programa de computadora para una aplicación industrial automatizada. La razón por la cual se le eligió fue simple: era la única persona en el grupo técnico que había asistido a un seminario de programación de computadoras. Él conocía las entradas y salidas del lenguaje ensamblador y de FORTRAN, pero nada acerca de ingeniería del software, e incluso menos acerca de calendarización y seguimiento de proyectos.

Su jefe le dio los manuales apropiados y una descripción verbal de lo que tenía que hacer. Se le informó que el proyecto debía terminarse en dos meses.

El ingeniero leyó los manuales, consideró su enfoque y comenzó a escribir el código. Después de dos semanas, el jefe lo llamó a su oficina y le preguntó cómo iban las cosas.

"Realmente bien", dijo el joven ingeniero con entusiasmo juvenil. "Esto es mucho más simple de lo que pensé. Probablemente he terminado cerca del 75 por ciento."

El jefe sonrió y alentó al joven ingeniero a seguir trabajando bien. Planearon reunirse de nuevo en una semana.

Una semana después, el jefe llamó al ingeniero a su oficina y le preguntó: "¿Dónde estamos?"

"Todo marcha bien", dijo el joven, "pero me he encontrado con algunos pequeños obstáculos. Los solucionaré y regresaré al ritmo de trabajo muy pronto."

"¿Cómo ves la fecha límite?", preguntó el jefe.

UN VISTAZO RÁPIDO

¿Qué es? Usted seleccionó un modelo de proceso apropiado; identificó las tareas de ingeniería del software que es preciso realizar; estimó la cantidad de trabajo y el número de personas; conoce la fecha límite; incluso consideró los riesgos. Ahora es tiempo de unir los puntos. Esto es, tiene que crear una red de tareas de ingeniería del software que le permitirán tener el trabajo listo a tiempo. Una vez creada la red, tiene que asignar responsabilidades a cada tarea, asegurarse de que se realice y adaptar la red conforme los riesgos se vuelvan realidad. En resumen, esto es la calendarización y el seguimiento del proyecto de software.

¿Quién lo hace? En el ámbito del proyecto los gestores del proyecto de software emplean la información solicitada a los ingenieros de software. En el plano individual, los mismos ingenieros de software.

¿Por qué es importante? En la construcción de un sistema complejo muchas tareas de ingeniería del software ocurren en paralelo, y el resultado del trabajo realizado durante una tarea puede tener un profundo efecto en el trabajo llevado a cabo en otra tarea. Estas interdependencias son muy difíciles de comprender sin una calendarización. También es virtualmente imposible evaluar el progreso de un proyecto de software moderado y grande sin una calendarización detallada.

¿Cuáles son los pasos? Las tareas de ingeniería del software que dicta el modelo de proceso de software se refinan para la funcionalidad que se construirá. A cada tarea se le asignan esfuerzo y duración, y se crea una red de tareas (también llamada una "red de actividad") de tal forma que permita al equipo de software cumplir con la fecha límite de entrega establecida.

¿Cuál es el producto obtenido? La calendarización del proyecto y la información relacionada.

¿Cómo me puedo asegurar que lo he hecho correctamente? La calendarización adecuada requiere que 1) todas las tareas aparezcan en la red, 2) el esfuerzo y el tiempo estén asignados de manera inteligente en cada tarea, 3) las interdependencias entre tareas estén indicadas adecuadamente, 4) los recursos estén asignados para el trabajo que se realizará y 5) los hitos estén espaciados de modo cercano para que se pueda seguir el progreso.

"No hay problema", dijo el ingeniero. "Estoy cerca de terminar el 90 por ciento."

Si se ha trabajado en el mundo del software durante unos cuantos años se es capaz de terminar la historia. No será sorpresa que el joven ingeniero[1] haya permanecido en el 90 por ciento durante todo el proyecto y terminado (con la ayuda de otros) un mes después.

Esta historia se ha repetido decenas de miles de veces con desarrolladores de software durante las pasadas cuatro décadas. La gran pregunta es por qué

24.1 Conceptos básicos

Aunque existen muchas razones por las cuales el software se entrega con retraso, la mayoría se encuadra en una o más de las siguientes causas:

- Una fecha límite irrealizable establecida por alguien externo al grupo de ingeniería del software e impuesta a los gestores y profesionales del grupo.
- Cambio en los requisitos del cliente que no se reflejan en modificaciones a la calendarización
- Una subestimación razonable de la cantidad de esfuerzo o de recursos que se requerirán para realizar el trabajo.
- Riesgos predecibles o impredecibles que no se consideraron cuando comenzó el proyecto.
- Dificultades técnicas que no pudieron preverse
- Dificultades humanas imprevisibles.
- Falta de comunicación entre el personal del proyecto, lo que genera demoras.
- Una falla en la gestión del proyecto porque no reconoció el retraso ni emprendió una acción para corregir el problema

1 En caso de que el lector se lo pregunte, la historia es autobiográfica

"Las calendarizaciones excesivas o irracionales probablemente son la fuerza particular más destructiva en todo el software."

Capers Jones

Las fechas límite muy audaces (léase "irrealizables") son un hecho de la vida en el negocio del software. En tales ocasiones las fechas límite se demandan por razones legítimas, desde el punto de vista de la persona que las establece. Pero el sentido común establece que la legitimidad también la deben advertir las personas que hacen el trabajo

Napoleón dijo alguna vez. "Cualquier comandante en jefe que pretenda llevar a cabo un plan que considera defectuoso comete un error; debe exponer sus razones insistir en que el plan debe cambiarse y finalmente presentar su renuncia en lugar de ser el instrumento de la destrucción de su ejército". Estas son palabras fuertes que muchos gestores de proyectos de software deben considerar.

Las actividades de estimación estudiadas en el capítulo 23 y las técnicas de calendarización descritas en éste con frecuencia se implementan atendiendo la restricción de una fecha límite definida. Si las mejores estimaciones indican que la fecha límite es irrealizable, un gestor de proyecto competente debe "proteger a su equipo de la presión excesiva [de la calendarización]... [y] devolver la presión a quienes la originan" [PAG85].

Para ilustrarlo, supóngase que a un equipo de ingeniería del software se le ha pedido construir un controlador en tiempo real para un instrumento de diagnóstico médico que será introducido al mercado en nueve meses. Después de una estimación y un análisis de riesgo cuidadosos (capítulo 25), el gestor del proyecto llega a la conclusión de que el software, como se solicitó, requerirá 14 meses para crearlo con el personal disponible. ¿Cómo procede el gestor del proyecto?

"Adoro las fechas límite. Me gusta que pasan como una exhalación cuando se alejan."

Douglas Adams

Es irreal presentarse en la oficina del cliente (en este caso el probable cliente es mercadotecnia-ventas) y demandarle que cambie la fecha de entrega Las presiones externas del mercado han dictado la fecha, y el producto debe liberarse. Es igualmente torpe rechazar el trabajo (desde el punto de vista profesional). Así que, ¿qué hacer? En esta situación se recomiendan los siguientes pasos:

1. Realizar una estimación detallada empleando datos históricos de proyectos previos Determinar el esfuerzo y la duración estimados para el proyecto.
2. Aplicar un modelo de proceso incremental (capítulo 3) y desarrollar una estrategia de ingeniería de software que entregará la funcionalidad crítica en la fecha límite impuesta, pero demorará otra. Documente el plan.

¿Qué se debe hacer cuando la gestión demanda que se cumpla con una fecha límite que es imposible?

3. Reunirse con el cliente y, con la estimación detallada, explicarle por qué la fecha límite impuesta es irrealizable. Asegúrese de señalar que todas las estimaciones están basadas sobre el desempeño en proyectos previos. También asegúrese de indicar el porcentaje de mejoría que se requeriría para lograr la fecha límite vigente.[2] Son apropiados los siguientes comentarios:

   "Creo que podemos tener un problema con la fecha de entrega para el software controlador XYZ. Le he dado a cada uno de ustedes un análisis abreviado de las tasas de producción en proyectos previos y una estimación que hemos hecho en algunas formas diferentes. Notarán que he supuesto un 20 por ciento de mejora respecto de ritmos de producción precedentes, pero todavía tenemos una fecha de entrega que está a 14 meses en lugar de 9."

4. Ofrezca la estrategia de desarrollo incremental como alternativa:
   "Tenemos unas cuantas opciones y me gustaría que tomase una decisión con base en ellas. Primero, podemos aumentar el presupuesto y conseguir recursos adicionales de modo que tendremos mucho éxito en lograr que este trabajo esté hecho en nueve meses. Pero comprenda que esto aumentará el riesgo de una calidad deficiente debido a la apretada fecha límite.[3] Segundo, podemos remover varias de las funciones y capacidades de software que está solicitando. Esto hará que la versión preliminar del producto sea un poco menos funcional, pero podemos anunciar toda la funcionalidad y luego entregarla en el periodo de 14 meses. Tercero, podemos prescindir de la realidad y esperar que el proyecto se complete en nueve meses. Terminaremos con nada que se pueda entregar a un cliente. La tercera opción, espero que esté de acuerdo, es inaceptable. La historia y nuestras mejores estimaciones indican que es irreal y un boleto hacia el desastre."

Habrá algunos gruñidos, pero si se presentan estimaciones sólidas basadas en buenos datos históricos es probable que se elegirán versiones negociadas de las opciones 1 o 2. La fecha límite irreal se evapora.

24.2 CALENDARIZACIÓN DE PROYECTO

A Fred Brooks, el bien conocido autor de *The Mythical Man-Month* [BRO95], se le preguntó una vez cómo se retrasaban los proyectos de software en la calendarización. Su respuesta fue tan simple como profunda: "Un día a la vez."

2 Si la mejora requerida es de 10 a 25 por ciento, de hecho tal vez sea posible tener listo el trabajo. Pero, con mayor probabilidad, la mejora requerida en el desempeño del equipo será mayor que el 50 por ciento. Esta es una expectativa irreal.

3 También puede agregar que el aumento en el número de personas no reduce proporcionalmente el tiempo.

Las tareas requeridas para lograr el objetivo de una gestión de proyecto no se deben llevar a cabo manualmente. Existen muchas excelentes herramientas de calendarización. Úselas.

La realidad de un proyecto técnico (ya sea que involucre la construcción de una planta hidroeléctrica o el desarrollo de un sistema operativo) es que cientos de pequeñas tareas deben realizarse para lograr una meta mayor. Algunas de tales tareas están fuera de la corriente principal y se pueden completar sin preocuparse acerca de su impacto sobre la fecha de terminación del proyecto. Otras tareas se encuentran en la "trayectoria crítica". Si estas tareas "críticas" se retrasan en la calendarización, la fecha de terminación del proyecto se pone en riesgo.

El objetivo del gestor es definir todas las tareas del proyecto, construir una red que bosqueje sus interdependencias, identificar las tareas cruciales dentro de la red y luego seguir su progreso para garantizar que la demora se reconoce "un día a la vez". Para lograrlo el gestor debe tener una calendarización que se haya definido en un grado de resolución que permita supervisar el progreso y controlar el proyecto.

La *calendarización del proyecto de software* es una actividad que distribuye estimaciones de esfuerzo a través de la duración planificada del proyecto al asignar el esfuerzo a tareas específicas de ingeniería del software. Sin embargo, es importante señalar que la calendarización evoluciona a lo largo del tiempo. Durante las primeras etapas de la planificación del proyecto se desarrolla una calendarización macroscópica. Este tipo de calendarización identifica las principales actividades del marco de trabajo del proceso y las funciones de producto a las que se aplican. Conforme el proyecto transcurre, cada entrada en la calendarización macroscópica se refina en una calendarización detallada. Aquí se identifican y calendarizan tareas específicas del software (requeridas para completar una actividad).

> "Una calendarización demasiado optimista no genera una calendarización real más corta, sino una mayor."
>
> **Steve McConnell**

La calendarización para proyectos de ingeniería de software se puede ver desde dos perspectivas más bien diferentes. En la primera ya se ha establecido (irrevocablemente) una fecha final para la liberación de un sistema basado en computadora. La organización de software está restringida a distribuir esfuerzo dentro del marco temporal prescrito. La segunda visión de la calendarización de software supone que se han comentado límites cronológicos aproximados, pero que la fecha final la establece la organización de ingeniería del software. El esfuerzo se distribuye para utilizar mejor los recursos y la fecha final se define después de un análisis cuidadoso del software. Por desgracia, la primera situación se encuentra con mucha más frecuencia que la segunda.

24.2.1 Principios básicos

Al igual que otras áreas de ingeniería del software, varios principios básicos guían la calendarización de los proyectos:

Compartimentación El proyecto debe dividirse en compartimentos en varias actividades, acciones y tareas manejables. Lograrlo requiere descomponer tanto el producto como el proceso.

PUNTO CLAVE

...a desarrolle ...ización, ...ntese el ...ntífense las ...encias de ...s, asígnense ...y tiempo a ..., definanse ...s hitos.

Interdependencia. Se debe determinar la interdependencia de cada actividad, acción o tarea compartimentada. Algunas tareas deben ocurrir en secuencia mientras que otras pueden ocurrir en paralelo. Algunas acciones o actividades no pueden comenzar mientras no esté disponible el producto de trabajo producido por otros. Otras acciones o actividades pueden ocurrir de manera independiente.

Asignación de tiempo. A cada tarea por calendarizar se le debe asignar cierto número de unidades de trabajo (por ejemplo, personas-día de esfuerzo). Además, a cada tarea se le debe asignar una fecha de inicio y una otra de terminación que sean función de las interdependencias, y ya sea que el trabajo sea realizado con base en tiempo completo o parcial.

Validación del esfuerzo. Todo proyecto tiene un número definido de personas en el equipo de software. Conforme ocurre la asignación de tiempo, el gestor de proyecto debe asegurarse de que, en un tiempo dado, no se han asignado más que el número de personas calendarizadas. Por ejemplo, considere un proyecto que tiene tres ingenieros de software asignados (por ejemplo, tres personas-día están disponibles por día de esfuerzo asignado).[4] En un día dado se deben completar siete tareas al mismo tiempo. Cada tarea requiere 0.50 personas-día de esfuerzo. Se ha asignado más esfuerzo que el número de personas para hacer el trabajo.

Definición de responsabilidades. Toda tarea calendarizada se le debe asignar a un miembro específico del equipo.

Definición de resultados. Toda tarea calendarizada debe tener un resultado definido. En proyectos de software el resultado normalmente es un producto de trabajo (por ejemplo, el diseño de un módulo) o una parte de él. Los productos de trabajo usualmente se combinan en los entregables.

Definición de hitos. Cualquier tarea o grupo de tareas debe estar asociado con un hito del proyecto. Un hito se logra cuando se ha revisado la calidad de uno o más productos de trabajo (capítulo 26) y se han aprobado.

Cada uno de estos principios se aplica conforme evoluciona la calendarización del proyecto.

24.2.2 Relación entre el personal y el esfuerzo

En un pequeño proyecto de desarrollo de software una sola persona puede analizar los requisitos, realizar el diseño, generar el código y dirigir las pruebas. Conforme aumenta el tamaño de un proyecto, más gente resulta involucrada. (¡Rara vez se puede dar el lujo de acometer un esfuerzo de 10 personas-año con una persona que trabaje durante 10 años!)

4 En realidad, hay disponibles menos de tres personas-día de esfuerzo debido a las reuniones no relacionadas, enfermedades, vacaciones y una diversidad de otras razones. Sin embargo, para los propósitos del texto, se supone una disponibilidad de 100 por ciento.

CONSEJO

Si se deben agregar personas a un proyecto retrasado, asegúrese de que se les ha asignado trabajo enormemente compartimentado.

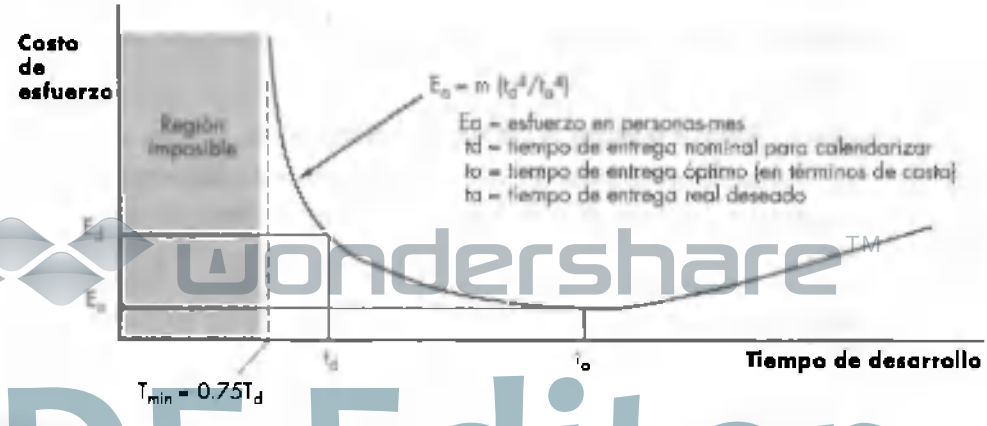
Existe un mito común que todavía creen muchos gestores responsables del esfuerzo de desarrollo del software: "Si nos retrasamos en la calendarización, siempre podemos incorporar más programadores y recuperarnos más adelante en el proyecto". Desgraciadamente, agregar más personas en etapas tardías de un proyecto con frecuencia tiene un efecto perturbador sobre éste, lo que provoca que la calendarización se desfase aún más. Las personas que se agregan deben aprender el sistema y la gente que les enseña es la misma que estaba haciendo el trabajo. Durante la enseñanza no se realiza trabajo y el proyecto experimenta mayores retrasos.

Además del tiempo que tarda en aprender el sistema, más personal aumenta el número de rutas de comunicación y la complejidad de ésta a lo largo de un proyecto. Aunque la comunicación es absolutamente esencial para el éxito del desarrollo de software, cada nueva ruta de comunicación requiere un esfuerzo adicional y, por lo tanto, tiempo suplementario.

A lo largo de los años, los datos empíricos y el análisis teórico han demostrado que las calendarizaciones de proyecto son elásticas. Es decir, es posible comprimir en cierta medida la fecha de terminación deseada del proyecto (al añadir recursos adicionales). También es posible extender la fecha de terminación (al reducir el número de recursos).

La *Curva Putnam-Norden-Rayleigh* (*PNR*)[5] proporciona un indicio de la relación entre el esfuerzo aplicado y el tiempo de entrega para un proyecto de software. En la figura 24.1 se muestra una versión de la curva, que representa el esfuerzo del proyecto como función del tiempo de entrega. La curva indica un valor mínimo, t_o, que indica el tiempo de entrega de menor costo (es decir, el tiempo de entrega que generará el menor gasto de esfuerzo). Conforme se mueve a la izquierda de t_o (es decir, conforme intenta acelerar la entrega), la curva se eleva en forma no lineal.

FIGURA 24.1 Relación entre esfuerzo y tiempo de entrega.



5 En [NOR70] y [PUT78] se puede encontrar investigación original.

PUNTO CLAVE

...la entrega puede ...arse, la curva ... indica que los ... del proyecto se ... reducir ...almente

Como ejemplo, supóngase que un equipo de proyecto ha estimado un nivel de esfuerzo E_a que se requerirá para lograr un tiempo de entrega nominal, t_d, que es óptimo en términos de calendarización y recursos disponibles. Aunque es posible acelerar la entrega, la curva se eleva muy pronunciadamente hacia la izquierda de t_d. De hecho, la curva PNR indica que el tiempo de entrega del proyecto no se puede comprimir más allá de 0.75 t_d. Si se intenta una mayor compresión, el proyecto se mueve hacia "la región imposible" y el riesgo de fracaso se eleva mucho La curva PNR también indica que la opción de entrega de menor costo, $t_o = 2t_d$ La implicación aquí es que la demora en la entrega del proyecto puede reducir los costos significativamente. Desde luego, esto debe sopesarse frente al costo del negocio asociado con la demora.

La ecuación del software [PUT92], introducida en el capítulo 23, se obtiene de la curva PNR y demuestra la relación enormemente lineal entre el tiempo cronológico para completar un proyecto y el esfuerzo humano aplicado a éste. El número de líneas de código entregadas (enunciados fuente), L, se relaciona con el esfuerzo y el tiempo de desarrollo mediante la ecuación:

CONSEJO

...la fecha ... del proyecto se ... cada vez más, ...alcanza un punto ...al que el trabajo ...debe completarse ...el calendario, ...reportar el ... de personas ...hagan el trabajo ...la realidad ...una nueva ... de entrega.

$$L = P \times E^{1/3} t^{4/3}$$

donde E es el esfuerzo de desarrollo en personas-mes; P, un parámetro de productividad que refleja una diversidad de factores que conducen a trabajo de ingeniería del software de alta calidad (los valores típicos de P varían entre 2 000 y 12 000); y t, la duración del proyecto en meses calendario.

Al reordenar esta ecuación del software se puede llegar a una expresión para el esfuerzo de desarrollo E:

$$E = L^3/(P^3 t^4) \qquad (24\text{-}1)$$

donde E es el esfuerzo utilizado (en personas-año) durante el ciclo de vida para el desarrollo y mantenimiento de software, y t es el tiempo de desarrollo en años. La ecuación para el esfuerzo de desarrollo se puede relacionar con el costo del desarrollo al incluir un factor de escala salarial (costo/persona-año).

Esto conduce a unos resultados interesantes Considérese un complejo proyecto de software de tiempo real estimado en 33 000 LDC, 12 personas-año de esfuerzo. Si se asignan ocho personas al equipo del proyecto, éste se puede completar en aproximadamente 1.3 años. Sin embargo, si se extiende la fecha final a 1.75 años, la naturaleza enormemente no lineal del modelo descrito en la ecuación (24-1) produce:

$$E = L^3/(P^3 t^4) \sim 3.8 \text{ personas-año}$$

Esto implica que, al extender la fecha final seis meses, ¡se puede reducir el número de personas de ocho a cuatro! La validez de tales resultados está abierta al debate, pero la implicación es clara: se pueden obtener beneficios al emplear menos personal durante un periodo un poco más largo para lograr el mismo objetivo.

24.2.3 Distribución del esfuerzo

¿Cómo se debe distribuir el esfuerzo a lo largo del flujo de trabajo del proceso de software?

Cada una de las técnicas de estimación de proyectos de software estudiadas en el capítulo 23 conduce a estimaciones de unidades de trabajo (por ejemplo, personas-mes) requeridas para completar el desarrollo del software. Una distribución recomendada del esfuerzo a través del proceso de software con frecuencia se conoce como la *regla 40-20-40*. Cuarenta por ciento de todos los esfuerzos se asignan al análisis y el diseño de sistemas de entrada Un porcentaje similar se aplica en poner a prueba los sistemas de salida Usted puede inferir correctamente que la codificación (20 por ciento del esfuerzo) no tiene tanto énfasis.

Esta distribución del esfuerzo se debe usar solamente como guía.[6] Las características de cada proyecto deben dictar la distribución del esfuerzo. El trabajo realizado en la planeación del proyecto rara vez explica más de 2-3 por ciento del esfuerzo, a menos que el plan comprometa a una organización a grandes gastos con alto riesgo Los análisis de requisitos pueden comprometer 10-25 por ciento del esfuerzo de. proyecto El esfuerzo empleado en el análisis o elaboración de prototipos debe aumentar en proporción directa con el tamaño y la complejidad del proyecto. Un intervalo del 20 al 25 por ciento de esfuerzo normalmente se aplica al diseño de software También se debe considerar el tiempo utilizado en la revisión del diseño y la subsiguiente iteración

Debido al esfuerzo aplicado al diseño de software, el código debe seguir con relativamente poca dificultad. Se puede lograr un rango de 15-20 por ciento del esfuerzo global. Las pruebas y la subsiguiente depuración explican el 30-40 por ciento del esfuerzo de desarrollo del software. El carácter crucial del software con frecuencia dicta la cantidad de pruebas que se requieren. Si el software se clasifica desde el punto de vista humano (es decir, la falla del software puede resultar en pérdida de vidas), son usuales porcentajes todavía más altos

24.3 Definición de un conjunto de tareas para el proyecto de software

En la parte 1 de este libro se describieron varios modelos de proceso diferentes. Sin importar si un equipo de software elige un modelo secuencial lineal, un modelo incremental, un modelo evolutivo o alguna combinación, el modelo de proceso está poblado de un conjunto de tareas que le permiten a un equipo de software definir, desarrollar y, a final de cuentas, brindar soporte al software de computadora

Ningún conjunto de tareas es apropiado por sí solo para todos los proyectos. El conjunto de tareas que sería apropiado para un sistema complejo y grande probable-

6 En la actualidad, la regla 40-20-40 enfrenta una ofensiva. Algunos creen que más del 40 por ciento del esfuerzo global se debe utilizar durante el análisis y el diseño. Por otra parte, algunos partidarios del desarrollo ágil (capítulo 4) argumentan que se debe emplear menos tiempo frontal "directo" y que un equipo se debe mover rápidamente hacia la construcción

mente se apreciaría como demasiado destructivo para un producto de software pequeño y relativamente simple. En consecuencia, un proceso de software eficaz debe definir una colección de conjuntos de tareas, cada una diseñada para satisfacer las necesidades de diferentes tipos de proyectos.

Como se mencionó en el capítulo 2, un conjunto de tareas es una colección de tareas de trabajo de ingeniería del software, hitos y productos de trabajo que se deben terminar para completar un proyecto particular. El conjunto de tareas debe proporcionar suficiente disciplina para lograr alta calidad de software. Pero, al mismo tiempo, no debe abrumar al equipo del proyecto con trabajo innecesario

El desarrollo de una calendarización del proyecto requiere distribuir un conjunto de tareas a lo largo de la línea de tiempo del proyecto. El conjunto de tareas variará según el tipo de proyecto y el grado de rigor con el que el equipo de software decide realizar su trabajo Aunque es difícil desarrollar una taxonomía completa de tipos de proyecto de software, en la mayoría de las organizaciones del ramo se encuentran los siguientes proyectos:

1. *Proyectos de desarrollo del concepto*, los cuales se inician para explorar algunas aplicaciones o conceptos de negocios de alguna nueva tecnología.
2. Proyectos de *desarrollo de nuevas aplicaciones*, los cuales se llevan a cabo como consecuencia de una solicitud específica del cliente
3. Proyectos de *mejora de la aplicación*, éstos ocurren cuando el software existente experimenta grandes modificaciones en la función, el desempeño o las interfases visibles para el usuario final
4. *Proyectos de mantenimiento de aplicación*, los cuales corrigen, adaptan o extienden el software existente en formas que no sean obvias inmediatamente para el usuario final.
5. *Proyectos de reingeniería*, éstos se llevan a cabo con la finalidad de reconstruir un sistema existente (heredado), en todo o en parte.

Referencia Web

Se ha desarrollado un modelo de proceso adaptable (APM, por sus siglas en inglés) que ayuda a definir los conjuntos de tareas para varios proyectos de software. Una descripción completa del APM se encuentra en www.rspa.com/apm.

Incluso dentro de un solo tipo de proyecto, muchos factores influyen en la elección del conjunto de tareas. Por ejemplo [PRE99]: tamaño del proyecto, número de usuarios potenciales, lo crucial de la misión, duración de la aplicación, estabilidad de los requisitos, facilidad de comunicación con el usuario o desarrollador, madurez de la tecnología aplicable, restricciones del desempeño, características anidadas y no anidadas, equipo del proyecto y factores de reingeniería. Cuando se consideran en conjunto, estos factores ofrecen un indicio del *grado de rigor* que se debe aplicar al proceso de software.

24.3.1 Ejemplo de conjunto de tareas

Cada uno de los tipos de proyecto descritos puede abordarse mediante un modelo de proceso lineal, secuencial, iterativo (por ejemplo, los modelos de elaboración de prototipo o incrementales) o evolutivo (por ejemplo, el modelo espiral). En algunos casos,

un tipo de proyecto fluye suavemente hacia el siguiente. Por ejemplo, los proyectos de desarrollo del concepto que triunfan con frecuencia evolucionan en proyectos de desarrollo de nuevas aplicaciones. Cuando termina un proyecto de desarrollo de nuevas aplicaciones, en ocasiones comienza un proyecto de mejora de una aplicación. Esta progresión es tanto natural como predecible y ocurrirá sin importar el modelo de proceso que adopte la organización. En consecuencia, las principales tareas de ingeniería del software descritas en las secciones que siguen son aplicables a todos los flujos de modelo de proceso. Como ejemplo, considérense las tareas de ingeniería del software para un proyecto de desarrollo del concepto.

Los proyectos de desarrollo del concepto se inician cuando se debe explorar el potencial para alguna nueva tecnología. No existe certeza de que la tecnología será aplicable, pero un cliente (por ejemplo, marketing) cree que existen beneficios potenciales. Los proyectos de desarrollo del concepto se enfocan en aplicar las siguientes tareas principales:

1.1 La **determinación del ámbito del concepto** precisa el ámbito global del proyecto.

1.2 La **planeación preliminar del concepto** establece la habilidad de la organización para acometer el trabajo que entraña el ámbito del proyecto.

1.3 La **valoración del riesgo de la tecnología** evalúa el riesgo asociado con la tecnología que se implementará como parte del ámbito del proyecto.

1.4 La **prueba del concepto** demuestra la viabilidad de una nueva tecnología en el contexto del software.

1.5 La **implementación del concepto** pone en práctica la representación del concepto en una forma que pueda revisarla un cliente y se utiliza para propósitos de "mercadotecnia" cuando se debe vender un concepto a otros clientes o gestores.

1.6 La **reacción del cliente** al concepto solicita realimentación acerca de un concepto de nueva tecnología y se dirige a aplicaciones específicas de los clientes.

Una rápida exploración de estas tareas debe producir pocas sorpresas. De hecho, el flujo de ingeniería del software para los proyectos de desarrollo del concepto (y también para todos los otros tipos de proyectos) es poco más que sentido común.

24.3.2 Refinamiento de las tareas principales

Las tareas principales descritas en la sección precedente se pueden utilizar para definir la calendarización macroscópica de un proyecto. Sin embargo, esta calendarización se debe refinar para crear una calendarización detallada del proyecto. El refinamiento comienza al tomar cada tarea principal y descomponerla en un conjunto de subtareas (con productos de trabajo e hitos relacionados).

Como ejemplo de descomposición de tarea, considérese la tarea 1.1, "determinación del ámbito del concepto". El refinamiento de la tarea se puede lograr empleando un bosquejo de formato, pero en este libro se aplica un enfoque de lenguaje en el diseño del proceso para ilustrar el flujo de la actividad de determinación del ámbito del concepto.

Definición tarea: Tarea 1.1 Determinación del ámbito del concepto
1.1.1 Identificar necesidades, beneficios y clientes potenciales;
1.1.2 Definir eventos de salida/control y entrada deseados que impulsen la aplicación;
Comienza Tarea 1.1.2
1.1.2.1 RTF: Revisar la descripción escrita de la necesidad[7]
1.1.2.2 Derivar una lista de salidas/entradas visibles al cliente
1.1.2.3 RTF: Revisar salidas/entradas con el cliente y modificar conforme se requiera
fintarea Tarea 1.1.2
1.1.3 Definir la funcionalidad/comportamiento para cada función principal:
Comienza Tarea 1.1.3
1.1.3.1 RTF: Revisar los objetos de datos de salida y entrada derivados en la tarea 1.1.2;
1.1.3.2 Derivar un modelo funciones/comportamientos;
1.1.3.3 RTF: Revisar funciones/comportamientos con el cliente y modificar conforme se requiera;
fintarea Tarea 1.1.3
1.1.4 Aislar aquellos elementos de la tecnología que se implementará en el software;
1.1.5 Disponibilidad de investigación del software existente;
1.1.6 Definir factibilidad técnica;
1.1.7 Realizar estimación rápida del tamaño;
1.1.8 Crear una Definición del ámbito;
fintarea definición: Tarea 1.1

Las tareas y subtareas anotadas en el proceso de refinamiento del lenguaje de diseño forman la base de una planeación detallada de la actividad de determinar el ámbito del concepto.

24.4 Definición de una red de tareas

Las tareas y subtareas individuales tienen interdependencias basadas en su secuencia. Además, cuando más de una persona está involucrada en un proyecto de ingeniería del software, es probable que las actividades y tareas de desarrollo se realicen en paralelo. Cuando esto ocurre, las tareas concurrentes deben estar coordinadas de modo que se completarán cuando las tareas posteriores requieran sus productos de trabajo.

Una *red de tareas*, también denominada *red de actividad*, es una representación gráfica del flujo de tareas en un proyecto. En ocasiones se utiliza como el mecanismo mediante el cual la secuencia y dependencias de tareas son la entrada a una herramienta automatizada de calendarización del proyecto. En su forma más simple

7 RTF indica que se debe realizar una revisión técnica formal (capítulo 26).

FIGURA 24.2 **Red de tareas para desarrollo del concepto.**

(empleada cuando se crea una calendarización macroscópica), la red de tareas muestra las principales tareas de la ingeniería del software. La figura 24.2 muestra una red de tareas esquemática para un proyecto de desarrollo del concepto

La naturaleza concurrente de las actividades de ingeniería del software conduce a varios requisitos importantes de la calendarización. Puesto que las tareas paralelas ocurren de manera asíncrona, el planificador debe determinar dependencias intertareas para asegurar el progreso continuo hacia la finalización. Además, el gestor del proyecto debe estar atento a estas tareas que se encuentran en la *ruta crítica*. Esto es, las tareas que se deben completar en la calendarización si el proyecto como un todo se debe completar a tiempo. Más adelante en este capítulo se tratan con más detalle estos temas.

Es importante notar que la red de tareas mostrada en la figura 24.2 es macroscópica. En una red de tareas detallada (un precursor de una calendarización detallada) cada actividad que muestra la figura se debe expandir. Por ejemplo, la tarea 1.1 se expandiría para mostrar todas las tareas detalladas en el refinamiento de las tareas 1.1. mostradas en la sección 24.3.2.

24.5 CALENDARIZACIÓN

La calendarización de un proyecto de software no difiere enormemente de la de cualquier esfuerzo de ingeniería multitarea. En consecuencia, las técnicas y herramientas generalizadas de calendarización de proyecto se pueden aplicar, poco modificadas, en proyectos de software.

La *técnica de evaluación y revisión de programa* (PERT, por sus siglas en inglés) y el *método de ruta crítica* (CPM, *por sus siglas en inglés*) son dos métodos de calendarización de proyecto que se pueden aplicar al desarrollo de software. Ambas técnicas

reciben impulso de la información ya desarrollada en actividades tempranas de la planeación del proyecto·

- Estimación del esfuerzo
- Descomposición de la función del producto.
- Selección del modelo de proceso y conjunto de tareas apropiados
- Descomposición de tareas.

Las interdependencias entre las tareas se pueden definir empleando una red de tareas. Las tareas, en ocasiones llamadas la *estructura de análisis del trabajo* (EAT, por sus siglas en inglés), se definen para el producto como un todo o para funciones individuales.

> "Todo lo que tenemos que decidir es qué hacer con el tiempo que nos han dado."
>
> **Gandalf en *El señor de los anillos: la hermandad del anillo***

Tanto PERT como CPM ofrecen herramientas cuantitativas que permiten al planificador de software 1) determinar la trayectoria crítica: la cadena de tareas que determinan la duración del proyecto; 2) establecer las estimaciones de tiempo "más probables" para tareas individuales al aplicar modelos estadísticos, y 3) calcular los "tiempos límite" que definen una "ventana" de tiempo para una tarea particular

HERRAMIENTAS DE SOFTWARE

Calendarización de proyectos

Objetivo: El objetivo de las herramientas de calendarización de proyectos es permitir que un gestor defina las tareas de trabajo, establezca sus dependencias, asigne recursos humanos a las tareas y desarrolle una variedad de gráficas, diagramas y tablas que auxilian en el seguimiento y el control del proyecto de software.

Mecánica: En general, las herramientas de calendarización de proyectos requieren la especificación de una estructura de análisis de trabajo o la generación de una red de tareas. Una vez que se define el análisis de la tarea (un bosquejo) o red, a cada tarea se ligan fechas de inicio y fin, recursos humanos, fechas límite duras y otros datos. Entonces la herramienta genera una variedad de cronogramas y otras tablas que le permiten a un gestor valorar el flujo de tareas de un proyecto. Dichos datos pueden actualizarse de manera continua conforme el proyecto se lleva a cabo

Herramientas representativas[8]

AMS Realtime, desarrollada por Advanced Management Systems (www.amsusa com), ofrece capacidades de calendarización para proyectos de todos los tamaños y tipos.

Microsoft Project, desarrollada por Microsoft (www.microsoft.com), es la herramienta de calendarización de proyectos basada en PC más ampliamente usada.

Viewpoint, desarrollada por Artemis Internation Solutions Corp (www.atemispm com), soporta todos los aspectos de la planificación del proyecto, incluso la calendarización.

Una lista detallada de empresas y productos de software de gestión de proyectos se puede encontrar en www.infogoal.com/pmc/pmcswr htm.

8 Las herramientas expuestas representan una muestra de esta categoría. En casi todos los casos los nombres de las mismas son marcas registradas de sus respectivos desarrolladores.

24.5.1 Cronogramas

Cuando se crea una calendarización de proyecto del software, el planificador comienza con un conjunto de tareas (la estructura de análisis del trabajo). Si se emplean herramientas automatizadas, el análisis del trabajo se introduce como una red de tareas o esbozo de tareas. Entonces se introduce el esfuerzo, la duración y la fecha de inicio de cada tarea. Además, las tareas se pueden asignar a individuos específicos.

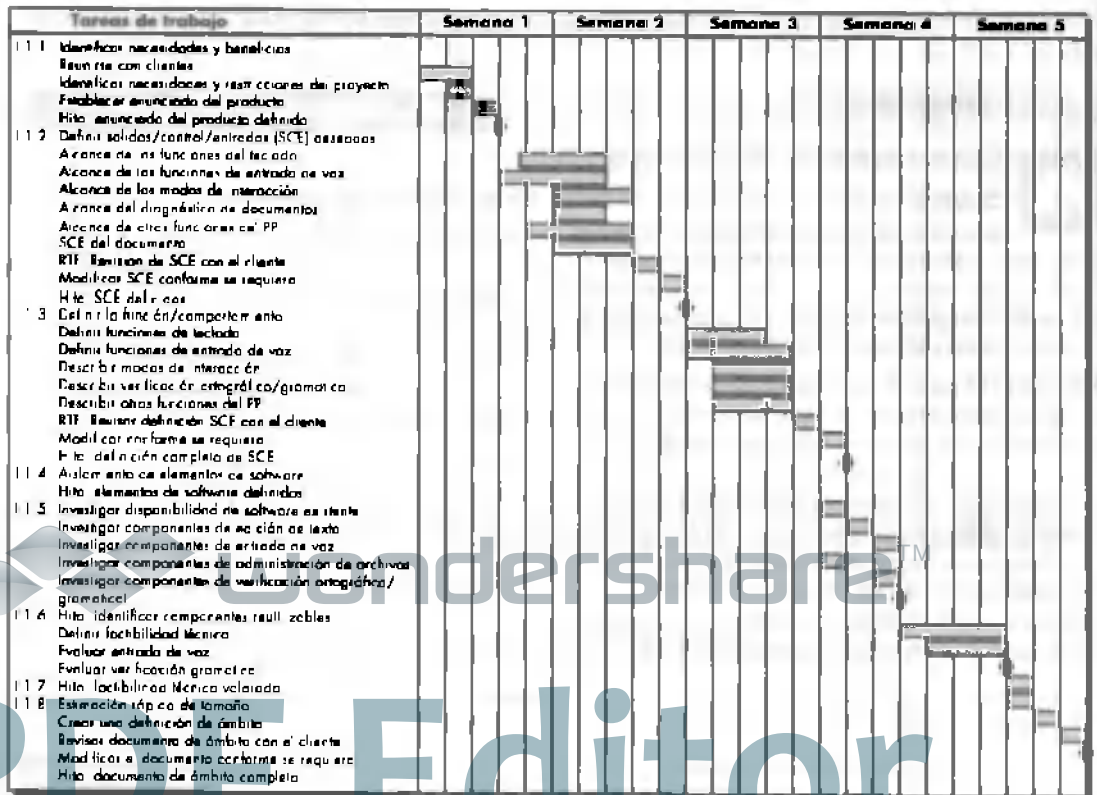
PUNTO CLAVE

Un cronograma permite determinar qué tareas se realizarán en un punto dado en el tiempo

Como consecuencia de esta entrada, se genera un *cronograma*, también llamado *gráfico Gantt*. Es posible desarrollar un cronograma para todo el proyecto. De manera alternativa, se pueden desarrollar cronogramas separados para cada función del proyecto o para cada individuo que trabaje en él.

La figura 24.3 ilustra el formato de un cronograma. Muestra una parte de la calendarización de un proyecto de software que resalta la tarea de determinación del ámbito del concepto para un producto de software procesador de palabra (PP). Todas las tareas del proyecto (para la determinación del ámbito del concepto) se citan en la columna de la izquierda. Las barras horizontales indican la duración de cada ta-

FIGURA 24.3 Ejemplo de cronograma



rea Cuando en el calendario se presentan al mismo tiempo múltiples barras, se implica la concurrencia de la tarea. Los diamantes indican hitos

Una vez ingresada la información necesaria para generar el cronograma, la mayoría de las herramientas de calendarización de proyectos de software producen *tablas de proyecto*: una lista tabular de todas las tareas del proyecto, sus fechas de inicio y conclusión, planeadas y reales, y una variedad de información relacionada (figura 24.4). Si se utilizan en conjunto con el cronograma, las tablas de proyecto permiten que el gestor del proyecto dé seguimiento al progreso.

24.5.2 Seguimiento de la calendarización

La calendarización del proyecto proporciona un mapa de carreteras al gestor del proyecto de software. Si se ha desarrollado de manera adecuada, la calendarización del proyecto define las tareas e hitos que se deben seguir y controlar conforme avance el proyecto El seguimiento se puede hacer de diferentes maneras:

- Con la realización periódica de reuniones para valorar el estado del proyecto en las cuales cada uno de los miembros del equipo informa del progreso y los problemas.
- Con la evaluación de los resultados de todas las revisiones realizadas a lo largo del proceso de ingeniería del software.
- Con la determinación de si se han logrado los hitos formales del proyecto en la fecha programada (los diamantes que se muestran en la figura 24.3).
- Al comparar la fecha de inicio real con la fecha de inicio prevista para cada tarea del proyecto mencionada en la tabla de recursos (figura 24 4)

Figura 24.4 Ejemplo de tabla de recursos.

Tareas de trabajo	Inicio previsto	Inicio real	Terminación prevista	Terminación real	Personas asignadas	Esfuerzo asignado	Notas
Identificar necesidades y beneficios							
Reunirse con los clientes	sem 1, d1	sem 1 d1	sem 1, d2	sem 1, d2	BLS	2 p-d	
Identificar necesidades y restricciones de proyecto	sem 1, d2	sem 1 d2	sem 1, d2	sem 1, d2	JPP	1 p-d	
Establecer alcance del producto	sem 1, d3	sem 1, d3	sem 1, d3	sem 1, d3	BLS/JPP	1 p-d	
Han anunciado de producto definido	sem 1 d3	sem 1, d3	sem 1 d3	sem 1, d3			
Definir salidas/control/entradas (SCE) deseados							
Alcance de las funciones del teclado	sem 1 d4	sem 1 d4	sem 2, d2		BLS	1.5 p-d	
Alcance de las funciones de entrada de voz	sem 1 d3	sem 1, d3	sem 2, d2		JPP	2 p-d	
Alcance de los modos de interacción	sem 2, d1		sem 2, d3		MLL	1 p-d	
Alcance del diagnóstico de documentos	sem 2, d1		sem 2, d2		BLS	1.5 p-d	
Alcance de otras funciones del PP	sem 1, d4	sem 1 d4	sem 2, d3		JPP	2 p-d	
Documento SCE	sem 2, d1		sem 2 d3		MLL	3 p-d	
RTF: revisión de SCE con el cliente	sem 2 d3		sem 2 d3		Todos	3 p-d	
Modificar SCE conforme se requiera	sem 2, d4		sem 2 d4		Todos	2 p-d	
Hito: SCE definidos	sem 2, d5		sem 2, d5				
Definir la función/comportamiento							

- Al reunirse de manera informal con los trabajadores para obtener su evaluación subjetiva del progreso hasta la fecha y los problemas que se vislumbran
- Con el uso del análisis del valor obtenido (sección 24.6) para evaluar el progreso cuantitativamente.

En realidad, todas estas técnicas de seguimiento las utilizan los gestores de proyecto experimentados.

> "La regla básica del reporte del estado del software se puede resumir en una sola frase: no hay sorpresas."
>
> **Capers Jones**

El mejor indicador de progreso es la terminación y revisión exitosa de un producto de trabajo de software definido.

El control lo emplea el gestor del proyecto para administrar los recursos del proyecto, lidiar con los problemas y dirigir al personal del proyecto. Si las cosas van bien (es decir, el proyecto está dentro del calendario y del presupuesto, las revisiones indican que se está realizando un progreso real y que los hitos se han alcanzado), el control es ligero. Pero cuando ocurren problemas, el gestor del proyecto debe ejercer control para solucionarlos tan pronto como sea posible. Cuando se haya diagnosticado el problema se destinan recursos adicionales al área correspondiente reubicando personal o redefiniendo la calendarización.

Cuando enfrentan severas presiones por la fecha límite, los gestores de proyecto experimentados en ocasiones emplean una calendarización de proyecto y técnica de control llamada *time-boxing* (encajonamiento de tiempo) [ZAH95]. Esta estrategia reconoce que el producto completo no podrá entregarse en la fecha límite programada. Por lo tanto, se elige un paradigma de software incremental (capítulo 3) y se elabora una calendarización para cada entrega incremental.

Entonces se encajonan en el tiempo las tareas asociadas con cada incremento. Esto significa que la calendarización para cada tarea se ajusta al trabajar hacia atrás desde la fecha de entrega para cada incremento. Se coloca una "caja" alrededor de cada tarea. Cuando una tarea se acerca al límite de su caja de tiempo (más o menos 10 por ciento), el trabajo se detiene y comienza la siguiente tarea.

La reacción inicial al enfoque del encajonamiento de tiempo usualmente es negativa: si el trabajo se termina, ¿cómo se procederá? La respuesta se encuentra en la forma en se realiza el trabajo. Cuando se llegue al límite de la caja de tiempo, es probable que se haya completado[9] 90 por ciento de la tarea. El restante 10 por ciento, aunque importante, puede 1) demorarse hasta el siguiente incremento o 2) completarse más tarde si se requiere. Más que quedarse "empantanado" en la tarea, el proyecto avanza hacia la fecha de entrega.

9 Un cínico puede recordar el dicho: el 90 por ciento del sistema toma 10 por ciento del tiempo; el restante 10 por ciento del sistema toma 90 por ciento del tiempo.

24.5.3 Seguimiento del progreso en un proyecto OO

Aunque un modelo iterativo es el mejor marco de trabajo para un proyecto OO, el paralelismo de las tareas dificulta el seguimiento del proyecto. El gestor del proyecto puede tener dificultades al establecer hitos significativos para un proyecto OO debido a varias cosas diferentes que ocurren a la vez. En general, los siguientes hitos principales se pueden considerar "completados" cuando se alcanzan los criterios anotados.

Hitos técnicos: análisis OO completado

- Se han definido y revisado todas las clases y la jerarquía de clase
- Se han definido y revisado los atributos de clase y las operaciones asociadas con una clase.
- Se han establecido y revisado las relaciones de clase (capítulo 8).
- Se ha creado y revisado un modelo de comportamiento (capítulo 8)
- Se han anotado las clases reutilizables.

Hitos técnicos: diseño OO completado

- Se ha definido y revisado el conjunto de subsistemas (capítulo 9).
- Las clases se han revisado y asignado a los subsistemas.
- Se ha establecido y revisado la asignación de tareas.
- Se han identificado las responsabilidades y colaboraciones (capítulos 8 y 9)
- Se ha creado y revisado el diseño de las clases.
- Se ha creado y revisado el modelo de comunicación.

Hitos técnicos: programación OO completada

- Cada nueva clase se ha implementado en código a partir del modelo de diseño.
- Se han implementado las clases obtenidas (de una librería de reutilización).
- Se ha construido el prototipo o incremento.

Hitos técnicos: prueba OO

- Se han revisado la corrección y que estén completos el análisis OO y los modelos de diseño.
- Se ha desarrollado y revisado una red clase-responsabilidad-colaboración (capítulo 8).
- Se han diseñado casos de prueba y se han llevado a cabo pruebas al nivel de clase (capítulo 14) para cada clase.
- Se han diseñado casos de prueba se han completado pruebas de agrupamientos (capítulo 14) y se han integrado las clases
- Se han completado las pruebas al nivel de sistema.

CONSEJO

La depuración y prueba son simultáneas. Con frecuencia, el estado de la depuración se evalúa al considerar el tipo y número de errores (bugs) "abiertos".

Recuérdese que el modelo de proceso OO es iterativo, cada uno de estos hitos pue de revisarse conforme diferentes incrementos se entreguen al cliente

HOGARSEGURO

Seguimiento de la calendarización

El escenario: Oficina de Doug Miller, antes del inicio del proyecto de software *HogarSeguro*.

Los actores: Doug Miller (gerente del equipo de ingeniería del software de *HogarSeguro*) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería del software del producto.

La conversación:

Doug (mira una diapositiva PowerPoint): La calendarización para el primer incremento de *HogarSeguro* parece razonable, pero tendremos problemas para darle seguimiento al progreso

Vinod (su rostro se nota preocupado): ¿Por qué? Hemos calendarizado las tareas en una base diaria, llena de productos de trabajo, y nos hemos asegurado que nos estamos asignando recursos de más

Doug: Todo está bien, ¿pero cómo sabemos cuándo está completo el modelo de análisis para el primer incremento?

Jamie: Las cosas son iterativas, por eso es difícil.

Doug: Entiendo eso, pero... bueno, por ejemplo, tomamos *análisis de clases definido*. Tú indicaste esto como un hito.

Vinod: Así es.

Doug: ¿Quién hizo esa determinación?

Jamie (irritada): Están hechas cuando están hechas

Doug: Eso no es suficientemente bueno, Jamie. Tenemos que calendarizar RTF [revisiones técnicas formales, capítulo 26] y no lo han hecho. La conclusión exitosa de una revisión en el modelo de análisis, por ejemplo, es un hito razonable. ¿Entendido?

Jamie (malhumorada): Está bien, de vuelta a la mesa de dibujo.

Doug: No les debe tomar más de una hora hacer las correcciones... todos los demás pueden comenzar cuando

24.6 ANÁLISIS DEL VALOR GANADO

El valor ganado ofrece un indicio cuantitativo del progreso.

En la sección 24.5 se trataron algunos enfoques cualitativos en cuanto al seguimiento del proyecto. Cada uno ofrece al gestor del proyecto un indicio del progreso, pero una evaluación de la información proporcionada es un poco subjetiva. Es razonable preguntar si existe una técnica cuantitativa para evaluar el progreso conforme el equipo de software avanza a través de las tareas de trabajo asignadas en la calendarización del proyecto. De hecho, existe una técnica para realizar análisis cuantitativos del progreso. Se llama *análisis del valor ganado* (AVG). Humphrey [HUM95] comenta el valor ganado en la forma siguiente:

> El sistema de valor ganado proporciona una escala de valor común para cualquier tarea [de proyecto de software], sin importar el tipo de trabajo que se realiza. Se estiman las horas totales para realizar todo el proyecto y a cada tarea se le da un valor ganado con base en su porcentaje estimado del total

Enunciado en una forma más simple, el valor ganado es una medida del progreso. Permite valorar el "porcentaje realizado" de un proyecto reempleando el análisis cuantitativo en lugar de apoyarse en una opinión personal. De hecho, Fleming y Kop-

pleman [FLE98] argumentan que el análisis del valor ganado "ofrece lecturas precisas y confiables del desempeño desde un momento tan temprano del proyecto como el 15 por ciento realizado."

Para determinar el valor ganado se realizan los siguientes pasos:

¿Cómo se calcula el valor ganado y cómo se utiliza para evaluar el progreso?

1. Se determina el *costo presupuestado para el trabajo calendarizado* (CPTC) respecto de cada tarea de trabajo representada en la calendarización. Durante la estimación se planifica el trabajo (en personas-hora o personas-día) de cada tarea de ingeniería. Por lo tanto, $CPTC_i$ es el esfuerzo palnificado para la tarea de trabajo *i*. Para determinar el progreso en un punto dado en la calendarización del proyecto, el valor de CPTC es la suma de los valores $CPTC_i$ para todas la tareas de trabajo que deben estar completadas en dicho punto en el tiempo en la calendarización del producto.
2. Los valores CPTC para todas las tareas de trabajo se resumen para obtener el *presupuesto a la terminación*, PAT. Por lo tanto,

   $PAT = \Sigma (PTC_k)$ para todas las tareas *k*.

3. A continuación se calcula el *costo presupuestado del trabajo realizado* (CPTR). El valor de CPTR es la suma de los valores CPTC para todas las tareas de trabajo que en realidad se han completado en un punto en el tiempo en la calendarización del proyecto.

Wilkens [WIL99] señala que "la distinción entre CPTC y CPTR es que la primera representa el presupuesto de las actividades que se planearon completar, y la última, el presupuesto de las actividades que en realidad se completaron" . Dados los valores de CPTC, PAT y CPTR, se pueden calcular importantes indicadores del progreso:

Índice de desempeño en la calendarización, IDCa = CPTR/CPTC
Varianza en la calendarización, VC = CPTR − CPTC

Referencia Web

Un vendedor de herramientas acerca del análisis del valor ganado se puede encontrar en www.acq.osd.mil/pm/.

IDCa es un indicador de la eficiencia con la que el proyecto utiliza los recursos calendarizados. Un valor IDCa cercano a 1.0 indica una ejecución eficiente del proyecto calendarizado. VC simplemente es un indicador absoluto de la variación a partir de la calendarización planeada

Porcentaje calendarizado para terminación = CPTC/PAT

ofrece un indicador del porcentaje de trabajo que debe estar completado en el tiempo *t*

Porcentaje de completado = CPTR/PAT

ofrece una indicación cuantitativa del porcentaje de avance del proyecto en un punto dado en el tiempo, *t*.

También es posible calcular el *costo real del trabajo realizado*, CRTR. El valor para CRTR es la suma del esfuerzo realmente utilizado en las tareas de trabajo que se han completado en un punto en el tiempo en la calendarización del proyecto. Entonces es posible calcular

Índice de desempeño del costo, IDCo = CPTR/CRTR
Varianza del costo, VC = CPTR − CRTR

Un valor de IDCo cercano a 1.0 ofrece un fuerte indicador de que el proyecto está dentro de su presupuesto definido. VC es un indicador absoluto del ahorro en costo (contra los costos planeados) o recortes en una etapa particular de un proyecto.

Al igual que un radar en el horizonte, el análisis del valor ganado ilumina las dificultades en la calendarización antes de que puedan advertirse de alguna otra forma. Esto permite que el gestor del proyecto de software tome medidas correctivas antes de que se desarrolle una crisis en el proyecto.

24.7 RESUMEN

La calendarización es la culminación de una actividad de planificación que es un componente principal de la gestión del proyecto de software. Cuando se combina con métodos de estimación y análisis de riesgo, la calendarización establece un mapa de carreteras para el gestor de proyectos.

La calendarización comienza con el proceso de descomposición. Las características del proyecto se utilizan para adaptar un conjunto de tareas apropiado al trabajo que se realizará. Una red de tareas bosqueja cada tarea de ingeniería, su dependencia de otras tareas y su duración proyectada. La red de tareas se utiliza para calcular la trayectoria crítica, un cronograma y una variedad de información del proyecto. Al usar la calendarización como guía, el gestor del proyecto puede dar seguimiento y controlar cada paso en el proceso del software.

REFERENCIAS

[BRO95] Brooks, M., *The Mythical Man-Month*, edición de aniversario, Addison-Wesley, 1995.
[FLE98] Fleming, Q. W., y J. M. Koppelman, "Earned Value Project Management", *Crosstalk*, vol. 11, núm. 7, julio de 1998, p. 19.
[HUM95] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, 1995.
[NOR70] Norden, P., "Useful Tools for Project Management", en *Management of Production*, M. K. Starr (ed.), Penguin Books, 1970.
[PAG85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, pp. 90-91.
[PRE99] Pressman, R. S., *Adaptable Process Model*, R. S. Pressman & Associates, 1999.
[PUT78] Putnam, L., y W. Myers, *Measures for Excellence*, Yourdon Press, 1992.
[WIL99] Wilkens, T. T., "Earned Value, Clear and Simple", Primavera Systems, 1 de abril de 1999, p. 2.
[ZAH95] Zahniser, R., "Time-boxing for Top Team Performance", en *Software Development*, marzo de 1995, pp. 34-38.

PROBLEMAS Y PUNTOS A CONSIDERAR

24.1. Las fechas límite "irracionales" son un hecho de la vida en el negocio del software. ¿Cómo se debe proceder si se enfrenta con una?

24.2. ¿Cuál es la diferencia entre una calendarización macroscópica y una calendarización detallada? ¿Es posible gestionar un proyecto si sólo se desarrolla una calendarización macroscópica? ¿Por qué?

24.3. ¿Existe algún caso donde un hito de un proyecto de software no esté ligado a una revisión? Si es así, ofrecer uno o más ejemplos

24.4. La "sobrecarga de comunicación" puede ocurrir cuando muchas personas trabajan en un proyecto de software. El tiempo empleado en la comunicación con otros reduce la productividad individual (LDC/persona-mes) y el resultado es menor productividad para el equipo. Ilustrar (cuantitativamente) cómo los ingenieros versados en las buenas prácticas de ingeniería del software y que usan revisiones técnicas formales pueden aumentar la tasa de producción de un equipo (cuando se compara con la suma de las tasas de producción individuales). Sugerencia: se puede suponer que las revisiones reducen la reelaboración y que ésta puede explicar el 20-40 por ciento del tiempo de una persona

24.5. Aunque agregar personal a un proyecto de software retrasado puede retrasarlo más, existen circunstancias en las cuales esto no es cierto. Descríbanse

24.6. La relación entre personal y tiempo es enormemente no lineal. Mediante la ecuación del software de Putnam (descrita en la sección 24.2.2), desarrollar una tabla que relacione el número de personas con la duración del proyecto para un proyecto de software que requiera 50 000 LDC y 15 personas-año de esfuerzo (el parámetro de productividad es 5 000). Supóngase que el software debe entregarse en un máximo de 24 meses o en un mínimo de 12.

24.7. Suponga que una universidad lo ha contratado para desarrollar un sistema de registro en línea a cursos (SRELC). Primero, actúe como el cliente (si es estudiante, ¡esto debe ser sencillo!) y especifique las características de un buen sistema. (Alternativamente, su instructor le proporcionará un conjunto de requisitos preliminares para el sistema.) Por medio de los métodos de estimación tratados en el capítulo 23, desarrolle una estimación del esfuerzo y la duración para el SRELC. Sugiera cómo:

a) Definirla las actividades de trabajo paralelas durante el proyecto de SRELC.
b) Distribuiría el esfuerzo a lo largo del proyecto.
c) Establecería hitos para el proyecto.

24.8. Seleccione un conjunto de tareas apropiado para el proyecto del SRELC.

24.9. Defina una red de tareas para el SRELC o, alternativamente, para otro proyecto de software que le interese. Asegúrese de mostrar las tareas e hitos y de vincular las estimaciones de esfuerzo y duración a cada tarea. Si es posible, utilice una herramienta automatizada de calendarización para realizar este trabajo

24.10. Si está disponible una herramienta automatizada de calendarización, determine la trayectoria crítica para la red definida en el problema 24.9

24.11. Mediante una herramienta de calendarización (si está disponible) o papel y lápiz (si es necesario), desarrolle un cronograma para el proyecto del SRELC

24.12. Suponga que es gestor de proyectos de software y que se le ha pedido calcular las estadísticas del valor ganado de un pequeño proyecto. El proyecto tiene planeadas 56 tareas de trabajo que se estima requerirán 582 personas-día para completarse. En el momento en que se le pide realizar el análisis del valor ganado, 12 tareas se han completado. Sin embargo, la calendarización del proyecto indica que se tenían que haber completado 15. Están disponibles los siguientes datos de la calendarización (en personas-día):

Tarea	Esfuerzo previsto	Esfuerzo real
1	12.0	12.5

Tarea	Esfuerzo previsto	Esfuerzo real
2	15.0	11.0
3	13.0	17.0
4	8 0	9.5
5	9 5	9.0
6	18 0	19.0
7	10 0	10.0
8	4.0	4 5
9	12.0	10.0
10	6.0	6.5
11	14.0	4.0
12	14.0	14.5
13	16.0	—
14	6.0	—
15	8 0	—

Calcule el IDCa, la varianza de calendarización, el porcentaje calendarizado para terminación el porcentaje completado, IDCo y la varianza de costo para el proyecto.

Otras lecturas y fuentes de información

Casi cualquier libro acerca de gestión de proyectos de software contiene una exposición de la calendarización. El Project Management Institute (*PMBOK Guide*, PMI, 2001), Wysoki y sus colegas (*Effective Project Management*, Wiley, 2000), Lewis (*Project Planning Scheduling and Control*, 3a ed., McGraw-Hill, 2000), Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques*, 3a ed., Wiley, 2000), McConnell (*Software Project Cost Survival Guide*, Microsoft Press, 1998) y Roetzheim y Beasley (*Software Project Cost and Schedule Estimating Best Practices*, Prentice-Hall, 1997) contienen análisis valiosos del tema. Boddie (Crunch Mode, Prentice-Hall, 1987) ha escrito un libro para todos los gestores que "tienen 90 días para hacer un proyecto de seis meses".

McConnell (*Rapid Development*, Microsoft Press, 1996) presenta una excelente exposición de los conflictos que conducen a calendarizaciones de proyectos de software demasiado optimistas y qué puede hacer acerca de ello. O'Connell (*How to Run Successful Projects II: The Silver Bullet*, Prentice-Hall, 1997) presenta un enfoque paso a paso para la gestión de proyectos que le ayudará a desarrollar una calendarización realista para sus proyectos.

Webb y Wake (*Using Earned Value: A Project Manager's Guide*, Ashgate Publishing, 2003) y Fleming y Koppelman (*Earned Value Project Management*, Project Management Institute Publications, 1996) examinan con considerable detalle el uso de las técnicas del valor ganado para planificación, seguimiento y control de proyectos.

En Internet hay disponible una gran variedad de fuentes de información acerca de la calendarización de proyectos de software. Una lista actualizada de referencias de la World Wide Web se puede encontrar en el sitio Web de SEPA: **http://www.mhhe.com/pressman.**

CAPÍTULO

25

GESTIÓN DEL RIESGO

CONCEPTOS CLAVE

En su libro acerca del análisis y la gestión del riesgo, Robert Charette [CHA89] presenta una definición conceptual de riesgo:

> En primer lugar, el riesgo se relaciona con los acontecimientos futuros. El hoy y el ayer están más allá de esta relación, pues ya se ha cosechado lo que previamente se sembró con nuestras acciones pasadas. La pregunta es, ¿podemos, por tanto, al cambiar nuestras acciones presentes, crear en lo futuro una oportunidad para una situación diferente, y esperanzadoramente mejor, para nosotros mismos? Esto significa, en segundo lugar, que el riesgo implica cambio, como cambios de mentalidad, opinión, acciones o lugares... [en tercer lugar] el riesgo implica elección y la incertidumbre que ésta conlleva. Por ende, paradójicamente, el riesgo, al igual que la muerte y los impuestos, es una de las pocas certezas en la vida.

Cuando el riesgo se considera en el contexto de la ingeniería del software, las tres bases conceptuales de Charette siempre se evidencian. El futuro es una preocupación de primer orden: ¿qué riesgos causarían que el proyecto de software salga mal? El cambio es una preocupación central: ¿cómo afectarán la actualidad y el éxito global los cambios en los requisitos del cliente, las tecnologías de desarrollo, los entornos que se tienen como objetivo y todas las otras entidades vinculadas con el proyecto? Por último, es necesario enfrentar las opciones: ¿qué métodos y herramientas se deben usar, cuántas personas deben estar involucradas, cuánto énfasis sobre la calidad es "suficiente"?

Peter Drucker [DRU75] dijo una vez: "Aunque es vano intentar eliminar el riesgo, y cuestionable intentar minimizarlo, es esencial que los riesgos tomados sean los riesgos correctos". Antes de identificar los "riesgos correctos" que se tomarán durante un proyecto de software, es importante identificar los riesgos que son obvios para los gestores y profesionales.

UN VISTAZO RÁPIDO

¿Qué es? El análisis y la gestión del riesgo son una serie de pasos que ayudan a un equipo de software a comprender y manejar la incertidumbre. Muchos problemas pueden desbordar un proyecto de software. Un riesgo es un problema potencial: puede ocurrir o no. Pero, sin importar el resultado, en realidad es una buena idea identificarlo, evaluar la probabilidad de que ocurra, estimar su impacto y establecer un plan de contingencia en caso de que el problema se presente.

¿Quién lo hace? Todos los involucrados en el proceso de software (gestores, ingenieros y participantes) intervienen en el análisis y la gestión del riesgo.

¿Por qué es importante? Piénsese en el lema de los boy scout: "estar preparado". El software es una empresa difícil. Muchas cosas pueden salir mal y, francamente, con mucha frecuencia lo

hacen. Por esta razón estar preparados (al comprender los riesgos y tomar medidas proactivas para evitarlos o gestionarlos) es un elemento clave de una buena gestión de proyecto de software

¿Cuáles son los pasos? Reconocer qué puede salir mal es el primer paso, llamado "identificación del riesgo". A continuación se analiza cada riesgo para determinar la probabilidad de que ocurrirá y el daño que causará si en efecto ocurre. Una vez establecida esta información, los riesgos se clasifican según su probabilidad e impacto. Finalmente, se desarrolla un plan para gestionar aquellos riesgos con gran probabilidad y alto impacto

¿Cuál es el producto obtenido? Se produce un plan de reducción, supervisión y gestión del riesgo (RSGR) o un conjunto de hojas informativas de riesgo.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Los riesgos que se analizan y gestionan deben proceder de un estudio amplio del personal, el producto, el proceso y el proyecto. El plan RSGR debe revisarse conforme el proyecto avanza para asegurarse de que los riesgos están actualizados. Los planes de contingencia para la gestión del riesgo deben ser realistas.

25.1 Estrategias de riesgo reactivas y proactivas

Las estrategias de riesgo reactivas han sido jocosamente denominadas la "escuela Indiana Jones de gestión del riesgo" [THO92]. En las películas de la década de 1980 que llevaban su nombre, Indiana Jones, cuando enfrentaba alguna dificultad abrumadora, invariablemente decía: "¡No te preocupes, pensaré en algo!". Al no preocuparse nunca por los problemas, sino hasta que ocurrían, Indy reaccionaba en alguna forma heroica.

> "Si usted no ataca activamente los riesgos, ellos lo atacarán activamente."
>
> **Tom Gilb**

Tristemente, el gestor promedio de proyectos de software no es Indiana Jones, y los miembros del equipo de proyecto de software no son sus confiables compañeros. Más aún, la mayoría de los equipos de software se apoya exclusivamente en las estrategias de riesgo reactivas. Los riesgos se apartan para tratarlos, lo que puede convertirlos en problemas reales. Más usualmente, el equipo de software no hace nada acerca de los riesgos hasta que algo sale mal. Entonces el equipo se precipita en la acción con la finalidad de corregir el problema rápidamente. Con frecuencia a esto se le llama el *modo bombero*. Cuando esto falla, la "gestión de crisis" [CHA92] asume el control y el proyecto está en un verdadero peligro.

Una estrategia considerablemente más inteligente para la gestión del riesgo es ser proactivo. Una estrategia proactiva comienza mucho antes de que se inicie el trabajo técnico. Se identifican los riesgos potenciales, se valoran su probabilidad e impacto, y se les clasifica según su importancia. Luego, el equipo de software establece un plan para gestionar el riesgo. El objetivo principal es evitar el riesgo, pero debido a que no todos los riesgos pueden evitarse, el equipo trabaja para desarrollar un plan

de contingencia que le permitirá responder en una forma controlada y efectiva. A lo largo del resto del capítulo se examina la estrategia proactiva para la gestión del riesgo.

25.2 RIESGOS DEL SOFTWARE

Aunque hay un considerable debate en torno a la definición propia para el riesgo de software, existe un acuerdo general en que el riesgo siempre involucra dos características [HIG95]:

- *Incertidumbre:* el riesgo puede o no ocurrir; esto es, no existen riesgos 100% probables.[1]
- *Pérdida:* si el riesgo se convierte en realidad, ocurrirán consecuencias o pérdidas indeseables.

Cuando se analizan los riesgos es importante cuantificar el grado de incertidumbre y el grado de pérdida asociado con cada riesgo. Esto se logra considerando diferentes categorías de riesgos.

¿Qué tipos de riesgos es probable encontrar conforme se construya software?

Los *riesgos del proyecto* amenazan el plan del proyecto. Es decir, si los riesgos del proyecto se vuelven reales es probable que la calendarización del proyecto se altere y que los costos aumenten. Los riesgos del proyecto identifican potenciales problemas en presupuesto, calendarización, personal (plantillas y organización), recursos, participantes y requisitos, y su impacto sobre un proyecto de software. En el capítulo 23 la complejidad, tamaño y grado de incertidumbre estructural del proyecto también se definieron como factores de riesgo del proyecto (y de la estimación).

Los *riesgos técnicos* amenazan la calidad y actualidad del software que se producirá. Si un riesgo técnico se vuelve real, la implementación se torna difícil o imposible. Los riesgos técnicos identifican potenciales problemas en diseño, implementación, interfaz, verificación y mantenimiento. Además, también son factores de riesgo la ambigüedad de la especificación, la incertidumbre técnica, la obsolescencia técnica y la tecnología "de punta". Los riesgos técnicos ocurren porque el problema es más difícil de resolver de lo que en un principio se pensó que sería.

Los *riesgos de negocios* amenazan la viabilidad del software que se construirá. Estos riesgos con frecuencia ponen en peligro el proyecto o el producto. Los candidatos para los cinco mayores riesgos de negocios son 1) la construcción de un producto o sistema excelente que en realidad nadie quiere (riesgo de mercado), 2) la construcción de un producto que ya no encaja en la estrategia comercial global de la compañía (riesgo de estrategia), 3) la construcción de un producto que la fuerza de ventas no sabe cómo vender (riesgo de ventas), 4) la pérdida del apoyo de los altos ejecutivos debido a un cambio en el enfoque o en el personal (riesgo administrativo), y 5) la pérdida presupuestaria o del personal asignado (riesgo presupuestal).

1 Un riesgo 100 por ciento probable es una restricción sobre el proyecto de software.

Es extremadamente importante destacar que la simple clasificación de los riesgos no siempre funciona. Algunos riesgos simplemente son impredecibles.

Charette [CHA89] ha propuesto otra categorización general de los riesgos. Los *riesgos conocidos* son aquellos susceptibles de descubrirse después de una evaluación cuidadosa del plan del proyecto, del entorno de negocios y técnico dentro de los cuales se desarrollará el proyecto, y otras fuentes de información confiables (por ejemplo, fechas de entrega irreales, falta de requisitos documentados o de ámbito del software, pobre entorno de desarrollo). Los *riesgos predecibles* se extrapolan de la experiencia con proyectos previos (por ejemplo, cambios en el personal, mala comunicación con el cliente, disminución del esfuerzo del personal conforme se atienden las solicitudes de mantenimiento). Los *riesgos impredecibles* son el comodín de la baraja. Pueden y de hecho ocurren, pero son extremadamente difíciles de identificar con antelación.

INFORMACIÓN

Siete principios de la gestión de riesgos

El Software Engineering Institute (SEI) (www.sei.cmu.edu) identifica siete principios que "ofrecen un marco de trabajo para lograr una gestión de riesgos efectiva". Dichos principios son:

Mantenimiento de una perspectiva global: ver los riesgos de software dentro del contexto del sistema en el que está un componente y el problema de negocios que se intenta resolver.

Tener una visión previsora: pensar en los riesgos que pudieran surgir en lo futuro (por ejemplo, debido a cambios en el software); establecer planes de contingencia de modo que los eventos futuros sean manejables.

Alentar la comunicación abierta: si alguien establece un riesgo potencial, no lo descarte. Si un riesgo se propone de una manera informal, considérelo. Aliente a todos los participantes y usuarios a sugerir riesgos en cualquier momento.

Integración: en el proceso del software debe estar integrada una consideración de los riesgos.

Enfatizar un proceso continuo: el equipo debe estar atento a lo largo de todo el proceso de software, modificar los riesgos identificados conforme se tenga más información y agregar unos nuevos a medida que se logre un mejor criterio.

Desarrollo de una visión conjunta del producto: si todos los participantes comparten la misma visión del software, es probable que ocurra mejor identificación y evaluación de riesgos.

Alentar el trabajo en equipo: los talentos, habilidades y conocimiento de los participantes se deben mezclar cuando se lleven a cabo actividades de gestión de riesgos.

25.3 IDENTIFICACIÓN DE RIESGOS

La identificación de los riesgos es un intento sistemático encaminado a especificar las amenazas al plan del proyecto (estimaciones, calendarización, carga de recursos, etc.). Al identificar los riesgos conocidos y predecibles, el gestor del proyecto da un primer paso para evitarlos cuando es posible y a controlarlos cuando es necesario.

Existen dos tipos distintos de riesgos para cada una de las categorías que se han presentado en la sección 25.2: los riesgos genéricos y los riesgos específicos del producto. Los *riesgos genéricos* son una amenaza potencial para todo el proyecto de software. Los *riesgos específicos del producto* los pueden identificar sólo aquellos con

un claro conocimiento de la tecnología, el personal y el entorno específico del software que se construirá. Los riesgos específicos del producto se identifican examinando el plan del proyecto y la declaración del ámbito del software, así como desarrollando una respuesta para la siguiente pregunta. "¿Qué características especiales de este producto podrían amenazar el plan del proyecto?".

> "Los proyectos sin riesgos reales son perdedores. Estos proyectos casi siempre están desprovistos de beneficios; por ello no fueron realizados años atrás."
>
> **Tom DeMarco y Tim Lister**

CONSEJO

Aunque es importante considerar los riesgos genéricos, los riesgos específicos del producto son los que provocan la mayoría de los dolores de cabeza. Asegúrese de emplear tiempo para identificar tantos riesgos específicos del producto como sea posible.

Un método para identificar riesgos consiste en crear una lista de verificación de riesgos. Con ésta se pueden identificar riesgos y enfocarse en algún subconjunto de riesgos conocidos y predecibles en las siguientes subcategorías genéricas:

- *Tamaño del producto:* riesgos asociados con el tamaño global del software que se construirá o modificará.
- *Impacto en el negocio:* riesgos asociados con las restricciones que impone la gerencia o el mercado.
- *Características del cliente:* riesgos asociados con la sofisticación del cliente y la habilidad del desarrollador para comunicarse con él en una forma oportuna.
- *Definición del proceso:* riesgos asociados con el grado en el que se ha definido el proceso de software y en que le da seguimiento la organización que lo desarrolla
- *Entorno de desarrollo:* riesgos asociados con la disponibilidad y calidad de las herramientas que se usarán en la construcción del producto
- *Tecnología que construir:* riesgos asociados con la complejidad del sistema que se construirá y la "novedad" de la tecnología que está empaquetada en el sistema.
- *Tamaño y experiencia de la plantilla de personal:* riesgos asociados con la experiencia global técnica y en el proyecto de los ingenieros de software que harán el trabajo.

La lista de verificación de riesgos se puede organizar en diferentes formas. Las preguntas relevantes respecto de cada uno de los tópicos se pueden responder para cada proyecto de software. Las respuestas a estas preguntas permiten que el planificador estime el impacto del riesgo. Un formato diferente de lista de verificación de riesgos simplemente menciona las características relevantes para cada subcategoría genérica. Finalmente, se menciona un conjunto de "componentes y controladores de riesgo" [AFC88] junto con su probabilidad de ocurrencia. Los controladores del desempeño, soporte, costo y calendarización se estudian como respuesta a las últimas preguntas

En la bibliografía se han propuesto varias listas de verificación detalladas para riesgos del proyecto de software (por ejemplo, [SEI93], [KAR96]), las cuales proporcionan una visión útil de los riesgos genéricos para proyectos de software, y se deben usar siempre que se instituyan análisis y gestión del riesgo. Sin embargo, se puede emplear una lista relativamente corta de preguntas [KEI98] para proporcionar un indicio preliminar de si un proyecto está "en riesgo".

