### 16.6.2 Seguridad del software

**Cita:**

"La seguridad de las personas debe ser la ley máxima."

**Cicerón**

La *seguridad del software* es una actividad del aseguramiento del software que se centra en la identificación y evaluación de los peligros potenciales que podrían afectarlo negativamente y que podrían ocasionar que falle todo el sistema. Si los peligros se identifican al principio del proceso del software, las características de su diseño se especifican de modo que los eliminen o controlen.

Como parte de la seguridad del software, se lleva a cabo un proceso de modelado y análisis. Inicialmente se identifican los peligros y se clasifican según su riesgo. Por ejemplo, algunos de los peligros asociados con un control de crucero basado en computadora para un automóvil podrían ser los siguientes: 1) ocasionar una aceleración incontrolada que no pudiera detenerse, 2) no responder a la presión en el pedal de frenado (porque se apague), 3) no encender cuando se active el interruptor y 4) perder o ganar velocidad poco a poco. Una vez identificados estos peligros en el nivel del sistema, se utilizan técnicas de análisis para asignar severidad y probabilidad de ocurrencia a cada uno.[3] Para ser eficaz, el software debe analizarse en el contexto de todo el sistema. Por ejemplo, un error sutil en la entrada de un usuario (las personas son componentes del sistema) podría ampliarse por una falla del software y producir datos de control que situaran equivocadamente un dispositivo mecánico. Si y sólo si se encontrara un único conjunto de condiciones ambientales externas, la posición falsa del dispositivo mecánico ocasionaría una falla desastrosa. Podrían usarse técnicas de análisis [Eri05], tales como árbol de fallas, lógica en tiempo real y modelos de red de Petri, para predecir la cadena de eventos que ocasionarían los peligros, así como la probabilidad de ocurrir que tendría cada uno de los eventos para generar la cadena.

**Cita:**

"No puedo imaginar ninguna condición que hiciera que esta nave se hundiera. La construcción naval moderna ha llegado más allá de eso".

**E. I. Smith, capitán del *Titanic***

Una vez identificados y analizados los peligros, pueden especificarse requerimientos relacionados con la seguridad para el software. Es decir, la especificación contendría una lista de eventos indeseables y las respuestas deseadas del sistema ante ellos. Después se indicaría el papel del software en la administración indeseable de los mismos.

Aunque la confiabilidad y la seguridad del software están muy relacionadas, es importante entender la sutil diferencia entre ellas. La primera utiliza técnicas de análisis estadístico para determinar la probabilidad de que ocurra una falla del software. Sin embargo, la ocurrencia de una falla no necesariamente da como resultado un peligro o riesgo. La seguridad del software examina las formas en las que las fallas generan condiciones que llevan a un peligro. Es decir, las fallas no se consideran en el vacío, sino que se evalúan en el contexto de la totalidad del sistema basado en computadora y de su ambiente.

**WebRef**

En la dirección **www.safeware-eng.com/**, se encuentran varios artículos sobre seguridad del software.

El estudio exhaustivo de la seguridad del software está más allá del alcance de este libro. Si el lector está interesado en la seguridad del software y en otros aspectos relacionados, consulte [Smi05], [Dun02] y [Lev95].

## 16.7 Las normas de calidad ISO 9000[4]

Un *sistema de aseguramiento de la calidad* se define como la estructura organizacional, responsabilidades, procedimientos, procesos y recursos necesarios para implementar la administración de la calidad [ANS87]. Los sistemas de aseguramiento de la calidad se crean para ayudar a las organizaciones a asegurar que sus productos y servicios satisfagan las expectativas del con-

3 Este enfoque es similar a los métodos de análisis del riesgo descritos en el capítulo 28. La diferencia principal es el énfasis que se pone en aspectos de la tecnología en lugar de en los relacionados con el proyecto.

4 Esta sección, escrita por Michael Stovski, ha sido adaptada a partir de "Fundamentos de ISO 9000", libro de trabajo desarrollado para *Essential Software Engineering*, video desarrollado por R. S. Pressman & Associates, Inc. Se reimprime con su autorización.

sumidor gracias a que cumplan con sus especificaciones. Estos sistemas cubren una amplia variedad de actividades, que contemplan todo el ciclo de vida del producto, incluidos planeación, control, medición, pruebas e informes, así como la mejora de los niveles de calidad en todo el proceso de desarrollo y manufactura. La norma ISO 9000 describe en términos generales los elementos de aseguramiento de la calidad que se aplican a cualquier negocio, sin importar los productos o servicios ofrecidos.

Para registrarse en alguno de los modelos del sistema de aseguramiento de la calidad contenidos en la ISO 9000, por medio de auditores externos se revisan en detalle el sistema y las operaciones de calidad de una compañía, respecto del cumplimiento del estándar y de la operación eficaz. Después de un registro exitoso, el grupo de registro representado por los auditores emite un certificado para la compañía. Auditorías semestrales de supervisión aseguran el cumplimiento continuo de la norma.

**WebRef**

En la dirección **www.tantara.ab.ca/info.htm**, se encuentran muchos vínculos hacia los recursos de la norma ISO 9000/9001.

Los requerimientos esbozados por la norma ISO 9001:2000 se dirigen a temas tales como responsabilidad de la administración, sistema de calidad, revisión del contrato, control del diseño, documentación y control de datos, identificación del producto y su seguimiento, control del proceso, inspección y pruebas, acciones correctivas y preventivas, registros del control de calidad, auditorías internas de calidad, capacitación, servicio y técnicas estadísticas. A fin de que una organización de software se registre en la ISO 9001:2000, debe establecer políticas y procedimientos que cumplan cada uno de los requerimientos mencionados (y otros más), y después demostrar que sigue dichas políticas y procedimientos. Si el lector desea más información sobre la norma ISO 9001:2000, consulte [Ant06], [Mut03] o [Dob04].

**INFORMACIÓN**

### *La norma ISO 9001:2000*

La descripción siguiente define los elementos básicos de la norma ISO 9001:2000. Información completa sobre la misma se obtiene en la Organización Internacional de Normas (**www.iso.ch**) y en otras fuentes de internet (como en **www.praxiom.com**).

Establecer los elementos de un sistema de administración de la calidad.
- Desarrollar, implementar y mejorar el sistema.
- Definir una política que ponga el énfasis en la importancia del sistema.

Documentar el sistema de calidad.
- Describir el proceso.
- Producir un manual de operación.
- Desarrollar métodos para controlar (actualizar) documentos.
- Establecer métodos de registro.

Apoyar el control y aseguramiento de la calidad.
- Promover la importancia de la calidad entre todos los participantes.
- Centrarse en la satisfacción del cliente.
- Definir un plan de calidad que se aboque a los objetivos, responsabilidades y autoridad.
- Definir mecanismos de comunicación entre los participantes.

Establecer mecanismos de revisión para el sistema de administración de la calidad.
- Identificar métodos de revisión y mecanismos de retroalimentación.
- Definir procedimientos para dar seguimiento.

Identificar recursos para la calidad, incluidos personal, capacitación y elementos de la infraestructura.
- Establecer mecanismos de control.
- Para la planeación
- Para los requerimientos del cliente
- Para las actividades técnicas (tales como análisis, diseño y pruebas)
- Para la vigilancia y administración del proyecto

Definir métodos de corrección.
- Evaluar datos y métricas de la calidad.
- Definir el enfoque para la mejora continua del proceso y la calidad.

## 16.8 EL PLAN DE ACS

El *Plan de ACS* proporciona un mapa de ruta para instituir el aseguramiento de la calidad del software. Desarrollado por el grupo de ACS (o por el equipo del software si no existe un grupo de ACS), el plan funciona como plantilla para las actividades de ACS que se instituyen para cada proyecto de software.

La IEEE [IEEE93] ha publicado una norma para el ACS. Ésta recomienda una estructura que identifica lo siguiente: 1) propósito y alcance del plan, 2) descripción de todos los productos del trabajo de ingeniería de software (tales como modelos, documentos, código fuente, etc.) que se ubiquen dentro del ámbito del ACS, 3) todas las normas y prácticas aplicables que se utilicen durante el proceso del software, 4) acciones y tareas del ACS (incluidas revisiones y auditorías) y su ubicación en el proceso del software, 5) herramientas y métodos que den apoyo a las acciones y tareas de ACS, 6) procedimientos para la administración de la configuración del software (véase el capítulo 22), 7) métodos para unificar las salvaguardas y para mantener todos los registros relacionados con el ACS y 8) roles y responsabilidades relacionados con la calidad del producto.

**HERRAMIENTAS DE SOFTWARE**

### *Administración de la calidad del software*

**Objetivos:** El objetivo de las herramientas del ACS es ayudar al equipo del proyecto a evaluar y mejorar la calidad del producto del trabajo de software.

**Mecánica:** La mecánica de las herramientas varía. En general, el objetivo consiste en evaluar la calidad de un producto específico. Nota: Es frecuente que dentro de la categoría de herramientas para el ACS, se incluya una amplia variedad de herramientas para someter a prueba al software (véanse los capítulos 17 a 20).

**Herramientas representativas**[5]

*ARM*, desarrollada por la NASA (**state.gsfc.nasa.gov/tools/index.html**), proporciona mediciones que se utilizan para evaluar la calidad de un documento de requerimientos de software.

*QPR ProcessGuide and Scorecard*, desarrollada por QPR Software (**www.qpronline.com**), da apoyo para establecer Seis Sigma y otros enfoques de administración de la calidad.

*Quality Tools and Templates*, desarrollada por iSixSigma (**www.isixsigma.com/tt/**), describe un amplio abanico de herramientas y métodos útiles para la administración de la calidad.

*NASA Quality Resources*, desarrollada por el Centro Coddard de Vuelos Espaciales (**sw-assurance.gsfc.nasa.gov/index.php**), contiene formatos, plantillas, listas de verificación y herramientas que son útiles para el ACS.

## 16.9 RESUMEN

El aseguramiento de la calidad del software es una actividad sombrilla de la ingeniería de software que se aplica en cada etapa del proceso del software. El ACS incluye procedimientos para la aplicación eficaz de métodos y herramientas, supervisa las actividades de control de calidad, tales como las revisiones técnicas y las pruebas del software, procedimientos para la administración del cambio, y procedimientos para asegurar el cumplimiento de las normas y mecanismos de medición y elaboración de reportes.

Para llevar a cabo el aseguramiento de la calidad del software de manera adecuada, deben recabarse, evaluarse y divulgarse datos sobre el proceso de la ingeniería de software. Los métodos estadísticos aplicados al ACS ayudan a mejorar la calidad del producto y del proceso de software mismo. Los modelos de confiabilidad del software amplían las mediciones, lo que permite que los datos obtenidos acerca de los defectos se extrapolen hacia tasas de falla proyectadas y hacia la elaboración de pronósticos de confiabilidad.

En resumen, deben tomarse en cuenta las palabras de Dunn y Ullman [Dun82]: "El aseguramiento de la calidad del software es el mapeo de los preceptos administrativos y de las disciplinas de diseño del aseguramiento de la calidad, en el ámbito administrativo y tecnológico aplicable a la ingeniería de software." La capacidad de asegurar la calidad es la medida de una

5 Las herramientas mencionadas aquí no son obligatorias, sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

disciplina madura de la ingeniería. Cuando el mapeo se lleva a cabo con éxito, el resultado es una ingeniería de software madura.

## Problemas y puntos por evaluar

**16.1.** Algunas personas afirman que "el control de la variación es el corazón del control de calidad". Como todo programa que se crea es diferente de cualquier otro programa, ¿cuáles son las variaciones que se buscan y cómo se controlan?

**16.2.** ¿Es posible evaluar la calidad del software si el cliente cambia continuamente lo que se supone que debe hacerse?

**16.3.** La calidad y confiabilidad son conceptos relacionados, pero difieren en lo fundamental por varias razones. Analice las diferencias.

**16.4.** ¿Un programa puede corregirse y aún así ser confiable? Explique su respuesta.

**16.5.** ¿Un programa puede corregirse y tener buena calidad? Explique lo que responda.

**16.6.** ¿Por qué es frecuente que haya tensiones entre el grupo de ingeniería de software y el del aseguramiento de la calidad? ¿Es saludable eso?

**16.7.** El lector tiene la responsabilidad de mejorar la calidad del software en su organización. ¿Qué es lo primero que debe hacer? ¿Qué es lo siguiente?

**16.8.** Además de contar los errores y defectos, ¿hay otras características cuantificables de software que impliquen calidad? ¿Cuáles son y cómo podrían medirse directamente?

**16.9.** El concepto del tiempo medio para la falla del software es objeto de críticas. Explique por qué.

**16.10.** Considere dos sistemas cuya seguridad sea crítica y que estén controlados por computadora. Enliste al menos tres peligros que se relacionen directamente con fallas del software.

**16.11.** Obtenga una copia de las normas ISO 9001:2000 e ISO 9000-3. Prepare una presentación que analice tres requerimientos de ISO 9001 y la forma en la que se apliquen en el contexto del software.

## Lecturas y fuentes de información adicionales

Los libros de Hoyle (*Quality Management Fundamentals*, Butterworth-Heinemann, 2007), Tian (*Software Quality Engineering*, Wiley-IEEE Computer Society Press, 2005), El Emam (*The ROI from Software Quality*, Auerbach, 2005) y Horch (*Practical Guide to Software Quality Management*, Artech House, 2003), y Nance y Arthur (*Managing Software Quality*, Springer, 2002) son presentaciones excelentes en el nivel de administración acerca de los beneficios de los programas formales de aseguramiento de la calidad del software de computadora. Las obras de Deming [Dem86], Juran (*Juran on Quality by Design*, Free Press, 1992) y Crosby ([Cro79], así como *Quality is Still Free*, McGraw-Hill, 1995) no se abocan al software, pero son una lectura obligada para los altos directivos que tengan responsabilidades en el desarrollo del software. Gluckman y Roome (*Everyday Heroes of the Quality Movement*, Dorset House, 1993) humanizan los aspectos de la calidad a través de la historia de los actores participantes en el proceso. Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995) presenta un enfoque cuantitativo de la calidad del software.

Los libros de Evans (*Total Quality: Management, Organization and Strategy*, 4a. ed., South Western College Publishing, 2004), Bru (*Six Sigma for Managers*, McGraw-Hill, 2005) y Dobb (*ISO 9001:2000 Quality Registration Step-by-Step*, 3a. ed., Butterworth-Heinemann, 2004) son representativos de los muchos que se han escrito sobre Seis Sigma e ISO 9001:2000, respectivamente.

Pham (*System Software Reliability,* Springer, 2006), Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, 2a. ed., McGraw-Hill, 2004) y Peled (*Software Reliability Methods*, Springer, 2001) proporcionan guías prácticas que describen los métodos para medir y analizar la confiabilidad del software.

Vincoli (*Basic Guide to System Safety*, Wiley 2006), Dhillon (*Engineering Safety*, World Scientific Publishing Co., Inc., 2003), Hermann (*Software Safety and Reliability*, Wiley-IEEE Computer Society Press, 2000), Storey (*Safety-Critical Computer Systems*, Addison-Wesley, 1996) y Leveson [Lev95] aportan los análisis más exhaustivos que se hayan publicado hasta la fecha acerca de la seguridad del software y del sistema. Además, Van

der Meulen (*Definitions for Hardware and Software Safety Engineers*, Springer-Verlag, 2000) ofrece un compendio completo de conceptos y términos importantes para la confiabilidad y la seguridad; Gartner (*Testing Safety-Related Software*, Springer-Verlag, 1999) ofrece una guía especializada para probar sistemas cuya seguridad sea crítica; Friedman y Voas (*Software Assesment: Reliability Safety and Testability*, Wiley, 1995) proveen modelos útiles para evaluar la confiabilidad y la seguridad. Ericson (*Hazard Analysis Techniques for System Safety*, Wiley, 2005) estudia el dominio cada vez más importante del análisis de los peligros.

En internet, hay una amplia variedad de fuentes de información sobre el aseguramiento de la calidad del software y otros temas relacionados. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, existe una lista actualizada de referencias existentes en la red mundial que son relevantes para el ACS.

CAPÍTULO

# 17

# ESTRATEGIAS DE PRUEBA DE SOFTWARE

CONCEPTOS CLAVE



Una estrategia de prueba de software proporciona una guía que describe los pasos que deben realizarse como parte de la prueba, cuándo se planean y se llevan a cabo dichos pasos, y cuánto esfuerzo, tiempo y recursos se requerirán. Por tanto, cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de casos de prueba, la ejecución de la prueba y la recolección y evaluación de los resultados.

Una estrategia de prueba de software debe ser suficientemente flexible para promover un uso personalizado de la prueba. Al mismo tiempo, debe ser suficientemente rígida para alentar la planificación razonable y el seguimiento de la gestión conforme avanza el proyecto. Shooman [Sho83] analiza estos temas:

> En muchas formas, la prueba es un proceso de individualización, y el número de tipos diferentes de pruebas varía tanto como los diferentes acercamientos para su desarrollo. Durante muchos años, la única defensa contra los errores de programación fue el diseño cuidadoso y la inteligencia natural del programador. Ahora estamos en una era en la que modernas técnicas de diseño (y revisiones técnicas) ayudan a reducir el número de errores iniciales que son inherentes al código. De igual modo, diferentes métodos de prueba comienzan a agruparse en métodos y filosofías distintos.

Estos "enfoques y filosofías" a los que denomino *estrategias* son el tema que se presenta en este capítulo. En los capítulos 18, 19 y 20 se exponen los métodos y técnicas de prueba que permiten desarrollar la estrategia.

## UNA MIRADA RÁPIDA

**¿Qué es?** El software se prueba para descubrir errores que se cometieron de manera inadvertida conforme se diseñó y construyó. Pero, ¿cómo se realizan las pruebas? ¿Debe realizarse un plan formal para las mismas? ¿Debe probarse el programa completo, como un todo, o aplicar pruebas sólo sobre una pequeña parte de él? ¿Debe volverse a aplicar las pruebas que ya se realizaron mientras se agregan nuevos componentes a un sistema grande? ¿Cuándo debe involucrarse al cliente? Éstas y muchas otras preguntas se responden cuando se desarrolla una estrategia de prueba de software.

**¿Quién lo hace?** El gerente de proyecto, los ingenieros de software y los especialistas en pruebas desarrollan una estrategia para probar el software.

**¿Por qué es importante?** Con frecuencia, la prueba requiere más esfuerzo que cualquiera otra acción de ingeniería del software. Si se realiza sin orden, se desperdicia tiempo, se emplea esfuerzo innecesario y, todavía peor, es posible que algunos errores pasen desapercibidos. Por tanto, parecería razonable establecer una estrategia sistemática para probar el software.

**¿Cuáles son los pasos?** La prueba comienza "por lo pequeño" y avanza "hacia lo grande". Es decir que las primeras etapas de prueba se enfocan sobre un solo componente o un pequeño grupo de componentes relacionados y se aplican pruebas para descubrir errores en los datos y en la lógica de procesamiento que se encapsularon en los componentes. Después de probar éstos, deben integrarse hasta que se construya el sistema completo. En este punto, se ejecuta una serie de pruebas de orden superior para descubrir errores en la satisfacción de los requerimientos del cliente. Conforme se descubren, los errores deben diagnosticarse y corregirse usando un proceso que se llama depuración.

**¿Cuál es el producto final?** Una *Especificación pruebas* documenta la forma en la que el equipo de software prepara la prueba al definir un plan que describe una estrategia global y un procedimiento con pasos de prueba específicos y los tipos de pruebas que se realizarán.

**¿Cómo me aseguro de que lo hice bien?** Al revisar la *Especificación pruebas* antes de realizar las pruebas, es posible valorar si están completos los casos de prueba y las tareas de la misma. Un plan de prueba y procedimientos efectivos conducirán a la construcción ordenada del software y al descubrimiento de errores en cada etapa del proceso de construcción.

## 17.1 Un enfoque estratégico para la prueba de software

La prueba es un conjunto de actividades que pueden planearse por adelantado y realizarse de manera sistemática. Por esta razón, durante el proceso de software, debe definirse una plantilla para la prueba del software: un conjunto de pasos que incluyen métodos de prueba y técnicas de diseño de casos de prueba específicos.

En la literatura sobre el tema, se han propuesto algunas estrategias de prueba de software. Todas proporcionan una plantilla para la prueba y tienen las siguientes características genéricas:

- Para realizar una prueba efectiva, debe realizar revisiones técnicas efectivas (capítulo 15). Al hacerlo, eliminará muchos errores antes de comenzar la prueba.
- La prueba comienza en los componentes y opera "hacia afuera", hacia la integración de todo el sistema de cómputo.
- Diferentes técnicas de prueba son adecuadas para distintos enfoques de ingeniería de software y en diferentes momentos en el tiempo.
- Las pruebas las realiza el desarrollador del software y (para proyectos grandes) un grupo de prueba independiente.
- Prueba y depuración son actividades diferentes, pero la depuración debe incluirse en cualquier estrategia de prueba.

**WebRef**

En **www.mtsu.edu/~storm** pueden encontrarse útiles recursos para la prueba de software.

Una estrategia para la prueba de software debe incluir pruebas de bajo nivel, que son necesarias para verificar que un pequeño segmento de código fuente se implementó correctamente, así como pruebas de alto nivel, que validan las principales funciones del sistema a partir de los requerimientos del cliente. Una estrategia debe proporcionar una guía para el profesional y un conjunto de guías para el jefe de proyecto. Puesto que los pasos de la estrategia de prueba ocurren cuando comienza a aumentar la presión por las fechas límite, el avance debe ser medible y los problemas deben salir a la superficie tan pronto como sea posible.

### 17.1.1 Verificación y validación

**Cita:**

"Probar es la parte inevitable de cualquier esfuerzo responsable por desarrollar un sistema de software".

**William Howden**

La prueba de software es un elemento de un tema más amplio que usualmente se conoce como verificación y validación (V&V). La *verificación* se refiere al conjunto de tareas que garantizan que el software implementa correctamente una función específica. La *validación* es un conjunto diferente de tareas que aseguran que el software que se construye sigue los requerimientos del cliente. Boehm [Boe81] afirma esto de esta forma:

> Verificación: "¿Construimos el producto correctamente?"
>
> Validación: "¿Construimos el producto correcto?"

La definición de V&V abarca muchas actividades de aseguramiento de calidad del software (capítulo 16).[1]

La verificación y la validación incluyen un amplio arreglo de actividades SQA: revisiones técnicas, auditorías de calidad y configuración, monitoreo de rendimiento, simulación, estudio de factibilidad, revisión de documentación, revisión de base de datos, análisis de algoritmos, pruebas de desarrollo, pruebas de usabilidad, pruebas de calificación, pruebas de aceptación y

1 Debe notarse que hay una fuerte divergencia de opinión acerca de qué tipos de pruebas constituyen la "validación". Algunas personas creen que *todas* las pruebas sirven para la verificación y que la validación se lleva a cabo cuando los requerimientos se revisan y aprueban, y, más tarde, por el usuario, cuando el sistema resulta operativo. Otras personas ven las pruebas de unidad y de integración (secciones 17.3.1 y 17.3.2) como verificación y las de orden superior (secciones 17.6 y 17.7) como validación.

pruebas de instalación. Aunque las pruebas juegan un papel extremadamente importante en V&V, también son necesarias muchas otras actividades.

CONSEJO

*Es un error pensar que las pruebas son una "red de seguridad" que atrapará todos los errores que ocurran como producto de deficientes prácticas de ingeniería de software. No lo hará. Enfatice la calidad y la detección de errores a lo largo del proceso de software.*

Las pruebas representan el último bastión desde donde puede valorarse la calidad y, de manera más pragmática, descubrirse errores. Pero las pruebas no deben verse como una red de seguridad. Como se dice: "no se puede probar la calidad. Si no está ahí antes de comenzar las pruebas, no estará cuando termine de probar". La calidad se incorpora en el software a lo largo de todo el proceso de ingeniería del software. La adecuada aplicación de métodos y herramientas, revisiones técnicas efectivas, y gestión y medición sólidas conducen a la calidad que se confirma durante las pruebas.

Miller [Mil77] relaciona la prueba del software con el aseguramiento de la calidad al afirmar que "la motivación subyacente de las pruebas de los programas es afirmar la claridad del software con métodos que puedan aplicarse de manera económica y efectiva a sistemas a gran y pequeña escala".

### 17.1.2 Organización de las pruebas del software

**Cita:**

"El optimismo es el riesgo ocupacional de la programación; la prueba es el tratamiento".

**Kent Beck**

En todo proyecto de software hay un conflicto inherente de intereses que ocurre conforme comienzan las pruebas. Hoy en día, a las personas que construyen el software se les pide probarlo. En sí, esto parece sencillo; después de todo, ¿quién conoce mejor el programa que sus desarrolladores? Por desgracia, estos mismos desarrolladores tienen mucho interés en demostrar que el programa está libre de errores, que funciona de acuerdo con los requerimientos del cliente y que se completará a tiempo y dentro del presupuesto. Cada uno de estos intereses tienen un efecto negativo sobre las pruebas más cuidadosas.

Desde un punto de vista psicológico, el análisis y diseño de software (junto con la codificación) son tareas constructivas. El ingeniero de software analiza, modela y luego crea un programa de computadora y su documentación. Como cualquier constructor, el ingeniero de software está orgulloso del edificio que construyó y ve con desconfianza a quien intente derrumbarlo. Cuando comienzan las pruebas, hay un sutil, pero definitivo, intento por "romper" lo que construyó el ingeniero de software. Desde el punto de vista del constructor, las pruebas pueden considerarse como (psicológicamente) destructivas. De modo que el constructor actuará con cuidado, y diseñará y ejecutará pruebas que demostrarán que el programa funciona, en lugar de descubrir errores. Desafortunadamente, los errores estarán presentes. Y si el ingeniero de software no los encuentra, ¡el cliente lo hará!

Con frecuencia, existen algunas malas interpretaciones que pueden inferirse de manera errónea a partir de la discusión anterior: 1) que el desarrollador de software no debe hacer pruebas en absoluto, 2) que el software debe "ponerse tras una pared" que lo separe de los extraños que lo probarán sin misericordia, 3) que quienes realicen las pruebas deben involucrarse con el proyecto sólo cuando los pasos de las pruebas estén por comenzar. Cada uno de estos enunciados es incorrecto.

El desarrollador de software siempre es responsable de probar las unidades individuales (componentes) del programa y de asegurarse de que cada una desempeña la función o muestra el comportamiento para el cual se diseñó. En muchos casos, el desarrollador también realiza pruebas de integración, una etapa en las pruebas que conduce a la construcción (y prueba) de la arquitectura completa del software. Sólo después de que la arquitectura de software está completa se involucra un grupo de prueba independiente (GPI).

PUNTO CLAVE

Un grupo de prueba independiente no tiene el "conflicto de intereses" que pueden experimentar los constructores del software.

El papel de un *grupo de prueba independiente* (GPI) es remover los problemas inherentes que están asociados con dejar al constructor probar lo que construyó. Las pruebas independientes remueven el conflicto de intereses que de otro modo puede estar presente. Después de todo, al personal del GPI se le paga por encontrar errores.

Sin embargo, el desarrollador no da el software al GPI y se retira. Él y el GPI trabajan de manera cercana a lo largo del proyecto de software para garantizar que se realizarán pruebas ex-

haustivas. Mientras se realizan éstas, el desarrollador debe estar disponible para corregir los errores que se descubran.

El GPI es parte del equipo de proyecto de desarrollo del software pues se involucra durante el análisis y el diseño, y sigue involucrado (mediante planificación y especificación de procedimientos de prueba) a lo largo de un proyecto grande. No obstante, en muchos casos, el GPI reporta a la organización de aseguramiento de calidad del software, y por tanto logra un grado de independencia que no puede existir si fuese parte de la organización de ingeniería del software.

**Cita:**

"El primer error que comete la gente es creer que el equipo de prueba es responsable de asegurar la calidad."

**Brian Marick**

### 17.1.3 Estrategia de prueba del software. Visión general

El proceso de software puede verse como la espiral que se ilustra en la figura 17.1. Inicialmente, la ingeniería de sistemas define el papel del software y conduce al análisis de los requerimientos del mismo, donde se establecen los criterios de dominio, función, comportamiento, desempeño, restricciones y validación de información para el software. Al avanzar hacia adentro a lo largo de la espiral, se llega al diseño y finalmente a la codificación. Para desarrollar software de computadoras, se avanza en espiral hacia adentro (contra las manecillas del reloj) a lo largo de una línea que reduce el nivel de abstracción en cada vuelta.

**? ¿Cuál es la estrategia global para la prueba del software?**

Una estrategia para probar el software también puede verse en el contexto de la espiral (figura 17.1). La *prueba de unidad* comienza en el vértice de la espiral y se concentra en cada unidad (por ejemplo, componente, clase o un objeto de contenido de una *webapp*) del software como se implementó en el código fuente. La prueba avanza al moverse hacia afuera a lo largo de la espiral, hacia la *prueba de integración*, donde el enfoque se centra en el diseño y la construcción de la arquitectura del software. Al dar otra vuelta hacia afuera de la espiral, se encuentra la *prueba de validación*, donde los requerimientos establecidos como parte de su modelado se validan confrontándose con el software que se construyó. Finalmente, se llega a la *prueba del sistema*, donde el software y otros elementos del sistema se prueban como un todo. Para probar el software de cómputo, se avanza en espiral hacia afuera en dirección de las manecillas del reloj a lo largo de líneas que ensanchan el alcance de las pruebas con cada vuelta.

**WebRef**

Quienes prueban software pueden encontrar recursos útiles en **www.SQAtester.com**

Al considerar el proceso desde un punto de vista procedural, las pruebas dentro del contexto de la ingeniería del software en realidad son una serie de cuatro pasos que se implementan de manera secuencial. Éstos se muestran en la figura 17.2. Inicialmente, las pruebas se enfocan en cada componente de manera individual, lo que garantiza que funcionan adecuadamente como unidad. De ahí el nombre de *prueba de unidad*. Esta prueba utiliza mucho de las técnicas de prueba que ejercitan rutas específicas en una estructura de control de componentes para asegurar una cobertura completa y la máxima detección de errores. A continuación, los componentes deben ensamblarse o integrarse para formar el paquete de software completo. La *prueba de integración* aborda los conflictos asociados con los problemas duales de verificación y construcción de programas. Durante la integración, se usan más las técnicas de diseño de casos de

**FIGURA 17.1**
**Estrategia de pruebas**

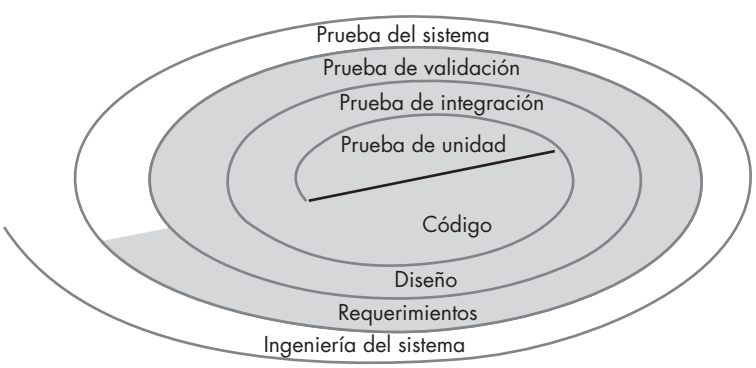
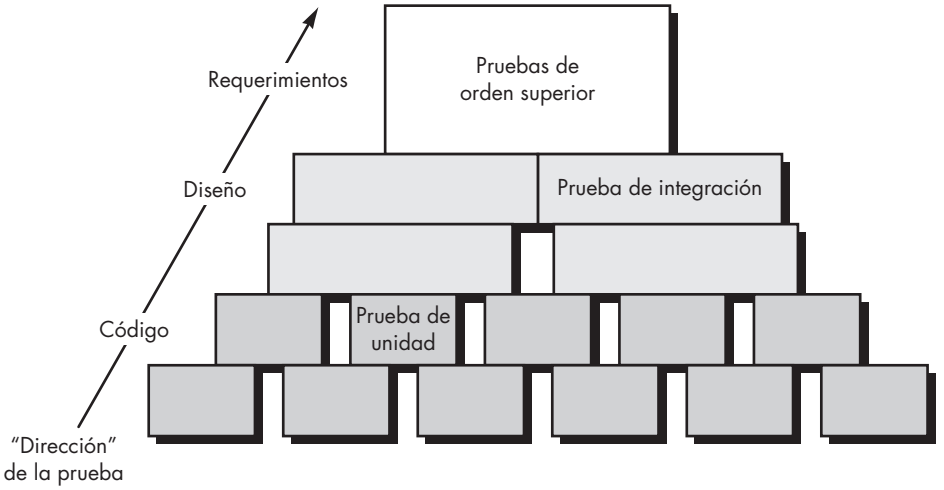


**FIGURA 17.2**
**Pasos de la prueba del software**




prueba que se enfocan en entradas y salidas, aunque también pueden usarse técnicas que ejercitan rutas de programa específicas para asegurar la cobertura de las principales rutas de control. Después de integrar (construir) el software, se realiza una serie de *pruebas de orden superior*. Deben evaluarse criterios de validación (establecidos durante el análisis de requerimientos). La *prueba de validación* proporciona la garantía final de que el software cumple con todos los requerimientos informativos, funcionales, de comportamiento y de rendimiento.

El último paso de la prueba de orden superior cae fuera de las fronteras de la ingeniería de software y en el contexto más amplio de la ingeniería de sistemas de cómputo. El software, una vez validado, debe combinarse con otros elementos del sistema (por ejemplo, hardware, personal, bases de datos). La *prueba del sistema* verifica que todos los elementos se mezclan de manera adecuada y que se logra el funcionamiento/rendimiento global del sistema.

## CASASEGURA

### *Preparación para la prueba*

**La escena:** Oficina de Doug Miller, mientras continúa el diseño en el nivel de componentes y comienza la construcción de algunos de ellos.

**Participantes:** Doug Miller, jefe de ingeniería de software; Vinod, Jamie, Ed y Shakira, miembros del equipo de ingeniería de software de CasaSegura.

**La conversación:**

**Doug:** Me parece que no hemos dedicado suficiente tiempo para hablar de las pruebas.

**Vinod:** Cierto, pero todos hemos estado un poco ocupados. Y además hemos pensado en ello... en realidad, hemos hecho más que pensar.

**Doug (sonríe):** Lo sé... tenemos exceso de trabajo, pero todavía debemos pensar en las cosas importantes.

**Shakira:** Me gusta la idea de diseñar pruebas de unidad antes de comenzar a codificar cualquiera de mis componentes, así que eso es lo que he intentado hacer. Tengo un archivo de pruebas bastante grande para aplicar cuando codifique mis componentes por completo.

**Doug:** Ése es un concepto de programación extrema [proceso de desarrollo de software ágil, véase el capítulo 3], ¿o no?

**Ed:** Lo es. Aun cuando no usamos programación extrema *per se*, decidimos que sería buena idea diseñar pruebas de unidad antes de construir el componente; el diseño nos dará la información que necesitamos.

**Jamie:** Yo he hecho lo mismo.

**Vinod:** Y yo tomé el papel de integrador, así que cada vez que uno de los muchachos me pase un componente, lo integraré y correré una serie de pruebas de regresión sobre el programa parcialmente integrado. He trabajado para diseñar un conjunto de pruebas adecuadas para cada función en el sistema.

**Doug (a Vinod):** ¿Con qué frecuencia corres las pruebas?

**Vinod:** Todos los días... hasta que el sistema esté integrado... bueno, quiero decir hasta que esté integrado el incremento de software que planeamos entregar.

**Doug:** ¡Muchachos, van adelante de mí!

**Vinod (ríe):** La anticipación lo es todo en el negocio del software, jefe.

### 17.1.4 Criterios para completar las pruebas

Cada vez que se analiza la prueba del software, surge una pregunta clásica: "¿cuándo terminan las pruebas?, ¿cómo se sabe que se ha probado lo suficiente?". Lamentablemente, no hay una respuesta definitiva a esta pregunta, pero existen algunas respuestas pragmáticas e intentos tempranos a manera de guía empírica.

**¿Cuándo terminan las pruebas?**

Una respuesta a la pregunta es: "nunca se termina de probar; la carga simplemente pasa de usted (el ingeniero de software) al usuario final". Cada vez que el usuario ejecuta un programa de cómputo, el programa se pone a prueba. Este instructivo hecho subraya la importancia de otras actividades a fin de garantizar la calidad del software. Otra respuesta (un tanto cínica, mas no obstante precisa) es: "las pruebas terminan cuando se agota el tiempo o el dinero".

Aunque algunos profesionales usarían estas respuestas, se necesitan criterios más rigurosos para determinar cuándo se han realizado suficientes pruebas. El enfoque de *ingeniería de software de salas limpias* (capítulo 21) sugiere el uso de técnicas estadísticas [Kel00] que ejecutan una serie de pruebas derivadas de una muestra estadística de todas las posibles ejecuciones de programa por parte de todos los usuarios de una población objetivo. Otros (por ejemplo, [Sin99]) abogan por el uso del modelado estadístico y la teoría de confiabilidad del software para predecir cuándo están completas las pruebas.

**WebRef**

Un glosario amplio de términos de pruebas puede encontrarse en el sitio **www.testingstandards.co.uk/living_glossary.htm**

Al coleccionar estadísticas durante las pruebas del software y usar los modelos existentes de confiabilidad del mismo, es posible desarrollar lineamientos significativos para responder la pregunta: "¿cuándo terminan las pruebas?". Hay poco debate acerca de que todavía queda mucho trabajo por hacer antes de poder establecer reglas cuantitativas para las pruebas, pero los acercamientos empíricos que existen en la actualidad son considerablemente mejores que la intuición pura.