25.3.1 Evaluación del riesgo global del proyecto

Las siguientes preguntas se basan en los datos de riesgo obtenidos al entrevistar, en diferentes partes del mundo, a gestores de proyecto de software experimentados [KEI98]. Las preguntas están ordenadas según su importancia relativa en el éxito de un proyecto.

¿El proyecto de software en el que actualmente trabaja enfrenta riesgos serios?

1. ¿Los altos ejecutivos de software y del cliente se han comprometido formalmente para apoyar el proyecto?
2. ¿Los usuarios finales están comprometidos con el proyecto y el sistema/producto que se construirá?
3. ¿Los requisitos los han entendido completamente el equipo de ingeniería de software y sus clientes?
4. ¿Los clientes estuvieron completamente involucrados en la definición de los requisitos?
5. ¿Los usuarios finales tienen expectativas realistas?
6. ¿El ámbito del proyecto es estable?
7. ¿El equipo de ingeniería del software tiene la mezcla correcta de habilidades?
8. ¿Los requisitos del proyecto son estables?
9. ¿El equipo del proyecto tiene experiencia con la tecnología que se implementará?
10. ¿El número de personas en el equipo de proyecto es adecuado para realizar el trabajo?
11. ¿Todos los votantes del cliente/usuario están de acuerdo en la importancia del proyecto y en los requisitos para el sistema/producto que se construirá?

Referencia Web

Riskmaster es una base de datos y herramientas que ayudan a los gestores a identificar, clasificar y controlar los riesgos de proyecto. Se puede encontrar en www.spmn.com.

> "La gestión de riesgos es la gestión de proyectos para adultos."
>
> **Tim Lister**

Si la respuesta a alguna de estas preguntas es negativa se deben instituir sin demora los pasos de reducción, supervisión y gestión. El grado en el que el proyecto está en riesgo es directamente proporcional al número de respuestas negativas a estas preguntas.

25.3.2 Componentes y controladores del riesgo

La Fuerza Aérea de Estados Unidos [AFC88] escribió un folleto con excelentes directrices para la identificación y supresión del riesgo de software. Este enfoque requiere que el gestor del proyecto identifique los controladores del riesgo que afectan los componentes de riesgo del software: desempeño, costo, soporte y calendarización. En el contexto de este estudio los componentes de riesgo se definen en la forma siguiente:

- *Riesgo de desempeño:* grado de incertidumbre de que el producto satisfaga los requisitos y se ajuste al uso que se pretende darle.
- *Riesgo de costo:* grado de incertidumbre de que se mantenga el presupuesto del proyecto.
- *Riesgo de soporte:* grado de incertidumbre de que el software resultante será fácil de corregir, adaptar y mejorar.

FIGURA 25.1 Evaluación del impacto [BOE89].

Componentes / Categoría		Desempeño	Soporte	Costo	Calendarización
Catastrófico	1	El fracaso en la satisfacción de los requisitos resultaría en un fracaso de la misión		El fracaso resulta en el aumento de costos y en demoras en la calendarización con valores esperados que superan 500K dls.	
	2	Cierta reducción en el desempeño técnico	Software que no responde o no se puede soportar	Recortes financieros significativos, probable superación del presupuesto	COI inalcanzable
Crítica	1	El fracaso para satisfacer los requisitos resultaría en un desempeño degradado del sistema hasta un punto donde el éxito de la misión es cuestionable		El fracaso resulta en demoras operativas o incremento de costos con valor esperado de 100K a 500K dólares	
	2	Cierta reducción en el desempeño técnico	Demoras menores en las modificaciones del software	Cierto recorte de recursos financieros, posibles excesos	Posible deslizamientos en la COI
Marginal	1	El fracaso para satisfacer los requisitos resultaría en degradación de la misión secundaria		Deslizamiento de costos, impactos o calendarización re cuperable con valor esperado de 1K a 100K dólares	
	2	Mínima o pequeña reducción en el desempeño técnico	Respuesta de soporte de software	Suficientes recursos financieros	Calendarización alcanzable y realista
Despreciable	1	El fracaso al satisfacer los requisitos crearía inconvenientes o impactos no operativos		El error resulta en costo menor o impacto en la calendarización con valor esperado de menos de 1K dólares	
	2	Ninguna reducción en el desempeño técnico	Software al que fácilmente se le da soporte	Posible superávit presupuestal	COI fácilmente alcanzable

Nota: 1. Consecuencia potencial de errores o fallas de software no detectados.
2. Consecuencia potencial si el resultado deseado no se alcanza.

- *Riesgo de calendarización*: grado de incertidumbre de que se mantenga la calendarización del proyecto y de que el producto se entregue a tiempo.

El impacto de cada controlador de riesgo sobre el componente de riesgo se divide en cuatro categorías de impacto: despreciable, marginal, crítico o catastrófico. En la figura 25.1 [BOE89] se describe una caracterización de las consecuencias potenciales de los errores (hileras etiquetadas 1) o una falla que no permite lograr un resultado deseado (hileras etiquetadas 2). La categoría de impacto se escoge con base en la caracterización que mejor encaje con la descripción en la tabla.

25.4 PROYECCIÓN DEL RIESGO

La *proyección del riesgo*, también llamada *estimación del riesgo*, intenta clasificar cada riesgo en dos formas: 1) la posibilidad o probabilidad de que el riesgo sea real, y 2) las consecuencias de los problemas asociados con el riesgo, en caso de que ocurra. El planificador del proyecto, junto con otros gestores y personal técnico, realizan cuatro pasos en la proyección del riesgo:

1. Establecimiento de una escala que refleje la posibilidad percibida de un riesgo.
2. Delineado de las consecuencias del riesgo.

FIGURA 25.2

Ejemplo de tabla de riesgos antes de la clasificación.

Riesgos	Categoría	Probabilidad	Impacto	RSGR
La estimación del tamaño puede ser significativamente baja	TP	60%	2	
Mayor número de usuarios de los previstos	TP	30%	3	
Menos reutilización que la prevista	TP	70%	2	
Los usuarios finales se resisten al sistema	CO	40%	3	
La fecha límite de entrega estará muy ajustada	CO	50%	2	
Pérdida de fondos	CL	40%	1	
El cliente cambiará requisitos	TP	80%	2	
La tecnología no satisfará las expectativas	RT	30%	1	
Falta de entrenamiento acerca de las herramientas	ED	80%	3	
Personal inexperto	PE	30%	2	
Elevada movilidad del personal	PE	60%	2	

Valores de impacto:
1: catastrófico
2: crítico
3: marginal
4: despreciable

3. Estimación del impacto del riesgo en el proyecto y el producto.
4. Tomar nota de la precisión global de la proyección del riesgo de modo que no haya malas interpretaciones.

La finalidad de estos pasos es considerar los riesgos en tal forma que conduzcan al establecimiento de prioridades. Ningún equipo de software tiene los recursos para enfrentar todos los riesgos potenciales con el mismo grado de rigor. Al priorizar los riesgos el equipo puede asignar los recursos donde tengan el mayor impacto.

25.4.1 Desarrollo de una tabla de riesgos

CONSEJO

...mucho acerca ...software que está ...o de construir y ...e usted ...o: ¿qué puede ...mal? Cree su ...lista y pida a ...s miembros del ...o que hagan lo ...

Una tabla de riesgos ofrece al gestor de un proyecto una técnica simple para la proyección de riesgos.[2] En la figura 25.2 se ilustra un ejemplo de tabla de riesgos

Un equipo de proyecto comienza una lista de todos los riesgos (sin importar cuán remotos sean) en la primera columna de la tabla. Esto se logra con la ayuda de la lista de verificación de riesgos mencionada en la sección 25.3. Cada riesgo se clasifica en la segunda columna (por ejemplo, TP implica un riesgo de tamaño del proyecto, NE implica un riesgo de negocios) En la siguiente columna de la tabla se registra la probabilidad de que ocurra cada riesgo El valor de la probabilidad de cada riesgo lo pueden estimar individualmente los miembros del equipo. Éstos se encuestan en una forma de "todos contra todos" hasta que su evaluación de la probabilidad del riesgo comience a convergir.

A continuación se evalúa el impacto de cada riesgo. Cada componente de riesgo se evalúa mediante la caracterización presentada en la figura 25.1, y se determina una categoría de impacto. Las categorías para cada uno de los cuatro componentes de riesgo (desempeño, soporte, costo y calendarización) se promedian[3] para determinar un valor de impacto global.

Una vez completadas las cuatro primeras columnas de la tabla de riesgos, ésta se ordena según la probabilidad y el impacto. Los riesgos de alta probabilidad y alto impacto se filtran hacia lo alto de la tabla, y los riesgos de baja probabilidad caen al fondo. Esto logra una priorización del riesgo de primer orden.

PUNTO CLAVE

...tabla de riesgos ...ordenada por ...probabilidad e impacto ...para dosificar los ...esgos.

El gestor del proyecto estudia la tabla ordenada resultante y define una línea de corte. La *línea de corte* (dibujada horizontalmente en algún punto en la tabla) implica que sólo los riesgos ubicados sobre la línea tendrán una atención posterior Los riesgos debajo de la línea se reevalúan para lograr una priorización de segundo orden. En la figura 25.3 el impacto y la probabilidad de riesgo influyen de manera distinta en la gestión. Un factor de riesgo que tiene un alto impacto, pero una probabilidad de que suceda muy baja, no debe absorber una cantidad significativa de tiem-

2 Es posible implementar la tabla de riesgos como un modelo en hoja de cálculo. Esto permite una manipulación sencilla y el ordenamiento de las entradas

3 El empleo de un promedio ponderado es factible si algún componente de riesgo tiene mayor relevancia para un proyecto.

po de gestión. Sin embargo, los riesgos de alto impacto con moderada a alta probabilidad y los riesgos de bajo impacto con alta probabilidad deben trasladarse a los pasos de análisis de riesgo que siguen.

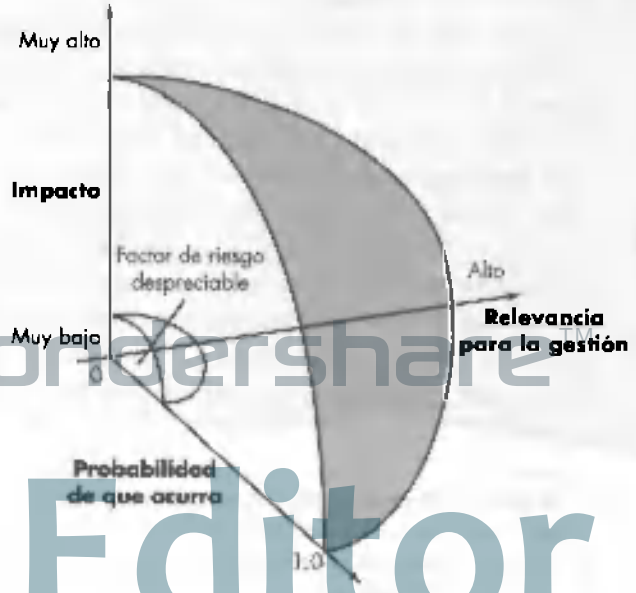
Todos los riesgos ubicados sobre la línea de corte deben gestionarse. La columna rotulada RSGR contiene una referencia que apunta hacia un *Plan de reducción, supervisión y gestión de riesgo* o, alternativamente, una colección de *hojas de información de riesgo* desarrolladas para todos los riesgos que están sobre el corte. En las secciones 25.5 y 25.6 se examinan el plan RSGR y las hojas de información de riesgo.

> "[En la actualidad] nadie tiene el lujo de llegar a conocer una tarea tan bien que no se lleve sorpresas, y las sorpresas significan riesgo."
>
> **Stephen Grey**

La probabilidad del riesgo se determina realizando estimaciones individuales y luego desarrollando un solo valor de consenso. Aunque dicho enfoque es valioso, se han desarrollado técnicas más elaboradas con las cuales determinar la probabilidad del riesgo [AFC88]. Los controladores de riesgo se pueden evaluar sobre una escala de probabilidad cualitativa que tiene los siguientes valores: imposible, improbable, probable y frecuente. Entonces se asocia la probabilidad matemática con cada valor cualitativo (por ejemplo, una probabilidad de 0.7 a 0.95 implica un riesgo enormemente probable).

FIGURA 25.3

Riesgo y preocupaciones de la gestión.



25.4.2 Evaluación del impacto del riesgo

Tres factores afectan las consecuencias que son probables si un riesgo ocurre: su naturaleza, ámbito y tiempo. La naturaleza indica los problemas que son probables si ocurre. Por ejemplo, una interfaz externa mal definida hacia el hardware del cliente (un riesgo técnico) evitará un diseño y prueba tempranos y tal vez genere problemas de integración de sistema más tarde. El ámbito combina la severidad (¿cuán serio es?) con su distribución global (¿cuánto del proyecto se afectará, o cuántos clientes resultarán dañados?). Finalmente, el tiempo de un riesgo considera cuándo y durante qué periodo se sentirá el impacto. En la mayoría de los casos un gestor de proyecto tal vez quiera que ocurran las "malas noticias" tan pronto como sea posible, pero en algunos casos, mientras mayor sea la demora, mejor.

Regresando una vez más al enfoque de análisis de riesgo que propuso la Fuerza Aérea de Estados Unidos [AFC88], se recomiendan los siguientes pasos para determinar las consecuencias globales de un riesgo:

¿Cómo se valoran las consecuencias de un riesgo?

1. Determinar el valor promedio de la probabilidad de que ocurra para cada componente de riesgo.
2. Empleando la figura 25.1, determinar el impacto para cada componente, con base en los criterios mostrados.
3. Completar la tabla de riesgos y analizar los resultados como se describe en las secciones precedentes.

La *exposición al riesgo* global, ER, se determina mediante la siguiente relación [HAL98]:

$$ER = P \times C$$

donde P es la probabilidad de que ocurra un riesgo, y C, el costo al proyecto en caso de que ocurra el riesgo.

Por ejemplo, suponga que el equipo de software define un riesgo de proyecto en la forma siguiente:

Identificación del riesgo. De hecho, sólo 70 por ciento de los componentes de software calendarizados para reutilización se integra en la aplicación. La funcionalidad restante tendrá que desarrollarse de modo personalizado.

Probabilidad de riesgo. 80 por ciento (quizá).

Impacto del riesgo. Se planificaron 60 componentes de software reutilizables. Si sólo se puede emplear el 70 por ciento, 18 componentes tendrían que desarrollarse desde cero (además de otro software personalizado que se ha calendarizado para desarrollo). Puesto que el componente promedio es 100 LDC y los datos locales indican que el costo de ingeniería del software para cada uno es de 14.00 dólares, el costo (impacto) global del desarrollo de los componentes sería $18 \times 100 \times 14 = 25\,200$ dólares.

Exposición al riesgo. ER = 0.80 × 25 200 dólares = 20 200 dólares

La exposición al riesgo se puede calcular para cada riesgo en la tabla de riesgos una vez que se estime el costo del riesgo. La exposición al riesgo total para todos los riesgos (sobre la línea de corte en la tabla) puede ofrecer un medio con que ajustar la estimación del costo final de un proyecto. También se emplea para predecir el aumento probable en los recursos de personal que se requieran en varios puntos durante la calendarización del proyecto.

CONSEJO

Compárese la ER de todos los riesgos con la estimación de costo para el proyecto. Si la ER es mayor que 50 por ciento del costo del proyecto, la viabilidad del proyecto debe reevaluarse.

La proyección del riesgo y las técnicas de análisis descritas en las secciones 25.4.1 y 25.4.2 se aplican de manera iterativa conforme avanza el proyecto de software. El equipo del proyecto debe revisar de nuevo la tabla de riesgos en intervalos regulares, reevaluar cada riesgo para determinar cuándo nuevas circunstancias cambiarán su probabilidad e impacto. Como consecuencia, tal vez sea necesario agregar nuevos riesgos a la tabla, eliminar algunos riesgos que ahora son irrelevantes y cambiar las posiciones relativas de otros

HOGARSEGURO

Análisis de riesgos

El escenario: Oficina de Doug Miller al inicio del proyecto de software *HogarSeguro*.

Los actores: Doug Miller (gerente del equipo de ingeniería del software *HogarSeguro*) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería del software del producto.

La conversación:

Doug: Me gustaría pasar un poco de tiempo en una lluvia de ideas acerca de los riesgos que enfrenta el proyecto *HogarSeguro*.

Jamie: ¿Cómo en qué puede salir mal?

Doug: Sí. Aquí hay algunas categorías de dónde pueden salir mal las cosas. [Muestra a todos las categorías anotadas en la introducción de la sección 25.3.]

Vinod: Mmmmm. ¿quieres que sólo las mencionemos o...?

Doug: No. Esto es lo que creo que debemos hacer. Todos hagan una lista de riesgos... ahora...

(Pasan diez minutos; todos escriben.)

Doug: Muy bien, alto.

Jamie: ¡Pero no he terminado!

Doug: Está bien. Volveremos a ver las listas. Ahora, en cada entrada de su lista, asignen un porcentaje de probabilidad de que el riesgo ocurrirá. Luego, asignen un impacto al proyecto en una escala de 1 (menor) a 5 (catastrófico).

Vinod: Así que si creo que el riesgo es como lanzar una moneda, especifico un 50 por ciento de probabilidad, y si creo que tendrá un impacto de proyecto moderado, especifico un 3, ¿cierto?

Doug: Exactamente.

(Pasan cinco minutos, todos escriben.)

Doug: Muy bien, alto. Ahora haremos una lista del grupo en el pizarrón. Yo escribiré, diré una entrada de su lista en formato de todos contra todos.

(Pasan quince minutos; se crea la lista.)

Jamie (señala al pizarrón y ríe): Vinod, ese riesgo (apunta hacia una entrada en el pizarrón) es ridículo. Existe una enorme probabilidad de que todos seamos golpeados por un rayo. Debemos quitarlo.

Doug: No, dejémoslo por ahora. Consideremos todos los riesgos, sin importar cuán locos sean. Más tarde ventilaremos la lista.

Jamie: Pero ya tenemos casi 40 riesgos... ¿cómo podremos manejar todos?

Doug: No podemos. Por eso definiremos una línea de corte después de ordenar estas linduras. Yo haré eso más tarde y nos reuniremos de nuevo mañana. Por ahora regresen a trabajar... y en su tiempo libre, piensen acerca de cualquier riesgo que hayamos olvidado.

25.5 REFINAMIENTO DEL RIESGO

Durante las primeras etapas de la planificación del proyecto se puede establecer un riesgo de manera muy general. Conforme pasa el tiempo y se aprende más acerca del proyecto y el riesgo, es posible refinar el riesgo en un conjunto de riesgos más detallados, cada uno un poco más sencillo de refinar, supervisar y gestionar.

¿Cuál es una buena forma de describir un riesgo?

Una forma de hacer esto es representar el riesgo en el formato *condición-transición-consecuencia* (CTC) [GLU94]. Es decir, el riesgo se establece en la forma siguiente:

> Dado que <condición> entonces existe una preocupación de que (posiblemente) <consecuencia>

Mediante el empleo del formato CTC en lugar del riesgo de reutilización anotado en la Sección 25.4.2, se puede escribir:

> Dado que todos los componentes de software reutilizables deben ajustarse con estándares de diseño específicos, y como algunos no lo hacen, entonces existe una preocupación de que (posiblemente) sólo 70 por ciento de los módulos reutilizables planeados puedan en realidad integrarse al sistema que se construirá, lo que resulta en la necesidad de ingeniería personalizada para el restante 30 por ciento de componentes.

Esta condición general se puede refinar en la forma siguiente:

> **Subcondición 1.** Ciertos componentes reutilizables fueron desarrollados por terceras personas sin conocimiento de los estándares de diseño internos.
>
> **Subcondición 2.** El estándar de diseño para el componente de interfases no se ha concretado y tal vez no se ajuste con ciertos componentes reutilizables existentes.
>
> **Subcondición 3.** Ciertos componentes reutilizables se han implementado en un lenguaje que no soporta el entorno destino.

Las consecuencias asociadas con estas subcondiciones refinadas siguen siendo las mismas (es decir, 30 por ciento de los componentes de software tienen que someterse a ingeniería personalizada), pero el refinamiento ayuda a aislar los riesgos subyacentes y puede conducir a un análisis y respuestas más sencillos.

25.6 REDUCCIÓN, SUPERVISIÓN Y GESTIÓN DEL RIESGO

Todas las actividades del análisis de riesgo presentadas hasta el momento tienen una sola meta: ayudar al equipo del proyecto a desarrollar una estrategia para luchar con el riesgo. Una estrategia eficaz debe considerar tres temas:

- Evitar del riesgo
- Supervisar el riesgo
- Gestionar el riesgo y los planes de contingencia

Si un equipo de software adopta un enfoque proactivo hacia el riesgo, evitarlo siempre es la mejor estrategia. Ésta se logra desarrollando un plan para reducir el

riesgo. Por ejemplo, supóngase que una elevada movilidad en el personal se anota como un riesgo del proyecto, r_1. Con base en la historia y la intuición administrativa la probabilidad, l_1, de una elevada movilidad se estima en 0.70 (70 por ciento, muy bien alta) y el impacto, x_1, se proyecta como crítico. Esto es: una tasa elevada de movilidad tendrá un impacto crítico en el costo del proyecto y la calendarización

> "Si tomo demasiadas precauciones, es porque no dejo nada al azar."
>
> **Napoleón**

¿Qué se puede hacer para reducir un riesgo?

Este riesgo se reduce si el gestor del proyecto desarrolla una estrategia para reducir la movilidad. Entre los posible pasos que se toman se encuentran:

- Reunirse con el personal actual para determinar las causas de la movilidad (por ejemplo, limitadas condiciones de trabajo, bajos salarios, mercado laboral competitivo).
- Reducir aquellas causas que se controlan antes de que comience el proyecto
- Una vez iniciado el proyecto, suponer que la movilidad ocurrirá y entonces desarrollar técnicas que aseguren la continuidad cuando la gente se aleje
- Organizar equipos de proyecto de modo que la información acerca de cada actividad de desarrollo se disperse con amplitud.
- Definir estándares de documentación y establecer mecanismos que aseguren que los documentos se desarrollen en una forma oportuna.
- Llevar a cabo revisiones por pares de todo el trabajo (de modo que más de una persona esté "en ritmo")
- Asignar un miembro de personal de respaldo por cada tecnología crítica

Conforme avanza el proyecto se inician las actividades de supervisión del riesgo. El gestor del proyecto supervisa los factores que pueden proporcionar un indicio de si el riesgo se está volviendo más o menos probable. En el caso de la elevada tasa de movilidad del personal, se pueden supervisar los siguientes factores:

- Actitud general de los miembros del equipo con base en las presiones del proyecto.
- El grado en el cual el equipo está cuajado
- Las relaciones interpersonales entre los miembros del equipo.
- Potenciales problemas con las compensaciones y los beneficios
- La disponibilidad de empleo dentro y fuera de la compañía.

Además de supervisar estos factores, un gestor de proyecto debe supervisar la efectividad de los pasos de reducción del riesgo. Por ejemplo, un paso de reducción del riesgo anotado líneas arriba pide la definición de estándares de documentación.

y mecanismos para garantizar que los documentos se elaboran en forma oportuna. Éste es un mecanismo para asegurar la continuidad en caso de que un individuo crucial abandone el equipo. El gestor del proyecto debe supervisar los documentos cuidadosamente para asegurarse de que cada uno puede permanecer por sí solo y que cada uno contiene información que sería necesaria si una nueva persona se viera obligada a unirse al equipo de software en algún momento a la mitad del proyecto.

La gestión del riesgo y los planes de contingencia suponen que los esfuerzos de reducción han fracasado y que el riesgo se ha vuelto una realidad. Continuando con el ejemplo, el proyecto ya está bien avanzado y varias personas anuncian que renunciarán. Si se ha seguido la estrategia de reducción, el respaldo está disponible, la información se ha documentado y el conocimiento se ha dispersado entre el equipo. Además, el gestor del proyecto puede reenfocar temporalmente los recursos (y reajustar la calendarización del proyecto) hacia aquellas funciones completamente estructuradas, así permite que los nuevos elementos que deben agregarse al equipo "tomen el ritmo". A los individuos que deciden marcharse se les pide detener todo el trabajo y pasar sus últimas semanas "aprendiendo el modo de transferencia". Esto puede incluir la adquisición de conocimiento en videos, el desarrollo de "documentos comentados" o reuniones con otros miembros del equipo que permanecerán en el proyecto.

CONSEJO

Si la ER para un riesgo específico es menor que el costo de la reducción del riesgo, no se intente reducir el riesgo sino continuar supervisándolo.

Es importante señalar que los pasos de reducción, supervisión y gestión del riesgo (RSGR) generan costos adicionales en el proyecto. Por ejemplo, utilizar el tiempo para "respaldar" cualquier tecnología crítica cuesta dinero. Por lo tanto, parte de la gestión del riesgo consiste en evaluar cuándo los beneficios que generan los pasos RSGR se rezagan frente a los costos asociados con su implementación. En esencia, el planificador del proyecto realiza un clásico análisis costo-beneficio. Si los pasos con que se evita el riesgo de elevada movilidad de personal aumentaran tanto el costo del proyecto como su duración en un estimado de 15 por ciento, pero el factor de costo predominante es el "respaldo", el gestor puede decidir no implementar este paso. Por otra parte, si los pasos con que se evita el riesgo se proyectan para aumentar los costos en 5 por ciento y la duración en sólo 3 por ciento, el gestor probablemente pondrá todo en su lugar.

En un proyecto grande es posible definir 30 o 40 riesgos. Si para cada uno se identifican entre tres y siete pasos de gestión del riesgo, ¡ésta puede convertirse por sí misma en un proyecto! Por esta razón se adapta la regla 80-20 de Pareto al riesgo de software. La experiencia indica que 80 por ciento del riesgo del proyecto global (es decir, 80 por ciento del potencial para falla del proyecto) puede explicarse sólo con 20 por ciento de los riesgos identificados. El trabajo realizado durante los primeros pasos del análisis de riesgo ayudará al planificador a determinar cuáles de los riesgos se encuentran en ese 20 por ciento (por ejemplo, riesgos que conduzcan a la mayor exposición al riesgo). Por esta razón, algunos de los riesgos identificados, evaluados y proyectados pueden no incluirse en el plan RSGR, ya que no se ubican en el crítico 20 por ciento (los riesgos con la mayor prioridad de proyecto).

Referencia Web
Un voluminoso archivo que contiene todos los entrados del Foro ACM acerca de Riesgos al Público se puede encontrar en **catless.ncl.ac.uk/Risks.**

El riesgo no está limitado al proyecto de software. Los riesgos pueden ocurrir después de que el software se ha desarrollado exitosamente y entregado al cliente. Estos riesgos están típicamente asociados con las consecuencias de la falla de software en el campo

El *análisis de seguridad y peligros de software* [LEV95] son actividades de aseguramiento de la calidad del software (capítulo 26) que se enfocan en la identificación y evaluación de los peligros potenciales que pudieran afectar al software negativamente y provocar una falla en todo el sistema. Si los peligros se pueden identificar temprano en el proceso de ingeniería del software, las características de diseño de software se pueden especificar de modo que eliminen o controlen los peligros potenciales.

FIGURA 25.4
Hoja de información del riesgo [WIL97].

Hoja de información del riesgo			
ID de riesgo: P02-4-32	Fecha: 9/5/04	Prob: 80%	Impacto: alto
Descripción: Sólo 70 por ciento de los componentes de software calendarizados para reutilización de hecho se integrarán en la aplicación. La funcionalidad restante tendrá que desarrollarse de manera personalizada.			
Refinamiento/contexto: Subcondición 1: Ciertos componentes de reutilización fueron desarrollados por un tercer participante sin conocimiento de los estándares de diseño internos Subcondición 2: El estándar de diseño para los componentes de interfaces no ha sido solidificado y tal vez no concuerden con ciertos componentes reutilizables existentes Subcondición 3: Ciertos componentes reutilizables se han implementado en un lenguaje que no soporta el entorno destino.			
Reducción/supervisión: 1. Contactar con el tercer participante para determinar la concordancia con los estándares de diseño. 2. Presionar para completar los estándares de interfaz; considerar la estructura del componente cuando se decida acerca del protocolo de la interfaz 3. Verificar para determinar el número de componentes en la categoría 3 de subcondición; verificar para determinar si se puede adquirir el soporte para el lenguaje			
Gestión/plan de contingencia/disparador: La ER se calcula en 20 200 dólares. Asignar esta cantidad dentro del costo de contingencia del proyecto. Desarrollar una calendarización revisada suponiendo que se tendrán que construir 18 componentes adicionales; asignar el personal en concordancia. Disparador: Los pasos de reducción son improductivos al 1/7/04			
Estado actual: 12/5/04: Inician los pasos de reducción.			
Elaboró: D. Gagne		Asignado a: B. Laster	

25.7 EL PLAN RSGR

En el plan del proyecto de software se puede incluir una estrategia de gestión de riesgo o los pasos de gestión del riesgo organizarse por separado en un *Plan de reducción, supervisión y gestión del riesgo*. El plan RSGR documenta todo el trabajo realizado como parte del análisis del riesgo y el gestor del proyecto lo emplea como parte del plan global del proyecto.

Algunos equipos de software no elaboran un documento RSGR formal. En su lugar, cada riesgo se documenta individualmente mediante una *hoja de información del riesgo* (HIR) [WIL97]. En la mayoría de los casos la HIR se mantiene empleando un sistema de base de datos, de modo que la creación y las entradas de información, ordenamiento de prioridades, búsquedas y otros análisis se logran fácilmente. En la figura 25.4 se ilustra el formato de la HIR.

Una vez documentado el plan RSGR y que el proyecto ha comenzado, se inician los pasos de reducción y supervisión del riesgo. Como ya se ha comentado, la reducción del riesgo es una actividad encaminada a evitar el problema. La supervisión del riesgo es una actividad de seguimiento del proyecto con tres objetivos principales: 1) valorar si los riesgos predichos de hecho ocurren; 2) asegurar que los pasos para evitar el riesgo definidos para éste se están aplicando con propiedad; y 3) recopilar información que pueda usarse en futuros análisis de riesgo. En muchos casos, a los problemas que ocurren durante un proyecto es posible darles seguimiento hacia más de un riesgo. Otra labor de la supervisión del riesgo es intentar ubicar el origen (qué riesgos provocaron qué problemas a través del proyecto).

HERRAMIENTAS DE SOFTWARE

Gestión del riesgo

Objetivo: El objetivo de las herramientas de gestión del riego es ayudar al equipo del proyecto a definir los riesgos, valorar su impacto y probabilidad, y seguir los riesgos a través de todo el proyecto de software.

Mecánica: En general, las herramientas de gestión del riesgo auxilian en la identificación de riesgos genéricos al proporcionar una lista de riesgos usuales de proyecto y de negocios, ofrecer listas de verificación u otras técnicas "de entrevista" que auxilien en la identificación de riesgos específicos del proyecto, asignar probabilidad e impacto a cada riesgo, apoyar las estrategias de reducción del riesgo y generar muchos reportes relacionados con el riesgo.

Herramientas representativas[4]

Riskman, desarrollada en Arizona State University (www.eas.asu.edu/~sdm/merrill/riskman.html), es un sistema experto en evaluación de riesgo que identifica riesgos relacionados con el proyecto.

Risk Radar, desarrollada por SPMN (www.spmn.com), auxilia a los gestores de proyectos a identificar y gestionar riesgos.

4 Las herramientas anotadas aquí son una muestra de esta categoría. En la mayoría de los casos los nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

RiskTrak, desarrollada por RST (www.risktrac.com), apoya la identificación, el análisis, el reporte y la gestión de riesgos a través de un proyecto de software.

Risk+, desarrollada por C/S Solutions (www.CS-solutions com), se integra con Microsoft Project para cuantificar costos e incertidumbres de calendorización.

X PRIMER, desarrollada por GrafP Technologies (www.grafp.com), es una herramienta genérica basada en Web que predice qué puede salir mal en un proyecto e identifica el origen de las causas de potenciales fallas y contramedidas efectivas.

25.8 RESUMEN

Siempre que en un proyecto de software esté mucho en juego, el sentido común dicta el análisis de riesgos Sin embargo, muchos gestores de proyecto de software lo hacen informal y superficialmente, si es que lo hacen El tiempo empleado en identificar, analizar y gestionar el riesgo paga por sí mismo dividendos en muchas formas: menos trastornos durante el proyecto, una mayor habilidad para seguir y controlar un proyecto, y la confianza que llega cuando se planifican los problemas antes de que ocurran.

El análisis de riesgos puede absorber una cantidad significativa de esfuerzo de planificación del proyecto La identificación, proyección, evaluación, gestión y supervisión toman tiempo. Pero el esfuerzo bien vale la pena. Para citar a Sun Tzu, el general chino que vivió hace 2 500 años: "Si usted conoce al enemigo y se conoce a sí mismo, no necesita temer el resultado de cien batallas". El enemigo del gestor del proyecto de software es el riesgo.

REFERENCIAS

[AFC88] *Software Risk Abatement*, AFCS/AFLC Pamphlet 800-45, U S. Air Force, 30 de septiembre de 1988.

[BOE89] Boehm, B. W., *Software Risk Management*, IEEE Computer Society Press, 1989.

[CHA89] Charette, R. N., *Software Engineering Risk Analysis and Management*, McGraw-Hill/Intertext, 1989.

[CHA92] Charette, R. N., "Building Bridges over Intelligent Rivers", en *American Programmer*, vol. 5, núm. 7, septiembre de 1992, pp. 2-9.

[DRU75] Drucker, P., *Management*, W H Heinemann, 1975.

[GIL88] Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, 1988.

[GLU94] Gluch, D. P., "A construct for Describing Software Development Risks", CMU/SEI-94-TR-14, Software Engineering Institute, 1994.

[HAL98] Hall, E. M., *Managing Risk. Methods for Software Systems Development*, Addison-Wesley, 1998.

[HIG95] Higuera, R P., "Team Risk Management", en *CrossTalk*, U.S. Dept. of Defense, enero de 1995, pp. 2-4.

[KAR96] Karolak, D. W , *Software Engineering Risk Management*, IEEE Computer Society Press, 1996.

[KEI98] Keil, M. *et al.*, "A Framework for Identifying Software Project Risks", en *CACM*, vol. 41, núm. 1, noviembre de 1998, pp. 76-83

[LEV95] Leveson, N G., *Software System Safety and Computers*, Addison-Wesley, 1995.

[SEI93] "Taxonomy-Based Risk Identification", Software Engineering Institute, CMU/SEI-93-TR-6, 1993.

[THO92] Thomsett, R., "The Indiana Jones School of Risk Management", en *American Programmer*, vol. 5, núm. 7, septiembre de 1992, pp. 10-18.
[WIL97] Williams, R. C., J. A Walker y A. J. Dorofee, "Putting Risk Management into Practice", en *IEEE Software*, mayo de 1997, pp. 75-81

PROBLEMAS Y PUNTOS A CONSIDERAR

25.1. Ofrecer cinco ejemplos de otros campos que ilustren los problemas asociados con una estrategia de riesgo reactiva.

25.2. Describir la diferencia entre "riesgos conocidos" y "riesgos predecibles"

25.3. Agregar tres preguntas o tópicos adicionales a cada una de las listas de verificación de riesgo que se presentan en el sitio Web SEPA.

25.4. A usted se le ha pedido construir software para apoyar un sistema de edición de video de bajo costo. El sistema acepta como entrada video digital, almacena el video en disco y luego permite que el usuario haga una amplia variedad de ediciones al video digitalizado. El resultado puede entonces grabarse en DVD u otro medio audiovisual. Investigue un poco acerca de sistemas de este tipo y luego elabore una lista de riesgos tecnológicos que enfrentaría al comenzar un proyecto con estas características.

25.5. Usted es el gestor de proyecto de una gran compañía de software Se le solicita dirigir un equipo que está desarrollando software de procesamiento de textos de "nueva generación" Cree una tabla de riesgos para el proyecto

25.6. Describir la diferencia entre componentes de riesgo y controladores de riesgo.

25.7. Desarrollar una estrategia de reducción de riesgo y especificar actividades de reducción de riesgo para tres de los riesgos anotados en la figura 25.2.

25.8. Desarrollar una estrategia de supervisión de riesgo y especificar actividades de supervisión de riesgo para tres de los riesgos anotados en la figura 25.2 Asegúrese la identificación de los factores que se supervisarán para determinar si el riesgo se está volviendo más o menos probable.

25.9. Desarrollar una estrategia de gestión del riesgo y especificar actividades de gestión del riesgo para tres de los riesgos anotados en la figura 25 2.

25.10. Inténtese refinar tres de los riesgos anotados en la figura 25.2 y luego créense hojas de información de riesgo para cada uno.

25.11. Represéntense tres de los riesgos anotados en la figura 25.2 empleando un formato CTC

25.12. Vuélvase a calcular la exposición al riesgo examinada en la sección 25.4.2 cuando el costo/LDC es de 16 dólares, y la probabilidad de 60 por ciento.

25.13. ¿Puede pensar en una situación en la que un riesgo de alta probabilidad y alto impacto no sería considerado como parte de su plan RSGR?

25.14. Descríbanse cinco áreas de aplicación de software en las que el análisis de la seguridad y los peligros del software serían una preocupación principal

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La bibliografía de gestión del riesgo de software se ha expandido significativamente durante la década pasada. DeMarco y Lister (*Dancing with Bears*. Dorset House, 2003) han escrito un libro entretenido y lúcido que guía a los gestores y profesionales de software a través de la gestión del riesgo. Moynihan (*Coping with IT/IS Risk Management*. Springer-Verlag, 2002) presenta consejos pragmáticos de gestores de proyecto que lidian continuamente con él. Royer (*Project Risk*

Management, Management Concepts, 2002) y Smith y Merritt (*Proactive Risk Management*, Productivity Press, 2002) sugieren un proceso proactivo para la gestión del riesgo. Karolak (*Software Engineering Risk Management*, Wiley, 2002) ha escrito una guía que introduce un modelo de análisis de riesgo fácil de usar y que contiene valiosas listas de verificación y cuestionarios que se apoyan en un paquete de software.

Schuyler (*Risk and Decision Analysis in Projects*, PMI, 2001) considera el análisis de riesgos desde una perspectiva estadística. Hall (*Managing Risk: Methods for Software Systems Development*, Addison-Wesley, 1998) presenta uno de los tratamientos más exhaustivos de la materia. Myerson (*Risk Management Processing for Software Engineering Models*, Artech House, 1997) considera métricas, seguridad, modelos de proceso y otros tópicos. Grey (*Practical Risk Assessment for Project Management*, Wiley, 1995) escribió un útil manual de la evaluación del riesgo. Su tratamiento abreviado ofrece una buena introducción a la materia.

Capers Jones (*Assessment and Control of Software Risks*, Prentice-Hall, 1994) presenta una exposición detallada de los riesgos de software que incluye datos recopilados a partir de cientos de proyectos de software. Jones define 60 factores de riesgo que pueden afectar el resultado de los proyectos de software. Boehm [BOE89] sugiere excelentes formatos de cuestionario y listas de verificación que pueden resultar invaluables en la identificación de riesgos. Charette [CHA89] presenta un tratamiento detallado de las mecánicas del análisis de riesgo, y utiliza teoría de probabilidad y técnicas estadísticas para analizar riesgos. En un volumen adicional, Charette (*Application Strategies for Risk Analysis*, McGraw-Hill, 1990) analiza el riesgo en el contexto de la ingeniería tanto de sistemas como de software, y sugiere estrategias pragmáticas para la gestión del riesgo. Gilb (*Principles of Software Engineering Management*, Addison-Wesley, 1988) presenta un conjunto de "principios" (en ocasiones graciosos y a veces profundos) que pueden servir como una guía valiosa para la gestión del riesgo.

Ewusi-Mensah (*Software Development Failures: Anatomy of Abandoned Projects*, MIT Press, 2003) y Yourdon (*Death March*, Prentice-Hall, 1997) analizan lo que ocurre cuando los riesgos abruman a un equipo de proyecto de software. Bernstein (*Against the Gods*, Wiley, 1998) presenta una entretenida historia del riesgo que se remonta a tiempos antiguos.

El Software Engineering Institute ha publicado muchos informes detallados y guías acerca del análisis y la gestión del riesgo. El folleto AFSCP 800-45 [AFC88] del Air Force Systems Command describe las técnicas de identificación y reducción de riesgos. Cualquier número del *ACM Software Engineering Notes* tiene una sección titulada "Riesgos para el público" (editor, P. G. Neumann). Si usted quiere las más recientes y mejores historias de horror de software, éste es el lugar al que tiene que ir.

En Internet hay disponible una amplia variedad de fuentes de información acerca de gestión del riesgo de software. Una lista actualizada de referencias en la World Wide Web se puede encontrar en el sitio Web SEPA:

http://www.mhhe.com/presssman.

CAPÍTULO

26 GESTIÓN DE LA CALIDAD

CONCEPTOS CLAVE

El enfoque de ingeniería del software descrito en este libro se dirige hacia una sola meta: producir software de alta calidad. Aunque a muchos lectores les parecerá un reto la pregunta: ¿qué es calidad del software?

Philip Crosby [CRO79], en su libro fundamental acerca de calidad, ofrece una respuesta irónica a esta pregunta:

> El problema de la gestión de la calidad no es lo que la gente ignora acerca de ella. El problema es lo que creen saber.
>
> A este respecto, la calidad tiene mucho en común con el sexo. Todo el mundo lo quiere. (En ciertas condiciones, desde luego.) Todos sienten que lo entienden. (Aun cuando no quieran explicarlo.) Todos piensan que su ejecución sólo es cuestión de seguir las inclinaciones naturales. (Después de todo, la gente se las arregla de alguna forma.) Y, desde luego, la mayoría de las personas piensa que los problemas en estas áreas los provocan otras personas. (Si sólo se tomaran el tiempo para hacer las cosas bien.)

Algunos desarrolladores de software continúan creyendo que la calidad de éste es algo en lo que se debe comenzar a preocupar sólo después de que se haya generado el código. ¡Nada podría estar más alejado de la verdad! La *gestión de la calidad* (con frecuencia llamada *garantía de la calidad del software*) es una actividad protectora o de sombrilla (capítulo 2) que se aplica a lo largo del proceso de software.

La gestión de la calidad abarca 1) un proceso de garantía de la calidad del software (SQA, *por sus siglas en inglés*), 2) tareas específicas de aseguramiento y control de la calidad (que incluyen revisiones técnicas formales y una estrategia de pruebas de varios niveles); 3) prácticas efectivas de ingeniería del software (métodos y herramientas); 4) control de todos los productos de trabajo del soft-

UN VISTAZO RÁPIDO

¿Qué es? No es suficiente hablar por hablar diciendo que la calidad del software es importante. Se tiene que 1) definir explícitamente qué quiere decir cuando dice "calidad del software", 2) crear un conjunto de actividades que ayudarán a asegurar que todo producto de trabajo de ingeniería del software presentará alta calidad, 3) realizar actividades de control y aseguramiento de la calidad en cada proyecto de software, 4) usar métricas para desarrollar estrategias que mejoren el proceso de software y, como consecuencia, la calidad del producto final.

¿Quién lo hace? Todos los involucrados en el proceso de ingeniería del software son responsables de la calidad.

¿Por qué es importante? Es posible hacerlo bien o hacerlo de nuevo otra vez. Si un equipo de software subraya la calidad en todas

las actividades de ingeniería del software, ello reduce la cantidad de reelaboración que se debe realizar. Esto resulta en menores costos y, más importante, mejorará el tiempo de llegada al mercado.

¿Cuáles son los pasos? Antes de se que inicien las actividades de aseguramiento de la calidad del software es importante definir "calidad del software" en diversos grados de abstracción. Una vez que entienda qué es calidad, un equipo de software debe identificar un conjunto de actividades SQA que filtrarán los errores de los productos de trabajo antes de que se aprueben.

¿Cuál es el producto obtenido? Se crea un "Plan de aseguramiento de la calidad del software" para definir la estrategia SQA del equipo del software. Durante el análisis, diseño y generación de código el principal producto de trabajo SQA es un breve informe de la revisión técnica formal. Durante las pruebas se producen los planes de prueba y los procedimientos. También se pueden generar otros productos de trabajo asociados con la mejora del proceso.

¿Cómo puedo estar seguro de que lo he hecho correctamente? ¡Encuentre los errores antes de que se conviertan en defectos! Es decir, trabaje para mejorar su eficiencia en la eliminación de defectos (capítulo 22), con lo que se reduce la cantidad de reelaboración que su equipo de software tiene que realizar.

ware y los cambios que generan (capítulo 27); 5) un procedimiento para garantizar la concordancia con los estándares de desarrollo del software (cuando sea aplicable), y 6) mecanismos de medición e informe.

Este capítulo se centra en los temas de gestión y las actividades específicas del proceso que permiten a una organización de software garantizar que hace las cosas correctas en el momento justo y en la forma correcta.

26.1 Conceptos de calidad[1]

PUNTO CLAVE

El control de la variación es la clave para un producto de alta calidad. En el contexto del software se lucha por controlar la variación en el proceso genérico que se aplica y el énfasis de calidad que permea el trabajo de ingeniería del software.

El *control de la variación* es el corazón del control de calidad. Un fabricante quiere minimizar la variación entre los productos que se producen, aun cuando haga algo relativamente simple como duplicar DVD. Seguramente, esto no puede ser un problema: la duplicación de los DVD es una operación simple de fabricación, y es posible garantizar que siempre se creen duplicados exactos del software.

¿Se puede? Es necesario asegurarse de que las pistas se colocan en los DVD dentro de una tolerancia especificada de modo que la mayoría abrumadora de los controladores de DVD pueda leer el medio. Las máquinas de duplicación de discos pueden, y lo hacen, aceptar y rechazar la tolerancia. Así que incluso un proceso "simple" como la duplicación de DVD puede encontrar problemas debidos a la variación entre muestras.

¿Pero cómo se aplica esto al trabajo de software? ¿Cómo puede una organización de desarrollo de software necesitar controlar la variación? De un proyecto a otro se quiere minimizar la diferencia entre los recursos predichos necesarios para comple-

1 Esta sección, escrita por Michael Stovsky, ha sido adaptada de "Fundamentals of ISO 9000", un libro de trabajo desarrollado para *Essential Software Engineering*, un video curriculum desarrollado por R S. Pressman & Associates, Inc. Reimpreso con permiso.

tar un proyecto y los recursos reales utilizados, que incluyen personal, equipo y tiempo. En general, se quisiera estar seguro de que el programa de pruebas abarca un porcentaje conocido del software, de una liberación a otra. No sólo se quiere minimizar el número de defectos que se liberan, sino que se quiere asegurar que la varianza en el número de *bugs* también se minimiza de una liberación a otra. (Los clientes probablemente se molestarán si la tercera liberación de un producto tiene diez veces más defectos que la liberación previa.) Nos gustaría minimizar las diferencias en rapidez y precisión de las respuestas de la línea de soporte para los problemas de los clientes. La lista puede continuar indefinidamente.

26.1.1 Calidad

El *American Heritage Dictionary* define *calidad* como "una característica o atributo de algo". Como un atributo de un elemento, la calidad se refiere a características mensurables, es decir: cosas que se pueden comparar para conocer estándares, como longitud, color, propiedades eléctricas y maleabilidad. Sin embargo, el software, principalmente una entidad intelectual, es más difícil de caracterizar que los objetos físicos.

No obstante, existen las mediciones de las características de un programa. Dichas propiedades incluyen complejidad ciclomática, cohesión, número de puntos de función, líneas de código y muchas otras examinadas en el capítulo 15. Cuando se examina un elemento con base en sus características mensurables se pueden encontrar dos tipos de calidad: calidad de diseño y calidad de concordancia.

La *calidad de diseño* se refiere a las características que los diseñadores especifican para un elemento. La *calidad de concordancia* es el grado en el que las especificaciones de diseño se aplican durante la fabricación.

> "La gente olvida cuán rápido hiciste un trabajo, pero siempre recuerdan cuán bien lo hiciste."
>
> **Howard Newton**

En el desarrollo de software, la calidad del diseño incluye requisitos, especificaciones y el diseño del sistema. La calidad de concordancia es un tema enfocado principalmente en la implementación. Si ésta sigue el diseño y el sistema resultante satisface sus requisitos y metas de desempeño, la calidad de concordancia es alta.

¿Pero la calidad del diseño y la calidad de concordancia son los únicos temas que deben considerar los ingenieros de software? Robert Glass [GLA98] argumenta que es conveniente una relación más "intuitiva":

satisfacción del usuario = producto manejable + buena calidad
+ entrega dentro de presupuesto y tiempo

En el fondo, Glass afirma que la calidad es importante, pero si el usuario no está satisfecho, nada más importa en realidad. DeMarco [DEM99] refuerza esta visión cuando afirma: "La calidad de un producto es una función de cuánto cambia el mundo para mejorar". Esta visión de la calidad afirma que si un producto de software pro-

porciona beneficio sustancial a sus usuarios finales, éstos estén dispuestos a tolerar problemas ocasionales en confiabilidad y desempeño.

26.1.2 Control de calidad

¿Qué es control de calidad de software?

El control de la variación puede equipararse con el control de calidad. Pero, ¿cómo se logra el control de calidad? Éste involucra la serie de inspecciones, revisiones y pruebas empleadas a lo largo del proceso de software para garantizar que cada producto de trabajo satisfaga los requisitos que se le han asignado. El control de calidad incluye un bucle de retroalimentación con el proceso que creó el producto de trabajo. La combinación de medición y retroalimentación permite afinar el proceso cuando los productos de trabajo creados fracasan en cuanto a satisfacer sus especificaciones.

Un concepto clave del control de calidad es que todos los productos de trabajo tienen especificaciones definidas mensurables con las cuales se puede comparar la salida de cada proceso. El bucle de retroalimentación es esencial para minimizar los defectos producidos

26.1.3 Garantía de la calidad

Referencia Web

Unos vínculos a recursos de SQA se pueden encontrar en www.qualityfree.com/links/index.htm.

La garantía de la calidad consiste en un conjunto de funciones de auditoría e información que evalúan la efectividad y qué tan completas son las actividades de control de calidad. La meta del aseguramiento de la calidad es brindarle al gestor los datos necesarios para que esté informado acerca de la calidad del producto, y por consiguiente que comprenda y confíe en que la calidad del producto está satisfaciendo sus metas. Desde luego, si los datos que ofrece el aseguramiento de la calidad identifican problemas, es responsabilidad del gestor abordarlos y aplicar los recursos necesarios para resolver los conflictos de calidad.

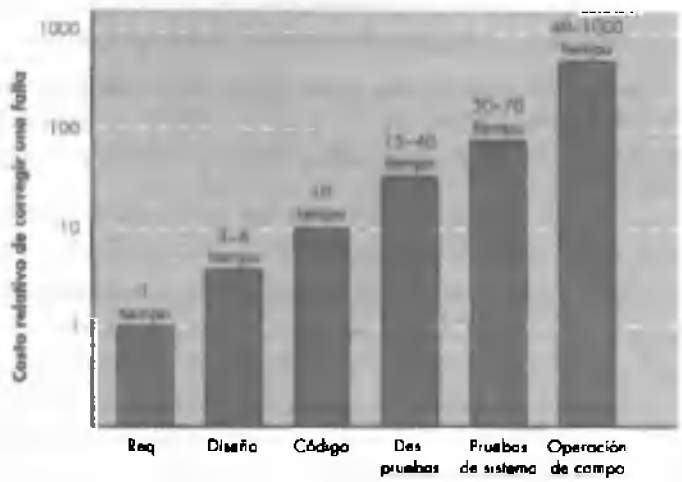
26.1.4 Costo de la calidad

El costo de la calidad incluye todos los costos que genera la búsqueda de calidad o que demanda el desarrollo de las actividades relacionadas con la calidad. Los estudios de costo de la calidad se llevan a cabo para ofrecer una línea base para el costo actual de la calidad, identificar oportunidades que reduzcan el costo de calidad y proporcionar una base normalizada de comparación. La base de la normalización casi siempre es monetaria. Una vez que se han normalizado los costos de la calidad sobre una base monetaria, se tienen los datos necesarios para evaluar dónde se encuentran las oportunidades para mejorar los procesos. Más todavía, se puede evaluar el efecto de los cambios en términos monetarios.

¿Cuáles son los componentes del costo de calidad?

Los costos de calidad se dividen en costos asociados con prevención, evaluación y fallas. Los *costos de prevención* incluyen planificación de la calidad, revisiones técnicas formales, equipo de pruebas y entrenamiento. Los *costos de evaluación* incluyen actividades para comprender mejor la condición del producto la "primera vez a través de" cada proceso. Los ejemplos de costos de evaluación incluyen inspección en el proceso y entre procesos, calibración y mantenimiento de equipo y pruebas

FIGURA 26.1 Costo relativo de corregir una falla.



CONSEJO

incurre en costos tranquilo de inversión

Los *costos de fallas* son aquellos que desaparecerían si no aparecieran defectos antes de enviar un producto a los clientes. Estos costos se subdividen en costos de fallas internas y externas. Se incurre en los *costos de fallas internas* cuando se detecta un defecto en el producto antes del envío. Los costos de fallas internas incluyen reelaboración, reparación y análisis en modo de falla. Los *costos de fallas externas* se asocian con defectos detectados después de que el producto ha sido enviado al cliente. Los ejemplos de costos de fallas externas son la resolución de las quejas, devolución y reemplazo del producto, soporte de ayuda en línea y trabajo de garantía.

Como se esperaba, los costos relativos para encontrar y reparar un defecto aumentan sustancialmente conforme se pasa de la prevención a detección y de los de falla interna a los de falla externa.

La figura 26.1, basada en datos recopilados por Boehm [BOE81] y otros, ilustra este fenómeno.

> "Toma menos tiempo hacer una cosa bien que explicar por qué la hiciste mal."
>
> **H. W. Longfellow**

26.2 GARANTÍA DE LA CALIDAD DEL SOFTWARE (SQA)

Incluso los desarrolladores de software más exhaustos estarán de acuerdo en que el software de alta calidad es una meta importante. Pero, ¿cómo se define calidad? Un bromista dijo una vez: "Todo programa hace algo bien, sólo que puede ser la cosa que no queremos que haga".

En la bibliografía se han propuesto muchas definiciones de la calidad del software. En cuanto a los propósitos del presente texto, la *calidad del software* se define de la siguiente manera:

¿Cómo se define la calidad del software?

> Concordancia con los requisitos funcionales y de desempeño explícitamente establecidos, estándares de desarrollo explícitamente documentados y características implícitas que se esperan de cualquier software desarrollado profesionalmente

No hay duda de que esta definición puede modificarse o extenderse. De hecho, la definición de calidad del software podría debatirse interminablemente. En cuanto a los propósitos de este libro, esta definición sirve para resaltar tres puntos importantes:

1. Los requisitos de software son la base de las medidas de la calidad. La falta de concordancia con los requisitos es una falta de calidad.
2. Los estándares especificados definen un conjunto de criterios de desarrollo que guían la forma en que el software se elabora. Si no se siguen los criterios casi seguramente resultará una falta de calidad.
3. Con frecuencia no se menciona un conjunto de requisitos implícito (por ejemplo, el deseo de uso sencillo y facilidad de mantenimiento). Si el software concuerda con sus requisitos explícitos pero fracasa al satisfacer los requisitos implícitos, su calidad está en duda.

26.2.1 Algunos antecedentes

El control y la garantía de la calidad son actividades esenciales en cualquier negocio que elabore productos de consumo. Antes del siglo xx, el control de calidad era responsabilidad exclusiva del empresario que fabricaba un producto. La primera función formal de garantía y control de calidad la introdujeron los Laboratorios Bell en 1916 y se extendió rápidamente a través del mundo industrial. Durante el decenio de 1940 se sugirieron enfoques más formales del control de calidad, los cuales se apoyaban en la medición y la mejora continua de los procesos [DEM86] como los elementos clave de la gestión de la calidad.

> "Cometiste demasiados malos errores."
>
> **Yogi Berra**

En la actualidad, toda compañía tiene mecanismos que garantizan la calidad en sus productos. De hecho, las afirmaciones explícitas de la preocupación de una compañía por la calidad se han convertido en una táctica de mercadotecnia durante las décadas pasadas.

La historia de la garantía de la calidad en el desarrollo de software avanza paralela a la de la calidad en la fabricación de hardware. Durante los primeros días de la computación (décadas de 1950 y 1960), la calidad era responsabilidad exclusiva del programador. Los estándares de garantía de la calidad para el software se introdujeron en los contratos militares de desarrollo de software durante el decenio de 1970 y se han extendido rápidamente en el desarrollo del software en el mundo de los negocios [IEE94]. Si se extiende la definición presentada anteriormente, la garantía de la calidad del software es un "patrón de acciones sistemático y planificado" [SCH98]

que se requiere para garantizar alta calidad en el software. Numerosos y diversos participantes tienen responsabilidad en la garantía de la calidad del software: ingenieros de software, gestores de proyecto, clientes, vendedores y los individuos que participan en un grupo de SQA.

El grupo de SQA funciona como el representante en casa del cliente. Es decir, las personas que realizan el SQA deben observar el software desde el punto de vista del cliente. ¿El software satisface adecuadamente los factores de calidad señalados en el capítulo 15? ¿El desarrollo de software se ha llevado a cabo de acuerdo con los estándares preestablecidos? ¿Las disciplinas técnicas han realizado adecuadamente sus tareas como parte de la actividad de SQA? El grupo de SQA intenta responder éstas y otras preguntas para garantizar que la calidad del software se conserva.

26.2.2 Actividades de SQA

La garantía de la calidad de software se compone de una variedad de tareas asociadas con dos integrantes diferentes: los ingenieros de software que realizan el trabajo técnico y un grupo de SQA que tiene la responsabilidad de planificar, supervisar, guardar registros, analizar y reportar la garantía de calidad

Los ingenieros de software abordan la calidad (y realizan actividades de aseguramiento y control de calidad) al aplicar sólidos métodos y medidas técnicas, llevar a cabo revisiones técnicas formales y desarrollar pruebas de software bien planificadas. En este capítulo sólo se examinan las revisiones. Los tópicos de tecnología se tratan en las partes 1, 2, 3 y 5 de este libro.

La misión del grupo de SQA es auxiliar al equipo de software a conseguir un producto final de alta calidad. El Software Engineering Institute (SEI) recomienda un conjunto de actividades de SQA que abordan la planificación, supervisión, conservación de registros, análisis y elaboración de informes de aseguramiento de la calidad. Dichas actividades las realiza (o facilita) un grupo de SQA independiente que se encarga de las siguientes actividades:

¿Cuál es el papel de un grupo de SQA?

Preparar un plan de SQA para un proyecto. El plan se desarrolla durante la planificación del proyecto y lo revisan todos los participantes. Las actividades de garantía de la calidad del equipo de ingeniería del software y del grupo de SQA las rige el plan. Éste identifica las evaluaciones que se realizarán, las auditorías y revisiones para llevar a cabo, los estándares aplicables al proyecto, los procedimientos para el informe y seguimiento de errores, los documentos que debe producir el grupo de SQA y la cantidad de retroalimentación proporcionada al equipo de proyecto de software.

Participar en el desarrollo de la descripción del proceso de software del proyecto. El equipo de software selecciona un proceso para el trabajo que habrá de realizarse. El grupo de SQA revisa la descripción del proceso para que concuerde con las políticas organizacionales, los estándares internos de software, los estándares impuestos de manera externa (por ejemplo, ISO-9001) y otras partes del plan de proyecto del software.

Revisar las actividades de ingeniería del software para verificar que se ajuste al proceso de software definido. El grupo de SQA identifica, documenta y sigue las desviaciones del proceso y verifica que se hayan hecho las correcciones.

Audita productos de trabajo de software seleccionados para verificar que se ajusten con los definidos como parte del proceso del software. El grupo de SQA revisa los productos de trabajo seleccionados, identifica, documenta y sigue las desviaciones; verifica que se hayan hecho las correcciones; y periódicamente informa de los resultados de su trabajo al gestor del proyecto.

Garantiza que las desviaciones en el trabajo del software y en los productos de trabajo estén documentadas y se manejen de acuerdo con el procedimiento establecido. Las desviaciones se pueden encontrar en el plan del proyecto, en la descripción del proceso, en los estándares aplicables o en los productos de trabajo técnicos.

Registra cualquier falta de ajuste y lo informa al gestor ejecutivo. A los elementos que no se ajustan se les da seguimiento hasta resolverlos.

Además de estas actividades, el grupo de SQA coordina el control y la gestión del cambio (capítulo 27) y ayuda a recopilar y analizar métricas de software.

26.3 REVISIONES DEL SOFTWARE

Las revisiones son como filtros en el flujo de trabajo del proceso de software. Muy poco y el flujo está "sucio". Demasiado y el flujo se reduce a un chorrito. Use métricas para determinar qué revisiones funcionan y resóltelos.

Las revisiones del software son un "filtro" para el proceso de software. Esto es, las revisiones se aplican en varios puntos durante la ingeniería del software y sirven para descubrir errores y defectos que luego pueden eliminarse. Las revisiones del software "purifican" las actividades de ingeniería del software que se han denominado análisis, diseño y codificación. Freddman y Weinberg [FRE90] abordan del modo siguiente la necesidad de las revisiones:

> El trabajo técnico necesita revisarse por la misma razón que los lápices necesitan gomas. *Errar es humano.* La segunda razón por la que se necesitan las revisiones técnicas es que, aunque la gente sea buena al captar algunos de sus propios errores, las grandes clases de errores escapan de su creador con más facilidad de lo que se le escapan a alguien más.

Como parte de la ingeniería del software se pueden llevar a cabo muchos tipos de revisiones. Cada uno tiene su lugar. Una reunión informal en torno a una máquina expendedora de café es una forma de revisión, si se examinan los problemas técnicos. Una presentación formal del diseño de software a un auditorio de clientes, gestores y personal técnico también es una forma de revisión. Sin embargo, este libro se enfoca sobre la *revisión técnica formal*, a veces llamada *comprobación manual del código (walkthrough)* o *inspección*. Una revisión técnica formal (RTF) es el filtro más efectivo desde un punto de vista de aseguramiento de la calidad. Dirigida por los ingenieros de software (u otras personas) para ingenieros de software, la RTF es un medio efectivo para descubrir errores y mejorar la calidad del software.

INFORMACIÓN

Bugs, errores y defectos

La meta del SQA es eliminar los problemas de calidad en el software. A estos problemas se les conoce con varios nombres. "bugs", "fallas", "errores" o "defectos", por mencionar unos cuantos. ¿Cada uno de éstos son términos sinónimos o existen sutiles diferencias entre ellos?

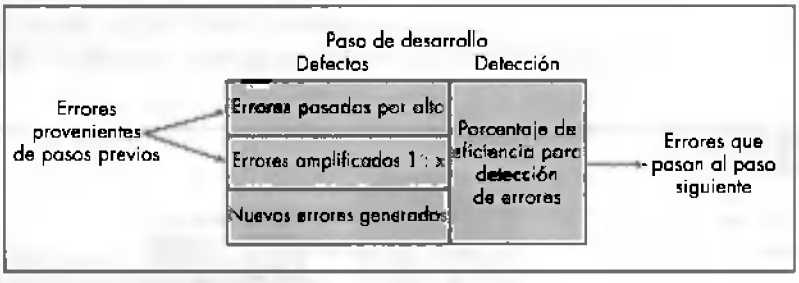
En este libro se ha hecho una clara distinción entre un *error* (un problema de calidad descubierto *antes* de que el software sea liberado entre los usuarios finales) y un *defecto* (un problema de calidad detectado sólo *después* de que el software ha sido liberado entre los usuarios finales).[2] Se ha hecho esta distinción porque los errores y defectos tienen impactos económicos, de negocios, psicológicos y humanos muy diferentes. Como ingenieros de software se quiere descubrir y corregir tantos errores como sea posible antes de que el cliente o usuario final los encuentre. Se quieren evitar los defectos porque (justificadamente) hacen ver mal a la gente de software.

Sin embargo, es importante mencionar que la distinción temporal hecha en este libro entre errores y defectos no es la tendencia predominante. El consenso general entre la comunidad de ingeniería del software es que defectos y errores, fallas y bugs son sinónimos. Es decir, el momento en que el problema se descubrió no tiene importancia en cuanto al término con que se describe el problema. Parte del argumento en favor de esta visión es que a veces es difícil distinguir con claridad entre preliberación y posliberación (por ejemplo, considérese un proceso incremental utilizado en el desarrollo ágil [capítulo 4]).

Sin importar cómo se elija interpretar estos términos, reconózcase que el momento en que se descubre un problema sí importa y que los ingenieros de software deben intentar duro, muy duro, detectar los problemas antes de que los clientes y usuarios finales los encuentren. Si se tiene un interés posterior en este tema, una revisión razonablemente amplia de la terminología que rodea a los "bugs" se puede encontrar en www.softwaredevelopment.ca/bugs.shtml

FIGURA 26.2

Modelo de amplificación de defecto.



26.3.1 Impacto de los defectos de software en el costo

El objetivo principal de las revisiones técnicas formales es descubrir los errores durante el proceso, de modo que no se conviertan en defectos después de liberar el software. El beneficio obvio de las revisiones técnicas formales es el descubrimiento temprano de los errores de modo que ya no se propaguen al paso siguiente en el proceso del software.

2 Si se considera la mejora en el proceso de software, un problema de calidad que se propaga desde una actividad del marco de trabajo del proceso (por ejemplo, modelado) hacia otra (por ejemplo, construcción) también se puede llamar "defecto" porque el problema se debió haber descubierto antes de que un producto de trabajo (por ejemplo, un modelo de diseño) se hubiese "liberado" hacia la actividad siguiente

CONSEJO

El objetivo principal de una RTF es encontrar los errores antes de que pasen a otra actividad de ingeniería del software o sean liberados al usuario final

Varios estudios industriales (realizados por TRW, NEC y Mitre Corp., entre otros) indican que las actividades de diseño introducen entre 50 y 65 por ciento de los errores (y, a final de cuentas, de los defectos) durante el proceso de software. Sin embargo, las técnicas de revisión formal han mostrado hasta 75 por ciento de efectividad [JON86] al descubrir fallos en el diseño. Al detectar y eliminar un gran porcentaje de dichos errores, el proceso de revisión reduce sustancialmente el costo de las actividades subsecuentes en el proceso de software.

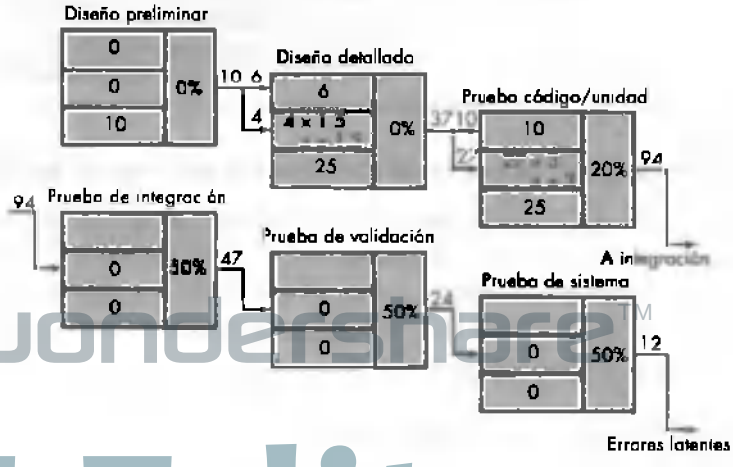
Para ilustrar el impacto en el costo de la detección temprana de errores, considérese una serie de costos relativos que se basan en datos de costo real recopilados para grandes proyectos de software [IBM81].[3] Supóngase que la corrección de un error descubierto durante el diseño costará 1.0 unidad monetaria. En relación con este costo, el mismo error descubierto justo antes de que comience el periodo de pruebas costará 6.5 unidades; durante las pruebas, 15 unidades; y después de la liberación, entre 60 y 100 unidades.

26.3.2 Amplificación y eliminación del defecto

Se puede usar un modelo de amplificación de defectos [IBM81] para ilustrar la generación y detección de errores durante el diseño preliminar, el diseño de detalles y los pasos de codificación de un proceso de ingeniería del software. En la figura 26.2 se ilustra esquemáticamente el modelo. Un recuadro representa un paso de desarrollo del software. Durante el paso, los errores se pueden generar de manera inadvertida. La revisión puede fallar en descubrir errores generados de manera reciente y errores de pasos anteriores, lo que resulta en cierto número de errores que se pasan por alto. En algunos casos, los errores que se pasan por alto desde pasos anteriores se am-

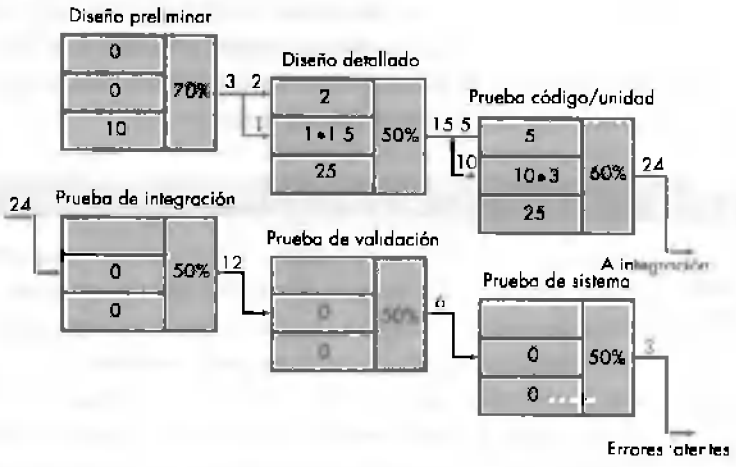
FIGURA 26.3

Amplificación de defecto sin revisiones.



3 Aunque estos datos tienen más de 20 años de antigüedad, aún son aplicables en un contexto moderno.

FIGURA 26.4
Amplificación de defecto con revisiones.



plifican (factor de amplificación *x*) con el trabajo actual. Las subdivisiones de los recuadros representan cada una de estas características y el porcentaje de eficiencia de la detección de errores, una función de la minuciosidad de la revisión.

> "Algunas enfermedades, dicen los médicos, son fáciles de curar en sus inicios aunque difíciles de reconocer ... pero en el transcurso del tiempo, cuando no han sido reconocidas a primera vista y tratadas, se vuelven fáciles de reconocer pero difíciles de curar."
>
> Nicolás Maquiavelo

La figura 26.3 ilustra un ejemplo hipotético de la amplificación del defecto para un proceso de software en el que no se llevan a cabo revisiones. En la figura se supone que cada paso de prueba descubre y corrige, sin introducir nuevos errores, 50 por ciento de los errores que llegan (una suposición optimista). Diez defectos de diseño preliminar se amplificaron a 94 errores antes de comenzar las pruebas. Doce defectos latentes se liberaron al campo. La figura 26.4 considera las mismas condiciones excepto que la revisión del diseño y del código se llevaron a cabo como parte de cada paso de desarrollo. En este caso, diez errores iniciales de diseño preliminar se amplificaron a 24 errores antes de comenzar el periodo de pruebas. Sólo existen tres defectos latentes. Al considerar los costos relativos asociados con el descubrimiento y la corrección de errores se puede establecer el costo global (con revisión y sin ella para el ejemplo hipotético). El número de errores descubiertos durante cada uno de los pasos anotados en las figuras 26.3 y 26.4 se multiplica por el costo para eliminar un error (1.5 unidades de costo para diseño, 6.5 unidades de costo antes de las pruebas, 15 unidades de costo durante las pruebas, y 67 unidades de costo después de la liberación). Empleando estos datos, el costo total para desarrollo y mantenimiento es de 783 unidades cuando se realizan revisiones. Si no se realizan revisiones el costo total es de 2 177 unidades, casi tres veces más costoso.

En las revisiones un ingeniero de software debe utilizar tiempo y esfuerzo, y la organización desarrolladora, dinero. Sin embargo, los resultados del ejemplo precedente no dejan duda acerca de pagar ahora o hacerlo más tarde. Las revisiones técnicas formales (para el diseño y otras actividades técnicas) ofrecen un beneficio demostrable en el costo. Se deben llevar a cabo.

26.4 REVISIONES TÉCNICAS FORMALES

¿Cuando se llevan a cabo RTF, ¿cuáles son los objetivos?

Una revisión técnica formal (RTF) es una actividad de control de calidad del software que llevan a cabo los ingenieros de software (y otros). Los objetivos de una RTF son 1) descubrir errores en la función, lógica o implementación en cualquier representación del software; 2) verificar que el software en revisión satisface sus requisitos, 3) garantizar que el software se ha representado de acuerdo con los estándares predefinidos; 4) lograr software desarrollado en una manera uniforme; y 5) hacer proyectos más manejables. Además, la RTF sirve como un campo de entrenamiento, pues permite que los ingenieros menos experimentados observen diferentes enfoques respecto del análisis, el diseño y la construcción del software. La RTF también funge como promotora del soporte y la continuidad, pues varias personas se familiarizan con las partes del software que de otra forma nunca verían.

> "No hay urgencia más grande que la que tiene un hombre por corregir el trabajo de otro."
>
> **Mark Twain**

La RTF es en realidad una clase de revisión que incluye recorridos, inspecciones, revisiones cíclicas y otro pequeño grupo de evaluaciones técnicas de software. Cada RTF se realiza en una junta y tendrá éxito sólo si se planifica, controla y atiende apropiadamente. En las siguientes secciones se presentan directrices similares a las de un recorrido (por ejemplo, [FRE90], [GIL93]) que se presenta como una revisión técnica formal representativa.

26.4.1 La junta de revisión

Sin importar el formato de RTF que se elija, cualquier junta de revisión debe atenerse a las siguientes restricciones:

- En la revisión se deben involucrar (usualmente) entre tres y cinco personas.
- Se debe preparar con anticipación, pero sin que requiera más de dos horas de trabajo de cada persona.
- La duración de la junta de revisión debe ser menor a dos horas.

Dadas estas restricciones, debe ser obvio que una RTF se enfoca en una parte específica (y pequeña) del software total. Por ejemplo, más que intentar revisar un diseño completo, se llevan a cabo recorridos para cada componente o grupo pequeño de componentes. Al estrechar el enfoque, la RTF tiene una mayor probabilidad de descubrir errores

Referencia Web

El *Inspection Guidebook* (Libro guía de inspección formal) de NASA SATC se puede descargar de satc.gsfc.nasa.gov/fi/fipage.

PUNTO CLAVE

Una RTF se enfoca en una porción relativamente pequeña de un producto de trabajo.

El enfoque de la RTF se dirige a un producto de trabajo (por ejemplo, una porción de una especificación de requisitos, un diseño detallado de componente, una lista de código fuente de un componente). El individuo que ha desarrollado el producto de trabajo —el *productor*— le informa al jefe del proyecto que el producto está completo y que se requiere una revisión. El jefe del proyecto se pone en contacto con un *jefe de revisión*, quien evalúa la disponibilidad del producto, genera copias del material del producto y las distribuye a dos o tres *revisores* para que realicen sus observaciones antes de la junta. Se espera que cada revisor emplee entre una y dos horas en revisar el producto, tomar notas y familiarizarse con el trabajo. Al mismo tiempo, el jefe de revisión también revisa el producto y establece una agenda para la junta de revisión, la que usualmente se programa para el día siguiente.

CONSEJO

En algunas situaciones es buena idea hacer que alguien distinto al productor recorra el producto que experimenta revisión. Esto conduce a una interpretación literal del producto de trabajo y a un mejor reconocimiento de los errores.

A la junta de revisión asisten el jefe de revisión, todos los revisores y el productor. Uno de los revisores asume el papel de *registrador*; es decir, el individuo que registra (por escrito) los temas importantes que surjan durante la revisión. La RTF comienza con una presentación de la agenda y una breve introducción a cargo del productor. Entonces el productor procede a recorrer el producto de trabajo y explica el material, mientras que los revisores exponen los problemas que descubrieron antes de la junta. Cuando se descubren problemas o errores válidos el registrador anota cada uno.