## 17.2 Aspectos estratégicos

Más adelante en este capítulo, se presenta una estrategia sistemática para probar el software. Pero incluso la mejor estrategia fracasará si no se aborda una serie de aspectos decisivos. Tom Gilb [Gil95] arguye que una estrategia de software triunfará cuando quienes prueban el software:

**¿Qué lineamientos conducen a una exitosa estrategia de prueba del software?**

*Especifican los requerimientos del producto en forma cuantificable mucho antes de comenzar con las pruebas.* Aunque el objetivo predominante de una prueba es encontrar errores, una buena estrategia de prueba también valora otras características de la calidad, como la portabilidad, el mantenimiento y la facilidad de uso (capítulo 14). Esto debe especificarse en una forma medible, de modo que los resultados de las pruebas no sean ambiguos.

*Establecen de manera explícita los objetivos de las pruebas.* Los objetivos específicos de las pruebas deben enunciarse en términos medibles. Por ejemplo, la efectividad de las pruebas, su cobertura, el tiempo medio antes de aparecer una falla, el costo por descubrir y corregir defectos, la densidad de defectos restantes o la frecuencia de ocurrencia, y las horas de trabajo de prueba deben enunciarse dentro del plan de la prueba.

**WebRef**

Una excelente lista de recursos de prueba puede encontrarse en el sitio **www.io.com/~wazmo/qa**

*Entienden a los usuarios del software y desarrollan un perfil para cada categoría de usuario*. Los casos de uso que describen el escenario de interacción para cada clase de usuario pueden reducir el esfuerzo de prueba global al enfocar las pruebas en el uso real del producto.

*Desarrollan un plan de prueba que enfatice "pruebas de ciclo rápido"*. Gilb [Gil95] recomienda que un equipo de software "aprenda a probar en ciclos rápidos (2 por ciento del esfuerzo del proyecto) de cliente-utilidad al menos la 'comprobabilidad' en campo, los incrementos de funcionalidad y/o la mejora de la calidad". La retroalimentación generada a partir de estas pruebas de ciclo rápido puede usarse para controlar niveles de calidad y las correspondientes estrategias de prueba.

*Construyen software "robusto" que esté diseñado para probarse a sí mismo.* El software debe diseñarse en forma que use técnicas antierrores (sección 17.3.1), es decir, el software debe poder diagnosticar ciertas clases de errores. Además, el diseño debe incluir pruebas automatizadas y pruebas de regresión.

*Usan revisiones técnicas efectivas como filtro previo a las pruebas.* Las revisiones técnicas (capítulo 15) pueden ser tan efectivas como probar para descubrir errores. Por esta razón, las revisiones pueden reducir la cantidad del esfuerzo de pruebas que se requieren para producir software de alta calidad.

*Realizan revisiones técnicas para valorar la estrategia de prueba y los casos de prueba.* Las revisiones de prueba pueden descubrir inconsistencias, omisiones y errores evidentes en el abordaje de las pruebas. Esto ahorra tiempo y también mejora la calidad del producto.

*Desarrollan un enfoque de mejora continuo para el proceso de prueba.* La estrategia de pruebas debe medirse. Las métricas recopiladas durante las pruebas deben usarse como parte de un enfoque de control de proceso estadístico para la prueba del software.

**Cita:**

"Probar sólo los **requerimientos** del usuario final es como inspeccionar un edificio con base en el trabajo realizado por el decorador de interiores a costa de cimientos, vigas y plomería."

**Boris Beizer**

## 17.3 Estrategias de prueba para software convencional[2]

Existen muchas estrategias que pueden usarse para probar el software. En un extremo, puede esperarse hasta que el sistema esté completamente construido y luego realizar las pruebas sobre el sistema total, con la esperanza de encontrar errores. Este enfoque, aunque atractivo, simplemente no funciona. Dará como resultado software defectuoso que desilusionará a todos los participantes. En el otro extremo, podrían realizarse pruebas diariamente, siempre que se construya alguna parte del sistema. Este enfoque, aunque menos atractivo para muchos, puede ser muy efectivo. Por desgracia, algunos desarrolladores de software son reacios a usarlo. ¿Qué hacer?

Una estrategia de prueba que eligen la mayoría de los equipos de software se coloca entre los dos extremos. Toma una visión incremental de las pruebas, comenzando con la de unidades de programa individuales, avanza hacia pruebas diseñadas para facilitar la integración de las unidades y culmina con pruebas que ejercitan el sistema construido. Cada una de estas clases de pruebas se describe en las secciones que siguen.

### 17.3.1 Prueba de unidad

La *prueba de unidad* enfoca los esfuerzos de verificación en la unidad más pequeña del diseño de software: el componente o módulo de software. Al usar la descripción del diseño de componente como guía, las rutas de control importantes se prueban para descubrir errores dentro de la frontera del módulo. La relativa complejidad de las pruebas y los errores que descubren están limitados por el ámbito restringido que se establece para la prueba de unidad. Las pruebas de unidad se enfocan en la lógica de procesamiento interno y de las estructuras de datos dentro de las fronteras de un componente. Este tipo de pruebas puede realizarse en paralelo para múltiples componentes.

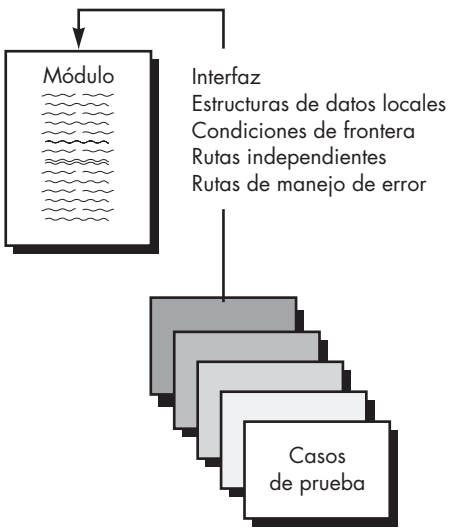
CONSEJO

*No es mala idea diseñar casos de prueba de unidad antes de desarrollar el código para un componente. Eso ayuda a garantizar que se desarrollará un código que pasará las pruebas.*

**Consideraciones de las pruebas de unidad.** Las pruebas de unidad se ilustran de manera esquemática en la figura 17.3. La interfaz del módulo se prueba para garantizar que la información fluya de manera adecuada hacia y desde la unidad de software que se está probando. Las estructuras de datos locales se examinan para asegurar que los datos almacenados temporal-

2 A lo largo de este libro, se usan los términos *software convencional* o *software tradicional* para referirse a arquitecturas de software jerárquica común, o de "llamar y regresar", que con frecuencia se encuentran en una variedad de dominios de aplicación. Las arquitecturas de software tradicional *no* son orientadas a objetos y no abarcan *webapps*.

**FIGURA 17.3**
**Prueba de unidad**




mente mantienen su integridad durante todos los pasos en la ejecución de un algoritmo. Todas las rutas independientes a través de la estructura de control se ejercitan para asegurar que todos los estatutos en un módulo se ejecuten al menos una vez. Las condiciones de frontera se prueban para asegurar que el módulo opera adecuadamente en las fronteras establecidas para limitar o restringir el procesamiento. Y, finalmente, se ponen a prueba todas las rutas para el manejo de errores.

El flujo de datos a través de la interfaz de un componente se prueba antes de iniciar cualquiera otra prueba. Si los datos no entran y salen de manera adecuada, todas las demás pruebas son irrelevantes. Además, deben ejercitarse las estructuras de datos locales y averiguarse (si es posible) el impacto local sobre los datos globales durante las pruebas de unidad.

**? ¿Qué errores se encuentran comúnmente durante las pruebas de unidad?**

La prueba selectiva de las rutas de ejecución es una tarea esencial durante la prueba de unidad. Los casos de prueba deben diseñarse para descubrir errores debidos a cálculos erróneos, comparaciones incorrectas o flujo de control inadecuado.

Las pruebas de frontera son una de las tareas de prueba de unidad más importantes. Con frecuencia, el software falla en sus fronteras. Es decir: con frecuencia los errores ocurren cuando se procesa el *en*ésimo elemento de un arreglo *ene*-dimensional, cuando se invoca la *en*ésima repetición de un bucle con *n* pasadas, cuando se encuentra el valor máximo o mínimo permisible. Es muy probable que los casos de prueba que ejercitan la estructura de datos, el flujo de control y los valores de datos justo abajo y arriba de máximos y mínimos descubran errores.

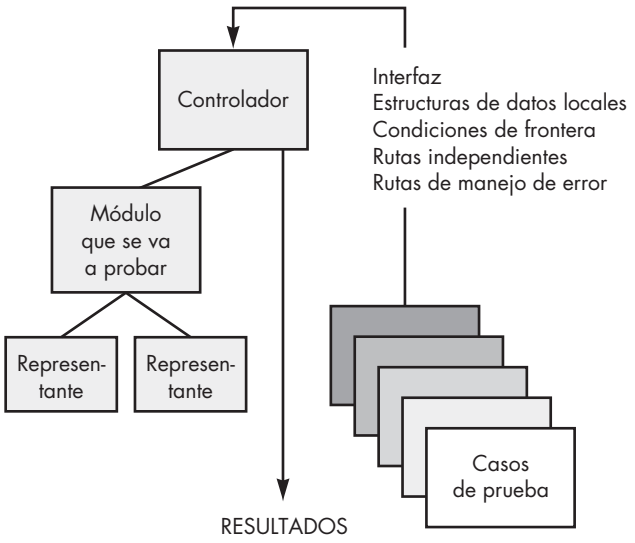
**WebRef**
Puede encontrar información útil acerca de una gran variedad de artículos y recursos para "prueba ágil" en **testing.com/agile**

Un buen diseño anticipa las condiciones de error y establece rutas de manejo de errores para enrutar o terminar limpiamente el procesamiento cuando ocurre un error. Yourdon [You75] llama a este enfoque *antierrores*. Desafortunadamente, hay una tendencia a incorporar el manejo de errores en el software y luego nunca probarlo. Una historia verídica puede servir como ilustración:

> Un sistema de diseño asistido por computadora se desarrolló bajo contrato. En un módulo de procesamiento de transacción, un bromista colocó el siguiente mensaje de manejo de error después de una serie de pruebas condicionales que invocaban varias ramas de flujo de control: ¡ERROR! NO HAY FORMA DE QUE PUEDA LLEGAR AQUÍ. ¡Este "mensaje de error" lo descubrió un cliente durante el entrenamiento para usuarios!

Entre los potenciales errores que deben ponerse a prueba cuando se evalúa el manejo de errores están: 1) la descripción de error ininteligible, 2) el error indicado no corresponde con el error que se encuentra, 3) la condición del error causa la intervención del sistema antes de manejar el

**FIGURA 17.4**
**Entorno de prueba de unidad**




*Asegúrese de diseñar pruebas para ejecutar cada ruta de manejo de error. Si no lo hace, la ruta puede fallar cuando se invoque, lo que agrava una situación de por sí peligrosa.*

error, 4) el procesamiento excepción-condición es incorrecto y 5) la descripción del error no proporciona suficiente información para auxiliar en la localización de la causa del error.

**Procedimientos de prueba de unidad.** Las pruebas de unidad por lo general se consideran como adjuntas al paso de codificación. El diseño de las pruebas de unidad puede ocurrir antes de comenzar la codificación o después de generar el código fuente. La revisión de la información del diseño proporciona una guía para establecer casos de prueba que es probable que descubran errores en cada una de las categorías analizadas anteriormente. Cada caso de prueba debe acoplarse con un conjunto de resultados esperados.

Puesto que un componente no es un programa independiente, con frecuencia debe desarrollarse software controlador y/o de resguardo para cada prueba de unidad. En la figura 17.4 se ilustran los entornos de prueba de unidad. En la mayoría de las aplicaciones, un *controlador* no es más que un "programa principal" que acepta datos de caso de prueba, pasa tales datos al componente (que va a ponerse a prueba) e imprime resultados relevantes. Los *representantes* (en inglés *stubs)* sirven para sustituir módulos que están subordinados al (invocados por el) componente que se va a probar. Un representante o "subprograma tonto" usa la interfaz de módulo subordinado, puede realizar mínima manipulación de datos, imprimir verificación de entradas y regresar el control al módulo sobre el que se realiza la prueba.

*Existen algunas situaciones donde no se tienen los recursos para realizar una prueba de unidad amplia. Seleccione los módulos cruciales o complejos y aplique sólo en ellos las pruebas de unidad.*

Los controladores y representantes añaden una "sobrecarga" a las pruebas. Es decir: ambos son software que debe escribirse (el diseño formal usualmente no se aplica), pero que no se entrega con el producto de software final. Si los controladores y representantes se mantienen simples, la sobrecarga real es relativamente baja. Por desgracia, muchos componentes no pueden tener prueba de unidad adecuada con un software de sobrecarga simple. En tales casos, la prueba completa puede posponerse hasta el paso de prueba de integración (donde también se usan controladores o representantes).

Las pruebas de unidad se simplifican cuando se diseña un componente con alta cohesión. Cuando un componente aborda una sola función, el número de casos de prueba se reduce y los errores pueden predecirse y descubrirse con mayor facilidad.

### 17.3.2 Pruebas de integración

Un neófito en el mundo del software podrá plantear una pregunta aparentemente legítima una vez que todos los módulos se hayan probado de manera individual: "si todos ellos funcionan

individualmente, ¿por qué dudan que funcionarán cuando se junten todos?". Desde luego, el problema es "juntarlos todos": conectarlos. Los datos pueden perderse a través de una interfaz; un componente puede tener un inadvertido efecto adverso sobre otro; las subfunciones, cuando se combinan, pueden no producir la función principal deseada; la imprecisión aceptable individualmente puede magnificarse a niveles inaceptables; las estructuras de datos globales pueden presentar problemas. Lamentablemente, la lista sigue y sigue.

Las pruebas de integración son una técnica sistemática para construir la arquitectura del software mientras se llevan a cabo pruebas para descubrir errores asociados con la interfaz. El objetivo es tomar los componentes probados de manera individual y construir una estructura de programa que se haya dictado por diseño.


*Tomar el enfoque de "big bang" para la integración es una estrategia perezosa condenada al fracaso. Integre de manera incremental y pruebe conforme avance.*

Con frecuencia existe una tendencia a intentar la integración no incremental, es decir, a construir el programa usando un enfoque de *big bang*. Todos los componentes se combinan por adelantado. Todo el programa se prueba como un todo. ¡Y usualmente resulta el caos! Se descubre un conjunto de errores. La corrección se dificulta pues el aislamiento de las causas se complica por la vasta extensión de todo el programa. Una vez corregidos estos errores, otros nuevos aparecen y el proceso continúa en un bucle aparentemente interminable.

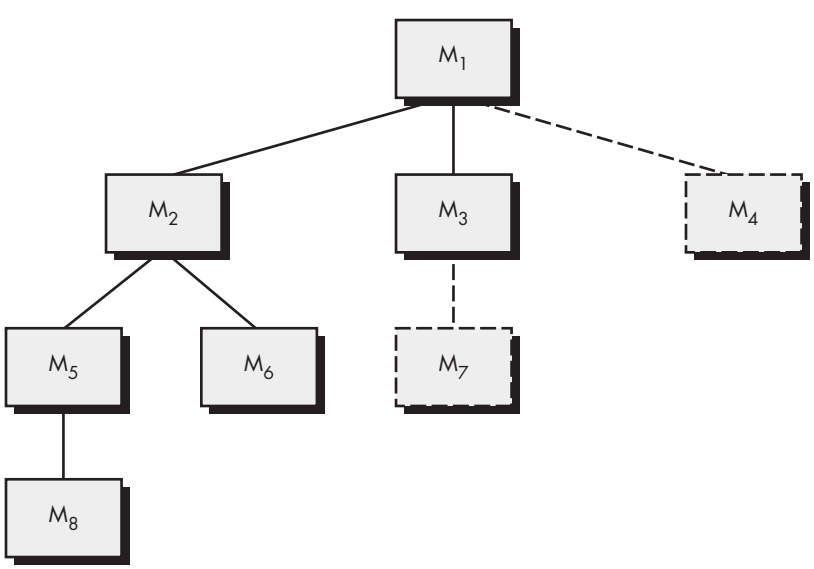
La integración incremental es la antítesis del enfoque *big bang*. El programa se construye y prueba en pequeños incrementos, donde los errores son más fáciles de aislar y corregir; las interfaces tienen más posibilidades de probarse por completo; y puede aplicarse un enfoque de prueba sistemático. En los siguientes párrafos se exponen algunas estrategias diferentes de integración incremental.


*Cuando desarrolle un calendario de proyecto, considere la forma en la que ocurrirá la integración, de modo que los componentes estén disponibles cuando se les necesite.*

**Integración descendente.** La *prueba de integración descendente* es un enfoque incremental a la construcción de la arquitectura de software. Los módulos se integran al moverse hacia abajo a través de la jerarquía de control, comenzando con el módulo de control principal (programa principal). Los módulos subordinados al módulo de control principal se incorporan en la estructura en una forma de primero en profundidad o primero en anchura.

Con referencia a la figura 17.5, la *integración primero en profundidad* integra todos los componentes sobre una ruta de control mayor de la estructura del programa. La selección de una ruta mayor es un tanto arbitraria y depende de las características específicas de la aplicación. Por ejemplo, al seleccionar la ruta de la izquierda, los componentes $M_1$, $M_2$, $M_5$ se integrarían primero. A continuación, $M_8$ o (si es necesario para el adecuado funcionamiento de $M_2$) se inte-

**FIGURA 17.5**
**Integración descendente**



graría $M_6$. Luego se construyen las rutas de control central y derecha. La *integración primero en anchura* incorpora todos los componentes directamente subordinados en cada nivel, y se mueve horizontalmente a través de la estructura. De la figura, los componentes $M_2$, $M_3$ y $M_4$ se integrarían primero. Le sigue el siguiente nivel de control, $M_5$, $M_6$, etc. El proceso de integración se realiza en una serie de cinco pasos:

**¿Cuáles son los pasos para la integración descendente?**

1. El módulo de control principal se usa como un controlador de prueba y los representantes (*stubs*) se sustituyen con todos los componentes directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque de integración seleccionado (es decir, primero en profundidad o anchura), los representantes subordinados se sustituyen uno a la vez con componentes reales.
3. Las pruebas se llevan a cabo conforme se integra cada componente.
4. Al completar cada conjunto de pruebas, otro representante se sustituye con el componente real.
5. Las pruebas de regresión (que se analizan más adelante en esta sección) pueden realizarse para asegurar que no se introdujeron nuevos errores.

El proceso continúa desde el paso 2 hasta que se construye todo la estructura del programa.

La estrategia de integración descendente verifica los principales puntos de control o de decisión al principio en el proceso de prueba. En una estructura de programa "bien factorizada", la toma de decisiones ocurre en niveles superiores en la jerarquía y, por tanto, se encuentra primero. Si existen grandes problemas de control, el reconocimiento temprano es esencial. Si se selecciona la integración primero en profundidad, es posible implementar y demostrar un funcionamiento completo del software. La demostración temprana de la capacidad funcional es un constructor de confianza para todos los participantes.

**¿Qué problemas pueden encontrarse cuando se elige la integración descendente?**

Pareciera que la estrategia descendente no tiene complicaciones, pero, en la práctica, pueden surgir problemas logísticos. El más común de éstos ocurre cuando se requiere procesamiento en niveles bajos en la jerarquía a fin de probar de manera adecuada los niveles superiores. Los representantes (*stubs*) sustituyen los módulos de bajo nivel al comienzo de la prueba descendente; por tanto, ningún dato significativo puede fluir hacia arriba en la estructura del programa. A la persona que realiza la prueba le quedan tres opciones: 1) demorar muchas pruebas hasta que los representantes se sustituyan con módulos reales, 2) desarrollar resguardos que realicen funciones limitadas que simulen al módulo real o 3) integrar el software desde el fondo de la jerarquía y hacia arriba.

El primer enfoque (demorar las pruebas hasta que los representantes se sustituyan con módulos reales) puede hacerle perder algo de control sobre la correspondencia entre pruebas específicas y la incorporación de módulos específicos. Esto puede conducir a dificultades para determinar la causa de los errores y tiende a violar la naturaleza enormemente restrictiva del enfoque descendente. El segundo enfoque vale la pena, pero puede conducir a una sobrecarga significativa conforme los representantes se vuelven cada vez más complejos. El tercero, llamado integración ascendente, se analiza en los siguientes párrafos.

**Integración ascendente.** La *prueba de integración ascendente*, como su nombre implica, comienza la construcción y la prueba con *módulos atómicos* (es decir, componentes en los niveles inferiores dentro de la estructura del programa). Puesto que los componentes se integran de abajo hacia arriba, la funcionalidad que proporcionan los componentes subordinados en determinado nivel siempre está disponible y se elimina la necesidad de representantes (*stubs*). Una estrategia de integración ascendente puede implementarse con los siguientes pasos:

**¿Cuáles son los pasos para la integración ascendente?**

1. Los componentes en el nivel inferior se combinan en grupos (en ocasiones llamados *construcciones* o *builds)* que realizan una subfunción de software específica.
2. Se escribe un *controlador* (un programa de control para pruebas) a fin de coordinar la entrada y salida de casos de prueba.
3. Se prueba el grupo.
4. Los controladores se remueven y los grupos se combinan moviéndolos hacia arriba en la estructura del programa.

**PUNTO CLAVE**

La integración ascendente elimina la necesidad de **representantes (*stubs*)** complejos.

La integración sigue el patrón que se ilustra en la figura 17.6. Los componentes se combinan para formar los grupos 1, 2 y 3. Cada uno de ellos se prueba usando un controlador (que se muestra como un bloque rayado). Los componentes en los grupos 1 y 2 se subordinan a $M_a$. Los controladores $D_1$ y $D_2$ se remueven y los grupos se ponen en interfaz directamente con $M_a$. De igual modo, el controlador $D_3$ para el grupo 3 se remueve antes de la integración con el módulo $M_b$. Tanto $M_a$ como $M_b$ al final se integrarán con el componente $M_c$, y así sucesivamente.

Conforme la integración avanza hacia arriba, se reduce la necesidad de controladores de prueba separados. De hecho, si los dos niveles superiores del programa se integran de manera descendente, el número de controladores puede reducirse de manera sustancial y la integración de grupos se simplifica enormemente.

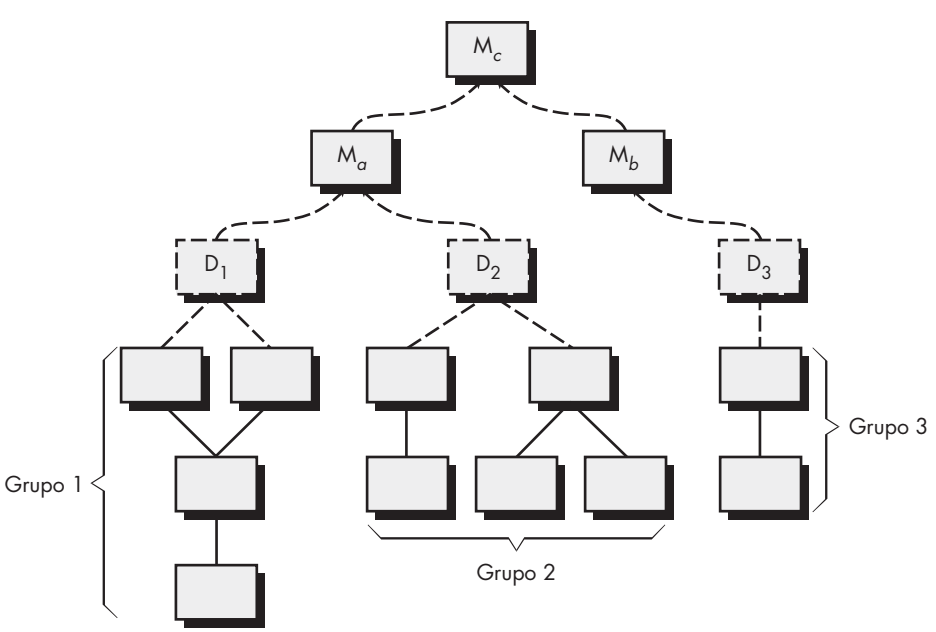
**CONSEJO**

*La prueba de regresión es una importante estrategia para reducir "efectos colaterales". Corra pruebas de regresión cada vez que se realiza 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 las pruebas de integración, el software cambia. Se establecen nuevas rutas de flujo de datos, ocurren nuevas operaciones de entrada/salida y se invoca nueva lógica de control. Dichos cambios pueden causar problemas con las funciones que anteriormente trabajaban sin fallas. En el contexto de una estrategia de prueba de integración, la *prueba de regresión* es la nueva ejecución de algún subconjunto de pruebas que ya se realizaron a fin de asegurar que los cambios no propagaron efectos colaterales no deseados.

En un contexto más amplio, las pruebas exitosas (de cualquier tipo) dan como resultado el descubrimiento de errores, y los errores deben corregirse. Siempre que se corrige el software, cambia algún aspecto de la configuración del software (el programa, su documentación o los datos que sustenta). Las pruebas de regresión ayudan a garantizar que los cambios (debidos

**FIGURA 17.6**
**Integración ascendente**



a pruebas o por otras razones) no introducen comportamiento no planeado o errores adicionales.

Las pruebas de regresión se pueden realizar manualmente, al volver a ejecutar un subconjunto de todos los casos de prueba o usando herramientas de captura/reproducción automatizadas. Las *herramientas de captura/reproducción* permiten al ingeniero de software capturar casos de prueba y resultados para una posterior reproducción y comparación. La *suite de prueba de regresión* (el subconjunto de pruebas que se va a ejecutar) contiene tres clases diferentes de casos de prueba:

- Una muestra representativa de pruebas que ejercitará todas las funciones de software.
- Pruebas adicionales que se enfocan en las funciones del software que probablemente resulten afectadas por el cambio.
- Pruebas que se enfocan en los componentes del software que cambiaron.

Conforme avanza la prueba de integración, el número de pruebas de regresión puede volverse muy grande. Por tanto, la suite de pruebas de regresión debe diseñarse para incluir solamente aquellas que aborden una o más clases de errores en cada una de las funciones del programa principal. Es impráctico e ineficiente volver a ejecutar toda prueba para cada función del programa cada vez que ocurre un cambio.

**Prueba de humo.** La *prueba de humo* es un enfoque de prueba de integración que se usa cuando se desarrolla software de producto. Se diseña como un mecanismo de ritmo para proyectos críticos en el tiempo, lo que permite al equipo del software valorar el proyecto de manera frecuente. En esencia, el enfoque de prueba de humo abarca las siguientes actividades:

PUNTO CLAVE

La prueba de humo puede caracterizarse como una estrategia de integración constante. El software se reconstruye (con el agregado de nuevos componentes) y se prueba cada día.

1. Los componentes de software traducidos en código se integran en una *construcción*. Una construcción incluye todos los archivos de datos, bibliotecas, módulos reutilizables y componentes sometidos a ingeniería que se requieren para implementar una o más funciones del producto.
2. Se diseña una serie de pruebas para exponer los errores que evitarán a la construcción realizar adecuadamente su función. La intención debe ser descubrir errores "paralizantes" que tengan la mayor probabilidad de retrasar el proyecto.
3. La construcción se integra con otras construcciones, y todo el producto (en su forma actual) se somete a prueba de humo diariamente. El enfoque de integración puede ser descendente o ascendente.

**Cita:**

"Trate a la construcción diaria como al latido del proyecto. Si no hay latido, el proyecto está muerto."

**Jim McCarthy**

La frecuencia diaria de las pruebas de todo el producto puede sorprender a algunos lectores. Sin embargo, las pruebas constantes brindan, tanto a gerentes como a profesionales, una valoración realista del progreso de la prueba de integración. McConnell [McC96] describe la prueba de humo de la forma siguiente:

> La prueba de humo debe ejercitar todo el sistema de extremo a extremo. No tiene que ser exhaustiva, pero debe poder exponer los problemas principales. La prueba de humo debe ser suficientemente profunda para que, si la construcción pasa, pueda suponer que es suficientemente estable para probarse con mayor profundidad.

La prueba de humo proporciona algunos beneficios cuando se aplica sobre proyectos de software complejos y cruciales en el tiempo:

**? ¿Qué beneficios pueden derivarse de las pruebas de humo?**

- *Se minimiza el riesgo de integración*. Puesto que las pruebas de humo se realizan diariamente, las incompatibilidades y otros errores paralizantes pueden descubrirse tempranamente, lo que reduce la probabilidad de impacto severo sobre el calendario cuando se descubren errores.

- *La calidad del producto final mejora.* Es probable que la prueba de humo descubra errores funcionales así como errores de diseño arquitectónico o en el componente debido a que el enfoque está orientado a la construcción (integración). Si tales errores se corrigen temprano, se tendrá una mejor calidad del producto.
- *El diagnóstico y la corrección de errores se simplifican.* Como todo enfoque de prueba de integración, es probable que los errores descubiertos durante la prueba de humo se asocien con "nuevos incrementos de software"; es decir, el software que se acaba de agregar a la(s) construcción(es) es causa probable de un error recientemente descubierto.
- *El progreso es más fácil de valorar.* Con cada día que transcurre, más software se integra y se demuestra que funciona. Esto incrementa la moral del equipo y brinda a los gerentes un buen indicio de que se está progresando.

**WebRef**

En **www.qalinks.com** pueden encontrarse enlaces a comentarios acerca de estrategias de pruebas.

**Opciones estratégicas.** Ha habido mucha discusión (por ejemplo, [Bei84]) acerca de las relativas ventajas y desventajas de las pruebas de integración descendente en comparación con las ascendentes. En general, las ventajas de una estrategia tienden a ser desventajas para la otra. La principal desventaja del enfoque descendente es la necesidad de representantes y las dificultades de prueba que pueden asociarse con ellos. Los problemas asociados con los representantes pueden compensarse con la ventaja de probar tempranamente las principales funciones de control. La principal desventaja de la integración ascendente es que "el programa como entidad no existe hasta que se agrega el último módulo" [Mye79]. Este inconveniente se atempera con la mayor facilidad en el diseño de casos de prueba y la falta de representantes.

La selección de una estrategia de integración depende de las características del software y, en ocasiones, del calendario del proyecto. En general, un enfoque combinado (a veces llamado *prueba sándwich*), que usa pruebas descendentes para niveles superiores de la estructura del programa acopladas con pruebas ascendentes para niveles subordinados, puede ser el mejor arreglo.

**? ¿Qué es un "módulo crítico" y por qué debe identificársele?**

Conforme se realiza la integración, quien efectúa la prueba debe identificar los módulos críticos. Un *módulo crítico* tiene una o más de las siguientes características: 1) aborda muchos requerimientos de software, 2) tiene un alto nivel de control (reside relativamente alto en la estructura del programa), 3) es complejo o proclive al error o 4) tiene requerimientos de rendimiento definidos. Los módulos críticos deben probarse tan pronto como sea posible. Además, las pruebas de regresión deben enfocarse en la función del módulo crítico.

**Productos de trabajo de las pruebas de integración.** Un plan global para integración del software y una descripción de las pruebas específicas se documentan en una *Especificación de pruebas.* Este producto de trabajo incorpora un plan de prueba y un procedimiento de prueba, y se vuelve parte de la configuración del software. La prueba se divide en fases y construcciones que abordan características del software funcionales y de comportamiento específicas. Por ejemplo, la prueba de integración para el sistema de seguridad *CasaSegura* puede dividirse en las siguientes fases de prueba:

- *Interacción con el usuario* (entrada y salida de comandos, representación de despliegue, procesamiento y representación de errores)
- *Procesamiento de sensores* (adquisición de salida de sensor, determinación de condiciones del sensor, acciones requeridas como consecuencia de las condiciones)
- *Funciones de comunicación* (capacidad para comunicarse con la estación de monitoreo central)
- *Procesamiento de alarma* (pruebas de acciones del software que ocurren cuando se encuentra una alarma)

Cada una de estas fases de la prueba de integración delinea una amplia categoría funcional dentro del software y por lo general puede relacionarse con un dominio específico dentro de la arquitectura del software. Por tanto, las construcciones de programas (grupos de módulos) se crean para corresponder a cada fase. Los siguientes criterios y pruebas correspondientes se aplican a todas las fases de prueba:

**? ¿Qué criterios deben usarse para diseñar pruebas de integración?**

*Integridad de interfaz*. Las interfaces internas y externas se prueban conforme cada módulo (o grupo) se incorpora en la estructura.

*Validez funcional*. Se realizan pruebas diseñadas para descubrir errores funcionales ocultos.

*Contenido de la información*. Se realizan pruebas diseñadas para descubrir errores ocultos asociados con las estructuras de datos locales o globales.

*Rendimiento*. Se realizan pruebas diseñadas para verificar los límites del rendimiento establecidos durante el diseño del software.

Como parte del plan de prueba, también se discute un calendario para la integración, el desarrollo de software de sobrecarga del sistema y temas relacionados. Se establecen las fechas de inicio y fin de cada fase y se definen "ventanas disponibles" para módulos de prueba de unidad. Una breve descripción del software de sobrecarga (representantes y controladores) se concentra en las características que pueden requerir de un esfuerzo especial. Finalmente, se describe el entorno y los recursos de la prueba. Configuraciones inusuales de hardware, simuladores peculiares y herramientas o técnicas de prueba especial son algunos de los muchos temas que también pueden analizarse.

A continuación se describe el procedimiento de prueba detallado que se requiere para lograr el plan de prueba. Se señala el orden de la integración y las pruebas correspondientes en cada paso de ésta. También se incluye una lista de todos los casos de prueba (anotados para referencia posterior) y los resultados esperados.

En un *Reporte de prueba*, que puede anexarse a la *Especificación pruebas* si se desea, se registra una historia de resultados, problemas o peculiaridades de prueba reales. La información contenida en esta sección puede ser vital durante el mantenimiento del software. También se presentan las referencias y apéndices apropiados.

Como todos los demás elementos de una configuración de software, el formato de la especificación pruebas puede adaptarse a las necesidades locales de una organización de ingeniería de software. Sin embargo, es importante señalar que una estrategia de integración (contenida en un plan de prueba) y los detalles de la prueba (descritos en un procedimiento de prueba) son ingredientes esenciales y deben aparecer.

## 17.4 Estrategias de prueba para software orientado a objeto[3]

Enunciado de manera simple, el objetivo de probar es encontrar el mayor número posible de errores con una cantidad manejable de esfuerzo aplicado durante un lapso realista. Aunque este objetivo fundamental se mantiene invariable para el software orientado a objeto, la naturaleza de este software cambia tanto la estrategia como las tácticas de la prueba (capítulo 19).

### 17.4.1 Prueba de unidad en el contexto OO

Cuando se considera software orientado a objeto, el concepto de unidad cambia. La encapsulación determina la definición de clases y objetos. Esto significa que cada clase y cada instancia de una clase empaqueta los atributos (datos) y las operaciones que manipulan estos datos. Por lo

3 En el apéndice 2 se presentan conceptos básicos orientados a objeto.

general, una clase encapsulada es el foco de la prueba de unidad. No obstante, las operaciones (métodos) dentro de la clase son las unidades comprobables más pequeñas. Puesto que una clase puede contener algunas operaciones diferentes, y una operación particular puede existir como parte de algunas clases diferentes, las tácticas aplicadas a la prueba de unidad deben cambiar.

Ya no es posible probar una sola operación en aislamiento (la visión convencional de la prueba de unidad) sino más bien como parte de una clase. Para ilustrarlo, considere una jerarquía de clase en la que una operación *X* se define para la superclase y la heredan algunas subclases. Cada subclase usa la operación *X*, pero se aplica dentro del contexto de los atributos y operaciones privados que se definieron para la subclase. Dado que el contexto en el que se usa la operación *X* varía sutilmente, es necesario probar la operación *X* en el contexto de cada una de las subclases. Esto significa que por lo general no es efectivo probar la operación *X* en forma aislada (el enfoque de prueba de unidad convencional) en el contexto orientado a objeto.

La prueba de clase para software OO es análoga a la prueba de módulo para software convencional. No es aconsejable probar operaciones en aislamiento.

La prueba de clase para software OO es el equivalente de la prueba de unidad para software convencional. A diferencia de la prueba de unidad del software convencional, que tiende a enfocarse sobre el detalle algorítmico de un módulo y en los datos que fluyen a través de la interfaz de módulo, la prueba de clase para software OO la dirigen las operaciones encapsuladas por la clase y el comportamiento de estado de ésta.

### 17.4.2 Prueba de integración en el contexto OO

Puesto que el software orientado a objeto no tiene una estructura de control jerárquico obvia, las estrategias tradicionales descendente y ascendente (sección 17.3.2) tienen poco significado. Además, con frecuencia es imposible integrar las operaciones una a la vez en una clase (el enfoque de integración incremental convencional) debido a las "interacciones directa e indirecta de los componentes que constituyen la clase" [Ber93].

Una importante estrategia para la prueba de integración del software OO es la prueba basada en hebra. Las hebras son conjuntos de clases que responden a una entrada o evento. Las pruebas basadas en uso se enfocan en clases que no colaboran fuertemente con otras clases.