Al final, todos los asistentes a la RTF deben decidir si 1) aceptan el producto sin modificaciones posteriores, 2) rechazan el producto debido a errores severos (una vez corregidos se tiene que realizar otra revisión) o 3) aceptan el producto provisionalmente (se encontraron errores menores que es necesario corregir, pero no se requerirá revisión adicional). Cuando se tome la decisión, todos los asistentes a la RTF llenan un documento de finalización en el que indican su participación en la revisión y su conformidad con los hallazgos del equipo revisor.

26.4.2 Informe de la revisión y conservación de registros

Durante la RTF, un revisor (el registrador) registra activamente todos los problemas que hayan surgido. Éstos se resumen al final de la junta de revisión y se genera una *lista de problemas de revisión*. Además, se llena un *informe resumen de la revisión técnica formal*. Un informe resumen de la revisión responde tres preguntas:

1. ¿Qué se revisó?
2. ¿Quién lo revisó?
3. ¿Cuáles fueron los hallazgos y conclusiones?

El informe resumen de la revisión es un formato de una sola página (con posibles anexos). Se vuelve parte del registro histórico del proyecto y es posible distribuirlo entre el jefe del proyecto y otras partes interesadas.

La lista de problemas de la revisión cumple dos propósitos: 1) identificar áreas problema en el producto y 2) funcionar como lista de verificación de elementos de acción que guían al productor conforme se hacen las correcciones. Normalmente al informe resumen se anexa una lista de problemas.

Es importante establecer un procedimiento de seguimiento para garantizar que los elementos en la lista de problemas se han corregido adecuadamente. A menos que esto se haga, es posible que los problemas surgidos "caigan entre las grietas". Un enfoque consiste en asignar la responsabilidad del seguimiento al jefe de revisión.

> "A menudo una reunión es un suceso en el que se toman los minutos y se pierden las horas."
>
> **Anónimo**

26.4.3 Directrices de la revisión

Las directrices para realizar las revisiones técnicas formales es necesario establecerlas con anticipación, distribuirlas entre todos los revisores, suscribirlas y luego seguirlas. Una revisión descontrolada usualmente es peor que carecer de una. Las siguientes representan un conjunto mínimo de directrices para las revisiones técnicas formales.

No señale los errores de manera hiriente. Una forma de ser gentil es preguntar algo que permita al productor descubrir el error

1. *Revisar el producto, no al productor.* Una RTF involucra personas y egos. Realizada con propiedad, la RTF debe dejar a todos los participantes con un cálido sentimiento de logro. Si se lleva a cabo de manera inadecuada, la RTF puede tomar un aura inquisitorial. Los errores se deben señalar con gentileza; el tono de la junta debe ser relajado y constructivo; la finalidad no debe ser avergonzar o menospreciar.
2. *Establecer una agenda y respetarla.* Un mal clave de las juntas de cualquier tipo es la divagación. Una RTF tiene que mantener el rumbo y seguir el programa. El jefe de revisión tiene la responsabilidad de mantener el programa de la junta y no vacilar en llamar la atención de la gente cuando se empiece a divagar.
3. *Limitar el debate y la impugnación.* Cuando un revisor plantee un problema, tal vez no haya un acuerdo universal sobre su impacto. En lugar de perder tiempo debatiendo la cuestión, el problema se debe registrar para tratarlo informalmente después.
4. *Enunciar áreas de problemas, pero no se intente resolver todos los que se hayan señalado.* Una revisión no es una sesión para resolver problemas. Esto se debe posponer hasta después de la junta de revisión.
5. *Tomar notas.* En ocasiones es buena idea que el registrador tome notas en una pizarra, de modo que las palabras y las prioridades puedan evaluarlas otros revisores conforme se registra la información.
6. *Limitar el número de participantes e insistir en la preparación anticipada.* Dos cabezas piensan mejor que una, pero 14 no necesariamente son mejores que cuatro. Manténgase el número de personas involucradas en el mínimo necesario. Sin embargo, todos los miembros del equipo revisor deben prepararse por anticipado. El jefe de revisión debe solicitar comentarios escritos (lo que ofrece un indicio de que el revisor ha analizado el material).

7. *Desarrollar una lista de verificación para cada producto que tenga probabilidad de ser revisado.* Una lista de verificación ayuda al jefe de revisión a estructurar la junta de RTF, y a cada revisor a enfocarse en los problemas importantes.
8. *Asignar recursos y programar las RTF.* Las revisiones serán efectivas si se programan como una tarea del proceso de software. Además, se debe programar tiempo para realizar las inevitables modificaciones que ocurrirán como resultado de una RTF.
9. *Realizar un entrenamiento significativo de todos los revisores.* Los participantes en la revisión serán eficientes si reciben algún entrenamiento formal. El entrenamiento debe subrayar tanto los problemas relacionados con el proceso como el lado psicológico y humano de las revisiones.
10. *Analizar las revisiones previas.* La junta es beneficiosa para descubrir problemas en el proceso de revisión mismo. El primer producto que se haya revisado debe establecer las directrices de revisión.

> "Una de las compensaciones más hermosas de la vida es que ningún hombre puede intentar sinceramente ayudar a otro sin ayudarse a sí mismo."
>
> **Ralph Waldo Emerson**

Puesto que muchas variables (por ejemplo, número de participantes, tipo de productos de trabajo, tiempo y duración, enfoque específico de revisión) inciden en una revisión provechosa, una organización de software debe experimentar para determinar qué enfoque funciona mejor en un contexto local. Porter y sus colegas [POR95] ofrecen una excelente guía para este tipo de experimentación.

26.4.4 Revisiones basadas en muestras

En un contexto ideal, cada producto de trabajo de ingeniería del software debería experimentar una revisión técnica formal. En el mundo real de los proyectos de software, los recursos son limitados y el tiempo es corto. Como consecuencia, usualmente las revisiones se soslayan, aunque se reconozca su valor como mecanismo de control de calidad. Thelin y sus colegas [THE01] abordan este tema cuando afirman:

> Las inspecciones [RTF] sólo son vistas como eficientes si se encuentran muchas fallas durante su búsqueda. Si en los artefactos [productos de trabajo] se encuentran muchas fallas, las inspecciones son necesarias. Si, por otra parte, sólo se encuentran pocas fallas, la inspección ha sido una pérdida de tiempo para muchas personas involucradas en la tarea.[4] Más aún, los proyectos de software que están atrasados con frecuencia disminuyen el tiempo de las actividades de inspección, lo que conduce a una falta de calidad. Una so-

4 Desde luego, se puede argumentar que, al llevar a cabo revisiones, se alienta a los productores a enfocarse en la calidad, incluso si no se encuentran errores.

lución sería asignar jerarquías a los recursos para las actividades de inspección y, en consecuencia, concentrar los recursos disponibles en los artefactos más proclives a las fallas.

CONSEJO

Las revisiones toman tiempo, pero es tiempo bien empleado. Sin embargo, si el tiempo es corto y no se tiene otra opción, no se dispensen las revisiones. En su lugar, utilícense revisiones basadas en muestras.

Thelin y sus colegas sugieren un proceso de revisión basado en que muestras de todos los productos de trabajo de ingeniería del software, éstas se inspeccionan para determinar qué productos de trabajo son más proclives al error. Entonces los recursos de las RTF completas se enfocan sólo en aquellos productos de trabajo con probabilidad (basándose en los datos recopilados durante el muestreo) de ser proclives al error.

Para ser eficaz, el proceso de revisión basado en muestras debe intentar cuantificar aquellos productos de trabajo que sean objetivos principales para las RTF completas. Para lograrlo se sugieren los siguientes pasos [THE01]:

1. Inspeccionar una fracción a_i de cada producto de trabajo de software i. Registre el número de fallas f_i encontradas dentro de a_i.
2. Desarrollar una estimación bruta del número de fallas dentro del producto de trabajo i al multiplicar f_i por $1/a_i$.
3. Ordenar los productos de trabajo en forma descendente de acuerdo con la estimación bruta del número de fallas en cada uno.
4. Enfocar los recursos de revisión disponibles en aquellos productos de trabajo con el mayor número estimado de fallas.

La fracción con la que se ha hecho un muestreo del producto de trabajo debe 1) ser representativa del producto de trabajo como un todo, y 2) ser lo suficientemente grande de tal manera que sea significativa para los revisores que realicen el muestreo. Conforme a_i aumenta, la probabilidad de que la muestra sea una representación válida del producto de trabajo también crece. Sin embargo, también aumentan los recursos requeridos para levantar la muestra. Un equipo de ingeniería del software debe establecer el mejor valor para a_i según los tipos de productos de trabajo producidos.[5]

HOGARSEGURO

Problemas en el SQA

El escenario: Oficina de Doug Miller cuando comienza el proyecto de software *HogarSeguro*.

Los actores: Doug Miller (gerente del equipo de ingeniería del software *HogarSeguro*) y otros miembros del equipo de ingeniería del software.

La conversación:

Doug: Ya sé que no hemos empleado tiempo para desarrollar un plan de SQA para este proyecto, pero ya estamos en él y tenemos que considerar la calidad... ¿cierto?

5 Thelin y sus colegas han realizado una simulación detallada que puede ayudar a tomar esta determinación. Véase [THE01] para detalles.

Jamie: Claro. Ya hemos decidido que, conforme desarrollemos el modelo de requisitos [capítulos 7 y 8], Ed se ha comprometido a desarrollar un procedimiento V&V para cada requisito.

Doug: Eso es muy bueno, pero no vamos a esperar hasta hacer las pruebas para evaluar la calidad, ¿o sí?

Vinod: ¡No! Desde luego que no. Hemos programado revisiones en el plan del proyecto para este incremento de software. Comenzaremos el control de calidad con las revisiones.

Jamie: Estoy un poco preocupada de que no tengamos tiempo suficiente para realizar todas las revisiones. De hecho, sé que no podremos.

Doug: Mmmmm. ¿Qué propones?

Jamie: Sugiero que seleccionemos aquellos elementos de los modelos de análisis y diseño cruciales para *HogarSeguro* y que los revisemos.

Vinod: ¿Pero qué ocurre si perdemos algo en una parte del modelo que no revisemos?

Shakira: Leí algo acerca de una técnica de muestreo [sección 26.4.4] que puede ayudarnos a seleccionar los candidatos para revisión. [Shakira explica el enfoque.]

Jamie: Tal vez... pero no estoy segura de que incluso tengamos tiempo para tomar muestras de cada elemento de los modelos.

Vinod: ¿Qué quieres que hagamos, Doug?

Doug: Robemos algo de Programación Extrema [capítulo 4]. Desarrollaremos los elementos de cada modelo en pares —dos personas— y realizaremos una revisión informal de cada uno conforme avancemos. Luego seleccionaremos los elementos "cruciales" para una revisión en equipo más formal, pero conservaremos dichas revisiones en un mínimo. De esa forma, todo será observado por más de un conjunto de ojos, pero aún así conservaremos nuestras fechas de entrega.

Jamie: Eso significa que tendremos que revisar la calendarización.

Doug: Así debe ser. La calidad triunfa sobre la calendarización en este proyecto.

26.5 ENFOQUES FORMALES ACERCA DEL SQA

Durante las dos décadas pasadas, un pequeño, pero ruidoso, segmento de la comunidad de ingeniería del software ha argumentado que se requiere un enfoque más formal de la garantía de la calidad del software. Se puede argumentar que un programa de computadora es un objeto matemático [SOM01]. En cada lenguaje de programación se definen una sintaxis y una semántica rigurosas, y existe un enfoque riguroso respecto de la especificación de requisitos de software (capítulo 28). Si el modelo de requisitos (especificación) y el lenguaje de programación se representan en una forma rigurosa, debe ser posible aplicar pruebas matemáticas de exactitud para demostrar que un programa concuerda exactamente con sus especificaciones.

Los intentos encaminados a probar la exactitud de los programas (capítulos 28 y 29) no son nuevos. Dijkstra [DIJ76] y Linger, Mills y Witt [LIN79], entre otros, aconsejaron pruebas de exactitud de programa y los vincularon con la aplicación de conceptos de programación estructurada (capítulo 11).

26.6 GARANTÍA DE LA CALIDAD ESTADÍSTICA DEL SOFTWARE

La garantía de la calidad estadística refleja una tendencia, creciente en la industria, por adoptar un enfoque más cuantitativo acerca de la calidad. Para el software, la garantía de la calidad estadística implica los pasos siguientes:

¿Qué pasos se requieren para realizar SQA estadística?

1. La información acerca de los defectos de software se recopila y clasifica.
2. Se intenta determinar la causa subyacente de cada defecto (por ejemplo, falta de concordancia con las especificaciones, errores de diseño, violación de estándares, deficiente comunicación con el cliente).
3. Mediante el principio de Pareto (80 por ciento de los defectos se encuentra en 20 por ciento de todas las causas posibles) se aísla un 20 por ciento (los "vitales").
4. Una vez que las causas vitales han sido identificadas, se corrigen los problemas que han provocado los defectos.

Este concepto relativamente simple representa un paso importante hacia la creación de un proceso de software adaptable en el que los cambios se hagan para mejorar aquellos elementos del proceso que introducen errores.

> "20 por ciento del código tiene 80 por ciento de los errores. ¡Encuéntrelos, corríjalos!"
>
> **Lowell Arthur**

26.6.1 Un ejemplo genérico

Para ilustrar la aplicación de los métodos estadísticos en el trabajo de ingeniería del software, supóngase que una organización de ingeniería del software recopila información acerca de defectos durante un año. Algunos de los defectos se descubren cuando el software está en desarrollo; otros, después de que se ha liberado entre sus usuarios finales. Aunque se descubren cientos de diferentes defectos, todos tienen una (o más) de las causas siguientes:

- Especificaciones incompletas o erróneas (EIE).
- Mala interpretación de la comunicación del cliente (MCC).
- Desviación intencional de las especificaciones (DIE).
- Violación de los estándares de programación (VEP).
- Errores en la representación de los datos (ERD).
- Interfaz de componente inconsistente (ICI).
- Error en la lógica del diseño (ELD).
- Prueba incompleta o errónea (PIE).
- Documentación imprecisa o incompleta (DII).
- Error en la traducción del diseño al lenguaje de programación (TLP).
- Interfaz hombre-computadora ambigua o inconsistente (IHC).
- Misceláneo (MIS).

Para aplicar el aseguramiento de la calidad estadística del software se construye la tabla de la figura 26.5. La tabla indica que EIE, MCC y ERD son las causas vitales que

FIGURA 26.5 Recolección de datos para SQA estadístico.

	Total		Serios		Moderados		Menores	
Error	Núm	%	Núm	%	Núm	%	Núm	%
EIE	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
DIE	48	5%	1	1%	24	6%	23	5%
VEP	25	3%	0	0%	15	4%	10	2%
ERD	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
ELD	45	5%	14	11%	12	3%	19	4%
PIE	95	10%	12	9%	35	9%	48	11%
DI	36	4%	2	2%	20	5%	14	3%
TLP	60	6%	15	12%	19	5%	26	6%
IHC	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totales	942	100%	128	100%	379	100%	435	100%

explican el 53 por ciento de todos los errores. Sin embargo, se debe observar que EIE, ERD, TLP y ELD se seleccionarían como las causas vitales si sólo se consideraran los errores serios. Una vez determinadas las causas vitales, la organización de ingeniería del software puede comenzar la acción correctiva. Por ejemplo, para corregir MCC, el desarrollador de software puede implementar técnicas que faciliten la recopilación de requisitos (capítulo 7) para mejorar la calidad de la comunicación y las especificaciones del cliente. Para mejorar ERD, el desarrollador puede adquirir herramientas para el modelado de datos y ejecutar revisiones de diseño de datos más rigurosas.

Es importante anotar que la acción correctiva se enfoca principalmente en las vitales. Conforme éstas se corrigen, nuevas candidatas ocupan la parte superior de la clasificación.

Las técnicas de garantía estadística de la calidad para software han demostrado que ofrecen una mejora sustancial en la calidad [ART97]. En algunos casos, las organizaciones de software han alcanzado 50 por ciento de reducción anual en los defectos después de aplicar estas técnicas.

La aplicación del SQA estadístico y el principio de Pareto se pueden resumir en una sola oración: *Emplee su tiempo enfocándose en las cosas que realmente importan, ¡pero primero asegúrese de entender qué es lo que realmente importa!*

Un detallado análisis del SQA estadístico está más allá del ámbito de este libro. Los lectores interesados deben consultar [GOH02], [SCH98] o [KAN95].

26.6.2 Seis sigma para ingeniería del software

Seis sigma es la estrategia más ampliamente empleada en la actualidad para el aseguramiento de la calidad estadístico en la industria. Originalmente popularizada por

Motorola en el decenio de 1980, la estrategia seis sigma "es una metodología rigurosa y disciplinada que utiliza análisis de datos y estadístico para medir y mejorar el desempeño operativo de una compañía al identificar y eliminar los 'defectos' en la fabricación y los procesos relacionados con el servicio" [ISI03]. El término "seis sigma" se deriva de seis desviaciones estándar —3.4 instancias (defectos) por millón de ocurrencias—, lo que implica un estándar de calidad extremadamente elevado. La metodología seis sigma define tres pasos centrales:

¿Cuáles son los pasos centrales de la metodología seis sigma?

- *Definir* los requisitos del cliente, entregables y metas del proyecto por medio de métodos bien definidos de comunicación con el cliente.
- *Medir* el proceso existente y su salida para determinar el desempeño de calidad actual (recopilación de métricas de defecto).
- *Analizar* las métricas de defecto y determinar las causas poco vitales.

Si un proceso de software existente está en marcha, pero se requiere mejoría, seis sigma sugiere dos pasos adicionales:

- *Mejorar* el proceso eliminando las causas originales de los defectos.
- *Controlar* el proceso para garantizar que el trabajo futuro no vuelva a introducir las causas de defectos.

Estos pasos centrales y adicionales a veces se conocen como el método DMAMC (definir, medir, analizar, mejorar y controlar).

Si una organización está desarrollando un proceso de software (en lugar de mejorar un proceso existente), los pasos centrales se aumentan de la siguiente manera:

- *Diseñar* el proceso para 1) evitar las causas originales de los defectos y 2) satisfacer los requisitos del cliente.
- *Verificar* que el modelo de proceso, de hecho, evitará los defectos y satisfará los requisitos del cliente

A esta variación a veces se le llama método DMADV (definir, medir, analizar, diseñar y verificar).

Una exposición detallada de seis sigma se encuentra en las fuentes bibliográficas dedicadas a la materia. El lector interesado debe consultar [ISI03], [SNE03] y [PAN00].

26.7 FIABILIDAD DEL SOFTWARE

La fiabilidad del software, a diferencia de otros factores de calidad, se puede medir, dirigir y estimar empleando datos históricos y de desarrollo. La *fiabilidad del software* se define en términos estadísticos como "la probabilidad de la operación libre de fallas de un programa de computadora en un entorno específico durante un tiempo específico" [MUS87]. Con fines ilustrativos, se estima que el programa *X* tiene una

fiabilidad de 0.96 durante un periodo de ocho horas de procesamiento. En otras palabras, si el programa *X* fuese ejecutado 100 veces y requiriese un total de ocho horas de tiempo de procesamiento (tiempo de ejecución), es probable que operaría correctamente (sin falla) 96 veces.

> "El precio inevitable de la fiabilidad es la simplicidad."
>
> C.A.R. Hoare

Siempre que se estudia la fiabilidad del software, surge una pregunta central: ¿qué significa el término *falla*? En el contexto de cualquier análisis de calidad y fiabilidad del software, la falla es la falta de concordancia con los requisitos del software. Sin embargo, incluso dentro de esta definición, existen gradientes. Las fallas sólo pueden ser molestas y catastróficas. Una falla puede corregirse en segundos, mientras que otra tal vez requiera semanas o incluso meses. Para complicar el tema aún más, la corrección de una falla puede, de hecho, resultar en la introducción de otros errores que a final de cuentas resultan en otras fallas

26.7.1 Medidas de fiabilidad y disponibilidad

Los primeros trabajos en la fiabilidad del software intentaron extrapolar las matemáticas de la teoría de fiabilidad del hardware (por ejemplo, [ALV64]) a la predicción de la fiabilidad del software. La mayoría de los modelos de fiabilidad relacionada con el hardware tratan acerca de las fallas debidas al uso más que a las que se deben a defectos de diseño. En el hardware, las fallas que se deben al uso físico (por ejemplo, los efectos de la temperatura, la corrosión, los choques eléctricos) son más probables que una falla relacionada con el diseño. Desdichadamente, lo opuesto es cierto para el software. De hecho, todas las fallas de software se originan en problemas de diseño o implementación, el uso (capítulo 1) no entra en el cuadro.

Ha habido debates acerca de la relación entre conceptos clave en la fiabilidad del hardware y su aplicabilidad al software (por ejemplo, [LIT89], [ROO90]). Aunque todavía se tiene que establecer un vínculo irrefutable, vale la pena considerar unos cuantos conceptos simples que se aplican a elementos de ambos sistemas.

Si se considera un sistema basado en computadora, una simple medida de fiabilidad es el *tiempo medio entre fallas* (TMEF), donde

$$\text{TMEF} = \text{TMDF} + \text{TMDR}$$

Las siglas TMDF y TMDR significan *tiempo medio de falla* y *tiempo medio de reparación*, respectivamente.[6]

Muchos investigadores argumentan que el TMEF es con mucho más fácil de medir que los defectos/KLDC o los defectos/PF. Establecido de manera simple, el usuario final está preocupado por las fallas, no por el conteo total de errores. Debido a

6 Aunque se pueda requerir la depuración (y correcciones relacionadas) como consecuencia de una falla, en muchos casos el software trabajará adecuadamente después de un reinicio sin otro cambio

Algunos aspectos de la disponibilidad (no estudiados aquí) no tienen nada que ver con las fallas. Por ejemplo, los recortes en la calendarización (para funciones de soporte) provocan que el software no esté disponible

que cada defecto contenido dentro de un programa no tiene la misma tasa de falla, la cuenta de defectos totales ofrece poca información de la fiabilidad del sistema

Además de una medida de fiabilidad, se debe desarrollar una medida de disponibilidad. La *disponibilidad del software* es la probabilidad de que un programa opere de acuerdo con los requisitos en un punto dado del tiempo, y se define como

$$\text{Disponibilidad} = [\text{TMDF}/(\text{TMDF} + \text{TMDR})] \times 100\%$$

La medida de fiabilidad TMEF es igualmente sensible a TMDF y TMDR. La medida de disponibilidad es un poco más sensible a TMDR, y es una medida indirecta de la facilidad de mantenimiento del software

26.7.2 Seguridad del software

La *seguridad del software* [LEV86] es una actividad de aseguramiento de la calidad de software que se enfoca en la identificación y evaluación de los peligros potenciales que pueden afectar negativamente al software y provocar una falla de todo el sistema. Si los peligros se pueden identificar temprano en el proceso de software, las características de diseño de software se pueden especificar de modo que eliminarán o controlarán los peligros potenciales.

> "No puedo imaginar alguna condición que provoque que este barco se hunda. La industria naviera moderna ha ido más allá."
>
> E. J. Smith, capitán del *Titanic*

Como parte de la seguridad del software se llevan a cabo procesos de modelado y análisis. Inicialmente, los peligros se identifican y clasifican por importancia y riesgo. Por ejemplo, algunos de los peligros asociados con un control basado en computadora para la conducción de un automóvil pueden ser:

- Provoca aceleración descontrolada que no se puede detener.
- No responde a la presión del pedal de freno (al apagarlo).
- No responde cuando el interruptor se activa
- Pierde o gana rapidez lentamente.

Una vez identificados estos peligros en el nivel del sistema, mediante técnicas de análisis se asignan severidad y probabilidad de ocurrencia.[7] Para ser eficaz, el software debe analizarse en el contexto de todo el sistema. Por ejemplo, un sutil error de entrada de usuario (las personas son componentes del sistema) tal vez lo magnifique una falla del software para producir datos de control que posicionan de manera inadecuada un dispositivo mecánico. Si se reúne un conjunto de condiciones am-

7 Este enfoque es similar a los métodos de análisis de riesgo descritos en el capítulo 25. La diferencia principal es el énfasis en los conflictos tecnológicos, más que en los tópicos relacionados con el proyecto

Referencia Web

Una colección valiosa de ensayos acerca de seguridad de software se puede encontrar en www.safeware-eng.com/.

bientales externas (y sólo si ellas se reúnen), la posición inadecuada del dispositivo mecánico provocará una falla desastrosa. Las técnicas de análisis, como el análisis del árbol de fallas [VES81], la lógica de tiempo real [JAN86] o los modelos de red de Petri [LEV87], se emplean para predecir la cadena de eventos que pueden provocar peligros y la probabilidad de que cada uno de los eventos ocurrirá para crear la cadena.

Una vez identificados y analizados los peligros, se especifican los requisitos relacionados con la seguridad del software. Esto es, la especificación puede contener una lista de eventos indeseables y las respuestas deseables del sistema ante dichos eventos. Entonces se indica el papel del software en la gestión de los eventos indeseables.

Aunque la confiabilidad del software y su seguridad están estrechamente relacionadas, es importante entender las sutiles diferencias entre ellas. La confiabilidad del software utiliza análisis estadístico para determinar la probabilidad de que ocurrirá una falla del software. Sin embargo, el hecho de que ocurra una falla no necesariamente resulta en un peligro o percance La seguridad del software examina las formas en las cuales las fallas resultan en condiciones que pueden conducir a un percance Esto es, las fallas no son consideradas en el vacío, sino que se evalúan en el contexto de todo un sistema basado en computadora y en su entorno Aquellos lectores con mayor interés deben remitirse al libro de Leveson [LEV95] para profundizar en el tema.

26.8 LOS ESTÁNDARES DE CALIDAD ISO 9000[8]

PUNTO CLAVE

ISO 9000 describe lo que se debe hacer para ser manejable, pero no dice cómo se debe hacer.

Es posible definir un *sistema de garantía de la calidad* como la estructura organizacional, responsabilidades, procedimientos, procesos y recursos para implementar la gestión de la calidad [ANS87]. Los sistemas de garantía de calidad fueron creados para ayudar a las organizaciones a garantizar que sus productos y servicios satisfacen las expectativas de los clientes al cumplir sus especificaciones. El estándar ISO 9000 describe un sistema de garantía de la calidad en términos genéricos que se aplican a cualquier negocio sin importar los productos o servicios ofrecidos.

El registro en uno de los modelos de sistema de garantía de la calidad contenidos en ISO 9000 requiere que los sistemas y operaciones de calidad de una compañía los sometan a escrutinio auditores de una tercera entidad respecto de su concordancia con el estándar y de su funcionamiento eficaz. Antes del registro exitoso, los auditores le extienden a la compañía un certificado de la organización de registro que representan. Entrevistas de auditoría semianuales garantizan la concordancia continua con el estándar.

8 Esta sección, escrita por Michael Slovsky, se ha adaptado de "Fundamentals of ISO 9000", un libro de trabajo desarrollado por *Essential Software Engineering*, un video curriculum elaborado por R. S. Pressman & Associates, Inc. Reimpreso con permiso.

Referencia Web
Extensos vínculos hacia recursos ISO 9000/9001 se pueden encontrar en **www.fontanero.ab.ca/info.htm.**

El estándar de garantía de la calidad que se aplica a la ingeniería del software es el ISO 9001:2000. El estándar contiene 20 requisitos que deben estar presentes para un sistema eficiente de garantía de la calidad. Puesto que el estándar ISO 9001:2000 es aplicable a todas las disciplinas de ingeniería, se ha desarrollado un conjunto especial de directrices ISO (ISO 9000-3) que ayudan a interpretar el estándar para emplearlo en el proceso de software.

Los requisitos que delinea ISO 9001:2000 abordan tópicos como responsabilidad de la gestión, sistema de calidad, revisión de contrato, control de diseño, control de documentos y datos, identificación y seguimiento de producto, control de proceso inspección y pruebas, acciones correctivas y preventivas, control de registros de calidad, auditorías de calidad interna, entrenamiento, servicio y técnicas estadísticas. Una organización de software obtendrá el registro ISO 9001:2000 si establece políticas y procedimientos para abordar cada uno de los requisitos anotados líneas arriba (y otros) y, además, ser capaz de demostrar que se siguen dichas políticas y procedimientos. Para mayor información acerca de ISO 9001, el lector interesado debe consultar [HOY02], [GAA01] o [CIA01]

INFORMACIÓN

El estándar ISO 9001:2000

Las siguientes líneas generales definen los elementos básicos del estándar ISO 9001:2000. Información más amplia acerca del estándar se puede obtener de International Organization for Standardization (www.iso.ch) y en otras fuentes de Internet (por ejemplo, www.praxiom.com).

Establecer los elementos de un sistema de gestión de calidad.
- Desarrollar, implementar y mejorar el sistema.
- Definir una política que enfatice la importancia del sistema.

Documentar el sistema de calidad
- Describir el proceso.
- Producir un manual operativo.
- Desarrollar métodos para controlar (actualizar) los documentos.
- Establecer métodos para la conservación de registros.

Soporte del control y la garantía de calidad.
- Promover la importancia de la calidad entre todos los participantes
- Enfocarse en la satisfacción del cliente
- Definir un plan de calidad que aborde objetivos, responsabilidades y autoridad.
- Definir mecanismos de comunicación entre los participantes

Establecer mecanismos de revisión para el sistema de gestión de calidad.
- Identificar métodos de revisión y mecanismos de retroalimentación
- Definir procedimientos de seguimiento.
- Identificar recursos de calidad que incluyan personal, entrenamiento, elementos de infraestructura

Establecer mecanismos de control.
- Para planeación.
- Para requisitos del cliente.
- Para actividades técnicas (por ejemplo, análisis, diseño, pruebas).
- Para supervisión y gestión del proyecto.

Definir métodos para corrección
- Valorar los datos y métricas de calidad.
- Definir enfoques para procesos continuos y mejora de la calidad.

26.8 El plan de SQA

El *plan de SQA* proporciona un mapa para instituir la garantía de la calidad del software. Desarrollado por el grupo de SQA (o el equipo de software si no existe un grupo SQA), el plan funciona como plantilla para las actividades de SQA que se instituyan para cada proyecto de software.

En el IEEE [IEE94] se ha publicado un estándar para planes de SQA. El estándar recomienda una estructura que identifica: 1) el propósito y ámbito del plan; 2) una descripción de todos los productos de trabajo de ingeniería del software (por ejemplo, modelos, documentos, código fuente) que caen dentro del alcance del SQA; 3) todos los estándares y prácticas aplicables que se aprovechan durante el proceso de software; 4) acciones y tareas de SQA (incluso revisiones y auditorías) y su ubicación a través del proceso de software; 5) las herramientas y métodos que soportan las acciones y tareas de SQA; 6) procedimientos de gestión de configuración de software (capítulo 27) para gestionar el cambio; 7) métodos para ensamblar, salvaguardar y mantener todos los registros relacionados con el SQA; y 8) papeles y responsabilidades en la organización relativas a la calidad de producto.

HERRAMIENTAS DE SOFTWARE

Gestión de la calidad del software

Objetivo: El objetivo de las herramientas de SQA es auxiliar al equipo de proyecto para valorar y mejorar la calidad del producto de trabajo de software.

Mecánica: La mecánica de las herramientas varía. En general, la finalidad es valorar la calidad de un producto de trabajo específico. Nota: con frecuencia, dentro de la categoría de herramientas de SQA, se incluye una amplia variedad de herramientas de prueba de software (capítulos 13 y 14).

Herramientas representativas[9]

ARM, desarrollado por la NASA (satc.gsfc.nasa.gov/tools/index.html), ofrece medidas con las cuales se evalúa la calidad de un documento de requisitos de software.

QPR ProcessGuide and Scorecard, desarrollada por QPR Software (www.qpronline.com), ofrece soporte para seis sigma y otros enfoques de gestión de calidad.

Quality Tools Cookbook, desarrollado por Sytsma and Manley (www.sytsma.com/tqmtools/tqmtoolmenu.html), proporciona descripciones útiles de herramientas clásicas de gestión de calidad tales como los diagramas de control, diagramas de dispersión, diagramas de afinidad y diagramas de matriz.

Quality Tools and Templates, desarrolladas por iSixSigma (http://www.isixsigma.com/tt/), describe una amplia variedad de herramientas y métodos útiles para gestión de calidad.

TQM Tools, desarrollado por Bain & Company (www.bain.com), brinda descripciones útiles de una variedad de herramientas de gestión usadas por TQM y relacionadas con los métodos de gestión de calidad.

9 Las herramientas expuestas sólo representan una muestra de esta categoría. En la mayoría de los casos los nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

26.10 RESUMEN

La gestión de la calidad del software es una actividad protectora o de sombrilla —que incorpora tanto control como aseguramiento de la calidad— que se aplica a cada paso en el proceso del software. La SQA abarca procedimientos para la aplicación eficaz de métodos y herramientas, revisiones técnicas formales, estrategias y técnicas de pruebas, procedimientos para control del cambio, procedimientos para garantizar la concordancia con los estándares y mecanismos de medición y reporte.

La SQA la complica la naturaleza compleja de la calidad del software, un atributo de los programas de computadora que se define como "concordancia con los requisitos especificados explícita e implícitamente". Pero cuando se considera de manera más general, la calidad de software abarca muchos productos diferentes, y factores de proceso y métricas relacionadas.

Las revisiones de software son una de las actividades de control de calidad más importantes. Las revisiones sirven como filtros a través de todas las actividades de ingeniería del software, que eliminan los errores mientras son relativamente poco costosos de encontrar y corregir. La revisión técnica formal es una junta que ha demostrado ser extremadamente eficaz para descubrir errores.

La actividad adecuada para garantizar la calidad del software requiere recopilar, evaluar y distribuir los datos acerca de los procesos de ingeniería del software. La SQA estadística ayuda a mejorar la calidad del producto y el proceso de software en sí mismo. Los modelos de fiabilidad del software extienden medidas, lo que permite recopilar datos de defecto para extrapolarlos en las tasas de falla proyectadas y las predicciones de fiabilidad.

En resumen, recuérdense las palabras de Dunn y Ullman [DUN82]: "El aseguramiento de la calidad del software es el mapeo (correlación) de los preceptos gerenciales y las disciplinas de diseño de la garantía de calidad en el espacio gerencial y tecnológico aplicable de la ingeniería del software". La habilidad para garantizar la calidad es la medida de una disciplina de ingeniería madura. Cuando el mapeo se logra de manera exitosa, el resultado es la ingeniería de software madura.

REFERENCIAS

[ALV64] Alvin, W. H., von (ed.), *Reliability Engineering*, Prentice-Hall, 1964.

[ANS87] ANSI/ASQC A3-1987, *Quality Systems Terminology*, 1987.

[ART92] Arthur, L. J., Improving Software Quality: an Insider's Guide to TQM, Wiley, 1992.

[ART97] Arthur, L. J., "Quantum Improvements in Software System Quality, en *CACM*, vol. 40, núm. 6, junio de 1997, pp. 47-52.

[BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.

[CIA01] Cianfrani, C. A. *et al.*, *ISO 9001:2000 Explained*, 2a. ed., American Society for Quality, 2001.

[CRO79] Crosby, P., *Quality Is Free*, McGraw-Hill, 1979.

[DEM86] Deming, W. E., *Out of the Crisis*, MIT Press, 1986.

[DEM99] DeMarco, T., "Management Can make Quality (Im)possible", Cutter IT Summit, Boston, abril de 1999.

[DIJ76] Dijkstra, E., *A Discipline of Programming*, Prentice-Hall, 1976.
[DUN82] Dunn, R., y R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982.
[FRE90] Freedman, D. P. y G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, 3a. ed., Dorset House, 1990.
[GAA01] Gaal, A., *ISO 9001:2000 for Small Business*, Saint Lucie Press, 2001.
[GIL93] Gilb, T., y D. Graham, *Software Inspections*, Addison-Wesley, 1993.
[GLA98] Glass, R., "Defining Quality Intuitively", en *IEEE Software*, mayo de 1998, pp. 103-104, 107.
[GOH02] Goh, T., V. Kuralmani y M. Xie, *Statistical Models and Control Charts for High Quality Processes*, Kluwer Academic Publishers, 2002.
[HOY02] Hoyle, D., *ISO 9000 Quality Systems Development Handbook: A Systems Engineering Approach*, 4a. ed., Butterworth Heinemann, 2002.
[IBM81] "Implementing Software Inspections", notas de curso, IBM Systems Sciences Institute, IBM Corporation, 1981.
[IEE94] *Software Engineering Standards*, 1994, IEEE Computer Society, 1994.
[ISI03] iSixSigma, LLC, "New to Six Sigma: A Guide for Both Novice and Experienced Quality Practitioners", 2003, disponible en http://www.isixsigma.com/library/content/six-sigma-newbie.asp.
[JAN86] Jahanian, F. y A. K. Mok, "Safety Analysis of Timing Properties of Real-Time Systems", en *IEEE Trans. Software Engineering*, vol. SE-12, núm. 9, septiembre de 1986, pp. 890-904.
[JON86] Jones, T. C., *Programming Productivity*, McGraw-Hill, 1986.
[KAN95] Kan, S. H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995.
[LEV86] Leveson, N. G., "Software Safety: Why, What, and How", en *ACM Computing Surveys*, vol. 18, núm. 2, junio de 1986, pp. 125-163.
[LEV87] Leveson, N. G. y J. L. Stolzy, "Safety Analysis Using Petri Nets", en *IEEE Trans. Software Engineering*, vol. SE-13, núm. 3, marzo de 1987, pp. 386-397.
[LEV95] Leveson, N. G., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.
[LIN79] Linger, R., H. Mills y B. Witt, *Structured Programming*, Addison-Wesley, 1979.
[LIT89] Littlewood, B., "Forecasting Software Reliability", en *Software Reliability: Modeling and Identification* (S. Bittanti, ed.), Springer-Verlag, 1989, pp. 141-209.
[MUS87] Musa, J. D., A. Iannino y K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.
[PAN00] Nande, P. *et al.*, *The Six Sigma Way*, McGraw-Hill, 2000.
[POR95] Porter, A., H. Siy, C. A. Toman y L. G. Volta, "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development", en *Proc. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, D.C., octubre de 1995, ACM Press, pp. 92-103.
[ROO90] Rook, J., *Software Reliability Handbook*, Elsevier, 1990.
[SCH98] Schulmeyer, G. C. y J. I. McManus (eds.), *Handbook of Software Quality Assurance*, 3a. ed., Prentice-Hall, 1998.
[SOM01] Somerville, I., *Software Engineering*, 6a. ed., Addison-Wesley, 2001.
[SNE03] Snee, R. y R. Hoerl, *Leading Six Sigma*, Prentice-Hall, 2003.
[THE01] Thelin, T., H. Petersson y C. Wohlin, "Sample Driven Inspections", en *Proceedings Workshop on Inspection in Software Engineering (WISE'01)*, Paris, Francia, julio de 2001, pp. 81-91, se puede descargar de http://www.cas.mcmaster.ca/wise/wise01/thelinPetersson-Wohlin.pdf.
[VES81] Veseley, W. E. *et al.*, *Fault Tree Handbook*, U.S. Nuclear Regulatory Commission, NUREG-0492, enero de 1981.

PROBLEMAS Y PUNTOS A CONSIDERAR

26.1. En las primeras páginas de este capítulo se anotó que "el control de la variación es el corazón del control de calidad". Dado que cualquier programa que se crea es diferente de todos los otros programas, ¿cuáles son las variaciones que se buscan y cómo se les controla?

26.2. ¿Es posible valorar la calidad del software si el cliente cambia con frecuencia lo que se supone debe hacer?

26.3. La calidad y la fiabilidad son conceptos relacionados pero fundamentalmente diferentes en varias formas. Coméntense.

26.4. ¿Puede un programa ser correcto y aún así no ser fiable? Explíquese.

26.5. ¿Puede un programa ser correcto y aún así no mostrar buena calidad? Explíquese.

26.6. ¿Por qué con frecuencia existe tensión entre un grupo de ingeniería de software y un grupo independiente de aseguramiento de la calidad del software? ¿Esto es saludable?

26.7. A usted le han dado la responsabilidad de mejorar la calidad del software por medio de su organización. ¿Qué es lo primero que debe hacer? ¿Qué sería lo siguiente?

26.8. Además de contar errores y defectos, ¿existen otras características contables del software que impliquen calidad? ¿Cuáles son y cómo se pueden medir directamente?

26.9. Una revisión técnica formal sólo es eficaz si todos se han preparado por anticipado. ¿Cómo reconoce en la revisión a un participante que no está preparado? ¿Qué hace si usted es el jefe de revisión?

26.10. Algunas personas argumentan que una RTF debe valorar el estilo de programación, así como la corrección. ¿Ésta es una buena idea? ¿Por qué?

26.11. Revise la tabla presentada en la figura 26.5 y seleccione cuatro causas vitales de errores serios y moderados. Sugiera acciones correctivas empleando la información presentada en otros capítulos.

26.12. Investigue la bibliografía acerca de la fiabilidad del software y escriba un ensayo que describa un modelo de fiabilidad de software. Asegúrese de proporcionar un ejemplo.

26.13. El concepto de TMEF para software está abierto a críticas. ¿Puede pensar en algunas razones de por qué sucede así?

26.14. Considere dos sistemas críticos de seguridad que se controlan mediante computadoras. Haga una lista con al menos tres peligros para cada uno que puedan estar ligados directamente con las fallas de software.

26.15. Adquiera una copia de ISO 9001:2000 e ISO 9000-3. Prepare una presentación que examine tres requisitos ISO 9001 y cómo se aplican en un contexto de software.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Los libros de Moriguchi (*Software Excellence: A Total Quality Management Guide*, Productivity Press, 1997) y Horch (*Practical Guide to Software Quality Management*, Artech Publishing, 1996) son excelentes presentaciones, en el ámbito gerencial, de los beneficios de los programas formales de aseguramiento de la calidad del software de computadora. Los libros de Deming [DEM86], Juran (*Juran on Quality by Design*, Free Press, 1992) y Crosby ([CRO79] y *Quality Is Still Free*, McGraw-Hill, 1995) no se enfocan en el software, pero son una lectura obligada para los gestores ejecutivos con responsabilidad en el desarrollo de software. Gluckman y Roome (*Everyday Heroes of the Quality Movement*, Dorset House, 1993) humanizan los temas de calidad al contar la historia de los actores en el proceso de calidad. Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995) presenta una visión cuantitativa de la calidad del software.

Cianfani y sus colegas (*ISO 9001:2000 Explained*, segunda edición, American Society for Quality, 2001) y Gaal (*ISO 9001:2000 for Small Business: Implementing Process-Approach Quality Management*, St. Lucie Press, 2001) estudian el estándar de calidad ISO 9001:2000. Tingley (*Comparing ISO 9000, Malcolm Baldrige, and the SEI CMM for Software*, Prentice-Hall, 1996) ofrecen una guía útil para las organizaciones que luchan por mejorar sus procesos de gestión de calidad.

Los libros de George (*Lean Six Sigma*, McGraw-Hill, 2002), Pande y sus colegas (*The Six Sigma Way Fieldbook*, McGraw-Hill, 2001) y Pyzdek (*The Six Sigma Handbook*, McGraw-Hill, 2000)

describen seis sigma, una técnica estadística de gestión de calidad que conduce a productos que tienen muy bajas tasas de defectos

Radice (*High Quality, Low Cost Software Inspections*, Paradoxicon Publishers, 2002), Wiegers (*Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2001), Gilb y Graham (*Software Inspection*, Addison-Wesley, 1993) y Freedman y Weinberg (*Handbook of Walkthroughs, Inspections and Technical Reviews*, Dorset House, 1990) ofrecen directrices valiosas para llevar a cabo revisiones técnicas formales efectivas.

Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, McGraw-Hill, 1998) ha escrito una guía práctica para técnicas de fiabilidad de software aplicado. Kapur *et al.* (*Contributions to Hardware and Software Reliability Modeling*, World Scientific Publishing Co., 1999), Gritzalis (*Reliability, Quality and Safety of Software-Intensive Systems*, Kluwer Academic Publishers, 1997) y Lyu (*Handbook of Software Reliability Engineering*, McGraw-Hill, 1996) han editado antologías de importantes ensayos acerca de la fiabilidad del software

Hermann (*Software Safety and Reliability*, Wiley-IEEE Press, 2000), Storey (*Safety Critical Computer Systems*, Addison-Wesley, 1996) y Leveson [LEV95] continúan siendo los estudios más detallados de la seguridad del software publicados a la fecha. Además, van der Meulen (*Definitions for Hardware and Software Safety Engineers*, Springer-Verlag, 2000) ofrece un compendio completo de importantes conceptos y términos de fiabilidad y seguridad. Gartner (*Testing Safety-Related Software*, Springer-Verlag, 1999) ofrece una guía especializada para probar sistemas cruciales de seguridad. Friedman y Voas (*Software Assessment: Reliability Safety and Testability*, Wiley, 1995) ofrecen modelos útiles para valorar la fiabilidad y la seguridad

En Internet hay disponible una amplia variedad de fuentes de información acerca de la gestión de calidad de software. Una lista actualizada de referencias en la World Wide Web se puede encontrar en el sitio Web SEPA:
http://www.mhhe.com/pressman.

CAPÍTULO

27 GESTIÓN DEL CAMBIO

CONCEPTOS CLAVE

El cambio es inevitable cuando se construye software de computadora. Y el cambio aumenta el grado de confusión entre los ingenieros de software que trabajan en un proyecto. La confusión surge cuando los cambios no se analizan antes de realizarlos, no se registran antes de implementarlos, no se reportan a quienes deben saberlo o no se controlan en una forma que mejorará la calidad y reducirá el error. Babich [BAB86] aborda esto cuando afirma:

> El arte de coordinar el desarrollo de software para minimizar la confusión se llama gestión de la configuración. La gestión de la configuración es el arte de identificar, organizar y controlar modificaciones al software que se construye por medio de un equipo de programación. La meta es maximizar la productividad al minimizar las equivocaciones.

La gestión del cambio, más usualmente llamada *gestión de la configuración del software* (GCS o GC), es una actividad protectora (sombrilla) que se aplica a lo largo del proceso de software. Puesto que el cambio puede ocurrir en cualquier momento, las actividades de GCS se desarrollan para 1) identificar el cambio, 2) controlar el cambio, 3) garantizar que el cambio se implementará de manera adecuada, y 4) reportar los cambios a otros que pudieran estar interesados.

Es importante distinguir con claridad entre soporte de software y gestión de la configuración del software. El soporte es un conjunto de actividades de ingeniería del software que ocurren después de que éste se ha entregado al cliente y fue puesto en operación. La gestión de la configuración del software es un conjunto de actividades de seguimiento y control que se inician cuando comienza un proyecto de ingeniería del software y terminan sólo cuando éste se retira de operación.

UN VISTAZO RÁPIDO

¿Qué es? Cuando se construye software de computadora los cambios ocurren. Y puesto que ocurren, es necesario gestionarlos con eficacia. La gestión del cambio, también llamada gestión de la configuración del software (GCS), es un conjunto de actividades diseñadas para gestionar el cambio al identificar los productos de trabajo que probablemente cambien, establecer relaciones entre ellos, definir mecanismos para gestionar diferentes versiones de estos productos de trabajo, controlar los cambios impuestos y auditar e informar los cambios realizados.

¿Quién lo hace? Todos los involucrados en el proceso de software están involucrados con la gestión del cambio en alguna medida, pero en ocasiones se crean posiciones de soporte especializado para gestionar el proceso de GCS.

¿Por qué es importante? Si no se controla el cambio, él toma el control. Y eso nunca es bueno. Es muy fácil que una corriente de cam-

bios incontrolados convierta en caótico un proyecto de software bien implementado. Por esta razón, la gestión del cambio es una parte esencial de la buena gestión del proyecto y de una sólida práctica de ingeniería de software

¿Cuáles son los pasos? Puesto que muchos productos de trabajo se producen cuando se construye el software, cada uno debe identificarse en forma individual. Una vez hecho esto se establecen los mecanismos de control de versión y cambio. El proceso se audita para garantizar que la calidad se conserva conforme el cambio se realiza y que quienes tienen necesidad de conocerlo reciben información acerca de los cambios mediante los informes respectivos

¿Cuál es el producto obtenido? Un plan de gestión de la configuración del software define la estrategia del proyecto para la gestión del cambio. Además, cuando se pide una GCS formal, el proceso de control del cambio produce solicitudes de cambio de software, informes y peticiones de cambio de ingeniería.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Cuando cualquier producto de trabajo puede explicarse, seguirse y controlarse; cuando los cambios puede seguirse y analizarse; cuando todos los que necesitan saber acerca de un cambio han sido informados, el trabajo se ha hecho bien

Una meta primordial de la ingeniería del software es mejorar la facilidad con la que los cambios se pueden acomodar y reducir el esfuerzo cuando los cambios se deben realizar. En este capítulo se estudian las acciones específicas que permiten gestionar el cambio.

27.1 Gestión de la configuración del software

La salida del proceso de software es información que se puede dividir en tres amplias categorías: 1) programas de computadora (tanto al nivel de fuente como de formas ejecutables); 2) productos de trabajo que describen los programas de computadora (dirigidos tanto a profesionales técnicos como a usuarios), y 3) datos (internos o externos al programa). Los elementos que comprenden la información producida como parte del proceso de software se denominan colectivamente *configuración del software*.

Si cada elemento de configuración simplemente condujera a otros elementos habría poca confusión. Por desgracia, otra variable entra en el proceso: el *cambio*. Éste puede ocurrir en cualquier momento, por cualquier razón. De hecho, la primera ley de la ingeniería de sistemas [BER80] afirma: "No importa dónde se encuentre en el ciclo de vida del sistema, el sistema cambiará y el deseo de cambiarlo persistirá a lo largo de todo el ciclo de vida".

> "No hay nada permanente, excepto el cambio."
>
> Heráclito, 500 a.C.

¿Cuál es el origen de estos cambios? La respuesta es tan variada como los cambios mismos. Sin embargo, existen cuatro fuentes fundamentales de cambio:

¿Cuál es el origen de los cambios que se requieren para el software?

- Nuevas condiciones en el negocio o mercado dictan los cambios en los requisitos del producto o las reglas del negocio.
- Nuevas necesidades del cliente demandan la modificación de los datos que producen los sistemas de información, de la funcionalidad que entregan los productos o los servicios que entrega un sistema basado en computadora
- La reorganización o el crecimiento o reducción del negocio provocan cambios en las prioridades del proyecto o en la estructura del equipo de ingeniería del software.
- Restricciones presupuestales o de calendarización inducen una redefinición del sistema o producto.

La gestión de la configuración del software es un conjunto de actividades que se ha desarrollado para gestionar el cambio a lo largo del ciclo de vida del software de computadora La GCS se considera como una actividad de aseguramiento de la calidad del software que se aplica a lo largo del proceso respectivo. En las secciones siguientes se examinan las principales tareas de la GCS y conceptos importantes que ayudan a gestionar el cambio.

27.1.1 Un escenario de GCS[1]

Un típico escenario operativo de GCS involucra un gestor de proyecto a cargo de un grupo de software; un gestor de configuración a cargo de los procedimientos y políticas de GC; los ingenieros de software responsables del desarrollo y mantenimiento del producto de software, y el cliente que emplea el producto. En el escenario supóngase que el producto es pequeño e involucra cerca de 15 000 líneas de código que desarrollará un equipo de seis personas. (Nótese que son posibles otros escenarios de equipos menores o mayores, pero, en esencia, existen conflictos genéricos que cada uno de estos proyectos enfrenta en relación con la GC.)

En el ámbito operativo el escenario involucra diversos papeles y tareas. La meta del gestor del proyecto es garantizar que el producto se entregue dentro de cierto periodo. En consecuencia, el gestor supervisa el progreso del desarrollo y reconoce y reacciona ante los problemas Esto se hace al generar y analizar los informes acerca del estado del sistema de software y al realizar revisiones en el sistema.

¿Cuáles son las metas y las actividades realizadas por cada uno de los participantes involucrados en la gestión del cambio?

Las metas del gestor de configuración son garantizar que se siguen los procedimientos y políticas para crear, cambiar y poner a prueba el código, así como posibilitar el acceso a la información acerca del proyecto. La implementación de técnicas para mantener el control sobre los cambios de código requiere que este gestor introduzca mecanismos para solicitar oficialmente cambios, evaluarlos (mediante una junta de control de cambios, que es la responsable de aprobar los cambios al siste-

1 Esta sección procede de [DAR01]. El permiso especial para reproducir "Spectrum of Functionality in CM Systems" de Susan Dart [DAR01], © 2001 por Carnegie Mellon University, lo otorgó el Software Engineering Institute.

ma de software) y autorizarlos. El gestor crea y distribuye las listas de tareas para los ingenieros y básicamente crea el contexto del proyecto Además, el gestor recopila estadísticas acerca de componentes en el sistema de software, por ejemplo: la información que determina cuáles componentes son problemáticos en el sistema.

La meta de los ingenieros de software es trabajar con eficiencia. Esto significa que no interfieren de manera innecesaria unos con otros en la creación y prueba del código ni en la producción de los documentos de soporte. No obstante, al mismo tiempo, intentan comunicarse y coordinarse de manera eficiente. Específicamente, los ingenieros utilizan herramientas que ayudan a construir un producto de software consistente. Ellos se comunican y coordinan al notificarse mutuamente las tareas que se requieren y las tareas cumplidas Los cambios se propagan por medio del trabajo de cada uno mediante archivos fusionados. Existen mecanismos para asegurar que, respecto de los componentes que experimentan cambios simultáneos, existe alguna forma de resolver los conflictos y fusionar los cambios. La historia de la evolución de todos los componentes del sistema se mantiene junto con un registro de las razones de los cambios y otro de lo que cambió en realidad. Los ingenieros tienen su propio espacio de trabajo para crear, cambiar, probar e integrar código. En cierto punto, el código se convierte en una línea base a partir de la que continúa el desarrollo posterior y desde la que se realizan las variantes para otras máquinas que también sean el objetivo.

El cliente emplea el producto. Dado que el producto lo controla la GC, el cliente sigue procedimientos formales para solicitar cambios e indicar los *bugs* en el producto.

Idealmente, un sistema de GC utilizado en este escenario apoyaría todas estas funciones y tareas; esto es, las funciones determinan la funcionalidad requerida de un sistema de GC. El gestor del proyecto ve una GC como un mecanismo de auditoría; el gestor de configuración, como un mecanismo de creación de control, seguimiento y políticas; el ingeniero de software, como un mecanismo de control del cambio, la construcción y el acceso; y el usuario, como un mecanismo de garantía de la calidad.

27.1.2 Elementos de un sistema de gestión de la configuración

En su detallado artículo acerca de la gestión de la configuración del software, Susan Dart [DAR01] identifica cuatro importantes elementos que deben estar presentes cuando se desarrolla un sistema de gestión de la configuración:

- *Elementos de componentes*. conjunto de herramientas acopladas dentro de un sistema de gestión de archivos (por ejemplo, una base de datos) que permiten el acceso y la gestión de cada elemento de configuración del software.
- *Elementos de proceso*: serie de procedimientos y tareas que definen un enfoque eficaz con el cual gestionar el cambio (y actividades relacionadas) para todos los participantes en la gestión, ingeniería y utilización del software de computadora.

- *Elementos de construcción:* conjunto de herramientas que automatizan la construcción del software al asegurar que se ha ensamblado un conjunto adecuado de componentes validados (es decir: la versión correcta).
- *Elementos humanos:* la implementación de una GCS eficaz requiere que el equipo de software utilice un conjunto de herramientas y características de procesos (que abarcan otros elementos de GC).

Estos elementos (que se estudiarán con más detalle en secciones venideras) no son mutuamente excluyentes. Por ejemplo, los elementos de componentes trabajan en conjunto con los de construcción conforme avanza el proceso de software. Los elementos de proceso guían muchas actividades humanas que se relacionan con GCS y, por tanto, también pueden considerarse elementos humanos.

27.1.3 Líneas base

La mayoría de los cambios de software están justificados, así que no hay razón para quejarse acerca de ellos. Más bien, es necesario asegurarse de que se tienen los mecanismos apropiados para manejarlos.

El cambio es un hecho de vida en el desarrollo del software. Los clientes quieren modificar los requisitos. Los desarrolladores quieren modificar el enfoque técnico. Los gestores quieren modificar la estrategia del proyecto. ¿Por qué todas estas modificaciones? La respuesta, en realidad, es bastante simple. Conforme pasa el tiempo, todos los participantes saben más (acerca de lo que necesitan, qué enfoque sería el mejor, cómo hacerlo y aun así obtener dinero). Este conocimiento adicional es la fuerza impulsora detrás de la mayoría de los cambios y conduce a una expresión difícil de aceptar para muchos profesionales de la ingeniería del software: *¡la mayoría de los cambios están justificados!*

Una *línea base* es un concepto de gestión de la configuración del software que ayuda a controlar el cambio sin impedir seriamente el cambio justificable. El IEEE (IEEE Std. No. 610.12-1990) define una línea base como:

> Una especificación o producto que se ha revisado formalmente y se está de acuerdo con los resultados, y que a partir de ahí sirve como la base para el desarrollo ulterior y que puede cambiarse sólo por medio de procedimientos formales de control del cambio

Antes de que un elemento de configuración del software se convierta en línea base, es posible realizar el cambio rápida e informalmente. Sin embargo, una vez establecida una línea base, metafóricamente se pasa a través de una puerta giratoria de una sola dirección. Los cambios se pueden realizar, pero se debe aplicar un procedimiento específico formal para evaluar y verificar cada uno.

Un producto de trabajo de ingeniería del software se convierte en línea base sólo después de que se ha revisado y aprobado.

En el contexto de la ingeniería del software, una línea base es un hito en el desarrollo del software. Se marca una línea base para la entrega de uno o más elementos de configuración del software (ECS) que se han aprobado como consecuencia de una revisión técnica formal (capítulo 26). Por ejemplo, los elementos de un modelo de diseño se han documentado y revisado. Se han encontrado errores y se han corregido. Una vez que todas las partes del modelo se han revisado, corregido y luego aprobado, el modelo de diseño se convierte en línea base. Los cambios posteriores a la arquitectura del programa (documentados en el modelo de diseño) sólo se pue-

CONSEJO

Es preciso asegurarse de que la base de datos del proyecto se mantiene en una ubicación central controlada.

den efectuar después de que cada uno se ha evaluado y aprobado. Aunque las líneas base se pueden definir en cualquier grado de detalle, en la figura 27.1 se muestran las líneas base de software más comunes.

En la figura 27.1 también se muestra la progresión de eventos que conducen a una línea base. Las tareas de ingeniería del software producen uno o más ECS. Después de que éstos se revisan y aprueban se colocan en una *base de datos del proyecto* (también llamada *librería del proyecto* o *depósito de software*, que se examinan en la sección 27.2). Cuando un miembro de un equipo de software quiere modificar un ECS que se ha convertido en línea base, se copia de la base de datos del proyecto en el espacio de trabajo privado del ingeniero. Sin embargo, este ECS extraído sólo se puede modificar si se siguen los controles de la GCS (tratados más adelante en este capítulo). Las flechas en la figura 27.1 ilustran la trayectoria de modificación para un ECS convertido en línea base.

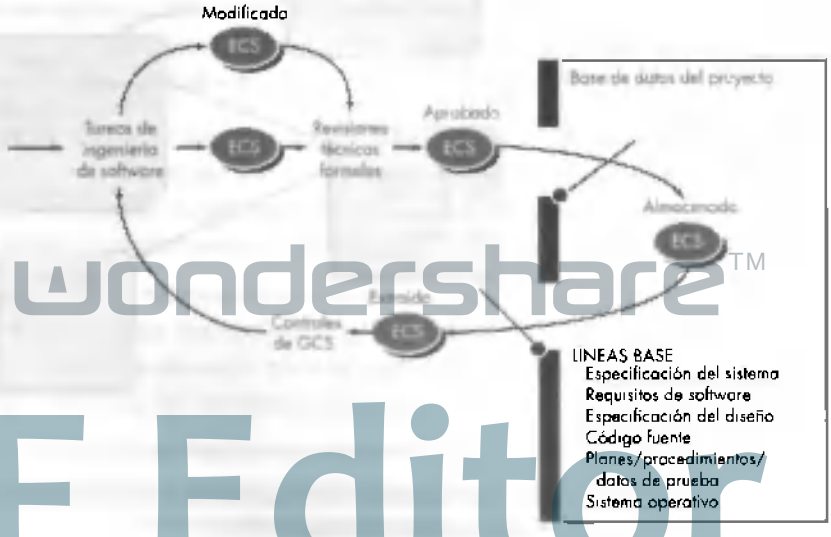
27.1.4 Elementos de configuración del software

Un elemento de configuración del software (ECS) es información que se crea como parte del proceso de ingeniería del software. En el extremo, se puede considerar que un ECS es una sola sección de una gran especificación o un caso de prueba de un gran conjunto de pruebas. De manera más realista, un ECS es un documento, un conjunto completo de casos de prueba o un componente de un programa dado (por ejemplo, una función C++ o un *applet* de Java).

Además de los ECS provenientes de los productos de trabajo de software, muchas organizaciones de ingeniería del software también colocan las herramientas respectivas bajo control de configuración. Esto es: versiones específicas de editores, com-

FIGURA 27.1

ECS convertidos en línea base y base de datos del proyecto.

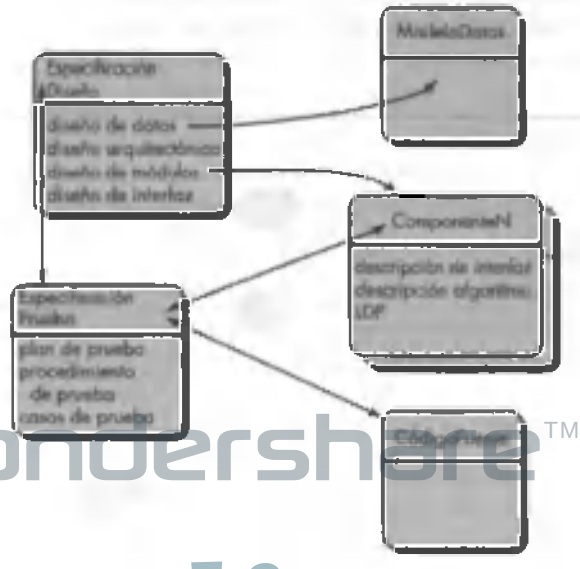


piladores, navegadores y otras herramientas automatizadas se "congelan" como parte de la configuración del software. Puesto que dichas herramientas se utilizan para producir documentación, código fuente y datos, deben estar disponibles al realizar cambios en la configuración del software. Aunque los problemas son raros, es posible que una nueva versión de una herramienta (por ejemplo, un compilador) produzca resultados diferentes a los de la versión original. Por esta razón, las herramientas, al igual que el software que ayudan a producir, pueden convertirse en línea base como parte de un proceso global de gestión de configuración.

En realidad, los ECS están organizados para formar objetos de configuración susceptibles de catalogar en la base de datos del proyecto con un solo nombre. Un *objeto de configuración* tiene un nombre, atributos y está "conectado" con otros objetos por medio de relaciones. Si se observa la figura 27.2, los objetos de configuración **EspecificacionDiseño**, **ModeloDatos**, **ComponenteN**, **CodigoFuente** y **EspecificacionPrueba** están definidos por separado. Sin embargo, cada uno de los objetos se relaciona con los otros como lo muestran las flechas. Una flecha curva indica una relación de composición. Esto es: **ModeloDatos** y **ComponenteN** son parte del objeto **EspecificacionDiseño**. Una flecha recta con doble punta indica una interrelación. Si se realizase un cambio al objeto **CodigoFuente**, las interrelaciones permiten que un ingeniero de software determine qué otros objetos (y ECS) pueden afectarse.[2]

FIGURA 27.2

Objetos de configuración.



[2] Estas relaciones se definen dentro de la base de datos. La estructura de la base de datos (almacén) se estudia con mayor detalle en la sección 27.2.

27.2 El depósito de ECS

En los primeros días de la ingeniería del software los elementos de configuración se conservaban como documentos de papel (¡o tarjetas perforadas!), que se colocaban en cartapacios o carpetas de anillos y se almacenaban en archiveros metálicos. Este enfoque era problemático por muchas razones. 1) con frecuencia era difícil encontrar un elemento de configuración cuando se le necesitaba; 2) usualmente era un reto determinar cuál elemento había sido cambiado, cuándo y por quién; 3) la construcción de una nueva versión de un programa existente consumía mucho tiempo y era proclive al error; 4) la descripción de relaciones detalladas o complejas entre elementos de configuración era virtualmente imposible.

En la actualidad, los ECS se conservan en una base de datos o depósito del proyecto. El diccionario Webster define la palabra *depósito* como "cualquier cosa o persona que se considera como centro de acumulación o almacenamiento". En los inicios de la ingeniería del software, el depósito de hecho era una persona: el programador, quien tenía que recordar la ubicación de toda la información relevante para un proyecto de software; además, tenía que recuperar la información que nunca se había respaldado por escrito y reconstruir la información perdida. Tristemente, emplear a una persona como "centro de acumulación y almacenamiento" (aunque concuerde con la definición del diccionario) no funciona muy bien. Hoy el depósito es una "cosa": una base de datos que actúa como el centro tanto de la acumulación como del almacenamiento de la información de ingeniería del software. El papel de la persona (el ingeniero de software) es interactuar con el depósito mediante las herramientas que tiene integradas.

27.2.1 El papel del depósito

El depósito de ECS es el conjunto de mecanismos y estructuras de datos que permite que un equipo de software maneje el cambio en una forma eficaz. El depósito proporciona las funciones obvias de un sistema de gestión de base de datos pero, además, el depósito realiza o impulsa las siguientes funciones [FOR89]:

¿Qué funciones implementa un depósito de ECS?

- La *integridad de los datos* incluye funciones para validar las entradas al depósito, garantizar la consistencia entre objetos relacionados y automáticamente realizar modificaciones "en cascada" cuando un cambio en un objeto demanda algún cambio a los objetos relacionados con él.
- El *compartir información* ofrece un mecanismo para distribuir la información entre múltiples desarrolladores y herramientas, manejar y controlar los accesos a los datos por parte de múltiples usuarios y cerrar y abrir los objetos de modo que los cambios no sean trasladados inadvertidamente hacia otros.
- La *integración de herramientas* establece un modelo de datos al que se puede tener acceso mediante muchas herramientas de ingeniería del software, controlar el acceso a los datos y realizar funciones adecuadas de gestión de la configuración.

- La *integración de los datos* brinda funciones de base de datos que permiten que varias tareas de GCS se realicen en uno o más ECS.
- El *fortalecimiento de la metodología* define un modelo de entidad-relación guardado en el depósito que implica un modelo de proceso específico para la ingeniería del software, como mínimo, las relaciones y objetos definen un conjunto de pasos que se deben llevar a cabo para construir los contenidos del depósito.
- *Estandarización de los documentos* es la definición de los objetos en la base de datos que conduce directamente a un enfoque estándar para la creación de documentos de ingeniería del software.

El depósito se define en función de un metamodelo. Para lograr estas funciones el *metamodelo* determina cómo se guarda la información en el depósito, cómo se tiene acceso a los datos mediante las herramientas y cómo los visualizan los ingenieros de software, cuán bien se puede mantener la seguridad e integridad de los datos, y cuán fácilmente se puede ampliar el modelo existente para adecuar las nuevas necesidades. Para mayor información, el lector interesado debe consultar [SHA95] y [GRI95].

27.2.2 Características y contenido generales

Las características y el contenido del depósito se comprenden mejor si se les observa desde dos perspectivas: qué se guardará en el depósito y qué servicios específicos ofrece éste. En la figura 27.3 se presenta un análisis detallado de los tipos de representaciones, documentos y productos de trabajo que se guardan en el depósito.

FIGURA 27.3 Contenido del depósito

Referencia Web

Ejemplos de depósitos disponibles comercialmente o su propio depósito se pueden obtener en www.software.hp.com/products/SCMGR o en otra dirección como www.oracle.com/technology/documentation/repository.html.

Un depósito robusto proporciona dos clases diferentes de servicios: 1) los mismos tipos de servicios que se pueden esperar de cualquier sistema sofisticado de gestión de base de datos, y 2) servicios específicos del entorno de la ingeniería del software.

Un depósito que atienda a un equipo de ingeniería de software debe 1) integrarse con o directamente apoyar las funciones de gestión de proceso; 2) apoyar reglas específicas que rigen la función de GCS y los datos conservados dentro del depósito; 3) ofrecer una interfaz a otras herramientas de ingeniería del software; y 4) acomodar el almacenamiento de datos sofisticados (por ejemplo, texto, gráficos, video, audio).

27.2.3 Características de la GCS

El apoyo a la GCS requiere que el almacén o depósito tenga un conjunto de herramientas que ofrezca soporte para las siguientes características:

Versiones. Conforme un proyecto progrese se crearán muchas versiones (sección 27.3.2) de productos de trabajo individuales. El depósito debe ser capaz de guardar todas estas versiones para permitir la gestión eficaz de las liberaciones de producto y permitir que los desarrolladores regresen a versiones anteriores durante las pruebas y la depuración.

PUNTO CLAVE

El depósito debe ser capaz de mantener los ECS relacionados con muchas versiones diferentes del software. Más importante, debe ofrecer los mecanismos para ensamblar dichos ECS en una configuración específica de versión.

El depósito debe ser capaz de controlar una amplia variedad de tipos de objetos, incluso texto, gráficos, mapas de bits, documentos complejos y objeto únicos como definiciones de pantallas e informes, archivos de objeto, datos de prueba y resultados. Un depósito maduro sigue las versiones de los objetos con grados arbitrarios de granularidad; por ejemplo, se puede seguir una sola definición de datos o un conjunto de módulos.

Gestión del seguimiento de la dependencia y del cambio. El depósito gestiona una amplia variedad de relaciones entre los objetos de configuración que guarda. Se incluyen relaciones entre entidades y procesos empresariales, entre las partes de un diseño de aplicación, entre componentes de diseño y la arquitectura de información del proyecto, entre elementos de diseño y otros productos de trabajo, etcétera. Algunas de estas relaciones son meramente asociaciones, y algunas son dependencias o relaciones obligatorias.

La habilidad con que se da seguimiento a todas estas relaciones es crucial para la integridad de la información guardada en el depósito y la generación de productos de trabajo basados en ella, y es una de las aportaciones más importantes del concepto de depósito a la mejora del proceso de desarrollo de software. Por ejemplo, si se modifica un diagrama de clase UML, el depósito puede detectar si las clases relacionadas, las definiciones de interfaz y los componentes de código también requieren modificación y pueden colocar en la atención del desarrollador los ECS afectados.

Seguimiento de requisitos. Esta función especial ofrece la habilidad de seguir todos los componentes y entregables de diseño y construcción que resulten de una determinación específica de requisitos (seguimiento hacia adelante o seguimiento

propiamente dicho). Además, proporciona la habilidad de identificar qué requisitos generaron algún producto de trabajo dado (seguimiento hacia atrás o rastreo)

Gestión de la configuración. Una gestión de la configuración facilita la conservación del rastro de una serie de configuraciones que representan hitos específicos del proyecto o liberaciones de producción.

Rutas de auditoría. Una ruta de auditoría establece información adicional acerca de cuándo, por qué y por quién se hicieron los cambios. La información acerca de la fuente de los cambios se puede ingresar como atributos de objetos específicos en el depósito.

27.3 El proceso de GCS

El proceso de gestión de la configuración del software define una serie de tareas que tienen cuatro objetivos principales: 1) identificar todos los elementos que colectivamente definen la configuración del software; 2) gestionar los cambios a uno o más de dichos elementos; 3) facilitar la construcción de diferentes versiones de una aplicación; y 4) garantizar que la calidad del software se conserva conforme la configuración evoluciona a lo largo del tiempo

Un proceso que logra estos objetivos no necesita ser burocrático y molesto, pero sí debe caracterizarse en una forma que permita que un equipo de software desarrolle respuestas a un conjunto de preguntas complejas:

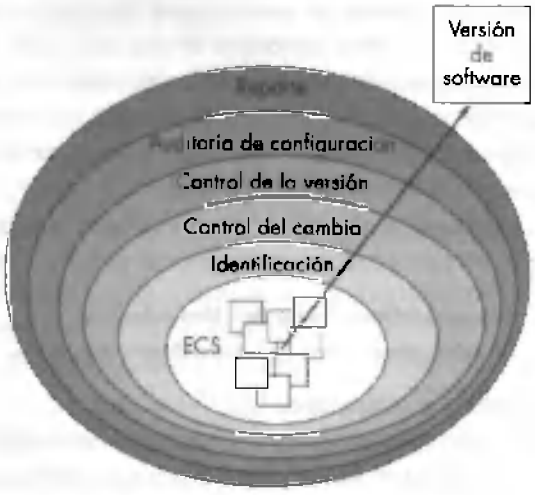
- ¿Cómo identifica un equipo de software los elementos discretos de una configuración de software?
- ¿Cómo gestiona una organización las numerosas versiones existentes de un programa (y su documentación) en una forma que permita que el cambio se acomode eficientemente?
- ¿Cómo controla una organización los cambios antes y después de que el software se libere al cliente?
- ¿Quién tiene la responsabilidad de aprobar y clasificar los cambios?
- ¿Cómo se garantiza que los cambios se hayan realizado adecuadamente?
- ¿Con qué mecanismo se valoran otros cambios que se realizan?

¿Para responder a qué preguntas se debe diseñar un proceso de GCS?

Estas preguntas conducen a la definición de las cinco tareas de la GCS ilustradas en la figura 27.4: identificación, control de la versión, control del cambio, auditoría de la configuración e informe.

En la misma figura las tareas de la GCS se aprecian como capas concéntricas. Los ECS fluyen hacia afuera a través de dichas capas a lo largo de su vida útil, y a final de cuentas se convierten en parte de la configuración del software de una o más versiones de una aplicación o sistema. Conforme un ECS se mueve a través de una capa, las acciones que implica cada capa de proceso de la GCS pueden o no aplicarse. Por ejemplo, cuando se crea un nuevo ECS debe ser identificado. Sin embargo, si

FIGURA 27.4
Capas del proceso de GCS.



no se solicitan cambios para el ECS, la capa de control de cambio no se aplica. El ECS se asigna a una versión específica del software (entran en juego mecanismos de control de la versión). Se conserva un registro del ECS (su nombre, fecha de creación, designación de versión, etc.) para propósitos de auditoría de la configuración e informes a quienes necesiten saberlo. En las secciones siguientes se examinan con más detalle cada una de estas capas del proceso de GCS.

27.3.1 Identificación de objetos en la configuración del software

El control y la gestión de elementos de configuración del software requiere nombrar cada uno por separado y luego organizarlo mediante un enfoque orientado a objetos. Es posible identificar dos tipos de objetos [CHO89]: básicos y agregados.[3] Un *objeto básico* es una unidad de información creada por un ingeniero de software durante el análisis, el diseño, el código o las pruebas. Por ejemplo, un objeto básico puede ser una sección de una especificación de requisitos, parte de un modelo de diseño, código fuente para un componente, o un conjunto de casos de prueba que se utilizan para ejercitar el código. Un *objeto agregado* es una colección de objetos básicos y otros objetos agregados. En la figura 27.2 **EspecificacionDiseño** es un objeto agregado. Conceptualmente, es posible verlo como una lista nombrada (identificada) de punteros que especifican objetos básicos como son **ModeloDatos** y **ComponenteN**.

3 El concepto de objeto agregado [GUS89] se ha propuesto como un mecanismo para representar una versión completa de una configuración de software.

Cada objeto tiene un conjunto de características distintivas que lo identifican de manera exclusiva: un nombre, una descripción, una lista de recursos, y una "realización". El nombre del objeto es una cadena de caracteres que identifican al objeto sin ambigüedades. La descripción del objeto es una lista de elementos de datos que identifican el tipo de ECS (por ejemplo, elemento modelo, programa, datos) que representa el objeto, un identificador de proyecto e información de cambio y/o versión

¡PUNTO CLAVE

Las interrelaciones establecidas para los objetos de configuración permiten que un ingeniero de software evalúe el impacto del cambio.