Existen dos estrategias diferentes para la prueba de integración de los sistemas OO [Bin94b]. La primera, la *prueba basada en hebra*, integra el conjunto de clases requeridas para responder a una entrada o evento para el sistema. Cada hebra se integra y prueba de manera individual. La prueba de regresión se aplica para asegurar que no ocurran efectos colaterales. El segundo enfoque de integración, la *prueba basada en uso*, comienza la construcción del sistema al probar dichas clases (llamadas *clases independientes*) que usan muy pocas clases *servidor* (si es que usan alguna). Después de probar las clases independientes, se prueba la siguiente capa de clases, llamadas *dependientes*, que usan las clases independientes. Esta secuencia de probar capas de clases dependientes continúa hasta que se construye todo el sistema.

El uso de controladores y representantes también cambia cuando se realiza la prueba de integración de los sistemas OO. Los controladores pueden usarse para probar operaciones en el nivel más bajo, y para la prueba de todos los grupos de clases. También puede usarse un controlador para sustituir la interfaz de usuario, de modo que las pruebas de funcionalidad del sistema puedan realizarse antes de la implementación de la interfaz. Los representantes (*stubs*) pueden usarse en situaciones donde se requiere la colaboración entre clases pero donde una o más de las clases colaboradoras todavía no se implementan por completo.

La *prueba de grupo* es un paso en la prueba de integración del software OO. Aquí, un grupo de clases colaboradoras (determinadas al examinar el CRC y el modelo objeto relacional) se ejercita al diseñar casos de prueba que intentan descubrir errores en las colaboraciones.

## 17.5 Estrategias de prueba para *webapps*

La estrategia para probar *webapps* adopta los principios básicos para todas las pruebas de software y aplica una estrategia y tácticas que se usan para sistemas orientados a objetos. Los siguientes pasos resumen el enfoque:

**PUNTO CLAVE**

La estrategia global para probar *webapps* puede resumirse en los 10 pasos que se anotan aquí.

1. El modelo de contenido para la *webapp* se revisa para descubrir errores.
2. El modelo de interfaz se revisa para garantizar que todos los casos de uso pueden adecuarse.
3. El modelo de diseño para la *webapp* se revisa para descubrir errores de navegación.
4. La interfaz de usuario se prueba para descubrir errores en los mecanismos de presentación y/o navegación.
5. A cada componente funcional se le aplica una prueba de unidad.
6. Se prueba la navegación a lo largo de toda la arquitectura.
7. La *webapp* se implementa en varias configuraciones ambientales diferentes y se prueba en su compatibilidad con cada configuración.
8. Las pruebas de seguridad se realizan con la intención de explotar vulnerabilidades en la *webapp* o dentro de su ambiente.
9. Se realizan pruebas de rendimiento.
10. La *webapp* se prueba mediante una población de usuarios finales controlada y monitoreada. Los resultados de su interacción con el sistema se evalúan por errores de contenido y navegación, preocupaciones de facilidad de uso, preocupaciones de compatibilidad, así como confiabilidad y rendimiento de la *webapp*.

**WebRef**

En **www.stickyminds.com/testing.asp** pueden encontrarse excelentes artículos acerca de pruebas de las *webapps*.

Puesto que muchas *webapps* evolucionan continuamente, el proceso de prueba es una actividad siempre en marcha, y se realiza para apoyar al personal que usa pruebas de regresión derivadas de las pruebas desarrolladas cuando se elaboró por primera vez la *webapp*. En el capítulo 20 se consideran métodos para probar la *webapp*.

## 17.6 PRUEBAS DE VALIDACIÓN

Las pruebas de validación comienzan en la culminación de las pruebas de integración, cuando se ejercitaron componentes individuales, el software está completamente ensamblado como un paquete y los errores de interfaz se descubrieron y corrigieron. En el nivel de validación o de sistema, desaparece la distinción entre software convencional, software orientado a objetos y *webapps*. Las pruebas se enfocan en las acciones visibles para el usuario y las salidas del sistema reconocibles por el usuario.

**PUNTO CLAVE**

Como todos los demás pasos de las pruebas, la validación intenta descubrir errores, pero el **enfoque se orienta** en **los requerimientos**: sobre las cosas que serán inmediatamente aparentes **para** el usuario final.

La validación puede definirse en muchas formas, pero una definición simple (aunque dura) es que la validación es exitosa cuando el software funciona en una forma que cumpla con las expectativas razonables del cliente. En este punto, un desarrollador de software curtido en la batalla puede protestar: "¿quién o qué es el árbitro de las expectativas razonables?". Si se desarrolló una *Especificación de requerimientos de software*, en ella se describen todos los atributos del software visibles para el usuario; contiene una sección de *Criterios de validación* que forman la base para un enfoque de pruebas de validación.

### 17.6.1 Criterios de pruebas de validación

La validación del software se logra a través de una serie de pruebas que demuestran conformidad con los requerimientos. Un plan de prueba subraya las clases de pruebas que se van a realizar y un procedimiento de prueba define casos de prueba específicos que se diseñan para garantizar que: se satisfacen todos los requerimientos de funcionamiento, se logran todas las características de comportamiento, todo el contenido es preciso y se presenta de manera adecuada, se logran todos los requerimientos de rendimiento, la documentación es correcta y se

satisfacen la facilidad de uso y otros requerimientos (por ejemplo, transportabilidad, compatibilidad, recuperación de error, mantenimiento).

Después de realizar cada caso de prueba de validación, existen dos posibles condiciones: 1) La característica de función o rendimiento se conforma de acuerdo con las especificaciones y se acepta, o 2) se descubre una desviación de la especificación y se crea una lista de deficiencias. Las desviaciones o errores descubiertos en esta etapa en un proyecto rara vez pueden corregirse antes de la entrega calendarizada. Con frecuencia es necesario negociar con el cliente para establecer un método para resolver deficiencias.

### 17.6.2 Revisión de la configuración

Un elemento importante del proceso de validación es una *revisión de la configuración*. La intención de la revisión es garantizar que todos los elementos de la configuración del software se desarrollaron de manera adecuada, y que se cataloga y se tiene el detalle necesario para reforzar las actividades de apoyo. La revisión de la configuración, en ocasiones llamada auditoría, se estudia con más detalle en el capítulo 22.

### 17.6.3 Pruebas alfa y beta

**Cita:**

"Teniendo los suficientes ojos, todos los errores son superficiales (por ejemplo, con una base suficientemente grande de personas que realizan pruebas beta y codesarrolladores, casi todo problema se caracterizará rápidamente y la corrección será obvia para alguien)."

**E. Raymond**

Virtualmente, es imposible que un desarrollador de software prevea cómo usará el cliente realmente un programa. Las instrucciones para usarlo pueden malinterpretarse; regularmente pueden usarse combinaciones extrañas de datos; la salida que parecía clara a quien realizó la prueba puede ser ininteligible para un usuario.

Cuando se construye software a la medida para un cliente, se realiza una serie de pruebas de aceptación a fin de permitir al cliente validar todos los requerimientos. Realizada por el usuario final en lugar de por los ingenieros de software, una prueba de aceptación puede variar desde una "prueba de conducción" informal hasta una serie de pruebas planificadas y ejecutadas sistemáticamente. De hecho, la prueba de aceptación puede realizarse durante un periodo de semanas o meses, y mediante ella descubrir errores acumulados que con el tiempo puedan degradar el sistema.

Si el software se desarrolla como un producto que va a ser usado por muchos clientes, no es práctico realizar pruebas de aceptación formales con cada uno de ellos. La mayoría de los constructores de productos de software usan un proceso llamado prueba alfa y prueba beta para descubrir errores que al parecer sólo el usuario final es capaz de encontrar.

**¿Cuál es la diferencia entre una prueba alfa y una prueba beta?**

La *prueba alfa* se lleva a cabo en el sitio del desarrollador por un grupo representativo de usuarios finales. El software se usa en un escenario natural con el desarrollador "mirando sobre el hombro" de los usuarios y registrando los errores y problemas de uso. Las pruebas alfa se realizan en un ambiente controlado.

La *prueba beta* se realiza en uno o más sitios del usuario final. A diferencia de la prueba alfa, por lo general el desarrollador no está presente. Por tanto, la prueba beta es una aplicación "en vivo" del software en un ambiente que no puede controlar el desarrollador. El cliente registra todos los problemas (reales o imaginarios) que se encuentran durante la prueba beta y los reporta al desarrollador periódicamente. Como resultado de los problemas reportados durante las pruebas beta, es posible hacer modificaciones y luego preparar la liberación del producto de software a toda la base de clientes.

En ocasiones se realiza una variación de la prueba beta, llamada *prueba de aceptación del cliente*, cuando el software se entrega a un cliente bajo contrato. El cliente realiza una serie de pruebas específicas con la intención de descubrir errores antes de aceptar el software del desarrollador. En algunos casos (por ejemplo, un gran corporativo o sistema gubernamental) la prueba de aceptación puede ser muy formal y abarcar muchos días o incluso semanas de prueba.

**CasaSegura**

### *Preparación para la validación*

**La escena:** Oficina de Doug Miller, mientras continúan tanto el diseño a nivel de componentes como la construcción de ciertos componentes.

**Participantes:** Doug Miller, jefe de ingeniería del software, Vinod, Jamie, Ed y Shakira, miembros del equipo de ingeniería del software *CasaSegura*.

**La conversación:**

**Doug:** El primer incremento estará listo para validación en... ¿cuánto tiempo? ¿Tres semanas?

**Vinod:** Más o menos. La integración va bien. Hacemos pruebas de humo todos los días y encontramos algunos *bugs*, pero nada que no podamos manejar. Hasta el momento va bien.

**Doug:** Háblame de la validación.

**Shakira:** Bueno, para el diseño de prueba, usaremos todos los casos de uso como base. Todavía no empiezo, pero desarrollaré pruebas para todos los casos de uso de las que sea responsable.

**Ed:** Igual yo.

**Jamie:** Yo también, pero debemos actuar juntos para la prueba de aceptación y también para las pruebas alfa y beta, ¿o no?

**Doug:** Sí. De hecho lo he pensado; podríamos traer un contratista externo para ayudarnos con la validación. Tengo dinero en el presupuesto... y él nos daría un nuevo punto de vista.

**Vinod:** Creo que lo tenemos bajo control.

**Doug:** Estoy seguro que sí, pero un GPI nos da un vistazo independiente del software.

**Jamie:** Estamos apretados de tiempo, Doug. Yo no tengo tiempo para vigilar a alguien que traigan para hacer el trabajo.

**Doug:** Lo sé, lo sé. Pero si un GPI funciona a partir de los requerimientos y los casos de uso, no necesitará demasiada vigilancia.

**Vinod:** Yo todavía creo que lo tenemos bajo control.

**Doug:** Te escuché, Vinod, pero voy a sostener mi opinión en esta ocasión. Más tarde planearemos la reunión con el representante del GPI para esta semana. Dejemos que comiencen y veamos lo que proponen.

**Vinod:** Muy bien, tal vez eso aligere un poco la carga.

## 17.7 Pruebas del sistema

**Cita:**

"Como la muerte y los impuestos, las pruebas son desagradables e inevitables".

Ed Yourdon

Al comienzo de este libro, se resaltó el hecho de que el software sólo es un elemento de un sistema basado en computadora más grande. A final de cuentas, el software se incorpora con otros elementos del sistema (por ejemplo, hardware, personas, información), y se lleva a cabo una serie de pruebas de integración y validación del sistema. Estas pruebas quedan fuera del ámbito del proceso de software y no se llevan a cabo exclusivamente por parte de ingenieros de software. Sin embargo, los pasos que se toman durante el diseño y la prueba del software pueden mejorar enormemente la probabilidad de integración exitosa del software en el sistema más grande.

Un problema clásico en la prueba del sistema es el "dedo acusador". Esto ocurre cuando se descubre un error y los desarrolladores de diferentes elementos del sistema se culpan unos a otros por el problema. En lugar de abandonarse a tal sinsentido, deben anticiparse los potenciales problemas de interfaz y: 1) diseñar rutas de manejo de error que prueben toda la información proveniente de otros elementos del sistema, 2) realizar una serie de pruebas que simulen los datos malos u otros errores potenciales en la interfaz del software, 3) registrar los resultados de las pruebas para usar como "evidencia" si ocurre el dedo acusador, y 4) participar en planificación y diseño de pruebas del sistema para garantizar que el software se prueba de manera adecuada.

En realidad, la *prueba del sistema* es una serie de diferentes pruebas cuyo propósito principal es ejercitar por completo el sistema basado en computadora. Aunque cada prueba tenga un propósito diferente, todo él funciona para verificar que los elementos del sistema se hayan integrado de manera adecuada y que se realicen las funciones asignadas. En las secciones que siguen se estudian los tipos de pruebas del sistema que valen la pena para los sistemas basados en software.

### 17.7.1 Pruebas de recuperación

Muchos sistemas basados en computadora deben recuperarse de fallas y reanudar el procesamiento con poco o ningún tiempo de inactividad. En algunos casos, un sistema debe ser tole-

rante a las fallas, es decir, las fallas del procesamiento no deben causar el cese del funcionamiento del sistema global. En otros casos, la falla de un sistema debe corregirse dentro de un periodo de tiempo específico u ocurrirán severos daños económicos.

La *recuperación* es una prueba del sistema que fuerza al software a fallar en varias formas y que verifica que la recuperación se realice de manera adecuada. Si la recuperación es automática (realizada por el sistema en sí), se evalúa el reinicio, los mecanismos de puntos de verificación, la recuperación de datos y la reanudación para correcciones. Si la recuperación requiere intervención humana, se evalúa el tiempo medio de reparación (TMR) para determinar si está dentro de límites aceptables.

### 17.7.2 Pruebas de seguridad

Cualquier sistema basado en computadora que gestione información sensible o cause acciones que puedan dañar (o beneficiar) de manera inadecuada a individuos es un blanco de penetración inadecuada o ilegal. La penetración abarca un amplio rango de actividades: *hackers* que intentan penetrar en los sistemas por deporte, empleados resentidos que intentan penetrar por venganza, individuos deshonestos que intentan penetrar para obtener ganancia personal ilícita.

La *prueba de seguridad* intenta verificar que los mecanismos de protección que se construyen en un sistema en realidad lo protegerán de cualquier penetración impropia. Para citar a Beizar [Bei84]: "La seguridad del sistema debe, desde luego, probarse para ser invulnerable ante ataques frontales; pero también debe probarse su invulnerabilidad contra ataques laterales y traseros."

Durante la prueba de seguridad, quien realiza la prueba juega el papel del individuo que desea penetrar al sistema. ¡Cualquier cosa vale! Quien realice la prueba puede intentar adquirir contraseñas por medios administrativos externos; puede atacar el sistema con software a la medida diseñado para romper cualquier defensa que se haya construido; puede abrumar al sistema, y por tanto negar el servicio a los demás; puede causar a propósito errores del sistema con la esperanza de penetrar durante la recuperación; puede navegar a través de datos inseguros para encontrar la llave de la entrada al sistema.

Con los suficientes tiempo y recursos, las buenas pruebas de seguridad a final de cuentas penetran en el sistema. El papel del diseñador de sistemas es hacer que el costo de la penetración sea mayor que el valor de la información que se obtendrá.

### 17.7.3 Pruebas de esfuerzo

Los primeros pasos de la prueba del software dieron como resultado una evaluación extensa de las funciones y el rendimiento normales del programa. Las pruebas de esfuerzo se diseñan para enfrentar los programas con situaciones anormales. En esencia, la persona que realiza las pruebas de esfuerzo pregunta: "¿cuánto podemos doblar esto antes de que se rompa?".

**Cita:**

"Si intenta encontrar verdaderos errores del sistema y no sujeta su software a una verdadera prueba de esfuerzo, entonces es el momento de comenzar."

**Boris Beizer**

La *prueba de esfuerzo* ejecuta un sistema en forma que demanda recursos en cantidad, frecuencia o volumen anormales. Por ejemplo, pueden 1) diseñarse pruebas especiales que generen diez interrupciones por segundo, cuando una o dos es la tasa promedio, (2) aumentarse las tasas de entrada de datos en un orden de magnitud para determinar cómo responderán las funciones de entrada, 3) ejecutarse casos de prueba que requieran memoria máxima y otros recursos, 4) diseñarse casos de prueba que puedan causar *thrashing* (que es un quebranto del sistema por hiperpaginación) en un sistema operativo virtual, 5) crearse casos de prueba que puedan causar búsqueda excesiva por datos residentes en disco. En esencia, la persona que realiza la prueba intenta romper el programa.

Una variación de la prueba de esfuerzo es una técnica llamada *prueba de sensibilidad*. En algunas situaciones (la más común ocurre en algoritmos matemáticos), un rango muy pequeño

de datos contenidos dentro de las fronteras de los datos válidos para un programa pueden causar procesamiento extremo, e incluso erróneo, o profunda degradación del rendimiento. La prueba de sensibilidad intenta descubrir combinaciones de datos dentro de clases de entrada válidas que puedan causar inestabilidad o procesamiento inadecuado.

### 17.7.4 Pruebas de rendimiento

Para sistemas en tiempo real y sistemas embebidos , el software que proporcione la función requerida, pero que no se adecue a los requerimientos de rendimiento, es inaceptable. La prueba de rendimiento se diseña para poner a prueba el rendimiento del software en tiempo de corrida, dentro del contexto de un sistema integrado. La prueba del rendimiento ocurre a lo largo de todos los pasos del proceso de prueba. Incluso en el nivel de unidad, puede accederse al rendimiento de un módulo individual conforme se realizan las pruebas. Sin embargo, no es sino hasta que todos los elementos del sistema están plenamente integrados cuando puede determinarse el verdadero rendimiento de un sistema.

Las pruebas de rendimiento con frecuencia se aparean con las pruebas de esfuerzo y por lo general requieren instrumentación de hardware y de software, es decir, con frecuencia es necesario medir la utilización de los recursos (por ejemplo, ciclos del procesador) en forma meticulosa. La instrumentación externa puede monitorear intervalos de ejecución y eventos de registro (por ejemplo, interrupciones) conforme ocurren, y los muestreos del estado de la máquina de manera regular. Con la instrumentación de un sistema, la persona que realiza la prueba puede descubrir situaciones que conduzcan a degradación y posibles fallas del sistema.

### 17.7.5 Pruebas de despliegue

En muchos casos, el software debe ejecutarse en varias plataformas y bajo más de un entorno de sistema operativo. La *prueba de despliegue*, en ocasiones llamada *prueba de configuración*, ejercita el software en cada entorno en el que debe operar. Además, examina todos los proce-

**HERRAMIENTAS DE SOFTWARE**

#### *Planeación y administración de pruebas*

**Objetivo:** Estas herramientas ayudan al equipo de software a planificar la estrategia de pruebas que se elija y a administrar el proceso de prueba mientras se lleva a cabo.

**Mecánica:** Las herramientas en esta categoría abordan la planificación de las pruebas, el almacenamiento, administración y control de las mismas; el seguimiento de los requisitos, la integración, el rastreo de errores y la generación de reportes. Los gestores de proyecto los usan para complementar las herramientas calendarizadas del proyecto. Quienes realizan las pruebas usan estas herramientas para planear actividades de prueba y controlar el flujo de información conforme avanza el proceso de pruebas.

**Herramientas representativas:**[4]

*QaTraq Test Case Management Tool*, desarrollada por Traq Software (**www.testmanagement.com**), "alienta un enfoque estructurado de la gestión de pruebas".

*QADirector*, desarrollada por Compuware Corp. (**www.compuware.com/qacenter**), proporciona un solo punto de control para gestionar todas las fases del proceso de pruebas.

*TestWorks*, desarrollada por Software Research, Inc. (**www.soft.com/Products/index.html**), contiene una suite completamente integrada de herramientas de prueba, incluidas herramientas para administración y reporte de pruebas.

*OpensourceTesting.org* (**www.opensourcetesting.org/testmgt.php**), cita varias herramientas de gestión y planificación de pruebas en fuente abierta.

*NI TestStand*, desarrollada por National Instruments Corp. (**www.ni.com**), le permite "desarrollar, gestionar y ejecutar secuencias de pruebas escritas en cualquier lenguaje de programación".

4 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

dimientos de instalación y el software de instalación especializado (por ejemplo, "instaladores") que usarán los clientes, así como toda la documentación que se usará para introducir el software a los usuarios finales.

Como ejemplo, piense en la versión accesible a internet del software *CasaSegura* que permitiría a un cliente monitorear el sistema de seguridad desde ubicaciones remotas. La *webapp* de *CasaSegura* debe probarse usando todos los navegadores web que es probable que se encuentren. Una prueba de despliegue más profunda puede abarcar combinaciones de navegadores web con varios sistemas operativos (por ejemplo, Linux, Mac OS, Windows). Puesto que la seguridad es un tema principal, un juego completo de pruebas de seguridad se integraría con la prueba de despliegue.

## 17.8 El arte de la depuración

**Cita:**

"Para nuestra sorpresa, descubrimos que no fue tan fácil obtener programas justo como los habíamos pensado. Recuerdo el instante exacto en el que me di cuenta de que una gran parte de mi vida, a partir de entonces, la iba a pasar descubriendo los errores en mis propios programas".

**Maurice Wilkes, descubre la depuración, 1949**

La prueba del software es un proceso que puede planearse y especificarse de manera sistemática. Puede realizarse el diseño de casos de prueba, definir una estrategia y evaluar los resultados comparándolos con las expectativas prescritas.

La *depuración* ocurre como consecuencia de las pruebas exitosas. Es decir, cuando un caso de prueba descubre un error, la depuración es el proceso que da como resultado la remoción del error. Aunque la depuración puede y debe ser un proceso ordenado, todavía en mucho es un arte. Los ingenieros de software con frecuencia se enfrentan con indicios "sintomáticos" de un problema de software mientras evalúan los resultados de una prueba, es decir, la manifestación externa del error y su causa interna pueden no tener relación obvia una con otra. El proceso mental, pobremente comprendido, que conecta un síntoma con una causa se conoce como depuración.

### 17.8.1 El proceso de depuración

La depuración no es una prueba, pero con frecuencia ocurre como consecuencia de una prueba.[5] De acuerdo con la figura 17.7, el proceso de depuración comienza con la ejecución de un caso de prueba. Los resultados se valoran y se encuentra la falta de correspondencia entre el rendimiento esperado y el real. En muchos casos, la no correspondencia de los datos es un síntoma de una causa subyacente y escondida. El proceso de depuración intenta relacionar síntoma con causa, lo que por tanto conduce a la corrección del error.

**CONSEJO**

*Asegúrese de evitar un tercer resultado: se encuentra la causa, pero la "corrección" no resuelve el problema o incluso introduce otro error.*

Por lo general, El proceso de depuración dará como resultado que: 1) la causa se encontrará y corregirá o 2) la causa no se encontrará. En el último caso, la persona que realiza la depuración puede sospechar una causa, diseñar un caso de prueba para auxiliarse en la validación de dicha suposición y trabajar hacia la corrección del error en forma iterativa.

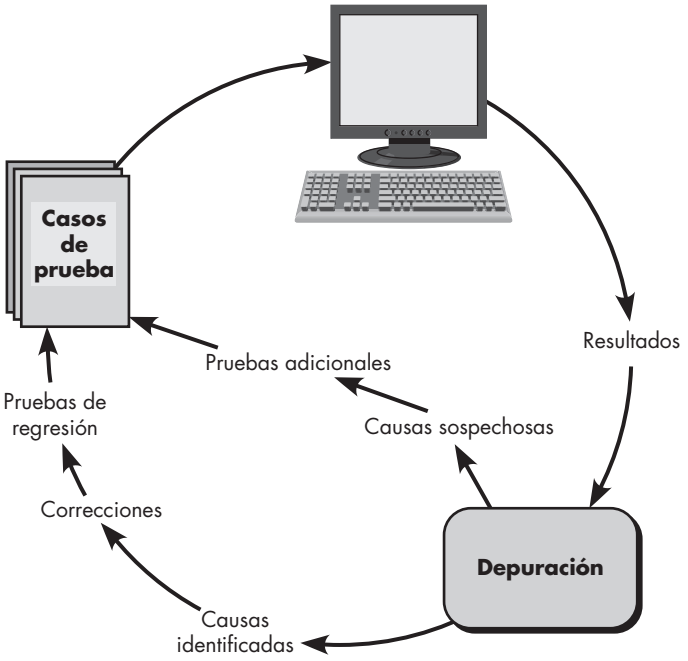
**? ¿Por qué es tan difícil la depuración?**

¿Por qué es tan difícil la depuración? Con toda probabilidad, la psicología humana (vea la sección 17.8.2) tiene más que ver con la respuesta que la tecnología del software. Sin embargo, ciertas características de los errores brindan algunas pistas:

1. El síntoma y la causa pueden ser geográficamente remotos. Es decir, el síntoma puede aparecer en una parte de un programa, mientras que la causa en realidad puede ubicarse en un sitio que esté alejado. Los componentes altamente acoplados (capítulo 8) exacerban esta situación.
2. El síntoma puede desaparecer (temporalmente) cuando se corrige otro error.

5 Al hacer esta afirmación, se toma la visión más amplia posible de las pruebas. No sólo el desarrollador prueba el software previo a su liberación, ¡sino que el cliente/usuario prueba el software cada vez que lo usa!

**FIGURA 17.7**
El proceso de depuración




3. El síntoma en realidad puede no ser causado por errores (por ejemplo, imprecisiones de redondeo).
4. El síntoma puede ser causado por un error humano que no se rastrea con facilidad.
5. El síntoma puede ser resultado de problemas de temporización más que de problemas de procesamiento.
6. Puede ser difícil reproducir con precisión las condiciones de entrada (por ejemplo, una aplicación en tiempo real en la que el orden de la entrada esté indeterminado).
7. El síntoma puede ser intermitente, particularmente común en sistemas embebidos que acoplan hardware y software de manera inextricable.
8. El síntoma puede deberse a causas que se distribuyen a través de algunas tareas que corren en diferentes procesadores.

**? "Todo mundo sabe que la depuración es el doble de difícil que escribir un programa por primera vez. De modo que, si se es tan inteligente como se puede ser cuando se escribe el programa, ¿cómo es que se depurará?".**

**Brian Kernighan**

Durante la depuración, encontrará errores que varían desde los ligeramente desconcertantes (por ejemplo, un formato de salida incorrecto) hasta los catastróficos (por ejemplo, la falla del sistema, que provoca serio daño económico o físico). Conforme aumentan las consecuencias de un error, también aumenta la cantidad de presión por encontrar la causa. Con frecuencia, la presión fuerza a algunos de los desarrolladores del software a corregir un error y, al mismo tiempo, introducir dos más.

### 17.8.2 Consideraciones psicológicas

Por desgracia, parece haber cierta evidencia de que la hazaña de la depuración es un rasgo humano innato. Algunas personas son buenas en ello y otras no lo son. Aunque la evidencia experimental de la depuración está abierta a muchas interpretaciones, se reportan grandes variaciones en la habilidad depuradora para programadores con la misma educación y experiencia. Al comentar acerca de los aspectos humanos de la depuración, Shneiderman [Shn80] afirma:

> La depuración es una de las partes más frustrantes de la programación. Tiene elementos de resolución de problemas o rompecabezas, junto con el desconcertante reconocimiento de que se cometió un error. La elevada ansiedad y la falta de voluntad para aceptar la posibilidad de los errores aumentan la dificultad de la tarea. Por fortuna, hay un gran alivio y la tensión se aligera cuando finalmente el error... se corrige.

Aunque puede ser difícil "aprender" a depurar, es posible proponer algunos enfoques al problema. Revise la sección 17.8.3.

**CASASEGURA**

### *Depuración*

**La escena:** Cubículo de Ed mientras se realiza la codificación y la prueba de unidad.

**Participantes:** Ed y Shakira, miembros del equipo de ingeniería de software *CasaSegura*.

**La conversación:**

**Shakira (observa a través de la entrada del cubículo):** Hola... ¿dónde estuviste a la hora del almuerzo?

**Ed:** Aquí... trabajando.

**Shakira:** Te ves horrible... ¿cuál es el problema?

**Ed (suspira):** He estado trabajando en este... error desde que lo descubrí a las 9:30 esta mañana y es, ¿qué?, 2:45... No tengo ni idea.

**Shakira:** Creí que todos estuvimos de acuerdo en no pasar más de una hora en tareas de depuración por cuenta propia; luego pediríamos ayuda, ¿verdad?

**Ed:** Sí, pero...

**Shakira (entra al cubículo):** ¿Así que cuál es el problema?

**Ed:** Es complicado y, además, lo he visto durante, ¿cuánto?, 5 horas. No lo vas a ver en 5 minutos.

**Shakira:** Permíteme... ¿cuál es el problema?

[Ed explica el problema a Shakira, quien lo observa durante 30 segundos sin hablar, luego...]

**Shakira (se asoma una sonrisa en su cara):** Oh, justo ahí, la variable llamada *setAlarmCondition*. ¿No debería ponerse en "falso" antes de comenzar el bucle?

[Ed mira la pantalla con incredulidad, se dobla hacia adelante y comienza a golpear su cabeza suavemente contra el monitor. Shakira, quien ahora sonríe abiertamente, se incorpora y sale del cubículo].

## 17.8.3 Estrategias de depuración


*Establezca un límite, por decir, dos horas, en la cantidad de tiempo que empleará al intentar depurar un problema por cuenta propia. Después de eso, ¡pida ayuda!*

Sin importar el enfoque que se tome, la depuración tiene un objetivo dominante: encontrar y corregir la causa de un error o defecto de software. El objetivo se realiza mediante una combinación de evaluación sistemática, intuición y suerte. Bradley [Bra85] describe el enfoque de depuración de la siguiente forma:

> La depuración es una aplicación directa del método científico que se ha desarrollado durante más de 2 500 años. La base de la depuración es localizar la fuente del problema [la causa] mediante una partición binaria, a través del trabajo con hipótesis que predicen nuevos valores por examinar.
>
> Tome un ejemplo simple que no sea de software: una lámpara en mi casa no funciona. Si nada en la casa funciona, la causa debe estar en el interruptor principal o en el exterior; observo mi alrededor para ver si el vecindario está a oscuras. Conecto la lámpara sospechosa en un tomacorriente que funcione y un electrodoméstico operativo en el circuito sospechoso. Y así continúo la alternación entre hipótesis y pruebas.

En general, se han propuesto tres estrategias de depuración [Mye79]: 1) fuerza bruta, 2) vuelta atrás (del inglés *backtracking*) y 3) eliminación de causas. Cada una de estas estrategias puede llevarse a cabo de manera manual, pero modernas herramientas de depuración pueden hacer el proceso mucho más efectivo.

**Tácticas de depuración.** La categoría *fuerza bruta* de la depuración probablemente es el método más común y menos eficiente para aislar la causa de un error de software. Los métodos

**Cita:**

"El primer paso para reparar un programa descompuesto es hacerlo fallar repetidamente (en el ejemplo más simple posible)."

**T. Duff**

de depuración de fuerza bruta se aplican cuando todo lo demás falla. Al usar una filosofía de "deje que la computadora encuentre el error", se toman copias de la memoria (*dumps*), se invocan rastreos en el tiempo de corrida y el programa se carga con enunciados de salida. La esperanza es que, en alguna parte del pantano de información que se produzca, se encontrará una pista que pueda conducir a la causa de un error. Aunque la masa de información producida a final de cuentas puede conducir al éxito, con más frecuencia conduce a desperdicio de esfuerzo y tiempo. ¡Piense que primero debe gastarse!

El *seguimiento hacia atrás* o vuelta atrás es un enfoque de depuración bastante común que puede usarse exitosamente en programas pequeños. Al comenzar en el sitio donde se descubrió un síntoma, el código fuente se rastrea hacia atrás (de manera manual) hasta que se encuentra la causa. Por desgracia, conforme aumenta el número de líneas fuente, el número de rutas potenciales hacia atrás puede volverse inmanejable.

El tercer enfoque de la depuración, la *eliminación de la causa*, se manifiesta mediante inducción o deducción, e introduce el concepto de partición binaria. Los datos relacionados con la ocurrencia del error se organizan para aislar las causas potenciales. Se plantea una "hipótesis de causa" y los datos anteriormente mencionados se usan para probar o refutar la hipótesis. De manera alternativa, se desarrolla una lista de las posibles causas y se realizan pruebas para eliminar cada una. Si las pruebas iniciales indican que una hipótesis de causa particular se muestra prometedora, los datos se refinan con la intención de aislar el error.

**Depuración automatizada.** Cada uno de estos enfoques de depuración puede complementarse con herramientas de depuración que puedan proporcionar apoyo semiautomático conforme se intenten estrategias de depuración. Hailpern y Santhanam [Hai02] resumen el estado de estas herramientas cuando apuntan: "... se han propuesto muchos nuevos enfoques y están disponibles muchos entornos de depuración comerciales. Los entornos de desarrollo integrados (IDE) brindan una forma de capturar algunos de los errores predeterminados específicos del lenguaje (por ejemplo, falta de caracteres de fin de sentencia , variables indefinidas, etc.) sin requerir compilación". Se dispone de una gran variedad de compiladores de depuración, ayudas dinámicas de depuración ("trazadores"), generadores automáticos de casos de prueba y herramientas de mapeo de referencia cruzada. Sin embargo, las herramientas no son un sustituto

**HERRAMIENTAS DE SOFTWARE**

### *Depuración*

**Objetivo:** Estas herramientas proporcionan asistencia automatizada para quienes deben depurar problemas de software. La intención es proporcionar conocimiento que puede ser difícil de obtener si se aborda el proceso de depuración de forma manual.

**Mecánica:** La mayoría de las herramientas de depuración son específicas del lenguaje de programación y del entorno.

**Herramientas representativas:**[6]

*Borland Gauntlet*, distribuido por Borland (**www.borland. com**), auxilia tanto en las pruebas como en la depuración.

*Coverty Prevent SQS*, desarrollada por Coverty (**www.coverty. com**), proporciona asistencia de depuración tanto para C++ como para Java.

*C++Test*, desarrollada por Parasoft (**www.parasoft.com**), es una herramienta de prueba de unidad que soporta un rango completo de pruebas en código C y C++. Las características de depuración ayudan en el diagnóstico de errores que se encuentren.

*CodeMedic*, desarrollada por NewPlanet Software (**www. newplanetsfotware.com/medic/**), proporciona una interfaz gráfica para el depurador estándar UNIX, *gdb*, e implementa sus características más importantes. En la actualidad, *gdb* soporta C/C++, Java, PalmOS, varios sistemas incrustados, lenguaje ensamblador, FORTRAN y Modula-2.

*GNATS*, una aplicación freeware (**www.gnu.org/software/ gnats**), es un conjunto de herramientas para rastrear reportes de error.

6 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

para la evaluación cuidadosa basada en un modelo completo de diseño y en código fuente claro.

**El factor humano.** Cualquier discusión de los enfoques y herramientas de depuración está incompleta sin mencionar un poderoso aliado: ¡otras personas! Un punto de vista fresco, no empañado por horas de frustración, puede hacer maravillas.[7] Una máxima final para la depuración puede ser: "Cuando todo lo demás falle, ¡consiga ayuda!"

### 17.8.4 Corrección del error

Una vez encontrado el error, debe corregirse. Pero, como ya se señaló, la corrección de un error puede introducir otros errores y, por tanto, hacer más daño que bien. Van Vleck [Van89] sugiere tres preguntas simples que deben plantearse antes de hacer la "corrección" que remueva la causa de un error:

**Cita:**

"El mejor examinador no es aquel que encuentra más errores... el mejor es quien consigue la corrección de más errores."

**Cem Kaner *et al.***

1. *¿La causa del error se reproduce en otra parte del programa?* En muchas situaciones, un defecto de programa es causado por un patrón de lógica erróneo que puede reproducirse en alguna otra parte. La consideración explícita del patrón lógico puede resultar en el descubrimiento de otros errores.
2. *¿Qué "siguiente error" puede introducirse con la corrección que está a punto de realizar?* Antes de hacer la corrección, debe evaluarse el código fuente (o, mejor, el diseño) para valorar el acoplamiento de las estructuras lógica y de datos. Si la corrección se realizará en una sección altamente acoplada del programa, debe tenerse especial cuidado cuando se realice algún cambio.
3. *¿Qué debió hacerse para evitar este error desde el principio?* Esta pregunta es el primer paso hacia el establecimiento de un enfoque de aseguramiento de calidad estadística del software (capítulo 16). Si se corrigen tanto el proceso como el producto, el error se removerá del programa actual y podrá eliminarse de todos los programas futuros.

## 17.9 Resumen

Las pruebas de software representan el porcentaje más grande de esfuerzo técnico en el proceso de software. Sin importar el tipo de software que se construya, una estrategia para planificar, ejecutar y controlar pruebas sistemáticas comienza por considerar pequeños elementos del software y moverse hacia afuera, hacia el programa como un todo.

El objetivo de las pruebas del software es descubrir errores. Para software convencional, este objetivo se logra mediante una serie de pasos de prueba. Las pruebas de unidad e integración se concentran en la verificación funcional de un componente y en la incorporación de componentes en una arquitectura de software. Las pruebas de validación demuestran la conformidad con los requerimientos del software y las pruebas del sistema validan el software una vez que se incorporó en un sistema más grande. Cada paso de la prueba se logra a través de una serie de técnicas de prueba sistemáticas que auxilian en el diseño de casos de prueba. Con cada paso de prueba, se amplía el nivel de abstracción con la que se considera el software.

La estrategia para probar software orientado a objeto comienza con pruebas que ejercitan las operaciones dentro de una clase y luego avanzan hacia la prueba basada en hebra para integración. Las hebras son conjuntos de clases que responden a una entrada o evento. Las pruebas basadas en uso se enfocan en clases que no colaboran demasiado con otras clases.

7 El concepto de programación por parejas (recomendado como parte del modelo de programación extrema que se estudió en el capítulo 3), proporciona un mecanismo de "depuración" conforme se diseña y codifica el software.

Las *webapps* se prueban en forma muy parecida a los sistemas OO. Sin embargo, las pruebas se diseñan para ejercitar contenido, funcionalidad, interfaz, navegación y aspectos de rendimiento y seguridad de la *webapp*.

A diferencia de las pruebas (una actividad sistemática planificada), la depuración puede verse como un arte. Al comenzar con una indicación sintomática de un problema, la actividad de depuración debe rastrear la causa de un error. De los muchos recursos disponibles durante la depuración, el más valioso es el consejo de otros miembros del equipo de ingeniería del software.

## Problemas y puntos por evaluar

**17.1.** Con sus palabras, describa la diferencia entre verificación y validación. ¿Ambas usan los métodos de diseño de casos de prueba y estrategias de pruebas?

**17.2.** Mencione algunos problemas que pueden asociarse con la creación de un grupo de prueba independiente. ¿Los GPI y el SQA se integran con las mismas personas?

**17.3.** ¿Siempre es posible desarrollar una estrategia para probar software que usa la secuencia de pasos de prueba descritos en la sección 17.1.3? ¿Qué posibles complicaciones pueden surgir para sistemas incrustados?

**17.4.** ¿Por qué un módulo altamente acoplado es difícil para la prueba de unidad?

**17.5.** El concepto de "antierrores" (sección 17.3.1) es una forma extremadamente efectiva de brindar asistencia de depuración interna cuando se descubre un error:

*a*) Desarrolle un conjunto de lineamientos para antierror.
*b*) Analice las ventajas de usar la técnica.
*c*) Analice las desventajas.

**17.6.** ¿Cómo puede la calendarización del proyecto afectar la prueba de integración?

**17.7.** ¿La prueba de unidad es posible o incluso deseable en todas las circunstancias? Proporcione ejemplos para justificar su respuesta.

**17.8.** ¿Quién debe realizar la prueba de validación: el desarrollador o el usuario del software? Justifique su respuesta.

**17.9.** Desarrolle una estrategia de prueba completa para el sistema *CasaSegura* que se estudió anteriormente en este libro. Documéntela en una *Especificación de pruebas*.

**17.10.** Como proyecto de clase, desarrolle una *Guía de depuración* para su instalación. ¡La guía debe brindar lenguaje y sugerencias orientadas a sistemas aprendidos en la escuela de la vida! Comience por destacar los temas que revisarán la clase y el instructor. Publique la guía para otros en su entorno local.

## Lecturas y fuentes de información adicionales

Virtualmente todo libro acerca de las pruebas del software analiza estrategias junto con métodos para diseño de casos de prueba. Everett y Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,* Wiley, 2007), Spiller *et al*. (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3a. ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement,* 2a. ed., Auerbach, 2004), Loveland *et al*. (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig y Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), Whittaker (*How to Break Software,* Addison-Wesley, 2002), y Kaner *et al*. (*Lessons Learned in Software Testing,* Wiley, 2001) son sólo una pequeña muestra de muchos libros que estudian los principios, conceptos, estrategias y métodos de las pruebas.

Para aquellos lectores con interés en los métodos de desarrollo de software ágiles, 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 *et al*. (*Lessons Learned*

*in Software Testing*, Wiley, 2001) presentan una colección de más de 300 "lecciones" pragmáticas (lineamientos) que todo examinador 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 todo tipo de software desarrollado o adquirido. Manges y O'Brien (*Agile Testing with Ruby and Rails*, Apress, 2008) abordan estrategias y técnicas de prueba para el lenguaje de programación Ruby y el marco conceptual web.

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 *et. al*. (*Testing Object-Oriented Software,* IEEE Computer Society Press, 1998) y Marick (*The Craft of Software Testing,* Prentice-Hall, 1997) presentan estrategias y métodos para probar sistemas OO.

Lineamientos para depuración se encuentran en los libros de Grötker *et al*. (*The Developer's Guide to Debugging,* Springer, 2008), Agans (*Debugging,* Amacon, 2006), Zeller (*Why Programs Fail: A Guide to Systematic Debugging,* Morgan Kaufmann, 2005), Tells y Hsieh (*The Science of Debugging,* The Coreolis Group, 2001), y Robbins (*Debugging Applications,* Microsoft Press, 2000). Kaspersky (*Hacker Debugging Uncovered,* A-List Publishing, 2005) addresses the technology of debugging tools. Younessi (*Object-Oriented Defect Management of Software,* Prentice-Hall, 2002) aborda la tecnología de las herramientas para depuración. Younessi (*Object-Oriented Defect Management of Software*, Prentice-Hall, 2002) presenta técnicas para manejar defectos que se encuentran en sistemas orientados a objetos. Beizer [Bei84] presenta una interesante "taxonomía de errores" que puede conducir a métodos efectivos para planificación de pruebas.

Los libros de Madisetti y Akgul (*Debugging Embedded Systems,* Springer, 2007), Robbins (*Debugging Microsoft.NET 2.0 Applications,* Microsoft Press, 2005), Best (*Linux Debugging and Performance Tuning,* Prentice-Hall, 2005), Ford y Teorey (*Practical Debugging in C++,* Prentice-Hall, 2002), Brown (*Debugging Perl,* McGraw-Hill, 2000) y Mitchell (*Debugging Java,* McGraw-Hill, 2000) abordan la naturaleza especial de la depuración para los entornos implicados en sus títulos.

Una gran variedad de fuentes de información acerca de estrategias para pruebas está disponible en internet. Una lista actualizada de referencias en la World Wide Web, que son relevantes para las estrategias de prueba de software, puede encontrarse en el sitio web del libro: **www.mhle.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CAPÍTULO

# 18

# PRUEBA DE APLICACIONES CONVENCIONALES

CONCEPTOS CLAVE



Las pruebas presentan una interesante anomalía para los ingenieros de software, quienes por naturaleza son personas constructivas. Las pruebas requieren que el desarrollador deseche nociones preconcebidas sobre lo "correcto" del software recién desarrollado y luego trabajen duro para diseñar casos de prueba a fin de "romper" el software. Beizer [Bei90] describe esta situación de manera efectiva cuando afirma:

> Existe el mito de que no habría errores que pescar si fuésemos realmente buenos en programación. Si realmente nos pudiéramos concentrar, si todo mundo usara programación estructurada, diseño descendente... entonces no habría errores. Ése es el mito. Hay errores, dice el mito, porque somos malos en lo que hacemos; y si lo somos, deberíamos sentirnos culpables por ello. Por tanto, la aplicación de pruebas y el diseño de casos de prueba es una admisión del fracaso, que inspira una buena dosis de culpa. Y el tedio de las pruebas es un justo castigo por nuestros errores. ¿El castigo por qué? ¿Por ser humanos? ¿Culpa por qué? ¿Por fracasar en lograr la perfección inhumana? ¿Por no distinguir entre lo que otro programador piensa y lo que dice? ¿Por no poder ser telépatas? ¿Por no resolver problemas de comunicación humana a los que se les ha dado la vuelta... durante siglos?

¿Las pruebas deben inspirar culpa? ¿Las pruebas son realmente destructivas? La respuesta a estas preguntas es: "¡No!"

En este capítulo se estudian técnicas para el diseño de casos de prueba de software para aplicaciones convencionales. Este diseño se enfoca en un conjunto de técnicas para la creación de casos de prueba que satisfacen los objetivos de prueba globales y las estrategias de pruebas que se estudiaron en el capítulo 17.

## UNA MIRADA RÁPIDA

**¿Qué es?** Una vez generado el código fuente, el software debe probarse para descubrir (y corregir) tantos errores como sea posible antes de entregarlo al cliente. La meta es diseñar una serie de casos de prueba que tengan una alta probabilidad de encontrar errores; ¿pero cómo? Ahí es donde entran en escena las técnicas de prueba de software. Dichas técnicas proporcionan lineamientos sistemáticos para diseñar pruebas que: 1) revisen la lógica interna y las interfaces de todo componente de software y 2) revisen los dominios de entrada y salida del programa para descubrir errores en el funcionamiento, comportamiento y rendimiento del programa.

**¿Quién lo hace?** Durante las primeras etapas del proceso, un ingeniero de software realiza todas las pruebas. Sin embargo, conforme avanza el proceso, pueden involucrarse especialistas en pruebas.

**¿Por qué es importante?** Las revisiones y otras acciones SQA pueden y deben descubrir errores, pero no son suficientes. Cada vez que el programa se ejecuta, ¡el cliente lo prueba! Por tanto, tiene que ejecutarse el programa antes de que llegue al cliente, con la intención específica de encontrar y remover todos los errores. Para encontrar el mayor número posible de éstos, las pruebas deben realizarse de manera sistemática y deben diseñarse casos de prueba usando técnicas sistematizadas.

**¿Cuáles son los pasos?** Para aplicaciones convencionales, el software se prueba desde dos perspectivas diferentes: 1) la lógica de programa interno se revisa usando técnicas de diseño de casos de prueba de "caja blanca" y 2) los requerimientos de software se revisan usando técnicas de diseño de casos de prueba de "caja negra". El uso de casos auxilia en el diseño de pruebas para descubrir errores de validación del software. En todo caso, la intención es encontrar el máximo número de errores con la mínima cantidad de esfuerzo y tiempo.

**¿Cuál es el producto final?** Se diseña y documenta un conjunto de casos de prueba elaborados para revisar la lógica interna, las interfaces, las colaboraciones de componentes y los requerimientos externos; se definen los resultados esperados y se registran los resultados reales.

**¿Cómo me aseguro de que lo hice bien?** Cuando se realizan pruebas, cambia el punto de vista. ¡Intente con ahínco "romper" el software! Diseñe casos de prueba en forma sistemática y revise minuciosamente los casos de prueba creados. Además, puede evaluar la cobertura de la prueba y rastrear las actividades de detección de errores.

## 18.1 FUNDAMENTOS DE LAS PRUEBAS DEL SOFTWARE

**Cita:**

"Todo programa hace algo bien, sólo que puede no ser aquello que queremos que haga."

**Anónimo**

La meta de probar es encontrar errores, y una buena prueba es aquella que tiene una alta probabilidad de encontrar uno. Por tanto, un sistema basado en computadora o un producto debe diseñarse e implementarse teniendo en mente la "comprobabilidad". Al mismo tiempo, las pruebas en sí mismas deben mostrar un conjunto de características que logren la meta de encontrar la mayor cantidad de errores con el mínimo esfuerzo.

**Comprobabilidad.** James Bach[1] proporciona la siguiente definición de comprobabilidad: "La *comprobabilidad del software* significa simplemente saber con cuánta facilidad puede probarse [un programa de cómputo]." Las siguientes características conducen a software comprobable.

**¿Cuáles son las características de la comprobabilidad?**

*Operatividad*. "Mientras mejor funcione, más eficientemente puede probarse." Si un sistema se diseña e implementa teniendo como objetivo la calidad, relativamente pocos errores bloquearán la ejecución de las pruebas, lo que permitirá avanzar en ellas sin interrupciones.

*Observabilidad*. "Lo que ve es lo que prueba." Las entradas proporcionadas como parte de las pruebas producen distintas salidas. Los estados del sistema y las variables son visibles o consultables durante la ejecución. La salida incorrecta se identifica con facilidad. Los errores internos se detectan y se reportan de manera automática. El código fuente es accesible.

*Controlabilidad*. "Mientras mejor pueda controlar el software, más podrá automatizar y optimizar las pruebas." Todas las salidas posibles pueden generarse a través de alguna combinación de entradas, y los formatos de entrada/salida (E/S) son consistentes y estructurados. Todo código es ejecutable a través de alguna combinación de entradas. El ingeniero de pruebas puede controlar directamente los estados del software, del hardware y las variables. Las pruebas pueden especificarse, automatizarse y reproducirse convenientemente.

*Descomponibilidad*. "Al controlar el ámbito de las pruebas, es posible aislar más rápidamente los problemas y realizar pruebas nuevas y más inteligentes." El sistema de software se construye a partir de módulos independientes que pueden probarse de manera independiente.

**Cita:**

"Los errores son más comunes, más dominantes y más problemáticos en software que en otras tecnologías."

**David Parnas**

*Simplicidad*. "Mientras haya menos que probar, más rápidamente se le puede probar." El programa debe mostrar *simplicidad funcional* (por ejemplo, el conjunto característico es el mínimo necesario para satisfacer los requerimientos); *simplicidad estructural* (la arquitectura es modular para limitar la propagación de fallos) y *simplicidad de código* (se adopta un estándar de codificación para facilitar la inspección y el mantenimiento).

*Estabilidad*. "Mientras menos cambios, menos perturbaciones para probar." Los cambios al software son raros, se controlan cuando ocurren y no invalidan las pruebas existentes. El software se recupera bien de los fallos.

*Comprensibilidad*. "Mientras más información se tenga, se probará con más inteligencia." El diseño arquitectónico y las dependencias entre componentes internos, externos y compartidos son bien comprendidos. La documentación técnica es accesible al instante, está bien organizada, es específica, detallada y precisa. Los cambios al diseño son comunicados a los examinadores.

Pueden usarse los atributos sugeridos por Bach para desarrollar una configuración de software (es decir, programas, datos y documentos) que sean fáciles de probar.

**Características de la prueba.** ¿Y qué hay acerca de las pruebas en sí? Kaner, Falk y Nguyen [Kan93] sugieren los siguientes atributos de una "buena" prueba:

1 Los párrafos que siguen se usan con permiso de James Bach (copyright 1994) y se adaptaron de material que originalmente apareció en un comentario en el grupo de noticias comp.software-eng.

**? ¿Qué es una "buena" prueba?**

*Una buena prueba tiene una alta probabilidad de encontrar un error.* Para lograr esta meta, el examinador debe comprender el software e intentar desarrollar una imagen mental de cómo puede fallar. De manera ideal, se prueban las clases de fallas. Por ejemplo, una clase de fallas potenciales en una interfaz gráfica de usuario es la falla para reconocer la posición adecuada del ratón. Se diseña entonces un conjunto de pruebas para revisar el ratón con la intención de demostrar un error en el reconocimiento de la posición del ratón.

*Una buena prueba no es redundante*. El tiempo y los recursos de la prueba son limitados. No se trata de realizar una prueba que tenga el mismo propósito que otra. Cada una debe tener un propósito diferente (incluso si es sutilmente diferente).

*Una buena prueba debe ser "la mejor de la camada"* [Kan93]. En un grupo de pruebas que tengan una intención similar, las limitaciones de tiempo y recursos pueden mitigar la ejecución de sólo un subconjunto de dichas pruebas. En tales casos, debe usarse la prueba que tenga la mayor probabilidad de descubrir toda una clase de errores.

*Una buena prueba no debe ser demasiado simple o demasiado compleja.* Aunque en ocasiones es posible combinar una serie de pruebas en un caso de prueba, los efectos colaterales posibles asociados con este enfoque pueden enmascarar errores. En general, cada prueba debe ejecutarse por separado.

**CASASEGURA**

### *Diseño de pruebas únicas*

**La escena:** Cubículo de Vinod.

**Participantes:** Vinod y Ed, miembros del equipo de ingeniería de software CasaSegura.

**La conversación:**

**Vinod:** Así que éstos son los casos de prueba que quieres aplicar para la operación *passwordValidation*.

**Ed:** Sí, deben cubrir muchas de las posibilidades para los tipos de contraseñas que pueda ingresar un usuario.

**Vinod:** Veamos... observas que la contraseña correcta será 8080, ¿verdad?

**Ed:** Ajá.

**Vinod:** ¿Y especificas las contraseñas 1234 y 6789 para probar el error en el reconocimiento de las contraseñas inválidas?

**Ed:** Exacto, y también pruebo las contraseñas que están cerca de la contraseña correcta, a ver... 8081 y 8180.

**Vinod:** Ésos están bien, pero no le veo mucho caso aplicar las entradas 1234 y 6789. Son redundantes... prueban la misma cosa, ¿o no?

**Ed:** Bueno, son valores diferentes.

**Vinod:** Es cierto, pero si 1234 no descubre un error... en otras palabras... la operación *passwordValidation* detecta que es una contraseña inválida, no es probable que 6789 nos muestre algo nuevo.

**Ed:** Ya veo lo que dices.

**Vinod:** No intento ser quisquilloso... es sólo que tenemos tiempo limitado para hacer las pruebas, así que es buena idea aplicar pruebas que tengan una alta probabilidad de encontrar nuevos errores.

**Ed:** No hay problema... Pensaré en esto un poco más.

## 18.2 VISIONES INTERNA Y EXTERNA DE LAS PRUEBAS

**Cita:**

"Sólo hay una regla en el diseño de casos de prueba: cubrir todas las características, mas no hacer demasiados casos de prueba."

Tsuneo Yamaura

Cualquier producto sometido a ingeniería (y la mayoría de otras cosas) pueden probarse en una de dos formas: 1) al conocer la función específica que se asignó a un producto para su realización, pueden llevarse a cabo pruebas que demuestren que cada función es completamente operativa mientras al mismo tiempo se buscan errores en cada función, 2) al conocer el funcionamiento interno de un producto, pueden realizarse pruebas para garantizar que "todos los engranes embonan"; es decir, que las operaciones internas se realizan de acuerdo con las especificaciones y que todos los componentes internos se revisaron de manera adecuada. El primer

enfoque de pruebas considera una visión externa y se llama prueba de caja negra. El segundo requiere una visión interna y se denomina prueba de caja blanca.[2]

**PUNTO CLAVE**

Las pruebas de caja blanca pueden diseñarse sólo después de que existe el diseño a nivel de componentes (o código fuente). Debe disponerse de los detalles lógicos del programa.

La *prueba de caja negra* se refiere a las pruebas que se llevan a cabo en la interfaz del software. Una prueba de caja negra examina algunos aspectos fundamentales de un sistema con poca preocupación por la estructura lógica interna del software. La *prueba de caja blanca* del software se basa en el examen cercano de los detalles de procedimiento. Las rutas lógicas a través del software y las colaboraciones entre componentes se ponen a prueba al revisar conjuntos específicos de condiciones y/o bucles.

A primera vista, parecería que las pruebas de caja blanca muy extensas conducirían a "programas 100 por ciento correctos". Lo único que se necesita es definir todas las rutas lógicas, desarrollar casos de prueba para revisarlas y evaluar resultados, es decir, generar casos de prueba para revisar de manera exhaustiva la lógica del programa. Por desgracia, las pruebas exhaustivas presentan ciertos problemas logísticos. Hasta para programas pequeños, el número de posibles rutas lógicas puede ser muy grande. Sin embargo, las pruebas de caja blanca no deben descartarse como imprácticas. Puede seleccionarse y revisarse un número limitado de rutas lógicas importantes. Puede probarse la validez de las estructuras de datos importantes.

**INFORMACIÓN**

### *Pruebas exhaustivas*

Considere un programa de 100 líneas en el 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, dependiendo de las condiciones especificadas en la entrada. Dentro del bucle interior, se requieren cuatro constructos if-then-else. ¡Existen aproximadamente $10^{14}$ rutas posibles que pueden ejecutarse en este programa!

Para poner este número en perspectiva, suponga que se desarrolló un procesador de prueba mágico ("mágico" porque no existe tal procesador) para realizar pruebas exhaustivas. El procesador puede desarrollar un caso de prueba, ejecutarlo y evaluar los resultados en un milisegundo. Si trabajara 24 horas al día los 365 días del año, el procesador trabajaría durante 3 170 años para probar el programa. Esto, sin duda alguna, causaría estragos en la mayoría de los calendarios de desarrollo.

Por tanto, es razonable afirmar que la prueba exhaustiva es imposible para sistemas de software grandes.

## 18.3 PRUEBA DE CAJA BLANCA

**Cita:**

"Los errores se esconden en las esquinas y se congregan en las fronteras."

**Boris Beizer**

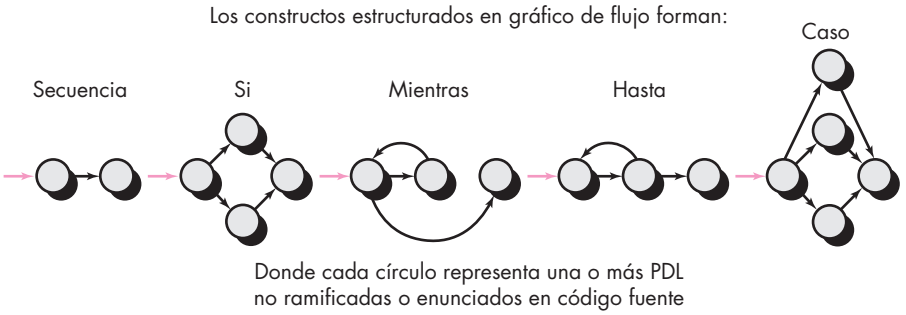
La *prueba de caja blanca*, en ocasiones llamada *prueba de caja de vidrio*, es una filosofía de diseño de casos de prueba que usa la estructura de control descrita como parte del diseño a nivel de componentes para derivar casos de prueba. Al usar los métodos de prueba de caja blanca, puede derivar casos de prueba que: 1) garanticen que todas las rutas independientes dentro de un módulo se revisaron al menos una vez, 2) revisen todas las decisiones lógicas en sus lados verdadero y falso, 3) ejecuten todos los bucles en sus fronteras y dentro de sus fronteras operativas y 4) revisen estructuras de datos internas para garantizar su validez.

## 18.4 PRUEBA DE RUTA BÁSICA

La *prueba de ruta* o *trayectoria básica* es una técnica de prueba de caja blanca propuesta por primera vez por Tom McCabe [McC76]. El método de ruta básica permite al diseñador de casos de prueba derivar una medida de complejidad lógica de un diseño de procedimiento y usar esta

2 En ocasiones, en lugar de pruebas de caja negra y de caja blanca, se usan, respectivamente, los términos *prueba funcional* y *prueba estructural*.

**FIGURA 18.1**
**Notación de gráfico de flujo**




medida como guía para definir un conjunto básico de rutas de ejecución. Los casos de prueba derivados para revisar el conjunto básico tienen garantía para ejecutar todo enunciado en el programa, al menos una vez durante la prueba.

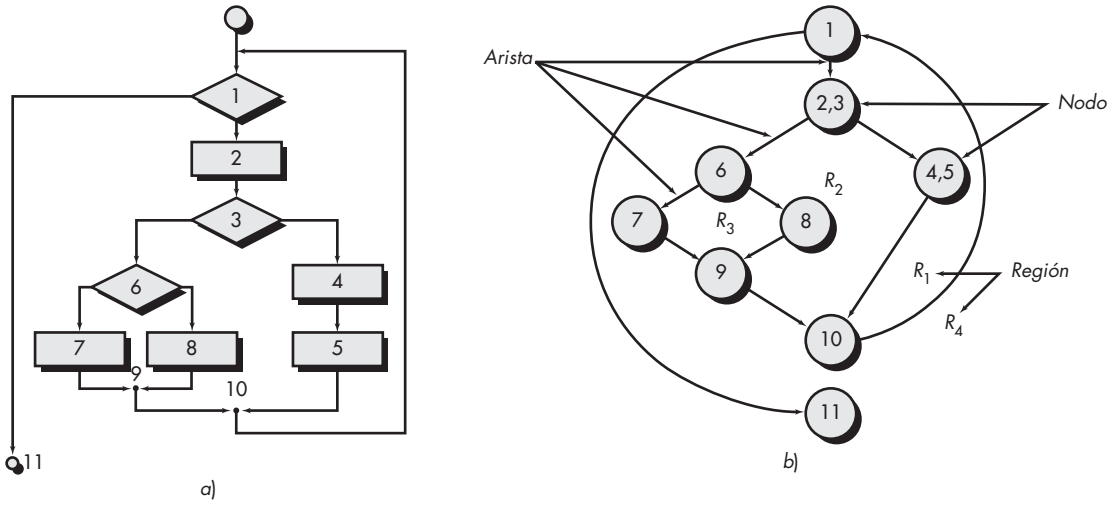
### 18.4.1 Notación de gráfico o grafo de flujo

Antes de considerar el método de ruta básica, debe introducirse una notación simple para la representación del flujo de control, llamado *gráfico de flujo* (o *gráfico de programa*).[3] El gráfico de flujo muestra el flujo de control lógico que usa la notación ilustrada en la figura 18.1. Cada constructo estructurado (capítulo 10) tiene un correspondiente símbolo de gráfico de flujo.


*Un gráfico de flujo debe dibujarse sólo cuando la estructura lógica de un componente es compleja. El gráfico de flujo le permite rastrear rutas de programa con más facilidad.*

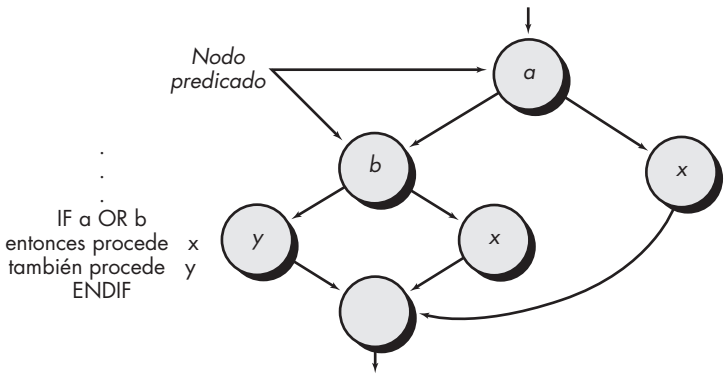
Para ilustrar el uso de un gráfico de flujo, considere la representación del diseño de procedimiento en la figura 18.2*a*). Aquí se usó un diagrama de flujo para mostrar la estructura de control del programa. La figura 18.2*b*) mapea el diagrama de flujo en un gráfico de flujo correspondiente (suponiendo que el diagrama de flujo no contiene condiciones compuestas en los diamantes de decisión). Con referencia a la figura 18.2*b*), cada círculo, llamado *nodo de gráfico de flujo*, representa uno o más enunciados de procedimiento. Una secuencia de cajas de proceso y un diamante de decisión pueden mapearse en un solo nodo. Las flechas en el gráfico de flujo, llamadas *aristas* o *enlaces*, representan flujo de control y son análogas a las flechas en el diagrama de flujo. Una arista debe terminar en un nodo, incluso si el nodo no representa algún enunciado de pro-

**FIGURA 18.2**
***a)* Diagrama de flujo y *b)* gráfico de flujo**




3 En la actualidad, el método de ruta básica puede realizarse sin el uso de gráficos de flujo. No obstante, sirven como una notación útil para comprender el flujo de control e ilustrar el enfoque.

**FIGURA 18.3**
**Lógica compuesta**




cedimiento (por ejemplo, vea el símbolo de gráfico de flujo para el constructo if-then-else). Las áreas acotadas por aristas y nodos se llaman *regiones*. Cuando se cuentan las regiones, el área afuera del gráfico se incluye como región.[4]

Cuando en un diseño de procedimiento se encuentran condiciones compuestas, la generación de un gráfico de flujo se vuelve ligeramente más complicada. Una condición compuesta ocurre cuando uno o más operadores booleanos (OR, AND, NAND, NOR lógicos) se presenta en un enunciado condicional. En la figura 18.3, el segmento en lenguaje de diseño de programa (PDL, por sus siglas en inglés) se traduce en el gráfico de flujo mostrado. Observe que se crea un nodo separado para cada una de las condiciones *a* y *b* en el enunciado IF *a* OR *b*. Cada nodo que contiene una condición se llama *nodo predicado* y se caracteriza por dos o más aristas que emanan de él.

### 18.4.2 Rutas de programa independientes

Una *ruta independiente* es cualquiera que introduce al menos un nuevo conjunto de enunciados de procesamiento o una nueva condición en el programa. Cuando se establece como un gráfico de flujo, una ruta independiente debe moverse a lo largo de al menos una arista que no se haya recorrido antes de definir la ruta. Por ejemplo, un conjunto de rutas independientes para el gráfico de flujo que se ilustra en la figura 18.2*b*) es

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

Observe que cada nueva ruta introduce una nueva arista. La ruta

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

no se considera como independiente porque simplemente es una combinación de rutas ya especificadas y no recorre alguna arista nueva.

Las rutas de la 1 a la 4 constituyen un *conjunto básico* para el gráfico de flujo de la figura 18.2*b*). Es decir, si se pueden diseñar pruebas para forzar la ejecución de estas rutas (un conjunto básico), todo enunciado en el programa tendrá garantizada su ejecución al menos una vez, y cada condición se ejecutará en sus lados verdadero y falso. Debe señalarse que el con-

4 En la sección 18.6.1 se presenta un análisis más detallado de los gráficos y sus usos.

junto básico no es único. De hecho, para un diseño de procedimiento dado, pueden derivarse algunos conjuntos básicos diferentes.

¿Cómo saber cuántas rutas buscar? El cálculo de la complejidad ciclomática proporciona la respuesta. La *complejidad ciclomática* es una medición de software que proporciona una evaluación cuantitativa de la complejidad lógica de un programa. Cuando se usa en el contexto del método de prueba de la ruta básica, el valor calculado por la complejidad ciclomática define el número de rutas independientes del conjunto básico de un programa y le brinda una cota superior para el número de pruebas que debe realizar a fin de asegurar que todos los enunciados se ejecutaron al menos una vez.

**CONSEJO**

*La complejidad ciclomática es una medición útil para predecir aquellos módulos proclives al error. Úsela para planificar las pruebas así como para el diseño de casos de prueba.*

**? ¿Cómo se calcula la complejidad ciclomática?**

La complejidad ciclomática tiene fundamentos en la teoría de gráficos y proporciona una medición de software extremadamente útil. La complejidad se calcula en una de tres formas:

1. El número de regiones del gráfico de flujo corresponde a la complejidad ciclomática.
2. La complejidad ciclomática $V(G)$ para un gráfico de flujo $G$ se define como

   $V(G) = E - N + 2$

   donde $E$ es el número de aristas del gráfico de flujo y $N$ el número de nodos del gráfico de flujo.
3. La complejidad ciclomática $V(G)$ para un gráfico de flujo $G$ también se define como

   $V(G) = P + 1$

   donde $P$ es el número de nodos predicado contenidos en el gráfico de flujo $G$.

**PUNTO CLAVE**

La complejidad ciclomática proporciona la cota superior sobre el número de casos de prueba que se requerirán para garantizar que cada enunciado en el programa se ejecuta al menos una vez.

En el gráfico de flujo de la figura 18.2*b*), la complejidad ciclomática puede calcularse usando cada uno de los algoritmos recién indicados:

1. El gráfico de flujo tiene cuatro regiones.
2. $V(G) = 11$ aristas $- 9$ nodos $+ 2 = 4$.
3. $V(G) = 3$ nodos predicado $+ 1 = 4$.

Por tanto, la complejidad ciclomática del gráfico de flujo en la figura 18.2*b*) es 4.

Más importante, el valor para $V(G)$ proporciona una cota superior para el número de rutas independientes que forman el conjunto básico y, por implicación, una cota superior sobre el número de pruebas que deben diseñarse y ejecutarse para garantizar cobertura de todos los enunciados del programa.

## CasaSegura

### *Uso de la complejidad ciclomática*

**La escena:** Cubículo de Shakira.

**Participantes:** Vinod y Shakira, miembros del equipo de ingeniería del software CasaSegura, quienes trabajan en la planificación de las pruebas para la función de seguridad.

**La conversación:**

**Shakira:** Mira... Sé que debemos hacer pruebas de unidad a todos los componentes para la función de seguridad, pero hay muchos de ellos, y si consideras el número de operaciones que se tienen que revisar, no sé... tal vez deberíamos olvidar la prueba de caja blanca, integrar todo y comenzar a aplicar las pruebas de caja negra.

**Vinod:** ¿Supones que no tenemos tiempo suficiente para hacer pruebas de componentes, revisar las operaciones y luego integrar?

**Shakira:** La fecha límite para el primer incremento está más cerca de lo que quisiera... sí, estoy preocupada.

**Vinod:** ¿Por qué al menos no aplicas pruebas de caja blanca sobre las operaciones que tienen probabilidad de ser más proclives a errores?

**Shakira (exasperada):** ¿Y exactamente cómo sé cuáles son las más proclives a errores?

**Vinod:** *V* de *G*.

**Shakira:** ¿Qué?

**Vinod:** Complejidad ciclomática, *V* de *G*. Sólo calcula *V*(*G*) para cada una de las operaciones dentro de cada uno de los componentes y ve cuáles tienen los valores más altos para *V*(*G*). Ésas son las que tienen más probabilidad de ser proclives a errores.

**Shakira:** ¿Y cómo calculo *V* de *G*?

**Vinod:** Realmente es sencillo. Aquí hay un libro que describe cómo hacerlo.

**Shakira (hojea el libro):** Muy bien, no parece difícil. Lo intentaré. Las operaciones con la *V*(*G*) más altas serán las candidatas para las pruebas de caja blanca.

**Vinod:** Sólo recuerda que no hay garantías. Un componente con una *V*(*G*) baja puede ser proclive a errores.

**Shakira:** Bien. Pero al menos esto me ayudará a reducir el número de componentes que tienen que experimentar pruebas de caja blanca.

### 18.4.3 Derivación de casos de prueba

**Cita:**

"El cohete Ariane 5 estalló en el despegue debido exclusivamente a un defecto de software (un bug, un error) que involucra la conversión de un valor en punto flotante de 64 bits en un entero de 16 bits. El cohete y sus cuatro satélites no estaban asegurados y valían 500 millones de dólares. [Pruebas de ruta que revisaran la ruta de conversión] habrían descubierto el error, pero se vetaron por razones presupuestarias."

**Reporte noticioso**

El método de prueba de ruta básica puede aplicarse a un diseño de procedimientos o a un código fuente. En esta sección se presenta la prueba de ruta básica como una serie de pasos. El procedimiento *average* (promedio), que en la figura 18.4 se muestra en PDL, se usará como ejemplo para ilustrar cada paso del método de diseño de caso de prueba. Observe que *average*, aunque es un algoritmo extremadamente simple, contiene condiciones y bucles compuestos. Es posible aplicar los siguientes pasos para derivar el conjunto básico:

1. **Al usar el diseño o el código como cimiento, dibuje el gráfico de flujo correspondiente.** El gráfico de flujo se crea usando los símbolos y reglas de construcción que se presentaron en la sección 18.4.1. En el PDL para *average* de la figura 18.4, el gráfico de flujo se crea al numerar aquellos enunciados PDL que se mapearán en los nodos correspondientes del gráfico de flujo. En la figura 18.5 se muestra el gráfico de flujo correspondiente.
2. **Determine la complejidad ciclomática del gráfico de flujo resultante.** La complejidad ciclomática *V*(*G*) se determina al aplicar los algoritmos descritos en la sección 18.4.2. Debe observarse que *V*(*G*) puede determinarse sin desarrollar un gráfico de flujo,

**FIGURA 18.4**
**PDL con identificación de nodos**

```
PROCEDIMIENTO average;

    *  Este procedimiento calcula el promedio de 100 o menos
       números que se encuentran entre valores frontera; también
       calcula la suma y el número total válido.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
1   i = 1;
    total.input = total.valid = 0;
    sum = 0;
    DO WHILE value[i] <> –999 AND total.input < 100   3
4      increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum   6
5         THEN increment total.valid by 1;
7                sum = s sum + value[i]
          ELSE skip
8      ENDIF
       increment i by 1;
9   ENDDO
       IF total.valid > 0   10
11        THEN average = sum / total.valid;
12        ELSE average = –999;
13  ENDIF
END average
```

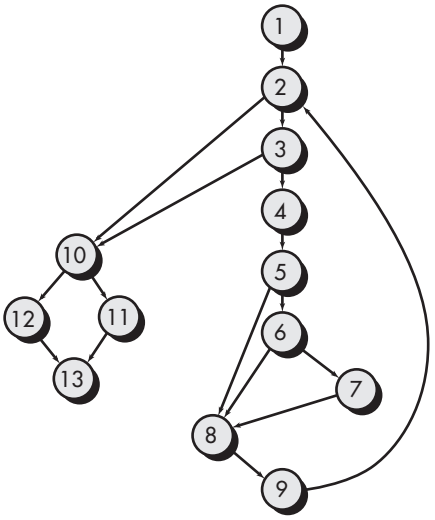



**FIGURA 18.5**
**Gráfico de flujo para el procedimiento *average***

al contar todos los enunciados condicionales en el PDL (para el procedimiento *average*, las condiciones compuestas son dos) y sumar 1. En la figura 18.5,

$V(G)$ = 6 regiones
$V(G)$ = 17 aristas − 13 nodos + 2 = 6
$V(G)$ = 5 nodos predicado + 1 = 6

3. **Determine un conjunto básico de rutas linealmente independientes.** El valor de $V(G)$ proporciona la cota superior sobre el número de rutas linealmente independientes a través de la estructura de control del programa. En el caso del procedimiento *average*, se espera especificar seis rutas:

   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-...