La identificación del objeto de configuración también puede considerar las relaciones entre los objetos nombrados. Por ejemplo, con la utilización de notación simple

DiagramadeClase <parte de> ModeloAnalisis;
ModeloAnalisis <parte de> EspecificacionRequisitos;

se crea una jerarquía de ECS.

CONSEJO

Incluso si la base de datos del proyecto ofrece la habilidad para establecer dichas relaciones, éstas consumen tiempo en su establecimiento y dificultan mantener la actualización. Aunque son muy útiles para el análisis de impacto, no son esenciales para la gestión global del cambio

En muchos casos, los objetos están interrelacionados a través de ramas de jerarquía de objetos. Dichas relaciones estructurales cruzadas se representan en la forma siguiente:

ModeloDatos <interrelacionado> ModeloFlujoDatos
ModeloDatos <interrelacionado> CasoPruebaClaseM

En el primer caso la interrelación es entre un objeto compuesto, mientras que la segunda relación es entre un objeto agregado (**ModeloDatos**) y un objeto básico (**CasoPruebaClaseM**).

El esquema de identificación para los objetos de configuración debe reconocer que los objetos evolucionan a lo largo del proceso de software. Antes de que un objeto se convierta en línea base puede cambiar muchas veces, e incluso después de establecida una línea base los cambios quizá sean muy frecuentes.

27.3.2 Control de la versión

El control de la versión combina procedimientos y herramientas para gestionar diferentes versiones de objetos de configuración que se crean durante el proceso del software. Un sistema de control de la versión implementa o está directamente integrado con cuatro grandes capacidades: 1) una base de datos del proyecto (depósito) que guarda todos los objetos de configuración relevantes; 2) una capacidad de *gestión de la versión* que almacena todas las versiones de un objeto de configuración (o permite que se construya cualquier versión empleando diferencias de versiones anteriores); 3) una *facilidad de hechura* que permita al ingeniero de software recopilar todos los objetos de configuración relevantes y construir una versión específica del software. Además, los sistemas de control de la versión y de control del cambio con frecuencia implementan una capacidad de *seguimiento de conflictos* (también llamada *seguimiento de bugs*) que permiten al equipo registrar y hacer el seguimiento del estado de todos los conflictos destacados que se asocian con cada objeto de configuración.

¡PUNTO CLAVE

Una facilidad "de hechura" permite a un ingeniero de software obtener todos los objetos de configuración relevantes y construir una versión específica del software

> "Cualquier cambio, incluso uno para mejorar, está acompañado con inconvenientes o incomodidades."
>
> **Arnold Bennett**

Varios sistemas de control de la versión establecen un *conjunto de cambio* —una colección de todos los cambios (con cierta configuración de línea base)— que requiere la creación de una versión específica del software. Dart [DAR91] advierte que un conjunto de cambios "captura todos los cambios de todos los archivos en la configuración junto con la razón para los cambios y detalles de quién los hizo y cuándo".

Es posible identificar varios conjuntos de cambio nombrados para una aplicación o sistema. Esto permite que un ingeniero de software construya una versión del software al especificar los conjuntos de cambio (por nombre) que se deben aplicar a la configuración de línea base. Esto se logra aplicando un enfoque de *modelado de sistema*. El modelo de sistema contiene 1) una *plantilla* que incluye una jerarquía de componentes y un "orden de construcción" para los componentes que describe cómo se debe construir el sistema, 2) reglas de construcción y 3) reglas de verificación.[4]

Durante las pasadas dos décadas se han propuesto varios enfoques automatizados para el control de la versión. La principal diferencia en los enfoques es la sofisticación de los atributos que se utilizan en la construcción de versiones específicas y variantes de un sistema y los mecanismos del proceso de construcción.

HERRAMIENTAS DE SOFTWARE

El Sistema de Versiones Concurrentes (SVC)

El empleo de herramientas con que lograr el control de la versión es esencial para una gestión del cambio eficaz. El *sistema de versiones concurrentes* (SVC; CVS, Concurrent Versions System) es una herramienta ampliamente empleada en el control de versiones. Originalmente diseñada para código fuente, pero útil para cualquier archivo basado en texto, el sistema SVC 1) establece un depósito simple, 2) conserva todas las versiones de un archivo en un archivo con un solo nombre al almacenar sólo las diferencias entre versiones progresivas del archivo original, y 3) protege un archivo contra cambios simultáneos al establecer diferentes directorios para cada desarrollador, con lo que se aíslan uno de otro. El SVC mezcla los cambios cuando cada desarrollador completa su trabajo.

Es importante notar que el SVC no es un sistema "de construcción"; esto es, no construye una versión específica del software. Esto se logra integrando al SVC otras herramientas (por ejemplo, *Makefile*). El SVC no implementa un proceso de control de cambio (por ejemplo, solicitudes de cambio, informes de cambio, seguimiento de bugs).

Pese a sus limitaciones, el SVC "es un sistema dominante en el control de versiones, transparente respecto a la red y de fuente abierta [que] es útil para todos, desde desarrolladores individuales hasta grandes equipos segmentados" [CVS02]. Su arquitectura cliente/servidor permite que los usuarios accedan a los archivos mediante conexiones de Internet y su filosofía de fuente abierta facilita su disponibilidad en la mayoría de las plataformas populares.

El SVC está disponible sin costo para entornos Windows, Macintosh y Unix. Visítese www.cvshome.org para mayores detalles.

4 También es posible consultar el modelo de sistema para valorar cómo un cambio en un componente impactará a otros componentes.

27.3.3 Control del cambio

La realidad del control del cambio en un contexto moderno de ingeniería del software la resumió bellamente James Bach [BAC98]:

> El control del cambio es vital. Pero las fuerzas que lo hacen necesario también lo tornan irritante. Nos preocupamos por los cambios porque una pequeña perturbación en el código puede crear una gran falla en el producto. Pero también puede resolver una gran falla o permitir maravillosas nuevas capacidades. Nos preocupamos por los cambios porque un solo desarrollador solitario podría hundir el proyecto; aunque en las mentes de dichos solitarios se originan ideas brillantes, y un proceso de control del cambio gravoso podría desalentarlos efectivamente de realizar trabajo creativo.

Bach reconoce que se enfrenta un acto de equilibrio. Demasiado control del cambio, y se crean problemas; poco, y se crean otros problemas.

> "El arte del progreso es preservar el orden entre el cambio, y preservar el cambio entre el orden."
> **Alfred North Whitehead**

Se debe destacar que varias solicitudes de cambio pueden combinarse para resultar en una sola OCI y que las OCI usualmente resultan en cambios a múltiples objetos de configuración.

En un gran proyecto de ingeniería de software el cambio incontrolado conduce rápidamente al caos. Respecto a tales proyectos el control del cambio combina procedimientos humanos y herramientas automatizadas. En la figura 27.5 se ilustra esquemáticamente el proceso de control del cambio. Se emite una *solicitud de cambio* y se estima para evaluar los méritos técnicos, los potenciales efectos colaterales, el impacto global sobre otros objetos de configuración y funciones del sistema, y el costo proyectado del cambio. Los resultados de la evaluación se presentan como un *informe de cambio*, que lo utiliza una *autoridad del control del cambio* (ACC): una persona o grupo que toman la decisión final acerca del estado y la prioridad del cambio. Se genera una *orden de cambio en la ingeniería* (OCI) para cada cambio aprobado. La OCI describe el cambio que se debe realizar, las restricciones insoslayables y los criterios de revisión y auditoría.

La confusión conduce a errores, algunos de ellos bastante serios. El control del acceso y de la sincronización evitan confusión. Empléense herramientas de control de la versión y del

El objeto que se cambiará se coloca en un directorio que controle exclusivamente el ingeniero de software que realiza el cambio. Un sistema de control de la versión (véase el recuadro acerca de SVC) actualiza el archivo original una vez realizado el cambio. Como alternativa, el objeto que se cambiará puede "salir" de la base de datos del proyecto (depósito), realizar el cambio y aplicar las actividades apropiadas de SQA. Luego el objeto "entra" a la base de datos y se aplican mecanismos adecuados de control de versión (sección 27.3.2) para crear la siguiente versión del software.

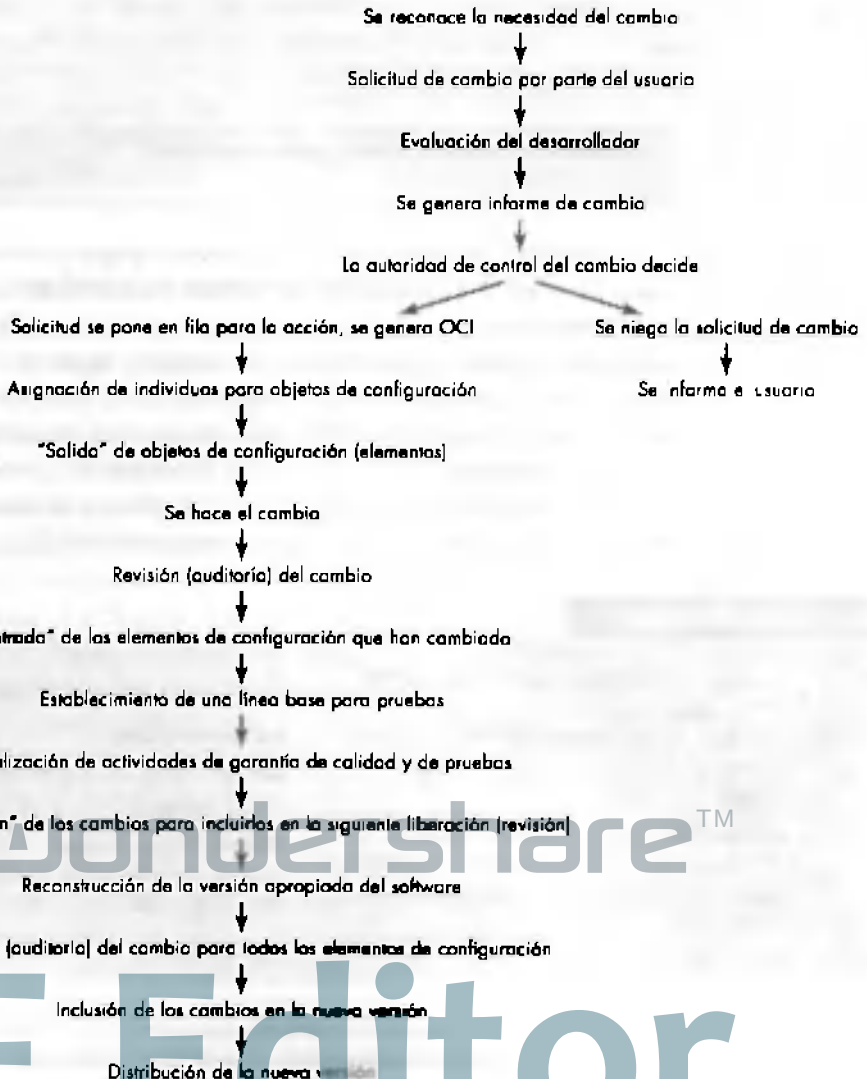
Estos mecanismos de control de la versión, integrados en el proceso de control de cambios, implementan dos importantes elementos de gestión del cambio: control del acceso y de la sincronización. El *control del acceso* rige qué ingenieros de software están autorizados para ingresar y modificar un objeto de configuración particular. El *control de la sincronización* ayuda a garantizar que los cambios paralelos, efectuados por dos personas diferentes, no se sobrescriben uno sobre otro[HAR89].

Algunos lectores quizá comiencen a sentirse incómodos con el grado de burocracia que implica la descripción del proceso de control del cambio mostrada en la figura 27.5. Este sentimiento es común. Sin las salvaguardas adecuadas el control del cambio puede retrasar el progreso y crear burocracia y papeleo innecesarios. La mayoría de los desarrolladores de software con mecanismos de control del cambio (desafortunadamente muchos no los tienen) ha creado varias capas de control para ayudarse a evitar los problemas mencionados aquí.

Antes de que un ECS se convierta en línea base sólo se necesita aplicar *control de cambio informal*. El desarrollador del objeto de configuración (ECS) en cuestión tiene

FIGURA 27.5

El proceso de control del cambio.



CONSEJO

Óptese por un poco más de control de cambio del que se crea que necesitará. Es probable que demasiado será la cantidad correcta

la posibilidad de realizar cualesquiera cambios justificados por el proyecto y los requisitos técnicos (en tanto los cambios no afecten requisitos de sistema mas amplios que se encuentran fuera del ámbito de trabajo del desarrollador). Una vez que el objeto haya experimentado una revisión técnica formal y haya sido aprobado se puede crear una línea base.[5] Una vez que un ECS se convierte en línea base se implementa un *control de cambio a nivel del proyecto*. Ahora, para realizar un cambio, el desarrollador debe obtener la aprobación del gestor del proyecto (si el cambio es "local") o de la ACC si el cambio afecta otros ECS. En algunos casos, la generación formal de las solicitudes de cambio, los informes de cambio y las OCI se distribuyen. Sin embargo, se lleva a cabo la evaluación de cada cambio, y todos los cambios se siguen y revisan.

Cuando el producto de software se libera entre los clientes se instituye el *control de cambio formal*. En la figura 27.5 se ha esbozado el procedimiento de control de cambio formal.

> "El cambio es inevitable, excepto para las máquinas expendedoras."
>
> **Calcomanía en un parachoques**

La autoridad de control del cambio juega un papel activo en la segunda y tercera capas del control. Dependiendo del tamaño y carácter de un proyecto de software la ACC puede estar compuesta de una persona (el gestor del proyecto) o varias personas (por ejemplo, representantes de software, hardware, ingeniería de bases de datos, soporte, mercadotecnia). El papel de la ACC es tener una visión global, esto es: evaluar el impacto del cambio más allá del ECS en cuestión. ¿Cómo afectará el cambio al hardware? ¿Cómo afectará al desempeño? ¿Cómo modificará la percepción del cliente acerca del producto? ¿Cómo afectará la calidad y fiabilidad del producto? Estas y muchas otras preguntas las aborda la ACC.

HOGARSEGURO

Problemas en la GCS

El escenario: Oficina de Doug Miller cuando comienza el proyecto del software *HogarSeguro*.

Los actores: Doug Miller (gerente del equipo de ingeniería del software de *HogarSeguro*) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería del software del producto.

La conversación:

Doug: Yo sé que es muy pronto, pero tenemos que hablar acerca de la gestión del cambio.

Vinod (ríe): Apenas Mercadotecnia llamó esta mañana con unos cuantos "segundos pensamientos". Nada importante, pero es sólo el comienzo.

5 Se puede crear una línea base también por otras razones. Por ejemplo, cuando se crean "construcciones diarias" todos los componentes que entran en un tiempo dado se convierten en la línea base para el trabajo del día siguiente.

Hemos sido bastante informales acerca de la del cambio en proyectos anteriores.

.o sé, pero éste es mayor y más visible, y según

(asiente con la cabeza): Fuimos asesinados incontrolados en el proyecto de control de la en el hogar . recuerda las demoras que .

frunce el ceño): Una pesadilla que prefiero no

Así que qué hacemos.

Como lo veo, tres cosas Primero, tenemos que ir, o pedir prestado, un proceso de control de

¿Te refieres a cómo la gente solicita los cambios?

Sí, pero también cómo evaluamos el cambio, cuándo hacerlo (si eso es lo que decidimos) y cómo conservamos los registros de lo que afecta el cambio.

Doug: Segundo, tenemos que obtener una herramienta de GCS realmente buena para control del cambio y de la versión

Jamie: Podemos construir una base de datos para todos nuestros productos de trabajo

Vinod: Se llaman ECS en este contexto, y la mayoría de las buenas herramientas ofrecen cierto soporte para eso

Doug: Ese es un buen comienzo, ahora tenemos que. .

Jamie: Oye, Doug: dijiste que eran tres cosas...

Doug (sonríe): Tercera: todos tenemos que comprometernos a seguir el proceso de gestión del cambio y usar las herramientas, sin importar cuáles sean, ¿de acuerdo?

27.3.4 Auditoría de la configuración

La identificación, el control de la versión y el control del cambio ayudan al desarrollador del software a mantener el orden en lo que de otro modo sería una situación caótica e inestable. Sin embargo, incluso el mecanismo de control más exitoso sólo sigue un cambio hasta que no se genera una OCI. ¿Cómo se puede garantizar que el cambio se ha implementado con propiedad? La respuesta es doble: 1) revisiones técnicas formales y 2) auditoría de la configuración del software

La revisión técnica formal (presentada con detalle en el capítulo 26) se enfoca en la corrección técnica del objeto de configuración que se ha modificado. Los revisores evalúan el ECS para determinar su consistencia con otros ECS, omisiones o potenciales efectos colaterales. Se debe realizar una revisión técnica formal en casi la mayoría de los cambios triviales.

Una *auditoría de configuración del software* complementa la revisión técnica formal al abordar las siguientes preguntas:

1. ¿Se ha realizado el cambio especificado en la OCI? ¿Se han incorporado modificaciones adicionales?
2. ¿Se ha realizado una revisión técnica formal para evaluar la corrección técnica?
3. ¿Se ha seguido el proceso de software? ¿Se han aplicado adecuadamente los estándares de ingeniería del software?
4. ¿El cambio se ha resaltado en el ECS? ¿Se han especificado la fecha y el autor del cambio? ¿Los atributos del objeto de configuración reflejan el cambio?

? ¿Cuáles son las que se durante auditoría de configuración?

5. ¿Se han seguido los procedimientos de GCS para destacar el cambio, registrarlo e informar de él?
6. ¿Todos los ECS relacionados se han actualizado de manera adecuada?

En algunos casos, las preguntas de la auditoría se plantean como parte de una revisión técnica formal. Sin embargo, cuando la GCS es una actividad formal, la auditoría de GCS la lleva a cabo por separado el grupo de aseguramiento de la calidad. Tales auditorías formales de configuración también aseguran que los ECS correctos (por versión) se han incorporado en una construcción específica y que toda la documentación está actualizada y es consistente con la versión que se ha construido.

27.3.5 Informe de estado

CONSEJO

Elabórese una lista de "necesito conocer" para cada objeto de configuración y consérvese actualizada. Cuando se realice un cambio, es necesario asegurarse de que todos los de la lista sean notificados.

El *informe de estado de la configuración* (a veces llamado *contabilidad de estado*) es una tarea de GCS que responde las siguientes preguntas: 1) ¿qué ocurrió? 2) ¿quién lo hizo? 3) ¿cuándo ocurrió? 4) ¿qué otra cosa será afectada?

En la figura 27.5 se muestra el flujo de información para el informe de estado de la configuración (IEC). Cada vez que se le asigna una identificación nueva o actualizada a un ECS se efectúa una entrada de IEC. Cada vez que la ACC aprueba un cambio (es decir, se expide una OCI) se genera una entrada en el IEC. Cada vez que se realiza una auditoría de la configuración los resultados se reportan como parte de la tarea de IEC. El resultado del IEC es posible colocarlo en una base de datos en línea o en un sitio Web, de modo que los desarrolladores y los encargados del mantenimiento del software pueden tener acceso a la información del cambio mediante categorías clave. Además, se genera un IEC con regularidad y su finalidad es mantener a los gestores y profesionales alertas ante de los cambios importantes

HERRAMIENTAS DE SOFTWARE

Soporte de la GCS

Objetivo: Las herramientas de GCS proporcionan soporte a una o más de las actividades del proceso estudiadas en la sección 27.3.

Mecánica: La mayoría de las modernas herramientas de GCS funciona en conjunto con un depósito (un sistema de base de datos) y ofrecen mecanismos para identificar, control de la versión y el cambio, auditoría e informe

Herramientas representativas[6]

CCC/Harvest, distribuida por Computer Associates (www.cai.com), es un sistema de GCS multiplataforma

ClearCase, desarrollada por Rational (www.rational.com), ofrece una familia de funciones de GCS.

Concurrent Versions System (SVC), una herramienta de fuente abierta (www.cvshome.org), es uno de los sistemas de control de versión más ampliamente empleados en la industria (véase un recuadro anterior)

PVCS, distribuida por Merant (www.merant.com), ofrece un conjunto completo de herramientas de GCS que son aplicables tanto en software convencional como en WebApps

6 Las herramientas mencionadas sólo representan una muestra de esta categoría. En la mayoría de los casos los nombres de las mismas son marcas registradas por sus respectivos desarrolladores

SourceForge, distribuida por VA Software (sourceforge.net), ofrece gestión de versión, capacidades de construcción, seguimiento de problemas/bugs y muchas otras características de gestión

SurroundSCM, desarrollada por Seapine Software (www.seapine.com), proporciona capacidades completas de gestión del cambio

Vesta, distribuida por Compaq (www.vestasys.org), es un sistema de GCS de dominio público que puede dar soporte tanto a proyectos pequeños (< 10 KLDC) como a grandes (10,000 KLDC)

Una extensa lista de herramientas comerciales y entornos para GCS, se puede encontrar en www.cmtoday.com/yp/commercial.html

27.4 Gestión de la configuración para ingeniería Web

En la parte 3 de este libro se estudió la naturaleza especial de las aplicaciones Web y el proceso de ingeniería Web necesario para construirlo. Entre las muchas características que diferencian a las WebApps del software convencional se encuentra la naturaleza ubicua del cambio.

La ingeniería Web utiliza un modelo de proceso incremental iterativo (capítulo 16) que aplica muchos principios derivados del desarrollo de software ágil (capítulo 4). Al utilizar este enfoque, un equipo de ingeniería con frecuencia desarrolla un incremento de WebApp en un periodo muy corto mediante un enfoque basado en el cliente. Los incrementos subsecuentes agregan contenido y funcionalidad adicionales, y tal vez cada uno implemente cambios que conduzcan a contenido aumentado, mejor facilidad de uso, estética mejorada, mejor navegación, desempeño aumentado y mayor seguridad. En consecuencia, en el mundo ágil de la ingeniería Web el cambio se ve de manera un poco diferente.

Los ingenieros Web deben adoptar el cambio, e incluso un típico equipo ágil evita todas las cosas que parecen pesados procesos, burocráticos y formales. Por lo general, se considera (aunque incorrectamente) que la gestión de la configuración del software posee estas características. Esta aparente contradicción se resuelve al no rechazar los principios, prácticas y herramientas de la GCS, sino más bien moldeándolas para satisfacer las necesidades especiales de los proyectos de ingeniería Web.

27.4.1 Problemas en la gestión de la configuración para WebApps

Conforme las WebApps se vuelven cada vez más importantes para la sobrevivencia y el crecimiento de los negocios, crece la necesidad de la gestión de la configuración. ¿Por qué? Porque sin controles eficaces los cambios inadecuados a una WebApp (recuérdese que la inmediatez y la evolución continua son los atributos dominantes de muchas WebApps) conducirían a la difusión no autorizada de información de un nuevo producto; funcionalidad errónea o pobremente probada que frustra a los visitantes a un sitio Web; hoyos en la seguridad que ponen en peligro los sistemas internos de la compañía; y otras consecuencias económicamente desagradables o incluso desastrosas.

¿Qué impacto tiene un cambio descontrolado sobre una WebApp?

Las estrategias generales para la gestión de configuración del software (GCS) descritas en este capítulo son aplicables, pero las tácticas y herramientas se deben adaptar para que concuerden con la naturaleza única de las WebApps. Se deben considerar cuatro temas cuando se desarrollen tácticas para la gestión de la configuración de la WebApp: contenido, personal, escalabilidad y políticas.

Contenido. Una WebApp típica contiene una amplia variedad de contenido: texto gráficos, applets, guiones, archivos de audio/video, formatos, elementos de página activos, tablas, datos clasificados por niveles y muchos otros. El reto es organizar este océano de contenido en un conjunto racional de objetos de configuración (sección 27.1.4) y luego establecer mecanismos de control de configuración adecuados para dichos objetos.

Personal. Puesto que un porcentaje significativo del desarrollo de la WebApp continúa realizándose en una forma *ad hoc*, cualquier persona involucrada en la WebApp puede (y con frecuencia lo hace) crear contenido. Muchos creadores de contenido no tienen conocimientos de ingeniería del software e ignoran por completo la importancia de la gestión de la configuración. Por lo tanto, la aplicación crece y cambia en una forma descontrolada.

Escalabilidad. Las técnicas y los controles aplicados a una WebApp pequeña no se escalan bien hacia arriba. No es inusual que una WebApp simple crezca significativamente conforme se implementan interconexiones con los sistemas de información existentes, bases de datos, depósitos de datos y portales. Conforme crecen el tamaño y la complejidad, los cambios pequeños pueden tener efectos de largo alcance e imprevistos que pueden ser problemáticos. En consecuencia, el rigor de los mecanismos de control de la configuración debe ser directamente proporcionales a la escala de la aplicación.

Políticas. ¿Quién "posee" una WebApp? Esta pregunta se plantea en compañías grandes y pequeñas, y su respuesta tiene un impacto significativo en las actividades de gestión y control asociadas con la IWeb. En ciertas instancias, los desarrolladores Web se ubican fuera de la organización TI, lo que crea potenciales dificultades de comunicación. Dart [DAR99] sugiere las siguientes preguntas para ayudar a entender las políticas asociadas con la IWeb:

¿Cómo se determina quién tiene la responsabilidad de la GC de la WebApp?

- ¿Quién asume la responsabilidad de la precisión de la información en el sitio Web?
- ¿Quién asegura que se han seguido los procesos de control de calidad antes de que la información se publique en el sitio?
- ¿Quién es el responsable de realizar los cambios?
- ¿Quién asume el costo del cambio?

Las respuestas a estas preguntas ayudan a determinar a las personas que, dentro de una organización, deben adoptar un proceso de gestión de la configuración para las WebApps.

27.4.2 Objetos de configuración WebApp

Las WebApps abarcan una amplia gama de objetos de configuración: objetos de contenido (por ejemplo, texto, gráficos, imágenes, video, audio), componentes funcionales (por ejemplo, guiones, applets) y objetos de interfaz (por ejemplo, COM o CORBA). Los objetos WebApp se pueden identificar (asignándoles nombres de archivo) en cualquier forma que sea apropiada para la organización. Sin embargo, se recomiendan las siguientes convenciones para garantizar que se conserva la compatibilidad entre plataformas cruzadas: los nombres de archivo deben estar limitados a 32 caracteres de longitud, se deben evitar los nombres con mayúsculas mezcladas o todas mayúsculas, así como el uso de subrayados. Además, las referencias URL (vínculos) dentro de un objeto de configuración siempre deben usar trayectorias relativas (por ejemplo, .. /productos/sensoresdealarma.html).

Todo el contenido de la WebApp tiene formato y estructura. Los formatos internos de archivo los dicta el entorno de cómputo en el que se almacena el contenido. Sin embargo, el *formato de representación* (usualmente llamado *formato de despliegue*) se define con el etilo estético y las reglas de diseño establecidas para la WebApp. La *estructura del contenido* define una arquitectura de contenido; esto es: define la forma en la que los objetos de contenido se ensamblan para presentar información significativa a un usuario final. Boiko [BOI02] define la estructura como "mapas que usted tiende sobre un conjunto de trozos [objetos] de contenido para organizarlos y hacerlos accesibles a las personas que los necesitan".

27.4.3 Gestión del contenido

La *gestión del contenido* se relaciona con la gestión de la configuración en el sentido en que un sistema de gestión del contenido (SGC) establece un proceso (apoyado por herramientas) que adquiere contenido existente (de un amplio ordenamiento de objetos de configuración de la WebApp), los estructura en una forma que permite presentarlos a un usuario final y luego los ofrece al entorno del lado del cliente para su despliegue

> "La gestión del contenido es un antídoto para el frenesí informativo de la actualidad."
>
> **Bob Boiko**

El uso más común del sistema de gestión del contenido ocurre cuando se construye una WebApp dinámica. Este tipo de WebApp crea páginas Web "al vuelo". Es decir, usualmente el usuario consulta la WebApp solicitando información específica. La WebApp consulta una base de datos, formatea la información en concordancia y la presenta al usuario. Por ejemplo, una compañía musical ofrece una librería de CD en venta. Cuando un usuario solicita un CD o su equivalente en música electrónica, se consulta una base de datos y una variedad de información acerca del artista, el CD (por ejemplo, su portada o gráfica), el contenido musical y muestras de audio se descargan y configuran en una plantilla de contenido estándar. La página Web resul-

PUNTO CLAVE

El subsistema de colección abarca todas las acciones que se requieren para crear, adquirir o convertir el contenido en una forma que se pueda presentar en el lado del cliente.

tante se construye en el lado del servidor y pasa al navegador del lado del cliente para que la examine el usuario final. En la figura 27.6 se muestra una representación genérica de esto

En el sentido más general, un SGC "configura" el contenido para el usuario final al invocar tres subsistemas integrados: de colección, de gestión y de publicación [BOI02].

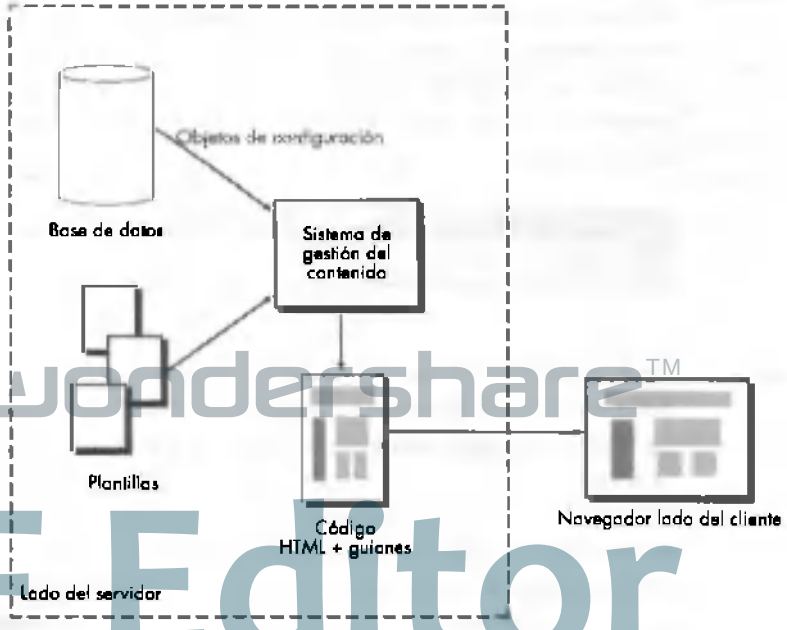
El subsistema de colección. El contenido procede de los datos y la información que debe crear o adquirir un desarrollador de contenido. El *subsistema de colección* abarca todas las acciones que se requieren para crear y/o adquirir contenido, así como las funciones técnicas necesarias para 1) convertir el contenido en una formato que se pueda representar en un lenguaje de marcas (por ejemplo, HTML, XML) y 2) organizar el contenido en paquetes que se puedan desplegar con eficacia en el lado del cliente.

El subsistema de gestión. Una vez que el contenido existe debe guardarse en un depósito, catalogarse para adquisición y uso subsecuentes, y etiquetarse para definir 1) su estado actual (por ejemplo, el objeto de contenido está completo o en desarrollo), 2) la versión apropiada del objeto de contenido, y 3) los objetos de contenido relacionados. Por lo tanto, el *subsistema de gestión* implementa un depósito que abarca los siguientes elementos:

- *Base de datos de contenido:* la estructura de información que se ha establecido para almacenar todos los objetos de contenido.

FIGURA 27.6

Sistema de gestión del contenido (SGC).



PUNTO CLAVE
...subsistema de ... implementa un ... para todo el ... La gestión ... configuración se ... cabo dentro de ... subsistema.

- *Capacidades de la base de datos.* funciones que permiten al SGC buscar objetos de contenido específicos (o categorías de objetos), almacenar y recuperar los objetos, y gestionar la estructura de archivos que se ha establecido para el contenido.
- *Funciones de gestión de la configuración:* los elementos funcionales y flujo de trabajo asociado que soportan la identificación del objeto de contenido, control de la versión, gestión del cambio, auditoría del cambio y creación de informes.

Además de estos elementos, el subsistema de gestión implementa una función de administración que abarca los metadatos y reglas que controlan la estructura global del contenido y la forma en la que recibe soporte.

El subsistema de publicación. El contenido se debe extraer del depósito, convertirse en una forma que esté dispuesta para la publicación y formatearse de modo que sea posible transmitirlo a los navegadores del lado del cliente. El *subsistema de publicación* logra estas tareas mediante una serie de plantillas. Cada *plantilla* es una función que construye una publicación empleando uno de tres componentes diferentes [BOI02]:

PUNTO CLAVE
...subsistema de ... obtiene ... del depósito ... entrega a los ... navegadores en el lado ... servidor.

- *Elementos estáticos:* los textos, gráficos, medios audiovisuales y guiones que ya no requieren procesamiento ulterior se transmiten directamente al lado del cliente.
- *Servicios de publicación:* función que solicita servicios específicos de recuperación y formateo que personalizan el contenido (mediante reglas predefinidas), efectúan conversión de datos y construyen vínculos de navegación apropiados.
- *Servicios externos:* proporcionan acceso a infraestructura de información corporativa externa como datos de la empresa o aplicaciones de "cuarto trasero".

Un sistema de gestión de contenido que abarque cada uno de estos subsistemas es aplicable a grandes proyectos de ingeniería Web. Sin embargo, la filosofía básica y la funcionalidad asociados con un SGC son aplicables a todas las WebApps dinámicas.

HERRAMIENTAS DE SOFTWARE

Gestión del contenido

Objetivo: Auxiliar a los ingenieros de software y desarrolladores de contenido a gestionar el contenido que se incorpora en las WebApps.

Mecánica: Las herramientas en esta categoría permiten que los ingenieros Web y proveedores de contenido actualicen el contenido de una WebApp en una forma controlada. La mayoría establece un simple sistema de gestión de archivos que asigna actualización página por página y permisos de edición para varios tipos de contenido WebApp. Algunos mantienen un sistema de versiones de modo que se pueden lograr versiones previas de contenido para propósitos históricos.

Herramientas representativas[7]

Content Management Tools Suite, desarrollada por interactivetools.com (www.interactivetools.com/), es un conjunto de herramientas de gestión de contenido que se enfoca en la gestión del contenido para dominios de aplicación específicos (por ejemplo, artículos nuevos, avisos clasificados, bienes raíces).

ektron-CMS300, desarrollada por ektron (www.ektron.com), es un conjunto de herramientas que ofrece capacidades de gestión de contenido, así como herramientas de desarrollo Web.

OmniUpdate, desarrollada por WebsiteASP, Inc. (www.omniupdate.com), es una herramienta que permite a los proveedores de contenido autorizados desarrollar actualizaciones controladas de contenido WebApp específicado.

Tower IDM, desarrollada por Tower Technologies (www.towertech.com), es un sistema de procesamiento de documentos y depósito de contenido para gestionar todas las formas de información comercial no estructurada: imágenes, formatos, informes generados por computadora, cuentas y facturas, documentos oficiales, correo electrónico y contenido Web.

En los siguientes sitios Web se puede encontrar información adicional acerca de la GCS y las herramientas de gestión del contenido para ingeniería Web:

Web Developer's Virtual Encyclopedia (www.wdlv.com), *WebDeveloper* (www.webdeveloper.com), *Developer Shed* (www.devshed.com), *webknowhow.net* (www.webknowhow.net) o *WebReference* (www.webreference.com).

27.4.4 Gestión del cambio

El flujo de trabajo asociado con el control del cambio para software convencional (sección 27.3.3) generalmente es demasiado laborioso para la ingeniería Web. Es improbable que se logre la secuencia petición de cambios, informe de cambio y orden de cambio de ingeniería en una forma ágil y aceptable para la mayoría de los proyectos de desarrollo WebApp. Entonces, ¿cómo se gestiona una corriente continua de cambios solicitada para el contenido y la funcionalidad de la WebApp?

La implementación de una gestión de cambio eficaz dentro de la filosofía "codifica y ve" que continúe dominando el desarrollo de las WebApps requiere modificar el proceso de control de cambios convencional. Cada cambio se debe clasificar en una de cuatro clases:

Clase 1: un cambio de contenido o función que corrija un error o mejore el contenido o funcionalidad locales.

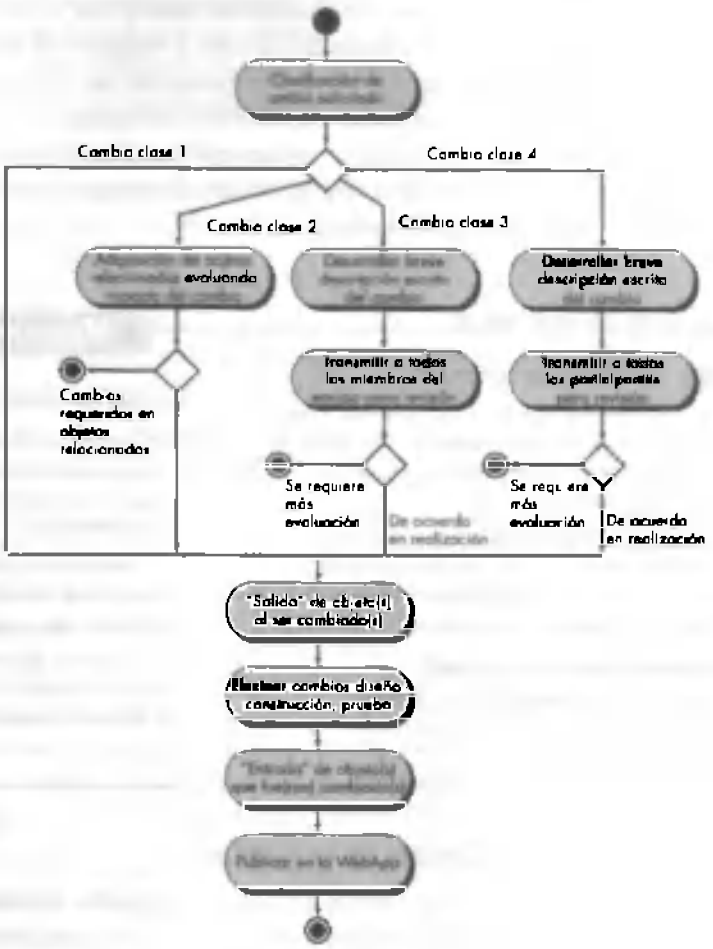
Clase 2: un cambio de contenido o función que tenga impacto sobre otros objetos de contenido o componentes funcionales.

Clase 3: un cambio de contenido o función que tenga amplio impacto a través de una WebApp (por ejemplo, gran ampliación de la funcionalidad, mejora significativa o reducción del contenido, grandes cambios requeridos en la navegación).

Clase 4: un gran cambio de diseño (por ejemplo, un cambio en el diseño de la interfaz o enfoque de navegación) que inmediatamente apreciarán una o más categorías de usuarios.

7 Las herramientas expuestas sólo representan una muestra de esta categoría. En la mayoría de los casos los nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

FIGURA 27.7
Gestión de cambios para WebApps.



Una vez clasificados los cambios solicitados se pueden procesar de acuerdo con el algoritmo mostrado en la figura 27.7

En la misma figura, los cambios de las clases 1 y 2 se tratan de manera informal y se manejan en una forma ágil. En un cambio de clase 1 el ingeniero Web evalúa el impacto del cambio, pero no se requiere revisión o documentación externa. Conforme se realiza el cambio, los procedimientos estándar de entrada y salida se refuerzan mediante las herramientas de configuración de depósito. En cuanto a los cambios de la clase 2, es obligación del ingeniero Web revisar el impacto del cambio sobre objetos relacionados (o pedir a los desarrolladores responsables de éstos que lo hagan). Si el cambio es factible sin que se requieran cambios significativos en otros objetos, la modificación ocurre sin revisión o documentación adicional. Si se requieren cambios sustantivos, son necesarias evaluación y planificación ulteriores.

Los cambios de las clases 3 y 4 también se tratan en una forma ágil, pero se requieren alguna documentación descriptiva y más procedimientos de revisiones formales. Los cambios de la clase 3 requieren el desarrollo de una *descripción de cambio* que ofrezca una breve evaluación del impacto del cambio. La descripción se distribuye entre todos los miembros del equipo de ingeniería Web, quienes lo revisan para evaluar mejor su impacto. También respecto a los cambios de la clase 4 se desarrolla una descripción del cambio, pero en este caso la revisión la llevan a cabo todos los participantes

HERRAMIENTAS DE SOFTWARE

Gestión del cambio

Objetivo: Auxiliar a los ingenieros Web y desarrolladores de contenido a gestionar los cambios conforme se realizan éstos en objetos de configuración WebApp

Mecánica: Las herramientas en esta categoría originalmente fueron desarrolladas para software convencional, pero es posible adaptarlas a la ingeniería Web y realizar cambios controlados en las WebApps

Herramientas representativas[8]

ChangeMan WCM, desarrollada por Serena (www.serena.com), es una de un conjunto de herramientas de gestión del cambio que ofrece capacidades de GCS.

ClearCase, desarrollada por Rational (www.rational.com), es un conjunto de herramientas que ofrecen capacidades completas de gestión de configuración para WebApps.

PVCS, desarrollada por Merant (www.merant.com), es un conjunto de herramientas que ofrecen capacidades completas de gestión de configuración para WebApps.

Source Integrity, desarrollada por mks (www.mks.com), es una herramienta GCS que se puede integrar con entornos de desarrollo seleccionados.

27.4.5 Control de la versión

Conforme una WebApp evoluciona por medio de una serie de incrementos, es posible que existan al mismo tiempo varias versiones diferentes. Una versión (la WebApp operativa actual) está disponible para los usuarios finales por Internet; otra versión (el siguiente incremento de la WebApp) quizá esté en las etapas finales de prueba previas al lanzamiento; una tercera versión está en desarrollo y representa una gran actualización en contenido, estética de interfaz y funcionalidad. Los objetos de configuración deben estar claramente definidos, de modo que cada uno pueda estar asociado con la versión adecuada. Además, deben estar establecidos los mecanismos de control. Dreilinger [DRE99] aborda la importancia del control de la versión (y del cambio) cuando escribe:

> En un sitio *descontrolado*, donde múltiples autores tienen acceso para editar y contribuir, surge el potencial para el conflicto y los problemas, más aún si los autores trabajan desde

8 Las herramientas expuestas sólo representan una muestra de esta categoría. En la mayoría de los casos los nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

diferentes oficinas en distintos momentos del día y la noche. Usted puede pasar el día mejorando el archivo *index.html* para un cliente. Después que ha realizado los cambios, otro desarrollador, quien trabaja en su casa después de su jornada laboral, o en otra oficina, puede pasar la noche cargando su propia novedosa versión revisada del archivo *index.html*, ¡y sobrescribir por completo el trabajo que usted ha realizado, sin forma alguna de retroceder!

Esta situación debe sonar familiar a todos los ingenieros de software, así como a todos los ingenieros Web. Para evitarlo, se debe establecer un proceso de control de la versión.

1. *Se debe establecer un depósito central para el proyecto WebApp.* El depósito contendrá las versiones actuales de todos los objetos de configuración de la WebApp (contenido, componentes funcionales y otros).
2. *Cada ingeniero Web crea su propia carpeta de trabajo.* La carpeta contiene aquellos objetos creados o cambiados en algún momento dado.
3. *Los relojes en las estaciones de trabajo de todos los desarrolladores deben estar sincronizados.* Esto tiene como fin evitar los conflictos de sobrescritura cuando dos desarrolladores realizan actualizaciones que están muy cercanas en el tiempo una de otra.
4. *Conforme se desarrollan nuevos objetos de configuración o se cambian los objetos existentes se importan al depósito central.* La herramienta de control de la versión (véase el estudio precedente del SVC en este capítulo) gestionará todas las funciones de entrada y salida de las carpetas de trabajo de cada ingeniero Web.
5. *Conforme los objetos se importan al o exportan del depósito se elabora un mensaje automático de registro cronometrado.* Esto ofrece información útil para auditoría y se puede convertir en parte de un efectivo esquema de elaboración de informes.

La herramienta de control de la versión mantiene diferentes versiones de la WebApp y puede revertirse a una versión más antigua si se requiere.

27.4.6 Auditoría y elaboración de informes

En búsqueda de agilidad, las funciones de auditoría y elaboración de informes no se resaltan en el trabajo de ingeniería Web. Sin embargo, no se eliminan por completo. Todos los objetos que entran o salen del depósito se anotan en un registro que puede revisarse en cualquier punto en el tiempo. Es factible crear un informe de registro completo de modo que todos los miembros del equipo de ingeniería Web tengan una cronología de los cambios sobre un periodo definido. Además, se envía una notificación automática de correo electrónico (dirigida a los desarrolladores y participantes interesados) cada vez que un objeto entre o salga del depósito.

INFORMACIÓN

Estándares de GCS

La siguiente lista de estándares GCS (procedente en parte de www.12207.com) es razonablemente extensa:

Estándares IEEE	**standards.ieee.org/catalog/olis**
IEEE 828	Planes de gestión de configuración del software
IEEE 1042	Gestión de configuración del software
Estándares ISO	**www.iso.ch/iso/en/ISOOnline.frontpage**
ISO 10007-1995	Gestión de calidad, lineamientos para GC
ISO/IEC 12207	Tecnología de información; procesos de ciclo de vida de software
ISO/IEC TR 15271	Guía para ISO/IEC 12207
ISO/IEC TR 15846	Ingeniería del software; procesos de ciclo de vida de software; gestión de configuración para software
Estándares EIA	**www.eia.org/**
EIA 649	Estándar de consenso nacional para gestión de configuración
EIA CMB4-1A	Definiciones de gestión de la configuración para programas de computadoras digitales
EIA CMB4-2	Identificación de configuración para programas de computadoras digitales
EIA CMB4-3	Librerías de software de computadora
EIA CMB4-4	Control de cambio de configuración para programas de computadoras digitales
EIA CMB6-1C	Referencias de gestión de configuración y datos
EIA CMB6-3	Identificación de configuración
EIA CMB6-4	Control de configuración
EIA CMB6-5	Libro de texto para contabilidad del estado de configuración
EIA CMB7-1	Intercambio electrónico de datos de gestión de configuración
Estándares militares estadounidenses	**www-library.itsi.disa.mil**
DoD MIL STD-973	Gestión de configuración
MIL-HDBK-61	Guía para gestión de configuración
Otros estándares	
DO-178B	Directrices para el desarrollo de software de aviación
ESA PSS-05-09	Guía para gestión de configuración de software
AECL CE-1001-STD rev. 1	Estándar para ingeniería del software de software de seguridad crítico
DOE SCM lista de verificación	cio.doe.gov/ITReform/sqse/download/cmcklst.doc
BS-6488	British Std., gestión de configuración de sistemas basados en computadora
Mejores prácticas-RU	Oficina de comercio gubernamental www.ogc.gov.uk
CMII	Instituto de mejores prácticas en GC: www.icmhq.com

Una *Guía de recursos de gestión de configuración* ofrece información complementaria para aquellos interesados en procesos y prácticas de GC. Está disponible en www.quality.org/config/cm-guide.html

27.5 RESUMEN

La gestión de configuración del software es una actividad protectora que se aplica a lo largo del proceso de software. La GCS identifica, controla, audita e informa modificaciones que invariablemente ocurren mientras se desarrolla software y después de

que se le libera a un cliente. Toda la información producida como parte de la ingeniería del software se vuelve parte de una configuración de software. La configuración está organizada de forma que permite la gestión ordenada del cambio.

La configuración del software está compuesta de un conjunto de objetos interrelacionados, también llamados elementos de configuración del software, que se producen debido a alguna actividad de ingeniería del software. Además de documentos, programas y datos, el entorno de desarrollo que se utiliza para crear software también se puede colocar bajo control de configuración. Todos los ECS se guardan en un depósito que implementa mecanismos y estructuras de datos para garantizar la integridad de los datos, ofrece soporte de integración para otras herramientas de software, apoya la distribución de información entre los miembros del equipo de software e implementa funciones en apoyo del control de la versión y del cambio.

Una vez desarrollado y revisado un objeto de configuración se convierte en línea base. Los cambios a un objeto convertido en línea base generan la creación de una nueva versión de dicho objeto. La evolución de un programa puede seguirse al examinar la historia de revisión de todos los objetos de configuración. Los objetos básicos y compuestos forman una piscina de objetos a partir de la que se crean las versiones. El control de la versión es el conjunto de procedimientos y herramientas para gestionar el empleo de dichos objetos.

El control de cambios es una actividad de procedimiento que asegura la calidad y la consistencia conforme los cambios se realizan en un objeto de configuración. El proceso de control de cambios comienza con una petición de cambio, conduce a una decisión para aceptar o rechazarla y culmina con una actualización controlada del ECS que se cambiará.

La auditoría de la configuración es una actividad de SQA que ayuda a garantizar que la calidad se conserva conforme se realizan los cambios. Los informes de estado ofrecen información acerca de cada cambio a quienes tengan necesidad de conocerla.

La gestión de la configuración para ingeniería Web es similar en muchos aspectos a la GCS para el software convencional. Sin embargo, cada una de las tareas principales de GCS se deben destacar para hacerlas tan simples como sea posible, y se deben implementar provisiones especiales para la gestión del contenido.

Referencias

[BAB86] Babich, W. A., *Software Configuration Management*, Addison-Wesley, 1986.

[BAC98] Bach, J., "The Highs and Lows of Change Control", en *Computer*, vol. 31, núm. 8, agosto de 1988, pp. 113-115.

[BER80] Bersoff, E. H., V. D. Henderson y S. G. Siegel, *Software Configuration Management*, Prentice-Hall, 1980.

[BOI02] Boiko, B., *Content Management Bible*, Hungry Minds Publishing, 2002.

[CHO89] Choi, S. C., y W. Scacchi, "Assuring the Correctness of a Configured Software Description", *Procs. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, octubre de 1989, pp. 66-75.

[CVS02] Concurrent Versions System Web site, www.cvshome.org, 2002.
[DAR91] Dart, S., "Concepts in Configuration Management Systems", *Proc. Third International Workshop on Software Configuration Management*, ACM SIGSOFT, 1991, se puede descargar de:http://www.sei.cmu.edu/legacy/scm/abstracts/abscm_concepts.html
[DAR99] Dart, S., "Change Management: Containing the Web Crisis", *Proc. Software Configuration Management Symposium*, Toulouse, Francia, 1999, disponible en http://www.perforce.com/perforce/conf99/dart.html.
[DAR01] Dart, S., *Spectrum of Functionality in Configuration Management Systems*, Software Engineering Institute, 2001, disponible en http://www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/TOC_TR11_90.html.
[DRE99] Dreilinger, S., "CVS Version Control for Web Site Projects", 1999, disponible en http://www.durak.org/cvswebsites/howto-cvs/howto-cvs.html
[FOR89] Forte, G., "Rally Round the Repository", en *CASE Outlook*, diciembre de 1989, pp. 5-27
[GRI95] Griffen, J., "Repositories: Data Dictionary Descendant Can Extend Legacy Code Investment", en *Application Development Trends*, abril de 1995, pp. 65-71
[GUS89] Gustavsson, A., "Maintaining the Evolution of Software Objects in an Integrated Environment", *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, octubre de 1989, pp. 114-117.
[HAR89] Harter, R., "Configuration Management", en *HP Professional*, vol. 3, núm. 6, junio de 1989
[IEE94] *Software Engineering Standards*, edición 1994, IEEE Computer Society, 1994.
[JAC02] Jacobson, I., "A resounding 'Yes' to Agile Processes-But Also More", en *Cutter IT Journal*, vol. 15, núm. 1, enero de 2002, pp. 18-24
[REI89] Reichenberger, C., "Orthogonal Version Management", *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, octubre de 1989, pp. 137-140
[SHA95] Sharon, D., y R. Bell, "Tools That Bind: Creating Integrated Environments", *IEEE Software*, marzo de 1995, pp. 76-85
[TAY85] Taylor, B., "A Database Approach to Configuration Management for Large Projects", *Proc. Conf. Software Maintenance-1985*, IEEE, noviembre de 1985, pp. 15-23.

PROBLEMAS Y PUNTOS A CONSIDERAR

27.1. ¿Por qué es cierta la primera ley de la ingeniería de sistemas? Ofrecer ejemplos específicos de cada una de las cuatro razones fundamentales para el cambio

27.2. ¿Cuáles son los cuatro elementos que existen cuando se implementa un sistema eficaz de GCS? Comentar cada uno brevemente

27.3. Con palabras propias, coméntense las razones para las líneas base

27.4. Suponga que usted es el gestor de un pequeño proyecto. ¿Qué líneas base definiría para el proyecto y cómo las controlaría?

27.5. Emplear UML agregados o compuestos (capítulo 8) para describir las interrelaciones entre los ECS (objetos de configuración) en la lista de la sección 27.1.4.

27.6. Diseñar un sistema de base de datos (depósito) de proyecto que le permitiría a un ingeniero de software almacenar, realizar referencias cruzadas, rastrear, actualizar y cambiar todos los elementos importantes de configuración de software. ¿Cómo manejaría la base de datos las diferentes versiones del mismo programa? ¿El código fuente se manejaría de manera diferente a la documentación? ¿Cómo se evitaría que dos desarrolladores realizasen diferentes cambios al mismo ECS en forma simultánea?

27.7. Investigar una herramienta existente de GCS y describir cómo implementa el control de las versiones y los objetos de configuración en general.

27.8. Las relaciones <parte de> e <interrelacionado> representan relaciones simples entre objetos de configuración. Describir cinco relaciones adicionales que puedan ser útiles en el contexto de un depósito de GCS.

27.9. Investigar una herramienta existente de GCS y describir cómo implementa los mecanismos de control de versiones. De manera alternativa, leer dos o tres de los artículos acerca de GCS y describir las diferentes estructuras de datos y mecanismos de referencia que se emplean para el control de las versiones.

27.10. Con la figura 27.5 como guía, desarrollar un análisis todavía más detallado para el control del cambio. Describir el papel de la ACC y sugerir formatos para la petición del cambio, el informe del cambio y la OCI.

27.11. Desarrollar una lista de verificación para emplearla durante las auditorías de la configuración.

27.12. ¿Cuál es la diferencia entre una auditoría GCS y una revisión técnica formal? ¿Sus funciones se pueden juntar en una revisión? ¿Cuáles son los pros y los contras?

27.13. Describir brevemente las diferencias entre GCS para el software convencional y la GCS para WebApps.

27.14. ¿Qué es la gestión del contenido? Empléese la Web para investigar las características de una herramienta de gestión del contenido y ofrézcase un breve resumen.

Otras lecturas y fuentes de información

Lyon (*Practical CM*, Raven Publishing, 2003, disponible en www.configuration.org) ha escrito una guía detallada para profesionales de GC que incluye directrices pragmáticas para implementar cada aspecto de un sistema de gestión de la configuración (actualizado anualmente). Hass (*Configuration Management: Principles and Practice*, Addison-Wesley, 2002) y Leon (*A Guide to Software Configuration Management*, Artech House, 2000) ofrecen exámenes útiles del tema. White y Clemm (*Software Configuration Management Strategies and Rational ClearCase*, Addison-Wesley, 2000) presentan la GCS dentro del contexto de una de las más populares herramientas de GCS.

Mikkelsen y Pherigo (*Practical Software Configuration Management: The Latenight Developer's Handbook*, Allyn & Bacon, 1997) y Compton y Callahan (*Configuration Management for Software*, VanNostrand-Reinhold, 1994) ofrecen guías pragmáticas acerca de importantes prácticas de GCS. Ben-Menachem (*Software Configuration Management Guidebook*, McGraw-Hill, 1994) y Ayer y Patrinnostro (*Software Configuration Management*, McGraw-Hill, 1992) presentan buenos panoramas para quienes necesitan mayor introducción a la materia. Berlack (*Software Configuration Management*, Wiley, 1992) presenta un examen útil de conceptos de GCS, donde resalta la importancia del depósito y las herramientas en la gestión del cambio. Babich [BAB86] proporciona un tratamiento abreviado, aunque eficaz, de temas pragmáticos en la gestión de configuración del software. Arnold y Bohner (*Software Change Impact Analysis*, IEEE Computer Society Press, 1996) han editado una antología que estudia cómo analizar el impacto del cambio al interior de sistemas complejos basados en software.

Berczuk y Appleton (*Software Configuration Management Patterns*, Addison-Wesley, 2002) presentan una diversidad de patrones útiles que auxilian en la comprensión de la GCS y la implementación de sistemas de GCS eficaces. Brown *et al* (*Anti-Patterns and Patterns in Software Configuration Management*, Wiley, 1999) estudian las cosas que no se hacen (antipatrones) cuando se implementa un proceso de GCS y luego consideran sus remedios.

Buckley (*Implementing Configuration Management*, IEEE Computer Society Press, 1993) considera enfoques de gestión de la configuración para todos los elementos del sistema —hardware, software y firmware— con estudios detallados de las principales actividades de GC. Rawlings (*SCM for Network Development Environments*, McGraw-Hill, 1994) considera el impacto de la GCS para desarrollo de software en un entorno de red. Bays (*Software Release Methodology*, Prentice-Hall, 1999) presenta una colección de mejores prácticas para todas las actividades que ocurren después de que se realizan cambios en una aplicación.

Conforme las WebApps se vuelven más dinámicas, la gestión del contenido se vuelve un tópico esencial para los ingenieros Web. Los libros de Addey y sus colegas (*Content Manage-*

ment Systems, Glasshaus, 2003), Boiko [BOI02], Hackos (*Content Management for Dynamic Web Delivery*, Wiley, 2002), Nakano (*Web Content Management*, Addison-Wesley, 2001) presentan valiosos tratamientos de la materia.

En Internet hay disponible una amplia variedad de fuentes de información acerca de la gestión de configuración del software. Una lista actualizada de referencias en la World Wide Web se puede encontrar en el sitio Web SEPA:
http://www.mhhe.com/pressman.

PARTE

Cinco

TEMAS AVANZADOS EN INGENIERÍA DEL SOFTWARE

En esta parte de *Ingeniería del software. Un enfoque práctico* se consideran varios temas avanzados que ampliarán la comprensión del lector acerca de la ingeniería del software. En los siguientes capítulos se abordarán las siguientes preguntas:

- ¿Qué notación y preliminares matemáticos ("métodos formales") se requieren para especificar formalmente el software?
- ¿Qué actividades técnicas clave se llevan a cabo durante el proceso de ingeniería del software de sala limpia?
- ¿Cómo se emplea la ingeniería del software basada en componentes para crear sistemas a partir de componentes reutilizables?
- ¿Qué actividades técnicas se requieren para la reingeniería del software?
- ¿Cuáles son las tendencias futuras de la ingeniería del software?

Una vez que se respondan estas preguntas, se entenderán los temas que tienen la posibilidad de tener un profundo impacto sobre la ingeniería del software durante la década siguiente.

CAPÍTULO

28 MÉTODOS FORMALES

CONCEPTOS CLAVE

Los métodos de la ingeniería del software se pueden clasificar sobre un espectro de "formalidad" ligeramente vinculado con el grado de rigor matemático aplicado durante el análisis y el diseño. Por esta razón, los métodos de análisis y diseño estudiados previamente en este libro se ubican en el extremo informal del espectro. En la creación de modelos de análisis y diseño se utilizan combinaciones de diagramas, texto, tablas y notación simple, pero se ha aplicado poco rigor matemático

Considérese ahora el otro extremo del espectro de formalidad. Aquí, una especificación y diseño se describen empleando sintaxis y semánticas formales que especifican la función y el comportamiento del sistema. La especificación es matemática en forma (por ejemplo, el cálculo de predicados se utiliza como base para un lenguaje formal de especificación)

En su estudio introductorio de los métodos formales, Anthony Hall [HAL96] afirma:

> Los métodos formales son controvertidos. Sus partidarios afirman que pueden revolucionar el desarrollo [del software] Sus detractores piensan que son increiblemente difíciles Mientras tanto, para la mayoría de la gente, los métodos formales son tan poco familiares que es difícil juzgar las afirmaciones en competencia.

En este capítulo se exploran los métodos formales y se examinan sus potenciales impactos sobre la ingeniería del software en los años por venir

UN VISTAZO RÁPIDO

¿Qué es? Los métodos formales permiten que un ingeniero de software cree una especificación mas completa, consistente y precisa que las que se producen empleando métodos convencionales. Se utilizan la notación de teoría de conjuntos y lógica para crear un claro planteamiento de hechos (requisitos). Esta especificación matemática luego se analiza para mejorar (o incluso probar) su corrección y consistencia. Puesto que la especificación se crea mediante notación matemática, es inherentemente menos ambigua que los modos informales de representación.

¿Quién lo hace? Un ingeniero de software especialmente entrenado crea una especificación formal.

¿Por qué es importante? En los sistemas de seguridad o de misión críticas, las fallas tienen un precio elevado. Cuando el software de computadora falla es posible que se pierdan vidas o surjan graves consecuencias económicas. En tales situaciones es esencial que los errores sean descubiertos antes de que el software sea puesto en operación. Los métodos formales reducen sustancialmente los errores de especificación y, como consecuencia, sirven como base para que el software tenga tan pocos errores una vez que el cliente comience a usarlo.

¿Cuáles son los pasos? La notación y heurística de conjuntos y la especificación constructiva —operadores de conjuntos, operadores lógicos y secuencias— forman la base de los métodos formales. Éstos definen los datos invariantes, estados y operaciones para la función de un sistema al traducir los requisitos formales del problema en una representación más formal.

¿Cuál es el producto obtenido? Cuando se aplican los métodos formales se produce una especificación representada en un lenguaje formal como OCL o Z.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Puesto que los métodos formales utilizan matemáticas discretas como mecanismo de especificación, se aplican pruebas lógicas a cada función del sistema para demostrar que la especificación es correcta. Sin embargo, incluso si no se aplican las pruebas lógicas, la estructura y disciplina de una especificación formal conducirán a una calidad de software mejorada.

28.1 Conceptos básicos

La *Encyclopedia of Software Engineering* [MAR94] define así los métodos formales:

> Un método es formal si tiene sólidas bases matemáticas, usualmente proporcionadas por un lenguaje formal de especificación. Esta base ofrece los medios de definir con precisión nociones como consistencia y completud, y, con más relevancia, especificación, implementación y corrección.

Las propiedades deseadas de una especificación formal (consistencia, integridad y falta de ambigüedad) son los objetivos de todos los métodos de especificación. Sin embargo, emplear métodos formales resulta en una probabilidad mucho mayor de lograr dichos ideales. La sintaxis de un lenguaje formal de especificación (sección 28.4) permite que los requisitos y el diseño se interpreten sólo en una forma, lo que elimina la ambigüedad que con frecuencia ocurre cuando un lenguaje natural (por ejemplo, inglés) o una notación gráfica debe interpretarlos un lector. Las facilidades descriptivas de la teoría de conjuntos y la notación lógica (sección 28.2) permiten un planteamiento claro de los hechos (requisitos). Para ser consistente, los hechos planteados en un lugar en una especificación no deben contradecirse en otro sitio. La consistencia se garantiza probando matemáticamente que los hechos iniciales pueden correlacionarse de manera formal (empleando reglas de inferencia) en planteamientos posteriores dentro de la especificación.

> "Los métodos formales tienen un enorme potencial para mejorar la claridad y la precisión de las especificaciones de requisitos y para encontrar los errores importantes y sutiles."
>
> **Steve Easterbrook *et al.***

La completud es difícil de lograr, aun cuando se apliquen los métodos formales. Algunos aspectos de un sistema tal vez queden indefinidos mientras se crea la especificación; otras características quizá se omitan a propósito para permitir que los diseñadores tengan cierta libertad de elección en el enfoque de implementación; y, finalmente, es imposible considerar cada escenario operativo en un gran sistema complejo. Las cosas tal vez se omitan por error.

Aunque el formalismo que ofrecen las matemáticas es atractivo para algunos ingenieros de software, otros (algunos dirían la mayoría) miran despectivamente la vi-

sión matemática del desarrollo del software. Comprender por qué un enfoque formal tiene su mérito requiere, primero, considerar las deficiencias asociadas con los enfoques menos formales

28.1.1 Deficiencias de los enfoques menos formales[1]

Los métodos estudiados para el análisis y el diseño en las partes 2 y 3 de ese libro emplean ampliamente el lenguaje natural y una extensa gama de notaciones gráficas. Aunque la aplicación cuidadosa de los métodos de análisis y diseño, en conjunto con revisiones exhaustivas, puede conducir y de hecho conduce a software de alta calidad, el descuido en la aplicación de estos métodos crea una diversidad de problemas. Una especificación de sistema puede contener contradicciones, ambigüedades, vaguedades, planteamientos incompletos y grados mixtos de abstracción.

CONSEJO

Aunque un buen índice de documento no puede eliminar las contradicciones, sí puede ayudar a descubrirlas. Considérese la creación de un índice para las especificaciones y otros documentos.

Las *contradicciones* son conjuntos de planteamientos que divergen unos con otros. Por ejemplo, parte de una especificación de sistema puede afirmar que el sistema debe supervisar todas las temperaturas en un reactor químico, mientras que otra parte, que tal vez escribió otra persona, puede afirmar que sólo deben supervisarse las temperaturas que ocurran dentro de cierto intervalo.

Las *ambigüedades* son planteamientos que se interpretan en varias formas. Por ejemplo, el siguiente planteamiento es ambiguo:

> La identidad del operador consiste de su nombre y la contraseña; ésta consiste de seis dígitos. Se debe mostrar en la UDV de seguridad y depositarse en el archivo de registro cuando un operador se registre en el sistema

En este fragmento, *¿se debe mostrar* se refiere a la contraseña o a la identidad del operador?

Las *vaguedades* con frecuencia ocurren porque la especificación de un sistema es un documento muy voluminoso. Lograr un elevado grado de precisión consistentemente es una tarea casi imposible.

> "Cometer errores es humano. Repetirlos, también."
>
> **Malcolm Forbes**

La *incompletud* es uno de los problemas que ocurren con mayor frecuencia con las especificaciones del sistema. Por ejemplo, considérese el requisito funcional:

> El sistema debe conservar el nivel horario del depósito a partir de los sensores profundos situados en el depósito. Dichos valores deben almacenarse respecto de los seis meses anteriores

Esto describe la parte principal del almacenamiento de datos de un sistema. Si uno de los comandos para el sistema fuese

1 Esta sección y otras en la primera parte de este capítulo se han adaptado del trabajo de Darrel Ince para la edición europea de la quinta edición de *Ingeniería del software: Un enfoque práctico*.

CONSEJO

Las revisiones técnicas formales eficaces pueden eliminar muchos de estos problemas. Sin embargo, algunos no serán descubiertos. Debe estarse alerta de las deficiencias durante el diseño, la codificación y la puesta a prueba.

La función del comando PROMEDIO es desplegar en una PC el nivel de agua promedio para un sensor particular entre dos tiempos.

y si supone que no se presentan más detalles para este comando, los detalles del comando estarían seriamente incompletos. Por ejemplo, la descripción del comando no incluye lo que debería ocurrir si un usuario de un sistema especifica un tiempo que fuese mayor a seis meses antes de la hora actual.

Los *grados mixtos de abstracción* ocurren cuando planteamientos muy abstractos se mezclan al azar con planteamientos que tienen un grado mucho menor de detalle. Aunque ambos tipos de planteamientos son importantes en la especificación de un sistema, quienes la realizan con frecuencia manejan la mezcla en tal forma que resulta muy difícil ver la arquitectura funcional global de un sistema.

28.1.2 Matemáticas en el desarrollo de software

Las matemáticas tienen muchas propiedades útiles para los desarrolladores de grandes sistemas. Una es que se puede describir sucinta y exactamente una situación física, un objeto o el resultado de una acción. Es posible desarrollar una especificación de un sistema basado en computadora empleando matemáticas especializadas, en forma muy similar a la que un ingeniero eléctrico aplica las matemáticas para describir un circuito.[2]

Las matemáticas sustentan la abstracción y por ende son un excelente medio para el modelado. Puesto que se trata de un medio exacto, existe poca posibilidad de ambigüedad. Las especificaciones pueden validarse matemáticamente para contradicciones e incompletudes, y se pueden eliminar las vaguedades. Además, las matemáticas se emplean para representar, en una forma organizada, grados de abstracción en la especificación de un sistema.

Finalmente, las matemáticas ofrecen un alto grado de validación al emplearlas como un medio de desarrollo de software. Es posible aplicar una prueba matemática para demostrar que un diseño encaja en una especificación y que el código del programa es un reflejo correcto de un diseño.

28.1.3 Conceptos de métodos formales

La meta de esta sección es presentar los principales conceptos involucrados en la especificación matemática de los sistemas de software, sin atiborrar al lector con demasiados detalles matemáticos. Esto se logra empleado unos cuantos ejemplos simples.

Ejemplo 1: una tabla de símbolos. Un programa se emplea para mantener una tabla de símbolos, la cual se utiliza frecuentemente en diferentes tipos de aplicacio-

2 En este momento es adecuada una advertencia. Las especificaciones del sistema matemático que se presentan en este capítulo no son tan sucintas como una especificación matemática para un circuito simple. Los sistemas de software son notablemente complejos y sería irreal esperar que se podrían especificar en una línea de material.

nes. Consiste de una colección de elementos sin duplicación alguna. En la figura 28.1 se muestra un ejemplo de una tabla de símbolos común. En ella se representa la tabla que utiliza un sistema operativo para retener los nombres de los usuarios del sistema. Otros ejemplos de tablas incluyen la colección de nombres del personal de un sistema de nómina o la colección de los nombres de computadoras en un sistema de comunicaciones en red.

PUNTO CLAVE

Un invariante de datos es un conjunto de condiciones verdaderas a lo largo de la ejecución del sistema que contiene una colección de datos

Supóngase que la tabla presentada en este ejemplo consiste de no más de *MaxIds* miembros de personal. Este planteamiento, que coloca una restricción en la tabla, es un componente de una condición conocida como *invariante de datos*, una importante idea que se retomará a lo largo de este capítulo.

Un invariante de datos es una condición verdadera a lo largo de la ejecución del sistema que contiene una colección de datos. La invariante de datos que se mantiene para la tabla de símbolos apenas estudiada tiene dos componentes: 1) que la tabla contendrá no más de *MaxIds* nombres y 2) que no habrá nombres duplicados en la tabla. En el caso del programa de tabla de símbolos, esto significa que no importa cuándo se examine la tabla durante la ejecución del sistema, siempre contendrá no más de *MaxIds* identificadores de personal y no incluirá duplicados.

CONSEJO

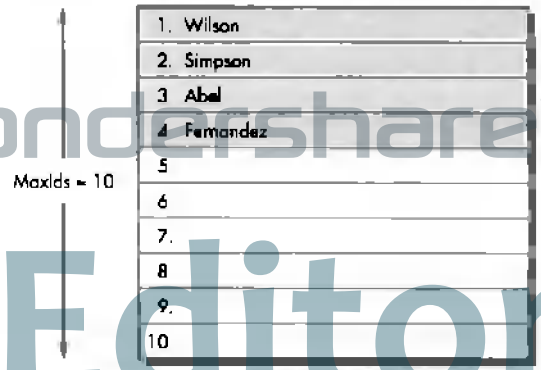
Otra forma de observar la noción de estado es decir que los datos determinan el estado. Esto es, se pueden examinar los datos para ver en qué estado está el sistema.

Otro concepto importante es el de *estado*. Muchos lenguajes formales, como OCL (sección 28.5), utilizan la noción de estado como se le estudió en los capítulos 7 y 8; es decir: un sistema puede estar en uno de varios estados, y cada uno de ellos representa un modo de comportamiento observable externamente. Sin embargo, una definición diferente para el término *estado* se utiliza en el lenguaje Z (sección 28.6). En Z (y en lenguajes relacionados), el estado de un sistema se representa mediante los datos almacenados del sistema (por tanto, Z sugiere un número mucho mayor de estados, que representan cada posible configuración de los datos). Al emplear la última definición en el ejemplo del programa de tabla de símbolos, el estado es la tabla de símbolos.

El concepto final es el de *operación*. Ésta es una acción que ocurre dentro de un sistema y lee o escribe datos. Si el programa de tabla de símbolos está preocupado

FIGURA 28.1

Tabla de símbolos.



por añadir y remover nombres de personal de la tabla de símbolos, entonces estará asociado con dos operaciones: una para *añadir* un nombre específico a la tabla de símbolos, y una para *eliminar* un nombre existente de la tabla.[3] Si el programa ofrece la capacidad de verificar si un nombre específico está en la tabla, entonces habría una operación que regresaría alguna indicación acerca de la presencia del nombre en la tabla.

Tres tipos de condiciones se asocian con las operaciones: invariantes, precondiciones y poscondiciones. Un *invariante* define lo que está garantizado que no cambiará. Por ejemplo, la tabla de símbolos tiene un invariante que establece que el número de elementos siempre será menor o igual a *MaxIds*. Una *precondición* define las circunstancias en las cuales es válida una operación particular. Por ejemplo, la precondición para una operación que añade un nombre a la tabla de símbolos de identificadores de personal sólo es válida si el nombre no está en la tabla y también si en ésta existen menos de *MaxIds* identificadores de personal. La *poscondición* de una operación define lo que está garantizado que será cierto hasta completar una operación. Esto lo define su efecto sobre los datos. En el ejemplo de una operación que añade un identificador a la tabla de símbolos de identificadores de personal, la poscondición especificaría matemáticamente que la tabla ha aumentado con el nuevo identificador.

CONSEJO

Las técnicas de lluvia de ideas pueden funcionar bien cuando se debe desarrollar un diccionario de datos para una función razonablemente compleja. Cada miembro del equipo de software debe escribir los nombres, restricciones y limitaciones para la función y luego combinarlos y editarlos

Ejemplo 2: un gestor de bloques. Una de las partes más importantes del sistema operativo de una computadora es el subsistema que mantiene los archivos que hayan creado los usuarios. Parte del subsistema de archivado es el *gestor de bloques*. Los archivos en el archivero están compuestos de bloques de almacenamiento que se mantienen en un dispositivo de almacenamiento de archivos. Durante la opera-

FIGURA 28.2

Gestor de bloques.

3 Se debe señalar que añadir un nombre no puede ocurrir en el estado lleno, y que borrar un nombre es imposible en el estado vacío

ción de la computadora, los archivos se crearán y borrarán, lo que requiere la adquisición y liberación de bloques de almacenamiento. El subsistema de archivado lidiará con esto manteniendo un depósito de bloques no utilizados (libres) y dará seguimiento a los bloques que actualmente están en uso. Cuando se borra un archivo, los bloques liberados normalmente se agregan a una fila de bloques que esperan ser agregados al depósito de bloques no utilizados. Esto se muestra en la figura 28.2. En esta figura se muestran varios componentes: el depósito de bloques no utilizados, los bloques que actualmente constituyen los archivos que administra el sistema operativo y aquellos bloques que esperan agregarse al depósito. Los bloques en espera se mantienen en una fila, y cada elemento de ésta contiene un conjunto de bloques provenientes de un archivo borrado.

En este subsistema el estado es la colección de bloques libres, la colección de bloques utilizados y la fila de bloques devueltos. El invariante de datos, expresado en lenguaje natural, es:

- Ningún bloque será marcado a la vez como no utilizado y utilizado.
- Todos los conjuntos de bloques mantenidos en la fila serán subconjuntos de la colección de bloques actualmente usados.
- Ningún número de bloque pertenecerá a dos o más elementos de la fila.
- La colección de bloques utilizados y no utilizados será la colección total de bloques que configuran los archivos.
- La colección de bloques no utilizados no tendrá números de bloque duplicados
- La colección de bloques utilizados no tendrá números de bloque duplicados.

Algunas de las operaciones asociadas con el invariante de datos son: *añadir( )* una colección de bloques al final de la fila, *eliminar( )* una colección de bloques utilizados del frente de la fila y colocarlos en la colección de bloques no utilizados, y *verificar( )* si la fila de bloques está vacía.

La precondición de la primera operación es que los bloques que se añadirán deben estar en la condición de bloques utilizados. La poscondición es que la colección de bloques ahora se encuentra al final de la fila. La precondición de la segunda operación es que la fila debe tener al menos un elemento en ella. La operación final —verificar si la fila de bloques devueltos está vacía— no tiene precondición. Esto significa que la operación siempre está definida, sin importar de qué valor sea el estado. La poscondición entrega el valor *cierto* si la fila está vacía y *falso* de cualquier otro modo.