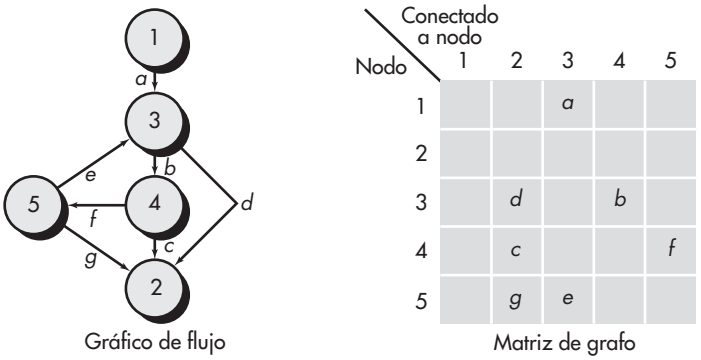
   ruta 6: 1-2-3-4-5-6-7-8-9-2-...

   La elipsis (...) después de las rutas 4, 5 y 6 indica que es aceptable cualquier ruta a través del resto de la estructura de control. Con frecuencia, vale la pena identificar nodos predicado como un auxiliar para la derivación de los casos de prueba. En este caso, los nodos 2, 3, 5, 6 y 10 son nodos predicado.

4. **Prepare casos de prueba que fuercen la ejecución de cada ruta en el conjunto básico.** Los datos deben elegirse de modo que las condiciones en los nodos predicado se establezcan de manera adecuada conforme se prueba cada ruta. Cada caso de prueba se ejecuta y compara con los resultados esperados. Una vez completados todos los casos de prueba, el examinador puede estar seguro de que todos los enunciados del programa se ejecutaron al menos una vez.

Es importante notar que algunas rutas independientes (tomemos por caso la ruta 1 del ejemplo) no pueden probarse en forma individual, es decir, la combinación de datos requerida para recorrer la ruta no puede lograrse en el flujo normal del programa. En tales casos, dichas rutas se prueban como parte de otra prueba de ruta.

**FIGURA 18.6**
Matriz de grafo




### 18.4.4 Matrices de grafo

El procedimiento para derivar el gráfico de flujo e incluso determinar un conjunto de rutas básicas es sensible a la mecanización. Una estructura de datos, llamada *matriz de un grafo,* puede ser bastante útil para desarrollar una herramienta de software que auxilie en la prueba de ruta básica.

Una matriz de grafo es una matriz cuadrada cuyo tamaño (es decir, número de filas y columnas) es igual al número de nodos del gráfico de flujo. Cada fila y columna corresponde a un nodo identificado y las entradas de la matriz corresponden a conexiones (una arista) entre nodos. En la figura 18.6 se muestra un ejemplo simple de gráfico de flujo y su correspondiente matriz de grafo [Bei90].

En esa figura, cada nodo en el gráfico de flujo se identifica mediante números, mientras que cada arista se identifica con letras. Una entrada de letra en la matriz corresponde a una conexión entre dos nodos. Por ejemplo, el nodo 3 se conecta con el nodo 4 mediante la arista *b*.

**? ¿Qué es una matriz de grafo y cómo se le extiende para su uso en las pruebas?**

En este punto, la matriz de grafo no es más que una representación tabular de un gráfico de flujo. Sin embargo, al agregar un enlace ponderado a cada entrada de matriz, la matriz de grafo puede convertirse en una poderosa herramienta para evaluar durante las pruebas la estructura de control del programa. El *enlace ponderado* proporciona información adicional acerca del flujo de control. En su forma más simple, el enlace ponderado es 1 (existe una conexión) o 0 (no existe conexión). Pero a los enlaces ponderados puede asignárseles otras propiedades más interesantes:

- La probabilidad de que un enlace (arista) se ejecutará.
- El tiempo de procesamiento que se emplea durante el recorrido de un enlace.
- La memoria requerida durante el recorrido de un enlace.
- Los recursos requeridos durante el recorrido de una prueba.

Beizer [Bei90] proporciona un tratamiento a fondo de algoritmos matemáticos adicionales que pueden aplicarse a las matrices gráficas. Con estas técnicas, el análisis requerido para diseñar casos de prueba puede ser parcial o completamente automatizado.

## 18.5 PRUEBA DE LA ESTRUCTURA DE CONTROL

La técnica de prueba de ruta básica descrita en la sección 18.4 es una de varias técnicas para probar la estructura de control. Aunque la prueba de ruta básica es simple y enormemente efectiva, no es suficiente en sí misma. En esta sección se estudian otras variaciones acerca de la prueba de la estructura de control. Esta prueba más amplia cubre y mejora la calidad de la prueba de caja blanca.

**Cita:**

"Poner más atención a la aplicación de las pruebas que a su diseño es un error clásico."

**Brian Marick**

### 18.5.1 Prueba de condición

La *prueba de condición* [Tai89] es un método de diseño de casos de prueba que revisa las condiciones lógicas contenidas en un módulo de programa. Una condición simple es una variable booleana o una expresión relacional, posiblemente precedida de un operador NOT (¬). Una expresión relacional toma la forma

$E_1$ <operador relacional> $E_2$

donde $E_1$ y $E_2$ son expresiones aritméticas y <operador relacional> es uno de los siguientes: $<$, $\leq$, $=$, $\neq$ (no igualdad), $>$ o $\geq$. Una *condición compuesta* se integra con dos o más condiciones simples, operadores booleanos y paréntesis. Se supone que los operadores booleanos permitidos en una condición compuesta incluyen OR (|), AND (&) y NOT (¬). Una condición sin expresiones relacionales se conoce como expresión booleana.

**PUNTO CLAVE**

Los errores son mucho más comunes en la cercanía de las condiciones lógicas que en el lugar de los enunciados de procesamiento secuencial.

Si una condición es incorrecta, entonces al menos un componente de la condición es incorrecto. Por tanto, los tipos de errores en una condición incluyen errores de operador booleano (operadores booleanos incorrectos/perdidos/adicionales), de variable booleana, de paréntesis booleanos, de operador relacional y de expresión aritmética. El método de prueba de condición se enfoca en la prueba de cada condición del programa para asegurar que no contiene errores.

**Cita:**

"Los buenos examinadores son maestros para notar 'algo divertido' y actuar sobre ello."

**Brian Marick**

### 18.5.2 Prueba de flujo de datos

El método de prueba de flujo de datos [Fra93] selecciona rutas de prueba de un programa de acuerdo con las ubicaciones de las definiciones y con el uso de variables en el programa. Para ilustrar el enfoque de prueba de flujo de datos, suponga que a cada enunciado en un programa se le asigna un número de enunciado único y que cada función no modifica sus parámetros o variables globales. Para un enunciado con *S* como su número de enunciado,

DEF(*S*) = {*X* | enunciado *S* contiene una definición de *X*}
USE(*S*) = {*X* | enunciado *S* contiene un uso de *X*}

Si el enunciado *S* es un *enunciado if* o *loop*, su conjunto DEF es vacío y su conjunto USE se basa en la condición del enunciado *S*. Se dice que la definición de la variable *X* en el enunciado *S* está *viva* en el enunciado *S*′ si existe una ruta desde el enunciado *S* hasta el enunciado *S*′ que no contiene otra definición de *X*.

**CONSEJO**

*Es irreal suponer que la prueba de flujo de datos se usará de manera extensa cuando se prueba un sistema grande. Sin embargo, puede usarse en forma dirigida para áreas de software que sean sospechosas.*

Una *cadena de definición de uso (DU)* de la variable *X* es de la forma [*X, S, S*′], donde *S* y *S*′ son números de enunciado, *X* está en DEF(*S*) y en USE(*S*′), y la definición de *X* en el enunciado *S* está viva en el enunciado *S*′.

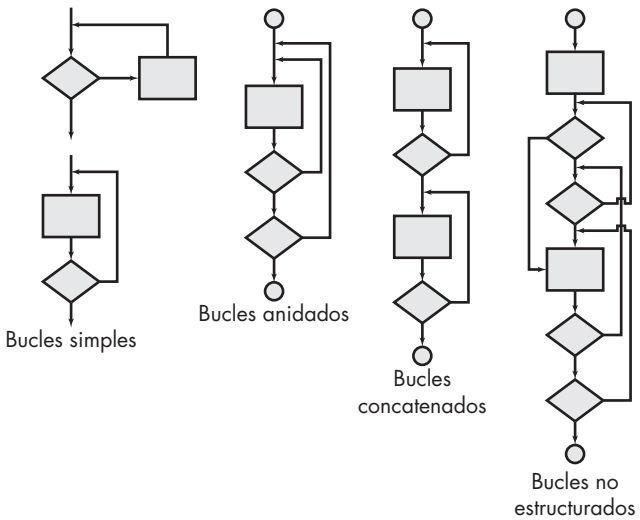
Una estrategia de prueba de flujo de datos simple es requerir que toda cadena DU se cubra al menos una vez. A esta estrategia se le conoce como estrategia de prueba DU. Se ha demostrado que la prueba DU no garantiza la cobertura de todas las ramas de un programa. Sin embargo, la prueba DU no garantiza la cobertura de una rama sólo en raras situaciones, como en los constructos if-then-else en los cuales la *parte then* no tiene definición de alguna variable y la *parte else* no existe. En esta situación, la rama *else* del enunciado *if* no necesariamente se cubre con la prueba DU.

### 18.5.3 Prueba de bucle

Los bucles son la piedra de toque de la gran mayoría de todos los algoritmos implementados en el software. Y aún así, con frecuencia se les pone poca atención mientras se realizan las pruebas de software.

La *prueba de bucle* es una técnica de prueba de caja blanca que se enfoca exclusivamente en la validez de los constructos bucle. Pueden definirse cuatro clases diferentes de bucles [Bei90]: simples, concatenados, anidados y no estructurados (figura 18.7).

**FIGURA 18.7**
**Clases de bucles**




**Bucles simples.** El siguiente conjunto de pruebas puede aplicarse a los bucles simples, donde *n* es el máximo número de pasadas permisibles a través del bucle.

1. Saltar por completo el bucle.
2. Sólo una pasada a través del bucle.
3. Dos pasadas a través del bucle.
4. *m* pasadas a través del bucle, donde $m < n$.
5. $n - 1$, $n$, $n + 1$ pasadas a través del bucle.

**Bucles anidados.** Si tuviera que extender el enfoque de la prueba para bucles simples a los bucles anidados, el número de pruebas posibles crecería geométricamente conforme el nivel de anidado aumenta. Esto daría como resultado un número impráctico de pruebas. Beizer [Bei90] sugiere un acercamiento que ayudará a reducir el número de pruebas:

1. Comience con el bucle más interno. Establezca todos los otros bucles a valores mínimos.
2. Realice pruebas de bucle simple para el bucle más interno mientras mantiene los bucles exteriores en sus valores mínimos de parámetro de iteración (por ejemplo, contador de bucle). Agregue otras pruebas para valores fuera-de-rango o excluidos.
3. Trabaje hacia afuera y realice pruebas para el siguiente bucle, pero mantenga los otros bucles exteriores en valores mínimos y los otros bucles anidados en valores "típicos".
4. Continúe hasta que todos los bucles se hayan probado.


*No es posible probar los bucles no estructurados de manera efectiva. Se deben refactorizar.*

**Bucles concatenados.** Los bucles concatenados pueden probarse usando el enfoque definido para bucles simples si cada uno de los bucles es independiente de los otros. No obstante, si dos bucles se concatenan y usa el contador de bucle para el bucle 1 como el valor inicial para el bucle 2, entonces los bucles no son independientes. Cuando los bucles no son independientes, se recomienda el enfoque aplicado a bucles anidados.

**Bucles no estructurados.** Siempre que sea posible, esta clase de bucles debe rediseñarse para reflejar el uso de los constructos de programación estructurada (capítulo 10).

## 18.6 PRUEBAS DE CAJA NEGRA

Las *pruebas de caja negra*, también llamadas *pruebas de comportamiento*, se enfocan en los requerimientos funcionales del software; es decir, las técnicas de prueba de caja negra le permiten derivar conjuntos de condiciones de entrada que revisarán por completo todos los requerimientos funcionales para un programa. Las pruebas de caja negra no son una alternativa para las técnicas de caja blanca. En vez de ello, es un enfoque complementario que es probable que descubra una clase de errores diferente que los métodos de caja blanca.

Las pruebas de caja negra intentan encontrar errores en las categorías siguientes: 1) funciones incorrectas o faltantes, 2) errores de interfaz, 3) errores en las estructuras de datos o en el acceso a bases de datos externas, 4) errores de comportamiento o rendimiento y 5) errores de inicialización y terminación.

**Cita:**

"Errar es humano, pero encontrar un error es divino."

**Robert Dunn**

A diferencia de las pruebas de caja blanca, que se realizan tempranamente en el proceso de pruebas, la prueba de caja negra tiende a aplicarse durante las últimas etapas de la prueba (vea el capítulo 17). Puesto que, a propósito, la prueba de caja negra no considera la estructura de control, la atención se enfoca en el dominio de la información. Las pruebas se diseñan para responder a las siguientes preguntas:

**¿Qué preguntas responden las pruebas de caja negra?**

- ¿Cómo se prueba la validez funcional?
- ¿Cómo se prueban el comportamiento y el rendimiento del sistema?
- ¿Qué clases de entrada harán buenos casos de prueba?
- ¿El sistema es particularmente sensible a ciertos valores de entrada?
- ¿Cómo se aíslan las fronteras de una clase de datos?
- ¿Qué tasas y volumen de datos puede tolerar el sistema?
- ¿Qué efecto tendrán sobre la operación del sistema algunas combinaciones específicas de datos?

Al aplicar las técnicas de caja negra, se deriva un conjunto de casos de prueba que satisfacen los siguientes criterios [Mye79]: 1) casos de prueba que reducen, por una cuenta que es mayor que uno, el número de casos de prueba adicionales que deben diseñarse para lograr pruebas razonables y 2) casos de prueba que dicen algo acerca de la presencia o ausencia de clases de errores, en lugar de un error asociado solamente con la prueba específica a mano.

### 18.6.1 Métodos de prueba basados en gráficos

**PUNTO CLAVE**

Una gráfica representa las relaciones entre objetos datos y objetos programa, lo que permite derivar casos de prueba que buscan errores asociados con dichas relaciones.

El primer paso en la prueba de caja negra es entender los objetos[5] que se modelan en software y las relaciones que conectan a dichos objetos. Una vez logrado esto, el siguiente paso es definir una serie de pruebas que verifiquen "que todos los objetos tengan la relación mutua esperada" [Bei95]. Dicho de otra forma, la prueba de software comienza con la creación de un gráfico de objetos importantes y sus relaciones, y luego diseña una serie de pruebas que cubrirán el gráfico, de modo que cada objeto y relación se revise y se descubran errores.

Para lograr estos pasos, comience por crear un *gráfico*: una colección de *nodos* que representen objetos, *enlaces* que representen las relaciones entre objetos, *nodos ponderados* que describan las propiedades de un nodo (por ejemplo, un valor de datos o comportamiento de estado específicos) y *enlaces ponderados* que describan alguna característica de un enlace.

En la figura 18.8*a*) se muestra la representación simbólica de un gráfico. Los nodos se representan como círculos conectados mediante ligas que tienen algunas formas diferentes. Un en-

5 En este contexto, debe considerar el término *objetos* en el contexto más amplio posible. Abarca objetos de datos, componentes tradicionales (módulos) y elementos orientados a objeto del software de cómputo.

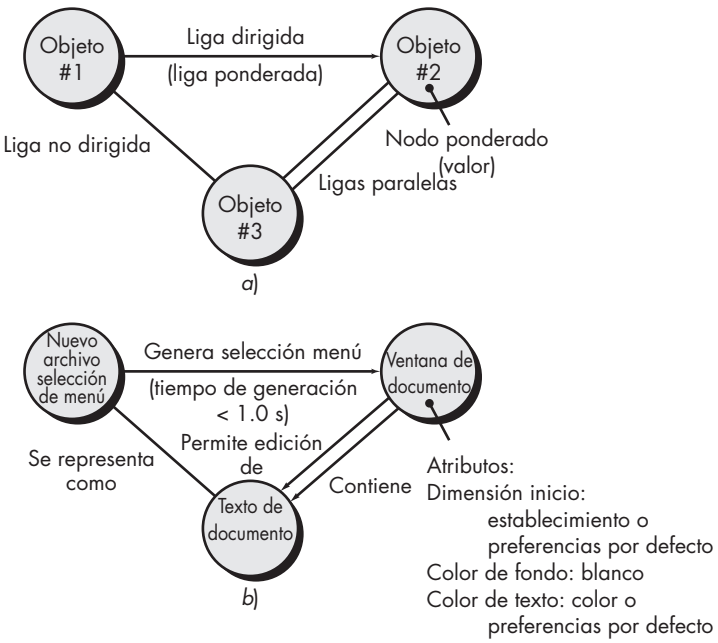



**FIGURA 18.8**
***a*) Notación de gráfico; *b*) ejemplo simple**

lace dirigido (representado mediante una flecha) indica que una relación sólo se mueve en una dirección. Un *enlace bidireccional*, también llamado *enlace simétrico*, implica que la relación se aplica en ambas direcciones. Los *enlaces paralelos* se usan cuando entre los nodos gráficos se establecen algunas relaciones diferentes.

Como ejemplo simple, considere una porción de un gráfico para una aplicación de un procesador de palabras (figura 18.8*b*) donde

*Objeto #1* = **newFile** (selección de menú)

*Objeto #2* = **documentWindow**

*Objeto #3* = **documentText**

En la figura, una selección de menú en **newFile** genera una ventana de documento. El nodo ponderado de **documentWindow** proporciona una lista de los atributos de ventana que se esperan cuando se genere la ventana. El enlace ponderado indica que la ventana debe generarse en menos de 1.0 segundo. Un enlace no dirigido establece una relación simétrica entre la selección de menú **newFile** y **documentText**, y los enlaces paralelos indican relaciones entre **documentWindow** y **documentText**. En realidad, tendría que generarse un gráfico más detallado como precursor para el diseño de casos de prueba. Entonces podrían derivarse casos de prueba al recorrer el gráfico y cubrir cada una de las relaciones mostradas. Dichos casos de prueba se designan con la intención de encontrar errores en alguna de las relaciones. Beizer [Bei95] describe algunos métodos de prueba de comportamiento que pueden usar gráficos:

**Modelado de flujo de transacción.** Los nodos representan pasos en alguna transacción (por ejemplo, los pasos requeridos para hacer una reservación en una aerolínea con el uso de un servicio en línea) y los enlaces representan la conexión lógica entre los pasos (por ejemplo, **ingresarInformaciónVuelo** se sigue de *validaciónProcesamientoDisponibilidad*). El diagrama de flujo de datos (capítulo 7) puede usarse para auxiliar en la creación de gráficos de este tipo.

**Modelado de estado finito.** Los nodos representan diferentes estados del software observables por el usuario (por ejemplo, cada una de las "pantallas" que aparecen cuando un

empleado ingresa información conforme toma una orden telefónica) y los enlaces representan las transiciones que ocurren para moverse de estado a estado (por ejemplo, **pedidoInformación** se verifica durante *inventarioBusquedaDisponibilidad*, y es seguido de la entrada **clienteFacturaInformación**). El diagrama de estado (capítulo 7) puede usarse para auxiliar en la creación de gráficos de este tipo.

**Modelado de flujo de datos.** Los nodos son objetos datos y los enlaces son las transformaciones que ocurren para traducir un objeto datos en otro. Por ejemplo, el nodo retención de impuesto FICA (**FTW**) se calcula a partir de los ingresos brutos (**IB**), usando la relación **FTW = 0.62 × IB**.

**Modelado de temporización.** Los nodos son objetos programa y los enlaces son las conexiones secuenciales entre dichos objetos. Los enlaces ponderados se usan para especificar los tiempos de ejecución requeridos conforme se ejecuta el programa.

Un análisis detallado de cada uno de estos métodos de prueba basados en gráfico está más allá del ámbito de este libro. Si se tiene mayor interés, consulte [Bei95] para conocer una cobertura más amplia.

### 18.6.2 Partición de equivalencia


*Las clases de entrada se conocen relativamente pronto en el proceso del software. Por esta razón, se debe pensar acerca de la partición de equivalencia conforme se crea el diseño.*

La *partición de equivalencia* es un método de prueba de caja negra que divide el dominio de entrada de un programa en clases de datos de los que pueden derivarse casos de prueba. Un caso de prueba ideal descubre de primera mano una clase de errores (por ejemplo, procesamiento incorrecto de todos los datos carácter) que de otro modo podrían requerir la ejecución de muchos casos de prueba antes de observar el error general.

El diseño de casos de prueba para la partición de equivalencia se basa en una evaluación de las *clases de equivalencia* para una condición de entrada. Con los conceptos introducidos en la sección precedente, si un conjunto de objetos puede vincularse mediante relaciones que son simétricas, transitivas y reflexivas, se presenta una clase de equivalencia [Bei95]. Una clase de equivalencia representa un conjunto de estados válidos o inválidos para condiciones de entrada. Por lo general, una 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 pueden definirse de acuerdo con los siguientes lineamientos:

**? ¿Cómo se definen las clases de equivalencia para pruebas?**

1. Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos inválidas.
2. Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos inválidas.
3. Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y una inválida.
4. Si una condición de entrada es booleana, se define una clase válida y una inválida.

Al aplicar los lineamientos para la derivación de clases de equivalencia, pueden desarrollarse y ejecutarse los casos de prueba para cada ítem de datos del dominio de entrada. Los casos de prueba se seleccionan de modo que se revise a la vez el número más grande de atributos de una clase de equivalencia.

**Cita:**

"Una forma efectiva para probar código es revisarlo en sus fronteras naturales."

**Brian Kernighan**

### 18.6.3 Análisis de valor de frontera

Un mayor número de errores ocurre en las fronteras del dominio de entrada y no en el "centro". Por esta razón es que el *análisis de valor de frontera* (BVA, del inglés *boundary value analysis*) se desarrolló como una técnica de prueba. El análisis de valor de frontera conduce a una selección de casos de prueba que revisan los valores de frontera.

El análisis de valor de frontera es una técnica de diseño de casos de prueba que complementan la partición de equivalencia. En lugar de seleccionar algún elemento de una clase de equivalencia, el BVA conduce a la selección de casos de prueba en los "bordes" de la clase. En lugar de enfocarse exclusivamente en las condiciones de entrada, el BVA también deriva casos de prueba a partir del dominio de salida [Mye79].

Los lineamientos para el BVA son similares en muchos aspectos a los proporcionados para la partición de equivalencia:

**PUNTO CLAVE**

El BVA extiende la partición de equivalencia al enfocarse en datos en los "bordes" de una clase de equivalencia.

1. Si una condición de entrada especifica un rango acotado por valores *a* y *b*, los casos de prueba deben designarse con valores *a* y *b*, justo arriba y justo abajo de *a* y *b*.
2. Si una condición de entrada especifica un número de valores, deben desarrollarse casos de prueba que revisen los números mínimo y máximo. También se prueban los valores justo arriba y abajo, mínimo y máximo.
3. Aplicar lineamientos 1 y 2 a condiciones de salida. Por ejemplo, suponga que como salida de un programa de análisis de ingeniería se requiere una tabla de temperatura contra presión. Deben diseñarse casos de prueba para crear un reporte de salida que produzca el número máximo (y mínimo) permisible de entradas de tabla.
4. Si las estructuras de datos de programa internos tienen fronteras prescritas (por ejemplo, una tabla que tenga un límite definido de 100 entradas), asegúrese de diseñar un caso de prueba para revisar la estructura de datos en su frontera.

La mayoría de los ingenieros de software realizan intuitivamente BVA en cierta medida. Al aplicar dichos lineamientos, la prueba de fronteras será más completa y, por tanto, tendrá una mayor probabilidad de detectar errores.

### 18.6.4 Prueba de arreglo ortogonal

Existen muchas aplicaciones en las cuales el dominio de entrada es relativamente limitado, es decir, el número de parámetros de entrada es pequeño y los valores que cada uno de los parámetros puede tomar están claramente acotados. Cuando dichos números son muy pequeños (por ejemplo, tres parámetros de entrada que toman tres valores discretos cada uno), es posible considerar cada permutación de entrada y probar de manera exhaustiva el dominio de entrada. Sin embargo, conforme crece el número de valores de entrada y el número de valores discretos para cada ítem de datos, la prueba exhaustiva se vuelve impráctica o imposible.

La *prueba de arreglo ortogonal* puede aplicarse a problemas en los que el dominio de entrada es relativamente pequeño pero demasiado grande para alojar la prueba exhaustiva. El método de prueba de arreglo ortogonal es particularmente útil para encontrar los *fallos de región*, una categoría de error asociada con lógica defectuosa dentro de un componente de software.

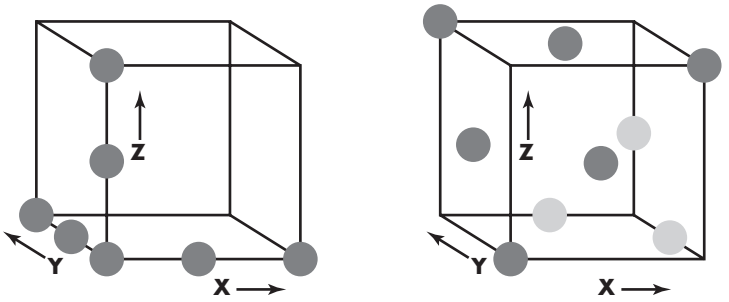
**PUNTO CLAVE**

La prueba de arreglo ortogonal permite diseñar casos de prueba que proporcionan cobertura máxima de prueba con un número razonable de casos de prueba.

Para ilustrar la diferencia entre prueba de arreglo ortogonal y enfoques más convencionales del tipo "un ítem de entrada a la vez", piense en un sistema que tiene tres ítems de entrada, *X*, *Y* y *Z*. Cada uno tiene tres valores discretos asociados consigo. Existen $3^3 = 27$ posibles casos de prueba. Phadke [Pha97] sugiere una visión geométrica de los posibles casos de prueba asociados con X, Y y Z, que se ilustra en la figura 18.9. En la figura, un ítem de entrada a la vez puede variar en secuencia a lo largo de cada eje de entrada. Esto da como resultado cobertura relativamente limitada del dominio de entrada (representado por el cubo de la izquierda en la figura).

Cuando ocurre la prueba de arreglo ortogonal, se crea un *arreglo ortogonal* L9 de casos de prueba. El arreglo ortogonal L9 tiene una "propiedad de equilibrio" [Pha97]. Es decir, los casos de prueba (representados con puntos oscuros en la figura) se "dispersan de manera uniforme a lo largo de todo el dominio de prueba", como se ilustra en el cubo de la derecha en la figura 18.9. La cobertura de prueba a través del dominio de entrada es más completa.

**FIGURA 18.9**
**Visión geométrica de casos de prueba**
**Fuente: [Pha97]**




Para ilustrar el uso del arreglo ortogonal L9, considere la función *send* para una aplicación de fax. A la función *send* pasan cuatro parámetros: P1, P2, P3 y P4. Cada uno toma tres valores discretos. Por ejemplo, P1 toma los valores:

P1 = 1, enviar ahora
P1 = 2, enviar una hora más tarde
P1 = 3, enviar después de medianoche

P2, P3 y P4 también tomarían los valores de 1, 2 y 3, que significan otras funciones de envío.

Si se eligiera la estrategia de prueba "un ítem de entrada a la vez", la siguiente secuencia de pruebas (P1, P2, P3, P4) se especificaría: (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] valora estos casos de prueba al afirmar:

> Tales casos de prueba son útiles sólo cuando uno está seguro de que estos parámetros de prueba no interactúan. Pueden detectar fallas lógicas donde un solo valor de parámetro genere mal funcionamiento del software. Estas fallas se llaman *fallos de modo individual*. Este método no puede detectar fallos lógicos que causen mal funcionamiento cuando dos o más parámetros toman simultáneamente ciertos valores; es decir, no pueden detectar todas las interacciones. Por tanto, su habilidad para detectar fallas es limitada.

Dado el número relativamente pequeño de los parámetros de entrada y de los valores discretos, es posible la prueba exhaustiva. El número de pruebas requeridas es $3^4 = 81$, grande pero manejable. Se encontrarían todos los fallos asociados con la permutación de ítems de datos, pero el esfuerzo requerido es relativamente elevado.

El enfoque de prueba de arreglo ortogonal permite proporcionar una buena cobertura de pruebas con muchos menos casos de prueba que la estrategia exhaustiva. En la figura 18.10 se ilustra un arreglo ortogonal L9 para la función *send* de fax.

Phadke [Pha97] valora el resultado de las pruebas usando el arreglo ortogonal L9 en la siguiente forma:

> **Detectar y aislar todos los fallos de modo individual.** Un fallo de modo individual es un problema congruente con cualquier nivel de cualquier parámetro individual. Por ejemplo, si todos los casos de prueba del factor P1 = 1 causan una condición de error, se trata de una falla de modo individual. En este ejemplo, las pruebas 1, 2 y 3 [figura 18.10] mostrarán errores. Al analizar la información acerca de qué pruebas muestran errores, uno puede identificar cuáles valores de parámetro causan el fallo. En este ejemplo, al notar que las pruebas 1, 2 y 3 causan un error, uno puede aislar [procesamiento lógico asociado con "enviar ahora" (P1 = 1)] la fuente del error. Tal aislamiento del fallo es importante para corregirlo.
>
> **Detectar todos los fallos de modo doble.** Si existe un problema consistente cuando ocurren en conjunto niveles específicos de dos parámetros, se le llama *fallo de modo doble*. De hecho, un fallo

**FIGURA 18.10**
**Un arreglo ortogonal L9**

| Caso de prueba | Parámetros de prueba | | | |
|---|---|---|---|---|
| | P1 | P2 | P3 | P4 |
| 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 |

de modo doble es indicio de incompatibilidad pareada o de interacciones dañinas entre dos parámetros de prueba.

**Fallos multimodo.** Los arreglos ortogonales [del tipo mostrado] sólo pueden garantizar la detección de fallos de modo individual y doble. No obstante, muchos fallos multimodo también son detectables por estas pruebas.

En [Pha89] se puede encontrar un análisis detallado de la prueba de arreglo ortogonal.

**HERRAMIENTAS DE SOFTWARE**

### *Diseño de casos de prueba*

**Objetivo:** Auxiliar al equipo de software en el desarrollo de un conjunto completo de casos de prueba tanto para prueba de caja negra como de caja blanca.

**Mecánica:** Estas herramientas se clasifican en dos categorías amplias: las herramientas de prueba estáticas y las herramientas de prueba dinámicas. En la industria se utilizan tres diferentes tipos de herramientas de prueba estáticas: herramientas de prueba basadas en código, lenguajes de prueba especializados y herramientas de prueba basadas en requerimientos. Las primeras aceptan código fuente como entrada y realizan algunos análisis que dan como resultado la generación de casos de prueba. Los lenguajes de prueba especializados (por ejemplo, ATLAS) permiten al ingeniero de software escribir especificaciones de prueba detalladas que describen cada caso de prueba y la logística para su ejecución. Las herramientas de prueba basadas en requerimientos aíslan requerimientos de usuario específicos y sugieren casos de prueba (o clases de pruebas) que revisarán los requerimientos. Las herramientas de prueba dinámicas interactúan con un programa en ejecución, comprueban la cobertura de ruta, prueban las afirmaciones acerca del valor de variables específicas e instrumentan el flujo de ejecución del programa.

**Herramientas representativas:**[6]

*McCabeTest*, desarrollada por McCabe & Associates (**www.mccabe.com**), implementa una variedad de técnicas de prueba de trayectoria derivadas de una valoración de complejidad ciclomática y de otras mediciones de software.

*TestWorks*, desarrollada por Software Research, Inc. (**www.soft.com/Products**), es un conjunto completo de herramientas de prueba automatizadas que auxilia en el diseño de casos de prueba para software desarrollado en C/C++ y Java, y que proporciona apoyo para pruebas de regresión.

*T-VEC Test Generation System*, desarrollada por T-VEC Technologies (**www.t-vec.com**), es un conjunto de herramientas que soportan pruebas de unidad, integración y validación al asistir en el diseño de casos de prueba, usando la información contenida en una especificación de requerimientos OO.

*e-TEST Suite*, desarrollada por Empirix, Inc. (**www.empirix.com**), abarca un conjunto completo de herramientas para probar *webapps*, incluidas herramientas que auxilian en el diseño de casos de prueba y planificación de pruebas.

6 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que existen en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

## 18.7 PRUEBA BASADA EN MODELO

**Cita:**

"Es suficientemente difícil encontrar un error en el código cuando se le busca; pero es todavía más difícil cuando se supone que el código está libre de errores."

**Steve McConnell**

La *prueba basada en modelo* (PBM) es una técnica de prueba de caja negra que usa la información contenida en el modelo de requerimientos como la base para la generación de casos de prueba. En muchos casos, la técnica de prueba basada en modelo usa diagramas de estado UML, un elemento del modelo de comportamiento (capítulo 7), como la base para el diseño de los casos de prueba.[7] La técnica PBM requiere cinco pasos:

1. **Analizar un modelo de comportamiento existente para el software o crear uno.** Recuerde que un *modelo de comportamiento* indica cómo responderá el software a los eventos o estímulos externos. Para crear el modelo, debe realizar los pasos expuestos en el capítulo 7: 1) evaluar todos los casos de uso para comprender por completo la secuencia de interacción dentro del sistema, 2) identificar los eventos que impulsen la secuencia de interacción y entender cómo dichos eventos se relacionan con objetos específicos, 3) crear una secuencia para cada caso de uso, 4) construir un diagrama de estado UML para el sistema (por ejemplo, véase la figura 7.6), y 5) revisar el modelo de comportamiento para verificar precisión y congruencia.
2. **Recorrer el modelo de comportamiento y especificar las entradas que forzarán al software a realizar la transición de estado a estado.** Las entradas dispararán eventos que harán que ocurra la transición.
3. **Revisar el modelo de comportamiento y observar las salidas esperadas, conforme el software realiza la transición de estado a estado.** Recuerde que cada transición de estado se dispara mediante un evento y que, como consecuencia de la transición, se invoca alguna función y se crean salidas. Para cada conjunto de entradas (casos de prueba) especificado en el paso 2, las salidas esperadas se especifican como se caracterizan en el modelo de comportamiento. "Una suposición fundamental de esta prueba es que existe cierto mecanismo, un *oráculo de prueba*, que determinará si los resultados de una prueba de ejecución son o no correctos" [DAC03]. En esencia, un oráculo de prueba establece la base para cualquier determinación de lo correcto de la salida. En la mayoría de los casos, el oráculo es el modelo de requerimientos, pero también podría ser otro documento o aplicación, datos registrados en cualquier otro lado o, incluso, un experto humano.
4. **Ejecutar los casos de prueba.** Las pruebas pueden ejecutarse manualmente o crearse y ejecutarse un guión de prueba usando una herramienta de prueba.
5. **Comparar los resultados reales y esperados y adoptar una acción correctiva según se requiera.**

La PBM ayuda a descubrir errores en el comportamiento del software y, como consecuencia, es extremadamente útil cuando se prueban aplicaciones impulsadas por un evento.

## 18.8 PRUEBA PARA ENTORNOS, ARQUITECTURAS Y APLICACIONES ESPECIALIZADOS

En ocasiones, los lineamientos y enfoques únicos para pruebas se garantizan cuando se consideran entornos, arquitecturas y aplicaciones especializados. Aunque las técnicas de prueba estudiadas anteriormente en este capítulo, y en los capítulos 19 y 20, con frecuencia pueden

7 La prueba basada en modelo también puede usarse cuando los requerimientos del software se representan con tablas de decisión, gramáticas o cadenas de Markov [DAC03].

adaptarse a situaciones especializadas, vale la pena considerar individualmente sus necesidades únicas.

### 18.8.1 Pruebas de interfaces gráficas de usuario

Las interfaces gráficas para usuario (GUI, por sus siglas en inglés) presentan interesantes retos de prueba. Puesto que los componentes reutilizables ahora son parte común en los entornos de desarrollo GUI, la creación de la interfaz para el usuario se ha vuelto menos consumidora de tiempo y más precisa (capítulo 11). Pero, al mismo tiempo, la complejidad de las GUI ha crecido, lo que conduce a más dificultad en el diseño y ejecución de los casos de prueba.

Debido a que muchas GUI modernas tienen la misma apariencia y ambiente, puede derivarse una serie de pruebas estándar. Es posible usar las gráficas de modelado de estado finito para derivar una serie de pruebas que aborden objetos de datos y programa específicos que sean relevantes para la GUI. Esta técnica de prueba basada en modelo se estudió en la sección 18.7.

Como producto del gran número de permutaciones asociadas con las operaciones de la GUI, la prueba de GUI debe abordarse usando herramientas automatizadas. Durante los últimos años apareció una amplia gama de herramientas de prueba GUI.[8]

### 18.8.2 Prueba de arquitecturas cliente-servidor

**Cita:**

"El tema de las pruebas es un área en la que existe una buena cantidad de comunión entre los sistemas tradicionales y los sistemas cliente-servidor."

**Kelley Bourne**

La naturaleza distribuida de los entornos cliente-servidor, los conflictos de rendimiento asociados con el procesamiento de transacciones, la potencial presencia de algunas plataformas de hardware diferentes, las complejidades de la comunicación en red, la necesidad de atender a múltiples clientes desde una base de datos centralizada (o en algunos casos, distribuida) y los requerimientos de coordinación impuestos al servidor se combinan para realizar las pruebas de las arquitecturas cliente-servidor y el software que reside dentro de ellas es considerablemente más difícil que las aplicaciones independientes. De hecho, estudios industriales recientes indican un aumento significativo en el tiempo y costo de las pruebas cuando se desarrollan los entornos cliente-servidor.

**WebRef**

En **www.csst-technologies.com** pueden encontrarse información y recursos útiles para pruebas cliente-servidor.

En general, la prueba del software cliente-servidor ocurre en tres niveles diferentes: 1) las aplicaciones cliente individuales se prueban en un modo "desconectado"; no se considera la operación del servidor ni la red subyacente. 2) El software cliente y las aplicaciones servidor asociadas se prueban en concierto, pero las operaciones de red no se revisan de manera explícita. 3) Se prueba la arquitectura cliente-servidor completa, incluidos la operación de red y el rendimiento.

Aunque en cada uno de estos niveles de detalle se realizan muchos tipos de pruebas diferentes, para las aplicaciones cliente-servidor se encuentran comúnmente los siguientes abordajes de prueba:

**? ¿Qué tipos de pruebas se llevan a cabo para los sistemas cliente-servidor?**

- **Pruebas de función de aplicación.** La funcionalidad de las aplicaciones cliente se prueba usando los métodos analizados anteriormente en este capítulo y en los capítulos 19 y 20. En esencia, la aplicación se prueba en forma independiente con la intención de descubrir errores en su operación.
- **Pruebas de servidor.** Se prueban las funciones de coordinación y gestión de datos del servidor. También se considera el rendimiento del servidor (tiempo de respuesta global y cantidad de datos transmitidos).
- **Pruebas de base de datos.** Se prueban la precisión y la integridad de los datos almacenados por el servidor. Se examinan las transacciones colocadas por las aplicaciones

8 Cientos, si no miles, de recursos acerca de herramientas de prueba GUI pueden evaluarse en la web. Un buen punto de partida para herramientas de fuente abierta es **www.opensourcetesting.org/functional.php**.

cliente para asegurar que los datos se almacenen, actualicen y recuperen de manera adecuada. También se prueba la forma de archivar.

- **Pruebas de transacción.** Se crea una serie de pruebas para garantizar que cada clase de transacciones se procese de acuerdo con los requerimientos. Las pruebas se enfocan en comprobar lo correcto del procesamiento y también en los conflictos de rendimiento (por ejemplo, tiempos de procesamiento de transacción y volumen de transacción).
- **Pruebas de comunicación de red.** Estas pruebas verifican que la comunicación entre los nodos de la red ocurre de manera correcta y que el mensaje que pasa, las transacciones y el tráfico de red relacionado ocurren sin errores. Como parte de estas pruebas, también pueden realizarse pruebas de seguridad de red.

Para lograr estos abordajes de prueba, Musa [Mus93] recomienda el desarrollo de *perfiles operativos* derivados de escenarios de uso cliente-servidor.[9] Un perfil operativo indica cómo interactúan con el sistema cliente-servidor diferentes tipos de usuarios. Es decir, los perfiles proporcionan un "patrón de uso" que puede aplicarse cuando las pruebas se diseñan y ejecutan. Por ejemplo, para un tipo particular de usuario, ¿qué porcentaje de transacciones serán consultas?, ¿cuántas serán actualizaciones?, ¿cuántos serán pedidos?

Para desarrollar el perfil operativo, es necesario derivar un conjunto de escenarios que sean similares a los casos de uso (capítulos 5 y 6). Cada escenario aborda quién, dónde, qué y por qué. Es decir: quién es el usuario, dónde (en la arquitectura cliente-servidor física) ocurre la interacción del sistema, cuál es la transacción y por qué ocurre. Los escenarios pueden derivarse usando técnicas de respuesta a requerimientos (capítulo 5) o a través de análisis menos formales con los usuarios finales. Sin embargo, el resultado debe ser el mismo. Cada escenario debe proporcionar un indicio de las funciones del sistema que se requerirán para atender a un usuario particular, el orden en el que se requieren dichas funciones, la temporización y la respuesta que se espera, así como la frecuencia con la que se usa cada función. Luego, estos datos se combinan (para todos los usuarios) a fin de crear el perfil operativo. En general, el esfuerzo de prueba y el número de casos de prueba por ejecutar se asignan a cada escenario de uso con base en la frecuencia de uso y en lo crítico de las funciones realizadas.

### 18.8.3 Documentación de prueba y centros de ayuda

El término *prueba de software* invoca imágenes de gran número de casos de prueba preparados para revisar los programas de cómputo y los datos que manipulan. Al recordar la definición de software que se presentó en el capítulo 1, es importante notar que las pruebas también deben extenderse al tercer elemento de la configuración del software: la documentación.

Los errores en la documentación pueden ser tan devastadores para la aceptación del programa como los errores en los datos o en el código fuente. Nada es más frustrante que seguir con exactitud una guía de usuario o un centro de ayuda en línea y obtener resultados o comportamientos que no coinciden con los predichos por la documentación. Por esta razón, las pruebas de documentación deben ser parte significativa de todo plan de prueba de software.

La prueba de documentación puede abordarse en dos fases. La primera, la revisión técnica (capítulo 15), examina el documento en su claridad editorial. La segunda, prueba en vivo, usa la documentación en conjunto con el programa real.

Sorprendentemente, una prueba en vivo para la documentación puede abordarse usando técnicas que son análogas a muchos de los métodos de prueba de caja negra estudiados anteriormente. La prueba basada en gráfico puede usarse para describir el uso del programa; la partición de equivalencia y el análisis del valor de frontera pueden usarse para definir varias

9 Debe señalarse que los perfiles operativos pueden usarse para probar todo tipo de arquitecturas de sistema, no sólo arquitectura cliente-servidor.

clases de entrada e interacciones asociadas. La PBM puede usarse para garantizar que el comportamiento documentado y el comportamiento real coinciden. Entonces, el uso del programa puede rastrearse a través de la documentación.

INFORMACIÓN

### *Pruebas de documentación*

Las siguientes preguntas deben responderse durante las pruebas de documentación y/o en el centro de ayuda:

- ¿La documentación describe con precisión cómo lograr cada modo de uso?
- ¿La descripción de cada secuencia de interacción es precisa?
- ¿Los ejemplos son precisos?
- ¿La terminología, descripciones de menú y respuestas del sistema son consistentes con el programa real?
- ¿Es relativamente fácil localizar guías dentro de la documentación?
- ¿La solución de problemas puede lograrse con facilidad usando la documentación?
- ¿La tabla de contenido y el índice del documento son consistentes, precisos y completos?
- ¿El diseño del documento (plantilla, fuentes, sangrías, gráficos) contribuye a comprender y asimilar rápidamente la información?
- ¿Todos los mensajes de error del software que se muestran al usuario se describen con más detalle en el documento? ¿Las acciones por tomar como consecuencia de un mensaje de error se delinean con claridad?
- Si se usan enlaces de hipertexto, ¿son precisos y completos?
- Si se usa hipertexto, ¿el diseño de navegación es apropiado para la información requerida?

La única forma viable para responder estas preguntas es hacer que una tercera parte independiente (por ejemplo, usuarios seleccionados) pruebe la documentación en el contexto del uso del programa. Todas las discrepancias se anotan y las áreas de ambigüedad o debilidad en el documento se definen para su potencial reescritura.

### 18.8.4 Prueba para sistemas de tiempo real

La naturaleza asíncrona, dependiente del tiempo de muchas aplicaciones de tiempo real, agrega un nuevo y potencialmente difícil elemento a la mezcla de pruebas: el tiempo. El diseñador de casos de prueba no sólo debe considerar los casos de prueba convencionales, sino también la manipulación de eventos (es decir, el procesamiento de interrupciones), la temporización de los datos y el paralelismo de las tareas (procesos) que manejan los datos. En muchas situaciones, probar los datos proporcionados cuando un sistema de tiempo real está en un estado dará como resultado un procesamiento adecuado, mientras que los mismos datos proporcionados cuando el sistema está en un estado diferente pueden conducir a error.

Por ejemplo, el software de tiempo real que controla una nueva fotocopiadora acepta interrupciones del operador (es decir, el operador de la máquina presiona teclas de control como RESET o DARKEN) sin error cuando la máquina saca copias (en el estado "copying"). Estas mismas interrupciones del operador, si se ingresan cuando la máquina está en el estado "jammed", generan una pantalla del código de diagnóstico que indica la ubicación del atasco que se tiene que resolver (un error).

Además, la íntima relación que existe entre el software de tiempo real y su entorno de hardware también puede causar problemas en las pruebas. Las pruebas del software deben considerar el impacto de los fallos de hardware en el procesamiento del software. Tales fallos pueden ser extremadamente difíciles de simular de manera realista.

Los métodos amplios de diseño de casos de prueba para sistemas en tiempo real continúan evolucionando. Sin embargo, puede proponerse una estrategia global de cuatro pasos:

**? ¿Cuál es una estrategia efectiva para probar un sistema en tiempo real?**

- **Prueba de tareas.** El primer paso en la prueba del software en tiempo real es probar cada tarea de manera independiente. Es decir, las pruebas convencionales se diseñan para cada tarea y se ejecutan independientemente durante dichas pruebas. La prueba de tareas descubre errores en lógica y función, mas no en temporización y comportamiento.

- **Prueba de comportamiento.** Con modelos de sistema creados con herramientas automatizadas, es posible simular el comportamiento de un sistema en tiempo real y examinar su comportamiento como consecuencia de eventos externos. Estas actividades de análisis pueden servir de base para el diseño de los casos de prueba que se realizan cuando se construye el software en tiempo real. Al usar una técnica similar a la partición de equivalencia (sección 18.6.2), los eventos (por ejemplo, interrupciones, señales de control) se categorizan para las pruebas. Por ejemplo, los eventos para la fotocopiadora pueden ser interrupciones del usuario (contador de restablecimiento), interrupciones mecánicas (atasco de papel), interrupciones del sistema (baja de tóner) y modos de fallo (sobrecalentamiento del rodillo). Cada uno se prueba de manera individual y el comportamiento del sistema ejecutable se examina para detectar los errores que ocurren como consecuencia del procesamiento asociado con dichos eventos. El comportamiento del modelo del sistema (desarrollado durante la actividad de análisis) y el software ejecutable pueden compararse para asegurar que actúan en conformidad. Una vez que se prueba cada clase de eventos, éstos se presentan al sistema en orden aleatorio y con frecuencia aleatoria. El comportamiento del software se examina para detectar errores de comportamiento.
- **Prueba intertarea.** Una vez aislados los errores en las tareas individuales y en el comportamiento del sistema, las pruebas se cambian a los errores relacionados con el tiempo. Las tareas asíncronas que se sabe que se comunican mutuamente se prueban con diferentes tasas de datos y carga de procesamiento para determinar si ocurrirán errores de sincronización intertarea. Además, las tareas que se comunican vía cola de mensaje o almacenamiento de datos se prueban para descubrir errores en el tamaño de estas áreas de almacenamiento de datos.
- **Prueba de sistema.** Al integrar software y hardware, se lleva a cabo un amplio rango de pruebas del sistema con la intención de descubrir errores en la interfaz software-hardware. La mayoría de los sistemas en tiempo real procesan las interrupciones. Por tanto, probar la manipulación de estos eventos booleanos es esencial. Al usar el diagrama de estado (capítulo 7), el examinador desarrolla una lista de las posibles interrupciones y del procesamiento que ocurre como consecuencia de las interrupciones. Entonces se diseñan pruebas para valorar las siguientes características del sistema:
  - ¿Las prioridades de interrupción se asignan y manejan de manera adecuada?
  - ¿El procesamiento para cada interrupción se maneja de manera correcta?
  - ¿El rendimiento (por ejemplo, tiempo de procesamiento) de cada procedimiento de manejo de interrupción se apega a los requerimientos?
  - ¿Un alto volumen de interrupciones que llegan en momentos críticos crea problemas en el funcionamiento o en el rendimiento?

Además, las áreas de datos globales que se usan para transferir como parte del procesamiento de interrupción deben probarse a fin de valorar el potencial para la generación de efectos colaterales.

## 18.9 PATRONES PARA PRUEBAS DE SOFTWARE

**WebRef**

Un catálogo de patrones de prueba de software puede encontrarse en **www.rbsc.com/pages/TestPatternList.htm**

El uso de patrones como un mecanismo para describir soluciones a problemas de diseño específicos se estudió en el capítulo 12. Pero los patrones también pueden usarse para proponer soluciones a otras situaciones de ingeniería de software; en este caso, prueba del software. Los *patrones de prueba* describen problemas y soluciones de prueba comunes que pueden auxiliar en su tratamiento.

Los patrones de prueba no sólo proporcionan lineamientos útiles conforme comienzan las actividades de prueba; también proporcionan tres beneficios adicionales descritos por Marick [Mar02]:

Los patrones de prueba pueden ayudar al equipo de software a comunicarse de manera más efectiva acerca de las pruebas y comprender mejor las fuerzas que conducen a un enfoque de prueba específico.

1. Proporcionan un vocabulario para quienes solucionan problemas. "Oiga, usted sabe, debemos usar un objeto nulo".
2. Enfocan la atención en las fuerzas que hay detrás de un problema. Esto permite que los diseñadores [de caso de prueba] entiendan mejor cuándo y por qué se aplica una solución.
3. Alientan el pensamiento iterativo. Cada solución crea un nuevo contexto en el que pueden resolverse nuevos problemas.

Aunque estos beneficios son "leves", no deben pasarse por alto. Gran parte de las pruebas del software, incluso durante la década pasada, han sido actividades *ad hoc*. Si los patrones de prueba pueden ayudar a un equipo de software a comunicarse de manera más efectiva acerca de las pruebas, a comprender las fuerzas de motivación que conducen a un enfoque específico para las pruebas y a abordar el diseño de las pruebas como una actividad evolutiva en la que cada iteración resulta en una suite más completa de casos de prueba, entonces los patrones lograron mucho.

Los patrones de prueba se describen en forma muy similar a los patrones de diseño (capítulo 12). En la literatura se han propuesto decenas de patrones de prueba (por ejemplo, [Mar02]). Los siguientes tres (presentados sólo en forma resumida) proporcionan ejemplos representativos:

**WebRef**

Patrones que describen la organización, eficiencia, estrategia y resolución de problemas de las pruebas pueden encontrarse en **www.testing.com/test-patterns/patterns**.

*Nombre del patrón:* **PairTesting**

*Resumen:* Patrón orientado a proceso, **PairTesting** describe una técnica que es análoga a la programación por parejas (capítulo 3) en la que dos examinadores trabajan en conjunto para diseñar y ejecutar una serie de pruebas que pueden aplicarse a actividades de prueba de unidad, integración o validación.

*Nombre del patrón:* **SeparateTestInterface**

*Resumen:* Hay necesidad de probar cada clase en un sistema orientado a objetos, incluidas "clases internas" (es decir, clases que no exponen alguna interfaz afuera del componente que los usa). El patrón **SeparateTestInterface** describe cómo crear "una interfaz de prueba que puede usarse para describir pruebas específicas sobre clases que son visibles solamente de manera interna en un componente" [Lan01].

*Nombre del patrón:* **ScenarioTesting**

*Resumen:* Una vez realizadas las pruebas de unidad e integración, hay necesidad de determinar si el software se desempeñará en forma que satisfaga a los usuarios. El patrón **ScenarioTesting** describe una técnica para revisar el software desde el punto de vista del usuario. Un fallo en este nivel indica que el software fracasó para satisfacer un requisito visible del usuario [Kan01].

Un análisis amplio de los patrones de prueba está más allá del ámbito de este libro. Si tiene más interés, vea [Bin99] y [Mar02] para información adicional acerca de este importante tema.

## 18.10 Resumen

El objetivo principal para el diseño de casos de prueba es derivar un conjunto de pruebas que tienen la mayor probabilidad de descubrir errores en el software. Para lograr este objetivo, se usan dos categorías diferentes de técnicas de diseño de caso de prueba: pruebas de caja blanca y pruebas de caja negra.

Las pruebas de caja blanca se enfocan en la estructura de control del programa. Los casos de prueba se derivan para asegurar que todos los enunciados en el programa se ejecutaron al me-

nos una vez durante las pruebas y que todas las condiciones lógicas se revisaron. La prueba de ruta o trayectoria básica, una técnica de caja blanca, usa gráficos de programa (o matrices gráficas) para derivar el conjunto de pruebas linealmente independientes que garantizarán la cobertura del enunciado. Las pruebas de condición y de flujo de datos revisan aún más la lógica del programa, y la prueba de bucles complementa otras técnicas de caja blanca al proporcionar un procedimiento para revisar los bucles de varios grados de complejidad.

Hetzel [Het84] describe las pruebas de caja blanca como "pruebas en lo pequeño". Su implicación es que las pruebas de caja blanca que se consideraron en este capítulo por lo general se aplican a pequeños componentes del programa (por ejemplo, módulos o pequeños grupos de módulos). Las pruebas de caja blanca, por otra parte, amplían el foco y pueden llamarse "pruebas en lo grande".

Las pruebas de caja negra se diseñan para validar los requerimientos funcionales sin considerar el funcionamiento interno de un programa. Las técnicas de prueba de caja negra se enfocan en el dominio de información del software, y derivan casos de prueba mediante la partición de los dominios de entrada y salida de un programa en forma que proporciona cobertura de prueba profunda. La partición de equivalencia divide el dominio de entrada en clases de datos que es probable que revisen una función de software específica. El análisis del valor de frontera sondea la habilidad del programa para manejar datos en los límites de lo aceptable. La prueba de arreglo ortogonal proporciona un método sistemático eficiente para probar sistemas con pequeño número de parámetros de entrada. La prueba basada en modelo usa elementos del modelo de requerimientos para probar el comportamiento de una aplicación.

Los métodos de prueba especializados abarcan un amplio arreglo de capacidades de software y áreas de aplicación. La prueba para interfaces gráficas de usuario, arquitecturas cliente-servidor, documentación y centros de ayuda, y los sistemas en tiempo real requieren cada uno lineamientos y técnicas especializadas.

Con frecuencia, los desarrolladores de software experimentados dicen: "las pruebas nunca terminan, sólo se transfieren de uno [el ingeniero de software] al cliente. Cada vez que el cliente usa el programa, se realiza una prueba". Al aplicar el diseño de casos de prueba, pueden lograrse pruebas más completas y, en consecuencia, descubrir y corregir el mayor número de errores antes de comenzar "las pruebas del cliente".

## Problemas y puntos para reflexionar

**18.1.** Myers [Mye79] usa el siguiente programa como una autovaloración de su habilidad para especificar pruebas adecuadas: un programa lee tres valores enteros. Los tres se interpretan como representación de las longitudes de los lados de un triángulo. El programa imprime un mensaje que indica si el triángulo es escaleno, isósceles o equilátero. Desarrolle un conjunto de casos de prueba que crea que probarán este programa de manera adecuada.

**18.2.** Diseñe e implemente el programa (con manipulación de error donde sea adecuado) que se especifica en el problema 18.1. Derive un gráfico de flujo para el programa y aplique prueba de ruta básica para desarrollar casos de prueba que garanticen la prueba de todos los enunciados en el programa. Ejecute los casos y muestre sus resultados.

**18.3.** ¿Puede pensar en algunos objetivos de prueba adicionales que no se estudiaron en la sección 18.1.1?

**18.4.** Seleccione un componente de software que haya diseñado e implementado recientemente. Diseñe un conjunto de casos de prueba que garantice que todos los enunciados se ejecutan, usando prueba de ruta o trayectoria básica.

**18.5.** Especifique, diseñe e implemente una herramienta de software que calcule la complejidad ciclomática para el lenguaje de programación de su elección. Use la matriz de grafo como la estructura de datos operativa en su diseño.

**18.6.** Lea Beizer [Bei95] o una fuente en web relacionada (por ejemplo, **www.laynetworks.com/Discret%20Mathematics_1g.htm**) y determine cómo puede extenderse el programa que desarrolló en el problema 18.5 a fin de alojar varias enlaces ponderados. Extienda su herramienta para procesar probabilidades de ejecución o tiempos de procesamiento de liga.

**18.7.** Diseñe una herramienta automatizada que reconozca bucles y que los clasifique como se indica en la sección 18.5.3.

**18.8.** Extienda la herramienta descrita en el problema 18.7 a fin de generar casos de prueba para cada categoría de bucle, una vez encontrada. Será necesario realizar esta función de manera interactiva con el examinador.

**18.9.** Proporcione al menos tres ejemplos en los que la prueba de caja negra puede dar la impresión de que "todo está bien", mientras que las pruebas de caja blanca pueden descubrir un error. Proporcione al menos tres ejemplos en los que las pruebas de caja blanca pueden dar la impresión de que "todo está bien", mientras que las pruebas de caja negra pueden descubrir un error.

**18.10.** ¿Las pruebas exhaustivas (incluso si es posible para programas muy pequeños) garantizarán que el programa es 100 por ciento correcto?

**18.11.** Pruebe un manual de usuario (o centro de ayuda) para una aplicación que use con frecuencia. Encuentre al menos un error en la documentación.

## Lecturas adicionales y fuentes de información

Virtualmente, todos los libros dedicados a las pruebas de software consideran tanto estrategia como tácticas. Por tanto, las lecturas adicionales anotadas para el capítulo 17 son igualmente aplicables para este capítulo. Everett y Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,* Wiley, 2007), Spiller *et al*. (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3d. ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement,* 2a. ed., Auerbach, 2004), Loveland *et al*. (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig y Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002) y Whittaker (*How to Break Software,* Addison-Wesley, 2002) son sólo una pequeña muestra de muchos libros que analizan los principios, conceptos, estrategias y métodos de las pruebas.

Una segunda edición del texto clásico de Myers [Mye79], producido por Myers *et al.* (*The Art of Software Testing*, 2a. ed., Wiley, 2004), cubre con mucho detalle las técnicas de diseño de casos de prueba. Pezze y Young (*Software Testing and Analysis,* Wiley, 2007), Perry (*Effective Methods for Software Testing,* 3a. ed., Wiley, 2006), Copeland (*A Practitioner's Guide to Software Test Design,* Artech, 2003), Hutcheson (*Software Testing Fundamentals,* Wiley, 2003), Jorgensen (*Software Testing: A Craftsman's Approach,* 2a. ed., CRC Press, 2002) proporcionan cada uno presentaciones útiles de los métodos y técnicas del diseño de casos de prueba. El texto clásico de Beizer [Bei90] proporciona una amplia cobertura de las técnicas de caja blanca e introduce un nivel de rigor matemático que con frecuencia falta en otros tratamientos de las pruebas. Su último libro [Bei95] presenta un tratamiento conciso de métodos importantes.

La prueba del software es una actividad que consume muchos recursos. Es por esto que muchas organizaciones automatizan partes del proceso de prueba. Los libros de Li y Wu (*Effective Software Test Automation,* Sybex, 2004); Mosely y Posey (*Just Enough Software Test Automation,* Prentice-Hall, 2002); Dustin, Rashka, y Poston (*Automated Software Testing: Introduction, Management, and Performance,* Addison-Wesley, 1999); Graham *et al*. (*Software Test Automation*, Addison-Wesley, 1999) y Poston (*Automating Specification-Based Software Testing,* IEEE Computer Society, 1996) exponen herramientas, estrategias y métodos para pruebas automatizadas. Nquyen *et al.* (*Global Software Test Automation*, Happy About Press, 2006) presentan un panorama ejecutivo de la automatización de las pruebas.

Thomas *et al.* (*Java Testing Patterns*, Wiley, 2004) y Binder [Bin99] describen patrones de prueba que abarcan pruebas de métodos, clases/grupos, subsistemas, reutilización de componentes, marcos conceptuales y sistemas, así como la automatización de las pruebas y la prueba de bases de datos especializadas.

En internet está disponible una amplia variedad de recursos de información acerca de los métodos de diseño de casos de pruebas. Una lista actualizada de referencias en la World Wide Web que son relevantes para las técnicas de prueba puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

# 19

# PRUEBA DE APLICACIONES ORIENTADAS A OBJETOS

CONCEPTOS CLAVE



En el capítulo 18 se señaló que el objetivo de las pruebas, dicho de manera simple, es encontrar la mayor cantidad posible de errores con una cantidad manejable de esfuerzo aplicado durante un lapso realista. Aunque este objetivo fundamental permanece invariable para el software orientado a objetos (OO), la naturaleza de los programas OO cambia en la estrategia y en las tácticas de las pruebas.

Podría argumentarse que, conforme las bibliotecas de clase reutilizables crecen en tamaño, un reuso mayor mitigará a los sistemas OO en su necesidad de pruebas pesadas. Lo opuesto es exactamente cierto. Binder [Bin94b] analiza esto cuando afirma:

> Cada reuso es un nuevo contexto de uso y es prudente una nueva comprobación. Parece probable que se necesitarán más pruebas, no menos, para obtener alta confiabilidad en los sistemas orientados a objetos.

Para probar adecuadamente los sistemas OO, deben realizarse tres cosas: 1) ampliar la definición de prueba para incluir las técnicas de descubrimiento de error aplicadas al análisis orientado a objetos y a modelos de diseño, 2) cambiar significativamente la estrategia para prueba de unidad e integración y 3) explicar las características únicas del software OO mediante el diseño de casos de prueba.

## UNA MIRADA RÁPIDA

**¿Qué es?** La arquitectura del software orientado a objetos (OO) da como resultado una serie de subsistemas en capas que encapsulan clases colaboradoras. Cada uno de estos elementos de sistema (subsistemas y clases) realiza funciones que ayudan a lograr los requerimientos del sistema. Es necesario probar un sistema OO en varios niveles diferentes con la intención de descubrir errores que puedan ocurrir conforme las clases colaboran unas con otras y conforme los subsistemas se comunican a través de capas arquitectónicas.

**¿Quién lo hace?** Ingenieros de software y examinadores especializados realizan la prueba orientada a objetos.

**¿Por qué es importante?** El programa tiene que ejecutarse antes de que llegue al cliente con la intención específica de remover todos los errores, de modo que el cliente no experimente la frustración que produce encontrarse con un producto de calidad pobre. Con la finalidad de encontrar el mayor número posible de errores, las pruebas deben realizarse de manera sistemática y los casos de prueba deben diseñarse usando técnicas disciplinadas.

**¿Cuáles son los pasos?** Las pruebas OO son estratégicamente análogas a la prueba de sistemas convencionales, pero tácticamente diferentes. Puesto que el análisis OO y los modelos de diseño son similares en estructura y contenido con el programa OO resultante, las "pruebas" se inician con la revisión de dichos modelos. Una vez generado el código, la prueba OO comienza "en lo pequeño", con las pruebas de clase. Se diseña una serie de pruebas que ejercitan las operaciones de clase y que examinan si existen errores conforme una clase colabora con otras clases. En la medida en la que las clases se integran para formar un subsistema, se aplican pruebas basadas en hebra, en uso y de grupo, junto con enfoques basados en fallo, a fin de ejercitar por completo clases colaboradoras. Finalmente, se usan casos de uso (desarrollados como parte del modelo de requerimientos) para descubrir errores de validación del software.

**¿Cuál es el producto final?** Se diseña y documenta un conjunto de casos de prueba, diseñados para ejercitar clases, sus colaboraciones y comportamientos; se definen los resultados esperados y se registran los resultados reales.

**¿Cómo me aseguro de que lo hice bien?** Cuando comienzan las pruebas, cambia el punto de vista. ¡Intente "romper" el software! Diseñe casos de prueba en forma disciplinada y revise con minuciosidad los casos de prueba creados.

## 19.1 AMPLIACIÓN DE LA DEFINICIÓN DE LAS PRUEBAS

La construcción de software orientado a objetos comienza con la creación de modelos de requerimientos (análisis) y de diseño.[1] Debido a la naturaleza evolutiva del paradigma de ingeniería del software OO, dichos modelos comienzan como representaciones relativamente informales de los requisitos de sistema y evolucionan hacia modelos detallados de clases, relaciones de clase, diseño y asignación de sistema, y diseño de objetos (que incorpora un modelo de conectividad de objetos mediante mensajería). En cada etapa, los modelos pueden "probarse" con la intención de descubrir errores previamente a su propagación hacia la siguiente iteración.

Puede argumentarse que la revisión de los modelos de análisis y de diseño OO es especialmente útil, pues los mismos constructos semánticos (por ejemplo, clases, atributos, operaciones, mensajes) aparecen en los niveles de análisis, diseño y código. Por tanto, un problema en la definición de los atributos de clase que se descubra durante el análisis soslayará los efectos colaterales que puedan ocurrir si el problema no se descubriera hasta el diseño o el código (o incluso en la siguiente iteración de análisis).


*Aunque la revisión de los modelos de análisis y diseño OO es parte integral de "la prueba" de una aplicación OO, reconozca que ésta no es suficiente en y por sí misma. También debe realizar pruebas ejecutables.*

Por ejemplo, considere una clase en la que se define un número de atributos durante la primera iteración de análisis. Un atributo extraño se anexa a la clase (debido a una mala interpretación del dominio del problema). Entonces pueden especificarse dos operaciones para manipular el atributo. Se lleva a cabo una revisión y un experto en dominio puntualiza el problema. Al eliminar el atributo extraño en esta etapa, durante el análisis pueden evitarse los siguientes problemas y un esfuerzo innecesario:

1. Tal vez se generen subclases especiales para alojar el atributo innecesario o las excepciones. Se evita el trabajo involucrado en la creación de subclases innecesarias.
2. Una mala interpretación de la definición de clase puede conducir a relaciones de clase incorrectas o extrañas.
3. El comportamiento del sistema o sus clases puede caracterizarse de manera inadecuada para alojar el atributo extraño.

Si el problema no se descubre durante el análisis y se propaga aún más, podrían ocurrir los siguientes problemas durante el diseño (que se evitarían con la revisión temprana):

1. Durante el diseño del sistema, puede ocurrir la asignación inadecuada de la clase a un subsistema o a algunas tareas.
2. Puede emplearse trabajo de diseño innecesario a fin de crear el diseño de procedimientos para las operaciones que abordan el atributo extraño.
3. El modelo de mensajería será incorrecto (porque los mensajes deben diseñarse para las operaciones que son extrañas).

**Cita:**

"Las herramientas que usamos tienen profunda (¡y tortuosa!) influencia sobre nuestros hábitos de pensamiento y, por tanto, sobre nuestras habilidades de pensamiento."

**Edsger Dijkstra**

Si el problema sigue sin detectarse durante el diseño y pasa hacia la actividad de codificación, se empleará esfuerzo considerable para generar código que implemente un atributo innecesario, dos operaciones innecesarias, mensajes que activen la comunicación interobjetos y muchos otros conflictos relacionados. Además, la prueba de la clase absorberá más tiempo que el necesario. Una vez que finalmente se haya descubierto el problema, la modificación del sistema debe realizarse con el potencial siempre presente de efectos colaterales que se generen por el cambio.

Durante las últimas etapas de su desarrollo, los modelos de análisis (AOO) y de diseño (DOO) orientado a objetos proporcionan información sustancial acerca de la estructura y comporta-

1 Las técnicas de modelado de análisis y diseño se presentan en la parte 2 de este libro. Los conceptos básicos de OO se presentan en el apéndice 2.

miento del sistema. Por esta razón, dichos modelos deben sujetarse a una rigurosa revisión, previa a la generación del código.

Todos los modelos orientados a objetos deben probarse (en este contexto, el término *prueba* incorpora revisiones técnicas) en relación con su exactitud, completitud y consistencia dentro del contexto de la sintaxis, la semántica y la pragmática del modelo [Lin94a].

## 19.2 Modelos de prueba AOO y DOO

Los modelos de análisis y diseño no pueden probarse de la manera convencional porque no pueden ejecutarse. Sin embargo, pueden usarse revisiones técnicas (capítulo 15) para examinar su exactitud y consistencia.

### 19.2.1 Exactitud de los modelos AOO y DOO

La notación y la sintaxis utilizadas para representar los modelos de análisis y diseño se ligarán a los métodos de análisis y diseño específicos que se elijan para el proyecto. Por tanto, la exactitud sintáctica se juzga mediante el uso adecuado de la simbología; cada modelo se revisa para garantizar que se mantienen las convenciones de modelado adecuadas.

Durante el análisis y el diseño, la exactitud semántica puede valorarse con base en la conformidad del modelo con el dominio de problemas del mundo real. Si el modelo refleja con precisión el mundo real (en un nivel de detalle que sea apropiado para la etapa de desarrollo en la que se revisó el modelo), entonces es semánticamente correcto. Para determinar si el modelo verdaderamente refleja los requerimientos del mundo real, debe presentarse a expertos de dominio de problemas, quienes examinarán las definiciones y jerarquía de clase en busca de omisiones y ambigüedad. Las relaciones de clase (conexiones de instancia) se evalúan para determinar si reflejan con precisión conexiones de objetos en el mundo real.[2]

### 19.2.2 Consistencia de los modelos orientados a objetos

La consistencia de los modelos orientados a objetos puede juzgarse al "considerar las relaciones entre entidades en el modelo. Un modelo de análisis o diseño inconsistente tiene representaciones en una parte del modelo que no se reflejan de manera correcta en otras porciones" [McG94].

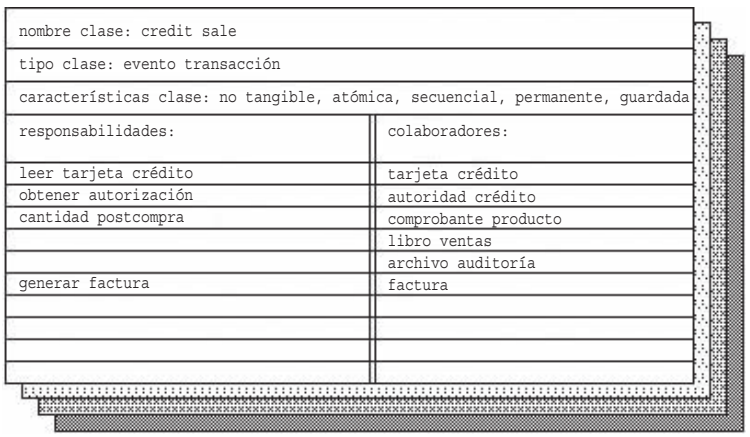
Para valorar la consistencia, debe examinarse cada clase y sus conexiones con otras clases. A fin de facilitar esta actividad, puede usarse el modelo clase-responsabilidad-colaboración (CRC) o un diagrama de objeto-relación. Como se estudió en el capítulo 6, el modelo CRC se compone de tarjetas índice CRC. Cada tarjeta CRC menciona el nombre de la clase, sus responsabilidades (operaciones) y sus colaboradores (otras clases a las que envía mensajes y de las que depende para lograr sus responsabilidades). Las colaboraciones implican una serie de relaciones (es decir, conexiones) entre clases del sistema OO. El modelo objeto-relación proporciona una representación gráfica de las conexiones entre clases. Toda esta información puede obtenerse a partir del modelo de análisis (capítulos 6 y 7).

Para evaluar el modelo de clase, se recomienda seguir los siguientes pasos [McG94]:

1. **Vuelva a consultar el modelo CRC y el modelo objeto-relación**. Realice una comprobación cruzada para garantizar que todas las colaboraciones implicadas por el modelo de requerimientos se reflejan de manera adecuada en ambas.

2 Los casos de uso pueden ser invaluables para cotejar los modelos de análisis y diseño contra escenarios de uso del sistema OO en el mundo real.

**FIGURA 19.1**
Ejemplo de tarjeta índice CRC utilizada para revisión




2. **Inspeccione la descripción de cada tarjeta índice CRC para determinar si una responsabilidad delegada es parte de la definición del colaborador.** Por ejemplo, considere una clase definida por un sistema de comprobación punto de venta y que se llame **CreditSale**. Esta clase tiene una tarjeta índice CRC como la que se ilustra en la figura 19.1.

   Para esta colección de clases y colaboraciones, pregunte si una responsabilidad (por ejemplo, *leer tarjeta crédito*) se cumple delegándola al colaborador mencionado (**CreditCard**). Es decir, ¿la clase **CreditCard** tiene una operación que le permite leerse? En este caso, la respuesta es "sí". El objeto-relación se recorre para garantizar que tales conexiones son válidas.

3. **Invertir la conexión para garantizar que cada colaborador al que se solicita servicio recibe solicitud de una fuente razonable.** Por ejemplo, si la clase **CreditCard** recibe una solicitud para `purchase amount` (cantidad compra) de la clase **CreditSale**, habría un problema. **CreditCard** no conoce la cantidad de compra.

4. **Al usar las conexiones invertidas que se examinaron en el paso 3, se determina si es posible requerir otras clases o si las responsabilidades se agrupan de manera adecuada entre las clases.**

5. **Determinar si las responsabilidades de amplia solicitud pueden combinarse en una sola responsabilidad**. Por ejemplo, *leer tarjeta de crédito* y *obtener autorización* ocurren en toda situación. Pueden combinarse en una responsabilidad *validar solicitud crédito* que incorpora obtener el número de tarjeta de crédito y conseguir la autorización.

Los pasos del 1 al 5 deben aplicarse de manera iterativa a cada clase y a lo largo de cada evolución del modelo de requerimientos.

Una vez creado el modelo de diseño (capítulos del 9 al 11), también deben realizarse revisiones del diseño del sistema y del diseño del objeto. El diseño del sistema bosqueja la arquitectura de producto global, los subsistemas que abarca el producto, la forma en la que los subsistemas se asignan a los procesadores, la asignación de clases a los subsistemas y el diseño de la interfaz de usuario. El modelo de objetos presenta los detalles de cada clase y las actividades de mensajería que se necesitan para implementar las colaboraciones entre clases.