En los ejemplos tratados en esta sección se introducen los conceptos clave de la especificación formal. Esto se hizo sin resaltar las matemáticas que se requieren para formalizar la especificación. En la sección 28.2 se consideran tales matemáticas.

28.2 PRELIMINARES MATEMÁTICOS

La aplicación eficaz de los métodos formales requiere que un ingeniero de software tenga un conocimiento operativo de la notación matemática asociada con los conjuntos y las secuencias, y de la notación lógica utilizada en el cálculo de predicados. La finalidad de la sección es proporcionar una breve introducción al tema. Para una exposición más detallada, se recomienda al lector consultar libros dedicados a estos temas (por ejemplo, [WIL87], [GRI93] y [ROS95])

28.2.1 Conjuntos y especificación constructiva

Un *conjunto* es una colección de objetos o elementos, y se utiliza como piedra angular de los métodos formales. Los elementos que contiene un conjunto son únicos (es decir: no se permiten duplicaciones) Los conjuntos con un número pequeño de elementos se escriben dentro de llaves, con los elementos separados por comas. Por ejemplo, el conjunto

{C++, Smalltalk, Ada, COBOL, Java}

contiene los nombres de cinco lenguajes de programación

El orden en el que aparecen los elementos dentro de un conjunto es irrelevante. Al número de elementos en un conjunto se le conoce como cardinalidad. El operador # proporciona la cardinalidad de un conjunto. Por ejemplo, la expresión

$$\#\{A, B, C, D\} = 4$$

implica que se ha aplicado el operador cardinalidad al conjunto mostrado, con un resultado que indica el número de elementos en el conjunto.

¿Qué es especificación constructiva de conjuntos?

Existen dos formas de definir un conjunto. Un conjunto se define enumerando sus elementos (así se han definido los conjuntos mencionados). El segundo enfoque consiste en crear una *especificación constructiva de conjuntos* La forma general de los miembros de un conjunto se especifica empleando una expresión booleana. La especificación constructiva de conjuntos es preferible a la enumeración porque ello permite una definición sucinta de conjuntos grandes. También define explícitamente la regla que se aplicó al construir el conjunto Considérese el siguiente ejemplo de especificación constructiva:

$$\{n : \mathbb{N} \mid n < 3 \bullet n\}$$

Esta especificación tiene tres componentes: una firma, $n : \mathbb{N}$; un predicado, $n < 3$; y un término, n. La *firma* especifica el intervalo de valores que se considerarán cuando se forme el conjunto; el *predicado* (una expresión Booleana) define cómo se restringirá el conjunto; y, finalmente, el *término* brinda la forma general del elemento del conjunto. En el ejemplo anterior, $\mathbb{N}$ representa los números naturales; por lo tanto, se considerarán los números naturales. El predicado indica que sólo se incluirán

CONSEJO

Es indispensable el conocimiento de las operaciones de conjuntos cuando se desarrollen especificaciones formales. Debe pasarse algún tiempo familiarizándose con cada una, si se tiene la intención de aplicar métodos formales.

los números naturales menores que 3; y el término especifica que cada elemento de conjunto será de la forma n. En consecuencia, esta especificación define el conjunto

$$\{0, 1, 2\}$$

Cuando la forma de los elementos de un conjunto es obvia, el término se puede omitir. Por ejemplo, el conjunto precedente se podría especificar como

$$\{n : \mathbb{N} \mid n < 3\}$$

Todos los conjuntos que se han descrito tienen elementos que son elementos individuales. También se pueden formar conjuntos a partir de elementos que sean pares, ternas, etcétera. Por ejemplo, la especificación de conjunto

$$\{x, y : \mathbb{N} \mid x + y = 10 \bullet (x, y^2)\}$$

describe el conjunto de pares de números naturales que tienen la forma (x, y^2) y donde la suma de x e y es 10. Éste es el conjunto

$$\{(1, 81), (2, 64), (3, 49), \ldots\}$$

Obviamente, la especificación constructiva de conjuntos requerida para representar algún componente de software de computadora será considerablemente más compleja que las anotadas aquí. Sin embargo, la forma y estructura básicas permanecen iguales

28.2.2 Operadores de conjuntos

En la representación de operaciones de conjuntos y lógicas se utiliza simbología especializada. El ingeniero de software que pretenda aplicar los métodos formales debe comprender estos símbolos.

El operador $\in$ indica la pertenencia de un conjunto. Por ejemplo, la expresión

$$x \in X$$

tiene el valor *verdadero* si x es miembro del conjunto X y el valor *falso* en caso contrario. Por ejemplo, el predicado

$$12 \in \{6, 1, 12, 22\}$$

tiene el valor *verdadero* dado que 12 es miembro del conjunto.

El opuesto del operador $\in$ es el operador $\notin$. La expresión

$$x \notin X$$

tiene el valor *verdadero* si x no es miembro del conjunto X y *falso* en caso contrario Por ejemplo, el predicado

$$13 \notin \{13, 1, 124, 22\}$$

tiene el valor *falso*.

Los operadores $\subset$ y $\subset$ tienen conjuntos como sus operandos. El predicado

$$A \subset B$$

tiene el valor *verdadero* si los miembros del conjunto *A* están en el conjunto *B* y tiene el valor *falso* en caso contrario. Por lo tanto, el predicado

{1, 2} ⊂ {4, 3, 1, 2}

tiene el valor *verdadero*. Sin embargo, el predicado

{HD1, LP4, RC5} ⊂ {HD1, RC2, HD3, LP1, LP4, LP6}

tiene un valor de *falso* porque el elemento RC5 no está en el conjunto a la derecha del operador.

El operador ⊆ es similar a ⊂. Sin embargo, si sus operandos son iguales, tiene el valor *verdadero*. Por lo tanto, el valor del predicado

{HD1, LP4, RC5} ⊆ {HD1, RC2, HD3, LP1, LP4, LP6}

es *falso*, y el predicado

{HD1, LP4, RC5} ⊆ {HD1, LP4, RC5}

es *verdadero*.

> "Las estructuras matemáticas están entre los descubrimientos más hermosos realizados por la mente humana."
> **Douglas Hofstadter**

Un conjunto especial es el conjunto vacío ∅. Éste corresponde a cero en las matemáticas normales. El conjunto vacío tiene la propiedad de ser un subconjunto de cualquier otro conjunto. Dos útiles identidades que involucran al conjunto vacío son

$$\varnothing \cup A = A \quad y \quad \varnothing \cap A = \varnothing$$

para cualquier conjunto *A*, donde ∪ se conoce como el operador unión, a veces conocido como *taza*; ∩ es el *operador intersección*, a veces conocido como *gorra*.

El operador unión admite dos conjuntos y forma uno que contiene los elementos de los dos conjuntos y elimina los duplicados. Por lo tanto, el resultado de la expresión

{Archivo1, Archivo2, Impuesto, Compilador} ∪ {ImpuestoNuevo, D2, D3, Archivo2}

es el conjunto

{Archivo1, Archivo2, Impuesto, Compilador, ImpuestoNuevo, D2, D3}

El operador de intersección admite dos conjuntos y forma uno que consiste de los elementos comunes en cada conjunto. Por lo tanto, la expresión

{12, 4, 99, 1} ∩ {1, 13, 12, 77}

genera el conjunto {12, 1}.

El *operador diferencia de conjuntos*, \, como su nombre sugiere, forma un conjunto al eliminar los elementos de su segundo operando de los elementos de su primer operando. Por lo tanto, el valor de la expresión

{Nuevo, Viejo, ArchivoImpuesto, ParamSis} \ {Viejo, ParamSis}

genera el conjunto {Nuevo, ArchivoImpuesto}.

El valor de la expresión

{a, b, c, d} ∩ {x, y}

será el conjunto vacío ∅ El operador siempre proporciona un conjunto; sin embargo, en este caso no existen elementos comunes entre sus operandos, así que el conjunto resultante no tendrá elementos.

El operador final es el *producto cruzado*, × , a veces conocido como *producto cartesiano*. Éste tiene dos operandos. El resultado es un conjunto de pares donde cada par consiste de un elemento tomado del primer operando combinado con un elemento del segundo Un ejemplo de una expresión que involucra al producto cruzado es

{1, 2} × {4, 5, 6}

El resultado de esta expresión es

{(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2,6)}

Nótese que cada elemento del primer operando está combinado con cada uno de los elementos del segundo

Un concepto importante en los métodos formales es el de *conjunto potencia* Un conjunto potencia de un conjunto es la colección de todos los posibles subconjuntos de dicho conjunto. El símbolo que se utiliza para este operador de conjunto en este capítulo es ℙ Se trata de un operador unitario que, cuando se aplica a un conjunto, devuelve el conjunto de subconjuntos de su operando Por ejemplo,

ℙ {1, 2, 3} = {∅, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}

ya que todos los conjuntos son subconjuntos de {1, 2, 3}.

28.2.3 Operadores lógicos

Otro componente importante de un método formal es la *lógica*: el álgebra de expresiones verdaderas y falsas El significado de los operadores lógicos comunes lo comprende bien cualquier ingeniero de software. Sin embargo, los operadores lógicos asociados con los lenguajes de programación comunes se escriben empleando símbolos disponibles fácilmente en el teclado. Los operadores matemáticos equivalentes son

∧ y

∨ o

¬ no

⇒ implica

La *cuantificación universal* es una forma de elaborar un planteamiento acerca de los elementos de un conjunto que resulta verdadero para cualquier miembro del

conjunto. La cuantificación **universal** utiliza el símbolo ∀. Un ejemplo de su utilización es

$$\forall i\, j : \mathbb{N} \bullet i > j \Rightarrow i^2 > j^2$$

en donde se establece que para cada par de valores en el conjunto de números naturales, si i es mayor que j, entonces i^2 es mayor que j^2.

28.2.4 Sucesiones

Una sucesión es una estructura matemática que modela el hecho de que sus elementos están ordenados. Una sucesión s es un conjunto de pares cuyos elementos varían de 1 al elemento de mayor número. Por ejemplo,

{(1, Jones), (2, Wilson), (3, Shapiro), (4, Estavez)}

es una sucesión. Los elementos que forman los primeros elementos de los pares se conocen colectivamente como *dominio* de la sucesión, y la colección de segundos elementos se conoce como el *intervalo* de la sucesión. En este libro, las sucesiones están indicadas mediante corchetes angulados. Por ejemplo, la sucesión precedente normalmente se escribiría como ⟨Jones, Wilson, Shapiro, Estavez⟩.

A diferencia de los conjuntos, la duplicación se permite en una sucesión, cuyo orden es importante. Por lo tanto,

⟨Jones, Wilson, Shapiro⟩ ≠ ⟨Jones, Shapiro, Wilson⟩

La sucesión vacía se representa como ⟨⟩.

En las especificaciones formales se utilizan varios operadores de sucesión. La concatenación, ⁀, es un operador binario que forma una sucesión construida al agregar su segundo operando al final de su primer operando. Por ejemplo,

⟨2, 3, 34, 1⟩ ⁀ ⟨12, 33, 34, 200⟩

genera la sucesión ⟨2, 3, 34, 1, 12, 33, 34, 200⟩.

Otros operadores que se aplican a las sucesiones son *cabeza*, *cola*, *frente* y *último*. El operador *cabeza* extrae el primer elemento de una sucesión; *cola* proporciona los últimos $n - 1$ elementos en una sucesión de longitud n; *último* extrae el elemento final en una sucesión; y *frente* proporciona los primeros $n - 1$ elementos en una sucesión de longitud n. Por ejemplo,

cabeza ⟨2, 3, 34, 1, 99, 101⟩ = 2
cola ⟨2, 3, 34, 1, 99, 101⟩ = ⟨3, 34, 1, 99, 101⟩
último ⟨2, 3, 34, 1, 99, 101⟩ = 101
frente ⟨2, 3, 34, 1, 99, 101⟩ = ⟨2, 3, 34, 1, 99⟩

Dado que una sucesión es un conjunto de pares, se pueden aplicar todos los operadores de conjunto descritos en la sección 28.2.2. Cuando se emplea una sucesión en un estado, se debe designar mediante la palabra *seq*. Por ejemplo,

ArchivoLista : *seq* ARCHIVOS
NumUsuarios : ℕ

describen un estado con dos componentes: una sucesión de archivos y un número natural.

28.3 APLICACIÓN DE LA NOTACIÓN MATEMÁTICA PARA LA ESPECIFICACIÓN FORMAL

El empleo de la notación matemática en la especificación formal de un componente de software se ilustrará repasando el gestor de bloques presentado en la sección 28.1.3. Un importante componente de un sistema operativo de computadora mantiene los archivos que han creado los usuarios. El gestor de bloques mantiene un depósito de bloques no utilizados y también seguirá los bloques actualmente en uso. Cuando se liberan bloques de un archivo borrado normalmente se añaden a una fila de bloques que esperan ser añadidos al depósito de bloques no utilizados. En la figura 28.2 se ha bosquejado esto esquemáticamente.[4]

¿Cómo se pueden representar los estados e invariantes de datos empleando los operadores lógicos y de conjuntos que ya se han introducido?

Un conjunto llamado *BLOQUES* consistirá de todos los números de bloque. *TodosBloques* es un conjunto de bloques que se ubica entre 1 y *MáxBloques*. El estado lo modelarán dos conjuntos y una sucesión. Los dos conjuntos son *usados* y *libres*. Ambos contienen bloques: el conjunto *usados* contiene los bloques que actualmente se están utilizando en los archivos, y el conjunto *libres* contiene los bloques disponibles para los archivos nuevos. La sucesión contendrá conjuntos de bloques que están listos para ser liberados de los archivos que se han borrado. El estado se puede describir como

$$usados, libres: \mathbb{P}\ BLOQUES$$
$$FilaBloques: seq\ \mathbb{P}\ BLOQUES$$

Esto es muy parecido a la declaración de variables de programa. Establece que *usados* y *libres* serán conjuntos de bloques y que *FilaBloques* será una sucesión, cada elemento de la cual será un conjunto de bloques. El invariante de datos se puede escribir como

$$usados \cap libres = \varnothing \wedge$$
$$usados \cup libres = TodosBloques \wedge$$
$$\forall i: \text{dom } FilaBloques \bullet FilaBloques\ i \subset usados \wedge$$
$$\forall i, j: \text{dom } FilaBloques \bullet i \neq j \Rightarrow FilaBloques\ i \cap FilaBloques\ j = \varnothing$$

Referencia Web
Información extensa acerca de los métodos formales se puede encontrar en www.afm.sbu.ac.uk.

Los componentes matemáticos del invariante de datos se corresponden con cuatro de los componentes de lenguaje natural marcados que se describieron anteriormente. La primera línea del invariante de datos establece que no existirán bloques comunes en las colecciones de bloques usados y libres. La segunda línea afirma que la colec-

4 Si no se recuerda bien el ejemplo del gestor de bloques, por favor véase de nuevo la sección 28.1.3 para revisar el invariante de datos, las operaciones, precondiciones y poscondiciones asociadas con el gestor de bloques.

ción de bloques usados y libres siempre será igual a toda la colección de bloques en el sistema. La tercera línea indica que el *i*-ésimo elemento en la fila de bloques siempre será un subconjunto de los bloques usados. La línea final afirma que, para cualesquier dos elementos de la fila de bloques que no son el mismo, no habrá bloques comunes en estos dos elementos. Los dos componentes finales de lenguaje natural del invariante de datos se implementan en virtud del hecho de que *usados* y *libres* son conjuntos y por lo tanto no contendrán duplicados.

La primera operación que se definirá es la que elimina un elemento de la cabeza de la fila de bloques. La precondición es que debe existir al menos un elemento en la fila:

$$\#\ \text{FilaBloques} > 0,$$

La poscondición es que la cabeza de la fila debe eliminarse y colocarse en la colección de bloques libres, y la Fila se debe ajustar para mostrar la eliminación:

$$usados' = usados \setminus cabeza\ FilaBloques \land$$
$$libres' = libres \cup cabeza\ FilaBloques \land$$
$$FilaBloques' = cola\ FilaBloques$$

Una convención que se utiliza en muchos métodos formales es que al valor de una variable después de una operación se le pone prima. Por lo tanto, el primer componente de la expresión precedente afirma que los nuevos bloques usados (*usados'*) serán iguales a los antiguos bloques usados menos los bloques que se han eliminado. El segundo componente afirma que los nuevos bloques libres (*libres'*) serán los antiguos bloques libres más la cabeza de la fila de bloques. El tercer componente establece que la nueva fila de bloques será igual a la cola del antiguo valor de la fila de bloques; esto es: todos los elementos en la fila, menos el primero.

¿Cómo se representan las pre y poscondiciones?

Una segunda operación agrega una colección de bloques, *Abloques*, a la fila de bloques. La precondición es que *Abloques* sea actualmente un conjunto de bloques usados:

$$Abloques \subset usados$$

La poscondición es que el conjunto de bloques sea añadido al final de la fila, y el conjunto de bloques usados y libres permanezca invariable:

$$FilaBloques' = FilaBloques \frown \text{Abloques} \land$$
$$usados' = usados \land$$
$$libres' = libres$$

No existe duda de que la especificación matemática de la fila de bloques es considerablemente más rigurosa que una narración en lenguaje natural o un modelo gráfico. El rigor adicional requiere esfuerzo, pero los beneficios obtenidos a partir de la consistencia y la completud mejoradas se pueden justificar para muchos tipos de aplicaciones.

28.4 LENGUAJES FORMALES DE ESPECIFICACIÓN

Un lenguaje formal de especificación usualmente está compuesto de tres componentes principales: 1) una *sintaxis* que define la notación específica con la que se representa la especificación, 2) *semántica* para ayudar a definir un "universo de objetos" [WIN90] que se empleará para describir el sistema, y 3) un *conjunto de relaciones* que definen las reglas que indican cuáles objetos satisfacen adecuadamente la especificación

Con frecuencia, el dominio sintáctico de un lenguaje formal de especificación se basa en una sintaxis que se deriva de la notación estándar de la teoría de conjuntos y el cálculo de predicados. Por ejemplo, variables tales como *x*, *y* y *z* describen un conjunto de objetos que se relacionan con un problema (en ocasiones llamado *dominio del discurso*) y se utilizan en conjunto con los operadores descritos en la sección 28 2 Aunque la sintaxis usualmente es simbólica, también se pueden utilizar iconos (por ejemplo, símbolos gráficos como cajas, flechas y círculos) si no son ambiguos

El *dominio semántico* de un lenguaje de especificación indica cómo el lenguaje representa los requisitos del sistema. Por ejemplo, un lenguaje de programación tiene un conjunto de semánticas formales que permite al desarrollador de software especificar algoritmos que transforman entrada en salida Se puede utilizar una gramática formal (como BNF) para describir la sintaxis del lenguaje de programación. Sin embargo, un lenguaje de programación no hace un buen lenguaje de especificación porque sólo representa funciones de cómputo. Un lenguaje de especificación debe tener un dominio semántico más amplio, es decir, debe ser capaz de expresar ideas como "para toda *x* en un conjunto infinito *A*, existe una *y* en un conjunto infinito *B* tal que la propiedad *P* se mantiene para *x* y *y*" [WIN90]. Otros lenguajes de especificación aplican semánticas que permiten la especificación del comportamiento del sistema. Por ejemplo, es posible desarrollar una sintaxis y una semántica para especificar estados y transiciones de estado, y eventos, junto con sus efectos sobre la transición de estado, la sincronización y la temporalidad

Es posible usar diferentes abstracciones semánticas para describir el mismo sistema en diferentes formas. En el capítulo 8 esto se hizo en una manera menos formal. Se representaron clases, datos, funciones y comportamiento. Se puede usar diferente notación de modelado para representar el mismo sistema. La semántica de cada representación ofrece visiones complementarias del sistema Para ilustrar este enfoque cuando se usan métodos formales, suponga que se usa un lenguaje formal de especificación para describir al conjunto de eventos que provocan que ocurra un estado particular en un sistema. Otra relación formal bosqueja todas las funciones que ocurren dentro de un estado dado. La intersección de estas dos relaciones proporciona una indicación de los eventos que causarán que ocurran funciones específicas.

En la actualidad se emplean varios lenguajes formales de especificación. OCL [OMG03], Z ([ISO02], [SPI88], [SPI92]), LARCH [GUT93] y VDM [JON91] son lenguajes

formales de especificación representativos que muestran las características anotadas con anterioridad. En este capítulo se presenta un breve panorama de OCL y Z.

28.5 LENGUAJE RESTRINGIDO A OBJETOS (OCL)[5]

El *lenguaje restringido a objetos* (OCL, *por sus siglas en inglés*) es una notación formal desarrollada de modo que los usuarios de UML puedan conferirle mayor precisión a sus especificaciones. El lenguaje dispone de todo el poder de la lógica y la matemáticas discretas. Sin embargo, los diseñadores de OCL decidieron que en este lenguaje sólo deberían usarse caracteres ASCII (en lugar de notación matemática convencional). Esto permite que el lenguaje sea más asequible a las personas menos inclinadas a las matemáticas y que la computadora los procese con mayor facilidad. Pero esto también favorece que OCL sea un poco farragoso en algunos lugares.

28.5.1 Un breve panorama de la sintaxis y la semántica del OCL

La utilización del OCL requiere que un ingeniero de software comience con uno o más diagramas UML: las clases, estados o diagramas de actividad más comunes. Por lo tanto, se agregan expresiones OCL que establecen hechos acerca de los elementos de los diagramas. Estas expresiones se llaman *restricciones*; cualquier implementación derivada del modelo debe garantizar que cada una de las restricciones siempre permanezca verdadera.

Al igual que el lenguaje de programación orientado a objetos, una expresión OCL involucra operadores que operan sobre los objetos. Sin embargo, el resultado de una expresión completa siempre debe ser booleana, es decir: verdadero o falso. Los objetos pueden ser instancias de la clase OCL **Colección**, de la cual **Conjunto** y **Sucesión** son dos subclases.

Referencia Web
Información detallada acerca de OCL se puede encontrar en www-3.ibm.com/software/awtools/library/standards/ocl.html.

El objeto **self** (propio) es el elemento del diagrama UML en cuyo contexto se evaluará la expresión OCL. Se pueden obtener otros objetos al *navegar* usando el símbolo . (punto) del objeto **self**. Por ejemplo:

- Si **self** es clase **C**, con atributo **a**, entonces **self.a** evalúa al objeto almacenado en **a**.
- Si **C** tiene una asociación multívoca llamada *asoc* con otra clase **D**, entonces **self.asoc** evalúa un **Conjunto** cuyos elementos son del tipo **D**.
- Finalmente (y un poco más sutil), si **D** tiene un atributo **b**, entonces la expresión **self.asoc.b** evalúa el conjunto de todas las **b** que pertenecen a todas las **D**.

5 Esta sección es una aportación del profesor Timothy Lethbridge, de la Universidad de Ottawa, y se presenta aquí mediante autorización.

El OCL proporciona operaciones construidas que implementan las matemáticas descritas en la sección 28.2, y más. En la tabla 28.1 se presenta una pequeña muestra de estas operaciones.

TABLA 28.1 RESUMEN DE NOTACIÓN CLAVE OCL

Notación OCL	Significado
x.y	Obtiene la propiedad y del objeto x. Una propiedad puede ser un atributo, el conjunto de objetos al final de una asociación, el resultado de evaluar una operación u otras cosas que dependan del tipo de diagrama UML. Si x es un Conjunto, entonces y se aplica a todo elemento de x; los resultados se juntan en un nuevo Conjunto.
c->f( )	Aplica la operación OCL incorporada f a la Colección c misma (en oposición a cada uno de los objetos en c). Los ejemplos de operaciones incorporadas se presentan líneas abajo
y, or, =, <>	y lógica, o lógica, igual, no igual
p implies q	Verdadero si q es verdadero o p es falso
Muestra de operaciones sobre colecciones (incluye conjuntos y secuencias)	
c->size( )	Número de elementos en la Colección c
c->esEmpty( )	Verdadero si c no tiene elementos, falso de cualquier otro modo
c1->includesAll(c2)	Verdadero si todo elemento de c2 se encuentra en c1.
c1->excludesAll(c2)	Verdadero si ningún elemento de c2 se encuentra en c1
c1->forAll(elem | boolexpr)	Verdadero si boolexpr (expresión booleana) es verdadera cuando se aplica a todo elemento de c. Conforme se evalúa un elemento, se liga a la variable elem (elemento), que se puede usar en boolexpr. Esto implementa cuantificación universal, estudiada previamente
c->forAll(elem1, elem2 | boolexpr)	Lo mismo que antes, excepto que boolexpr se evalúa para todo posible par de elementos tomados de c, incluso casos donde el par consiste del mismo elemento.
c->is unique(elem | expr)	Verdadero si expr (expresión) evalúa un valor diferente cuando se aplica a todo elemento de c
Muestra de operaciones específicas a conjuntos	
s1->intersection(s2)	El conjunto de aquellos elementos que se encuentran en s1 y también en s2.
s1->union(s2)	El conjunto de aquellos elementos que se encuentran en s1 o en s2
s1->excluding(x)	El conjunto s1 con el objeto x omitido
Muestra de operación específica a secuencias	
seq->first( )	El objeto que es el primer elemento en la sucesión seq.

28.5.2 Ejemplo de uso del OCL

En esta sección se utiliza el OCL para ayudar a formalizar la especificación del ejemplo del gestor de bloques, introducido en la sección 28.1.3. El primer paso consiste en desarrollar un modelo UML. En este ejemplo se comienza con el diagrama de clase de la figura 28 3. Este diagrama especifica muchas relaciones entre los objetos involucrados; sin embargo, se deben agregar expresiones OCL que garanticen que quienes implementen el sistema conozcan con mayor precisión qué deben asegurar que permanezca verdadero conforme se desarrolle el sistema

Las expresiones OCL que se agregarán corresponden a las seis partes del invariante tratadas en la sección 28.1.3. Respecto de cada una se repetirá el invariante en español y luego se brindará la correspondiente expresión OCL. Se considera una buena práctica ofrecer el texto en español junto con la lógica formal; hacerlo ayuda al lector a comprender la lógica, y también apoya a los revisores a descubrir errores, por ejemplo· situaciones donde el español y la lógica no correspondan.

1. Ningún bloque se marcará al mismo tiempo como no usado y usado.

```
context GestorBloques inv:
  (self.usados->intersection(self.libres)) -> isEmpty()
```

Nótese que cada expresión comienza con la palabra clave **context** (contexto). Esto indica el elemento del diagrama UML que la expresión restringe. En forma alterna, el ingeniero de software podría colocar la restricción directamente en el diagrama UML, rodeada por llaves {}. La palabra clave **self** se refiere a la instancia de **GestorBloques**, en lo que sigue, como es permisible en OCL, se omitirá **self**.

2. Todos los conjuntos de bloques que se mantienen en la cola serán subconjuntos de la colección de bloques usados actualmente.

```
context GestorBloques inv:
  colaBloques->forAll(aConjuntoBloques | usados->includesAll(aConjuntoBloques))
```

FIGURA 28.3

Diagrama de clase para un gestor de bloques.

3. Ningún número de bloque pertenecerá a dos o más elementos de la fila.

```
context GestorBloques inv:
  colaBloques->forAll(conjuntoBloques1,conjuntoBloques2 |
    conjuntoBloques1 <> conjuntoBloques2 implies
    conjuntoBloques1.elementos.número->excludesAll(conjuntoBloques2.
    elementos.número))
```

La expresión antes de implies (implica) es necesaria para asegurar que se ignoran los pares donde ambos elementos son el mismo bloque.

4. La colección de bloques utilizados y bloques sin usar será la colección total de bloques que constituyen los archivos.

```
context GestorBloques inv:
  allBlocks = usados->union(libres)
```

5. La colección de bloques sin usar no tendrá números de bloques duplicados.

```
context GestorBloques inv:
  libres->isUnique(aBloque | aBloque.número)
```

6. La colección de bloques utilizados no tendrá números de bloques duplicados

```
context GestorBloques inv:
  usados->isUnique(aBloque | aBloque.número)
```

El OCL también se utiliza para especificar precondiciones y poscondiciones de operaciones. Por ejemplo, considérense las operaciones que eliminan y agregan conjuntos de bloques de la cola. La notación **x@pre** indica que el objeto **x** existe *antes* de la operación; esto es, opuesto a la notación matemática expuesta con anterioridad, donde la **x** *después* de la operación es la que está designada especialmente (como **x'**).

```
context GestorBloques::eliminarBloques()
  pre: colaBloques->size() >0
  post: usados = usado@pre - colaBloques@pre->first() and
    libres = libre@pre->union(colaBloques@pre->first() and
    colaBloques = colaBloques@pre->excluding(colaBloques@pre->first)

context GestorBloques::agregarBloques(aConjuntoBloques :ConjuntoBloques)
  pre: usados->includesAll(aConjuntoBloques.elementos)
  post: (colaBloques.elementos = colaBloques.elementos@pre
    ->append(aConjuntoBloques))
    and usados = usado@pre
    and libres = libre@pre
```

El OCL es un lenguaje de modelado, pero tiene todos los atributos de un lenguaje formal. También permite la expresión de varias restricciones, precondiciones y poscon-

diciones, guardias y otras características que relacionan a los objetos representados en varios modelos UML.

28.6 EL LENGUAJE DE ESPECIFICACIÓN Z

Z es un lenguaje de especificación que ha evolucionado durante las dos décadas pasadas y hoy se utiliza ampliamente entre la comunidad de los métodos formales. El lenguaje Z aplica conjuntos tipificados, relaciones y funciones dentro del contexto de predicados lógicos de primer orden para construir *esquemas*, un medio para estructurar una especificación formal.

Referencia Web

Información detallada acerca del lenguaje Z se puede encontrar en www-users.cs.york.ac.uk/~susan/sbs/z.htm.

28.6.1 Breve panorama de la sintaxis y semántica Z

Las especificaciones Z están organizadas como un conjunto de esquemas; es decir, una estructura parecida a un recuadro que introduce variables y especifica la relación entre éstas. Un *esquema* es, en esencia, la especificación formal análoga del componente de lenguaje de programación. En la misma forma que los componentes se emplean para estructurar un sistema, los esquemas se utilizan al estructurar una especificación formal.

Un esquema describe los datos almacenados a los que un sistema accede y altera. En el contexto de Z esto se denomina el "estado". La utilización del término *estado* en Z es ligeramente diferente de la que se emplea en el resto de este libro.[6] Además, el esquema identifica las operaciones que se aplican para cambiar estado y las relaciones que ocurren dentro del sistema. La estructura genérica de un esquema asume la forma:

nombreEsquema
declaraciones
invariante

donde las declaraciones identifican las variables que comprenden el estado del sistema, y el invariante impone restricciones en la forma en la que el estado puede evolucionar. En la tabla 28.2 se presenta un resumen de la notación del lenguaje Z.

28.6.2 Un ejemplo que utiliza Z

En esta sección se utiliza el lenguaje de especificación Z para modelar el ejemplo del gestor de bloques introducido previamente en este capítulo. El siguiente ejemplo de un esquema describe el estado del gestor de bloques y el invariante de datos:

6 Recuérdese que en otros capítulos *estado* se emplea para identificar un modo de comportamiento observable externamente para un sistema.

GestorBloques

usados, libres: ℙ *BLOQUES*
FilaBloques. seq ℙ *BLOQUES*

---

usados ∩ *libres* = ∅ ∧
usados ∪ *libres* = *TodosBloques* ∧
∀*i*: **dom** *ColaBloques* • *FilaBloques i* ⊆ *usados* ∧
∀*i, j*: dom *ColaBloques* • *i* ≠ *j* => *ColaBloques i* ∩ *ColaBloques j* = ∅

TABLA 28.2 RESUMEN DE NOTACIÓN Z

La notación Z se basa en teoría de conjuntos tipificada y en lógica de primer orden. Z proporciona un constructo, llamado esquema, para describir el espacio y las operaciones de estado de una especificación. Un esquema agrupa declaraciones de variable con una lista de predicados que restringen el posible valor de una variable. En Z el esquema X se define por la forma

X
declaraciones
predicados

Las funciones globales y las constantes se definen por la forma

declaraciones
predicados

La declaración brinda el tipo de la función o constante, mientras que el predicado proporciona su valor. En esta tabla sólo se presenta un conjunto abreviado de símbolos Z.

Conjuntos:

$S : \mathbb{P} X$	S es declarado como conjunto de Xs
$x \in S$	x es miembro de S
$x \notin S$	x no es miembro de S
$S \subset T$	S es un subconjunto de T: todo miembro de S también está en T
$S \cup T$	La unión de S y T. contiene a todo miembro de S o T o ambos
$S \cap T$	La intersección de S y T· contiene a todo miembro que está tanto en S como en T.
$S \setminus T$	La diferencia de S y T· contiene a todo miembro de S, excepto a aquellos que también están en T.
$\varnothing$	Conjunto vacío. no contiene miembros.
$\{x\}$	Conjunto unitario: sólo contiene a x
$\mathbb{N}$	El conjunto de los números naturales 0, 1, 2,..
$S : \mathbb{F} X$	S es declarado como conjunto finito de X
max (S)	El máximo del conjunto no vacío de números S.

Funciones:

$f : X \rightarrowtail Y$	f es declarado como una inyección parcial de X a Y
dom f	El dominio de f el conjunto de valores x para los cuales está definida f(x)
ran f	El rango de f: el conjunto de valores que toma f(x) conforme x varía sobre el dominio de f
$f \oplus \{x \mapsto y\}$	Una función que concuerda con f, excepto que x se correlaciona con y
dom f	Una función como f, excepto que x es eliminado de su dominio.

Lógica:

$P \wedge Q$	P y Q: es verdadero si tanto P como Q son verdaderos.
$P \Rightarrow Q$	P implica Q: es verdadero si o Q es verdadero o P es falso
$\theta S' = \theta S$	Ningún componente del esquema S cambia en una operación

Como se ha señalado, el esquema consiste de dos partes. La parte sobre la línea central representa las variables del estado, mientras que la parte debajo de la línea central describe el invariante de datos. Siempre que el esquema especifica las operaciones que cambian el estado lo precede el símbolo Δ. El siguiente ejemplo de esquema describe la operación que elimina un elemento de la cola de bloques:

EliminarBloques

Δ *GestorBloques*

#GestorBloques > 0,
usados' = *usados* \ *cabeza ColaBloques* ∧
libres' = *libres* ∪ *cabeza ColaBloques* ∧
ColaBloques' = *cola ColaBloques*

La inclusión de Δ *GestorBloques* resulta en todas las variables que configuran el estado que estará disponible para el esquema *EliminarBloques* y asegura que el invariante de datos se mantendrá antes y después de que la operación se haya ejecutado.

La segunda operación, que añade una colección de bloques al final de la cola, está representada como

AñadirBloques

Δ GestorBloques
Abloques? : BLOQUES

Abloques? ⊂ *usados*
FilaBloques' = *FilaBloques* ⌢ ⟨Abloques?⟩ ∧
usados' = *usados* ∧
libres' = *libres* ∧

Por convención en Z, una variable de entrada que se lee pero no forma parte del estado termina con un signo de interrogación. Por ende, Abloques?, que actúa como parámetro de entrada, termina con un signo de interrogación.

HERRAMIENTAS DE SOFTWARE

Métodos formales

Objetivo: El objetivo de las herramientas de los métodos formales es auxiliar al equipo de software en la especificación y verificación de la corrección.

Mecánica: La mecánica de las herramientas varía. En general, ayudan en la especificación y en la prueba automática de la corrección, usualmente al definir un lenguaje especializado para la prueba de teoremas. Muchas herramientas no se comercializan y se han desarrollado con propósitos de investigación.

Herramientas representativas[7]

ACL2, desarrollada en la Universidad de Texas (www.cs.utexas.edu/users/moore/acl2/), es "tanto un lenguaje de programación con el que puede modelar sistemas de cómputo, como una herramienta para ayudarlo a probar las propiedades de dichos modelos".

EVES, desarrollada por ORA Canadá (www.ora.on.ca/eves.html), implementa el lenguaje Verdi para especificación formal y un generador de pruebas automatizado.

Una lista extensa con más de 90 herramientas de métodos formales se puede encontrar en http://www.afm.sbu.ac.uk/.

28.7 LOS DIEZ MANDAMIENTOS DE LOS MÉTODOS FORMALES

La decisión de usar métodos formales en el mundo real no es una que se tome a la ligera. Bowan y Hinchley [BOW95] han acuñado "los diez mandamientos de los métodos formales" como una guía para aquellos que están a punto de aplicar este importante enfoque de la ingeniería de software [8]

1. *Elegirás la notación apropiada.* Para elegir con eficacia de una amplia variedad de lenguajes formales de especificación, un ingeniero de software debe considerar el vocabulario del lenguaje, el tipo de aplicación que se especificará y la amplitud de uso del lenguaje.
2. *Formalizarás pero no en exceso.* Por lo general no es necesario aplicar los métodos formales en todos los aspectos de un gran sistema. Aquellos componentes cruciales para la seguridad son las primeras elecciones, seguidos por los componentes cuya falla no puede tolerarse (por razones de negocios).
3. *Estimarás los costos.* Los métodos formales tienen elevados costos de arranque El entrenamiento del equipo, la adquisición de herramientas de apoyo y la utilización de consultores redunda en altos costos al inicio. Es preciso considerar dichos costos cuando se examina el rendimiento sobre la inversión asociada con los métodos formales.

7 Las herramientas mencionadas sólo representan una muestra de esta categoría. En la mayoría de los casos, los nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

8 Este tratamiento es una versión más abreviada de [BOW95].

4. *Tendrás un experto en métodos formales a tu disposición.* El entrenamiento experto y la consultoría de seguimiento son esenciales para el éxito cuando se emplean los métodos formales por primera ocasión.
5. *No abandonarás los métodos tradicionales de desarrollo.* Es posible, y en muchos casos deseable, integrar los métodos formales con métodos convencionales u orientados a objetos (parte 2 de este libro). Cada uno tiene fortalezas y debilidades. Una combinación, si se aplica con propiedad, puede producir excelentes resultados.[9]
6. *Documentarás suficientemente.* Los métodos formales proporcionan un método conciso, sin ambigüedades y consistente para documentar los requisitos del sistema. Sin embargo, es recomendable que un comentario en lenguaje natural acompañe la especificación formal y sirva como mecanismo para reforzar la comprensión del lector acerca del sistema.
7. *No comprometerás los estándares de calidad.* "No hay nada mágico en los métodos formales" [BOW95], y por esta razón debe continuar la aplicación de otras actividades de SQA (capítulo 26) conforme los sistemas se desarrollen.
8. *No serás dogmático.* Un ingeniero de software debe reconocer que los métodos formales no regarantizan la corrección. Es posible (alguien diría, probable) que el sistema final, aun cuando se desarrolle empleando métodos formales, puede tener pequeñas omisiones, bugs menores y otros atributos que no satisfagan las expectativas.
9. *Probarás, probarás y probarás de nuevo.* La importancia de las pruebas del software se ha estudiado en los capítulos 13 y 14. Los métodos formales no absuelven al ingeniero de software de la necesidad de llevar a cabo pruebas extensivas bien planeadas.
10. *Reutilizarás.* A largo plazo, la única forma racional de reducir los costos de software y aumentar la calidad de éste es mediante la reutilización (capítulo 30). Los métodos formales no cambian esta realidad. De hecho, es posible que los métodos formales sean un enfoque apropiado cuando se han de crear componentes para librerías de reutilización.

28.6 MÉTODOS FORMALES: EL CAMINO POR RECORRER

Aunque las técnicas formales de especificación basadas matemáticamente no se emplean con amplitud en la industria, sí ofrecen ventajas sustanciales sobre las técnicas menos formales. Liskov y Bersins [LIS86] resumen esto así:

9 La ingeniería de software de sala limpia (capítulo 29) es un ejemplo de un enfoque integrado que utiliza métodos formales y métodos de desarrollo más convencionales.

> Las especificaciones formales se pueden estudiar matemáticamente, pero no las informales. Por ejemplo, un programa correcto se prueba para satisfacer sus especificaciones, o se puede probar que dos conjuntos alternativos de especificaciones son equivalentes.. Ciertas formas de incompletud o inconsistencia se pueden detectar automáticamente.

Además, la especificación formal elimina la ambigüedad y alienta el mayor rigor en las primeras etapas del proceso de ingeniería del software.

No obstante, los problemas persisten. La especificación formal se enfoca principalmente en la función y los datos La temporalidad, el control y los aspectos de comportamiento de un problema son más difíciles de representar. Además, ciertos elementos de un problema (por ejemplo, interfaces humano/máquina) se especifican mejor empleando técnicas gráficas o prototipos. Finalmente, la especificación que emplea métodos formales es más difícil de aprender que los métodos que incorporan notación UML y representan un significativo "choque cultural" para algunos profesionales del software.

28.9 RESUMEN

Los métodos formales ofrecen un cimiento para los entornos de especificación que conducen a modelos de análisis más completos, consistentes y sin ambigüedades que aquellos producidos con métodos convencionales u orientados a objetos. Las facilidades descriptivas de la teoría de conjuntos y la notación lógica permiten que un ingeniero de software cree un planteamiento claro de hechos (requisitos).

Los conceptos subyacentes que rigen los métodos formales son 1) el invariante de datos, una condición verdadera a través de la ejecución del sistema que contiene una colección de datos; 2) el estado, una representación del modo de comportamiento observable externamente de un sistema, o (en Z y lenguajes relacionados) los datos almacenados a los que un sistema tiene acceso y altera; y 3) la operación, una acción que tiene lugar en un sistema y lee o escribe datos a un estado. Una operación está asociada con dos condiciones· una precondición y una poscondición.

Las matemáticas discretas —la notación y heurísticas asociados con conjuntos y la especificación constructiva, operadores de conjuntos, operadores lógicos y sucesiones— forman la base de los métodos formales. Las matemáticas discretas se implementan en el contexto de los lenguajes formales de especificación, tales como OCL y Z. Estos lenguajes tienen dominios tanto sintáctico como semántico. El dominio sintáctico utiliza una simbología alineada de manera cercana con la notación de conjuntos y el cálculo de predicados. El dominio semántico permite que el lenguaje exprese los requisitos en una forma concisa.

La decisión de usar métodos formales debe considerar los costos de arranque, así como los cambios culturales asociados con una tecnología radicalmente diferente. En la mayoría de las instancias, los métodos formales tienen mayores rendimientos respecto de los sistemas cruciales para la seguridad y los negocios.

Referencias

[BOW95] Bowan, J. P., y M. G. Hinchley, "Ten Commandments of Formal Methods", en *Computer*, vol. 28, núm. 4, abril de 1995

[GRI95] Gries, D., y F. B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, 1993.

[GUT93] Guttag, J. V., y J. J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[HAL90] Hall, A., "Seven Myths of Formal Methods", en *IEEE Software*, septiembre de 1990, pp. 11-20.

[HOR85] Hoare, C. A.R., *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[ISO02] *Z formal Specification Notation-Syntax, Type System and Semantics*, ISO/IEC 13568 2002, Intl. Standards Organization, 2002

[JON91] Jones, C. B., *Systematic Software Development Using VDM*, 2a. ed., Prentice-Hall, 1991.

[LIS86] Liskov, B. H., y V. Berzins, "An Appraisal of Program Specifications", en *Software Specification Techniques*, N. Gehani y A. T. McKittrick (eds.), Addison-Wesley, 1986, p. 3.

[MAR94] Marciniak, J. J. (ed.) *Encyclopedia of Software Engineering*, Wiley, 1994

[OMG03] "Object Constraint Language Specification", en *Unified Modeling Language*, v2.0, Object Management Group, septiembre de 2003, se puede descargar de www.omg.org

[ROS95] Rosen, K. H., *Discrete Mathematics and Its Applications*, 3a. ed., McGraw-Hill, 1995

[SPI88] Spivey, J. M., *Understanding Z: A Specification Language and Its Formal Semantics*, Cambridge University Press, 1988

[SPI92] Spivey, J. M., *The Z Notation: A Reference Manual*, Prentice-Hall, 1992.

[WIL87] Wiltala, S. A., *Discrete Mathematics: A Unified Approach*, McGraw-Hill, 1987

[WIN90] Wing, J. M., "A Specifier's Introduction to Formal Methods", *Computer*, vol. 23, núm. 9, septiembre de 1990, pp. 8-24.

[YOU94] Yourdon, E., "Formal Methods", *Guerrilla Programmer*, Cutter Information Corp., octubre de 1994.

Problemas y puntos a considerar

28.1. Revisar los tipos de deficiencias asociadas con los enfoques menos formales para la ingeniería del software de la sección 28.1.1. Ofrecer tres ejemplos de cada uno a partir de la experiencia propia.

28.2. Los beneficios de las matemáticas como mecanismo de especificación se han tratado extensamente en este capítulo. ¿Existe algún aspecto negativo?

28.3. Usted ha sido asignado a un equipo que está desarrollando software para un fax módem. Su tarea es desarrollar la parte de "directorio" de la aplicación. La función directorio permite que almacenen hasta *MaxNombres* personas junto con los asociados de nombres de compañía, números de fax y otra información relacionada. Empleando lenguaje natural defina

a) El invariante de datos.
b) El estado.
c) Las operaciones probables.

28.4. Usted ha sido asignado a un equipo que está desarrollando software, llamado *Duplicador de memoria*, que ofrece mayor memoria aparente para una PC que la memoria física. Esto se logra al identificar, recopilar y reasignar bloques de memoria que se han asignado a una aplicación existente pero que no se utilizan. Los bloques no utilizados se reasignan a las aplicaciones que requieren memoria adicional. Con las suposiciones apropiadas y el uso de lenguaje natural, defina

a) El invariante de datos.
b) El estado.
c) Las operaciones probables.

28.5. Desarrolle una especificación constructiva para un conjunto que contiene duplas de números naturales de la forma (x, y, z) tales que la suma de x y y es igual a z.

28.6. El instalador para una aplicación basada en PC determina primero si están presentes un conjunto aceptable de hardware y recursos del sistema. Verifica la configuración del hardware para determinar si están presentes varios dispositivos (de muchos posibles dispositivos) y determina si ya están instaladas versiones específicas de software y controladores del sistema. ¿Qué operador de conjunto se podría usar para lograr esto? Ofrecer un ejemplo en este contexto.

28.7. Intentar desarrollar una expresión utilizando operadores lógicos y de conjunto para el siguiente enunciado: "Para toda x e y, si x es el padre de y y y es el padre de z, entonces x es el abuelo de z. Todos tienen un padre." Sugerencia: emplear las funciones $P(x, y)$ y $G(x, z)$ para representar las funciones padre y abuelo, respectivamente.

28.8. Desarrollar una especificación constructiva de conjunto del conjunto de pares donde el primer elemento de cada par es la suma de dos números naturales distintos de cero, y el segundo elemento es la diferencia entre los mismos números. Ambos números deben estar entre 100 y 200, inclusive.

28.9. Desarrollar una descripción matemática para el estado y el invariante de datos del problema 28.3. Refinar esta descripción en el lenguaje de especificación OCL o Z.

28.10. Desarrollar una descripción matemática para el estado y el invariante de datos del problema 28.4. Refinar esta descripción en el lenguaje de especificación OCL o Z.

28.11. Mediante la notación OCL o Z presentadas en las tablas 28.1 o 28.2, seleccionar alguna parte del sistema de seguridad *HogarSeguro* descrito previamente en este libro e intentar describirla con OCL o Z.

28.12. Empleando una o más de las fuentes de información que aparecen en las referencias de este capítulo o en "Otras lecturas y fuentes de información", desarrollar una presentación de media hora acerca de la sintaxis y la semántica básicas de un lenguaje formal de especificación distintos a OCL o Z.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Además de los libros empleados como referencias en este capítulo, durante la década pasada se publicaron numerosos libros acerca de temas de métodos formales. A continuación se presenta una lista de algunos de los más útiles:

Bowan, J., *Formal Specification and Documentation using Z: A Case Study Approach*, International Thomson Computer Press, 1996.

Casey, C., *A Programming Approach to Formal Methods*, McGraw-Hill, 2000.

Clark, T., *et al.* (eds.), *Object Modeling with OCL*, Springer-Verlag, 2002.

Cooper, D., y R. Barden, *Z in Practice*, Prentice-Hall, 1995.

Craigen, D., S. Gerhart y T. Ralston, *Industrial Application of Formal Methods to Model, Design and Analyze Computer Systems*, Noyes Data Corp., 1995.

Harry, A., *Formal Methods Fact File: VDM y Z*, Wiley, 1997.

Hinchley, M., y J. Bowan, *Applications of Formal Methods*, Prentice-Hall, 1995.

Hinchley, M., y J. Bowan, *Industrial Strenght Formal Methods*, Academic Press, 1997.

Hussmann, H., *Formal Foundations for Software Engineering Methods*, Springer-Verlag, 1997.

Jacky, J., *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997.

Monin, F., y M. Hinchley, *Understanding Formal Methods*, Springer-Verlag, 2003.

Rann, D., J. Turner y J. Whitworth, *Z: A Beginner's Guide*, Chapman and Hall, 1994.

Ratcliff, B., *Introducing Specification Using Z: A Practical Case Study Approach*, McGraw-Hill, 1994.

Sheppard, D., *An Introduction to Formal Specification with Z and VDM*, McGraw-Hill, 1995.

Warner, J., y A. Kleppe, *Object Constraint Language*, Addison-Wesley, 1998

Dean (*Essence of Discrete Mathematics*, Prentice-Hall, 1996), Gries y Schneider [GRI93] y Lipschultz y Lipson (*2000 Solved Problems in Discrete Mathematics*, McGraw-Hill, 1991) presentan información útil para quienes deben aprender más acerca de las bases de los métodos formales.

En Internet hay disponible una amplia variedad de fuentes de información acerca de los métodos formales. Una lista actualizada de referencias en la World Wide Web se puede encontrar en el sitio Web SEPA
http://www.mhhe.com/pressman.

CAPÍTULO

29 INGENIERÍA DEL SOFTWARE DE SALA LIMPIA

CONCEPTOS CLAVE

El uso integrado del modelado convencional de ingeniería del software (y posiblemente los métodos formales), la verificación de programas (pruebas de corrección) y el SQA estadístico se han combinado en una técnica que puede conducir a software de calidad extremadamente alta. La *ingeniería del software de sala limpia* es un enfoque que resalta la necesidad de construir la corrección en el software conforme se desarrolla. En lugar del clásico ciclo de análisis, diseño, código, prueba y depuración, el enfoque de sala limpia sugiere un punto de vista diferente [LIN94]:

> La filosofía detrás de la ingeniería del software de sala limpia consiste en evitar la dependencia de costosos procesos de eliminación de defectos al escribir los incrementos de código correctos la primera vez y verificar su corrección antes de ponerlo a prueba. Su modelo de proceso incorpora la certificación estadística de calidad de los incrementos de código conforme se acumulan en el sistema.

En muchos aspectos, el enfoque de sala limpia eleva la ingeniería de software a otro nivel. Al igual que los métodos formales presentados en el capítulo 28, el proceso de sala limpia destaca el rigor en la especificación y el diseño, y la verificación formal de cada elemento de diseño mediante pruebas de corrección con bases matemáticas. Al extender el enfoque adoptado en los métodos formales, el enfoque de sala limpia también resalta las técnicas para el control estadístico de la calidad, incluso pruebas que se basan en el uso anticipado del software por parte de los clientes.

Cuando el software falla en el mundo real, abundan los peligros inmediatos y a largo plazo. Los peligros se relacionan con la seguridad humana, las pérdidas económicas o la operación efectiva del negocio y la infraestructura social. La ingeniería del software de sala limpia es un modelo de proceso que elimina los defectos antes de que puedan generar peligros serios.

UN VISTAZO RÁPIDO

¿Qué es? ¿Cuántas veces se ha escuchado la expresión: "Hazlo bien la primera vez"? Esa es la filosofía primordial de la ingeniería del software de sala limpia: un proceso que destaca la verificación matemática de la corrección antes de que comiencen la construcción del programa y la certificación de la confiabilidad del software como parte de la actividad de pruebas. El rasgo fundamental es tasas de falla extremadamente bajas que serían difíciles o imposibles de lograr empleando métodos menos formales.

¿Quién lo hace? Un ingeniero de software especialmente entrenado.

¿Por qué es importante? Los errores implican la relaboración. Ésta lleva tiempo y aumenta los costos. ¿No sería agradable si se pudiese reducir sustancialmente el número de errores (bugs) introducidos conforme el software se dise-

ña y construye? Esa es la premisa de la ingeniería del software de sala limpia.

¿Cuáles son los pasos? Los modelos de análisis y diseño se crean empleando una representación de estructura de cajas. Una "caja" encapsula el sistema (o a cierto aspecto del sistema) en un grado específico de abstracción. La verificación de la corrección se aplica una vez que está completo el diseño de la estructura de cajas. Una vez verificada la corrección de cada estructura de caja comienzan las pruebas estadísticas de utilización. El software se prueba al definir un conjunto de escenarios de utilización, al determinar la probabilidad de utilización para cada escenario y luego definir pruebas aleatorias que concuerden con las probabilidades. Los registros de error resultantes se analizan para permitir el cálculo matemático de la fiabilidad proyectada del componente de software.

¿Cuál es el producto obtenido? Se desarrollan especificaciones de caja negra, caja de estado y caja transparente. Los resultados de las pruebas de corrección formales y de las pruebas estadísticas de utilización se registran.

¿Cómo puedo estar seguro de que lo he hecho correctamente? La prueba formal de corrección se aplica a la especificación de estructura de cajas. Las pruebas estadísticas de utilización ejercitan los escenarios de utilización para garantizar que los errores en la funcionalidad de usuario se descubren y corrigen. Los datos de prueba se utilizan para proporcionar un indicio de la fiabilidad del software.

29.1 EL ENFOQUE DE SALA LIMPIA

La filosofía de la "sala limpia" en las tecnologías de fabricación de hardware en realidad es bastante simple: es eficaz en cuanto a costo y tiempo para establecer un enfoque de fabricación que evite la introducción de defectos de producción. Más que fabricar un producto y luego trabajar para eliminar los defectos, el enfoque de sala limpia demanda la disciplina requerida para eliminar los errores en la especificación y el diseño y luego fabricarlo en una forma "limpia".

Mills, Dyer y Linger [MIL87] propusieron, durante el decenio de 1980, originalmente la filosofía de sala limpia para la ingeniería del software. Aunque las primeras experiencias con este enfoque disciplinado respecto al trabajo de software fueron significativamente prometedoras [HAU94], no obtuvo gran difusión. Henderson [HEN95] sugiere tres posibles razones:

1. Una creencia de que la metodología de sala limpia es demasiado teórica, demasiado matemática y demasiado radical para aplicarla en el desarrollo de software real.
2. No aboga por una prueba unitaria de parte de los desarrolladores, sino que la sustituye con la verificación de la corrección y el control estadístico de la calidad, conceptos que representan una gran desviación de la forma en la que se desarrolla actualmente la mayoría del software.
3. La madurez de la industria de desarrollo del software. La utilización de los procesos de sala limpia requiere la rigurosa aplicación de procesos definidos en todas las fases del ciclo de vida. Dado que gran parte de la industria conti-

núa operando en grados relativamente bajos de madurez del proceso, los ingenieros de software no han estado listos para aplicar las técnicas de sala limpia.

A pesar de los elementos de verdad en cada una de estas preocupaciones, los beneficios potenciales de la ingeniería del software de sala limpia superan con mucho la inversión requerida para superar la resistencia cultural ubicada en el centro de estas preocupaciones

> "La única forma de que en un programa ocurran los errores es que un autor los coloque ahí. No se conocen otros mecanismos... La práctica correcta busca evitar la inserción de errores y, cuando se falla al respecto, eliminarlos antes de probarlo o cualquiera otra forma de ejecutar el programa."
>
> **Harlan Mills**

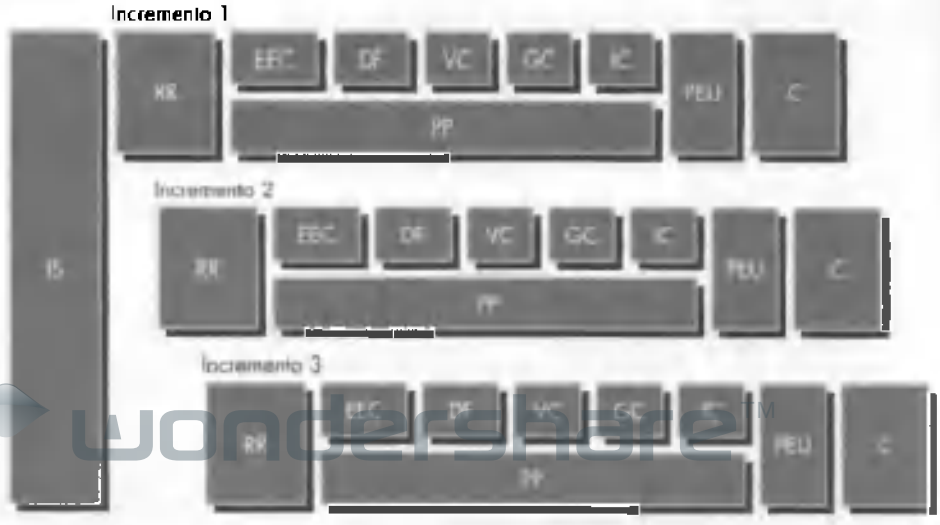
29.1.1 La estrategia de sala limpia

El enfoque de sala limpia utiliza una versión especializada del modelo de proceso incremental (capítulo 3). Mediante pequeños equipos de software independientes se desarrolla una "línea de incrementos de software" [LIN94]. Conforme cada incremento se certifica se integra en el todo Por ende, la funcionalidad del sistema crece con el tiempo.

En la figura 29 1 se ilustra la sucesión de las tareas de sala limpia para cada incremento. Los requisitos globales del sistema o producto se desarrollan empleando los

FIGURA 29.1

Modelo del proceso de sala limpia.



IS = ingeniería de sistemas
RR = recopilación de requisitos
EEC = especificación de estructura de cajas
DF = diseño formal
VC = verificación de corrección
GC = generación de código
IC = inspección de código
PEU = prueba estadística de uso
C = certificación
PP = planeación de pruebas

métodos de ingeniería del software estudiados en el capítulo 6. La línea de incrementos de sala limpia se inicia una vez que la funcionalidad se ha asignado al elemento de software del sistema. Se producen las siguientes tareas:

¿Cuáles son las principales tareas llevadas a cabo como parte de la ingeniería del software de sala limpia?

Planificación del incremento. Se desarrolla un plan de proyecto que adopta la estrategia incremental. Se crean la funcionalidad de cada incremento, su tamaño proyectado y un plan de desarrollo de sala limpia. Se debe tener especial cuidado para asegurar que los incrementos certificados se integrarán en forma oportuna.

Recopilación de requisitos. Mediante técnicas similares a las introducidas en el capítulo 7 se elabora una descripción más detallada de los requisitos del cliente (para cada incremento).

Especificación de la estructura de cajas. Se utiliza un método de especificación que emplea *estructuras de caja* [HEV93] para describir la especificación funcional. Para ajustarse a los principios de análisis operativo tratados en los capítulos 5 y 7, las estructuras de cajas "aíslan y separan la definición creativa de comportamiento, datos y procedimientos en cada grado de refinamiento".

Referencia Web

Una variedad de información y recursos para la aplicación del software de sala limpia se puede encontrar en www.cleansoft.com.

Diseño formal. Empleando el enfoque de estructura de cajas el diseño de sala limpia es una extensión natural y uniforme de la especificación. Aunque es posible establecer una distinción clara entre las dos actividades, la especificaciones (llamadas *cajas negras*) son iterativamente refinadas (dentro de un incremento) para volverse análogas a los diseños arquitectónico y al nivel de componente (llamados *cajas de estado* y *cajas transparentes*, respectivamente).

Verificación de la corrección. El equipo de sala limpia lleva a cabo una serie de rigurosas actividades de verificación de la corrección en el diseño y luego en el código. La verificación (secciones 29.3 y 29.4) comienza con la estructura de caja de nivel superior (especificación) y se mueve hacia el detalle de diseño y el código. El primer nivel de verificación de corrección ocurre al aplicar un conjunto de "preguntas de corrección" [LIN88]. Si éstas no demuestran que la especificación es correcta se emplean métodos más formales (matemáticos) en la verificación.

Generación de código, inspección y verificación. Las especificaciones de estructura de caja, representadas en un lenguaje especializado, se traducen al lenguaje de programación apropiado. Entonces se utilizan comprobaciones manuales estándar o técnicas de inspección (capítulo 26) que garantizan la conformidad semántica del código y las estructuras de cajas, así como la corrección sintáctica del código. Luego se lleva a cabo la verificación de la corrección para el código fuente

> *"La ingeniería del software de sala limpia logra el control estadístico de la calidad sobre el desarrollo del software al separar estrictamente el proceso de diseño del proceso de prueba en una línea de desarrollo incremental de software."*
>
> **Harlan Mills**

Planificación de pruebas estadísticas. Se analiza el uso proyectado del software y se planifica y diseña un conjunto de casos de prueba que ejercitan una "dis-

CONSEJO

La sala limpia destaca las pruebas que ejercitan la forma en que el software es realmente utilizado. Los casos de uso ofrecen una introducción al proceso de planeación de pruebas.

tribución de probabilidad" de utilización (sección 29.4). Como se muestra en la figura 29.1, esta actividad de sala limpia se lleva a cabo en paralelo con la especificación, la verificación y la generación de código.

Prueba estadística de la utilización. Como se recordará, la prueba exhaustiva del software de computadora es imposible (capítulo 14), por lo que siempre es necesario diseñar un número finito de casos de prueba. Las técnicas estadísticas de utilización [POO88] ejecutan una serie de pruebas derivadas de una muestra estadística (la distribución de probabilidad anotada previamente) de todas las posibles ejecuciones de programa por parte de todos los usuarios a partir de una población objetivo (sección 29.4).

Certificación. Una vez completadas la verificación, la inspección y las pruebas de utilización (y de que todos los errores hayan sido corregidos), el incremento se certifica listo para la integración.

Al igual que otros modelos de proceso de software tratados en otras partes de este libro, el proceso de sala limpia se apoya sustancialmente en la necesidad de producir análisis de alta calidad y modelos de diseño. Como se verá más adelante en este capítulo, la notación de estructura de cajas es simplemente otra forma en la que un ingeniero de software puede representar los requisitos y el diseño. La distinción real del enfoque de sala limpia es que la verificación formal se aplica a los modelos de ingeniería.

29.1.2 ¿Qué hace diferente a la sala limpia?

Dyer [DYE92] alude a las diferencias del enfoque de sala limpia cuando define el proceso:

> La sala limpia representa el primer intento práctico de someter el proceso de desarrollo de software al control estadístico de la calidad con una estrategia bien definida para la mejora continua de los procesos. Con el fin de alcanzar esta meta se definió un ciclo de vida único de sala limpia, el cual se enfoca en la ingeniería del software basada en matemáticas para corregir diseños de software y en la prueba de software basada en estadísticas para la certificación de la fiabilidad del software.

PUNTO CLAVE

Los más importantes características distintivas de la sala limpia son la prueba de la corrección y los [illegible] estadísticos de utilización.

La ingeniería del software de sala limpia difiere de los métodos convencionales y orientados a objetos de la ingeniería del software porque:

1. Emplea en forma explícita el control estadístico de la calidad.
2. Verifica las especificaciones del diseño utilizando una prueba de corrección basada matemáticamente.
3. Implementa técnicas de prueba con una alta probabilidad de descubrir errores de alto impacto.

Obviamente, el enfoque de sala limpia aplica la mayoría, si no todos, los principios y conceptos básicos de la ingeniería del software presentados a lo largo de este libro.

Los buenos análisis y procedimientos de diseño son esenciales si se quiere obtener alta calidad. Pero la ingeniería de sala limpia difiere de las prácticas del software convencional porque le resta importancia (algunos dirían, elimina) al papel de la prueba unitaria y la depuración y reduce drásticamente (o elimina) la cantidad de pruebas que realiza el desarrollador del software.[1]

En el desarrollo de software convencional los errores se aceptan como un hecho ineludible. Puesto que los errores están condenados a ser inevitables, cada componente de programa debe probarse en forma individual (para descubrir los errores) y luego depurarse (para eliminar los errores). Cuando finalmente se libera el software, durante su utilización se descubren todavía más defectos y comienza otro ciclo de prueba y depuración. La reelaboración asociada con dichas actividades es costosa y consume mucho tiempo. Peor aún, puede resultar degenerativa: la corrección de errores tal vez conduzca (¡inadvertidamente!) a la introducción de más errores todavía

> "Un aspecto curioso de la vida es que si la rehúsas en absoluto a aceptar lo mejor con frecuencia lo obtienes."
>
> **W. Somerset Maugham**

En la ingeniería del software de sala limpia la prueba unitaria y la depuración se sustituyen verificando la corrección y las pruebas basadas en estadísticas. Dichas actividades, combinadas con la conservación de registros necesaria para la mejora continua, hacen que el enfoque de sala limpia sea único.

29.2 ESPECIFICACIÓN FUNCIONAL

Sin importar el método de análisis elegido, se aplican los principios de análisis operativo presentados en el capítulo 7. Los datos, las funciones y el comportamiento se modelan. Los modelos que se obtienen deben particionarse (refinarse) para proporcionar cada vez mayor detalle. El objetivo global es moverse desde una especificación (o modelo) que capture la esencia de un problema hasta una especificación que ofrezca sustanciales detalles de implementación.

La ingeniería del software de sala limpia cumple con los principios de análisis operativo empleando un método llamado *especificación de estructura de cajas*. Una "caja" encapsula al sistema (o algún aspecto de éste) en algún grado de detalle. Por medio de un proceso de elaboración o refinamiento en niveles, las cajas se refinan en una jerarquía donde cada una tiene transparencia referencial. Esto es: "el contenido de información de cada especificación de caja es suficiente para definir su refinamiento, sin depender de la implementación de alguna otra caja" [LIN94]. Esto le permite al analista partir un sistema jerárquicamente, moverse desde la representación esencial en la parte superior hacia un detalle específico de la implementación en el fondo. Se utilizan tres tipos de cajas

1 La prueba la realiza un equipo de **prueba independiente**

¿Cómo se logra el refinamiento como parte de una especificación de estructura de cajas?

Caja negra. La caja negra especifica el comportamiento de un sistema o de una parte de éste. El sistema (o parte de él) responde a estímulos específicos (eventos) al aplicar un conjunto de reglas de transición que correlacionan el estímulo con una respuesta.

Caja de estado. La caja de estado encapsula los datos de estado y servicios (operaciones) en una forma análoga a los objetos. En esta visión de especificación se representan las entradas a la caja de estado (estímulos) y las salidas (respuestas). La caja de estado también representa la "historia de estímulo" de la caja negra, esto es los datos encapsulados en la caja de estado que deben conservarse entre las transiciones implicadas.

Caja transparente. Las funciones de transición que implica la caja de estado se definen en la caja transparente. Enunciado de manera simple, una caja transparente contiene el diseño de procedimiento para la caja de estado

La figura 29.2 ilustra el enfoque de refinamiento empleando la especificación de estructura de cajas. Una caja negra (CN_1) define respuestas para un conjunto completo de estímulos. CN_1 se puede refinar en un conjunto de cajas negras, $CN_{1.1}$ hasta $CN_{1.n}$, cada una de las cuales aborda una clase de comportamiento. El refinamiento continúa hasta que se identifica una clase cohesiva de comportamiento (por ejemplo, $CN_{1.1.1}$). Entonces se define una caja de estado ($CE_{1.1.1}$) para la caja negra ($CN_{1.1.1}$) En este caso, $CE_{1.1.1}$ contiene todos los datos y servicios que se requieren para implementar el comportamiento que define $CN_{1.1.1}$. Finalmente, $CE_{1.1.1}$ se refina en cajas transparentes ($CT_{1.1.1}$ y se especifican los detalles de diseño del procedimiento

PUNTO CLAVE

El refinamiento de la estructura de cajas y la verificación de la corrección ocurren simultáneamente

Conforme ocurre cada uno de estos pasos de refinamiento, también ocurre la verificación de la corrección. Las especificaciones de caja de estado se verifican para

FIGURA 29.2

Refinamiento de estructura de cajas.

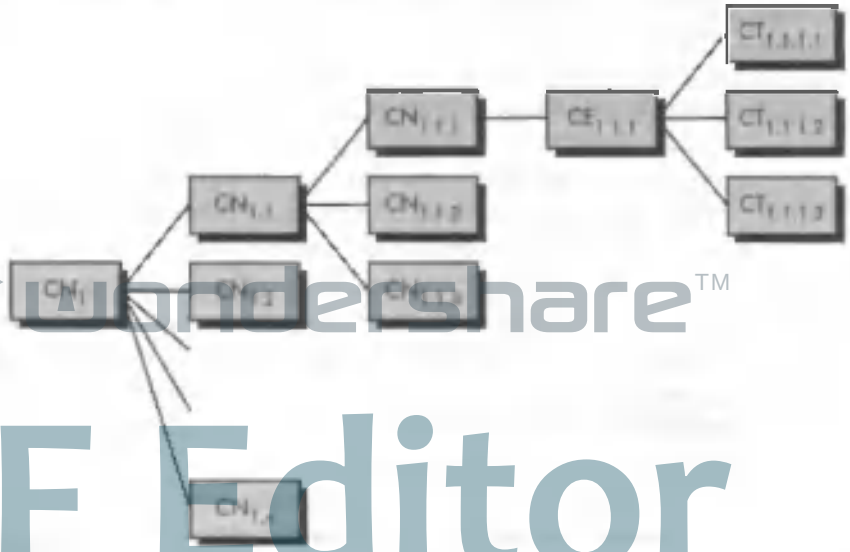
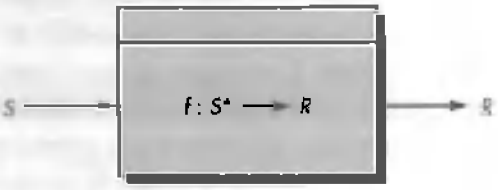


FIGURA 29.3
Especificación de caja negra.



garantizar que cada uno concuerda con el comportamiento definido por la especificación padre de caja negra. De igual modo, las especificaciones de caja transparente se verifican contra la caja de estado padre.

Se debe notar que los métodos de especificación basados en lenguajes como OCL o Z (capítulo 28) es posible usarlos en conjunción con el enfoque de especificación de estructura de cajas. El único requisito es que cada grado de especificación sea verificable formalmente.

29.2.1 Especificación de caja negra

Una *especificación de caja negra* describe una abstracción, estímulos y respuesta mediante la notación mostrada en la figura 29.3 [MIL88]. La función *f* se aplica a una sucesión, *S**, de entradas (estímulos), *S*, y las transforma en una salida (respuesta), *R*. Respecto a componentes de software simples *f* puede ser una función matemática, pero, en general, *f* se describe usando lenguaje natural (o un lenguaje formal de especificación).

Muchos de los conceptos introducidos para los sistemas orientados a objetos también son aplicables respecto de la caja negra. Las abstracciones de datos y las operaciones que manipulan dichas abstracciones las encapsula la caja negra. Al igual que una jerarquía de clase, la especificación de caja negra puede exhibir el uso de jerarquías en las cuales las cajas de nivel inferior heredan las propiedades de cajas superiores en la estructura de árbol.

FIGURA 29.4
Especificación de caja de estado.

29.2.2 Especificación de caja de estado

La *caja de estado* es "una generalización simple de una máquina de estado" [MIL88] Si se recuerda la descripción del modelado de comportamiento y los diagramas de estado del capítulo 8, un estado es algún modo observable de comportamiento del sistema. Conforme ocurren los procesamientos, un sistema responde a los eventos (estímulos) mediante la realización de transiciones desde el estado actual hasta cierto estado nuevo. Conforme se realiza la transición es posible que ocurra una acción. La caja de estado utiliza una abstracción de datos para determinar la transición hacia el siguiente estado y la acción (respuesta) que ocurrirá como consecuencia de la transición.

Según se muestra en la figura 29 4, la caja de estado incorpora una caja negra. El estímulo, S, que ingresa a la caja negra llega desde alguna fuente externa y un conjunto de estados internos del sistema, T. Mills [MIL88] proporciona una descripción matemática de la función, f, de la caja negra contenida dentro de la caja de estado·

$$g : S^* \times T^* \rightarrow R \times T$$

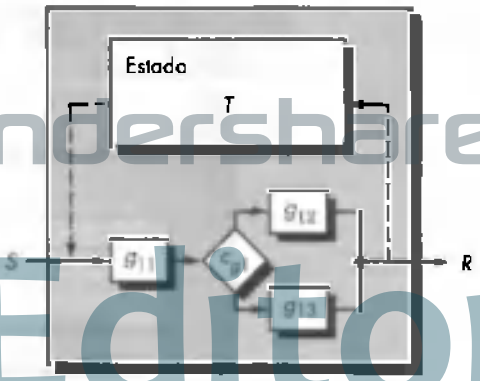
donde g es una subfunción ligada a un estado específico, t. Cuando se consideran en conjunto, los pares de subfunciones de estado (t, g) definen la función f de caja negra.

29.2.3 Especificación de caja transparente

La especificación de caja transparente está cercanamente relacionada con el diseño de procedimientos y la programación estructurada (capítulo 11). En esencia, la subfunción g dentro de la caja de estado se sustituye con las estructuras de programación estructurada que implementa g

Como ejemplo, considérese la caja transparente que se muestra en la figura 29 5. La caja negra, g, de la figura 29.4 se sustituye con una sucesión de estructuras que incorpora una condicional. Estas estructuras, a su vez, se refinan en cajas transparentes de nivel inferior conforme procede el refinamiento en niveles.

FIGURA 29.5
Especificación de caja transparente.



Es importante observar que puede demostrarse la corrección de la especificación de procedimiento descrita en la jerarquía de caja transparente. Este tema se considera en la sección siguiente.

29.3 DISEÑO DE SALA LIMPIA

El enfoque de diseño utilizado en la ingeniería del software de sala limpia utiliza con amplitud la filosofía de programación estructurada. Pero, en este caso, la programación estructurada se aplica mucho más rigurosamente.

Las funciones de procesamiento básico (descritas durante las primeras etapas del refinamiento de la especificación) se refinan utilizando una "expansión progresiva de funciones matemáticas en estructuras de conectivos lógicos [por ejemplo, *if-then-else*] y subfunciones, donde la expansión [se] realiza hasta que todas las subfunciones identificadas puedan establecerse directamente en el lenguaje de programación usado para la implementación" [DYE92]

El enfoque de programación estructurada se emplea con eficacia para refinar la función, ¿pero qué hay acerca del diseño? Aquí se involucran varios conceptos fundamentales de diseño (capítulos 5 y 9). Los datos de programa se encapsulan como un conjunto de abstracciones que atienden las subfunciones. Los conceptos de encapsulado de datos, ocultamiento de información y clasificación de datos se aprovechan para crear el diseño de datos

29.3.1 Refinamiento y verificación del diseño

Cada especificación de caja transparente representa el diseño de un procedimiento (subfunción) necesario para lograr una transición de caja de estado. Con la caja transparente las estructuras de programación estructurada y el refinamiento progresivo se utilizan como se ilustra en la figura 29.6. Una función de programa, *f*, se refina en una sucesión de subfunciones *g* y *h*. Éstas, a su vez, se refinan en estructuras condicionales (*if-then-else* y *do-while*). El refinamiento adicional ilustra la continuación del refinamiento lógico.

¿Qué condiciones son aplicables para probar que son adecuadas las estructuras estructuradas?