El diseño del sistema se revisa al examinar el modelo de comportamiento del objeto desarrollado durante el análisis y el mapeo orientado a objetos requerido por el comportamiento del

sistema contra los subsistemas diseñados para lograr este comportamiento. La concurrencia y la asignación de tarea también se revisan en el contexto del comportamiento del sistema. Los estados de comportamiento del sistema se evalúan para determinar cuál existe de manera concurrente. Los casos de uso se utilizan para ejercitar el diseño de la interfaz de usuario.

El modelo de objetos debe ponerse a prueba contra la red de relación de objetos a fin de asegurar que todos los objetos diseñados contienen los atributos y operaciones necesarios para implementar las colaboraciones definidas para cada tarjeta índice CRC. Además, se revisa la especificación minuciosa de los detalles de operación (es decir, los algoritmos que implementan las operaciones).

## 19.3 Estrategias de pruebas orientadas a objetos

Como se anotó en el capítulo 18, la estrategia clásica de prueba de software comienza "probando en lo pequeño" y funciona hacia afuera, "probando en lo grande". Dicho en el lenguaje de las pruebas de software (capítulo 18), comienza con la *prueba de unidad*, luego avanza hacia la *prueba de integración* y culmina con las *pruebas de validación y sistema*. En aplicaciones convencionales, la prueba de unidad se enfoca en la unidad de programa compatible más pequeña: el subprograma (por ejemplo, componente, módulo, subrutina, procedimiento). Una vez que cada una de estas unidades se prueba de manera individual, se integra en una estructura de programa mientras se aplica una serie de pruebas de regresión para descubrir errores debidos a la puesta en interfaz de los módulos y los efectos colaterales que se generan al sumar nuevas unidades. Finalmente, el sistema como un todo se prueba para garantizar que se descubren los errores en los requerimientos.

### 19.3.1 Prueba de unidad en el contexto OO


La "unidad" comprobable más pequeña en el software OO es la clase. La prueba de clase se activa mediante las operaciones encapsuladas por la clase y por el comportamiento de estado de la misma.

Cuando se piensa en software orientado a objetos, cambia el concepto de unidad. La encapsulación impulsa la definición de clases y objetos. Esto significa que cada clase y cada instancia de una clase (objeto) encapsulan los atributos (datos) y las operaciones (también conocidas como métodos o servicios) que manipulan dichos datos. En lugar de probar un módulo individual, la unidad comprobable más pequeña es la clase encapsulada. Puesto que una clase puede contener algunas operaciones diferentes y una operación particular puede existir como parte de un número de clases diferentes, el significado de prueba de unidad cambia dramáticamente.

Ya no es posible probar una sola operación aislada (la visión convencional de la prueba de unidad) sino, más bien, como parte de una clase. Para ilustrar lo anterior, considere una jerarquía de clase en la que se define una operación *X( )* para la superclase y la heredan algunas subclases. Cada subclase usa la operación *X( )*, pero se aplica dentro del contexto de los atributos y operaciones privados que se definieron para cada subclase. Puesto que el contexto donde se usa la operación *X( )* varía en formas sutiles, es necesario probarla en el contexto de cada una de las subclases. Esto significa que probar la operación *X( )* en el vacío (el enfoque tradicional de la prueba de unidad) no es efectivo en el contexto orientado a objetos.

La prueba de clase para el software OO es el equivalente de la prueba de unidad para software convencional.[3] A diferencia de la prueba de unidad del software convencional, que tiende a enfocarse en el detalle algorítmico de un módulo y en los datos que fluyen a través de la interfaz de módulo, la prueba de clase para el software OO se activa mediante las operaciones encapsuladas por la clase y por el comportamiento de estado de la misma.

3 En las secciones 19.4 a 19.6 se estudian los métodos de diseño de casos de prueba para clases OO.

### 19.3.2 Prueba de integración en el contexto OO

Puesto que el software orientado a objetos no tiene una estructura de control jerárquica, las estrategias de integración tradicionales, descendente y ascendente, tienen poco significado. Además, integrar operaciones una a la vez en una clase (el enfoque de integración incremental convencional) con frecuencia es imposible debido a las "interacciones directas e indirectas de los componentes que constituyen la clase" [Ber93].

**PUNTO CLAVE**

La prueba de integración para software OO examina un conjunto de clases que se requieren para responder a un evento dado.

Existen dos diferentes estrategias para la prueba de integración de los sistemas OO [Bin94a]. La primera, *prueba basada en hebra*, integra el conjunto de clases requeridas para responder a una entrada o evento del sistema. Cada hebra se integra y prueba de manera individual. La prueba de regresión se aplica para asegurar que no ocurran efectos colaterales. El segundo enfoque de integración, *prueba basada en uso*, comienza la construcción del sistema al probar aquellas clases (llamadas *independientes*) que usan muy pocas clases de servidor (si es que emplean alguna). Después de probar las clases independientes, se examina la siguiente capa de clases que usan las clases independientes, llamadas *dependientes*. Esta secuencia de pruebas para las capas de clases dependientes continúa hasta que se construye todo el sistema. A diferencia de la integración convencional, cuando sea posible debe evitarse el uso de controladores y representantes (proxies) (capítulo 18) como operaciones de reemplazo.

La *prueba de grupo* [McG94] es un paso en la prueba de integración del software OO. En ella, se ejercita un grupo de clases colaboradoras (determinadas al examinar el CRC y el modelo objeto-relación) al diseñar casos de prueba que intentan descubrir errores en las colaboraciones.

### 19.3.3 Prueba de validación en un contexto OO

En el nivel de validación o de sistema, desaparecen los detalles de las conexiones de clase. Como la validación convencional, la del software OO se enfoca en las acciones visibles para el usuario y en las salidas del sistema reconocibles por él mismo. Para auxiliar en la derivación de pruebas de validación, el examinador debe recurrir a casos de uso (capítulos 5 y 6) que sean parte del modelo de requerimientos. El caso de uso proporciona un escenario que tiene una alta probabilidad de descubrir errores en los requerimientos de interacción de usuario.

Los métodos convencionales de prueba de caja negra (capítulo 18) pueden usarse para activar pruebas de validación. Además, puede elegirse derivar casos de prueba del modelo de comportamiento del objeto y de un diagrama de flujo de evento creado como parte del AOO.

## 19.4 MÉTODOS DE PRUEBA ORIENTADA A OBJETOS

**Cita:**

"Veo a los examinadores como los guardaespaldas del proyecto. Defendemos del fallo el flanco de nuestros desarrolladores, mientras ellos se enfocan en crear éxito."

**James Bach**

La arquitectura del software orientado a objetos da como resultado una serie de subsistemas en capas que encapsulan clases colaboradoras. Cada uno de estos elementos de sistema (subsistemas y clases) realiza funciones que ayudan a lograr requerimientos de sistema. Es necesario probar un sistema OO en varios niveles diferentes con la intención de descubrir errores que puedan ocurrir conforme las clases colaboran unas con otras y conforme los subsistemas se comunican a través de las capas arquitectónicas.

Los métodos de diseño de casos de prueba para el software orientado a objetos siguen evolucionando. Sin embargo, Berard [Ber93] sugiere un enfoque global en el diseño de casos de prueba OO:

1. Cada caso de prueba debe identificarse de manera única y explícita asociado con la clase que se va a probar.
2. Debe establecerse el propósito de la prueba.

3. Debe desarrollarse una lista de pasos de prueba para cada una de ellas, que debe contener:
   a. Una lista de estados especificados para la clase que se probará
   b. Una lista de mensajes y operaciones que se ejercitarán como consecuencia de la prueba
   c. Una lista de excepciones que pueden ocurrir conforme se prueba la clase
   d. Una lista de condiciones externas (es decir, con la finalidad de realizar adecuadamente las pruebas, cambios en el entorno externo al software que debe existir)
   e. Información complementaria que ayudará a comprender o a implementar la prueba

A diferencia del diseño convencional de casos de prueba, que se activan mediante una visión entrada-proceso-salida del software o con el detalle algorítmico de módulos individuales, la prueba orientada a objetos se enfoca en el diseño de secuencias apropiadas de operaciones para ejercitar los estados de una clase.

### 19.4.1 Implicaciones del diseño de casos de prueba de los conceptos OO

**WebRef**

Se puede encontrar una excelente serie de documentos y recursos sobre pruebas OO en **www.rbsc.com**.

Conforme una clase evoluciona a través de los modelos de requerimientos y diseño, se convierte en un blanco para el diseño de casos de prueba. Puesto que los atributos y las operaciones están encapsulados, por lo general es improductivo probar operaciones afuera de la clase. Aunque la encapsulación es un concepto de diseño esencial para OO, puede crear un obstáculo menor cuando se prueba. Como anota Binder [Bin94a]: "las pruebas requieren reportar el estado concreto y abstracto de un objeto". No obstante, la encapsulación puede hacer que esta información sea un poco difícil de obtener. A menos que se proporcionen operaciones internas a fin de reportar los valores para los atributos de clase, puede ser difícil adquirir una instantánea del estado de un objeto.

La herencia también puede presentar retos adicionales durante el diseño de casos de prueba. Ya se anotó que cada nuevo contexto de uso requiere un nuevo examen, aun cuando se haya logrado el reuso. Además, la herencia múltiple[4] complica la prueba todavía más al aumentar el número de contextos para los cuales se requiere la prueba [Bin94a]. Si dentro del mismo dominio de problema se usan subclases instanciadas de una superclase, es probable que el conjunto de casos de prueba derivados para la superclase pueda usarse cuando se prueba la subclase. Sin embargo, si la superclase se usa en un contexto completamente diferente, los casos de prueba de superclase tendrán poca aplicabilidad y debe diseñarse un nuevo conjunto de pruebas.

### 19.4.2 Aplicabilidad de los métodos convencionales de diseño de casos de prueba

Los métodos de prueba de caja blanca descritos en el capítulo 18 pueden aplicarse a las operaciones definidas para una clase. Las técnicas de ruta básica, prueba de bucle o flujo de datos pueden ayudar a garantizar que se probaron todos los enunciados en 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 puede redirigirse mejor para probar en un nivel de clase.

Los métodos de prueba de caja negra son tan apropiados para los sistemas OO como para los sistemas desarrollados, usando métodos de ingeniería del software convencional. Como se observó en el capítulo 18, los casos de uso pueden proporcionar entrada útil en el diseño de las pruebas de caja negra y en las basadas en estado.

4 Un concepto OO que debe usarse con cuidado extremo.

### 19.4.3 Prueba basada en fallo[5]

**PUNTO CLAVE**

La estrategia para la prueba basada en fallo es elaborar hipótesis acerca de un conjunto de fallos plausibles y luego derivar pruebas para corroborar o descartar cada hipótesis.

El objeto de la *prueba basada en fallo* dentro de un sistema OO es diseñar pruebas que tengan una alta probabilidad de descubrir fallos plausibles. Puesto que el producto o sistema debe adecuarse a los requerimientos del cliente, la planificación preliminar requerida para realizar alguna prueba basada en fallo comienza con el modelo de análisis. El examinador busca fallos plausibles, es decir, aspectos de la implementación del sistema que pueden resultar en defectos. Para determinar si existen dichos fallos, los casos de prueba se diseñan a fin de ejercitar el diseño o código.

Desde luego, la efectividad de dichas técnicas depende de cómo perciben los examinadores un fallo plausible. Si los fallos reales en un sistema OO se perciben como improbables, entonces este enfoque realmente no es mejor que cualquier técnica de prueba aleatoria. Sin embargo, si los modelos de análisis y diseño pueden proporcionar comprensión acerca de lo que es probable que vaya mal, entonces la prueba basada en fallo puede encontrar un significativo número de errores con gastos de esfuerzo relativamente bajos.

**? ¿Qué tipos de fallos se encuentran en los llamados de operación y en las conexiones de mensaje?**

La prueba de integración busca fallos plausibles en los llamados de operación y en las conexiones de mensaje. En este contexto se encuentran tres tipos de fallos: resultado inesperado, uso de operación/mensaje equivocado e invocación incorrecta. Para determinar fallos plausibles cuando se invocan funciones (operaciones), debe examinarse el comportamiento de la operación.

La prueba de integración se aplica a los atributos así como a las operaciones. Los "comportamientos" de un objeto están definidos por los valores que le son asignados a sus atributos. Las pruebas deben ejercer los atributos para determinar si los valores adecuados ocurren para los distintos tipos de comportamiento de los objetos.

Es importante observar que la prueba de integración intenta encontrar errores en el objeto cliente, no en el servidor. Dicho en términos convencionales, el foco de la prueba de integración es determinar si existen errores en el código que llama, no en el código llamado. La llamada de operación se usa como pista: es una forma de encontrar requerimientos de prueba que ejerciten el código que llama.

### 19.4.4 Casos de prueba y jerarquía de clase

La herencia no dispensa la necesidad de pruebas amplias de todas las clases derivadas. De hecho, en realidad puede complicar el proceso de prueba. Considere la siguiente situación. Una clase **Base** contiene operaciones *inherited( )* y *redefined( )*. Una clase **Derived** redefine *redefined( )* para servir en un contexto local. Hay poca duda de que **Derived::redefined( )** tiene que probarse porque representa un nuevo diseño y un nuevo código. Pero, ¿**Derived::inherited( )** debe probarse nuevamente?

**PUNTO CLAVE**

Aun cuando una clase base se probó ampliamente, todavía tendrá que probar todas las clases derivadas de ella.

Si **Derived::inherited( )** llama a *redefined( )* y el comportamiento de *redefined( )* cambió, **Derived::inherited( )** puede manejar mal el nuevo comportamiento. Por tanto, necesita nuevas pruebas aun cuando el diseño y el código no hayan cambiado. No obstante, es importante observar que es posible que sólo se ejecute un subconjunto de todos las pruebas para **Derived::inherited( )**. Si parte del diseño y código para *inherited( )* no depende de *redefined( )* (es decir, no lo llama ni llama a código alguno que lo llama de manera indirecta), dicho código no necesita probarse de nuevo en la clase derivada.

**Base::redefined( )** y **Derived::redefined( )** son dos operaciones diferentes con diferentes especificaciones e implementaciones. Cada una tendrá un conjunto de requerimientos de prueba derivadas de la especificación y la implementación. Dichos requerimientos de prueba sondean fallos plausibles: de integración, de condición, de frontera, etcétera. Pero es probable que las opera-

5 En las secciones 19.4.3 a 19.4.6 se realizó una adaptación de un artículo de Brian Marick publicado en el grupo de noticias de internet llamado comp.testing. Esta adaptación se incluye con el permiso del autor. Para mayor información acerca de estos temas, vea [Mar94]. Debe observarse que las técnicas estudiadas en estas secciones también son aplicables a software convencional.

ciones sean similares. Sus conjuntos de requerimientos de prueba se traslaparán. Mientras mejor sea el diseño OO, mayor es el traslape. Es necesario derivar nuevas pruebas sólo para aquellos requerimientos **Derived::redefined( )** que no se satisfagan con las pruebas **Base::redefined( )**.

Para resumir, las pruebas **Base::redefined( )** se aplican a objetos de la clase **Derived**. Las entradas de prueba pueden ser adecuadas tanto para la clase base como para la derivada, pero los resultados esperados pueden diferir en la clase derivada.

### 19.4.5 Diseño de pruebas basadas en escenario


La prueba basada en escenario descubrirá errores que ocurren cuando cualquier actor interactúa con el software.

Las pruebas basadas en fallo pierden dos tipos principales de errores: 1) especificaciones incorrectas e 2) interacciones entre subsistemas. Cuando ocurren errores asociados con una especificación incorrecta, el producto no hace lo que el cliente quiere. Puede hacer lo correcto u omitir funcionalidad importante. Pero en cualquier circunstancia, la calidad (conformidad con los requerimientos) se resiente. Los errores asociados con la interacción de subsistemas ocurren cuando el comportamiento de un subsistema crea circunstancias (por ejemplo, eventos, flujo de datos) que hacen que otro subsistema falle.

La prueba basada en escenario se concentra en lo que hace el usuario, no en lo que hace el producto. Esto significa capturar las tareas (por medio de casos de uso) que el usuario tiene que realizar y luego aplicar éstas y sus variantes como pruebas.

Los escenarios descubren errores de interacción. Pero, para lograr esto, los casos de prueba deben ser más complejos y más realistas que las pruebas basadas en fallo. La prueba basada en escenario tiende a ejercitar múltiples subsistemas en una sola prueba (los usuarios no se limitan al uso de un subsistema a la vez).

Como ejemplo, tome en cuenta el diseño de pruebas basadas en escenario para un editor de texto al revisar los casos de uso que siguen:

**Caso de uso: corrección del borrador final**

**Antecedentes:** No es raro imprimir el borrador "final", leerlo y descubrir algunos errores desconcertantes que no fueron obvios en la imagen de la pantalla. Este caso de uso describe la secuencia de eventos que ocurren cuando esto sucede.

1. Imprimir todo el documento.
2. Moverse en el documento, cambiar ciertas páginas.
3. Conforme cada página cambia, imprimirla.
4. En ocasiones se imprime una serie de páginas.


"Si quiere y espera que un programa funcione, muy probablemente verá un programa en funcionamiento: no percibirá los errores."

Cem Kaner *et al.*

Este escenario describe dos cosas: una prueba y necesidades específicas del usuario. Las necesidades del usuario son obvias: 1) un método para imprimir páginas solas y 2) un método para imprimir un rango de páginas. Mientras avanzan las pruebas, hay necesidad de probar la edición después de imprimir (así como lo inverso). Por tanto, se trabaja para diseñar pruebas que descubrirán errores en la función de edición que fueron provocados por la función de impresión, es decir, errores que indicarán que las dos funciones de software no son adecuadamente independientes.

**Caso de uso: imprimir una nueva copia**

**Antecedentes:** Alguien pide al usuario una copia reciente del documento. Debe imprimirla.

1. Abrir el documento.
2. Imprimirlo.
3. Cerrar el documento.

De nuevo, el enfoque de las pruebas es relativamente obvio. Excepto que este documento no aparece de la nada. Se creó en una tarea anterior. ¿Dicha tarea afecta a la actual?

En muchos editores modernos, los documentos recuerdan cómo se imprimieron la última vez. Por defecto, imprimen de la misma forma la siguiente ocasión. Después del escenario **Corrección del borrador final**, seleccionar solamente "Imprimir" en el menú y dar clic en el botón Imprimir en el recuadro de diálogo hará que la última página corregida se imprima de nuevo. De manera que, de acuerdo con el editor, el escenario correcto debe verse del modo siguiente:


*Aunque la prueba basada en escenario tiene méritos, obtendrá un mayor rendimiento en el tiempo invertido al revisar los casos de uso cuando estas pruebas se desarrollen como parte del modelo de análisis.*

**Caso de uso: imprimir una nueva copia**

1. Abrir el documento.
2. Seleccionar "Imprimir" en el menú.
3. Comprobar si se imprime un rango de páginas; si es así, dar *clic* para imprimir todo el documento.
4. Dar *clic* en el botón Imprimir.
5. Cerrar el documento.

Pero este escenario indica una potencial especificación de error. El editor no hace lo que el usuario razonablemente espera que haga. Los clientes con frecuencia pasan por alto la comprobación anotada en el paso 3. Entonces quedarán desconcertados cuando vayan a la impresora y encuentren una página cuando querían 100. Los clientes desconcertados señalan errores de especificación.

Esta dependencia puede perderse cuando se diseñan pruebas, pero es probable que el problema salga a la luz durante las pruebas. Entonces tendría que lidiar con la probable respuesta: "¡así se supone que debe trabajar!"

### 19.4.6 Pruebas de las estructuras superficial y profunda

Cuando se habla de *estructura superficial* se hace referencia a la estructura observable externamente de un programa OO, es decir, la estructura que es inmediatamente obvia para un usuario final. En lugar de realizar funciones, a los usuarios de muchos sistemas OO se les pueden dar objetos para manipular en alguna forma. Pero, cualquiera que sea la interfaz, las pruebas se basan todavía en tareas de usuario. Capturar estas tareas involucra comprensión, observación y hablar con usuarios representativos (y tantos usuarios no representativos como valga la pena considerar).


Examinar la estructura superficial es análogo a la prueba de caja negra. La prueba de la estructura profunda es similar a la prueba de caja blanca.

Seguramente habrá alguna diferencia en los detalles. Por ejemplo, en un sistema convencional con una interfaz orientada a comandos, el usuario puede usar la lista de todos los comandos como una lista de comprobación de la prueba. Si no existieran escenarios de prueba para ejercitar un comando, la prueba probablemente pasaría por alto algunas tareas (o la interfaz tendría comandos inútiles). En una interfaz orientada a objetos, el examinador puede usar la lista de todos los objetos como una lista de comprobación de prueba.

Las mejores pruebas se derivan cuando el diseñador observa el sistema en una forma nueva o no convencional. Por ejemplo, si el sistema o producto tiene una interfaz basada en comando, se derivarán pruebas más profundas si el diseñador de casos de prueba pretende que las operaciones sean independientes de los objetos. Plantee preguntas como "¿el usuario querrá usar esta operación, que se aplica sólo al objeto **Scanner**, mientras trabaja con la impresora?". Cualquiera que sea el estilo de la interfaz, el diseño de casos de prueba que ejercitan la estructura superficial debe usar objetos y operaciones como pistas que conduzcan a tareas pasadas por alto.

**Cita:**

"No se avergüence por los errores y, por tanto, no los convierta en crímenes."

**Confucio**

Cuando se habla de *estructura profunda*, se hace referencia a los detalles técnicos internos de un programa OO, es decir, la estructura que se comprende al examinar el diseño y/o el código. La prueba de estructura profunda se diseña para ejercitar dependencias, comportamientos y mecanismos de comunicación que se establezcan como parte del modelo de diseño para el software OO.

Los modelos de requerimientos y diseño se usan como base para la prueba de la estructura profunda. Por ejemplo, el diagrama de colaboración UML o el modelo de despliegue muestran

colaboraciones entre objetos y subsistemas que pueden no ser visibles de manera externa. Entonces el diseño de caso de prueba pregunta: "¿Se capturó (como prueba) alguna tarea que ejercita la colaboración anotada en el diagrama de colaboración? Si no fue así, ¿por qué no se hizo?"

## 19.5 MÉTODOS DE PRUEBA APLICABLES EN EL NIVEL CLASE


*El número de posibles permutas para la prueba aleatoria puede volverse muy grande. Para mejorar la eficiencia de la prueba, puede usarse una estrategia similar a la prueba de arreglo ortogonal.*

La prueba "en lo pequeño" se enfoca en una sola clase y en los métodos que encapsula ésta. La prueba aleatoria y la partición son métodos que pueden usarse para ejercitar una clase durante la prueba OO.

### 19.5.1 Prueba aleatoria para clases OO

Para ofrecer breves ilustraciones de estos métodos, considere una aplicación bancaria en la que una clase **Account** (cuenta) tiene las siguientes operaciones: *open( )*, *setup( )*, *deposit( )*, *withdraw( )*, *balance( )*, *sumaries( )*, *creditLimit( )* y *close( )* (abrir, configurar, depósito, retiro, saldo, resumen, límite de crédito y cerrar) [Kir94]. Cada una de estas operaciones puede aplicarse a **Account**, pero ciertas restricciones (por ejemplo, la cuenta debe abrirse antes de que otras operaciones puedan aplicarse y debe cerrarse después de que todas las operaciones se completen) están implícitas por la naturaleza del problema. Incluso con estas restricciones, existen muchas permutas de las operaciones. La historia de vida de comportamiento mínima de una instancia de **Account** incluye las siguientes operaciones:

**open • setup • deposit • withdraw • close**

Esto representa la secuencia de prueba mínima para account. Sin embargo, dentro de esta secuencia puede ocurrir una amplia variedad de otros comportamientos:

**open • setup • deposit • [deposit | withdraw | balance | summarize | creditLimit]$^n$ • withdraw • close**

Varias secuencias diferentes de operaciones pueden generarse al azar. Por ejemplo:

*Caso de prueba* $r_1$: **open • setup • deposit • deposit • balance • summarize • withdraw • close**

*Caso de prueba* $r_2$: **open • setup • deposit • withdraw • deposit • balance • creditLimit • withdraw • close**

Éstas y otras pruebas de orden aleatorio se realizan para ejercitar diferentes historias de vida de las instancias de clase.

**CASASEGURA**

*Prueba de clase*

**La escena:** Cubículo de Shakira.

**Participantes:** Jamie y Shakira, miembros del equipo de ingeniería de software CasaSegura, que trabajan en el diseño de casos de prueba para la función seguridad.

**La conversación:**

**Shakira:** Desarrollé algunas pruebas para la clase **Detector** [figura 10.4]; tú sabes, la que permite el acceso a todos los objetos **Sensor** para la función seguridad. ¿Estás familiarizado con ella?

**Jamie (ríe):** Seguro, es la que te permite agregar el sensor "angustia de perrito".

**Shakira:** La única. De cualquier forma, tiene una interfaz con cuatro operaciones: *read( )*, *enable( )*, *disable( )* y *test( )*. Antes de poder leer un sensor, debe habilitársele. Una vez habilitado, puede leerse y probarse. Puede deshabilitarse en cualquier momento, excepto si se procesa una condición de alarma. Así que definí una secuencia de prueba simple que ejercitará su historia de vida de comportamiento. [Muestra a Jamie la siguiente secuencia].

#1: enable•test•read•disable

**Jamie:** Eso funcionará, ¡pero tienes que hacer más pruebas que eso!

**Shakira:** Ya sé, ya sé, aquí hay otras secuencias que encontré. [Muestra a Jamie las siguientes secuencias].

#2: enable•test*[read]$^n$•test•disable

#3: [read]$^n$

#4: enable*disable•[test | read]

**Jamie:** Déjame ver si entiendo la intención de éstos. El número 1 pasa a través de una historia de vida normal, una especie de uso convencional. El número 2 repite la operación leer n veces, y ése es un escenario probable. El número 3 intenta leer el sensor antes de que esté habilitado... eso produciría un mensaje de error de algún tipo, ¿cierto? El número 4 habilita y deshabilita el sensor y luego intenta leerlo. ¿No es lo mismo que la prueba 2?

**Shakira:** En realidad, no. En el número 4, el sensor se habilitó. Lo que realmente prueba el número 4 es si la operación deshabilitar funciona como debe. Un *read( )* o *test( )* después de *disable( )* generaría el mensaje de error. Si no lo hace, entonces hay un error en la operación deshabilitar.

**Jamie:** Bien. Sólo recuerda que las cuatro pruebas tienen que aplicarse para cada tipo de sensor, pues todas las operaciones pueden tener diferencias sutiles dependiendo del tipo de sensor.

**Shakira:** No hay que preocuparse. Ése es el plan.

### 19.5.2 Prueba de partición en el nivel de clase

**? ¿Qué opciones de prueba están disponibles en el nivel de clase?**

La *prueba de partición* reduce el número de casos de prueba requeridos para ejercitar la clase, en una forma muy similar a la partición de equivalencia (capítulo 18) para el software tradicional. Las entradas y salidas se categorizan y los casos de prueba se diseñan para ejercitar cada categoría. ¿Pero cómo se derivan las categorías de partición?

La *partición con base en estado* categoriza las operaciones de clase a partir de su capacidad para cambiar el estado de la clase. Considere de nuevo la clase **Account**, las operaciones de estado incluyen *deposit( )* y *withdraw( )*, mientras que las operaciones de no estado incluyen *balance( )*, *sumaries( )* y *creditLimit( )*. Las pruebas se diseñan para que ejerciten por separado las operaciones que cambian el estado y aquellas que no lo cambian. En consecuencia,

*Caso de prueba* $p_1$: **open • setup • deposit • deposit • withdraw • withdraw • close**
*Caso de prueba* $p_2$: **open • setup • deposit • summarize • creditLimit • withdraw • close**

El caso de prueba $p_1$ cambia el estado, mientras que el $p_2$ ejercita las operaciones que no cambian el estado (distintas a las que están en la secuencia de prueba mínima).

La *partición con base en atributo* categoriza las operaciones de clase con base en los atributos que usan. Para la clase **Account**, los atributos **balance** y **creditLimit** pueden usarse para definir particiones. Las operaciones se dividen en tres particiones: 1) operaciones que usan **creditLimit**, 2) operaciones que modifican **creditLimit** y 3) operaciones que no usan ni modifican **creditLimit**. Entonces se diseñan secuencias de prueba para cada partición.

La *partición basada en categoría* jerarquiza las operaciones de clase con base en la función genérica que cada una realiza. Por ejemplo, las operaciones en la clase **Account** pueden categorizarse en operaciones de inicialización (*open*, *setup*), de cálculo (*deposit, withdraw*), consultas (*balance, summarize, creditLimit*) y de terminación (*close*).

## 19.6 Diseño de casos de prueba interclase

El diseño de casos de prueba se vuelve más complicado conforme comienza la integración del sistema orientado a objetos. En esta etapa debe comenzar la prueba de las colaboraciones entre clases. Para ilustrar "la generación de casos de prueba interclase" [Kir94], se expande el ejemplo bancario presentado en la sección 19.5 a fin de incluir las clases y colaboraciones anotadas en la figura 19.2. La dirección de las flechas en la figura indica la dirección de los mensajes y las etiquetas indican las operaciones que se involucran como consecuencia de las colaboraciones que implican los mensajes.

**Cita:**

"La frontera que define el ámbito de las pruebas de unidad y de integración es diferente para el desarrollo orientado a objetos. Las pruebas pueden diseñarse y ejercitarse en muchos puntos en el proceso. Por tanto, "diseñe un poco, codifique un poco" se convierte en "diseñe un poco, codifique un poco, pruebe un poco".

**Robert Binder**

Al igual que la prueba de clases individuales, la de colaboración de clase puede lograrse aplicando métodos aleatorios y de partición, así como pruebas basadas en escenario y pruebas de comportamiento.

### 19.6.1 Prueba de clase múltiple

Kirani y Tsai [Kir94] sugieren la siguiente secuencia de pasos para generar casos de prueba aleatorios de clase múltiple:

1. Para cada clase cliente, use la lista de operaciones de clase a fin de generar una serie de secuencias de prueba aleatorias. Las operaciones enviarán mensajes a otras clases servidor.
2. Para cada mensaje generado, determine la clase colaborador y la correspondiente operación en el objeto servidor.
3. Para cada operación en el objeto servidor (invocado por los mensajes enviados desde el objeto cliente), determine los mensajes que transmite.
4. Para cada uno de los mensajes, determine el siguiente nivel de operaciones que se invocan e incorpore esto en la secuencia de prueba.

Para ilustrar [Kir94], considere una secuencia de operaciones para la clase **Bank** en relación con una clase **ATM** (figura 19.2):

**verifyAcct • verifyPIN • [[verifyPolicy • withdrawReq] | depositReq | acctInfoREQ]$^n$**

Un caso de prueba aleatorio para la clase **Bank** puede ser

*Caso de prueba* $r_3$ = **verifyAcct • verifyPIN • depositReq**

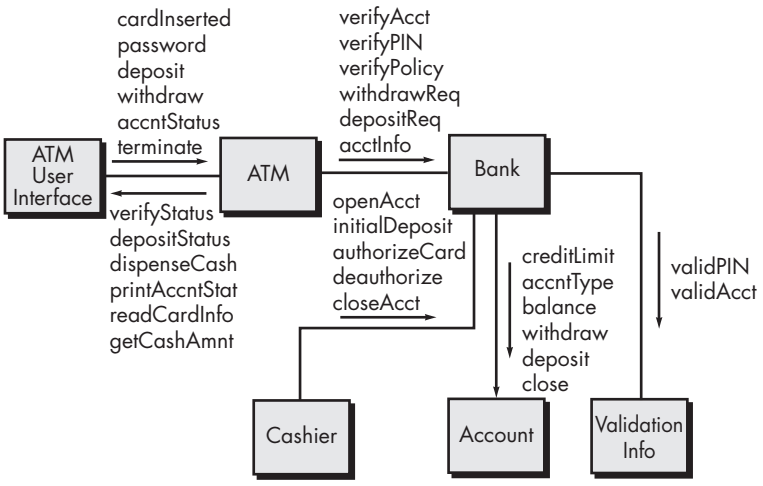
Para considerar los colaboradores involucrados en esta prueba, se consideran los mensajes asociados con cada una de las operaciones anotadas en el caso de prueba $r_3$. **Bank** debe colaborar con **ValidationInfo** para ejecutar *verifyAcct( )* y *verifyPIN( )*. **Bank** debe colaborar con **Account** para ejecutar *depositReq( )*. Por tanto, un nuevo caso de prueba que ejercita estas colaboraciones es

*Caso de prueba* $r_4$ = **verifyAcct [Bank:validAcctValidationInfo] • verifyPIN [Bank: validPinValidationInfo] • depositReq [Bank: depositaccount]**

**FIGURA 19.2**

**Diagrama de colaboración de clases para aplicación bancaria**

*Fuente:* Adaptado de [Kir94].



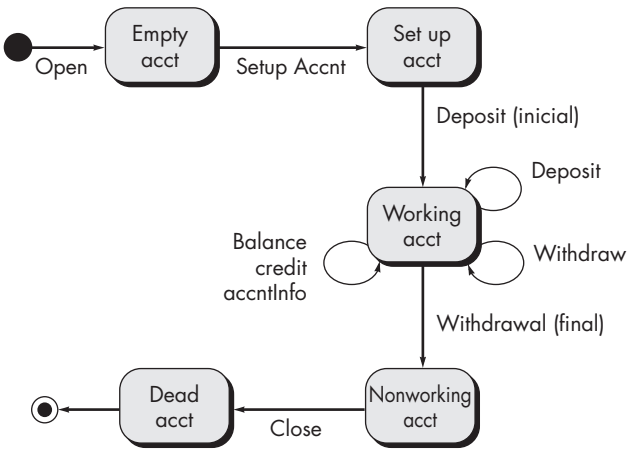



**FIGURA 19.3**
**Diagrama de estado para la clase Account**
*Fuente:* Adaptado de [Kir94].

El enfoque de prueba de partición de clase múltiple es similar al que se usó para la prueba de partición de clases individuales. Una sola clase se divide, como se estudió en la sección 19.5.2. Sin embargo, la secuencia de prueba se expande para incluir aquellas operaciones que se invocan mediante mensajes a clases que colaboran. Un enfoque alternativo divide las pruebas con base en las interfaces en una clase particular. En la figura 19.2, la clase **Bank** recibe mensajes de las clases **ATM** y **Cashier**. Por tanto, los métodos dentro de **Bank** pueden probarse al dividirlos en los que sirven a **ATM** y los que sirven a **Cashier**. La partición con base en estado (sección 19.5.2) puede usarse para refinar aún más las particiones.

### 19.6.2 Pruebas derivadas a partir de modelos de comportamiento

El uso del diagrama de estado como modelo que representa el comportamiento dinámico de una clase se analiza en el capítulo 7. El diagrama de estado para una clase puede usarse a fin de ayudar a derivar una secuencia de pruebas que ejercitarán el comportamiento dinámico de la clase (y de aquellas clases que colaboran con ella). La figura 19.3 [Kir94] ilustra un diagrama de estado para la clase **Account** estudiada anteriormente. En la figura, las transiciones iniciales se mueven a través de los estados *empty acct* (cuenta vacía) y *setup acct* (configuración de cuenta). La mayoría de los comportamientos para instancias de la clase ocurren mientras está en el estado *working acct* (cuenta operativa). Un retiro final y un cierre de cuenta harán que la clase **Account** realice transiciones hacia los estados *nonworking acct* (cuenta no operativa) y *dead acct* (cuenta muerta), respectivamente.

Las pruebas que se van a diseñar deben lograr cobertura de todos los estados, es decir, las secuencias de operación deben hacer que la clase **Account** realice transiciones a través de todos los estados permisibles:

*Caso de prueba* $s_1$: **open • setupAccnt • deposit (initial) • withdraw (final) • close**

Debe observarse que esta secuencia es idéntica a la de prueba mínima que se estudió en la sección 19.5.2. Al agregar secuencias de prueba adicionales a la secuencia mínima,

*Caso de prueba* $s_2$: **open • setupAccnt • deposit(initial) • deposit • balance • credit • withdraw (final) • close**
*Caso de prueba* $s_3$: **open • setupAccnt • deposit(initial) • deposit • withdraw • accntInfo • withdraw (final) • close**

Es necesario derivar todavía más casos de prueba para garantizar que todos los comportamientos para la clase se ejercitaron adecuadamente. En situaciones en las que el compor-

tamiento de clase da como resultado una colaboración con una o más clases, se usan diagramas de estado múltiple para rastrear el flujo de comportamiento del sistema.

El modelo de estado puede recorrerse en una forma "ancho primero" [McG94]. En este contexto, ancho primero implica que un caso de prueba ejercita una sola transición y que, cuando se prueba una nueva transición, sólo se usan transiciones previamente probadas.

Considere un objeto **CreditCard** que es parte del sistema bancario. El estado inicial de **CreditCard** es *indefinido* (es decir, no se proporcionó número de tarjeta de crédito). Hasta leer la tarjeta de crédito durante una venta, el objeto toma un estado *definido*, es decir, se definen los atributos **card number** y **expiration date**, junto con identificadores específicos del banco. La tarjeta de crédito se somete cuando se envía para autorización y se aprueba cuando se recibe la autorización. La transición de **CreditCard** de un estado a otro puede probarse al derivar casos de prueba que hacen que ocurra la transición. Un enfoque de ancho primero aplicado a este tipo de prueba no ejercitaría *submitted* antes de ejercitar *undefined* y *defined*. Si lo hiciera, usaría transiciones que no se probaron anteriormente y, por tanto, violaría el criterio de ancho primero.