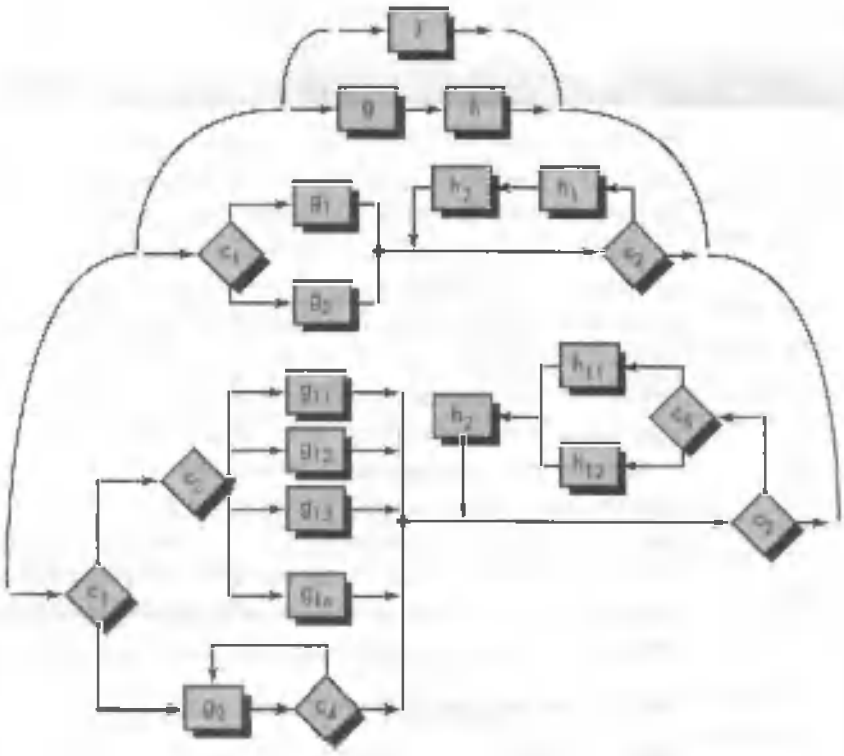
En cada nivel de refinamiento el equipo de sala limpia[2] realiza una verificación formal de corrección. Esto se logra anexando un conjunto de condiciones de corrección genéricas a las estructuras de programación estructurada. Si una función *f* se expande en una sucesión *g* y *h*, la condición de corrección para cualquier entrada a *f* es

- *¿g* seguida de *h* hace *f*?

Cuando una función p se refina en un condicional de la forma if <*c*> then *q*, else *r* (si <*c*> entonces *q*, de otro modo *r*), la condición de corrección para cualquier entrada a *p* es

2 Puesto que el equipo completo está involucrado en el proceso de verificación, es menos probable que se cometa un error al realizar la verificación

FIGURA 29.6
Refinamiento progresivo.



CONSEJO

Si el lector se limita sólo a las estructuras estructuradas conforme desarrolla un diseño de procedimiento, la prueba de corrección es directa. Si se violan las estructuras las pruebas de corrección son difíciles o imposibles.

- ¿Siempre que la condición <c> es verdadera, *q* hace *p*; y siempre que <c> es falsa, *r* hace *p*?

Cuando la función *m* se refina como un bucle las condiciones de corrección para cualquier entrada a *m* son

- ¿La terminación está garantizada?
- ¿Siempre que <c> es verdadera, *n* seguida de *m* hace *m*; y siempre que <c> es falsa, soslayar el bucle todavía hace *m*?

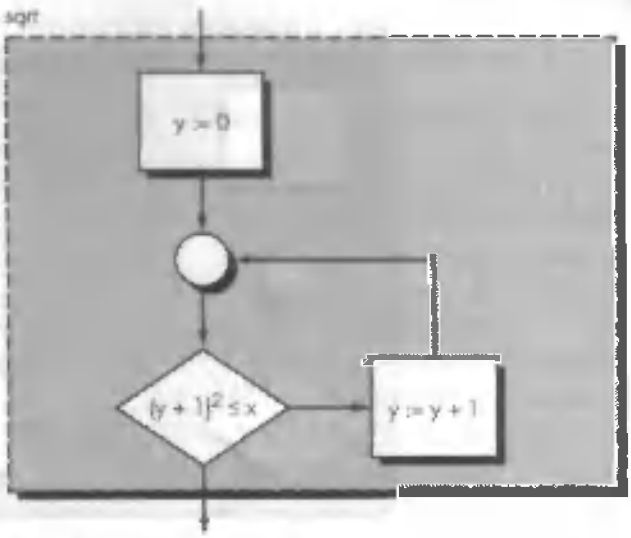
Cada vez que una caja transparente se refina al siguiente nivel de detalle se aplican dichas condiciones de corrección.

Es importante señalar que la utilización de estructuras de programación estructurada restringe el número de pruebas de corrección que se deben realizar. Una sola condición se verifica para las sucesiones; dos condiciones se prueban para *if-then-else*; y tres condiciones se verifican para los bucles

La verificación de corrección de un diseño de procedimiento se ilustra empleando un ejemplo simple que introdujeron Linger, Mills y Witt [LIN79]. El objetivo es diseñar y verificar un pequeño programa que encuentra la parte entera, *y*, de una

FIGURA 29.7

Cálculo de la parte entera de una raíz cuadrada [LIN79].



raíz cuadrada de un entero dado, x. El diseño de procedimiento se representa en la figura 29.7 empleando el diagrama de flujo.

Verificar la corrección de este diseño requiere definir las condiciones de entrada y salida como se indica en la figura 29.8. La condición de entrada advierte que x debe ser mayor que o igual a 0. La condición de salida requiere que x permanezca inalterada y que y satisfaga la expresión indicada en la figura. Probar la corrección del diseño requiere comprobar que, en todos los casos, son verdaderas las condiciones *inicio*, *bucle*, *cuenta*, *sí* y *salida* que se muestran en la figura 29.8. En ocasiones, a éstas se les llama *subpruebas*.

¡PUNTO CLAVE

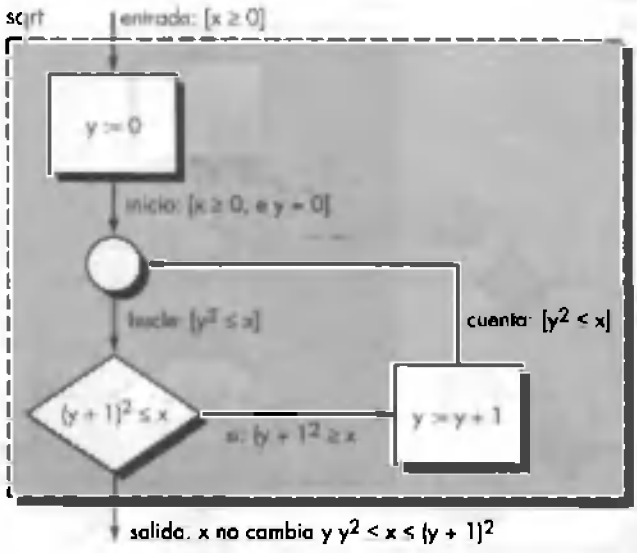
Probar que un diseño es correcto requiere, primero, identificar todas las condiciones y luego probar que cada una toma el valor booleano adecuado. A estas condiciones se les llama subpruebas.

1. La condición inicio demanda que [$x \geq 0$ e $y = 0$]. Con base en los requisitos del problema se supone que la condición de entrada es correcta.[3] En consecuencia, se satisface la primera parte de la condición inicio, $x \geq 0$. Según el diagrama de flujo, el enunciado que precede inmediatamente a la condición inicio establece $y = 0$. Por lo tanto, la segunda parte de la condición inicio también se satisface. En consecuencia, inicio es verdadera.
2. La condición bucle se puede encontrar en una de dos formas posibles: 1) directamente a partir de inicio (en este caso, la condición bucle se satisface directamente) o por medio del flujo de control que pasa a través de la condición cuenta. Dado que la condición cuenta es idéntica a la condición bucle, bucle es verdadera sin importar la trayectoria de flujo que conduce a ella.

3 En este contexto, un valor negativo para la raíz cuadrada no tiene significado.

FIGURA 29.8

Prueba de que el diseño es correcto. Diseño con subpruebas.



3. La condición cuenta se encuentra sólo después de que el valor de y aumenta en 1. Además, la trayectoria del flujo de control que conduce a cuenta se puede invocar sólo si la condición sí también es verdadera. Por lo tanto, si $(y + 1)^2 \leq x$, se sigue que $y^2 \leq x$. La condición cuenta se satisface.
4. La condición sí se prueba en la lógica condicional mostrada. Por lo tanto, la condición sí debe ser verdadera cuando el flujo de control se mueve a lo largo de la trayectoria mostrada.
5. La condición salida demanda primero que x no haya cambiado. Un examen del diseño indica que x no aparece en ningún sitio a la izquierda de un operador de asignación. No existe llamado de función que use x. Por lo tanto, no cambia. Dado que la prueba condicional $(y + 1)^2 \leq x$ debe fallar para alcanzar la condición salida, se sigue que $(y + 1)^2 \leq x$. Además, la condición bucle todavía debe ser verdadera (es decir, $y^2 \leq x$). En consecuencia, se pueden combinar $(y + 1)^2 > x$ y $y^2 \leq x$ para satisfacer la condición salida.

Además se debe garantizar que el bucle termina. Un examen de la condición *bucle* indica que, puesto que *y* aumenta y $x > 0$, el bucle a la larga debe terminar.

Los cinco pasos apenas anotados prueban la corrección del diseño del algoritmo indicado en la figura 29.7. Ahora se tiene certeza de que el diseño, de hecho, calculará la parte entera de una raíz cuadrada.

Es posible emplear un enfoque matemático más riguroso en el diseño de la verificación. Sin embargo, una exposición de este tema rebasa el ámbito de este libro. Los lectores interesados pueden consultar [LIN79].

¿Qué se gana al realizar pruebas de corrección?

29.3.2 Ventajas de la verificación del diseño[4]

La verificación rigurosa de la corrección de cada refinamiento del diseño de la caja transparente tiene varias ventajas. Linger [LIN94] las describe de la siguiente forma:

- Reducen la verificación a un proceso finito. La manera secuencial y anidada en que se organizan las estructuras de control en una caja transparente define con naturalidad una jerarquía que revela las condiciones de corrección que se deben verificar. Un "axioma de reemplazo" [LIN79] permite sustituir las funciones proyectadas con sus refinamientos de estructura de control en la jerarquía de las subpruebas. Por ejemplo, la subprueba de la función proyectada f1 de la figura 29.9 requiere probar que la composición de las operaciones g1 y g2 con la función proyectada f2 tiene el mismo efecto sobre los datos que f1. Nótese que f2 sustituye todos los detalles de su refinamiento en la prueba. Esta sustitución localiza el argumento de prueba con la estructura de control que se esta estudiando. De hecho, deja que el ingeniero de software lleve a cabo las pruebas en cualquier orden.
- Es imposible sobreestimar el efecto positivo que la reducción de la verificación a un proceso finito tiene sobre la calidad. Aun cuando todos los programas, excepto los más triviales,muestren un número esencialmente infinito de trayectorias de ejecución, pueden verificarse en un número finito de pasos.

FIGURA 29.9
Diseño con subpruebas.

```
[f1]                          Subpruebas:
  DO
    g1                        f1 = [DO g1; g2: [f2] END] ?
    g2
    [f2]                      f2 = [WHILE p1 DO [f3] END] ?
      WHILE
        p1
      DO [f3]                 f3 = [DO g3: [f4]: g8 END] ?
        g3
        [f4]                  f4 = [IF p2; THEN [f5] ELSE [f8] END] ?
        IF
          p2
        THEN [f5]             f5 = [DO g4: g5 END] ?
          g4
          g5
        ELSE [f8]             f8 = [DO g6: g7 END] ?
          g6
          g7
        END
        g8
    END
END
```

4 Esta sección y las figuras 29.7 a 29.9 han sido adaptadas de [LIN94] y se usaron con permiso.

PUNTO CLAVE

A pesar del número extremadamente grande de trayectorias de ejecución en un programa, el número de pasos para probar que el programa es correcto es bastante pequeño

- *Permiten al equipo de sala limpia verificar cada línea de diseño y código.* Los equipos pueden realizar la verificación por medio del análisis y la discusión en grupo sobre la base del teorema de corrección, y son capaces de producir pruebas escritas donde se requiere una confianza adicional en un sistema crucial para la vida o la misión.
- *Resultan en un nivel de defecto cercano a cero.* Durante una revisión de equipo se verifica por turnos la condición de corrección de cada estructura de control. Cada miembro del equipo debe estar de acuerdo en que cada condición es correcta, de modo que un error sólo es posible si cada miembro del equipo verifica de manera incorrecta una condición. El requisito del acuerdo unánime basado en la verificación individual genera un software con pocos o ningún defecto antes de su primera ejecución.
- *Es escalable.* Cualquier sistema de software, sin importar cuán grande sea, tiene procedimientos de caja transparente de nivel superior compuestos de estructuras de sucesión, alternación e iteración. Cada una de ellas usualmente invoca un gran subsistema con miles de líneas de código, y cada uno de dichos subsistemas tiene sus propios procedimientos y funciones proyectadas de nivel superior. De modo que las condiciones de corrección para tales estructuras de nivel superior se verifican en la misma forma que las estructuras de nivel inferior. La verificación en niveles superiores puede tomar, y vale la pena, más tiempo, pero no requiere más teoría.
- *Produce mejor código que la prueba unitaria.* La prueba unitaria verifica los efectos de ejecutar sólo las trayectorias de prueba seleccionadas entre muchas trayectorias posibles. Al basar la verificación en la teoría de funciones, el enfoque de sala limpia puede verificar cualquier efecto posible sobre todos los datos porque, mientras un programa puede tener muchas trayectorias de ejecución, sólo tiene una función. La verificación también es más eficiente que la prueba unitaria. La mayoría de las condiciones de verificación se puede comprobar en unos cuantos minutos, pero las pruebas unitarias requieren un tiempo sustancial en su preparación, ejecución y comprobación.

Es importante advertir que la verificación del diseño debe, a final de cuentas, aplicarse al propio código fuente. En este contexto, con frecuencia se le llama *verificación de la corrección*.

29.4 PRUEBAS DE SALA LIMPIA

La estrategia y las tácticas de las pruebas de sala limpia son fundamentalmente diferentes de los enfoques de prueba convencionales. Los métodos convencionales generan un conjunto de casos de prueba para descubrir errores de diseño y codificación. La meta de las pruebas de sala limpia es validar los requisitos de software

demostrando que una muestra estadística de casos de uso (capítulo 7) se ha ejecutado exitosamente.

> "La calidad no es un acto; es un hábito."
>
> Aristóteles

29.4.1 Pruebas estadísticas de uso

El usuario de un programa de computadora rara vez necesita entender los detalles técnicos del diseño. El comportamiento del programa que ve el usuario lo alimentan entradas y eventos que con frecuencia él mismo produce. Pero, en los sistemas complejos, el posible espectro de entradas y eventos (es decir, los casos de uso) tal vez sea extremadamente amplio. ¿Qué subconjunto de casos de uso verificará adecuadamente el comportamiento del programa? Esta es la primera pregunta que aborda la prueba estadística de uso

CONSEJO

Incluso si no se es partidario del enfoque de sala limpia, vale la pena considerar las pruebas estadísticas de utilización como parte integral de su estrategia de pruebas

La prueba estadística de uso "equivale a probar el software en la forma que los usuarios intentarían usarlo" [LIN94]. Esto se logra si los equipos de prueba de sala limpia (también llamados *equipos de certificación*) determinan una distribución de probabilidad de uso para el software. La especificación (caja negra) de cada incremento del software se analiza para definir un conjunto de estímulos (entradas o eventos) que provocan el cambio de comportamiento del software. Con base en entrevistas con usuarios potenciales, la creación de escenarios de uso y una comprensión general del dominio de la aplicación, se asigna una probabilidad de uso a cada estímulo.

Los casos de prueba se generan para cada conjunto de estímulos[5] de acuerdo con la distribución de probabilidad de uso. Con fines ilustrativos, considérese el sistema *HogarSeguro* estudiado previamente en este libro. La ingeniería del software de sala limpia se aplica en el desarrollo de un incremento de software que gestiona la interacción del usuario con el teclado numérico del sistema de seguridad. Respecto de este incremento se han identificado cinco estímulos. El análisis indica el porcentaje de distribución de probabilidad de cada estímulo. Con el fin simplificar la selección de los casos de prueba dichas probabilidades están se correlacionan con los intervalos numerados entre 1 y 99 [LIN94] y se ilustran en la tabla siguiente:

Estímulo del programa	Probabilidad	Intervalo
Habilitar/deshabilitar (HD)	50%	1-49
Fijar zona (FZ)	15%	50-63
Consulta (C)	15%	64-78
Prueba (P)	15%	79-94
Alarma(A)	5%	95-99

5 Es posible utilizar herramientas automatizadas para lograr esto. Véase [DYE92]para mayor información

Crear una sucesión de casos de prueba de uso que concuerde con la distribución de probabilidad de uso requiere generar números aleatorios entre 1 y 99. Cada número aleatorio corresponde a un intervalo en la distribución de probabilidad precedente. Por lo tanto, la secuencia de casos de prueba de uso se define aleatoriamente, pero corresponde a la probabilidad correspondiente de presencia de estímulos. Por ejemplo, supóngase que se generan las siguientes secuencias de números aleatorios:

13-94-22-24-45-56
81-19-31-69-45-9
38-21-52-84-86-4

Al seleccionar los estímulos apropiados con base en el intervalo de distribución que se muestra en la tabla se derivan los casos de uso siguientes:

HD-P-HD HD-HD-FZ
P-HD-HD-C-HD-HD
HD-HD-FZ-P-P-HD

El equipo de prueba ejecuta estos casos de uso y verifica el comportamiento del software frente a la especificación del sistema. El tiempo para las pruebas se registra de modo que sea posible determinar los tiempos de intervalo. Al usar tiempos de intervalo, el equipo de certificación tiene la posibilidad de calcular el tiempo medio entre fallas (TMEF). Si se lleva a cabo una larga secuencia de pruebas sin fallas, el TMEF es bajo y es probable que la fiabilidad del software sea alta.

29.4.2 Certificación

Las técnicas de verificación y prueba ya descritas en este capítulo llevan a componentes de software (e incrementos completos) que pueden certificarse. En el contexto del enfoque de ingeniería del software de sala limpia la certificación implica que la fiabilidad (medida en TMEF) se especifica para cada componente.

El impacto potencial de los componentes de software certificables va mucho más allá de un solo proyecto de sala limpia. Los componentes de software reutilizables se pueden almacenar junto con sus escenarios de uso, estímulos de programa y distribuciones de probabilidad. Cada componente tendría una fiabilidad certificada en el escenario de uso y el régimen de pruebas descritos. Esta información es invaluable para otros que intenten emplear los componentes.

El enfoque de la certificación involucra cinco pasos [WOH94]:

1. Creación de escenarios de uso.
2. Especificación de perfiles de uso.
3. Generación de casos de prueba a partir del perfil.
4. Ejecución de pruebas y registro y análisis de datos de fallas.
5. Cálculo y certificación de la fiabilidad.

¿Cómo se certifica un componente de software?

Los pasos del 1 al 4 se han tratado en una sección anterior. Esta sección se concentrará en la certificación de la fiabilidad

La certificación para la ingeniería del software de sala limpia requiere la creación de tres modelos [POO93]:

Modelo de muestreo. La prueba del software ejecuta *m* casos de prueba aleatorios y se certifica si no ocurren fallas o un número específico de éstas. El valor de *m* se deriva matemáticamente para asegurar que se logra la fiabilidad requerida.

Modelo de componentes. Se certificará un sistema compuesto de *n* componentes. El modelo de componentes permite que el analista determine la probabilidad de que el componente *i* fallará antes de completarse.

Modelo de certificación. La fiabilidad global del sistema se proyecta y certifica.

En el momento de completar las pruebas estadísticas de uso el equipo de certificación tiene la información necesaria para entregar el software que tiene un TMEF certificado, el cual se calcula empleando cada uno de dichos modelos.

Una descripción detallada del cálculo de los modelos de muestreo, componentes y certificación está más allá del ámbito de este libro. El lector interesado hallará detalles adicionales en [MUS87], [CUR86] y [POO93].

29.5 Resumen

La ingeniería del software de sala limpia es un enfoque formal para el desarrollo de software que puede llevar a software con calidad notablemente alta. Utiliza la especificación de estructura de cajas (o métodos formales) para el modelado de análisis y diseño y resalta la verificación de la corrección, en lugar de las pruebas, como el principal mecanismo para detectar y eliminar los errores. Se aplican pruebas estadísticas de utilización para desarrollar la información necesaria de tasa de fallas con que certificar la fiabilidad del software entregado.

El enfoque de sala limpia comienza con los modelos de análisis y diseño que utilizan una representación en estructura de cajas. Una "caja" encapsula el sistema (o algún aspecto de él) en un grado específico de abstracción. Las cajas negras se aprovechan para representar el comportamiento de un sistema observable de manera externa. Las cajas de estado encapsulan datos y operaciones de estado. Una caja transparente se emplea en el modelado del diseño de procedimiento que implican los datos y operaciones de una caja de estado.

La verificación de la corrección se aplica cuando se completa el diseño de la estructura de cajas. El diseño de procedimiento para un componente de software se divide en una serie de subfunciones. La prueba de la corrección de las subfunciones requiere definir condiciones de salida para cada subfunción y se aplica un conjunto de subpruebas. Si cada condición de salida se satisface el diseño debe ser correcto.

Una vez completada la verificación de la corrección comienza la prueba estadística de uso. A diferencia de las pruebas convencionales, la ingeniería del software de sala limpia no subraya la importancia de las pruebas unitarias o de integración. En vez de ello el software se prueba definiendo un conjunto de escenarios de uso, determinando la probabilidad de uso para cada escenario y definiendo entonces las pruebas aleatorias que concuerden con las probabilidades. Los registros de error resultantes se combinan con los modelos de muestreo, componentes y certificación para permitir el cálculo matemático de la fiabilidad proyectada respecto al componente de software.

La filosofía de sala limpia es un enfoque riguroso para la ingeniería del software. Es un modelo de proceso de software que destaca la verificación matemática de la corrección y la certificación de la fiabilidad del software. La línea fundamental es las tasas de falla extremadamente bajas que serían difíciles o imposibles de lograr empleando métodos menos formales.

REFERENCIAS

[CUR86] Currit, P. A., M. Dyer y H. D. Mills, "Certifying the Reliability of Software", en *IEEE Trans Software Engineering*, vol. SE-12, núm. 1, enero de 1994.

[DYE92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.

[HAU94] Hausler, P. A., R. Linger y C. Trammel, "Adopting Cleanroom Software Engineering with a Phased Approach", en *IBM Systems Journal*, vol. 33, núm. 1, enero de 1994, pp. 89-109.

[HEN95] Henderson, J., "Why Isn't Cleanroom the Universal Software Development Methodology", en *Crosstalk*, vol. 8, núm. 5, mayo de 1995, pp. 11-14.

[HEV93] Hevner, A. R., y H. D. Mills, "Box Structure Methods for System Development with Objects", en *IBM Systems Journal*, vol. 31, núm. 2, febrero de 1993, pp. 232-251.

[LIN79] Linger, R. M., H. D. Mills y B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.

[LIN88] Linger, R. M., y H. D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility", *Proc. COMPSAC '88*, Chicago, octubre de 1988.

[LIN94] Linger, R., "Cleanroom Process Model", en *IEEE Software*, vol. 11, núm. 2, marzo de 1994, pp. 50-58.

[MIL87] Mills, H. D., M. Dyer y R. Linger, "Cleanroom Software Engineering", en *IEEE Software*, vol. 4, núm. 5, septiembre de 1987, pp. 19-24.

[MIL88] Mills, H. D., "Stepwise Refinement and Verification in Box Structured Systems", en *Computer*, vol. 21, núm. 6, junio de 1988, pp. 23-35.

[MUS87] Musa, J. D., A. Iannino y K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.

[POO88] Poore, J. H., y H. D. Mills, "Bringing Software Under Statistical Quality Control", en *Quality Progress*, noviembre de 1988, pp. 52-55.

[POO93] Poore, J. H., H. D. Mills y D. Mutchler, "Planning and Certifying Software System Reliability", en *IEEE Software*, vol. 10, núm. 1, enero de 1993, pp. 88-99.

[WOH94] Wohlin, C., y P. Runeson, "Certification of Software Components", en *IEEE Trans. Software Engineering*, vol. SE-20, núm. 6, junio de 1994, pp. 494-499.

PROBLEMAS Y PUNTOS A CONSIDERAR

29.1. Si se tuviese que elegir un aspecto de la ingeniería del software de sala limpia que la hiciese radicalmente diferente de los enfoques convencionales de ingeniería de software, ¿cuál sería?

29.2. ¿Cómo trabajan en conjunto un modelo de proceso incremental y la certificación para producir software de alta calidad?

29.3. Empleando la especificación de estructura de cajas desarróllense modelos de análisis "de primer paso" y de diseño para el sistema *HogarSeguro*.

29.4. Desarróllese una especificación de estructura de cajas para una porción del sistema PHTRS presentado en el problema 8.10.

29.5. Un algoritmo de ordenamiento en burbuja se define del modo siguiente:

```
procedure bubblesort;
var i, t, integer;
begin
repeat until t = a[1]
      t: = a[1];
      for j: = 2 to n do
         if a [j-1] > a[j] then begin
            t: = a[j-1];
            a[j-1]: = a[j];
            a[j]: = t;
            end
endrep
end
```

Divídase el diseño en subfunciones y defínase un conjunto de condiciones que permitirían probar que este algoritmo es correcto.

29.6. Documéntese una prueba de verificación de corrección para el ordenamiento en burbuja tratado en el problema 29.5.

29.7. Selecciónese un componente de programa que se haya diseñado en otro contexto (o uno que haya asignado el instructor) y desarróllese respecto de él una prueba completa de corrección

29.8. Selecciónese un programa que se use regularmente (por ejemplo, un gestor de correo electrónico, un procesador de palabra, un programa de hojas de cálculo) y créese un conjunto de escenarios de uso para el programa. Defínase la probabilidad de uso para cada escenario y luego desarróllese una tabla de estímulos de programa y de distribución de probabilidad similar al que se muestra en la sección 29.4.1.

29.9. Para la tabla de estímulos de programa y distribución de probabilidad desarrollada en el problema 29.8, utilícese un generador de números aleatorios con el fin de desarrollar un conjunto de casos de prueba para emplearlo en pruebas estadísticas de uso

29.10. Con palabras propias, describase el intento de certificación en el contexto de ingeniería del software de sala limpia.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Prowell *et al* (*Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999) ofrecen un tratamiento detallado de los aspectos importantes del enfoque de sala limpia. Poore y Trammell (*Cleanroom Software Engineering. A Reader*, Blackwell Publishing, 1996) han editado exposiciones útiles de temas de sala limpia. Becker y Whittaker (*Cleanroom Software Engineering Practices*, Idea Group Publishing, 1996) presentan un excelente panorama para quienes no están familiarizados con las prácticas de sala limpia.

The Cleanroom Pamphlet (Software Technology Support Center, Hill AFBase, abril de 1995) contiene reimpresiones de varios artículos importantes. Linger [LIN94] produjo una de las mejores introducciones a la materia. El Data and Analysis Center for Software (DACS) (www.dacs.dtic.mil) ofrece muchos artículos útiles, libros guía y otras fuentes de información acerca de la ingeniería del software de sala limpia.

Linger y Trammell ("Cleanroom Software Engineering Reference Model", SEI Technical Report CMU/SEI-96-TR-022, 1996) han definido un conjunto de 14 procesos de sala limpia y 20 productos de trabajo que forman la base para la SEI CMM de la ingeniería de software de sala limpia (CMU/SEI-96-TR-023).

Michael Deck de Cleanroom Software Engineering (www.cleansoft.com) ha preparado una bibliografía acerca de temas de sala limpia. Muchos están disponibles en formato descargable.

La verificación del diseño mediante la prueba de las correcciones se encuentra en el centro del enfoque de sala limpia. Los libros de Stavely (*Toward Zero-Defect Software*, Addison-Wesley, 1998), Baber (*Error-Free Software*, Wiley, 1991) y Schulmeyer (*Zero Defect Software*, McGraw-Hill, 1990) abordan la prueba de corrección en forma muy detallada.

En Internet hay disponible una amplia variedad de fuentes de información acerca de la ingeniería del software de sala limpia. Una lista actualizada de referencias en la World Wide Web se puede encontrar en el sitio Web SEPA:
http://www.mhhe.com/pressman.

CAPÍTULO

30

INGENIERÍA DEL SOFTWARE BASADA EN COMPONENTES

CONCEPTOS CLAVE

En el contexto de la ingeniería del software la reutilización es una idea tanto antigua como nueva. Los programadores han reutilizado ideas, abstracciones y procesos desde los primeros días de la computación, pero el enfoque original para la reutilización era específico. En la actualidad, los complejos sistemas de alta calidad basados en computadoras se deben construir en un tiempo muy corto y demanda un enfoque más organizado de la reutilización.

La *ingeniería del software basada en componentes* (ISBC) es un proceso que concede particular importancia al diseño y la construcción de sistemas basados en computadoras que utilizan "componentes" de software reutilizables. Clements [CLE95] describe así la ISBC:

> [la ISBC] está cambiando la forma en que se desarrollan los grandes sistemas de software. [La ISBC] encarna la filosofía de "comprar, no construir" de la cual son partidarios Fred Brooks y otros. En la misma forma como las primeras subrutinas liberaron al programador de pensar acerca de los detalles, [la ISBC] cambió el interés de programar software por el de componer sistemas de software. La implementación ha dado paso a la integración como centro del enfoque. En sus cimientos está la suposición de que existe suficiente base común entre muchos grandes sistemas de software para justificar el desarrollo de componentes reutilizables para explotarla y satisfacerla.

Pero surgen varias preguntas. ¿Es posible construir sistemas complejos mediante el ensamblado de componentes de software reutilizables provenientes de un catálogo? ¿Esto se puede lograr en una forma eficaz tanto en costo como en tiempo? ¿Es posible establecer incentivos adecuados que alienten a los ingenieros de software a reutilizar en vez de reinventar? ¿Los gestores tienen buena disposición para incurrir en el gasto adicional asociado con la creación de componentes de software reutilizables? ¿La biblioteca de componentes es necesaria para lograr que la reutilización se cree en una forma que sea accesible a quienes la necesitan? ¿Los componentes que existen pueden encontrarlos quienes los necesiten?

UN VISTAZO RÁPIDO

¿Qué es? Usted compra un sistema de entretenimiento y lo lleva a casa. Cada componente ha sido diseñado para encajar en una arquitectura específica de audio y video: las conexiones son estandarizadas y el protocolo de comunicación se ha preestablecido. El ensamblado es sencillo porque usted no tiene que construir el sistema a partir de cientos de partes discretas. La ingeniería de software basada en componentes (ISBC) lucha por lograr la misma cosa. Un conjunto de componentes de software estandarizados preconstruidos se hacen disponibles para encajar en un estilo arquitectónico específico para cierto dominio de aplicación. Entonces la aplicación es ensamblada usando dichos com

ponentes, en lugar de las partes discretas de un lenguaje de programación convencional.

¿Quién lo hace? Los ingenieros de software aplican el proceso de ISBC.

¿Por qué es importante? Toma sólo unos cuantos minutos ensamblar el sistema de entretenimiento del hogar porque los componentes están diseñados para ser integrados con facilidad. Aunque el software es considerablemente más complejo, se sigue que los sistemas basados en componentes son más fáciles de ensamblar y por lo tanto menos costosos de construir que los sistemas que se construyen a partir de partes discretas. Además, la ISBC alienta el uso de patrones arquitectónicos predecibles y de infraestructura de software estándar, y por lo tanto conduce a un resultado de mayor calidad.

¿Cuáles son los pasos? La ISBC abarca dos actividades de ingeniería paralelas: la ingeniería de dominios y el desarrollo basado en componentes. La ingeniería de dominios explora un dominio de aplicación con la intención específica de encontrar componentes funcionales, de comportamiento y de datos que sean candidatos para la reutilización. Dichos componentes son colocados en librerías de reutilización. El desarrollo basado en componentes obtiene requisitos de los clientes; selecciona un estilo arquitectónico apropiado para satisfacer los objetivos del sistema a construir; y luego 1) selecciona componentes potenciales para reutilización, 2) califica los componentes para asegurarse de que encajan adecuadamente en la arquitectura para el sistema, 3) adapta los componentes si se deben hacer modificaciones para integrarlos adecuadamente, y 4) integra los componentes para formar subsistemas y la aplicación como un todo. Además, algunos componentes personalizados son sometidos a ingeniería para enfrentar aquellos aspectos del sistema que no pueden ser implementados con el uso de los componentes existentes.

¿Cuál es el producto de trabajo? Un software operativo, ensamblado con el uso de componentes de software existentes y desarrollados recientemente, es el resultado de la ISBC.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Use las mismas prácticas de SQA que se aplican en todo proceso de ingeniería del software: las revisiones técnicas formales valoran los modelos de análisis y de diseño, las revisiones especializadas consideran los conflictos asociados con los componentes adquiridos, las pruebas se aplican para descubrir errores en el software desarrollado recientemente y en los componentes reutilizables que se han integrado en la arquitectura.

Incluso en la actualidad, los ingenieros de software luchan con éstas y otras preguntas acerca de la reutilización de componentes de software. En este capítulo se abordan algunas de las respuestas.

30.1 INGENIERÍA DE SISTEMAS BASADA EN COMPONENTES

Referencia Web
Información útil acerca de ISBC para WebApps se puede encontrar en www.cbd-hq.com.

En la superficie, la ISBC parece bastante similar a la ingeniería del software orientada a objetos convencional. El proceso comienza cuando un equipo de software establece requisitos para el sistema que se construirá mediante técnicas convencionales de investigación de requisitos (capítulo 7). Se establece un diseño arquitectónico (capítulo 10), pero en lugar de dirigirse inmediatamente hacia tareas de diseño más detalladas, el equipo examina los requisitos para determinar qué subconjunto está directamente dispuesto para la *composición*, y no para la construcción. Es decir, el equipo plantea las siguientes preguntas para cada requisito del sistema:

- ¿Hay componentes comerciales de línea (CDL) disponibles para implementar los requisitos?
- ¿Hay disponibles componentes reutilizables desarrollados internamente para implementar los requisitos?
- ¿Las interfases para los componentes disponibles son compatibles dentro de la arquitectura del sistema que se construirá?

El equipo tal vez intente modificar o eliminar aquellos requisitos del sistema que no sea posible implementar con CDL o de desarrollo propio[1]. Si el requisito no puede cambiarse o eliminarse se aplican los métodos de ingeniería del software en la construcción de aquellos nuevos componentes que deben desarrollarse para satisfacer los requisitos. Pero para los requisitos que se abordan con los componentes disponibles comienza un conjunto diferente de actividades de ingeniería del software: cualificación, adaptación, composición y actualización. Cada una de estas actividades de ISBC se examina con mayor detalle en la sección 30.4.

En la primera parte de esta sección se ha utilizado repetidamente el término *componente*, aunque falta una descripción definitiva del término. Brown y Wallnau [BRO96] sugieren las siguientes posibilidades:

- *Componente*: parte importante, casi independiente y reemplazable de un sistema que satisface una función clara en el contexto de una arquitectura bien definida
- *Componente del software en ejecución*: paquete dinámico de unión de uno o más programas gestionados como unidad y a los cuales se tiene acceso por medio de interfases documentadas que se pueden descubrir en la ejecución.
- *Componente de software*: unidad de composición que sólo tiene dependencias de contexto explícitas y especificadas en forma contractual.
- *Componente de negocio*: la implementación de software de un concepto o proceso de negocio "autónomo".

Además de estas descripciones, los componentes de software también se pueden caracterizar con base en sus aplicaciones en el proceso de ISBC. Además de los componentes CDL, el proceso de ISBC produce:

- *Componentes cualificados*: evaluados por ingenieros de software para garantizar que no sólo la funcionalidad, sino el desempeño, la fiabilidad, la facilidad de uso y otros factores de calidad (capítulo 26) concuerdan con los requisitos del sistema o producto que se construirá.

1 La implicación es que la organización ajusta los requisitos de su negocio o producto de modo que la implementación basada en componentes se consiga sin que sea necesaria la ingeniería de personalización. Este enfoque reduce los costos y mejora el tiempo en que el producto llega al mercado, pero no siempre es posible

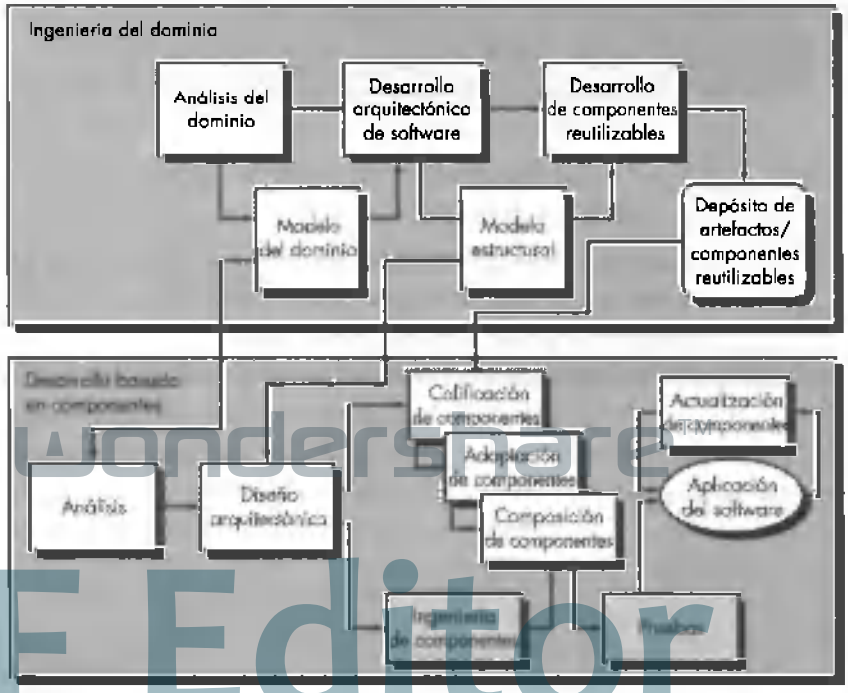
- *Componentes adaptados:* adaptados para modificar (acción también llamada *enmascarar* o *envolver*) [BRO96] características que no se requieren o indeseables
- *Componentes actualizados:* sustituyen el software existente conforme están disponibles las nuevas versiones de los componentes.

30.2 El proceso de ISBC

El *proceso de ISBC* se caracteriza de tal forma que no sólo identifica los componentes candidatos sino que también cualifica la interfaz de cada componente, adapta los componentes para eliminar las equivocaciones arquitectónicas, ensambla los componentes en un estilo arquitectónico seleccionado y actualiza los componentes conforme los requisitos del sistema cambian [BRO96]. El modelo de proceso para la ingeniería del software basada en componentes destaca las pistas paralelas en las cuales la ingeniería del dominio (sección 30.3) se lleva a cabo concurrentemente con el desarrollo basado en componentes

La figura 30.1 ilustra un modelo de proceso típico que explícitamente acopla la ISBC [CHR95]. La *ingeniería del dominio* crea un modelo del dominio de aplicación que se utiliza como base para analizar los requisitos del usuario en el flujo de ingeniería del software. Una arquitectura genérica de software proporciona la entrada para el diseño de la aplicación. Finalmente, después de que los componentes reutilizables se han comprado, seleccionado de bibliotecas existentes o construido (co-

FIGURA 30.1
Modelo de proceso que apoya la ISBC.



mo parte de la ingeniería del dominio), están disponibles para los ingenieros de software durante el desarrollo basado en componentes.

Los pasos de *análisis* y *diseño arquitectónico* definidos como parte del *desarrollo basado en componentes* (figura 30.1) se pueden implementar en el contexto de un *paradigma de diseño abstracto* (ADP, por sus siglas en inglés) [DOG03]. Un ADP implica que el modelo global del software —representado como datos, funciones y comportamiento (control)— se puede descomponer jerárquicamente. Conforme la descomposición comienza, el sistema se representa como una colección de marcos de trabajo arquitectónico, cada uno compuesto de uno o más patrones de diseño (capítulo 10). Un refinamiento mayor identifica los componentes necesarios para crear cada patrón de diseño. En un contexto ideal, todos los componentes se adquirirían a partir de un depósito (aplicando actividades de *calificación*, *adaptación* y *composición de componentes*). Cuando se requieren componentes especializados se aplica la *ingeniería de componentes*

30.3 INGENIERÍA DEL DOMINIO

La finalidad de la *ingeniería del dominio* es identificar, construir, catalogar y diseminar un conjunto de componentes de software que sean aplicables para el software existente y futuro en un dominio de aplicación particular. La meta global es establecer mecanismos que les permitan a los ingenieros de software compartir dichos componentes —para reutilizarlos— durante el trabajo en sistemas nuevos y existentes. La ingeniería del dominio incluye tres grandes actividades: análisis, construcción y diseminación.

> "La ingeniería del dominio trata de encontrar los aspectos comunes entre los sistemas para identificar los componentes que sea posible aplicar en muchos sistemas, y para identificar familias de programas que se posicionen para sacar la mayor ventaja de dichos componentes."
>
> **Paul Clements**

CONSEJO

El proceso de análisis estudiado en esta sección se enfoca en los componentes reutilizables. Sin embargo, el análisis de sistemas OO completos (por ejemplo, aplicaciones de comercio electrónico, aplicaciones de automatización de fuerza de ventas) también puede ser una parte del análisis del dominio.

Se puede argumentar que "la reutilización desaparecerá, no por eliminación, sino por integración" en la estructura de la práctica de la ingeniería del software [TRA95]. Como la reutilización cada vez tiene mayor auge algunos creen que la ingeniería del dominio adquirirá tanta importancia como la ingeniería del software durante la década siguiente.

30.3.1 El proceso de análisis del dominio

El enfoque global para el análisis del dominio usualmente se caracteriza en el contexto de la ingeniería del software orientada a objetos. Los pasos en el proceso se definen como:

1. Definir el dominio que se investigará
2. Categorizar los elementos extraídos del dominio.

¿Qué componentes identificados durante el análisis del dominio serán candidatos para la reutilización?

3. Recopilar una muestra representativa de las aplicaciones en el dominio.
4. Analizar cada aplicación en la muestra y definir las clases de análisis.
5. Desarrollar un modelo de análisis para las clases.

Es importante advertir que el análisis del dominio es aplicable a cualquier paradigma de ingeniería del software, y que se puede aplicar al desarrollo convencional y al orientado a objetos.

Aunque los pasos citados ofrecen un modelo útil para el análisis del dominio, no brindan una guía para decidir cuáles componentes de software son candidatos a la reutilización. Hutchinson y Hindley [HUT88] sugieren el siguiente conjunto de preguntas prácticas como una guía para identificar los componentes de software reutilizables:

- ¿En las implementaciones futuras se requiere la funcionalidad del componente?
- ¿Cuán común es la función del componente dentro del dominio?
- ¿Existe duplicidad de la función del componente dentro del dominio?
- ¿El componente depende del hardware? Si es así, ¿el hardware permanece invariable entre las implementaciones o las especificaciones del hardware pueden trasladarse hacia otro componente?
- ¿El diseño está lo suficientemente optimizado para la siguiente implementación?
- ¿Se pueden establecer parámetros respecto de un componente no reutilizable de modo que se vuelva reutilizable?
- ¿El componente es reutilizable en muchas implementaciones sólo con cambios menores?
- ¿Es factible la reutilización por medio de la modificación?
- ¿Un componente no reutilizable se puede descomponer para producir componentes reutilizables?
- ¿Cuán válida es la descomposición de un componente para la reutilización?

Para información adicional acerca del análisis del dominio véase [ATK01], [HEI01] y [PRI93].

30.3.2 Funciones de caracterización

A veces es difícil determinar si un componente potencialmente reutilizable es de hecho aplicable en una situación particular. Esta determinación requiere definir un conjunto de características del dominio que comparta todo el software dentro de un dominio. Una característica del dominio define algún atributo genérico de todos los productos que existen dentro de él. Por ejemplo, las características genéricas podrían incluir la importancia de la seguridad y fiabilidad, el lenguaje de programación, la concurrencia en el procesamiento y muchas otras.

Referencia Web
Información útil acerca del análisis del dominio se puede encontrar en www.sei.cmu.edu/str/descriptions/dade.html.

Un conjunto de características de dominio de un componente reutilizable se puede representar como $[D_p]$, donde cada elemento, D_{pi}, en el conjunto representa una característica específica del dominio. El valor asignado a D_{pi} representa una escala ordinal que indica la relevancia de la característica para el componente p. Una escala típica [BAS94] podría ser

1: No es relevante si la reutilización es apropiada.
2: Relevante sólo en circunstancias inusuales.
3: Relevante: el componente se modifica para usarlo, a pesar de las diferencias.
4: Claramente relevante, y si el nuevo software no tiene esta característica, la reutilización será ineficiente pero tal vez sea posible.
5: Claramente relevante, y si el nuevo software no tiene esta característica, la reutilización será ineficiente y la reutilización sin dicha característica no se recomienda.

Cuando dentro del dominio de aplicación se construirá nuevo software, w, se deriva para él un conjunto de características del dominio. Entonces se comparan D_{pi} y D_{wi} para determinar si el componente existente p se reutiliza con eficacia en la aplicación w.

Aunque el software que se construirá claramente existe dentro de un dominio de aplicación, los componentes reutilizables en él se deben analizar para determinar su aplicabilidad. En algunos casos (con suerte, un número limitado), "reinventar la rueda" tal vez sea la mejor elección en cuanto a costo

30.3.3 Modelado estructural y puntos de estructura

Cuando se aplica el análisis del dominio el analista busca los patrones repetitivos en las aplicaciones que residen dentro de un dominio. El modelado estructural es un enfoque de ingeniería del dominio basada en patrones que funciona bajo la premisa de que cualquier dominio de aplicación tiene patrones repetitivos (de función, datos y comportamiento) que tienen un potencial de reutilización

Cada dominio de aplicación se caracteriza mediante un modelo estructural (por ejemplo, los sistemas aviónicos de las aeronaves difieren enormemente en especificaciones, pero todo el software moderno en este dominio tiene el mismo modelo estructural). Por lo tanto, el modelo estructural es un estilo arquitectónico (capítulo 10) que puede y debe reutilizarse mediante las aplicaciones dentro del dominio.

McMahon [MCM95] describe un *punto de estructura* como "una estructura distinta dentro de un modelo estructural". Los puntos de estructura tienen tres características distintas:

¿Qué es un punto de estructura y cuáles son sus características?

1. Un punto de estructura es una abstracción que debe tener un número limitado de instancias. Además, la abstracción debe recurrir a través de las aplicaciones en el dominio. De otro modo no se justifica el costo de verificar, documentar y diseminar el punto de estructura

2. Las reglas que rigen el uso del punto de estructura deben comprenderse con facilidad. Además, la interfaz para el punto de estructura debe ser relativamente simple.
3. El punto de estructura debe implementar la ocultación de información al aislar toda la complejidad dentro del mismo punto de estructura. Esto reduce la complejidad percibida del sistema globales conjunto.

¡PUNTO CLAVE

Un punto de estructura es análogo a un patrón de diseño que se puede encontrar repetidamente en aplicaciones con un dominio específico

Como un ejemplo de puntos de estructura como patrones arquitectónicos de un sistema, considérese el dominio de software de sistemas de alarma. Este dominio puede abarcar sistemas tan simples como *HogarSeguro* (descritos en capítulos anteriores) o tan complejos como el sistema de alarma para un proceso industrial. Sin embargo, en cada caso se encuentra un conjunto de patrones estructurales predecibles. una *interfaz* que le permite al usuario interactuar con el sistema; un *mecanismo de establecimiento de límites* que le permite al usuario establecer límites a los parámetros que se medirán; un *mecanismo de gestión de sensores* que se comunica con todos los sensores de supervisión; un *mecanismo de respuesta* que reacciona a la entrada proporcionada por el sistema de gestión de sensores, y un *mecanismo de control* que le permite al usuario controlar la forma en la que se realiza la supervisión. Cada uno de estos puntos de estructura se integra en una arquitectura de dominio.

Es posible definir puntos de estructura genéricos que trasciendan diferentes dominios de aplicación [STA94]:

- *Aplicación frontal (cliente):* la GUI que incluye todos los menús, paneles y entradas y ordena las funciones de edición.
- *Bases de datos:* el depósito para todos los objetos relevantes respecto del dominio de la aplicación.
- *Motor de cálculo:* los modelos numéricos y no numéricos que manipulan datos.
- *Función de generación de informes:* la función que produce salidas de cualquier tipo.
- *Editor de aplicaciones:* el mecanismo para personalizar la aplicación respecto a las necesidades de usuarios específicos.

Los puntos de estructura se han sugerido como una alternativa a las líneas de código y puntos de función para la estimación del costo del software [MCM95]. En la sección 30.6.2 se presenta una breve descripción del empleo de los puntos de estructura en la cotización.

30.4 DESARROLLO BASADO EN COMPONENTES

El *desarrollo basado en componentes* (DBC) es una actividad de ISBC que ocurre en paralelo con la ingeniería del dominio. Al aplicar los métodos de diseño de análisis y arquitectónico ya tratados en este libro, el equipo de software refina un estilo ar-

quitectónico apropiado para el modelo de análisis creado para la aplicación que se construirá [2]

Una vez que la arquitectura se ha establecido, deben agregársele componentes que 1) estén disponibles en bibliotecas de reutilización 2) sean diseñados para satisfacer las necesidades personales del cliente Por tanto, el flujo de tareas para el desarrollo basado en componentes tiene dos trayectorias paralelas (figura 30.1). Cuando los componentes reutilizables están disponibles para su potencial integración en la arquitectura tienen que cualificarse y adaptarse. Cuando se requieren nuevos componentes es preciso diseñarlos. Entonces los componentes resultantes se "componen" (integran) en la plantilla arquitectónica y se prueban en forma minuciosa.

30.4.1 Calificación, adaptación y composición de componentes

Como ya se ha visto, la ingeniería del dominio proporciona la biblioteca de componentes reutilizables indispensables para la ingeniería del software basada en componentes. Algunos de estos componentes se desarrollan especialmente para el dominio, otros pueden extraerse de aplicaciones existentes e incluso otros pueden adquirirse de terceras partes.

Desgraciadamente, la existencia de componentes reutilizables no garantiza que puedan integrarse con facilidad o eficacia en la arquitectura elegida para una nueva aplicación Por esta razón se aplica una sucesión de actividades de desarrollo basada en componentes cuando se propone el uso de un componente.

Calificación de componentes. Esta actividad garantiza que el componente candidato realizará la función requerida, "encajará" adecuadamente en el estilo arquitectónico que especifica el sistema y mostrará las características de calidad (por ejemplo, desempeño, fiabilidad, facilidad de uso) que requiere la aplicación

¿Qué factores se consideran durante la calificación de componentes?

La descripción de la interfaz suministra información útil acerca de la operación y la utilización de un componente de software, pero no proporciona toda la información que se requiere para determinar si un componente propuesto puede, en la práctica, reutilizarse con eficacia en una aplicación nueva. Entre los muchos factores considerados durante la cualificación de componentes están [BRO96]: interfaz de programación de la aplicación (IPA); herramientas de desarrollo e integración que requiere el componente; requisitos de tiempo de ejecución, que incluyen uso de recursos (por ejemplo, memoria o almacenamiento), tiempos o velocidad y protocolo de red; requisitos de servicio, que incluyen interfases de sistema operativo y apoyo de otros componentes; características de seguridad, que incluyen controles de acceso y protocolos de autenticación; suposiciones de diseño anidado, que incluyen el empleo de algoritmos numéricos o no numéricos específicos; y manejo de excepciones.

2 Se debe señalar que en el estilo arquitectónico con frecuencia influye el modelo estructural genérico creado durante la ingeniería del dominio (véase la figura 30.1).

Cada uno de estos factores es relativamente sencillo de evaluar cuando se proponen componentes reutilizables que se han desarrollado especialmente para la aplicación. Sin embargo, es mucho más difícil determinar la operatividad interna de los CDL o de componentes de provenientes de terceros porque la única información disponible tal vez sea la misma especificación de la interfaz.

Adaptación de componentes. En un contexto ideal, la ingeniería del dominio crearía una biblioteca de componentes que podrían integrarse fácilmente en una arquitectura de aplicación. La implicación de la "integración fácil" es que 1) se han implementado métodos consistentes de gestión de recursos para todos los componentes en la biblioteca, 2) existen actividades comunes como la gestión de datos para todos los componentes, y 3) se han implementado interfases dentro de la arquitectura y con el entorno externo en una forma consistente.

Además de valorar si es justificado el costo de adaptación para la reutilización, el equipo de software también evalúa si logrará la funcionalidad requerida y el desempeño se pueden hacer eficientes respecto del costo

En realidad, incluso después de que un componente se ha cualificado para emplearlo dentro de una arquitectura de aplicación, es posible que haya conflictos en una o más de las áreas indicadas. Estos conflictos usualmente se evitan utilizando una técnica de adaptación llamada *encubrimiento de componente* [BRO96]. Cuando un equipo de software tiene pleno acceso al diseño interno y el código de un componente (con frecuencia no es el caso cuando se utilizan componentes CDL) se aplica el *encubrimiento de caja blanca*. Al igual que su contraparte en la prueba de software (capítulo 14), el encubrimiento de caja blanca examina los detalles de procesamiento interno del componente y hace modificaciones en el código para eliminar cualquier conflicto. El *encubrimiento de caja gris* se aplica cuando la biblioteca de componentes proporciona un lenguaje de extensión de componente o IPA que permite eliminar o enmascarar los conflictos. El *encubrimiento de caja negra* requiere la introducción de pre y posprocesamiento en la interfaz del componente para eliminar o enmascarar los conflictos. El equipo de software debe determinar si el esfuerzo requerido para encubrir adecuadamente un componente está justificado o si, en lugar de ello, debe diseñarse un componente personalizado (designado para eliminar los conflictos encontrados).

Composición de componentes. La tarea de composición de componente ensambla componentes cualificados, adaptados y diseñados con el fin de agregárselos a la arquitectura establecida para una aplicación. Esto se logra estableciendo una infraestructura que una los componentes en un sistema operativo. La infraestructura (usualmente una biblioteca de componentes especializados) proporciona un modelo para coordinar los componentes y los servicios específicos que permiten que los componentes se coordinen mutuamente y realicen tareas comunes.

Entre los muchos mecanismos que existen para crear una infraestructura eficaz hay un conjunto de cuatro "ingredientes arquitectónicos" [ADL95] que debe estar presente para lograr la composición de componentes:

Modelo de intercambio de datos. Respecto de los componentes reutilizables se deben definir mecanismos que permitan que los usuarios y aplicaciones interac-

¿Qué ingredientes se necesitan para lograr la composición de componentes?

túen y transfieran datos (por ejemplo, arrastrar y soltar, cortar y pegar). Los mecanismos de intercambio de datos no sólo permiten la transferencia de datos humano-software y componente-componente, sino también la transferencia entre recursos del sistema (por ejemplo, arrastrar un archivo a un ícono de impresora para imprimirlo).

Automatización. Se deben implementar varias herramientas, macros y guiones para facilitar la interacción entre componentes reutilizables.

Almacenamiento estructurado. Los datos heterogéneos (por ejemplo, datos gráficos, voz, video, texto y datos numéricos) que contiene un "documento compuesto" deben estar organizados y ofrecer acceso como una sola estructura de datos y no como una colección de archivos separados. "Los datos estructurados conservan un índice descriptivo de estructuras anidadas por las cuales las aplicaciones pueden navegar libremente para ubicar, crear o editar contenidos de datos individuales según ordene el usuario final" [ADL95].

Referencia Web
La información más reciente acerca de CORBA se puede obtener en **www.omg.org.**

Modelo de objeto subyacente. El modelo de objeto asegura que los componentes desarrollados en diferentes lenguajes de programación y que residen en diferentes plataformas pueden ser interoperables. Es decir, los objetos deben ser capaces de comunicarse a través de una red. Esto se logra si el modelo de objeto define un estándar para la interoperabilidad de los componentes.

Puesto que el impacto potencial de la reutilización y la ISBC sobre la industria del software es enorme, varias grandes compañías y consorcios industriales han propuesto estándares para el software de componentes.

Referencia Web
La información más reciente acerca de COM se puede obtener en **www.microsoft.com/COM.**

OMG/CORBA. El Grupo de Gestión de Objetos (OMG, por sus siglas en inglés) ha publicado una *arquitectura común de distribución de objetos* (CORBA: por sus siglas en inglés). Un *distribuidor de objetos* (ORB, por sus siglas en inglés) proporciona una diversidad de servicios que permiten que los componentes reutilizables (objetos) se comuniquen con otros componentes, sin importar su ubicación dentro de un sistema.

COM de Microsoft. Microsoft ha desarrollado un *modelo de objetos para componentes* (COM, por sus siglas en inglés) que ofrece una especificación para utilizar componentes producidos por varias empresas dentro de una sola aplicación que corra bajo el sistema operativo Windows. El COM incluye dos elementos: interfaces COM (implementadas como objetos COM) y un conjunto de mecanismos que registra y pasa mensajes entre interfaces COM.

Componentes Sun JavaBeans. El sistema de componentes JavaBeans es una infraestructura de ISBC portátil e independiente de la plataforma que utiliza y desarrolla empleando el lenguaje de programación Java. El sistema de componentes JavaBeans incluye un conjunto de herramientas, llamado *Kit de Desarrollo Bean* (BDK, *Bean Development Kit*), que permite a los desarrolladores 1) analizar cómo funcionan los Beans existentes (componentes), 2) personalizar su comportamiento y apariencia, 3) establecer mecanismos para coordinación y comunicación, 4) desarrollar

SUGERENCIAS WEB

La información más reciente acerca de JavaBeans se puede obtener en **java.sun.com/products/javabeans/docs/**

Beans personalizados para usarlos en una aplicación específica, y 5) probar y evaluar el comportamiento Bean

¿Cuál de estos estándares dominará la industria? En este momento no existe una respuesta sencilla. Aunque muchos desarrolladores han adoptado uno de los estándares, tal vez las grandes organizaciones de software quieran optar por uno de los tres estándares, según las categorías de aplicación y las plataformas que elijan

INFORMACIÓN

Arquitectura común de distribución de objetos

Los sistemas cliente-servidor se implementan empleando componentes (objetos) de software que deben ser capaces de interactuar unos con otros dentro de una sola máquina (cliente o servidor) o a través de la red. Un *distribuidor de objetos* (ORB) es *middleware* (software personalizado) que permite que un objeto residente en un cliente envíe un mensaje a un método que está encapsulado en un objeto residente en un servidor. En esencia, el ORB intercepta el mensaje y maneja las actividades de comunicación y coordinación necesarias para encontrar el objeto al cual fue dirigido el mensaje, invoca su método, pasa los datos apropiados al objeto y transfiere los datos resultantes de vuelta al objeto que generó primero el mensaje.

CORBA, COM y JavaBeans implementan una filosofía de *distribuidor de objetos*. En este recuadro CORBA se usará para ilustrar el middleware ORB.

En la figura 30.2 se ilustra la estructura básica de una arquitectura CORBA. Cuando CORBA se implementa en un sistema cliente-servidor, los objetos servidores ambos se definen utilizando un *lenguaje de descripción de interfase* (IDL, *interface description language*), un lenguaje de declaraciones que permite que un ingeniero de software defina objetos, atributos, métodos y los mensajes que se requieren para invocarlos. Para que un objeto residente en el cliente acomode una petición para un método residente en el servidor se crean *stubs* (adaptadores) del IDL en el cliente y el servidor. Los *stubs* proporcionan la compuerta a través de la que se acomodan las peticiones de objetos a lo largo del sistema cliente-servidor.

Puesto que las peticiones de objetos a través de la red ocurren en tiempo de ejecución, se debe establecer un mecanismo para almacenar la descripción de objeto de modo que la información pertinente acerca del objeto y su ubicación estén disponibles cuando se necesite. El depósito de interfaz logra esto.

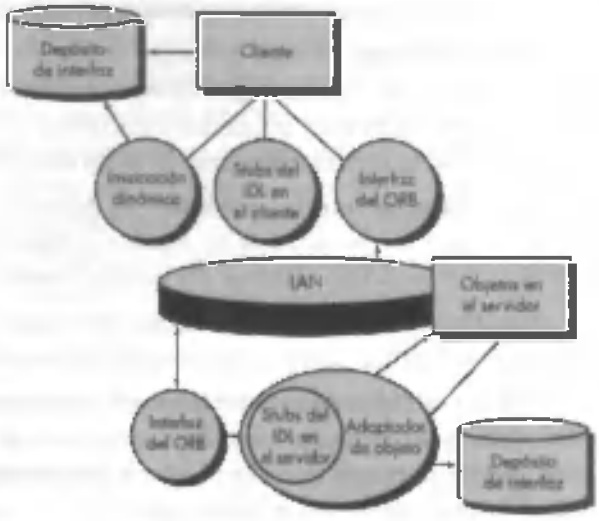
Cuando una aplicación en el cliente debe invocar un método ubicado dentro de un objeto en cualquier parte del sistema, CORBA utiliza invocación dinámica para 1) obtener información pertinente acerca del método deseado a partir del depósito de interfaz, 2) crear una estructura de datos con parámetros que pasarán al objeto, 3) crear una petición para el objeto, y 4) invocar la petición. Entonces la petición pasa al núcleo del ORB —una parte del sistema operativo de la red específica de la implementación que gestiona las peticiones— y se cumple la petición.

La petición pasa a través del núcleo y la procesa el servidor. En el sitio de éste un adaptador de objeto almacena información de la clase y el objeto en un depósito de interfaz residente en el servidor, acepta y gestiona las peticiones que llegan del cliente y realiza una diversidad de otras funciones de gestión de objetos. En el servidor stubs del IDL similares a los definidos en la máquina cliente se utilizan como la interfaz para la implementación del objeto real residente en el sitio del servidor.

30.4.2 Ingeniería de componentes

Como ya se indicó en este capítulo, el proceso de ISBC alienta el uso de componentes de software existentes. Sin embargo, hay ocasiones en que los componentes deben diseñarse. Es decir, se deben desarrollar nuevos componentes de software e integrarse con los CDL ya existentes y con los componentes de desarrollo propio. Puesto que los nuevos componentes se integran a la biblioteca propia de componentes reutilizables, deben diseñarse para su reutilización.

FIGURA 30.2
Arquitectura CORBA básica.



No hay nada mágico en la creación de componentes de software reutilizables. Los conceptos de diseño tales como abstracción, ocultación, independencia funcional, refinamiento y programación estructurada, junto con métodos orientados a objeto, pruebas de SQA y métodos de verificación de corrección, todos contribuyen a la creación de componentes de software reutilizables [3] En esta sección no se volverán a tratar estos temas. Más bien, se considerarán los temas específicos de la reutilización que complementan las prácticas sólidas de ingeniería del software.

30.4.3 Análisis y diseño para la reutilización

El modelo de análisis se analiza para determinar aquellos elementos del modelo que apuntan hacia los componentes reutilizables existentes El problema es extraer información a partir del modelo de requisitos en una forma que conduzca a la "concordancia de especificaciones".

Si la concordancia de especificaciones produce componentes que se ajustan con las necesidades de la aplicación actual, el diseñador puede extraer dichos componentes de una biblioteca (depósito) de reutilización y aplicarlos en el diseño de nuevos sistemas. Si no encuentra componentes de diseño, el ingeniero de software debe aplicar métodos de diseño convencional u OO para crearlos. En este punto —cuando el diseñador comienza a crear un nuevo componente— se debe considerar el *diseño para la reutilización* (DPR)

Como ya se indicó, el DPR requiere que el ingeniero de software aplique sólidos conceptos y principios de diseño de software (capítulo 9). Pero también se deben

3 Para aprender más acerca de estos conceptos véanse las partes 2 y 5 de este libro.

El DPR puede ser bastante difícil cuando los componentes deben estar en interfaz o integrados con sistemas heredados o con sistemas múltiples cuya arquitectura y protocolos de interfaz sean inconsistentes.

considerar la características del dominio de la aplicación. Binder [BIN93] sugiere varios temas clave[4] que forman una base para el diseño destinado a la reutilización

Datos estándar. Se debe investigar el dominio de la aplicación e identificar las estructuras de datos globales (por ejemplo, estructuras de archivos o una base de datos completa) Entonces se pueden caracterizar todos los componentes de diseño para aprovechar dichas estructuras de datos estándar

Protocolos de interfaz estándar. Se deben establecer tres niveles de protocolo de interfaz: la naturaleza de las interfaces intramodulares, el diseño de interfaces técnicas (no humanas) externas y la interfaz hombre-máquina.

Plantillas de programa. El modelo de estructura (sección 30.3.3) sirve como una plantilla para el diseño arquitectónico de un programa nuevo.

Una vez establecidas las interfases, los datos estándar y las plantillas de programa, el diseñador tiene un marco de trabajo en el que puede crear el diseño. Los nuevos componentes que se ajusten a este marco de trabajo tienen una mayor probabilidad de que se les reutilice posteriormente

30.5 Clasificación y recuperación de componentes

Considérese una biblioteca universitaria. Cientos de miles de libros, publicaciones periódicas y otras fuentes de información están disponibles para utilizarlos Pero el ingreso a dichas fuentes requiere desarrollar un sistema de clasificación. Para navegar por este gran volumen de información, los bibliotecarios han definido un sistema de clasificación que incluye el código de clasificación de la Biblioteca del Congreso (en los Estados Unidos de América), palabras clave, nombres de autor y otras entradas de índice. Todo esto permite que el usuario encuentre rápida y fácilmente la fuente requerida.

Ahora, considérese un gran depósito de componentes Cientos de miles de componentes de software reutilizable se hallan en él. Pero, ¿cómo encuentra un ingeniero de software el componente que necesita? Para responder esta pregunta surge otra: ¿cómo se describen los componentes de software en términos clasificables y sin ambigüedades? Éstas son preguntas difíciles y todavía no se ha desarrollado una respuesta definitiva. En esta sección se exploran las tendencias actuales que permitirán a los futuros ingenieros de software navegar entre las bibliotecas de reutilización.

30.5.1 Descripción de los componentes reutilizables

Un componente de software reutilizables se describe en muchas formas, pero una descripción ideal incluye lo que Tracz [TRA90] ha llamado el *modelo 3C*: concepto, contenido y contexto.

4 En general, se deben realizar preparativos para el DPR como parte de la ingeniería del dominio (sección 30 3).

El *concepto* de un componente de software es "una descripción de lo que hace el componente" [WHI95]. La interfaz con el componente está completamente descrita y la semántica —representada dentro del contexto de las precondiciones y las poscondiciones—, identificada. El concepto debe comunicar la intención del componente.

El *contenido* de un componente describe cómo se construye el concepto. En esencia, el contenido es información oculta para los usuarios habituales y que sólo necesitan conocerla quienes quieran modificar o probar el componente.

El *contexto* sitúa un componente de software reutilizable en su dominio de aplicabilidad. Es decir, al especificar las características conceptuales, operativas y de implementación el contexto permite que un ingeniero de software encuentre el componente apropiado para satisfacer los requisitos de la aplicación.

Para que sean útiles en la práctica, concepto, contenido y contexto se deben traducir en un esquema de especificación concreto. Se han escrito docenas de ensayos y artículos acerca de los esquemas de clasificación para componentes de software reutilizables (por ejemplo, [LUC01] y [WHI95] contienen bibliografías extensas). Los métodos propuestos se pueden clasificar en tres grandes áreas: métodos de biblioteconomía y de ciencias de la comunicación, métodos de inteligencia artificial y sistemas de hipertexto. La gran mayoría del trabajo realizado hasta la fecha sugiere el empleo de métodos de biblioteconomía para la clasificación de componentes.

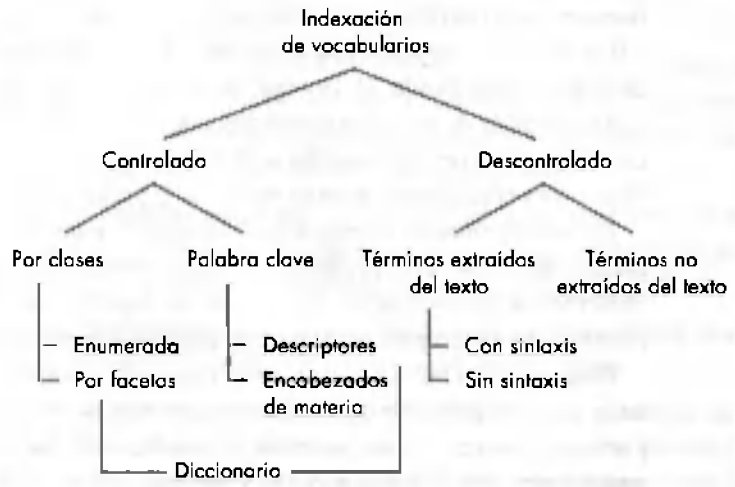
La figura 30.3 presenta una taxonomía de los métodos de indexación en la biblioteconomía. Los *vocabularios controlados de indexación* limitan los términos o sintaxis con que se clasifica un objeto (componente). Los *vocabularios de indexación no controlados* no ponen restricciones en la naturaleza de la descripción. La mayoría de los esquemas de clasificación para los componentes de software se incluye en tres categorías.

Clasificación enumerada. Los componentes se describen mediante una estructura jerárquica en la cual se definen las clases y los niveles variables de subclases de los componentes de software. La estructura jerárquica de un esquema de clasificación enumerada facilita comprenderlo y utilizarlo. Sin embargo, antes de construir una jerarquía se debe llevar a cabo la ingeniería del dominio de modo que haya suficiente conocimiento de las entradas adecuadas en la jerarquía.

Clasificación por facetas. Se analiza una área del dominio y se identifica un conjunto de características descriptivas básicas. Estas características, llamadas *facetas*, entonces se clasifican según su importancia y se conectan con un componente. Una faceta describe la función que el componente realiza, los datos que se manipulan, el contexto en el que se aplican o cualquiera otra característica. El conjunto de facetas que describe un componente se denomina *descriptor de facetas*. En general, la descripción por facetas se limita a no más de siete u ocho facetas.

Clasificación de valores y atributos. Un conjunto de atributos se define para todos los componentes en cierta área del dominio. Enseguida se asignan valores a dichos atributos en una forma muy similar a la clasificación por facetas. De hecho,

FIGURA 30.3
Taxonomía de los métodos de indexación [FRA94].



la clasificación de valores y atributos es similar a la clasificación por facetas, con las siguientes excepciones: 1) no se limita el número de atributos que se pueden utilizar, 2) no se asignan prioridades a los atributos, y 3) no se utiliza la función diccionario

Con base en un estudio empírico de cada una de estas técnicas de clasificación, Frakes y Pole [FRA94] indican que no existe una técnica claramente "mejor" y que "ningún método se desempeñó más que moderadamente en la eficacia de búsqueda..." Parecería que todavía falta realiza más trabajo en el desarrollo de esquemas de clasificación eficaces para bibliotecas de reutilización.

30.5.2 El entorno de reutilización

La reutilización de componentes de software debe apoyarla un entorno que incluya los siguientes elementos:

- Una base de datos de componentes capaz de almacenar componentes de software, así como la información de clasificación necesaria para recuperarlos.
- Un sistema de gestión de bibliotecas que ofrezca acceso a la base de datos.
- Un sistema de recuperación de componentes de software (por ejemplo, un distribuidor de objetos) que permita que una aplicación cliente recupere componentes y servicios del servidor de la biblioteca.
- Herramientas de ISBC que apoyen la integración de los componentes reutilizados en un nuevo diseño o implementación.

Cada una de estas funciones interactúa con o está incorporada en los confines de una biblioteca de reutilización.

La biblioteca de reutilización es un elemento de un depósito de software mayor (capítulo 27) y proporciona medios para el almacenamiento de componentes de software y una amplia gama de productos de trabajo reutilizables (por ejemplo, especificaciones, diseños, patrones, marcos de trabajo, fragmentos de código, casos de prueba, guías de usuario). La biblioteca contiene una base de datos y las herramientas necesarias para consultarla y recuperar componentes de ella. Un esquema de clasificación de componentes (sección 30.5.1) sirve como base para consultar la biblioteca.

Las consultas usualmente se caracterizan mediante el elemento contexto del modelo 3C ya descrito en esta sección. Si una consulta inicial resulta en una extensa lista de componentes candidatos, la consulta se refina para reducirla. Entonces se extrae la información de concepto y contenido (después de hallar los componentes candidatos) para auxiliar al desarrollador en la selección del componente apropiado.

Una descripción detallada de la estructura de las bibliotecas de reutilización y de las herramientas que las gestionan es mejor dejársela a las fuentes especializadas en la materia. El lector interesado obtendrá mayor información consultando [FIS00] y [LIN95].

Una extensa colección de recursos valiosos de ISBC se puede encontrar en http://www.cbd-hq.com/.

HERRAMIENTAS DE SOFTWARE

Desarrollo basado en componentes

Objetivo: Auxiliar en el modelado, diseño, revisión e integración de los componentes de software como parte de un sistema mayor.

Mecánica: La mecánica de las herramientas varía. En general, las herramientas de DBC auxilian en una o más de las siguientes capacidades: especificación y modelado de la arquitectura del software; navegación y selección de los componentes de software disponibles; integración de componentes.

Herramientas representativas[5]

ComponentSource (www.componentsource.com) proporciona un amplia serie de componentes (y herramientas) de software CDL apoyado en muchos estándares de componentes diferentes.

Component Manager, desarrollada por Flashline (www.flashline.com), "es una aplicación que posibilita, promueve y mide la reutilización de componentes de software"

Select Component Factory, desarrollada por Select Business Solutions (www.selectbs.com/products), "es un conjunto integrado de productos para el diseño de software, revisión de diseño, gestión de servicios y componentes, gestión de requisitos y generación de código"

Software Through Pictures-ACD, distribuida por Aonix (www.aonix.com), permite el modelado integral empleando UML para la arquitectura que rige el modelo OMG; un enfoque para la ISBC abierta e independiente de la empresa.

30.6 ECONOMÍA DE LA ISBC

La ingeniería del software basada en componentes tiene un atractivo intuitivo. En teoría, debe proporcionar a una organización de software ventajas en cuanto a cali-

5 Las herramientas expuestas sólo representan una muestra de esta categoría. En la mayoría de los casos los nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

Referencia Web
Una diversidad de artículos que ofrecen directrices para el DBC y los sistemas basados en CDL se puede encontrar en **www.sei.cmu.edu.**

dad y oportunidad, lo que debe traducirse en ahorros. Pero, ¿existen datos reales que apoyen esta intuición?

La respuesta a esta pregunta primero requiere entender lo que en realidad se puede reutilizar en un contexto de ingeniería del software y luego cuáles son en realidad los costos asociados con la reutilización. Como consecuencia, será posible desarrollar un análisis costo-beneficio para la reutilización de componentes

30.6.1 Impacto sobre la calidad, la productividad y el costo

Existen numerosas evidencias, a partir de estudios de caso industriales (por ejemplo, [ALL02], [HEN95], [MCM95]), que indican la posibilidad de derivar sustanciales beneficios de negocios a partir de la reutilización vigorosa del software. Mejoran la calidad del producto, la productividad de desarrollo y el costo global.

Calidad. En un entorno ideal, un componente de software que se desarrolle para reutilización se verificaría como correcto (véase el capítulo 29) y no contendría defectos. En realidad, la verificación formal no se lleva a cabo de manera rutinaria y existe la posibilidad de que ocurran defectos, y de hecho ocurren. Sin embargo, con cada reutilización los defectos se encuentran y eliminan, y, como resultado, mejora la calidad del componente. Con el tiempo el componente queda virtualmente libre de defectos

En un estudio realizado en Hewlett-Packard, Lim [LIM94] reportó que la tasa de defectos para el código reutilizado es de 0.9 defectos por KLDC, mientras que la tasa para software desarrollado recientemente es de 4.1 defectos por KLDC. En una aplicación compuesta de 68 por ciento de código reutilizado la tasa de defecto fue de 2.0 defectos por KLDC, es decir: un 51 por ciento de mejora respecto de la tasa esperada si la aplicación hubiese sido desarrollada sin reutilización. Henry y Faller [HEN95] reporta un 35 por ciento de mejora en la calidad. Aunque los reportes anecdóticos abarcan un espectro razonablemente amplio de porcentajes de mejora en la calidad, es justo afirmar que la reutilización ofrece un beneficio importante en cuanto a la calidad y fiabilidad para el software entregado.

Productividad. Cuando los componentes reutilizables se aplican a lo largo del proceso de software, se dedica menos tiempo a la creación de planes, modelos, documentos, código y datos que se requieren para crear un sistema fiable. Por lo tanto, se entrega al cliente el mismo nivel de funcionalidad con menos esfuerzo, lo que mejora la productividad. Aunque los reportes de mejora porcentual en la productividad son notablemente difíciles de interpretar,[6] parece que la reutilización del 30 al 50 por ciento puede resultar en mejoras en la productividad en el rango del 25-40 por ciento.

6 Muchas circunstancias atenuantes (por ejemplo, dominio de aplicación, complejidad del problema, estructura y tamaño del equipo, duración del proyecto, tecnología aplicada) tienen un profundo impacto sobre la productividad del equipo del proyecto

CONSEJO

El costo de desarrollar un componente reutilizable con frecuencia es mayor que el de desarrollar un componente específico para una aplicación. Asegúrese de que en el futuro habrá una necesidad respecto del componente reutilizable. Aquí es donde se realiza la retribución

Costo. Los ahorros en el costo neto por la reutilización se estiman al proyectar el costo del proyecto si éste fuese desarrollado desde cero, C_o, y luego se resta la suma de los costos asociados con la reutilización, C_r, y el costo real del software en el momento de la entrega, C_e.

El factor C_o se puede determinar al aplicar una o más de las técnicas de estimación estudiadas en el capítulo 23. Los costos asociados con la reutilización, C_r, incluyen [CHR95]: análisis y modelado del dominio, desarrollo de arquitectura del dominio, aumento en la documentación para facilitar la reutilización, soporte y mejora de los componentes de reutilización, regalías y licencias para componentes adquiridos externamente, creación o adquisición y operación de un depósito de reutilización, y entrenamiento del personal en diseño y construcción para reutilización. Aunque los costos asociados con el análisis del dominio (sección 30.3) y la operación de un depósito de reutilización pueden ser sustanciales, muchos de los otros costos indicados aquí abordan los conflictos que forman parte de una buena práctica de ingeniería de software, ya sea que la reutilización sea o no una prioridad.

30.6.2 Análisis de costo empleando puntos de estructura

En la sección 30.3.3 se definió un punto de estructura como un patrón arquitectónico recurrente en la totalidad de un dominio de aplicación particular. Un diseñador de software (o ingeniero de sistemas) puede desarrollar una arquitectura para una nueva aplicación, sistema o producto al definir una arquitectura del dominio y luego dotarla con puntos de estructura. Éstos son o componentes reutilizables individuales o paquetes de componentes reutilizables.

Aunque los puntos de estructura sean reutilizables, sus costos de cualificación, adaptación, integración y mantenimiento no son insignificantes. Antes de proceder a la reutilización el gestor del proyecto debe comprender los costos asociados con la utilización de los puntos de estructura.

Dado que todos los puntos de estructura (y los componentes reutilizables en general) tienen una historia, es posible recopilar datos de costos de cada uno. En un contexto ideal los costos de calificación, adaptación, integración y mantenimiento asociados con cada componente en una biblioteca de reutilización se mantienen para cada caso de utilización. Entonces se pueden analizar dichos datos para desarrollar los costos proyectados respecto del siguiente caso de reutilización.

Como ejemplo, considérese una nueva aplicación, X, que requiere 60 por ciento de código nuevo y la reutilización de tres puntos de estructura: PE_1, PE_2 y PE_3. Cada uno de estos componentes reutilizables se ha utilizado en muchas otras aplicaciones, y están disponibles los costos promedio para cualificación, adaptación, integración y mantenimiento.

La estimación del esfuerzo necesario para entregar X requiere determinar lo siguiente:

esfuerzo global = E_{nu} + E_{PE_1} + E_{PE_2} + E_{PE_3} + E_{int}

donde

E_{nuevo} = esfuerzo requerido para diseñar y construir nuevos componentes de software (determinados empleando las técnicas descritas en el capítulo 23)

E_{calif} = esfuerzo requerido para calificar PE_1, PE_2 y PE_3

E_{adapt} = esfuerzo requerido para adaptar PE_1, PE_2 y PE_3

E_{int} = esfuerzo requerido para integrar PE_1, PE_2 y PE_3

El esfuerzo requerido para cualificar, adaptar e integrar PE_1, PE_2 y PE_3 se determina al tomar el promedio de los datos históricos recopilados para la cualificación, adaptación e integración de los componentes reutilizables en otras aplicaciones.

30.7 Resumen

La ingeniería del software basada en componentes ofrece beneficios inherentes en la calidad del software, la productividad del desarrollador y el costo global del sistema. Sin embargo, falta superar muchos obstáculos antes de que el modelo de proceso de ISBC se utilice ampliamente en la industria.

Además de los componentes del software, un ingeniero de software puede adquirir una amplia gama de artefactos reutilizables. Entre éstos se encuentran representaciones técnicas del software (por ejemplo, especificaciones, modelos arquitectónicos, diseños), documentos, patrones, marcos de trabajo, datos de prueba e incluso tareas relacionadas con el proceso (por ejemplo, inspecciones técnicas).

El proceso de ISBC incluye dos subprocesos concurrentes: la ingeniería del dominio y el desarrollo basado en componentes. La finalidad de la ingeniería del dominio es identificar, construir, catalogar y diseminar un conjunto de componentes de software en un dominio de aplicación específico. Entonces el desarrollo basado en componentes califica, adapta e integra dichos componentes para emplearlos en un nuevo sistema. Además, el desarrollo basado en componentes diseña los componentes nuevos que se basan en los requisitos personalizados de un sistema nuevo.

Las técnicas de análisis y diseño para componentes reutilizables se basan en los mismos principios y conceptos que forman parte de una buena práctica de ingeniería del software. Los componentes reutilizables deben diseñarse en un entorno que establezca estructuras de datos estándar, protocolos de interfaz y arquitecturas de programa para cada dominio de la aplicación.

La ingeniería del software basada en componentes utiliza un modelo de intercambio de datos, herramientas, almacenamiento estructurado y un modelo de objeto subyacente para construir las aplicaciones. El modelo de objeto generalmente concuerda con uno o más estándares de componentes (por ejemplo, OMG/CORBA) que definen la forma en que una aplicación puede acceder a los objetos reutilizables. Los esquemas de clasificación permiten que un desarrollador encuentre y recupere componentes reutilizables y se ajuste a un modelo que identifica concepto, contenido y contexto. La clasificación enumerada, la clasificación por facetas y la clasificación de valores y atributos son representativas de muchos esquemas de clasificación de componentes.

La economía de la reutilización del software se aborda mediante una sola pregunta: ¿es efectivo en costo el construir menos y reutilizar más? En general, la respuesta es sí, pero un planificador de proyectos de software debe considerar los costos importantes asociados con la calificación, adaptación e integración de los componentes reutilizables.

Referencias

[ADL95] Adler, R. M., "Emerging Standards for Component Software", en *Computer*, vol 28, núm. 3, marzo de 1995, pp. 68-77.

[ALL02] Allen, P., "CBD Survey. The State of the Practice", en *The Cutter Edge*, marzo de 2002, disponible en http://www cutter.com/research/2002/edge020305.html.

[ATK01] Atkinson, C., et al., *Component-Based Product Line Engineering with UML*, Addison-Wesley, 2001.

[BAS94] Basili, V. R., L. C. Briand y W. M. Thomas, "Domain Analysis for the Reuse of Software Development Experiences", *Proc. Of the 19th Annual Software Engineering Workshop*, NASA/GSFC, Greenbelt, MD, diciembre de 1994

[BIN93] Binder, R., "Design for Reuse is for Real", en *American Programmer*, vol. 6, núm. 8, agosto de 1993, pp. 30-37

[BRO96] Brown, A. W., y K. C. Wallnau, "Engineering of Component-Based Systems", en *Component-Based Software Engineering*, IEEE Computer Society Press, 1996, pp. 7-15

[CHR95] Christensen, S. R., "Software Reuse Initiatives at Lockheed", en *CrossTalk*, vol. 8, núm. 5, mayo de 1995, pp. 26-31.

[CLE95] Clements, P. C., "From Subroutines to Subsystems: Component-Based Software Development", en *American Programmer*, vol. 8, núm. 11, noviembre de 1995.

[DOG03] Dogru, A., y M. Tanik, "A Process Model for Component-Oriented Software Engineering", en *IEEE Software*, vol. 20, núm. 2, marzo-abril 2003, pp. 34-41

[FIS00] Fischer, B., "Specification-Based Browsing of Software Component Libraries", en *J. Automated Software Engineering*, vol. 7, núm. 2, 2000, pp. 179-200, disponible en http://ase.arc-.nasa.gov/people/fischer/papers/ase-00.html.

[FRA94] Frakes, W. B., y T. P. Pole, "An Empirical Study of Representation Methods for Reusable Software Components", en *IEEE Trans. Software Engineering*, vol. SE-20, núm. 8, agosto de 1994, pp. 617-630

[HEI01] Heineman, G., y W. Councill (eds.), *Component-Based Software Engineering*, Addison-Wesley, 2001.

[HEN95] Henry, E., y B. Faller, "Large Scale Industrial Reuse to Reduce Cost and Cycle Time", en *IEEE Software*, septiembre de 1995, pp. 47-53.

[HUT88] Hutchinson, J. W., y P. G. Hindley, "A Preliminary Study of Large Scale Software Reuse", en *Software Engineering Journal*, vol. 3, núm. 5, 1988, pp. 208-212

[LIA93] Liao, H., y Wang, F., "Software Reuse Based on a Large Object-Oriented Library", en *ACM Software Engineering Notes*, vol. 18, núm. 1, enero de 1993, pp. 74-80.

[LIM94] Lim, W. C., "Effects of Reuse on Quality, Productivity, and Economics", en *IEEE Software*, septiembre de 1994, pp. 23-30

[LIN95] Linthicum, D. S., "Component Development (a Special Feature)", *Application Development Trends*, junio de 1995, pp. 57-78.

[LUC01] deLucena, Jr., V., "Facet-Based Classification Scheme for Industrial Software Components", 2001, se puede descargar de http://research microsoft com/users/cszypers/events/WCOP2001/Lucena pdf

[MCM95] McMahon, P. E., "Pattern-Based Architecture: Bridging Software Reuse and Cost Management", en *Crosstalk*, vol. 8, núm. 3, marzo de 1995, pp. 10-16.

[ORF96] Orfali, R., D. Harkey y J. Edwards, *The Essential Distributed Objects Survival Guide*, Wiley, 1996.

[PRI93] Prieto-Díaz, R., "Issues and Experiences in Software Reuse", en *American Programmer*, vol. 6, núm. 8, agosto de 1994, pp. 67-68

[POL94] Pollak, W y M. Rissman, "Structural Models and Patterned Architectures", en *Computer* vol. 27, núm. 8, agosto de 1994, pp.67-68.
[STA94] Staringer, W., "Constructing Applications from Reusable Components", en *IEEE Software*, septiembre de 1994, pp 61-68.
[TRA90] Tracz, W., "Where Does Reuse Start?", *Proc Realities of Reuse Workshop*, Syracuse University CASE Center, enero de 1990
[TRA95] Tracz, W., "Third International Conference on Software Reuse-Summary", en *ACM Software Engineering Notes*, vol. 20, núm. 2, abril de 1995, pp 21-22.
[WHI95] Whittle, B., "Models and Languages for Component Description and Reuse", en *ACM Software Engineering Notes*, vol 20, núm. 2, abril de 1995, pp. 76-89
[YOU98] Yourdon, E. (ed.), "Distributed Objects", en *Cutter IT Journal*, vol. 11, núm. 12, diciembre de 1998.

Problemas y puntos a considerar

30.1. Uno de los obstáculos clave para la reutilización consiste en hacer que los desarrolladores de software consideren la reutilización de componentes existentes en lugar de reinventar unos nuevos (después de todo, ¡construir cosas es divertido!) Sugerir de tres a cuatro formas diferentes en que una organización de software puede ofrecer incentivos para que los ingenieros de software empleen la reutilización ¿Qué tecnologías deben existir para apoyar el esfuerzo de la reutilización?

30.2. Aunque los componentes de software son los "artefactos" reutilizables más obvios, se pueden reutilizar muchos otros productos de trabajo producidos como parte de la ingeniería del software. Considerar los planes de proyecto y las estimaciones de costo. ¿Cómo se pueden reutilizar y cuál es el beneficio de hacerlo?

30.3. Investíguese un poco acerca de la ingeniería del dominio y detállese el modelo de proceso esbozado en la figura 30.1. Identifíquense las tareas que se requieren para el análisis del dominio y el desarrollo arquitectónico del software

30.4. ¿En qué son semejantes las funciones de caracterización para dominios de aplicación y los esquemas de clasificación de componentes? ¿En qué se diferencian?

30.5. Desarrollar un conjunto de características de dominio para sistemas de información que sean relevantes respecto al procesamiento de datos de estudiantes de una universidad

30.6. Desarrollar un conjunto de características de dominio que sean relevantes para un software de procesamiento de textos y publicación.

30.7. Desarrollar un modelo estructural simple para un dominio de aplicación que asigne individualmente el instructor o uno con el cual se esté familiarizado.

30.8. ¿Qué es un punto de estructura?

30.9. Adquirir información acerca del más reciente estándar CORBA, COM o JavaBeans y elaborar un ensayo de tres a cinco páginas que aborde sus principales atributos Obtener información acerca de una herramienta de distribución de solicitudes de objetos e ilustrar cómo la herramienta se ajusta al estándar

30.10. Desarrollar una clasificación enumerada para un dominio de aplicación que asigne el instructor o uno con el que se esté familiarizado.

30.11. Desarrollar un esquema de clasificación por facetas para un dominio de aplicación que asigne el instructor o uno con el que se esté familiarizado

30.12. Investíguese en la bibliografía para obtener datos recientes de calidad y productividad que apoyen el uso de la ISBC Preséntense los datos a la clase.

Otras lecturas y fuentes de información

En años recientes se han publicado muchos libros acerca del desarrollo basado en componentes y la reutilización de éstos. Heineman y Councill [HEI01], Brown (*Large Scale Component-Based Development*, Prentice-Hall, 2000), Allen (*Realizing e-Business with Components*, Addison-Wesley, 2000), Herzum y Sims (*Business Component Factory*, Wiley, 1999) y Allen, Frost y Yourdon (*Component-Based Development for Enterprise Systems: Applying the Select Perspective*, Cambridge University Press, 1998) cubren todos los aspectos importantes del proceso de ISBC. Apperly y sus colegas (*Service- and Component-Based Development*, Addison-Wesley, 2003), Atkinson [ATK01] y Cheesman y Daniels (*UML Components*, Addison-Wesley, 2000) examinan la ISBC poniendo especial cuidado en UML.

Leach (*Software Reuse: Methods, Models, and Costs*, McGraw-Hill, 1997) proporciona un análisis detallado de los conflictos de costo asociados con la ISBC y la reutilización. Poulin (*Measuring Software Reuse: Principles, Practices, and Economic Models*, Addison-Wesley, 1996) sugieren algunos métodos cuantitativos para valorar los beneficios de la reutilización de software.

En años recientes se han publicado docenas de libros que describen los estándares basados en componentes de la industria. En ellos se abordan las funciones internas de los estándares mismos, pero también consideran muchos tópicos importantes de la ISBC. A continuación se presenta una muestra de los tres estándares estudiados en este capítulo:

CORBA

Bolton, F., *Pure CORBA*, Sams Publishing, 2001.

Doss, G. M., *CORBA Networking With Java*, Wordware Publishing, 1999.

Hoque, R., *CORBA for Real Programmers*, Academic Press/Morgan Kaufmann, 1999.

Siegel, J., *CORBA Fundamentals and Programming*, Wiley, 1999.

Slama, D., J. Garbis y P. Russell, *Enterprise CORBA*, Prentice-Hall, 1999.

COM

Box, D., K. Brown, T. Ewald y C. Sells, *Effective COM: 50 Ways to Improve Your COM and MTS-Based Applications*, Addison-Wesley, 1999.

Gordon, A., *The COM and COM+ Programming Primer*, Prentice-Hall, 2000.

Kirtland, M., *Designing Component Based Applications*, Microsoft Press, 1999.

Tapadiya, P., *COM+ Programming*, Prentice-Hall, 2000.

Muchas organizaciones aplican una combinación de estándares de componentes. Los libros de Geraghty y sus colegas (*COM-CORBA Interoperability*, Prentice-Hall, 1999), Pritchard (*COM and CORBA Side by Side: Architectures, Strategies, and Implementations*, Addison-Wesley, 1999), y Rosen y sus colegas (*Integrating CORBA y COM Applications*, Wiley, 1999) consideran los conflictos asociados con el uso tanto de CORBA como de COM como la base para el desarrollo basado en componentes.

JavaBeans

Asbury, S., y S. R. Weiner, *Developing Java Enterprise Applications*, Wiley, 1999.

Anderson, G., y P. Anderson, *Enterprise Javabeans Component Architecture*, Prentice-Hall, 2002.

Monson-Haefel, R., *Enterprise Javabeans*, tercera edición, O'Reilly & Associates, 2001.

Roman, E., *et al.*, *Mastering Enterprise Javabeans*, 2a. ed., Wiley, 2001.

En Internet hay disponible una amplia variedad de fuentes de información acerca de la ingeniería del software basada en componentes. Una lista actualizada de referencias en la World Wide Web se puede encontrar en el sitio Web SEPA:
http://www.mhhe.com/pressman.

CAPÍTULO

31 REINGENIERÍA

CONCEPTOS CLAVE

En un relevante artículo publicado en la *Harvard Business Review*, Michael Hammer [HAM90] sentó las bases para una revolución en el pensamiento administrativo acerca de los procesos de negocios y la computación:

> Es el momento de dejar de pavimentar los senderos para vacas. En lugar de incrustar procesos anticuados en silicio y software, debemos eliminarlos y comenzar de nuevo. Debemos someter a "reingeniería" nuestros negocios: usar el poder de las modernas tecnologías de la información para rediseñar radicalmente nuestros procesos de negocios con la finalidad de alcanzar mejoras radicales en su desempeño.
>
> Cualquier compañía opera de acuerdo con una gran cantidad de reglas desarticuladas . . . La reingeniería lucha por separarse de las viejas reglas acerca de cómo organizar y dirigir nuestros negocios.

Al igual que todas las revoluciones, el llamado a las armas de Hammer generó cambios positivos y negativos. Durante el decenio de 1990, algunas compañías aplicaron un esfuerzo legítimo en la realización de reingeniería, y los resultados las condujeron a mejorar su competitividad. Otras se apoyaron exclusivamente en el redimensionamiento y la subcontratación (en lugar de la reingeniería) para mejorar sus líneas base; por lo tanto, con frecuencia resultaron organizaciones con poco potencial para un crecimiento futuro [DEM95].

Durante esta primera década del siglo XXI, la promoción exagerada de la reingeniería ha decaído, pero el proceso en sí continúa en compañías grandes y pequeñas. Los nexos entre la reingeniería de negocios e ingeniería del software se encuentran en una revisión de sistema.

UN VISTAZO RÁPIDO

¿Qué es? Considere cualquier producto tecnológico que le haya servido bien. Usted lo utiliza regularmente, pero se está volviendo obsoleto. Se rompe con mucha frecuencia, su reparación toma más tiempo del que usted quisiera, y ya no representa más la nueva tecnología. ¿Qué hacer? Si el producto es hardware, probablemente usted lo tirará a la basura y comprará un modelo más nuevo. Pero si es software personalizado, dicha opción tal vez no esté disponible. Necesitará reconstruirlo. Creará un producto con una mejor funcionalidad, mejor desempeño y fiabilidad, así como una mejor facilidad de mantenimiento. A eso se le llama reingeniería.

¿Quién lo hace? En el ámbito de las organizaciones, la reingeniería la llevan a cabo especialistas en negocios (con frecuencia compañías consultoras). En el ámbito del software la reingeniería la realizan los ingenieros de software.

¿Por qué es importante? Se vive en un mundo en cambio constante. Las demandas acerca de las funciones de negocios y la tecnología de la información que las soportan están cambiando a un ritmo que impone una enorme presión competitiva en las

organizaciones comerciales. Tanto el negocio como el software que soporta (o es) el negocio debe rediseñarse para mantener el ritmo.

¿Cuáles son los pasos? La reingeniería de procesos de negocio (RPN) define las metas del negocio, identifica y evalúa los procesos vigentes del negocio, y crea procesos de negocios renovados que cumplen mejor las metas actuales. El proceso de reingeniería de software incluye análisis de inventarios, reestructuración de documentos, ingeniería inversa, reestructuración de programas y datos, e ingeniería avanzada. La finalidad de estas actividades es crear versiones de programas existentes que muestren mayor calidad y mejor facilidad de mantenimiento.

¿Cuál es el producto obtenido? Se produce una diversidad de productos de trabajo de reingeniería (por ejemplo, modelos de análisis, modelos de diseño, procedimientos de prueba). El resultado final es un proceso de reingeniería de negocios o el software de reingeniería que lo soporta.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Utilice las mismas prácticas de SQA que se aplican a cualquier proceso de ingeniería del software: las revisiones técnicas formales evalúan los modelos de análisis y de diseño; las revisiones especializadas consideran la aplicabilidad y la compatibilidad en el negocio, y las pruebas se aplican para descubrir errores en contenido, funcionalidad e interoperabilidad.

PUNTO CLAVE

La RPN con frecuencia genera una nueva funcionalidad de software, mientras que la reingeniería del software trabaja para reemplazar la funcionalidad del software existente con un mejor software y de mayor facilidad de mantenimiento

Usualmente el software es la realización de las reglas del negocio que Hammer describe. Conforme cambien dichas reglas, el software también debe hacerlo. En la actualidad, las grandes compañías tienen decenas de miles de programas de computadora que apoyan a las viejas reglas del negocio. A medida que los administradores trabajen en la modificación de las reglas y logren mayores efectividad y competitividad, el software debe mantener el ritmo. En algunos casos, esto implica la creación de mayores y nuevos sistemas basados en computadora.[1] Pero, en muchos otros, significa la modificación o reconstrucción de las aplicaciones existentes.

En este capítulo se examina la reingeniería en forma descendente, comenzando con un breve panorama de la reingeniería de procesos de negocio y después se abordan con mayor detalle las actividades técnicas que se llevan a cabo al realizar la reingeniería del software.

31.1 Reingeniería de procesos de negocio

La reingeniería de procesos de negocio (RPN) rebasa el ámbito de las tecnologías de la información y de la ingeniería del software. Entre las muchas definiciones (la mayoría un tanto abstracta) sugeridas para la RPN destaca una publicada en la revista *Fortune* [STE93]: "La búsqueda e implementación de un cambio radical en el proceso de negocios para lograr resultados de vanguardia". Pero, ¿cómo se lleva a cabo la búsqueda y cómo se logra la implementación? Más importante aún, ¿cómo se puede garantizar que el "cambio radical" sugerido conducirá a "resultados de vanguardia" en lugar de caos organizacional?

1 La explosión de aplicaciones y sistemas basados en Web estudiados en la parte 3 de este libro es un indicio de esta tendencia.

"Enfrentar el mañana con la idea de emplear los métodos de ayer es visualizar la vida como una parálisis."
James Bell

31.1.1 Procesos de negocios

Un proceso de negocio es "un conjunto de tareas lógicamente relacionadas que se ejecutan para lograr un resultado de negocios específico" [DAV90]. Dentro del proceso de negocio, la gente, el equipo, los recursos materiales y los procedimientos del negocio se combinan para producir un resultado específico. Los ejemplos de procesos de negocios incluyen el diseño de un nuevo producto, la compra de servicios y suministros, la contratación de un nuevo empleado y el pago a proveedores. Cada uno demanda un conjunto de tareas y también emplea diversos recursos dentro del negocio.

Cada proceso de negocio tiene un cliente definido: una persona o grupo que recibe el resultado (por ejemplo, una idea, un informe, un diseño, un producto). Además, los procesos de negocios traspasan las fronteras de la organización. Esto requiere que diferentes grupos de organizaciones participen en las "tareas lógicamente relacionadas" que definen el proceso.

CONSEJO

Como ingeniero de software, su trabajo la desempeña en la base de esta jerarquía. Sin embargo, asegúrese de que alguien ha pensado seriamente en los niveles superiores. Si esto no ha ocurrido, su trabajo está en riesgo

En el capítulo 6 se indicó que todo sistema es en realidad una jerarquía de subsistemas. Un negocio no es la excepción. Cada sistema de negocio (también llamado una función negocio) está compuesto de uno o más procesos de negocio, y a cada proceso de negocio lo define un conjunto de subprocesos.

La RPN se puede aplicar en cualquier nivel de la jerarquía, pero conforme se amplía su ámbito (es decir, conforme uno se mueve hacia arriba en la jerarquía) los riesgos asociados con ello crecen sustancialmente. Por esta razón, la mayoría de los esfuerzos de la RPN se enfoca en procesos individuales o subprocesos

"Tan pronto se nos presenta algo viejo en una cosa nueva, nos tranquilizamos."
F. W. Nietzsche

31.1.2 Un modelo de RPN

Referencia Web

Amplia información acerca de la RPN se puede encontrar en www.brint.com/BPR.htm.

Como la mayoría de las actividades de ingeniería, la reingeniería de procesos de negocio es iterativa. Las metas del negocio y los procesos con que se logran se deben adaptar a un entorno de negocios cambiante. Por esta razón no existe principio ni fin para la RPN: se trata de un proceso evolutivo. En la figura 31.1 se muestra un modelo de reingeniería de procesos de negocio. El modelo define seis actividades

Definición del negocio. Las metas del negocio se identifican dentro del contexto de cuatro controladores clave: reducción de costo, reducción de tiempos, mejora de la calidad y desarrollo y fortalecimiento del personal. Es posible definir las metas al nivel del negocio o respecto de un componente específico del negocio.

Identificación del proceso. Se identifican los procesos cruciales para lograr las metas precisadas en la definición del negocio. Luego podría clasificarse de acuerdo

con su importancia, necesidad de cambio o en cualquier otra forma que sea adecuada para la actividad de reingeniería.

Evaluación del proceso. El proceso existente se analiza y mide exhaustivamente. Se identifican las tareas del proceso; se anotan los costos y el tiempo que consumen las tareas del proceso; y se aíslan los problemas de calidad y desempeño

Especificación y diseño del proceso. Con base en la retroalimentación obtenida durante las primeras tres actividades de la RPN, se preparan casos de uso (capítulo 7) para cada proceso que será rediseñado. En del contexto de la RPN los casos de uso identifican un escenario que entrega cierto resultado a un cliente. Con el cas de uso como la especificación del proceso se diseña un nuevo conjunto de tareas para el proceso.

Elaboración de prototipos. Un proceso de negocio rediseñado debe convertirse en prototipo antes de que sea integrado por completo en el negocio. Esta actividad "prueba" el proceso de modo que puedan llevarse a cabo refinamientos.

Refinamiento y particularización. Con base en la retroalimentación del prototipo, el proceso de negocio se refina y luego se particulariza dentro de un sistema de negocio.

En ocasiones, estas actividades de RPN se utilizan junto con las herramientas de análisis de flujo de trabajo La finalidad de estas herramientas es elaborar un modelo de flujo de trabajo práctico, esfuerzo encaminado a analizar mejor los procesos existentes. Además, para implementar las primeras cuatro actividades descritas en el modelo de proceso se pueden aplicar las técnicas de modelado usualmente asociadas con las actividades de ingeniería de procesos de negocio (capítulo 6).

FIGURA 31.1
Modelo de RPN.

HERRAMIENTAS DE SOFTWARE

Reingeniería de procesos de negocio (RPN)

Objetivo: El objetivo de las herramientas de la RPN es apoyar el análisis y la evaluación de los procesos de negocio existentes y la especificación y el diseño de unos nuevos.

Mecánica: La mecánica de las herramientas varía. En general, las herramientas de la RPN permiten que un analista de negocios modele los procesos de negocio existentes en un esfuerzo destinado a evaluar las ineficiencias del flujo de trabajo o problemas funcionales. Una vez que se identifican los problemas existentes las herramientas permiten que los analistas elaboren prototipos o simulen procesos de negocio revisados.

Herramientas representativas[2]

Extend, desarrollada por ImagineThat, Inc. (www.imaginethatinc.com), es una herramienta de simulación para el modelado de procesos existentes y la exploración de unos nuevos. Extend proporciona una extensa capacidad "si... entonces" que permite que un analista de negocios explore diferentes escenarios de proceso.

e-Work, desarrollada por Metastorm (www.metastorm.com), apoya la gestión de procesos de negocios en procesos manuales y automatizados

IceTools, desarrollados por Blue Ice (www.blueice.com), es una colección de plantillas de RPN para Microsoft Office y Microsoft Project.

SpeedDev, desarrollada por NimbleStar Group (www.nimblestar.com), es una de muchas herramientas que permiten que una organización modele flujos de trabajo de procesos (en este caso, flujo de trabajo de TI).

Workflow tools, desarrolladas por MetaSoftware (www.metasoftware.com), incorpora un conjunto de herramientas para modelado, simulación y calendarización del flujo de trabajo.

Una lista útil de vínculos a herramientas de RPN se puede encontrar en http://www.donald-firesmith.com/Components/Producers/Tools/BusinessProcessReengineeringTools.html.

31.2 REINGENIERÍA DEL SOFTWARE

El escenario es tan común: una aplicación ha cubierto las necesidades de negocios de una compañía durante 10 o 15 años. Durante ese lapso se ha corregido, adaptado y mejorado muchas veces. El personal enfrentó este trabajo con las mejores intenciones, pero las buenas prácticas de ingeniería del software siempre fueron soslayadas (debido a la presión de otros asuntos importantes). Ahora la aplicación es inestable. Todavía funciona pero, cada vez que se intenta un cambio, ocurren efectos colaterales, inesperados y serios. Y la aplicación todavía tiene que evolucionar. ¿Qué hacer?

El software al cual no se le puede dar mantenimiento no es un problema nuevo. De hecho, la importancia cada vez mayor que se le concede a la reingeniería del software la han impulsado los problemas en el mantenimiento del software que han ido creciendo durante más de 40 años.

2 Las herramientas expuestas el autor no las respalda; sólo representan una muestra de las herramientas incluidas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

31.2.1 Mantenimiento del software

Durante las tres décadas pasadas el mantenimiento del software se caracterizó [CAN72] como un "iceberg". Se esperaba que lo inmediatamente visible fuese todo lo que había, pero se sabía que bajo la superficie se encontraba una enorme masa de problemas y costos potenciales. A principios del decenio de 1970, el mantenimiento del iceberg era suficientemente grande como para hundir un portaaviones. En la actualidad, ¡fácilmente podría hundir a toda la marina!

El mantenimiento del software existente explica casi el 60 por ciento del esfuerzo que emplea una organización de desarrollo, y el porcentaje continúa elevándose conforme se produce más software [HAN93]. Los lectores con escasos conocimientos sobre el tema podrían preguntarse por qué se requiere tanto mantenimiento y por qué se dedica tanto esfuerzo. Osborne y Chikofsky [OSB90] ofrecen una respuesta parcial:

> Gran parte del software del que dependemos en la actualidad tiene en promedio de 10 a 15 años de antigüedad. Aun cuando dichos programas se crearon empleando las mejores técnicas de diseño y codificación conocidas en la época [y la mayoría no lo eran], se crearon cuando el tamaño de los programas y el espacio de almacenamiento eran las principales preocupaciones. Entonces emigraron hacia nuevas plataformas, se ajustaron para adecuarlos a los cambios en las máquinas y a la tecnología de los sistemas operativos y aumentaron para satisfacer las necesidades de nuevos usuarios; todo se hizo sin considerar lo suficiente la arquitectura global. El resultado es estructuras mal diseñadas, codificación deficiente, lógica inadecuada y escasa documentación de los sistemas de software por los que ahora se nos llama para mantenerlos en operación...

Otra razón respecto del problema del mantenimiento del software es la movilidad del personal. Es probable que el equipo (o persona) de software que realizó el trabajo original ya no esté. Peor aún, las generaciones subsecuentes de personal han modificado el sistema y lo han dañado. En la actualidad, tal vez no haya nadie que tenga algún conocimiento directo del sistema heredado.

Como se indicó en el capítulo 27, la naturaleza ubicua del cambio subyace en todo el trabajo de software. El cambio es inevitable cuando se construyen sistemas basados en computadora; en consecuencia, se deben desarrollar mecanismos para evaluar, controlar y efectuar modificaciones.

> "La facilidad de mantenimiento de los programas y la comprensibilidad de los programas son conceptos paralelos: mientras más difícil sea comprender un programa, más difícil será su mantenimiento."
>
> **Gerald Berns**

Después de leer los párrafos anteriores un lector podría protestar: "Pero yo no paso el 60 por ciento de mi tiempo componiendo errores en los programas que desarrollo". Desde luego, el mantenimiento del software es mucho más que "componer errores". Es posible definir el mantenimiento describiendo cuatro actividades [SWA76] que se realizan después de que un programa es liberado para su utilización

PUNTO CLAVE

El mantenimiento del software incluye cuatro actividades: corrección de error, adaptación, mejora y reingeniería.

El mantenimiento del software se define identificando cuatro actividades diferentes mantenimiento correctivo, mantenimiento adaptativo, mejora o mantenimiento de perfeccionamiento y mantenimiento preventivo o reingeniería. Sólo cerca del 20 por ciento del trabajo de mantenimiento se emplea en "componer errores". El restante 80 por ciento se dedica a adaptar los sistemas existentes a los cambios en su entorno externo, realizar las mejoras que solicitan los usuarios y rediseñar una aplicación para usarla en lo futuro. Al considerar que el mantenimiento incluye todas estas actividades es relativamente sencillo observar por qué absorbe tanto esfuerzo.

31.2.2 Un modelo de procesos de reingeniería del software

Referencia Web

Una excelente fuente de información acerca de la reingeniería del software se puede encontrar en www.reengineering.net.

La reingeniería requiere tiempo, cuesta cantidades significativas de dinero y absorbe recursos que de otro modo se ocuparían en problemas inmediatos. Por todas estas razones la reingeniería no se logra en unos cuantos meses, ni siquiera en unos cuantos años. La reingeniería de los sistemas de información es una actividad que absorberá recursos de la tecnología de la información durante muchos años. Por tanto, toda organización necesita una estrategia pragmática respecto de la reingeniería del software.

El modelo de proceso de reingeniería incluye una estrategia operativa. El modelo se tratará más adelante en esta sección, pero primero se presentarán algunos principios básicos.

La reingeniería es una actividad de reconstrucción, y la reingeniería de los sistemas de información se comprende mejor si se considera una actividad análoga: la reconstrucción de una casa. Considérese la siguiente situación.

El lector compra una casa en otro estado. En realidad nunca ha visto la propiedad, pero la adquirió en un precio sorprendentemente bajo, con la advertencia de que tal vez tenga que reconstruirse por completo. ¿Cómo se procedería?

- Antes de que se pueda iniciar la reconstrucción sería razonable inspeccionar la casa. Determinar si es necesario reconstruirla requiere que el lector (o un inspector profesional) elabore una lista de criterios de modo que la inspección resulte sistemática
- Antes de tirar y reconstruir toda la casa se debe tener la certeza que la estructura es débil. Si la casa es estructuralmente sólida tal vez sea posible "remodelarla" sin reconstruirla (a un costo mucho más bajo y en mucho menos tiempo).
- Antes de iniciar la reconstrucción se debe tener la certeza de que se entiende cómo se construyó la original. Eche un vistazo detrás de las paredes. Entienda el alambrado, la plomería y los componentes estructurales. Incluso si se tiran todos a la basura, la comprensión que se adquiera será útil cuando comience la construcción.
- Si se comienza a reconstruir sólo se utilizarán los materiales más modernos y de larga duración. Esto puede costar un poco más ahora, pero ayudará a evitar un mantenimiento costoso y tardado más adelante

- Si se decide reconstruir es preciso disciplinarse en cuanto a ello Utilícense prácticas que redundarán en alta calidad, hoy y mañana

Aunque estos principios se enfocan en la reconstrucción de una casa, también se aplican igualmente bien a la reingeniería de sistemas y aplicaciones basadas en computadoras.

La implementación de estos principios requiere aplicar un modelo de proceso de reingeniería del software que define seis actividades, como se muestra en la figura 31.2. En algunos casos dichas actividades ocurren en una secuencia lineal, pero éste no siempre es el caso. Por ejemplo, tal vez la ingeniería inversa (comprender el funcionamiento interno de un programa) tenga que ocurrir antes de que comience la reestructuración de documentos

El paradigma de reingeniería que se muestra en la figura es un modelo cíclico. Esto significa que cada una de las actividades presentadas como parte del paradigma pueden volver a visitarse. En algún ciclo particular el proceso quizá termine después de cualquiera de dichas actividades.

Análisis de inventarios. Las organizaciones de software deberían tener un inventario de todas sus aplicaciones El inventario tal vez no sea más que un modelo en una hoja de cálculo que contenga información que proporcione una descripción detallada (por ejemplo, tamaño, edad, importancia para el negocio) de las aplicaciones activas. Al ordenar esta información —de acuerdo con la importancia para el negocio, antigüedad, facilidad actual de mantenimiento y otros criterios localmente importantes— aparecen los candidatos para reingeniería. Entonces se pueden asignar los recursos a las aplicaciones candidatas para el trabajo de reingeniería

FIGURA 31.2

Modelo de proceso de la reingeniería del software.

Si el tiempo y los recursos escasean, puede considerar la aplicación del principio de Pareto al software que será sometido a ingeniería. Aplique el proceso de reingeniería al 20 por ciento del software que explica el 80 por ciento de los problemas.

Cree solamente tanta documentación como necesite para entender el software, ni una página más.

Es importante señalar que el inventario deberá visitarse con regularidad. El estado de las aplicaciones (por ejemplo, importancia respecto del negocio) puede cambiar en función del tiempo y, como resultado, cambiarán las prioridades para la reingeniería.

Reestructuración de documentos. La documentación débil es la marca de muchos sistemas heredados ¿Pero qué se hace acerca de ello? ¿Cuáles son las opciones?

1. *Crear documentación consume muchísimo tiempo.* Si el sistema funciona vivirá con lo que se tenga En algunos casos éste es el enfoque correcto. No es posible recrear documentación para cientos de programas de computadora. Si un programa es relativamente estático está llegando al final de su vida útil, por lo que es improbable que experimente un cambio significativo, ¡déjelo ser!
2. *La documentación debe actualizarse, pero se tienen recursos limitados.* Se utilizará un enfoque de "documentar cuando se toque" Tal vez sea innecesario volver a documentar por completo la aplicación. En cambio, se documentan completamente aquellas porciones del sistema que en la actualidad experimentan cambios. Con el tiempo evolucionará una colección de documentación útil y relevante.
3. *El sistema es crucial para el negocio y debe volver a documentarse por completo.* Incluso en este caso un enfoque inteligente es recortar la documentación a un mínimo esencial.

Cada una de estas opciones es viable. Una organización de software debe elegir la más apropiada para cada caso

Ingeniería inversa. El término ingeniería inversa tiene sus orígenes en el mundo del hardware Una compañía desensambla un producto de hardware de un competidor con la finalidad de comprender sus "secretos" de diseño y fabricación. Tales secretos podrían comprenderse fácilmente si se obtuviesen las especificaciones de diseño y fabricación del competidor. Pero dichos documentos están patentados y no están disponibles para la compañía que realiza la ingeniería inversa. En esencia, la ingeniería inversa exitosa obtiene una o más especificaciones de diseño y fabricación para un producto cuando se examinan especímenes reales del producto.

La ingeniería inversa para el software es bastante similar Sin embargo, en ambos casos el programa objeto de la ingeniería inversa no es el de un competidor, sino el trabajo de la propia compañía (con frecuencia elaborado muchos años atrás) Los "secretos" que se comprendan serán oscuros, pues nunca se desarrolló una especificación Por lo tanto, la ingeniería inversa del software es el proceso de analizar un programa con la finalidad de crear una representación del programa en un mayor grado de abstracción que el código fuente. La ingeniería inversa es un proceso de recuperación de diseño Las herramientas de la ingeniería inversa obtienen infor-

Referencia Web

Un asesoramiento de recursos para la comunidad de la reingeniería se puede obtener en **www.comp.leeds.ac.uk/projects/RenaissanceWeb/**.

mación del diseño de datos, arquitectónico y de procedimientos a partir de un programa existente.

Reestructuración de código. El tipo más común de reingeniería (en realidad, en este caso el empleo del término reingeniería es cuestionable) es la reestructuración de código.[3] Algunos sistemas heredados tienen una arquitectura de programa relativamente sólida, pero los módulos individuales se codificaron en una forma que dificulta comprenderlos, probarlos y mantenerlos. En tales casos se puede reestructurar el código dentro de los módulos sospechosos.

Llevar a cabo esta actividad requiere analizar el código fuente empleando una herramienta de reestructuración. Se indican las violaciones de las estructuras de programación estructurada y entonces el código se reestructura (esto se puede hacer automáticamente). El código reestructurado resultante se revisa y prueba para garantizar que no se han introducido anormalidades. La documentación del código interno se actualiza.

Reestructuración de datos. Un programa con una arquitectura de datos débil será difícil de adaptar y mejorar. De hecho, en muchas aplicaciones la arquitectura de datos está más relacionada con la viabilidad a largo plazo de un programa que el código fuente.

A diferencia de la reestructuración de código, que ocurre en un grado relativamente bajo de abstracción, la reestructuración de datos es una actividad de reingeniería a gran escala. En la mayoría de los casos, la reestructuración de datos comienza con una actividad de ingeniería inversa. La arquitectura de datos actual se analiza con minuciosidad y se definen los modelos de datos necesarios (capítulo 9). Se identifican los objetos de datos y los atributos, y después se revisa la calidad de las estructuras de datos existentes.

Cuando la estructura de datos es débil (por ejemplo, actualmente se implementan archivos planos, cuando un enfoque relacional simplificaría enormemente el procesamiento), los datos se someten a reingeniería.

Puesto que la arquitectura de datos tiene una fuerte influencia sobre la arquitectura del programa y los algoritmos que lo pueblan, los cambios a los datos invariablemente resultarán en cambios arquitectónicos o al nivel de código.

Ingeniería directa. En un mundo ideal, las aplicaciones se reconstruirían empleando un "motor de reingeniería" automatizado. El programa antiguo sería insertado en el motor, analizado, reestructurado y luego regenerado en una forma que exhibiese los mejores aspectos de la calidad del software. A corto plazo es improbable que tal "motor" aparezca, pero las empresas han introducido herramientas que mejoran un limitado subconjunto de dichas capacidades que aborden dominios de aplicación específicos (por ejemplo, las aplicaciones que se implementan mediante

3 La reestructuración de código tiene algunos de los elementos de "refactorización", un concepto de rediseño introducido en el capítulo 4 y tratado en otras partes de este libro.

un sistema de base de datos específico). Más importante, dichas herramientas de reingeniería se están volviendo cada vez más sofisticadas

La ingeniería directa, también llamada renovación o reclamación [CHI90], no sólo recupera la información de diseño a partir del software existente, también utiliza esta información para alterar o reconstituir el sistema existente con la finalidad de mejorar su calidad global. En la mayoría de los casos el software sometido a reingeniería vuelve a implementar la función del sistema existente y también añade nuevas funciones o mejora el desempeño global.

31.3 INGENIERÍA INVERSA

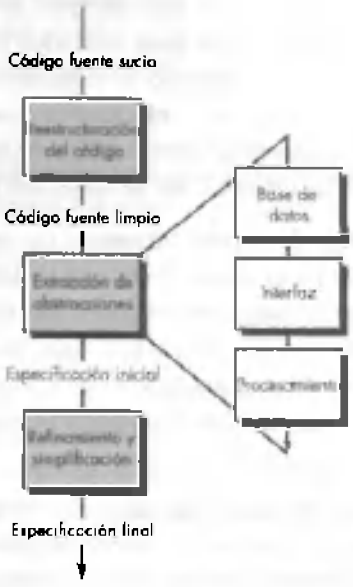
La ingeniería inversa invoca una imagen de "ranura mágica". En la ranura se inserta una lista fuente sin documentar y diseñado casualmente, y del otro extremo sale una descripción (y toda la documentación) completa del diseño para el programa de computadora. Desdichadamente, la ranura mágica no existe. La ingeniería inversa puede obtener información de diseño a partir del código fuente, pero el grado de abstracción, la completud de la documentación, el grado en el que las herramientas y un analista humano trabajan en conjunto, y la direccionalidad del proceso son enormemente variables.

El grado de abstracción de un proceso de ingeniería inversa y las herramientas utilizadas para efectuarlo se refieren a la sofisticación de la información del diseño que es posible obtener del código fuente. Idealmente, el grado de abstracción debe ser tan alto como sea posible. Esto es, el proceso de ingeniería inversa debe ser capaz de derivar representaciones de diseño de procedimiento (un grado de abstracción bajo), información de estructura de programa y datos (un grado de abstracción un poco más elevado), modelos de objeto, modelos de flujo de datos o control (un grado de abstracción relativamente alto) y clases UML, diagramas de estado y despliegue (un grado alto de abstracción). Conforme el grado de abstracción aumenta, el ingeniero de software obtiene información que le permitirá comprender con más facilidad el programa.

La completud de un proceso de ingeniería inversa se refiere al grado de detalle que se ofrece en un grado de abstracción. En la mayoría de los casos, la integridad disminuye conforme el grado de abstracción aumenta. Por ejemplo, dada una lista del código fuente, es relativamente sencillo desarrollar una representación completa del diseño del procedimiento. También se pueden derivar representaciones sencillas de diseño, pero es mucho más difícil desarrollar un conjunto completo de diagramas o modelos UML.

La completud mejora en proporción directa con la cantidad de análisis que efectúa quien realiza la ingeniería inversa. La *interactividad* se refiere al grado en el que el ser humano está "integrado" con las herramientas automatizadas para crear un proceso de ingeniería inversa efectivo. En la mayoría de los casos, conforme aumenta el grado de abstracción la interactividad debe aumentar o la completud sufrirá.

FIGURA 31.3
Proceso de ingeniería inversa.



PUNTO CLAVE
Se deben abordar tres temas de la ingeniería inversa: grado de abstracción, integridad y direccionalidad

Si la *direccionalidad* del proceso de ingeniería inversa es unidireccional, toda la información extraída del código fuente se ofrece al ingeniero de software que entonces puede usarla durante cualquier actividad de mantenimiento. Si la direccionalidad es bidireccional, la información alimenta a una herramienta de reingeniería que intenta reestructurar o regenerar el programa antiguo.

En la figura 31.3 se representa el proceso de ingeniería inversa. Antes de que comiencen las actividades de ingeniería inversa, el código fuente no estructurado ("sucio") se reestructura (sección 31.4.1) de modo que sólo contenga las estructuras de programación estructurada.[4] Esto facilita la lectura del código fuente y ofrece la base para las subsecuentes actividades de ingeniería inversa.

El núcleo de la ingeniería inversa es una actividad llamada *extracción de abstracciones*. El ingeniero debe evaluar el programa antiguo y, a partir del código fuente (con frecuencia sin documentar), desarrollar una especificación significativa del procesamiento que realiza, la interfaz del usuario que se aplica y las estructuras de datos del programa o las bases de datos que se utilizan

31.3.1 Ingeniería inversa para comprender los datos

La ingeniería inversa de datos ocurre en diferentes grados de abstracción y con frecuencia es la primera tarea de reingeniería. Al nivel del programa, las estructuras de datos internos del programa usualmente deben someterse a reingeniería inversa

4 El código se reestructura empleando un motor de reestructuración, una herramienta que reestructura código fuente

como parte de un esfuerzo global de reingeniería. En el nivel del sistema las estructuras globales de datos (por ejemplo, archivos, bases de datos) con frecuencia se someten a reingeniería para ajustarlos con los nuevos paradigmas de gestión de bases de datos (por ejemplo, el movimiento desde los archivos planos hacia los sistemas de bases de datos relacionales u orientados a objetos). La ingeniería inversa de las actuales estructuras globales de datos establece el escenario para la introducción de una nueva base de datos que abarque todo el sistema.

Referencia Web

Recursos útiles para la "recuperación de diseño y comprensión de programa" se pueden encontrar en www.sel.iit.nrc.ca/projects/dr/dr.html.

Estructuras de datos internos. Las técnicas de ingeniería inversa para datos internos del programa se enfoca en la definición de clases de objetos. Esto se logra al examinar el código del programa con el propósito de agrupar las variables de programa relacionadas. En muchos casos, la organización de los datos dentro del código identifica tipos abstractos de datos. Por ejemplo, estructuras de registro, archivos, listas y otras estructuras de datos con frecuencia ofrecen una indicación inicial de las clases.

CONSEJO

Los compromisos aparentemente insignificantes en las estructuras de datos pueden conducir a problemas potencialmente catastróficos en años futuros. Considere como ejemplo el problema Y2K.

Estructura de bases de datos. Sin importar su organización lógica y estructura física, una base de datos permite la definición de objetos de datos y apoya algún método para establecer relaciones entre los objetos. En consecuencia, la reingeniería de un esquema de base de datos en otro requiere comprender los objetos existentes y sus relaciones.

Los siguientes pasos [PRE94] se pueden utilizar para definir el modelo de datos existente como un precursor para la reingeniería de un nuevo modelo de base de datos: 1) construcción de un modelo inicial de objeto, 2) determinación de los candidatos clave, 3) refinar las clases tentativas, 4) definición de generalizaciones y 5) descubrimiento de asociaciones (empleo de técnicas análogas al enfoque CRC). Una vez que se conoce la información definida en los pasos precedentes, se aplica una serie de transformaciones [PRE94] para correlacionar la estructura antigua de la base de datos con una nueva estructura de base de datos

31.3.2 Ingeniería inversa para comprender el procesamiento

La ingeniería inversa para comprender el procesamiento comienza con un intento por comprender y luego extraer abstracciones de procedimientos representadas por el código fuente. Para comprender las abstracciones de procedimientos el código se analiza en grados variables de abstracción: sistema, programa, componentes, patrón y planteamiento.

La funcionalidad global de todo el sistema de aplicación se debe comprender antes de que ocurra un trabajo de ingeniería inversa más detallado. Esto establece un contexto para un mayor análisis y ofrece poca visión de los conflictos de interoperabilidad entre las aplicaciones dentro del sistema. Cada uno de los programas que conforman el sistema de la aplicación representa una abstracción funcional en un mayor grado de detalle. Se crea un diagrama de bloques que representa la interacción entre dichas abstracciones funcionales. Cada componente realiza alguna

subfunción y representa una abstracción de procedimiento definida. Para cada componente se desarrolla una narrativa de procesamiento. En algunas situaciones ya existen especificaciones del sistema, el programa y los componentes Cuando éste es el caso, las especificaciones se revisan para verificar si concuerdan con el código existente.[5]

> "Existe una pasión por la comprensión, así como existe una por la música. Dicha pasión es más común en los niños, pero más tarde se pierde en la mayoría de las personas."
>
> **Albert Einstein**

Las cosas se complican más cuando se considera el código dentro de un componente. El ingeniero busca las secciones de código que representen patrones de procedimiento genéricos. Casi en cada componente una sección del código prepara los datos para el procesamiento (dentro del módulo), una sección diferente de código realiza el procesamiento y otra sección del código prepara los resultados del procesamiento para exportarlos desde el componente Dentro de cada una de estas secciones se pueden encontrar pequeños patrones; por ejemplo, la validación de los datos y la verificación de enlaces con frecuencia ocurre dentro de la sección de código que prepara los datos para el procesamiento.

En sistemas grandes la ingeniería inversa, por lo general, se logra utilizando un enfoque semiautomatizado. Las herramientas automatizadas se utilizan para ayudar al ingeniero de software a comprender la semántica del código existente. Entonces la salida de este proceso pasa a la reestructuración y a las herramientas de ingeniería avanzada para completar el proceso de reingeniería.

31.3.3 Ingeniería inversa de interfaces de usuario

Las IGU sofisticadas ahora son indispensables en productos basados en computadora y en sistemas de todo tipo. En consecuencia, desarrollar de nuevo las interfaces de usuario se ha vuelto uno de los tipos más comunes de actividad de reingeniería. Pero antes de que una interfaz de usuario se pueda reconstruir deberá realizarse una actividad de ingeniería inversa.

Entender por completo una interfaz de usuario existente requiere especificar la estructura y el comportamiento de la interfaz. Merlo y sus colegas [MER93] sugieren tres preguntas básicas que se deben responder conforme comienza la ingeniería inversa de la IGU:

- ¿Cuáles son las acciones básicas (por ejemplo, presiones de tecla o clics de ratón) que debe procesar la interfaz?

5 Con frecuencia, las especificaciones escritas en las primeras etapas en la historia de vida de un programa nunca se actualizan. Conforme los cambios se realizan, el código ya no concuerda más con las especificaciones.

¿Cómo entender las funciones de una interfaz de usuario existente?

- ¿Cuál es la descripción compacta de las respuestas de comportamiento del sistema a estas acciones?
- ¿Qué se entiende por "reemplazo" o, más exactamente, qué concepto de equivalencia de interfaces es relevante en este caso?

La notación de modelado de comportamiento (capítulo 8) puede ofrecer un medio para desarrollar respuestas a las primeras dos preguntas. Gran parte de la información necesaria para crear un modelo de comportamiento se puede obtener observando la manifestación externa de la interfaz existente. Pero la información adicional necesaria para crear el modelo de comportamiento se debe extraer del código.

Es importante señalar que una GUI de reemplazo tal vez no refleje exactamente la interfaz antigua (de hecho, quizá sea radicalmente diferente). Con frecuencia vale la pena desarrollar nuevas metáforas de interacción. Por ejemplo, una GUI antigua solicita que un usuario proporcione un factor de escala (que varía desde 1 hasta 10) para encoger o ampliar una imagen gráfica. Una GUI sometida a reingeniería puede utilizar una barra de deslizamiento y un ratón para realizar la misma función.

HERRAMIENTAS DE SOFTWARE

Ingeniería inversa

Objetivo: Ayudar a los ingenieros de software a comprender la estructura de diseño interna de los programas complejos.

Mecánica: En la mayoría de los casos, las herramientas de ingeniería inversa aceptan código fuente como entrada y producen una diversidad de representaciones de diseño estructural, procedimiento, datos y comportamiento.

Herramientas representativas[6]

Imagix 4D, desarrollado por Imagix (www.imagix.com), "ayuda a los desarrolladores de software a comprender software C y C++ complejo o heredado" al someter a ingeniería inversa y documentar el código fuente.

Understand, desarrollado por Scientific Toolworks, Inc. (www.scitools.com), analiza gramaticalmente Ada, Fortran, C, C++ y Java "para realizar ingeniería inversa, documentar automáticamente, calcular métricas de código y auxiliarlo a comprender, navegar y mantener el código fuente".

Una extensa lista de herramientas de ingeniería inversa se puede encontrar en http://scgwiki.iam.unibe.ch:8080/SCG/370.

31.4 REESTRUCTURACIÓN

La reestructuración de software modifica el código fuente o los datos con la finalidad de adecuarlos para futuros cambios. En general, la reestructuración no modifica la arquitectura global del programa. Tiende a enfocarse sobre los detalles de diseño de los módulos individuales y en las estructuras de datos locales definidos dentro de los módulos. Si el trabajo de reestructuración se extiende más allá de las fronteras

6 Las herramientas expuestas el autor no las respalda, sólo representan una muestra de las herramientas incluidas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

del módulo y abarca la arquitectura del software, la reestructuración se convierte en ingeniería avanzada (sección 31.5).

La reestructuración ocurre cuando la arquitectura básica de una aplicación es sólida, aun cuando el interior técnico necesite trabajarse. Se inicia cuando grandes partes del software son funcionales y sólo un subconjunto de los componentes y datos necesitan una modificación extensa.[7]

31.4.1 Reestructuración del código

CONSEJO

Aunque la reestructuración del código puede aliviar inmediatamente los problemas asociados con la depuración o los cambios pequeños, esto no es reingeniería. El beneficio real se logra sólo cuando se reestructuran los datos y la arquitectura

La reestructuración del código se realiza para generar un diseño que produzca la misma función que el programa original, pero con mayor calidad. En general, las técnicas de reestructuración de código (por ejemplo, técnicas de simplificación lógica de Warnier [WAR74]) modelan la lógica del programa utilizando álgebra booleana y luego aplican una serie de reglas de transformación que producen lógica reestructurada. El objetivo es tomar el "tazón de espagueti" de código y derivar un diseño de procedimiento que concuerde con la filosofía de la programación estructurada (capítulo 11).

También se han propuesto otras técnicas de reestructuración para utilizarlas con las herramientas de reingeniería. Un diagrama de intercambio de recursos correlaciona cada módulo de programa y los recursos (tipos de datos, procedimientos y variables) que se intercambian entre ellos y otros módulos. Mediante la creación de representaciones del flujo de recursos se puede reestructurar la arquitectura del programa para lograr mínimos acoplamientos entre módulos.

31.4.2 Reestructuración de los datos

Antes de comenzar la reestructuración de datos se debe llevar a cabo una actividad de ingeniería inversa denominada *análisis del código fuente*. Primero se evalúan todos los enunciados del lenguaje de programación que contengan definiciones de datos, descripciones de archivos, I/O y descripciones de interfaz. La finalidad es extraer elementos y objetos de datos para obtener información acerca del flujo de datos y comprender las estructuras de datos existentes que se han implementado. Esta actividad a veces se denomina *análisis de datos* [RIC89].

Una vez completado el análisis de datos comienza el *rediseño de datos*. En su forma más simple, un paso de *estandarización de registro* de datos clarifica las definiciones de datos para lograr consistencia entre los nombres de elementos de datos o formatos de registro físicos dentro de una estructura de datos existente o formato de archivo. Otra forma de rediseño, denominada *racionalización del nombre* de los datos asegura que todas las convenciones de nombramiento de los datos concuerdan con el estándar local y que los pseudónimos se eliminan como flujo de datos a través del sistema.

7 A veces es difícil distinguir entre reestructuración extensa y volver a desarrollar. Ambos son reingeniería.

Cuando la reestructuración rebasa la estandarización y la nacionalización se realizan modificaciones físicas a las estructuras de datos existentes para lograr que el diseño de los datos sea más efectivo. Esto puede significar una traducción de un formato de archivo a otro o, en algunos casos, la traducción desde un tipo de base de datos a otro.

HERRAMIENTAS DE SOFTWARE

Reestructuración de software

Objetivo: El objetivo de las herramientas de reestructuración es transformar el antiguo software de computadora carente de estructura en lenguajes de programación y estructuras de diseño modernos.

Mecánica: En general, el código fuente se ingresa y transforma en un mejor programa estructurado. En algunos casos, la transformación ocurre dentro del mismo lenguaje de programación. En otros casos, un lenguaje de programación antiguo se transforma en un lenguaje más moderno.

Herramientas representativas[8]

DMS Software Reengineering Toolkit, desarrollada por Semantic Design (www.semdesigns.com), proporciona una diversidad de capacidades de reestructuración para COBOL, C/C++, Java, FORTRAN 90 y VHDL.

FORESYS, desarrollada por Simulog (www.simulog.fr), analiza y transforma programas escritos en FORTRAN.

Function Encapsulation Tool, desarrollada en la Wayne State University (www.cs.wayne.edu/~vip/RefactoringTools/), refactoriza en C++ los antiguos programas en C.

plusFORT, desarrollada por Polyhedron (www.polyhedron.com), es un conjunto de herramientas de FORTRAN que tiene capacidades para reestructurar en FORTRAN moderno o C estándar los programas en FORTRAN deficientemente diseñados.

31.5 INGENIERÍA DIRECTA

Un programa con flujo de control —el equivalente gráfico de un tazón de espagueti, con "módulos" que tienen 2 000 enunciados de longitud, con pocas líneas significativas de comentarios en los 290 000 enunciados fuente y ninguna otra documentación, se debe modificar para que se ajuste a los cambiantes requerimientos de los usuarios. Se tienen las siguientes opciones:

¿Qué opciones existen cuando se enfrenta un programa deficientemente diseñado e implementado?

1. Se puede trabajar modificación tras modificación, luchando con el diseño implícito y el código fuente para implementar los cambios necesarios.
2. Se puede intentar comprender el extenso funcionamiento interno del programa con el propósito de realizar modificaciones de manera más eficiente.
3. Se puede rediseñar, recodificar y probar aquellas porciones del software que requieran modificación mediante la aplicación de un enfoque de ingeniería del software en todos los segmentos revisados.

8 Las herramientas expuestas sólo representan una muestra de esta categoría. En la mayoría de los casos los nombres de las mismas son marcas registradas por sus respectivos desarrolladores.

4. Se puede rediseñar, recodificar y probar el programa completamente empleando herramientas de reingeniería como auxiliares para comprender el diseño actual.

No existe una opción individual "correcta". Las circunstancias pueden dictar la primera opción, incluso si las otras son más deseables.

En lugar de esperar hasta que se reciba la solicitud de mantenimiento, la organización de desarrollo o soporte utiliza los resultados del análisis de inventarios para seleccionar un programa que 1) se empleará durante un número determinado de años, 2) actualmente se utilice con éxito, y 3) es probable que experimentará grandes modificaciones o mejoras en el futuro cercano. Entonces se aplican las opciones 2, 3 o 4.

Este enfoque de *mantenimiento preventivo* lo introdujo Miller [MIL81] con el nombre de *retrofit* estructurado*. Este concepto se define como "la aplicación de las metodologías de hoy a los sistemas del ayer para apoyar los requisitos del mañana".

A primera vista, la sugerencia de que se desarrolle de nuevo un gran programa cuando ya existe una versión operativa puede parecer bastante extravagante. Antes de emitir un juicio, considérense los puntos siguientes:

CONSEJO

La reingeniería es muy parecida a la limpieza dental. Puede pensar en miles de razones para demorarla, y la aplazará muchas veces. Pero eventualmente sus tácticas dilatorias regresarán para provocarle dolor

1. El costo de mantener una línea de código fuente tal vez oscile entre 20 y 40 veces el costo de su desarrollo inicial.
2. El rediseño de la arquitectura de software (estructura de programa o datos) empleando conceptos modernos de diseño puede facilitar enormemente el mantenimiento futuro.
3. Puesto que ya existe un prototipo del software, el desarrollo de la productividad debe ser mucho mayor que el promedio.
4. El usuario ahora tiene experiencia con el software. En consecuencia, los nuevos requisitos y la dirección del cambio pueden afirmarse con mayor facilidad
5. Las herramientas automatizadas para reingeniería facilitarán algunas partes del trabajo.
6. Antes de terminar el mantenimiento preventivo existirá una configuración completa del software (documentos, programas y datos).

Cuando una organización de desarrollo de software vende software como producto, el mantenimiento preventivo se considera como "nuevas liberaciones" de un programa. Un gran desarrollador de software local (por ejemplo, un grupo de desarrollo de software para sistemas de negocios destinados a una gran compañía consumidora de productos) puede tener 500-2 000 programas de producción dentro de

* El término *retrofit* (literalmente retroajuste) recibe muchas denominaciones en español, entre las que destacan: remodelado, modernización, retrocambio, reajuste, modificación retroactiva. Para no generar confusión con otros conceptos y denominaciones similares, pero no relacionados, utilizaré el término original, que por lo demás es reconocido dentro del medio. (N.T.)

su dominio de responsabilidad. Dichos programas pueden clasificarse según su importancia y luego revisarlos como candidatos para mantenimiento preventivo.

El proceso de ingeniería avanzada aplica los principios, conceptos y métodos de la ingeniería del software para recrear una aplicación existente. En la mayoría de los casos, la ingeniería directa no simplemente crea el equivalente moderno de un programa antiguo. Más bien, los nuevos requisitos de usuario y tecnología se integran en el trabajo de reingeniería. El programa que se desarrolla de nuevo amplía las capacidades de la aplicación anterior.

31.5.1 Ingeniería directa para arquitecturas cliente/servidor

Durante las décadas pasadas muchas aplicaciones para computadora central se han sometido a reingeniería para adaptarlas a arquitecturas cliente/servidor (incluso WebApps). En esencia, los recursos de cómputo centralizados (que incluyen software) se distribuyen entre muchas plataformas cliente. Aunque se pueden diseñar varios entornos distribuidos diferentes, la aplicación típica de computadora central que se somete a reingeniería en una arquitectura cliente/servidor tiene las siguientes características:

- La funcionalidad de la aplicación migra hacia cada computadora cliente.
- En los sitios cliente se implementan nuevas interfases IGU.
- Las funciones de base de datos se asignan al servidor.
- La funcionalidad especializada (por ejemplo, análisis intenso de cómputo) puede permanecer en el sitio servidor.
- Tanto en los sitios cliente como servidor se deben establecer nuevos requisitos de comunicaciones, seguridad, archivado y control.

Es importante señalar que la migración desde la computadora central hacia el cómputo cliente/servidor requiere reingeniería tanto de negocio como de software. Además, se debe establecer una "infraestructura de red de empresa" [JAY94].

CONSEJO

En algunos casos, la migración hacia la arquitectura cliente-servidor no debe enfocarse como reingeniería, sino como un nuevo esfuerzo de desarrollo. La reingeniería ingresa al cuadro sólo cuando la funcionalidad específica del sistema antiguo se integrará en la nueva arquitectura.

La reingeniería para aplicaciones cliente/servidor comienza con un amplio análisis del entorno de negocios que incluye la computadora central existente. Se pueden identificar tres capas de abstracción. La capa de base de datos pone los cimientos de una arquitectura cliente/servidor y gestiona las transacciones y consultas desde aplicaciones cliente. Aunque dichas transacciones y consultas se deben controlar dentro del contexto de un conjunto de reglas de negocios (definidas mediante un proceso de negocio existente o sometido a reingeniería). Las aplicaciones cliente ofrecen la funcionalidad deseada para la comunidad de usuarios.

Las funciones del sistema de gestión de bases de datos existente y la arquitectura de datos de la base de datos existente deben someterse a ingeniería inversa como precursores del rediseño de la capa de base de datos. En algunos casos se crea un nuevo modelo de datos (capítulo 8). En cada caso la base de datos cliente/servidor se somete a reingeniería para garantizar que las transacciones se ejecutan en forma

consistente, que todas las actualizaciones las realizan sólo usuarios autorizados, que las reglas centrales del negocio se refuerzan (por ejemplo, antes de que se borre el registro de una empresa el servidor se asegura de que no haya cuentas por pagar, contratos o comunicaciones relacionados con dicha empresa), que las consultas se pueden ajustar eficientemente y que se ha establecido una capacidad completa de archivado.

La *capa de reglas de negocios* representa el software que reside tanto en el cliente como en el servidor. Este software realiza tareas de control y coordinación para garantizar que las transacciones y consultas entre la aplicación cliente y la base de datos se ajustan al proceso de negocios establecido.

La *capa de aplicaciones cliente* implementa funciones de negocios que requieren grupos específicos de usuarios finales. En muchas instancias, una aplicación de computadora central se segmenta en varias aplicaciones de escritorio más pequeñas y sometidas a reingeniería. La comunicación entre las aplicaciones de escritorio (cuando es necesario) se controla mediante la capa de reglas de negocios.

Un estudio completo del diseño y la reingeniería del software cliente/servidor es materia de libros especializados. El lector interesado debe consultar [VAN02], [COU00] y [ORF99].

31.5.2 Ingeniería directa para arquitecturas orientadas a objetos

La ingeniería del software orientado a objetos se ha convertido en la alternativa en cuanto al paradigma de desarrollo para muchas organizaciones de software. Pero, ¿qué hay acerca de las aplicaciones existentes que se desarrollaron empleando métodos convencionales? En algunos casos la respuesta es dejar tales aplicaciones "como están". En otros, las aplicaciones viejas deben someterse a reingeniería de modo que se integren con facilidad en grandes sistemas orientados a objetos.

La reingeniería del software convencional en una implementación orientada a objetos utiliza muchas de las mismas técnicas estudiadas en la parte 2 de este libro. Primero, el software existente se somete a ingeniería inversa de modo que sea posible crear modelos de datos, funcionales y de comportamiento apropiados. Si el sistema de reingeniería amplía la funcionalidad o el comportamiento de la aplicación original, se crean casos de uso (capítulos 7 y 8). Luego los modelos de datos creados durante la ingeniería inversa se utilizan junto con el modelado CRC (capítulo 8) para establecer la base con que se definirán las clases. Enseguida se definen las jerarquías de clases, los modelos de relación de objetos, los modelos de comportamiento de objetos y los subsistemas y entonces comienza el diseño orientado a objetos.

Conforme la ingeniería directa orientada a objetos progresa desde el análisis hasta el diseño se puede invocar un modelo de proceso ISBC (capítulo 30). Si la aplicación antigua se encuentra en un dominio que ya ocupan muchas aplicaciones orientadas a objetos, es probable que haya una buena biblioteca de componentes y que se pueda utilizar durante la ingeniería directa.

En aquellas clases que daban construirse desde el principio tal vez sea posible reutilizar algoritmos y estructuras de datos de la aplicación convencional existente. Sin embargo, quizá sea preciso diseñarlos de nuevo para ajustarlos a la arquitectura orientada a objetos.

31.5.3 Ingeniería directa de interfaces de usuario

Conforme las aplicaciones migran de la computadora central hacia el escritorio, los usuarios ya no desean tolerar las interfaces de usuario misteriosas basadas en caracteres. De hecho, una porción significativa del trabajo empleado en la transición de la computadora central a la computación cliente-servidor se puede dedicar a la reingeniería de las interfaces de usuario de la aplicación cliente.

Merlo y sus colegas [MER95] sugieren el siguiente modelo para la reingeniería de interfaces de usuario:

CONSEJO

¿Qué pasos se deben seguir para someter a reingeniería una interfaz de usuario?

1. *Comprender la interfaz original y los datos que se trasladan entre ella y el resto de la aplicación.* La finalidad es entender cómo otros elementos de un programa interactúan con el código existente que implementa la interfaz. Si se desarrollará una nueva GUI, los datos que fluyan entre ésta y el programa restante deben ser consistentes con los datos que actualmente fluyen entre la interfaz basada en caracteres y el programa.
2. *Remodelar el comportamiento implícito en la interfaz existente en una serie de abstracciones que tengan sentido en el contexto de una GUI.* Aunque el modo de interacción puede ser radicalmente diferente, el comportamiento de negocios que muestran los usuarios de las interfases vieja y nueva (cuando se le considera en términos de un escenario de utilización) debe permanecer igual. Una interfaz rediseñada deberá permitir que un usuario muestre el comportamiento de negocios apropiado. Por ejemplo, cuando se consulta una base de datos la vieja interfaz puede requerir una larga serie de comandos basados en texto para especificar la consulta. La GUI sometida a reingeniería puede dirigir la consulta a una pequeña secuencia de elecciones con el ratón, pero el propósito y el contenido de la consulta permanecen intactos.
3. *Introducir mejoras que hagan más eficiente el modo de interacción.* Las fallas ergonómicas de la interfaz existente se estudian y corrigen en el diseño de la nueva GUI.
4. *Construir e integrar la nueva GUI.* La existencia de bibliotecas de clases y herramientas automatizadas puede reducir significativamente el trabajo requerido para construir la GUI. Sin embargo, la integración con el software de la aplicación existente puede requerir más tiempo. Debe tenerse cuidado de garantizar que la GUI no propagará efectos colaterales adversos en el resto de la aplicación.

Referencia Web

Un manual de más de 300 páginas acerca de los patrones de [illegible] (desarrollado como parte del proyecto FAMOOS ESPRIT) se puede descargar desde www.iam.unibe.ch/~scg/Archive/Iumaas/patterns/index3.html.

> "Puede pagar poco dinero ahora, o puede pagar mucho dinero más adelante."
>
> **Anuncio de taller mecánico que sugiere un ajuste**

31.6 La economía de la reingeniería

En un mundo perfecto, cualquier programa al que no se le pudiera dar mantenimiento sería retirado inmediatamente, y sería sustituido por aplicaciones de mayor calidad con reingeniería desarrollada empleando modernas prácticas de ingeniería del software. Sin embargo, se vive en un mundo de recursos limitados. La reingeniería demanda recursos que pueden utilizarse para otros propósitos del negocio. En consecuencia, antes de que una organización intente someter a reingeniería una aplicación existente, debe realizar un análisis costo-beneficio.

Sneed [SNE95] ha propuesto un modelo de análisis costo-beneficio para la reingeniería. Se definen nueve parámetros:

P_1 = costo de mantenimiento anual actual para una aplicación
P_2 = costo de operación anual actual para una aplicación
P_3 = valor de negocios anual actual de una aplicación
P_4 = costo de mantenimiento anual predicho después de la reingeniería
P_5 = costo de operación anual predicho después de la reingeniería
P_6 = valor de negocios anual predicho después de la reingeniería
P_7 = costo estimado de la reingeniería
P_8 = fecha estimada de la reingeniería
P_9 = factor de riesgo de la reingeniería ($P_9 = 1.0$ es el valor nominal)
L = vida esperada del sistema

El costo asociado con el mantenimiento continuo de una aplicación candidata (es decir, si la reingeniería no se realiza) se puede definir como

$$C_{mant} = [P_3 - (P_1 + P_2)] \times L \tag{31-1}$$

Los costos asociados con la reingeniería se definen empleando la siguiente relación:

$$C_{reing} = [P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9)] \tag{31-2}$$

Con la utilización de los costos presentados en las ecuaciones (31-1) y (31-2) el beneficio global de la reingeniería se puede calcular como

$$\text{costo beneficio} = C_{reing} - C_{mant} \tag{31-3}$$

El análisis costo-beneficio presentado en las ecuaciones se puede realizar para todas las aplicaciones de alta prioridad identificadas durante el análisis de inventario (sección 31.2.2). Aquellas aplicaciones que muestren el mayor costo-beneficio podrán destinarse a la reingeniería, mientras el trabajo con otras se puede posponer hasta que haya recursos disponibles.

31.7 Resumen

La reingeniería se presenta en dos diferentes grados de abstracción. En el ámbito del negocio, la reingeniería se centra en el proceso de negocios con el propósito de efec-

tuar los cambios para mejorar la competitividad en alguna área del negocio. En el ámbito del software, la reingeniería examina los sistemas y aplicaciones de información con la finalidad de reestructurarlos o reconstruirlos de modo que muestren mayor calidad.

La reingeniería de procesos de negocio (RPN) define metas del negocio, identifica y evalúa los procesos de negocios existentes (en el contexto de metas definidas), especifica y diseña procesos revisados, y elabora prototipos, los refina y particulariza dentro de un negocio. La RPN tiene un objetivo que va más allá del software. Su resultado con frecuencia es la definición de las formas en las cuales las tecnologías de la información pueden apoyar mejor a los negocios.

La reingeniería de software comprende una serie de actividades que incluyen análisis de inventario, reestructuración de documentos, ingeniería inversa, reestructuración de programas y datos, e ingeniería directa. La finalidad de estas actividades es crear versiones de programas existentes que sean de mayor calidad y tengan mayor facilidad de mantenimiento (programas que serán viables ya muy avanzado el siglo xxi).

El análisis de inventarios permite que una organización evalúe cada aplicación sistemáticamente, con la finalidad de determinar cuáles son candidatas a la reingeniería. La reestructuración de documentos crea un marco de trabajo de documentación que es necesario para brindar apoyo a largo plazo a una aplicación. La ingeniería inversa es el proceso de analizar un programa con el propósito de obtener información de diseño de datos, arquitectónico y de procedimiento. Finalmente, la ingeniería directa reconstruye un programa empleando modernas prácticas de ingeniería del software y la información aprendida durante la ingeniería inversa.

El costo-beneficio de la reingeniería se determina cuantitativamente. El costo del *statu quo*, esto es, el costo asociado con el soporte y el mantenimiento actuales de una aplicación existente, se compara con los costos proyectados de la reingeniería y la reducción resultante en los costos de mantenimiento. En casi todos los casos en los que un programa tenga una vida larga y en la actualidad muestre escasa facilidad de mantenimiento, la reingeniería representa una estrategia de negocios efectiva en cuanto al costo.

Referencias

[CAN72] Canning, R., "The Maintenance 'Iceberg'", en *EDP Analyzer*, vol. 10, núm. 10, octubre de 1972.

[CAS88] "Case Tools for Reverse Engineering", en *CASE Outlook*, CASE Consulting Group, vol 2, núm. 2, 1988, pp. 1-15.

[CHI90] Chikofsky, E. J. y J. H. Cross, II, "Reverse Engineering and Design Recovery: A Taxonomy", en *IEEE Software*, enero de 1990, pp. 13-17.

[COU00] Coulouris, G., J. Dollimore y T. Kindberg, *Distributed Systems: Concepts and Design*, 3a. ed., Addison-Wesley, 2000.

[DAV90] Davenport, T. H. y J. E. Young, "The New Industrial Engineering. Information Technology and Business Process Redesign", en *Sloan management Review*, verano de 1990, pp. 11-27.

[DEM95] DeMarco, T., "Lean and Mean", en *IEEE Software*, noviembre de 1995, pp. 101-102.
[HAM90] Hammer, M., "Reengineer Work: Don't Automate, Obliterate", en *Harvard Business Review*, julio-agosto de 1990, pp. 104-112.
[HAN93] Manna, M., "Maintenance Burden Begging for a Remedy", en *Datamation*, abril de 1993, pp. 53-63.
[JAY94] Jaychandra, Y., *Re-engineering the Networked Enterprise*, McGraw-Hill, 1994.
[MER93] Merlo, E. *et al.*, "Reverse Engineering of user Interfaces", *Proc. Working Conference on Reverse Engineering*, IEEE, Baltimore, mayo de 1993, pp. 171-178.
[MER95] Merlo, E. *et al.*, "Reengineering User Interfaces", en *IEEE Software*, enero de 1995, pp. 64-73.
[MIL81] Miller, J., en *Techniques of Program and System Maintenance*, (G. Parikh, ed.) Winthrop Publishers, 1981.
[ORF99] Orfali, R., D. Harkey y J. Edwards, *Client/Server Survival Guide*, 3a. ed., Wiley, 1999.
[OSB90] Osborne, W. M. y E. J. Chikofsky, "Fitting Pieces to the Maintenance Puzzle", en *IEEE Software*, enero de 1990, pp. 10-11.
[PRE94] Premerlani, W. J. y M. R. Blaha, "An Approach for Reverse Engineering of Relational Databases", en *CACM*, vol. 37, núm. 5, mayo de 1994, pp. 42-49.
[RIC89] Ricketts, J. A., J. C. DelMonaco y M. W. Weeks, "Data Reengineering for Application Systems", *Proc. Conf. Software Maintenance-1989*, IEEE, 1989, pp. 174-179.
[SNE95] Sneed, H., "Planning the Reengineering of Legacy Systems", en *IEEE Software*, enero de 1995, pp. 24-25.
[STE93] Stewart, T. A., "Reengineering: The Hot New Managing Tool", en *Fortune*, 23 de agosto de 1993, pp. 41-48.
[SWA76] Swanson, E. B., "The Dimensions of Maintenance", *Proc. Second Intl. Conf. Software Engineering*, IEEE, octubre de 1976, pp. 492-497.
[VAN02] Van Steen, M. y A. Tanenbaum, Distributed Systems: *Principles and Paradigms*, Prentice-hall, 2002.
[WAR74] Warnier, J. D., *Logical Construction of Programs*, Van Nostrand-Reinhold, 1974.

PROBLEMAS Y PUNTOS A CONSIDERAR

31.1. Considerar cualquier empleo realizado en los últimos cinco años. Describir el proceso de negocio en el que se participó. Emplear el modelo de RPN descrito en la sección 31.1.3 para recomendar cambios al proceso con la finalidad de hacerlo más eficiente.

31.2. Investigar un poco acerca de la eficacia de la reingeniería de procesos del negocio. Presentar argumentos en favor y en contra de este enfoque.

31.3. El instructor seleccionará uno de los programas que todos en la clase han desarrollado durante este curso. Intercambie su programa en forma aleatoria con alguien más en la clase. No explique u ofrezca un "paseo" por el programa. Ahora, implemente una mejora (que haya especificado el instructor) en el programa que ha recibido.

a) Realice todas las tareas de ingeniería del software, incluso una breve prueba manual (mas no con el autor del programa).
b) Conserve un cuidadoso seguimiento de todos los errores encontrados durante las pruebas.
c) Describa sus experiencias en clase.

31.4. Explorar la lista de verificación del análisis de inventario presentado en el sitio Web SEP e intentar el desarrollo de un sistema cuantitativo de calificación de software que pudiese aplicarse a programas existentes con la finalidad de elegir programas candidatos para reingeniería. El sistema debe rebasar el análisis presentado en la sección 31.6.

31.5. Sugerir opciones a la documentación por escrito o electrónica convencional que pudiesen servir como base para la reestructuración de documentos. (Sugerencia: piénsese en las nuevas tecnologías descriptivas que se pudieran usar para comunicar el propósito del software.)

31.6. Algunas personas creen que la tecnología de inteligencia artificial aumentará el grado de abstracción del proceso de ingeniería inversa. Realizar algo de investigación acerca de esta materia (es decir, el uso de IA en la ingeniería inversa) y escribir un breve ensayo que contenga una opinión acerca de este punto.

31.7. ¿Por qué la completud es difícil de lograr conforme aumenta el grado de abstracción?

31.8. ¿Por qué debe aumentar la interactividad si la completud aumenta?

31.9. Con base en la información obtenida vía Internet, presente a su clase las características de tres herramientas de ingeniería inversa

31.10. Existe una sutil diferencia entre reestructuración e ingeniería directa ¿Cuál es?

31.11. Investigue la bibliografía o fuentes de Internet para encontrar uno o más escritos que borden estudios de caso de reingeniería de computadora central a cliente-servidor. Presente un resumen.

31.12. ¿Cómo determinaría de P_4 a P_7 en el modelo costo-beneficio presentado en la sección 31.6?

Otras lecturas y fuentes de información

Al igual que muchos temas apasionantes en la comunidad de negocios, las exageraciones que rodeaban la reingeniería de procesos de negocio han dado paso a una visión más pragmática de la materia. Hammer y Champy (*Reengineering the Corporation*, HarperBusiness, edición revisada, 2001) precipitó el interés temprano con el éxito de ventas de su libro. Más tarde, Hammer (*Beyond Reengineering. How the Processed-Centered Organization Is Changing Our Work and Our Lives*, HarperCollins, 1997) refinó su visión al enfocarse sobre los temas "centrados en el proceso"

Los libros de Smith y Fingar (*Business Process Management (BPM): The Third Wave*, Meghan-Kiffer Press, 2003), Jacka y Keller (*Business Process Mapping: Improving Customer Satisfaction*, Wiley, 2001), Sharp y McDermott (*Workflow Modeling*, Artech House, 2001), Andersen (*Business Process Improvement Toolbox*, American Society for Quality, 1999) y Harrington *et al.* (*Business Process Improvement Workbook*, McGraw-Hill, 1997) presentan estudios de caso y directrices detalladas para la RPN

Feldmann (*The Practical Guide to Business Process Reengineering Using IDEFO*, Dorset House, 1998) analiza una notación de modelado que auxilia en la RPN Berztiss (*Software Methods for Business Reengineering*, Springer, 1996) y Spurr *et al.* (*Software Assistance for Business Reengineering*, Wiley, 1994) examinan herramientas y técnicas que facilitan la RPN

Secord y sus colegas (*Modernizing Legacy Systems*, Addison-Wesley, 2003), Ulrich (*Legacy Systems Transformation Strategies*, Prentice-Hall, 2002), Valenti (*Successful Software Reengineering*, IRM Press, 2002) y Rada (*Reengineering Software: How to Reuse Programming to Build new, State-of-the-Art Software*, Fitzroy Dearborn Publishers, 1999) se enfocan en las estrategias y prácticas para la reingeniería en un contexto técnico. Miller (*Reengineering Legacy Software Systems*, Digital Press, 1998) "ofrece un marco de trabajo para mantener las aplicaciones de los sistemas sincronizadas con las estrategias del negocio y los cambios tecnológicos". Umar (*Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*, Prentice-Hall, 1997) ofrece lineamientos valiosos para las organizaciones que quieren transformar los sistemas heredados en un entorno basado en Web. Cook (*Building Enterprise Information Architectures: Reengineering Information Systems*, Prentice-Hall, 1996) analiza el puente entre la RPN y la tecnología de la información. Aiken (*Data Reverse Engineering*, McGraw-Hill, 1996) estudia cómo recuperar, reorganizar y reutilizar datos de la organización. Arnold (*Software Reengineering*, IEEE Computer Society Press, 1993) ha reunido una excelente antología de los primeros ensayos que trataban acerca de las tecnologías de la reingeniería del software.

Una amplia variedad de fuentes de información acerca de reingeniería de software está disponible en Internet Una lista actualizada de referencias en la World Wide Web se puede encontrar en el sitio Web SEPA:
http://www.mhhe.com/pressman.

CAPÍTULO

32

EL CAMINO POR RECORRER

CONCEPTOS CLAVE

En los 31 capítulos precedentes se exploró un proceso para la ingeniería del software. Se presentaron tanto procedimientos de gestión como métodos técnicos, principios básicos y técnicas especializadas, actividades orientadas a las personas y tareas adecuadas para automatizarlas, notación de papel y lápiz y herramientas de software. Se argumentó que la medición, la disciplina y una vigilancia estricta sobre la calidad generarán un software que satisfaga las necesidades de los clientes, que sea fiable, que tenga facilidad de mantenimiento, que sea mejor. Sin embargo, nunca se ha prometido que la ingeniería del software sea una panacea.

Conforme se continúe el viaje en el nuevo siglo, las tecnologías de software y sistemas seguirán siendo un desafío para los profesionales del software y las compañías que construyan sistemas basados en computadoras. Aunque escribió estas palabras con una visión del siglo XX, Max Hopper [HOP90] describió con precisión el estado actual del asunto:

> Puesto que los cambios en la tecnología de la información se están volviendo tan rápidos e implacables, y las consecuencias de caer ante ellos son irreversibles, las compañías dominarán la tecnología o morirán... Piense en ello como en un molino de tecnología. Las compañías tendrán que correr cada vez más rápido para permanecer en su lugar.

UN VISTAZO RÁPIDO

¿Qué es? El futuro nunca es fácil de predecir, no obstante eruditos, conferencistas televisivos y expertos industriales no se resisten. El camino por recorrer está plagado de restos de excitantes nuevas tecnologías que en realidad nunca lo fueron (a pesar de las exageraciones publicitarias), y con frecuencia lo conforman las tecnologías más modestas que de alguna forma modificaron la dirección y amplitud del camino principal. En consecuencia, no se intentará predecir el futuro sino que se estudiarán algunos de los conflictos que será necesario considerar para comprender cómo el software y la ingeniería del software cambiarán en los años por venir.

¿Quién lo hace? ¡Todo el mundo!

¿Por qué es importante? ¿Por qué los antiguos reyes contrataban adivinos? ¿Por qué las grandes corporaciones multinacionales contratan firmas consultoras y grupos de analistas para elaborar predicciones? ¿Por qué un porcentaje sustancial del público lee horóscopos? Porque quieren saber qué vendrá para estar preparados.

¿Cuáles son los pasos? No existe una fórmula para predecir el camino que se recorrerá. Se intenta hacerlo mediante la recopilación de datos —y su organización para proporcionar información útil— y el examen de asociaciones sutiles para obtener conocimiento y, a partir de éste, sugerir probables hechos que predigan cómo serán las cosas en cierto tiempo futuro.

¿Cuál es el producto obtenido? Una visión del futuro cercano que podría o no ser correcta.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Predecir el camino que se recorrerá es un arte, no una ciencia. De hecho, es bastante raro cuando una predicción

seria acerca del futuro es absolutamente correcta o inequívocamente errónea (con la excepción, por fortuna, de las predicciones del fin del mundo). Se buscan tendencias y se intenta extrapolarlas hacia adelante en el tiempo. Las correcciones de la extrapolación sólo se pueden valorar conforme pasa el tiempo

Los cambios en la tecnología de ingeniería de software son de hecho "rápidos e implacables", mientras que, al mismo tiempo, el progreso, por lo general, es bastante lento. Pero cuando se toma la decisión de adoptar un nuevo método (o una nueva herramienta), de llevar a cabo el entrenamiento necesario para comprender su aplicación y de introducir la tecnología en la cultura de desarrollo del software, algo más nuevo (e incluso mejor) ha llegado, y el proceso comienza de nuevo.

En este capítulo se examinan tendencias hacia el futuro. La finalidad no es explorar todas las áreas de investigación que resulten prometedoras. Tampoco es mirar en una "bola de cristal" y pronosticar el futuro. Más bien, se explora el ámbito del cambio y la forma en la que éste afectará el proceso de ingeniería del software en los años por venir.

32.1 La importancia del software. Segunda parte

La importancia del software de computadora se puede establecer en muchas formas. En el capítulo 1 el software se caracterizó como un diferenciador. La función que proporciona el software diferencia productos, sistemas y servicios, y ofrece una ventaja competitiva en el mercado. Pero el software es más que un diferenciador. Los programas, documentos y datos que constituyen el software ayudan a generar el producto más importante que cualquier individuo, negocio o gobierno pueda adquirir: *información*. Pressman y Herron [PRE91] describen al software en la forma siguiente:

> El software de computadora es una de sólo unas cuantas tecnologías clave que tendrán un impacto significativo en casi cualquier aspecto de la sociedad moderna... Es un mecanismo para automatizar negocios, industrias y gobiernos, y un medio para transferir nueva tecnología, un método de captura de experiencias valiosas para que las utilicen otros, un medio para diferenciar los productos de una compañía de los de sus competidores, y una ventana al conocimiento colectivo de una corporación. El software es un pivote para casi cualquier aspecto de los negocios. Pero, en muchas formas, el software también es una tecnología oculta. Se encuentra software (usualmente sin darse cuenta de ello) cuando se viaja al trabajo, se realiza alguna compra al menudeo, se detiene en el banco, se hace una llamada telefónica, se visita al médico o se realiza alguna de las cientos de actividades cotidianas que reflejan la vida moderna.

La omnipresencia del software conduce a una conclusión simple: siempre que una tecnología tenga un amplio impacto (un impacto que pueda salvar vidas o ponerlas en peligro, construir negocios o destruirlos, informar a los líderes de gobiernos o mal informarlos) debe manejársele con cuidado.

"Las predicciones son muy difíciles de hacer, especialmente cuando se relacionan con el futuro."

Mark Twain

32.2 EL ÁMBITO DEL CAMBIO

Los cambios en la informática durante los pasados 50 años han sido dirigidos por avances en las ciencias básicas: física, química, ciencia de materiales e ingeniería. Esta tendencia continuará durante el primer cuarto del siglo XXI. El impacto de las nuevas tecnologías es profundo: en las comunicaciones, la energía, el cuidado de la salud, la transportación, el entretenimiento, la economía, la industria manufacturera y la guerra, por mencionar sólo unas cuantas.

INFORMACIÓN

Tecnologías a observar. Las tecnologías para mirar.

Los editores de *PC Magazine* [PCM03] preparan un número anual de "Tecnologías del futuro" que "[busca] a través de todos los espacios de chat (hay varios de ellos) identificar las 20 tecnologías más prometedoras del mañana". Las tecnologías registradas abarcan toda una amplia gama, desde el cuidado de la salud hasta la guerra. Sin embargo, es interesante observar que el software y la ingeniería del software tienen un significativo papel que jugar en todas, ya sea como impulsores de la tecnología o como una parte integral de ellas. Las siguientes representan una muestra de las tecnologías registradas.

Nanotubos de carbono. Con una fina estructura parecida al grafito, los nanotubos de carbono pueden funcionar como alambres, para transmitir señales desde un punto a otro, y como transistores, usando cambios de señal para almacenar información. El uso de estos dispositivos parece prometedor en el desarrollo de dispositivos electrónicos más pequeños, más rápidos, de menor energía y menos costosos (por ejemplo, microprocesadores, memorias, pantallas).

Biosensores. Los sensores microelectrónicos, externos o implantables, ya se utilizan ampliamente en la detección, desde agentes químicos en el aire que se respira hasta niveles de sangre en pacientes cardiacos. Conforme estos sensores se vuelvan más sofisticados, podrán implantarse en pacientes médicos para supervisar una variedad de condiciones relacionadas con la salud, o incorporarlos a los uniformes de los soldados para supervisar la presencia de armas biológicas o químicas.

Pantallas OLED. Un OLED (diodo emisor de luz orgánico) "utiliza una molécula diseñadora con base de carbono que emite luz cuando una corriente eléctrica pasa a través de ella. Junte muchas de estas moléculas y obtendrá una pantalla muy delgada de asombrosa calidad; no requiere fuente de luz trasera que provoca pérdidas de energía" [PCM03]. El resultado: pantallas ultradelgadas que se pueden enrollar o doblar, extender sobre una superficie curva o adaptarse de alguna otra forma a un entorno específico.

Retícula de cómputo. Esta tecnología (disponible en la actualidad) crea una red que aprovecha los miles de millones de ciclos de CPU no utilizados de cada máquina en la red y permite que se completen tareas de cómputo excesivamente complejas sin contar con una supercomputadora. Véase un ejemplo práctico que abarca 4.5 millones de computadoras en http://setiathome.berkeley.edu/.

Máquinas cognitivas. El "santo grial" en el campo de la robótica es desarrollar máquinas que estén conscientes de su entorno, que puedan "recabar pistas, responder a situaciones siempre cambiantes e interactuar con personas de modo natural" [PCM03]. Las máquinas cognitivas todavía están en las primeras etapas de desarrollo, pero el potencial (si se logra alguna vez) es enorme.

Referencia Web
Para predicciones acerca del futuro de la tecnología y otras materias, véase **www.futurelaxing.com.**

A largo plazo, los avances revolucionarios en la informática bien pueden dirigirlos las ciencias sociales: psicología humana, sociología, filosofía, antropología y otras. El periodo de gestación de las tecnologías informáticas que se puedan derivar de estas disciplinas es muy difícil de predecir, pero las primeras influencias ya han comenzado (por ejemplo, las comunidades —una estructura antropológica— de usuarios, que son ramificaciones de las redes de pares a pares)

La influencia de las ciencias sociales quizá ayude a moldear la dirección de la investigación en informática en las ciencias básicas. Por ejemplo, el diseño de las futuras computadoras tal vez lo guíe más el conocimiento de la psicología cerebral que el de la microelectrónica convencional.

A corto plazo, los cambios que incidirán en la ingeniería del software durante la siguiente década recibirán la influencia de cuatro fuentes simultáneas: 1) las personas que realicen el trabajo, 2) el proceso que apliquen, 3) la naturaleza de la información y 4) la tecnología informática subyacente. En las secciones que siguen se examinan con más detalle cada uno de estos componentes: personas, proceso, información y tecnología.

32.3 Las personas y la forma en la que construyen sistemas

El software que requieren los sistemas de alta tecnología se vuelve más complejo cada año, y el tamaño de los programas resultantes aumenta proporcionalmente. El rápido crecimiento en el tamaño del programa "promedio" presentaría pocos problemas si no fuese por un hecho simple: conforme aumenta el tamaño del programa, también debe aumentar el número de personas que deben trabajar en él.

La experiencia indica que conforme aumenta el número de personas de un equipo de proyecto de software, tal vez la productividad global del grupo disminuya. Una forma para resolver este problema consiste en crear equipos de ingeniería del software, y en consecuencia dividir al personal en grupos de trabajo individuales. Sin embargo, conforme crece el número de equipos de trabajo de ingeniería del software, la comunicación entre ellos se vuelve tan difícil y tardada como la comunicación entre los individuos. Peor aún, la comunicación (entre individuos o equipos) tiende a ser ineficiente; es decir, se pasa demasiado tiempo transfiriendo muy poco contenido de información, y con mucha frecuencia la información importante "cae entre las grietas".

> "El choque del futuro es la agotadora tensión y desorientación que inducimos en los individuos al sujetarlos a demasiados cambios en un periodo demasiado corto."
>
> **Alvin Toffler**

Si la comunidad de la ingeniería del software tiene que tratar con eficacia el dilema de la comunicación, el camino que recorrerán los ingenieros de software debe incluir cambios radicales en la forma en que los individuos y los equipos se comunican entre sí. El correo electrónico, los sitios Web y las conferencias de video centra-

lizadas ahora son mecanismos comunes para conectar a un gran número de personas a una red de información. La importancia de estas herramientas en el contexto del trabajo de ingeniería del software no se debe sobrevaluar. Con un correo electrónico eficiente o un sistema de mensajería instantánea, el problema que encuentre un ingeniero de software en la ciudad de Nueva York puede resolverse con la ayuda de un colega en Tokio. En realidad, las sesiones de conversación [*chat*] sobre un tema particular y los grupos de noticias especializados se vuelven depósitos de conocimiento que permiten que el saber colectivo de un gran grupo de técnicos sea aplicado para solucionar un problema técnico o un conflicto de gestión

El video personaliza la comunicación En el mejor de los casos, permite que los colegas en diferentes ubicaciones (o en diferentes continentes) se "reúnan" con cierta regularidad Además, el video también ofrece otro beneficio: se puede usar como depósito de conocimiento acerca del software y para entrenar a los recién llegados a un proyecto.

> "La respuesta artística adecuada ante la tecnología digital es adoptarla como una nueva ventana a todo lo eternamente humano, y usarla con pasión, sabiduría, temeridad y alegría."
>
> **Ralph Lombreglia**

CONSEJO

Más y más "no programadores" están construyendo sus propios (pequeños) aplicaciones. Esta tendencia actual es probable que se acelere en lo futuro ¿Estos "civiles" deberían aplicar la tecnología estudiada en este libro? Probablemente no Pero deberían adoptar una filosofía de ingeniería del software ágil, incluso si no adoptan la práctica.

La evolución de los agentes inteligentes también cambiará los patrones laborales de un ingeniero de software al extender sustancialmente las capacidades de las herramientas de software Los agentes inteligentes mejorarán la habilidad del ingeniero al comprobar varias veces los productos de trabajo de ingeniería empleando conocimiento específico del dominio, realizando tareas administrativas, llevando a cabo una investigación dirigida y coordinando la comunicación entre las personas.

Finalmente, la adquisición de conocimiento está cambiando en forma radical. En Internet, un ingeniero de software puede suscribirse a grupos de noticias que se enfoquen en áreas de tecnología que le interesen directamente. Una pregunta enviada a un grupo de noticias precipita las respuestas de otras partes interesadas alrededor del mundo La World Wide Web ofrece a un ingeniero de software la más grande biblioteca del mundo de trabajos e informes de investigación, manuales, comentarios y referencias acerca de la ingeniería del software.

Si la historia es un indicio, es acertado decir que las personas no cambiarán. Sin embargo, las formas en las que se comunican, el entorno en el que trabajan, la manera en la que adquieren conocimiento, los métodos y herramientas que usan, la disciplina que aplican y, por lo tanto, la cultura general del desarrollo del software cambiarán en formas significativas e incluso profundas

32.4 EL "NUEVO" PROCESO DE INGENIERÍA DEL SOFTWARE

Es razonable caracterizar las primeras dos décadas de la práctica de la ingeniería del software como la era del "pensamiento lineal" Fomentada por el modelo clásico del ciclo vital, la ingeniería del software se enfocó como una actividad lineal en la que

se podrían aplicar una serie de pasos secuenciales con la finalidad de resolver problemas complejos. Sin embargo, los enfoques lineales acerca del desarrollo de software corren contra la forma en la que la mayoría de los sistemas realmente se construye. En realidad, los sistemas complejos evolucionan iterativamente, incluso en forma incremental. Por esta razón, un gran segmento de la comunidad de ingeniería del software se desplaza hacia modelos incrementales ágiles para el desarrollo del software.

Los modelos de proceso incremental ágil reconocen que la incertidumbre domina la mayoría de los proyectos, que los plazos de entrega con frecuencia son imposibles de cumplir y cortos, y que la iteración proporciona la habilidad de dar una solución parcial, incluso cuando un producto completo no es realizable dentro del tiempo asignado. Los modelos evolutivos subrayan la necesidad de productos de trabajo incrementales, análisis de riesgo, planeación y luego revisión del plan, y retroalimentación con el cliente. En muchos casos el equipo de software aplica un "manifiesto ágil" (capítulo 4) que subraya "los individuos e interacciones sobre los procesos y herramientas; el software operativo sobre la documentación detallada; la colaboración del cliente sobre la negociación de contratos, y la respuesta al cambio sobre el seguimiento de un plan" [BEC01].

> "La mejor preparación para el buen trabajo mañana es hacer un buen trabajo hoy."
>
> **Elbert Hubbard**

Las tecnologías de objetos, junto con la ingeniería del software basada en componentes (capítulo 30), son una consecuencia natural de la tendencia hacia los modelos de procesos incrementales y evolutivos. Ambos tendrán un profundo impacto sobre la productividad del desarrollo de software y la calidad del producto. La reutilización de componentes ofrece beneficios inmediatos y convincentes. Cuando la reutilización se une con las herramientas CASE para los prototipos de una aplicación, los incrementos del programa se pueden construir mucho más rápidamente que mediante los enfoques convencionales. La elaboración de prototipos involucra al cliente con el proceso. En consecuencia, es probable que los clientes y usuarios se involucren mucho más en el desarrollo del software. Esto, a su vez, puede conducir a una mayor satisfacción del usuario final y en general a mejorar la calidad del software.

El rápido crecimiento en las tecnologías de red y multimedia (por ejemplo, el aumento exponencial en las WebApps durante las pasadas décadas) está cambiando tanto al proceso de ingeniería del software como a sus participantes. De nuevo, se está ante un paradigma incremental ágil que destaca la inmediatez, la seguridad y la estética, así como preocupación por la ingeniería de software más convencional. Los modernos equipos de software (por ejemplo, un equipo de ingeniería Web) con frecuencia mezclan técnicos con especialistas de contenido (por ejemplo, artistas, músicos, videográfos) para construir una fuente de información destinada a una

comunidad de usuarios grande e impredecible. El software que se ha desarrollado con base en estas tecnologías ha generado radicales cambios económicos y culturales. Aunque los conceptos y principios básicos tratados en este libro son aplicables, el proceso de ingeniería del software se debe adaptar

32.5 NUEVOS MODOS DE REPRESENTAR LA INFORMACIÓN

A lo largo de la historia de la informática ha ocurrido una transición sutil en la terminología con que se describe el trabajo de desarrollo del software realizado para la comunidad de negocios. Hace cuarenta años, el término *procesamiento de datos* era la frase operativa para describir la utilización de las computadoras en un contexto de negocios. En la actualidad, el procesamiento de datos ha dado paso a otra frase —*tecnología de la información*— que significa lo mismo pero presenta un sutil cambio en el enfoque. La importancia radica no sólo en procesar grandes cantidades de datos, sino en obtener información significativa de dichos datos. Obviamente, ésta fue siempre la finalidad, pero el cambio en la terminología refleja un cambio mucho más importante en la filosofía de la gestión

Cuando en la actualidad se abordan las aplicaciones de software las palabras *datos* e *información* aparecen repetidamente. La palabra *conocimiento* se encuentra en algunas aplicaciones de inteligencia artificial, pero su utilización es relativamente escasa. Virtualmente nadie se refiere a la *sabiduría* en el contexto de las aplicaciones de software.

Los datos son información en bruto: colecciones de hechos que deben procesarse para que sean significativos La información se deriva al asociar los hechos en un

FIGURA 32.1

Espectro de "información".

contexto dado. El conocimiento asocia la información obtenida en un contexto con otra información obtenida en un contexto diferente. Finalmente, la sabiduría se presenta cuando los principios generalizados se derivan de conocimientos dispares. Cada una de estas cuatro visiones de la "información" se representa esquemáticamente en la figura 32.1.

A la fecha, la gran mayoría del software se ha construido para procesar datos o información. Los ingenieros de software ahora están igualmente preocupados con los sistemas que procesan el conocimiento.[1] El conocimiento es bidimensional. La información recopilada acerca de una diversidad de temas relacionados e inconexos se relaciona para formar un cuerpo de hechos que se llamará *conocimiento*. La clave es la habilidad personal para asociar la información proveniente de diversas fuentes, que tal vez no tengan alguna conexión evidente, y combinarla en una forma que ofrezca algún beneficio distinto.

> "La sabiduría es el poder que permite utilizar el conocimiento para el beneficio de nosotros mismos y de otros."
> **Thomas J. Watson**

Para ilustrar la progresión desde datos hasta conocimiento, considérense los datos censales que indican que los nacimientos en 1996 en Estados Unidos fueron 4.9 millones. Este número representa un valor de datos. Al relacionar esta pieza de datos con las tasas de nacimiento para los 40 años precedentes, se puede derivar una útil pieza de información: los cada vez más viejos *baby boomers* de la década de 1950 y de principios de la de 1960 hacían un último esfuerzo para tener hijos antes del final de su vida reproductiva. Además, los *gen Xers* (miembros de la generación X) comenzaban su vida reproductiva. Los datos censales entonces pueden vincularse con otras piezas de información aparentemente no relacionada. Por ejemplo, el número actual de profesores de escuela elemental que se retirarán durante la siguiente década, el número de estudiantes universitarios que se graduarán en educación primaria y secundaria, la presión sobre los políticos para mantener bajos los impuestos y, por lo tanto, limitar los aumentos salariales a los profesores.

Todos estos elementos de información se pueden combinar para formular una representación del conocimiento: existirá una presión significativa sobre el sistema educativo en Estados Unidos en la primera década del siglo XXI, y esta presión continuará durante una década. Con la utilización de este conocimiento puede surgir una oportunidad de negocio. Quizá haya significativas oportunidades para desarrollar nuevos modos de aprendizaje que sean más eficaces y menos costosos que los enfoques actuales.

1 El rápido crecimiento de las tecnologías de extracción y almacenamiento de datos refleja esta tendencia creciente.

El camino que recorrerá el software conduce a sistemas que procesan el conocimiento. Se han estado procesando datos empleando computadoras durante 50 años y extrayendo información durante más de tres décadas. Uno de los desafíos más significativos que enfrenta la comunidad de ingeniería del software es construir sistemas que den el próximo paso a lo largo del espectro· sistemas que extraigan conocimiento a partir de los datos e información en una forma práctica y beneficiosa

32.6 La tecnología como impulsor

La gente que construye y utiliza software, el proceso de ingeniería del software que se aplica y la información que se produce resultan afectados por los avances en el hardware y la tecnología del software. Históricamente, el hardware ha servido como el impulsor tecnológico en la computación. Una nueva tecnología de hardware proporciona potencial Entonces los constructores de software reaccionan a las demandas de los consumidores con la finalidad de aprovechar el potencial.

Es probable que las tendencias futuras de la tecnología de hardware avancen por dos trayectorias paralelas A lo largo de una trayectoria las tecnologías de hardware continuarán evolucionando con rapidez Ante la mayor capacidad que ofrecen las arquitecturas de hardware tradicionales, las demandas a los ingenieros de software continuarán creciendo

Pero los cambios verdaderos en la tecnología de hardware podrían producirse en otra dirección. El desarrollo de arquitecturas de hardware no tradicionales (por ejemplo, nanotubos de carbono, microprocesadores EUL, máquinas cognitivas, retículas de cómputo) podrían provocar cambios radicales en el tipo de software que se construirá y cambios fundamentales en el enfoque hacia la ingeniería del software. Dado que estos enfoques no tradicionales se están madurando, es difícil determinar cuál tendrá un impacto significativo e incluso es más difícil predecir cómo cambiará el mundo del software para adaptarse a ellos.

Las tendencias futuras de la ingeniería del software las impulsan las tecnologías de software. La reutilización y la ingeniería del software basada en componentes ofrecen la mejor oportunidad en cuanto a mejoras en la magnitud de la calidad de los sistemas y en el tiempo en que llegan al mercado. De hecho, conforme pasa el tiempo, el negocio del software puede comenzar a parecerse mucho al negocio de hardware de la actualidad. Quizá haya empresas que construyan dispositivos discretos (componentes de software reutilizables), otras empresas que construyan componentes de sistemas (por ejemplo, un conjunto de herramientas para la interacción hombre-máquina) e integradores de sistemas que ofrezcan soluciones (productos y sistemas construidos de forma personalizada) para el usuario final.

La ingeniería del software cambiará, de eso se puede estar seguro. Pero, sin importar cuán radicales sean los cambios, se puede estar seguro de que la calidad nunca perderá su importancia, y de que el análisis y el diseño efectivos y las pruebas competentes siempre tendrán un lugar en el desarrollo de los sistemas basados en computadoras.

INFORMACIÓN

Tendencias tecnológicas

P. Cripwell Associates (www.jpcripwell.com), una firma de consultoría especializada en gestión del conocimiento e ingeniería de la información, analiza cinco impulsores tecnológicos que influirán en las direcciones de la tecnología en los años por venir.

Combinación de tecnologías. Cuando dos importantes tecnologías se funden, el impacto del resultado con frecuencia es mayor que la suma del impacto de cada una por separado. Por ejemplo, la tecnología de los satélites GPS (sistemas de posicionamiento global) junto con la capacidad de cómputo a bordo y las tecnologías de pantallas LCD han permitido construir sofisticados sistemas de localización en los automóviles. Las tecnologías con frecuencia evolucionan a rutas separadas, pero el impacto en los negocios o social significativo sólo ocurre cuando alguien los combina para resolver un problema.

Fusión de datos. Mientras más datos se adquieran, más datos se necesitarán. Más importante aún, mientras más datos se adquieran, más difícil es extraer información útil. De hecho, con frecuencia se necesita adquirir todavía más datos para comprender qué datos son importantes; qué datos son relevantes para una necesidad o fuente particular, y qué datos se deben emplear para la toma de decisiones. Éste es el problema de la fusión de datos. J. P. Cripwell utiliza como ejemplo un sistema avanzado de supervisión de tráfico automovilístico. Sensores digitales de rapidez (en el camino) y cámaras digitales detectan un accidente. La severidad del accidente se debe determinar (¿a través de las cámaras?). Con base en la severidad, el sistema de supervisión debe contactar a la policía, los bomberos o ambulancias; el tráfico se debe redirigir; los medios de comunicación (radio) deben difundir advertencias; y debe informárseles a los automóviles individuales (si están equipados con sensores digitales o comunicación inalámbrica). Para lograrlo se debe tomar una variedad de decisiones, con base en los datos adquiridos a partir del sistema de supervisión (fusión de datos).

Tecnología de empuje. En años pasados surgió un problema y se desarrolló tecnología para resolverlo. Puesto que el problema era evidente para muchas personas, el mercado para la nueva tecnología estaba bien definido. En la actualidad, algunas tecnologías evolucionan como soluciones que buscan problemas. Un mercado debe empujarse para reconocer que necesita la nueva tecnología (por ejemplo, teléfonos móviles, PDA). Conforme las personas reconocen la necesidad, la tecnología se acelera, mejora y con frecuencia se transforma conforme evoluciona la combinación de tecnologías.

Red y casualidad. En este contexto, red implica conexiones entre personas o entre personas e información. Conforme crece la red, la probabilidad de sinergia entre dos nodos de red (por ejemplo, personas, fuentes de información) también crece. Una conexión fortuita (casual) puede conducir a una inspiración y a nueva tecnología o aplicación.

Sobrecarga de información. Un amplio océano de información está a disposición de cualquiera con una conexión de Internet. El problema, desde luego, es encontrar la información correcta, determinar su validez y luego traducirla en una aplicación práctica en un ámbito de negocios o personal.

32.7 LA RESPONSABILIDAD DE LA INGENIERÍA DEL SOFTWARE

Referencia Web

Un estudio completo del código de ética ACM/IEEE se puede encontrar en www.acm.org/serving/se/code.htm.

La ingeniería del software ha evolucionado en una profesión respetada en el ámbito mundial. Como profesionales, los ingenieros de software deben regirse por un código de ética que guíe el trabajo que realizan y los productos que producen. Una fuerza de trabajo conjunto ACM/IEEE-CS ha producido un *Código de ética y práctica profesional para los ingenieros de software* (versión 5.1). El código [ACM98] afirma:

> Los ingenieros de software se deben comprometer a sí mismos a convertir el análisis, la especificación, el diseño, el desarrollo, la prueba y el mantenimiento del software en una profesión beneficiosa y respetada. En concordancia con su compromiso con la salud, la

seguridad y el bienestar del público, los ingenieros de software deben adherirse a los siguientes Ocho Principios:

1. PÚBLICO. Los ingenieros de software deben actuar consistentemente con el interés del público.

2. CLIENTES Y EMPLEADORES. Los ingenieros de software deben actuar en una forma que beneficie los intereses de sus clientes y empleadores y sea consistente con el interés público.

3. PRODUCTO. Los ingenieros de software deben garantizar que sus productos y modificaciones relacionadas satisfacen los mayores estándares profesionales posibles

4 JUICIO. Los ingenieros de software deben mantener la integridad y la independencia en su juicio profesional.

5. GESTIÓN Los gestores y líderes de la ingeniería del software deben suscribir y promover un enfoque ético de la gestión del desarrollo y el mantenimiento del software.

6. PROFESIÓN. Los ingenieros de software deben promover la integridad y la buena reputación de la profesión en una forma consistente con el interés público

7 COLEGAS. Los ingenieros de software deben ser justos con sus colegas y apoyarlos

8 COMPROMISO PERSONAL. Los ingenieros de software deben participar en un aprendizaje permanente en relación con la práctica de su profesión y promover un enfoque ético para su práctica

Aunque cada uno de estos ocho principios es igualmente importante, aparece un tema más relevante: un ingeniero de software debe trabajar en pro del interés público En el ámbito personal, un ingeniero de software debe atenerse a las siguientes reglas:

- Nunca robar datos para beneficio personal.
- Nunca distribuir o vender información patentada que haya obtenido como parte de su trabajo en un proyecto de software
- Nunca destruir o modificar maliciosamente los programas, archivos o datos de otra persona.
- Nunca violar la privacidad de un individuo, grupo u organización.
- Nunca *atacar* un sistema por deporte o beneficio.
- Nunca crear o difundir un virus o gusano de computadora.
- Nunca usar la tecnología de computación para facilitar la discriminación o el hostigamiento.

Durante la década pasada, ciertos miembros de la industria del software han cabildeado por una legislación protectora que [SEE03]:

1. Permita a las compañías liberar el software sin revelar los defectos conocidos.
2. Exentar a los desarrolladores de responsabilidad penal por cualesquiera daños que resulten debido a dichos defectos conocidos.

3. Restringir a otros la revelación de defectos sin permiso del desarrollador original.
4. Permitir la incorporación de software de "autoayuda" dentro de un producto que pueda desactivar (vía comandos remotos) la operación del producto.
5. Exentar a los desarrolladores de software con "autoayuda" de los daños en caso de que el software lo desactive una tercera persona.

Al igual que con cualquier legislación, el debate con frecuencia se centra en conflictos políticos, no tecnológicos. Sin embargo, mucha gente (incluso este autor) considera que la legislación protectora, si se propone de manera inadecuada, entra en conflicto con el código de ética de la ingeniería del software al exentar indirectamente a los ingenieros de software de su responsabilidad para producir software de alta calidad.

32.8 UN COMENTARIO FINAL

Ya han pasado 25 años desde que se escribió la primera edición de este libro. Todavía me recuerdo sentado en mi escritorio como un joven profesor, escribiendo el manuscrito de un libro acerca de una materia de la que poca gente se preocupaba e incluso todavía menos realmente comprendía. Recuerdo las cartas de rechazo de los editores, quienes argumentaban (gentil, pero firmemente) que nunca habría un mercado para un libro acerca de "ingeniería del software". Afortunadamente, McGraw-Hill decidió darle una oportunidad,[2] y el resto, como dicen, es historia.

Durante los pasados 25 años, este libro ha cambiado sustancialmente: en alcance, en tamaño, en estilo y en contenido. Al igual que la ingeniería del software, ha crecido y (espero) madurado con los años.

Un enfoque de ingeniería para el desarrollo del software de computadora es ahora sabiduría convencional. Aunque el debate continúa acerca del "paradigma correcto", la importancia de la agilidad, el grado de automatización y los métodos más efectivos, los principios subyacentes de la ingeniería del software ahora son aceptados a lo largo de la industria. ¿Por qué, entonces, se ha visto su amplia aceptación sólo recientemente?

La respuesta, creo, se encuentra en la dificultad de la transición tecnológica y el cambio cultural que la acompaña. Aun cuando la mayoría de las personas aprecian la necesidad de una disciplina de ingeniería para el software, se lucha contra la inercia de la práctica pasada y se enfrentan nuevos dominios de aplicación (y los desarrolladores que trabajan en ellos), que parecen listos a repetir los errores del pasado.

2 En realidad, el crédito corresponde a Peter Freeman y Eric Munson, quienes convencieron a McGraw-Hill de que valía la pena intentarlo.

Para facilitar la transición se necesitan muchas cosas: un proceso de software ágil, adaptable y sensible; métodos más efectivos; herramientas más poderosas; mejor aceptación de los profesionales y apoyo de los gestores; y no pequeñas dosis de educación y "publicidad". La ingeniería del software no ha tenido el beneficio de la publicidad masiva, pero, conforme pasa el tiempo, el concepto se vende a sí mismo. De alguna manera, este libro es un "anuncio publicitario" para la tecnología.

El lector tal vez no esté de acuerdo con todos los enfoques descritos en este libro. Algunas de las técnicas y opiniones son controvertidas; otras deberán ajustarse para trabajar bien en diferentes entornos de desarrollo de software. Sin embargo, mi sincera esperanza es que *Ingeniería del software. Un enfoque práctico* haya delineado los problemas que se enfrentan, demostrado la fuerza de los conceptos de la ingeniería del software y ofrecido un marco de trabajo de los métodos y herramientas.

Conforme se avanza en el siglo xxi, el software se ha convertido en el producto más importante y en una industria primordial en el escenario mundial. Su impacto e importancia han transitado un largo camino. E incluso, una nueva generación de desarrolladores de software debe enfrentar muchos de los mismos desafíos que enfrentaron las primeras generaciones. Espero que las personas que enfrenten el reto —ingenieros de software— tendrán la sabiduría de desarrollar sistemas que mejoren la condición humana.

Referencias

[ACM98] ACM/IEEE-CS Joint Task Force, *Software Engineering Code of Ethics and Professional Practice*, 1998, disponible en http://www.acm.org/serving/se/code.htm

[BEC01] Beck, K. *et al.*, "Manifesto for Agile Software Development", http://www.agilemanifesto.org/.

[BOL91] Bollinger, T. y C. McGowen, "A Critical Look at software Capability Evaluations", en *IEEE Software*, julio de 1991, pp. 25-41.

[GIL96] Gilb, T., "What is Level Six?", *IEEE Software*, enero de 1996, pp. 97-98, 103

[HOP90] Hopper, M. D., "Rattling SABRE, New Ways to Compete on Information", en *Harvard Business Review*, mayo-junio de 1990.

[PAU93] Paulk, M. *et al* , *Capability Maturity Model for Software*, Software Engineering Institute, Carnegie Mellon University, 1993.

[PCM03] "Technologies to Watch", en *PC Magazine*, julio de 2003, disponible en http://www.pcmag.com/article2/0,4149,1130591,00.asp.

[PRE91] Pressman, R. S. y S. R. Herron, *Software Shock*, Dorset House, 1991.

[SEE03] The Software Engineering Ethics Research Institute, "UCITA Updates", 2003, disponible en http://seeri.etsu.edu/default.htm

Problemas y puntos a considerar

32.1. Obtener una copia de las principales revistas de negocios y noticias de esta semana (por ejemplo, *Newsweek*, *Time*, *Business Week*). Elaborar una lista de cada elemento del artículo o noticia que se pueda utilizar para ilustrar la importancia del software.

32.2. Uno de los dominios más reñidos de la aplicación del software son los sistemas y aplicaciones basados en Web. Estudiar cómo la gente, la comunicación y el proceso tienen que evolucionar para adaptarse al desarrollo de las WebApps de "siguiente generación".

32.3. Escribir una breve descripción del entorno de desarrollo de un ingeniero de software ideal hacia el 2010. Describir los elementos del entorno (hardware, software y tecnologías de comunicación) y su impacto sobre la calidad y el tiempo en que llega al mercado

32.4. Revisar el estudio de los modelos de proceso incrementales ágiles en el capítulo 4 Realizar una investigación y recopilar artículos recientes acerca de la materia Resumir las fortalezas y debilidades de los paradigmas ágiles con base en las experiencias subrayadas en los artículos

32.5. Inténtese desarrollar un ejemplo que comience con la recopilación de datos en bruto y llévese a cabo la adquisición de información, luego de conocimiento y, por último, de sabiduría

32.6. Proporcionar ejemplos específicos que ilustren uno de los ochos principios éticos de la ingeniería del software mencionados en la sección 32.7.

Otras lecturas y fuentes de información

Los libros que estudian las tendencias futuras del software y la computación abarcan una amplia variedad de temas técnicos, científicos, económicos, políticos y sociales Sterling (*Tomorrow Now*, Random House, 2002) recuerda que el progreso real rara vez es ordenado y eficiente Teich (*Technology and the Future*, Wadworth, 2002) presenta reflexivos ensayos acerca del impacto social de la tecnología y cómo la cultura cambiante da forma a la tecnología Naisbitt, Philips y Naisbitt (*High Tech/High Touch*, Nicholas Brealey, 2001) señalan que muchas personas se han "intoxicado" con la alta tecnología y que la "gran ironía de la era de la alta tecnología es que la humanidad se ha esclavizado a los dispositivos que se supone brindarían libertad". Zey (*The Future Factor*, McGraw-Hill, 2000) analiza cinco fuerzas que darán forma al destino humano durante este siglo. Canton (*Technofutures*, Hay House, 1999) estudia cómo la tecnología transformará los negocios en el siglo xxi Robertson (*The New Renaissance: Computers and the Next Level of Civilization*, Oxford University Press, 1998) argumenta que la revolución en la computación puede ser el avance individual más significativo en la historia de la civilización

Broderick (*Spike*, Forge, 2001) analiza el impacto de las tecnologías emergentes Dertrouzos y Gates (*What Will be: How the New World of Information Will Change Our Lives*, Harper-Business, 1998) ofrecen un estudio detallado de algunas de las direcciones que pueden tomar las tecnologías de la información en las primeras décadas de este siglo Barnatt (*Valueware: Technology, Humanity and Organization*, Praeger Publishing, 1999) presenta un intrigante estudio de una "economía de ideas" y cómo el valor económico se creará conforme evolucionen los cibernegocios. El de Negroponte (*Being Digital*, Alfred A. Knopf, 1995) fue un éxito de ventas a mediados del decenio de 1990 y continúa ofreciendo una interesante visión de la computación y de su impacto global

Kroker y Kroker (*Digital Delirium*, New World Perspectives, 1997) han editado una controvertida colección de ensayos, poemas y humor que examinan el impacto de las tecnologías digitales sobre las personas y la sociedad. Brin (*The Transparent Society: Will Technology Force Us to Choose Between Privacy and Freedom?*, Perseus Books, 1999) vuelven a revisar el continuo debate asociado con la inevitable pérdida de privacidad personal que acompaña al crecimiento de las tecnologías de la información Shenk (*Data Smog: Surviving the Information Glut*, HarperCollins, 1998) estudia los problemas asociados con una "sociedad infestada de información" que se sofoca por el volumen de información que produce el software.

Brockman (*The Next Fifty Years*, Vintage Books, 2002) y Miller y sus colegas (*21st Century Technologies: Promises and Perils of a Dynamic Future*, Brookings Institution Press, 1999) han editado una colección de artículos y ensayos acerca del impacto de la tecnología sobre las estructuras sociales, empresariales y económicas. Para aquellos interesados en los temas técnicos, Luryi, Xu y Zaslavsky (*Future Trends in Microelectronics*, Wiley, 1999) han editado una colección de artículos acerca de las probables direcciones para el hardware de computadora, con énfasis en las nanotecnologías. Hayzelden y Bigham (*Software Agents for Future Communication Systems*, Springer-Verlag, 1999) han editado una colección que analiza las tendencias en el desarrollo de agentes de software inteligentes

Conforme el software se vuelve cada vez más parte de la fabricación de virtualmente todas las facetas de la vida moderna, la "ciberética" ha evolucionado como un tema importante de estudio. Los libros de Spinello (*Cyberethics: Morality and Law in Cyberspace*, Jones & Bartlett Publishers, 2002), Halbert e Ingulli (*Cyberethics*, South-Western College Publishers, 2001) y Baird y sus colegas (*Cyberethics: Social and Moral Issues in the Computer Age*, Prometheus Books, 2000) consideran el tema en detalle. El gobierno estadounidense ha publicado un voluminoso reporte en CD-ROM (*21st Century Guide to Cybercrime*, Progressive Management, 2003) que considera todos los aspectos del crimen computacional, conflictos de propiedad intelectual y el Centro de Protección a la Infraestructura Nacional (NIPC, por sus siglas en inglés).

Kurzweil (*The Age of Spiritual Machines, When Computers Exceed Human Intelligence*, Viking/Penguin Books, 1999) argumenta que, dentro de 20 años, la tecnología de hardware tendrá la capacidad de modelar por completo el cerebro humano. Borgmann (*Holding on to Reality The Nature of Information at the Turn of the Millenium*, University of Chicago Press, 1999) ha escrito una interesante historia de la información, y rastrea su papel en la transformación de la cultura. Devlin (*InfoSense: Turning Information Into Knowledge*, W. H. Freeman & Co., 1999) intenta darle sentido al constante flujo de información que bombardea a la población diariamente. Gleick (*Faster: The Acceleration of Just About Everything*, Pantheon Books, 2000) estudia la tasa siempre en aceleración del cambio tecnológico y su impacto sobre todos los aspectos de la vida moderna. Jonscher (*The Evolution of Wired Life: From the Alphabet to the Soul-Catcher Chip-How Information Technologies Change Our World*, Wiley, 2000) argumenta que el pensamiento y la interacción humanas trascienden la importancia de la tecnología.

En Internet hay disponible una amplia variedad de fuentes de información acerca de las direcciones futuras en las tecnologías relacionadas con el software y la ingeniería de software. Una lista actualizada de referencias en la World Wide Web se puede encontrar en el sitio Web SEPA:
http://www.mhhe.com/pressman.

ÍNDICE ANALÍTICO

ÍNDICE DE SIGLAS MÁS COMUNES EN INGENIERÍA DEL SOFTWARE

Siglas español/inglés

Término equivalente en español	*Término equivalente en inglés*
ABC MPI (apreciación basada en el CMM para el mejoramiento del proceso interno) 37	**CMM** (based appraisal for internal procces improvement CBA IPI)
ACC (autoridad del control del cambio) 810	**CCA** (change control authority)
ADP (paradigma de diseño abstracto) 883	**ADP** (abstract design paradigm)
AECO (acoplamiento entre clases de objetos) 485	**CBO** (coupling between object classes)
AG2D (análisis geométrico bidimensional) 701	**2DGA** (two-dimensional geometric analysis)
AG3D (análisis geométrico tridimensional) 701	**3DGA** (three-dimensional geometric analysis)
AIE (archivos de interfaz externos) 475	**EIFs** (external interface files)
ALI (archivos lógicos internos) 474	**ILFs** (internal logical files)
AOO (análisis orientado a objetos) 201	**OODA** (object-oriented domain analysis)
APD (acceso público a datos) 495	**PAD** (public access to data members)
APH (árbol de profundidad de la herencia) 484	**DIT** (depth of the inheritance tree)
ARN (análisis relación-navegación) 560	**RNA** (relationship-navigation-analysis)
AVG (análisis del valor ganado) 742	**EVA** (earned value analysis)
AVL (análisis de valores límite) 437	**BVA** (boundary value analysis)
CAD (diseño asistido por computadora) 700	**CAD** (computer aided design)
CCYD (componentes comerciales ya desarrollados) 695	**COTS** (comercial off-the-shelf)
CDL (componentes comerciales de línea) 881	**COTS** (comercial off-the-shelf)
CE (consultas externas) 474	**EQs** (external inquiries)
CN (contador numérico) 696	**NC** (numerical control)
CO (complejidad de la operación) 491	**OC** (operation complexity)
COCOMO (modelo constructivo de costos) 710	**COCOMO** (constructive cost model)
COM (modelo de objetos para componentes) 889	**COM** (component object model)
CORBA (arquitectura común de distribución de objetos) 889	**CORBA** (common object request broker architecture)
CPTC (costo presupuestado para el trabajo calendarizado) 743	**BCWS** (budgeted cost of work scheduled)
CPTR (costo presupuestado del trabajo realizado) 743	**BCWP** (budgeted cost of work performed)
CRC (clase-responsabilidad-colaborador) 225	**FTR** (formal technical reviews)
CRTR (costo real del trabajo realizado) 743	**ACWP** (actual cost of work performer)
CTC (condición-transición-consecuencia) 759	**CTC** (condition–transition–consequence)
DAS (desarrollo adaptativo de software) 89	**ASD** (adaptative software development)
DBC (desarrollo basado en componentes) 63	**CBD** (component based development)
DBMS (gestor de bases de datos) 614	**DBMS** (database manager system)
DCA (diseño de contexto arquitectónico) 288	**ACD** (architecture context diagram)
DCBD (descubrimiento de conocimiento en base de datos) 278	**KDD** (knowledge discovery in databases)
DCC (desarrollo conducido por características) 95	**FDD** (feature driven development)
DCS (diagrama de contexto del sistema) 145	**SCD** (system context diagram)
DFD (diagrama de flujo de datos) 211, 298	**DFD** (data flow diagram)
DFS (diagrama de flujo del sistema) 146	**SFD** (system flow diagram)
DIE (desviación intencional de las especificaciones) 784	**IDS** (intentional deviation from specifications)
DII (documentación imprecisa o incompleta) 784	**IID** (inaccurate or incomplete documentation)

DMADV (definir, medir, analizar, diseñar y verificar) 786
DPR (diseño para la reutilización) 891
DRA (desarrollo rápido de aplicaciones) 53
DSOA (desarrollo de software orientado a aspectos) 65
DU (cadena definición-uso) 431
EAT (estructura de análisis del trabajo) 737
EC (especificación de control) 215
ECS (elementos de configuración del software) 800
EDI (entornos de desarrollo integrado) 413
EE (entradas externas) 474
EED (eficacia en la eliminación de defectos) 677
EIE (especificaciones incompletas o erróneas) 784
EIS (entorno de ingeniería del software) 696
ELD (error de la lógica del diseño) 784
EP (especificación de proceso) 217
EPI (equipos de producto integrado) 40
ER (exposición al riesgo) 757
ERD (errores de la representación de los datos) 784
FA (factor de acoplamiento) 487
FCM (falta de cohesión en métodos) 486
FCP (función de control periférica) 701
FDN (formas de navegación) 591
FIN (dependencia hacia dentro) 495
FIUC (facilidades de interfaz del usuario y control) 701
FPGC (facilidades de presentación gráfica de computadora) 701
GBD (gestión de bases de datos) 701
GCS (gestión de la configuración del software) 796
GIP (grupo independiente de prueba) 386
GUIs (interfaces gráficas de usuario) 452
ICED (índice de calidad de la estructura de diseño) 480
ICI (interfaz de componente inconsistente) 784
ICOA (ingeniería de componentes orientada a aspectos) 65
IDCo (índice de desempeño del costo) 744
IHC (interfaz hombre-computadora ambigua o inconsistente) 784
IMCM (integración del modelo de capacidad de madurez) 29
IMS (índice de madurez del software) 496
IPA (interfaz de programación de la aplicación) 887
IPN (ingeniería de procesos de negocios) 140
IR (ingeniería de requisitos) 157
ISBC (ingeniería del software basada en componentes) 879
ISO (organización internacional de estandarización) 38
IU (interfaz con el usuario) 264
KLDC (miles de líneas de código) 669
LDA (lenguaje de descripción arquitectónica) 296
LDP (lenguaje de diseño de programas) 217
MA (modelado ágil) 97
MACA (método de análisis de compensación para la arquitectura) 294
MAD (módulos de análisis de diseño) 701
MCC (mala interpretación de la comunicación del cliente) 784
MDHOO (método de diseño hipermedia orientado a objetos) 595

DMADV (define, measure, analyze, design, and verify)
DFR (design for reuse)
RAD (rapid application development)
AOSD (aspect - oriented software development)
DU (definition-use chain)
WBS (work breakdown structure)
CSPEC (control specification)
SCIs (software configuration items)
IDEs (integrated development environment)
EIs (external inputs)
DRE (deffect removal efficiency)
IES (incomplete or erroneous specification)
SEE (software engineering environment)
EDL (error in design logic)
PSPEC (process specification)
IPT (integrated product teams)
RE (risk exposure)
ERD (error in data representation)
CF (coupling factor)
LCOM (lack of cohesion in methods)
FCF (peripheral control function)
WoN (ways of navigating)
FIN (fan-in)
UICF (user interface and control facilities)
CGDF (computer graphics display facilities)

DBM (database management)
SCM (software configuration management)
ITG (independent test group)
GUIs (graphical user interfaces)
DSQI (design structure quality index)
ICI (inconsistent component interface)
AOCE (aspect-oriented component engineering)

CPI (cost performance index)
HCI (human-computer interaction)

CMMI (capability maturity model integration)

SMI (software maturity index)
API (application programming interface)
BPE (business processes engineering)
RE (requirements engineering)
CBSE (component based software engineering)
ISO (international organization for standardization)
UI (user interface)
KLOC (thousands lines of code)
ADLs (architectural description languages)
PDL (program design language)
AM (agile modeling)
ATAM (architecture trade-off analisis method)

DAM (design analysis modules)
MCC (misinterpretation of customer communication)

OOHDM (object-oriented hipermedia design method)

MDSD (método de desarrollo de sistemas dinámicos) 91
ME (metas específicas) 31
MEIEMP (método de evaluación de la IMCM estándar para el mejoramiento del proceso) 37
MEPS (mejora estadística del proceso de software) 667
MFH (método del factor de herencia) 487
MG (metas genéricas) 32
MIS (misceláneo) 784
MMCGP (modelo de madurez de la capacidad de gestión de personal) 641
MPC (métodos ponderados por clase) 484
MRC (método de ruta crítica) 736
MVC (modelo-vista-controlador) 589
NCR (número de clases raíz) 495
NCSC (nuevos componentes de software comerciales) 63
NDD (número de descendientes) 485
NOA (número de operaciones añadidas) 488
NPO (nuevos puntos objeto) 711
NPO_{prom} (número promedio de parámetros de la operación) 492
OCI (orden de cambio de la ingeniería) 810
OCL (lenguaje de restricción de objeto) 332
OMG (grupo de gestión de objetos) 889
OPM (objetivo/pregunta/métrica) 470
ORB (distribuidor de objetos) 889
PAC (principio abierto-cerrado) 322
PCC (principio del cierre común) 325
PCR (principio común de la reutilización) 325
PDL (lenguaje de diseño de programas) 343
PE (prácticas específicas) 31
PER (principio de equivalencia entre reutilización y versión) 325
PERT (técnica de evaluación y revisión de programa) 736
PF (punto de función) 474
PG (prácticas genéricas) 32
PID (principio de inversión de la dependencia) 324
PIE (prueba incompleta o errónea) 784
PNR (curva Putnam-Norden-Rayleigh) 730
POA (programación orientada a aspectos) 65
PSE (proceso de software en equipo) 40
PSI (principio de segregación de la interfaz) 324
PSL (principio de sustitución de Liskov) 324
PSP (proceso de software personal) 39
PU (proceso unificado) 67
PYP (porcentaje público y protegido) 495
QFD (despliegue de la función de calidad) 171
RPC (respuesta para una clase) 485
RSGR (reducción, supervisión y gestión del riesgo) 761
RTF (revisión técnica formal) 774
SCCT (sistema de clasificación de cinta transportadora) 145
SE (salidas externas) 474
SEI (instituto de ingeniería del software) 29
SQA (garantía de la calidad del software) 767
SQL (lenguaje de consultas estructurado) 614

DSDM (dynamic systems development method)
SG (specific goals)
SCAMPI (standard CMMI assessment method for process improvement)
SSPI (statistical software process improvement)
MIF (method inheritance factor)
GG (generic goals)
MIS (miscellaneous)
PM-CMM (people management capability maturity model)
WMC (weighted methods per class)
CPM (critical path method)
MVC (model–view–controller)
NOR (number of root classes)
NCSC (commercial off-the-self (COTS) software component)
NOC (number of children)
NOA (number of operations added by a subclass)
NOO (number of operations overridden by subclass)
NP_{avg} (average number of parameters per operation)

ECO (engineering change order)
OCL (object constraint language)
OMG (object management group)
GQM (goal/question/metric)
ORB (object request broker)
OCP (open closed principle)
CCP (common closure principle)
CRP (common reuse principle)
PDL (program design language)
SP (specific practices)
REP (release reuse equivalency principle)

PERT (program evaluation and review technique)

FP (function points)
GP (generic practices)
DIP (dependency inversion principle)
IET (incomplete or erroneous testing)
PNR (Putnam-Norden-Rayleigh curve)
AOP (aspect-oriented programming)
TSP (team software process)
ISP (interface segregation principle)
LSP (Liskov substitution principle)
PSP (personal software process)
UP (unified process)
PAP (percent public protected)
QFD (quality function deployment)
RFC (response for a class)
RMMM (risk mitigation, monitoring and management)
FTR (formal technical reviews)
CLSS (conveyor line sorting system)

EOs (eos (external outputs)
SEI (software engineering institute)
SQA (software quality assurance)
SQL (structured query language)

SVC (sistema de versiones concurrentes) 809
TC (tamaño de la clase) 488
TI (tecnología de la información) 278
TLP (error de la traducción del diseño al lenguaje de programación) 784
TMC (tiempo medio de cambio) 677
TMDF y **TMDR** (tiempo medio de falla y tiempo medio de reparación) 787
TMEF (tiempo medio entre fallas) 787, 874
TMR (tiempo medio de reparación) 407
TO_{prom} (tamaño promedio de operación) 491
UML (lenguaje de modelado unificado) 68
USN (unidad semántica de navegación) 591
VAD (visión abstracta de datos) 597
VC (varianza del costo) 744
VEP (violación de los estándares de programación) 784
VyV (verificación y validación) 384

CVS (concurrent versions system)
CS (class size)
IT (information technologies)
PLT (error programming language translation)

MTTC (mean-time-to-change)
MTTC & MTTF (mean-time-to-charge) and (mean-time-to-failure)
MTBF (mean time between failures)
MTTR (mean time to repair)
OS_{avg} (average operation size)
UML (unified modeling language)
NSU (navigation semantic unit)
ADV (abstract data view)
CV (cost variance)
VPS (violation of programming standards)
V&V (verification and validation)

Siglas inglés/español

Término en inglés	*Término equivalente en español*
3DGA (three-dimensional geometric analysis)	**AG3D** (análisis geométrico tridimensional) 701
2DGA (two-dimensional geometric analysis)	**AG2D** (análisis geométrico bidimensional) 701
ACD (architecture context diagram)	**DCA** (diseño de contexto arquitectónico) 288
ACWP (actual cost of work performer)	**CRTR** (costo real del trabajo realizado) 743
ADLs (architectural description languages)	**LDA** (lenguaje de descripción arquitectónica) 296
ADP (abstract desing paradigm)	**ADP** (paradigma de diseño abstracto) 883
ADV (abstract data view)	**VAD** (visión abstracta de datos) 597
AM (agile modeling)	**MA** (modelado ágil) 97
AOCE (aspect-oriented component engineering)	**ICOA** (ingeniería de componentes orientada a aspectos) 65
AOP (aspect-oriented programming)	**POA** (programación orientada a aspectos) 65
AOSD (aspect-oriented software development)	**DSOA** (desarrollo de software orientado a aspectos) 65
API (application programming interface)	**IPA** (interfaz de programación de la aplicación) 887
ASD (adaptative software development)	**DAS** (desarrollo adaptativo de software) 89
ATAM (architecture trade-off analisis method)	**MACA** (método de análisis de compensación para la arquitectura) 294
BCWP (budgeted cost of work performed)	**CPTR** (costo presupuestado del trabajo realizado) 743
BCWS (budgeted cost of work scheduled)	**CPTC** (costo presupuestado para el trabajo calendarizado) 743
BPE (business processes engineering)	**IPN** (ingeniería de procesos de negocios) 140
BVA (boundary value analysis)	**AVL** (análisis de valores límite) 437
CAD (computer aided design)	**CAD** (diseño asistido por computadora) 700
CBD (component based development)	**DBC** (desarrollo basado en componentes) 63, 886
CBO (coupling between object classes)	**AECO** (acoplamiento entre clases de objetos) 485
CBSE (component based software engineering)	**ISBC** (ingeniería del software basada en componentes) 879
CCA (change control authority)	**ACC** (autoridad del control del cambio) 810
CCP (common closure principle)	**PCC** (principio del cierre común) 325
CF (coupling factor)	**FA** (factor de acoplamiento) 487
CGDF (computer graphics display facilities)	**FPGC** (facilidades de presentación gráfica de computadora) 701

COTS (comercial off-the-shelf)
CLSS (conveyor line sorting system)

CMM (based appraisal for internal procces improvement CBA IPI)
CMMI (capability maturity model integration)

COCOMO (constructive cost model)
COM (component object model)
CORBA (common object request broker architecture)

CPI (cost performance index)
CPM (critical path method)
CRP (common reuse principle)
CS (class size)
CSPEC (control specification)
CTC (condition – transition – consequence)
CV (cost variance)
CVS (concurrent versions system)
DAM (design analysis modules)
DBM (database management)
DBMS (database manager system)
DFD (data flow diagram)
DFR (design for reuse)
DIP (dependency inversión principle)
DIT (depth of the inheritance tree)
DMADV (define, measure, analyze, design, and verify)

DRE (deffect removal efficiency)
DSDM (dynamic systems development method)

DSQI (design structure quality index)
DU (definition-use chain)
ECO (engineering change order)
EDL (error in design logic)
EIs (external inputs)
EIFs (external interface files)
EOs (external outputs)
EQs (external inquiries)
ERD (error in data representation)
EVA (earned value analysis)
FDD (feature driven development)
FIN (fan in)
FP (function points)
FTR (formal technical reviews)
FTR (formal technical reviews)
GG (generic goals)
GQM (goal/question/metric)
GP (generic practices)
GUIs (graphical user interfaces)
HCI (human–computer interaction)

ICI (inconsistent component interface)
IDEs (integrated development environments)
IDS (intentional deviation from specifications)
IES (incomplete or erroneous specification)
IET (incomplete or erroneous testing)

CDL (componentes comerciales de línea) 695
SCCT (sistema de clasificación de cinta transportadora) 146
ABC MPI (apreciación basada en el cmm para el mejoramiento del proceso interno) 37
IMCM (integración del modelo de capacidad de madurez) 29
COCOMO (modelo constructivo de costos) 710
COM (modelo de objetos para componentes) 889
CORBA (arquitectura común de distribución de objetos) 889
IDCo (índice de desempeño del costo) 744
MRC (método de ruta crítica) 736
PCR (principio común de la reutilización) 325
TC (tamaño de la clase) 488
EC (especificación de control) 215
CTC (condición-transición-consecuencia) 759
VC (varianza del costo) 744
SVC (sistema de versiones concurrentes) 809
MAD (módulos de análisis de diseño) 701
GBD (gestión de bases de datos) 701
DBMS (gestor de bases de datos) 614
DFD (diagrama de flujo de datos) 211, 298
DPR (diseño para la reutilización) 891
PID (principio de inversión de la dependencia) 324
APH (árbol de profundidad de la herencia) 484
DMADV (definir, medir, analizar, diseñar y verificar) 786
EED (eficacia en la eliminación de defectos) 677
MDSD (método de desarrollo de sistemas dinámicos) 91
ICED (índice de calidad de la estructura de diseño) 480
DU (cadena definición-uso) 431
OCI (orden de cambio de la ingeniería) 810
ELD (error de la lógica del diseño) 784
EE (entradas externas) 474
AIE (archivos de interfaz externos) 475
SE (salidas externas) 474
CE (consultas externas) 474
ERD (errores de la representación de los datos) 784
AVG (análisis del valor ganado) 742
DCC (desarrollo conducido por características) 95
FIN (dependencia hacia dentro) 495
PF (punto de función) 474, 670
CRC (clase-responsabilidad-colaborador) 225
RTF (revisión técnica formal) 774
MG (metas genéricas) 32
OPM (objetivo/pregunta/métrica) 470
PG (prácticas genéricas) 32
GUIs (interfaces gráficas de usuario) 452
IHC (interfaz hombre-computadora ambigua o inconsistente) 784
ICI (interfaz de componente inconsistente) 784
EDI (entornos de desarrollo integrado) 413
DIE (desviación intencional de las especificaciones) 784
EIE (especificaciones incompletas o erróneas) 784
PIE (prueba incompleta o errónea) 784

IID (inaccurate or imcomplete documentation)
ILFs (internal logical files)
IPT (integrated product teams)
ISO (international organization for standardization)

ISP (interface segregation principle)
IT (information technologies)
ITG (independent test group)
KDD (knowledge discovery in databases)

KLOC (thousands lines of code)
LCOM (lack of cohesion in methods)
LSP (Liskov substitution principle)
MCC (misinterpretation of customer communication)

MIF (method inheritance factor)
MIS (miscellaneous)
MTBF (mean time between failures)
MTTC & MTTF (mean-time-to-charge) and (mean-time-to-failure)
MTTC (mean-time-to-change)
MTTR (mean time to repair)
MVC (model–view–controller)
NC (numerical control)
NCSC (commercial off–the–self (COTS) software component)
NOA (number of operations added by a subclass)
NOC (number of children)
NOO (number of operations overridden by subclass)
NOR (number of root classes)
NP_{avg} (average number of parameters per operation)

NSU (navigation semantic unit)
OC (operation complexity)
OCL (object constraint language)
OCP (open closed principle)
OMG (object management group)
OODA (object-oriented domain analysis)
OOHDM (object -oriented hipermedia design method)

ORB (object request broker)
OS_{avg} (average operation size)
PAD (public access to data members)
PAP (percent public and protected)
PCF (peripheral control function)
PDL (program design language)
PERT (program evaluation and review technique)
PLT (error programming language translation)

PM-CMM (people management capability maturity model)

PNR (Putnam-Norden-Rayleigh curve)
PSP (personal software process)
PSPEC (process specification)
QFD (quality function deployment)
RAD (rapid application development)
RE (requirements engineering)
RE (risk exposure)

DII (documentación imprecisa o incompleta) 784
ALI (archivos lógicos internos) 474
EPI (equipos de producto integrado) 40
ISO (organización internacional de estandarización) 38
PSI (principio de segregación de la interfaz) 324
TI (tecnología de la información) 278
GIP (grupo independiente de prueba) 386
DCBD (descubrimiento de conocimiento en base de datos) 278
KLDC (miles de líneas de código) 669
FCM (falta de cohesión en métodos) 485
PSL (principio de sustitución de Liskov) 324
MCC (mala interpretación de la comunicación del cliente) 784
MFH (método del factor de herencia) 487
MIS (misceláneo) 784
TMEF (tiempo medio entre fallas) 787
TMDF y TMDR (tiempo medio de falla y tiempo medio de reparación) 787
TMC (tiempo medio de cambio) 677
TMR (tiempo medio de reparación) 407
MVC (modelo-vista-controlador) 589
CN (contador numérico) 696
NCSC (nuevos componentes de software comerciales) 63
NOA (número de operaciones añadidas) 488
NDD (número de descendientes) 485
NPO (nuevos puntos objeto) 711
NCR (número de clases raíz) 495
NPO_{prom} (número promedio de parámetros de la operación) 492
USN (unidad semántica de navegación) 591
CO (complejidad de la operación) 491
OCL (lenguaje de restricción de objeto) 332
PAC (principio abierto-cerrado) 322
OMG (grupo de gestión de objetos) 889
AOO (análisis orientado a objetos) 201
MDHOO (método de diseño hipermedia orientado a objetos) 595
ORB (distribuidor de objetos) 889
TO_{prom} (tamaño promedio de operación) 491
APD (acceso público a datos) 495
PYP (porcentaje público y protegido) 495
FCP (función de control periférica) 701
LDP (lenguaje de diseño de programas) 217
PERT (técnica de evaluación y revisión de programa) 736
TLP (error de la traducción del diseño al lenguaje de programación) 784
MMCGP (modelo de madurez de la capacidad de gestión de personal) 641
PNR (curva Putnam-Norden-Rayleigh) 730
PSP (proceso de software personal) 39
EP (especificación de proceso) 217
QFD (despliegue de la función de calidad) 171
DRA (desarrollo rápido de aplicaciones) 53
IR (ingeniería de requisitos) 157
ER (exposición al riesgo) 757

REP (release reuse equivalency principle)

RFC (response for a class)
RMMM (risk mitigation, monitoring, and management)
RNA (relationship-navegation-analisys)
SCAMPI (standard CMMI assessment method for process improvement)
SCD (system context diagram)
SCIs (software configuration items)
SCM (software configuration management)
SG (specific goals)
SEE (software engineering environment)
SEI (software engineering institute)
SFD (system flow diagram)
SMI (software maturity index)
SQA (software quality assurance)
SQL (structured query language)
SP (specific practices)
SSPI (statistical software process improvement)
TSP (team software process)
UI (user interface)
UICE (user interface and control facilities)
UML (unified modeling language)
UP (unified process)
V&V (verification and validation)
VPS (violation of programming standards)
WBS (work breakdown structure)
WMC (weighted methods per class)
WoN (ways of navegating)

PER (principio de equivalencia entre reutilización y versión) 325
RPC (respuesta para una clase) 485
RSGR (reducción, supervisión y gestión del riesgo) 761
ARN (análisis relación-navegación) 560
MEIEMP (método de evaluación de la IMCM estándar para el mejoramiento del proceso) 37
DCS (diagrama de contexto del sistema) 145
ECS (elementos de configuración del software) 800
GCS (gestión de la configuración del software) 796
ME (metas específicas) 31
EIS (entorno de ingeniería del software) 696
SEI (instituto de ingeniería del software) 29
DFS (diagrama de flujo del sistema) 146
IMS (índice de madurez del software) 496
SQA (garantía de la calidad del software) 767
SQL (lenguaje de consultas estructurado) 614
PE (prácticas específicas) 31
MEPS (mejora estadística del proceso de software) 667
PSE (proceso de software en equipo) 40
IU (interfaz con el usuario) 264
FIUC (facilidades de interfaz del usuario y control) 701
UML (lenguaje de modelado unificado) 68
PU (proceso unificado) 67
VyV (verificación y validación) 384
VEP (violación de los estándares de programación) 784
EAT (estructura de análisis del trabajo) 737
MPC (métodos ponderados por clase) 484
FdN (formas de navegación) 591

Roger S. Pressman es una autoridad reconocida a nivel internacional en el mejoramiento del proceso del software y en las tecnologías de ingeniería del software.

Lo nuevo en esta edición

Los 32 capítulos de la sexta edición se han organizado en cinco partes:

- Parte 1. *El proceso del software*, presenta diferentes perspectivas del proceso del software y considera todos los modelos de proceso importantes; además, aborda el debate entre las filosofías del proceso prescriptivo y del proceso ágil.
- Parte 2. *Práctica de la ingeniería del software*, presenta métodos de análisis, diseño y prueba con especial interés en las técnicas orientadas a objetos y el modelado UML.
- Parte 3. *Aplicación de la ingeniería Web*, presenta un enfoque completo de ingeniería para el análisis, diseño y prueba de aplicaciones Web.
- Parte 4. *Gestión de proyectos de software*, presenta temas relevantes para quienes planean, gestionan y controlan un proyecto de software.
- Parte 5. *Temas avanzados en ingeniería del software*, presenta capítulos que abordan métodos formales, ingeniería del software de sala limpia, ingeniería de software basada en componentes, reingeniería y tendencias futuras.

Además de muchos capítulos nuevos y significativamente revisados, la sexta edición incluye aproximadamente 120 recuadros que:

- Permiten al lector seguir a un equipo de proyecto (ficticio) conforme planifica y diseña un sistema basado en computadora.
- Proporciona estudios complementarios de temas selectos.
- Subraya los "conjuntos de tareas" que describen el flujo de trabajo para actividades selectas de ingeniería del software.
- Sugiere herramientas automatizadas de interés para los temas de los capítulos.

McGraw-Hill Interamericana

ISBN 970-10-5473-3

9 789701 054734

Visite nuestra página WEB
www.mcgraw-hill-educacion.com