## 19.7 Resumen

El objetivo global de las pruebas orientadas a objetos (encontrar el número máximo de errores con una cantidad mínima de esfuerzo) es idéntico al de la prueba de software convencional. Pero la estrategia y las tácticas de la prueba OO difieren significativamente. La visión de las pruebas se ensancha para incluir la revisión de los modelos de requerimientos y de diseño. Además, el foco de la prueba se mueve alejándose del componente procedimental (el módulo) y acercándose hacia la clase.

Puesto que los modelos de requerimientos y diseño OO y el código fuente resultante están semánticamente acoplados, la prueba (en la forma de revisiones técnicas) comienza durante la actividad de modelado. Por esta razón, la revisión de los modelos CRC, objeto-relación y objeto-comportamiento puede verse como pruebas de primera etapa.

Una vez disponible el código, la prueba de unidad se aplica para cada clase. El diseño de pruebas para una clase usa varios métodos: prueba basada en fallo, prueba aleatoria y prueba de partición. Cada uno de éstos ejercita las operaciones encapsuladas por la clase. Las secuencias de prueba se diseñan para garantizar que se ejercitan las operaciones relevantes. El estado de la clase, representado por los valores de sus atributos, se examina para determinar si existen errores.

La prueba de integración puede lograrse usando una estrategia basada en hebra o en uso. La prueba basada en hebra integra el conjunto de clases que colaboran para responder a una entrada o evento. La prueba basada en uso construye el sistema en capas, comenzando con aquellas clases que no utilizan clases servidor. La integración de métodos de diseño de caso de prueba también puede usar pruebas aleatorias y de partición. Además, la prueba basada en escenario y las pruebas derivadas de los modelos de comportamiento pueden usarse para probar una clase y a sus colaboradores. Una secuencia de prueba rastrea el flujo de operaciones a través de las colaboraciones de clase.

La prueba de validación del sistema OO está orientada a caja negra y puede lograrse al aplicar los mismos métodos de caja negra estudiados para el software convencional. Sin embargo, la prueba basada en escenario domina la validación de los sistemas OO, lo que hace al caso de uso un impulsor primario para la prueba de validación.

## Problemas y puntos por evaluar

**19.1.** Con sus palabras, describa por qué la clase es la unidad razonable más pequeña para probar dentro de un sistema OO.

**19.2.** ¿Por qué es necesario volver a probar las subclases que se instancian a partir de una clase existente si ésta ya se probó ampliamente? ¿Puede usarse el diseño de casos de prueba para la clase existente?

**19.3.** ¿Por qué la "prueba" debe comenzar con el análisis y el diseño orientado a objetos?

**19.4.** Derive un conjunto de tarjetas índice CRC para *CasaSegura* y realice los pasos anotados en la sección 19.2.2 para determinar si existen inconsistencias.

**19.5.** ¿Cuál es la diferencia entre las estrategias basadas en hebra y basadas en uso para la prueba de integración? ¿Cómo encaja la prueba de grupo?

**19.6.** Aplique pruebas aleatorias y de partición a tres clases definidas en el diseño del sistema *CasaSegura*. Produzca casos de prueba que indiquen las secuencias de operación que se invocarán.

**19.7.** Aplique prueba de clase múltiple y pruebas derivadas del modelo de comportamiento al diseño de *CasaSegura*.

**19.8.** Derive cuatro pruebas adicionales usando prueba aleatoria y métodos de partición, así como prueba de clase múltiple y pruebas derivadas del modelo de comportamiento, para la aplicación bancaria que se presentó en las secciones 19.5 y 19.6.

## Lecturas adicionales y fuentes de información

Muchos de los libros acerca de pruebas mencionados en las secciones *Lecturas adicionales y fuentes de información* de los capítulos 17 y 18 estudian en cierta medida las pruebas de los sistemas OO. Schach (*Object-Oriented and Classical Software Engineering*, McGraw-Hill, 6a. ed., 2004) considera la prueba OO dentro del contexto de una práctica de ingeniería de software más amplia. Sykes y McGregor (*Practical Guide to Testing Object-Oriented Software,* Addison-Wesley, 2001), Bashir y Goel (*Testing Object-Oriented Software,* Springer 2000), Binder (*Testing Object-Oriented Systems,* Addison-Wesley, 1999) y Kung *et al.* (*Testing Object-Oriented Software,* Wiley-IEEE Computer Society Press, 1998) tratan la prueba de OO con significativo detalle.

En internet está disponible gran variedad de fuentes de información acerca de métodos de prueba orientados a objeto. En el sitio del libro **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm** puede encontrarse una lista actualizada de referencias en la World Wide Web que son relevantes para las técnicas de prueba.

CAPÍTULO

# 20

# PRUEBA DE APLICACIONES WEB

CONCEPTOS CLAVE



Existe una urgencia que siempre impregna un proyecto web. Los participantes (intranquilos por la competencia de otras *webapps*, presionados por las demandas del cliente y preocupados porque perderán la ventana de mercado) fuerzan para poner la *webapp* en línea. Como consecuencia, en ocasiones desechan por completo las actividades técnicas que frecuentemente ocurren tarde en el proceso, como las pruebas de la aplicación web. Esto puede ser un error catastrófico. Para evitarlo, los miembros del equipo deben asegurarse de que cada producto resultante muestre alta calidad. Wallace *et al.* [Wal03] observan esto cuando afirman:

> Las pruebas no deben esperar hasta que el proyecto finalice. Comience a probar antes de escribir una línea de código. Pruebe constante y efectivamente, y desarrollará un sitio web mucho más duradero.

Los modelos de requerimientos y de diseño no pueden probarse en el sentido clásico: por ello, el equipo debe realizar revisiones técnicas (capítulo 15) y pruebas ejecutables. La intención es descubrir y corregir errores antes de que la *webapp* esté disponible para sus usuarios finales.

## 20.1 CONCEPTOS DE PRUEBAS PARA APLICACIONES WEB

Probar es el proceso de ejecución del software con la intención de encontrar (y a final de cuentas corregir) errores. Esta filosofía fundamental, que se expuso por primera vez en el capítulo 17, no cambia para las *webapps*. De hecho, puesto que los sistemas y las aplicaciones basadas en web residen en una red e interactúan con muchos sistemas operativos, navegado-

**UNA MIRADA RÁPIDA**

**¿Qué es?** La prueba de una *webapp* es una colección de actividades relacionadas con una sola meta: descubrir errores en el contenido, función, utilidad, navegabilidad, rendimiento, capacidad y seguridad de esa aplicación. Para lograr esto, se aplica una estrategia de prueba que abarca tanto revisiones como pruebas ejecutables.

**¿Quién lo hace?** En las pruebas de una *webapp* participan ingenieros en web y otros participantes en el proyecto (gestores, clientes y usuarios).

**¿Por qué es importante?** Si los usuarios finales encuentran errores que derrumben su fe en la *webapp*, irán a algún otro lado en busca del contenido y de la función que necesitan, y la aplicación fracasará. Por esta razón, debe trabajarse para eliminar tantos errores como sea posible antes de poner en línea la *webapp*.

**¿Cuáles son los pasos?** El proceso de prueba de una *webapp* comienza enfocándose en los aspectos visibles para el usuario de la aplicación y avanza hacia pruebas que ejercitan la tecnología y la infraestructura. Se realizan siete pasos durante la prueba: prueba de contenido, prueba de interfaz, prueba de navegación, prueba de componente, prueba de configuración, prueba de rendimiento y prueba de seguridad.

**¿Cuál es el producto final?** En algunas ocasiones, se produce un plan de prueba para la *webapp*. En todo caso, se desarrolla una suite de casos de prueba para cada paso de prueba y se mantiene un archivo de los resultados de la prueba para un uso futuro.

**¿Cómo me aseguro de que lo hice bien?** Aunque nunca se puede estar seguro de que se realizaron todas las pruebas que se necesitan, es posible tener la certeza de que se descubrieron errores (y se corrigieron). Además, si se estableció un plan de prueba, puede realizarse la comprobación para garantizar que todas las pruebas planeadas se llevaron a cabo.

res (residentes en varios dispositivos), plataformas de hardware, protocolos de comunicaciones y aplicaciones "de cuarto trasero" diferentes, la búsqueda de errores representa un reto significativo.

Para entender los objetivos de las pruebas dentro de un contexto de ingeniería web, debe considerar las muchas dimensiones de calidad de la *webapp*.[1] En el contexto de esta discusión, se consideran las dimensiones de calidad que son particularmente relevantes en cualquier análisis de las pruebas de la *webapp*. También se considera la naturaleza de los errores que se encuentran como consecuencia de las pruebas y la estrategia de prueba que se aplica 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 aplicando una serie de revisiones técnicas que valoran varios elementos del modelo de diseño y un proceso de prueba que se estudia a lo largo de este capítulo. Tanto las revisiones como las pruebas examinan una o más de las siguientes dimensiones de calidad [Mil00a]:

**? ¿Cómo se valora la calidad dentro del contexto de una *webapp* y su entorno?**

- El *contenido* se evalúa tanto en el nivel sintáctico como en el semántico. En el primero, se valora vocabulario, puntuación y gramática para documentos basados en texto. En el segundo, se valora la corrección (de la información presentada), la consistencia (a través de todo el objeto de contenido y de los objetos relacionados) y la falta de ambigüedad.
- La *función* se prueba para descubrir errores que indican falta de conformidad con los requerimientos del cliente. Cada función de la *webapp* se valora en su corrección, inestabilidad y conformidad general con estándares de implantación adecuados (por ejemplo, estándares de lenguaje Java o AJAX).
- La *estructura* se valora para garantizar que entrega adecuadamente el contenido y la función de la aplicación, que es extensible y que puede soportarse conforme se agregue nuevo contenido o funcionalidad.
- La *usabilidad* se prueba para asegurar que la interfaz soporta a cada categoría de usuario y que puede aprender y aplicar toda la sintaxis y semántica de navegación requerida.
- La *navegabilidad* se prueba para asegurar que toda la sintaxis y la semántica de navegación se ejecutan para descubrir cualquier error de navegación (por ejemplo, vínculos muertos, inadecuados y erróneos).
- El *rendimiento* se prueba bajo condiciones operativas, configuraciones y cargas diferentes a fin de asegurar que el sistema responde a la interacción con el usuario y que maneja la carga extrema sin degradación operativa inaceptable.
- La *compatibilidad* se prueba al ejecutar la *webapp* en varias configuraciones anfitrión, tanto en el cliente como en el servidor. La intención es encontrar errores que sean específicos de una configuración anfitrión única.
- La *interoperabilidad* se prueba para garantizar que la *webapp* tiene interfaz adecuada con otras aplicaciones y/o bases de datos.
- La *seguridad* se prueba al valorar las vulnerabilidades potenciales e intenta explotar cada una. Cualquier intento de penetración exitoso se estima como un fallo de seguridad.

**Cita:**

"La innovación es una negociación agridulce para los examinadores de software. Justo cuando parece que se sabe cómo probar una tecnología particular, aparece una nueva [*webapp*] y cualquier cosa puede ocurrir."

**James Bach**

La estrategia y las tácticas para probar las *webapps* se desarrollaron a fin de ejercitar cada una de estas dimensiones de calidad y se estudian más adelante, en este capítulo.

1 Las dimensiones genéricas de la calidad del software, igualmente válidas para las *webapps*, se estudiaron en el capítulo 14.

### 20.1.2 Errores dentro de un entorno de *webapp*

Los errores que se encuentran como consecuencia de una prueba exitosa de una *webapp* tienen algunas características únicas [Ngu00]:

**¿Qué hace que los errores encontrados durante la ejecución de una *webapp* sean un poco diferentes a los que se encuentran para el software convencional?**

1. Puesto que muchos tipos de pruebas de *webapps* descubren problemas que se evidencian primero en el lado del cliente (es decir, mediante una interfaz implantada en un navegador específico o en un dispositivo de comunicación personal), con frecuencia se ve un síntoma del error, no el error en sí.
2. Puesto que una *webapp* se implanta en algunas configuraciones distintas y dentro de diferentes entornos, puede ser difícil o imposible reproducir un error afuera del entorno en el que originalmente se encontró.
3. Aunque algunos errores son resultado de diseño incorrecto o codificación HTML (u otro lenguaje de programación) impropia, muchos errores pueden rastrearse en la configuración de la *webapp*.
4. Dado que las *webapps* residen dentro de una arquitectura cliente-servidor, los errores pueden ser difíciles de rastrear a través de tres capas arquitectónicas: el cliente, el servidor o la red en sí.
5. Algunos errores se deben al *entorno operativo estático* (es decir, a la configuración específica donde se realiza la prueba), mientras que otros son atribuibles al entorno operativo dinámico (es decir, a la carga de recurso instantánea o a errores relacionados con el tiempo).

Estos cinco atributos de error sugieren que el entorno juega un importante papel en el diagnóstico de todos los errores descubiertos durante la prueba de *webapps*. En algunas situaciones (por ejemplo, la prueba de contenido), el sitio del error es obvio, pero en muchos otros tipos de prueba de *webapps* (por ejemplo, prueba de navegación, prueba de rendimiento, prueba de seguridad), la causa subyacente del error puede ser considerablemente más difícil de determinar.

### 20.1.3 Estrategia de las pruebas

La estrategia para probar *webapps* adopta los principios básicos de todas las pruebas de software (capítulo 17) y aplica una estrategia y las tácticas que se recomendaron para los sistemas orientados a objetos (capítulo 19). Los siguientes pasos resumen el enfoque:

**PUNTO CLAVE**

La estrategia global para probar *webapp* puede resumirse en los 10 pasos que se anotan aquí.

1. El modelo de contenido para la *webapp* a se revisa a fin de descubrir errores.
2. El modelo de interfaz se examina para garantizar que todos los casos de uso pueden alojarse.
3. El modelo de diseño para la *webapp* se revisa para descubrir errores de navegación.
4. La interfaz de usuario se prueba para descubrir errores en la mecánica de presentación y/o navegación.
5. Los componentes funcionales se someten a prueba de unidad.
6. Se prueba la navegación a lo largo de toda la arquitectura.
7. La *webapp* se implanta en varias configuraciones de entorno diferentes y se prueba para asegurar la compatibilidad con cada configuración.
8. Las pruebas de seguridad se realizan con la intención de explotar las vulnerabilidades en la *webapp* o dentro de su entorno.

**WebRef**

En **www.stickyminds.com/testing.asp** se encuentran excelentes artículos acerca de pruebas de *webapps*

9. Se realizan pruebas de rendimiento.
10. La *webapp* se prueba con una población controlada y monitoreada de usuarios finales; los resultados de su interacción con el sistema se evalúan para detectar errores de contenido y de navegación, preocupaciones de usabilidad y compatibilidad, y seguridad, confiabilidad y rendimiento de la *webapp*.

Puesto que muchas *webapps* evolucionan continuamente, el proceso de prueba es una actividad siempre en marcha que realiza el personal de apoyo web, quien usa pruebas de regresión derivadas de las pruebas desarrolladas cuando comenzó la ingeniería de las *webapps*.

### 20.1.4 Planificación de pruebas

El uso de la palabra *planificación* (en cualquier contexto) es un anatema para algunos desarrolladores web que no planifican; sólo arrancan, con la esperanza de que surja una *webapp* asesina. Un enfoque más disciplinado reconoce que la planificación establece un mapa de ruta para todo el trabajo que va después. Vale la pena el esfuerzo. En su libro acerca de las pruebas de *webapps*, Splaine y Jaskiel [Spl01] afirman:

> Excepto por el más simple de los sitios web, rápidamente resulta claro que es necesaria alguna especie de planificación de pruebas. Con demasiada frecuencia, el número inicial de errores encontrados a partir de una prueba *ad hoc* es suficientemente grande como para que no todos se corrijan la primera vez que se detectan. Esto impone una carga adicional sobre el personal que prueba sitios y *webapps*. No sólo deben idear nuevas pruebas imaginativas, sino que también deben recordar cómo se ejecutaron las pruebas anteriores con la finalidad de volver a probar de manera confiable el sitio/*webapp*, y garantizar que se removieron los errores conocidos y que no se introdujeron algunos nuevos.

Las preguntas que deben plantearse son: ¿cómo se "idean nuevas pruebas imaginativas" y sobre qué deben enfocarse dichas pruebas? Las respuestas a estas preguntas se integran en un plan de prueba que identifica: 1) el conjunto de tareas[2] que se van a aplicar cuando comiencen las pruebas, 2) los productos de trabajo que se van a producir conforme se ejecuta cada tarea de prueba y 3) la forma en la que se evalúan, registran y reutilizan los resultados de la prueba cuando se realizan pruebas de regresión. En algunos casos, el plan de prueba se integra con el plan del proyecto. En otros, es un documento separado.

**PUNTO CLAVE**

El plan de prueba identifica el conjunto de tareas de pruebas, los productos de trabajo que se van a desarrollar y la forma en la que deben evaluarse, registrarse y reutilizarse los resultados.

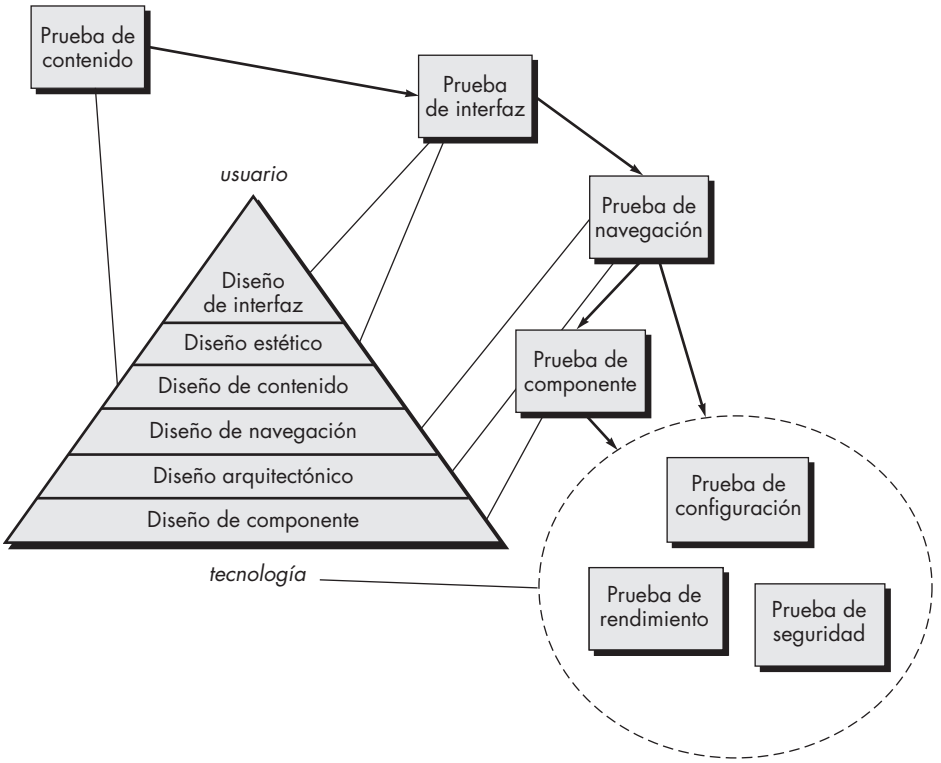
## 20.2 Un panorama del proceso de prueba

El proceso de prueba de *webapps* comienza con pruebas que ejercitan la funcionalidad del contenido y la interfaz que son inmediatamente visibles para el usuario final. Conforme avanza la prueba, se ejercitan aspectos de la arquitectura del diseño y de la navegación. Finalmente, la atención se centra en las pruebas que examinan las capacidades tecnológicas que no siempre son aparentes para los usuarios finales: los temas de infraestructura e instalación/ implantación de la *webapp*.

La figura 20.1 yuxtapone el proceso de prueba de la *webapp* con la pirámide de diseño para este tipo de aplicaciones (capítulo 13). Observe que, conforme el flujo de la prueba avanza de izquierda a derecha y de arriba abajo, los elementos visibles para el usuario del diseño de la *webapp* (elementos superiores de la pirámide) se prueban primero, seguidos por los elementos de diseño de infraestructura.

2 Los conjuntos de tareas se estudian en el capítulo 2. También se usa un término relacionado, *flujo de trabajo*, para describir una serie de tareas requeridas para lograr una actividad de ingeniería del software.

**FIGURA 20.1**
**El proceso de prueba**




## 20.3 PRUEBA DE CONTENIDO

Los errores en el contenido de la *webapp* pueden ser tan triviales como errores tipográficos menores o tan significativos como información incorrecta, organización inadecuada o violación de leyes de la propiedad intelectual. La *prueba de contenido* intenta descubrir éstos y muchos otros problemas antes de que el usuario los encuentre.

CONSEJO

*Aunque las revisiones técnicas no son parte de las pruebas, debe realizarse la revisión del contenido para garantizar que éste tiene calidad.*

La prueba de contenido combina tanto revisiones como generación de casos de prueba ejecutables. Las revisiones se aplican para descubrir errores semánticos en el contenido (que se estudia en la sección 20.3.1). Las pruebas ejecutables se usan para descubrir errores de contenido que puedan rastrearse a fin de derivar dinámicamente contenido que se impulse por 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 de contenido tiene tres objetivos importantes: 1) descubrir errores sintácticos (por ejemplo, errores tipográficos o gramaticales) en documentos de texto, representaciones gráficas y otros medios; 2) descubrir errores semánticos (es decir, errores en la precisión o completitud de la información) en cualquier objeto de contenido que se presente conforme ocurre la navegación y 3) encontrar errores en la organización o estructura del contenido que se presenta al usuario final.

Para lograr el primer objetivo, pueden usarse correctores automáticos de vocabulario y gramática. Sin embargo, muchos errores sintácticos evaden la detección de tales herramientas y los debe descubrir un revisor humano (examinador). De hecho, un sitio web grande debe considerar los servicios de un editor profesional para descubrir errores tipográficos, gazapos gramaticales, errores en la consistencia del contenido, errores en las representaciones gráficas y en referencias cruzadas.

La prueba semántica se enfoca en la información presentada dentro de cada objeto de contenido. El revisor (examinador) debe responder las siguientes preguntas:

**¿Qué preguntas deben plantearse y responderse para descubrir errores semánticos en el contenido?**

- ¿La información realmente es precisa?
- ¿La información es concisa y puntual?
- ¿La plantilla del objeto de contenido es fácil de comprender para el usuario?
- ¿La información incrustada dentro de un objeto de contenido puede encontrarse con facilidad?
- ¿Se proporcionaron referencias adecuadas para toda la información derivada de otras fuentes?
- ¿La información presentada es consistente internamente y con la información presentada en otros objetos de contenido?
- ¿El contenido es ofensivo, confuso o abre la puerta a demandas?
- ¿El contenido infringe derechos de autor o nombres comerciales existentes?
- ¿El contenido incluye 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?

**Cita:**

"En general, las técnicas de prueba del software que se emplean en otras aplicaciones son las mismas que las usadas en aplicaciones basadas en web [...] La diferencia [...] es que las variables tecnológicas en el entorno web se multiplican."

**Hung Nguyen**

Obtener respuestas a cada una de estas preguntas para una gran *webapp* (que contiene cientos de objetos de contenido) puede ser una tarea atemorizante. Sin embargo, el fracaso para descubrir los errores semánticos sacudirá la fe del usuario en la *webapp* y puede conducir al fracaso de la aplicación basada en web.

Los objetos de contenido existen dentro de una arquitectura que tiene un estilo específico (capítulo 13). Durante la prueba de contenido, la estructura y organización de la arquitectura de contenido se prueba para garantizar que el contenido requerido se presente al usuario final en el orden y relaciones adecuados. Por ejemplo, la *webapp* **CasaSeguraAsegurada.com** presenta información variada acerca de los sensores que se utilizan como parte de los 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 **CasaSeguraAsegurada.com** luchan 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 base de datos

Las *webapps* modernas hacen mucho más que presentar objetos de contenido estáticos. En muchos dominios de aplicación, la *webapp* tiene interfaz con sofisticados sistemas de gestión de base de datos y construyen objetos de contenido dinámico que se crean en tiempo real, usando los datos adquiridos desde una base de datos.

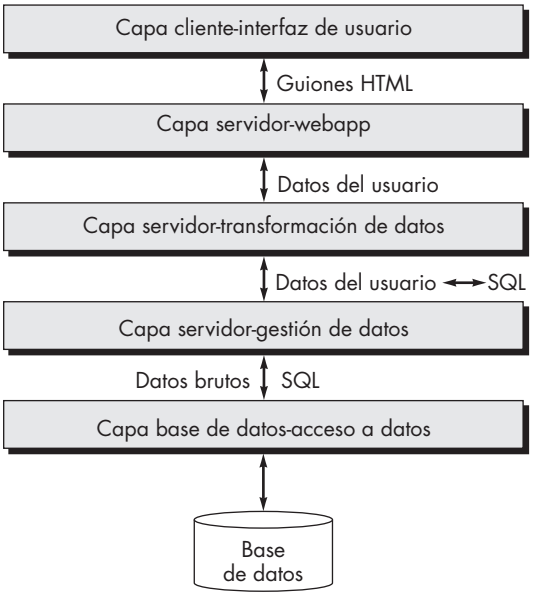
Por ejemplo, una *webapp* de servicios financieros puede producir información compleja basada en texto, tablas tabulares y gráficas acerca de un fondo específico (por ejemplo, una acción o fondo mutualista). El objeto de contenido compuesto que presenta esta información se crea de manera dinámica después de que el usuario hace una solicitud de información acerca de un fondo específico. Para lograrlo, se requieren los siguientes pasos: 1) consulta a una gran base de datos de fondos, 2) extracción de datos relevantes de la base de datos, 3) organización de los datos extraídos como un objeto de contenido y 4) transmisión de este objeto de contenido (que representa información personalizada que requiere un usuario final) al entorno del cliente para su despliegue. Los errores pueden ocurrir, y ocurren, como consecuencia de cada uno de estos pasos. El objeto de la prueba de la base de datos es descubrir dichos errores, pero esta prueba es complicada por varios factores:

**¿Qué cosas complican la prueba de base de datos para *webapps*?**

1. *El lado cliente original solicita información que rara vez se presenta en la forma [por ejemplo, lenguaje de consulta estructurado (SQL)] en la que puede ingresarse a un sistema de gestión de base de datos (DBMS).* Por tanto, las pruebas deben diseñarse para descubrir errores cometidos al traducir la solicitud del usuario de manera que pueda procesar el DBMS.
2. *La base de datos puede ser remota en relación con el servidor que alberga la webapp.* En consecuencia, deben desarrollarse pruebas que descubran errores en la comunicación entre la *webapp* y la base de datos remota.[3]
3. *Los datos brutos adquiridos de la base de datos deben transmitirse al servidor de la webapp y formatearse de manera adecuada para su posterior transmisión al cliente.* Por tanto, deben desarrollarse pruebas que demuestren la validez de los datos brutos recibidos por el servidor de la *webapp* y también deben crearse pruebas adicionales que demuestren la validez de las transformaciones aplicadas a los datos brutos para crear objetos de contenido válidos.
4. *El objeto de contenido dinámico debe transmitirse al cliente de forma que pueda desplegarse al usuario final.* Por ende, debe diseñarse una serie de pruebas para 1) descubrir errores en el formato del objeto de contenido y 2) probar la compatibilidad con diferentes configuraciones del entorno del cliente.

Al considerar estos cuatro factores, los métodos de diseño de caso de prueba deben aplicarse a cada una de las "capas de interacción" [Ngu01] que se mencionan en la figura 20.2. Las pruebas deben garantizar que 1) información válida pasa entre el cliente y el servidor desde la capa interfaz, 2) la *webapp* procesa los guiones de manera correcta y extrae o formatea adecuadamente los datos del usuario, 3) los datos del usuario pasan correctamente a una función de transformación de datos del lado servidor que formatea consultas adecuadas (por ejemplo, SQL) y 4) las

**FIGURA 20.2**
**Capas de interacción**




3 Estos datos pueden volverse complejos cuando se encuentran bases de datos distribuidas o cuando se requiere el acceso a un almacén de datos (capítulo 1).

**Cita:**

"... es improbable que uno tenga confianza en un sitio web que sufre de constantes periodos de inactividad, que se pasma en medio de una transacción o que tiene una pobre sensación de funcionalidad. Por tanto, las pruebas tienen un papel vital en el proceso de desarrollo global."

**Wing Lam**

consultas pasan a una capa de gestión de datos[4] que se comunica con las rutinas de acceso a la base de datos (potencialmente ubicadas en otra máquina).

Las capas de transformación de datos, de gestión de datos y de acceso a base de datos que se muestran en la figura 20.2, con frecuencia se construyen con componentes reutilizables que se validaron por separado y como paquete. Si éste es el caso, la prueba de *webapps* se enfoca en el diseño de casos de prueba para ejercitar las interacciones entre la capa cliente y las primeras dos capas servidor (*webapp* y transformación de datos) que se muestran en la figura.

La capa de interfaz de usuario se prueba para garantizar que los guiones se construyeron de manera adecuada para cada consulta de usuario y que transmiten adecuadamente al lado servidor. La capa *webapp* en el lado servidor se prueba para asegurar que los datos de usuario se extraen de manera adecuada de los guiones y que se transmite adecuadamente a la capa de transformación de datos en el lado servidor. Las funciones de transformación de datos se prueban para asegurar que se creó el SQL correcto y que pasó a componentes de gestión de datos adecuados.

Un análisis detallado de la tecnología subyacente que debe comprenderse para diseñar adecuadamente estas pruebas de base de datos está más allá del ámbito de este libro. Si usted tiene interés adicional, vea [Sce02], [Ngu01] y [Bro01].

## 20.4 Prueba de interfaz de usuario

La verificación y validación de una interfaz de usuario de *webapp* ocurre en tres puntos distintos. Durante el análisis de requerimientos, el modelo de interfaz se revisa para garantizar que se da conformidad a los requerimientos de los participantes y a otros elementos del modelo de requerimientos. Durante el diseño, se revisa el modelo de diseño de interfaz para garantizar que se logran los criterios de calidad genéricos establecidos para todas las interfaces de usuario (capítulo 11) y que los temas de diseño de interfaz específicos de la aplicación se abordaron de manera adecuada. Durante la prueba, la atención se centra en la ejecución de aspectos específicos de la aplicación de la interacción con el usuario, conforme se manifiesten por la sintaxis y la semántica de la interfaz. Además, la prueba proporciona una valoración final de la usabilidad.

**CONSEJO**

*Con excepción de especificaciones orientadas a webapp, la estrategia de interfaz que se anota aquí es aplicable a todo tipo de software cliente-servidor.*

### 20.4.1 Estrategia de prueba de interfaz

La *prueba de interfaz* ejercita los mecanismos de interacción y valida los aspectos estéticos de la interfaz de usuario. La estrategia global para la prueba de interfaz es 1) descubrir errores relacionados con mecanismos de interfaz específicos (por ejemplo, en la ejecución adecuada de un vínculo de menú o en la forma como entran los datos en un formulario) y 2) descubrir errores en la forma como la interfaz implanta la semántica de navegación, la funcionalidad de la *webapp* o el despliegue de contenido. Para lograr esta estrategia, se inician algunos pasos tácticos:

- *Las características de la interfaz se prueban para garantizar que las reglas del diseño, estética y contenido visual relacionado estén disponibles sin error para el usuario.* Las características incluyen tipo de fuente, uso de color, marcos, imágenes, bordes, tablas y características de interfaz relacionadas que se generan conforme avanza la ejecución de la *webapp*.
- *Los mecanismos de interfaz individuales se prueban en forma análoga a la prueba de unidad.* Por ejemplo, las pruebas se diseñan para ejercitar todas las formas, guiones del lado cliente, HTML dinámicos, guiones, contenido de *streaming* (transmisión continua) y

4 La capa de gestión de datos por lo general incorpora una interfaz SQL en el nivel de llamado (SQL-CLI), como Microsoft OLE/ADO o Java Database Connectivity (JDBC).

mecanismos de interfaz específicos de la aplicación (por ejemplo, un carro de mandado para una aplicación de comercio electrónico). En muchos casos, la prueba puede enfocarse exclusivamente en uno de estos mecanismos (la "unidad") y excluir otras características y funciones de interfaz.

- *Cada mecanismo de interfaz se prueba dentro del contexto de un caso de uso o de una unidad semántica de navegación (USN)* (capítulo 13) *para una categoría de usuario específica.* Este enfoque de pruebas es análogo a la prueba de integración porque las pruebas se realizan conforme los mecanismos de interfaz se integran para permitir la ejecución de un caso de uso o USN.
- *La interfaz completa se prueba contra los casos de uso seleccionados y las USN a fin de descubrir errores en la semántica de la interfaz.* Este enfoque de prueba es análogo a la prueba de validación porque el propósito es demostrar conformidad con la semántica de casos de uso o USN específicas. En esta etapa se lleva a cabo una serie de pruebas de usabilidad.
- *La interfaz se prueba dentro de varios entornos (por ejemplo, navegadores) para garantizar que será compatible.* En realidad, esta serie de pruebas también puede considerarse como parte de las pruebas de configuración.

### 20.4.2 Prueba de mecanismos de interfaz

Cuando un usuario interactúa con una *webapp*, la interacción ocurre a través de uno o más mecanismos de interfaz. En los párrafos que siguen se presenta un breve panorama de las consideraciones de prueba para cada mecanismo de interfaz [Spl01].


*La prueba de vínculos externos debe ocurrir durante la vida de la webapp. Parte de una estrategia de apoyo debe ser la calendarización regular de pruebas de vínculos.*

**Vínculos.** Cada vínculo de navegación se prueba para garantizar que se alcanza el objetivo de contenido o función apropiados.[5] Se construye una lista de todos los vínculos asociados con la plantilla de interfaz (por ejemplo, barras de menú e ítems de índice) y luego se ejecuta cada uno individualmente. Además, deben ejercitarse los vínculos dentro de cada objeto de contenido para descubrir URL o vínculos defectuosos con objetos de contenido o funciones inadecuadas. Finalmente, los vínculos con *webapps* externas deben probarse en su precisión y también evaluarse para determinar el riesgo de que se vuelvan inválidos con el tiempo.

**Formularios.** En un nivel macroscópico, las pruebas se realizan para asegurarse de que 1) las etiquetas identifican correctamente los campos dentro del formulario y los campos obligatorios se identifican visualmente para el usuario, 2) el servidor recibe toda la información contenida dentro del formulario y ningún dato se pierde en la transmisión entre cliente y servidor, 3) se usan 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 la entrada de datos en un formulario y 5) los guiones que realizan la comprobación de errores en los datos ingresados funcionan de manera adecuada y proporcionan mensajes de error significativos.

En un nivel más dirigido, las pruebas deben garantizar que 1) los campos del formulario tienen ancho y tipos de datos adecuados, 2) el formulario establece salvaguardas adecuadas que prohíben que el usuario ingrese cadenas de texto más largas que cierto máximo predefinido, 3) todas las opciones adecuadas para menús desplegables se especifican y ordenan en forma significativa para el usuario final, 4) las características de "autollenado" del navegador no conducen a errores en la entrada de datos y 5) la tecla de tabulación (o alguna otra) inicia el movimiento adecuado entre los campos del formulario.

5 Estas pruebas pueden realizarse como parte de la prueba de interfaz o de navegación.


*Las pruebas de guión en el lado cliente y las pruebas asociadas con HTML dinámico deben repetirse siempre que se libera una nueva versión de un navegador popular.*

**Guión en el lado cliente.** Las pruebas de caja negra se realizan para descubrir cualquier error en el procesamiento conforme se ejecuta el guión. Estas pruebas con frecuencia se acoplan con pruebas de formularios porque la entrada del guión con frecuencia se deriva de los datos proporcionados como parte del procesamiento de formulario. Debe realizarse una prueba de compatibilidad para garantizar que el lenguaje del guión elegido funcionará adecuadamente en las configuraciones de entorno que soporten la *webapp*. Además de probar el guión en sí, Splaine y Jaskiel [Spl01] sugieren que "debe asegurarse de que los estándares [de *webapps*] de la compañía enuncien el lenguaje y versión preferidos del lenguaje de guión que se va a usar para la escritura de guiones en el lado cliente (y en el lado servidor)".

**HTML dinámico.** Cada página web que contenga HTML dinámico se ejecuta para asegurar que el despliegue dinámico es correcto. Además, debe llevarse a cabo una prueba de compatibilidad para asegurarse que el HTML dinámico funciona adecuadamente en las configuraciones de entorno que soportan la *webapp*.

**Ventanas *pop-up*.** Una serie de pruebas garantiza que 1) la aparición instantánea tiene el tamaño y posición adecuadas, 2) la aparición no cubre la ventana de la *webapp* original, 3) el diseño estético de la aparición es consistente con el diseño estético de la interfaz y 4) las barras de desplazamiento y otros mecanismos de control anexados a la ventana de aparición se ubican y funcionan de manera adecuada, como se requiere.

**Guiones CGI.** Las pruebas de caja negra se realizan con énfasis sobre la integridad de los datos (conforme los datos pasan al guión CGI) y del procesamiento del guión (una vez recibidos los datos validados). Además, la prueba de rendimiento puede realizarse para garantizar que la configuración del lado servidor puede alojar las demandas de procesamiento de múltiples invocaciones de los guiones CGI [Spl01].

**Contenido de *streaming*.** Las pruebas deben demostrar que los datos de *streaming* están actualizados, que se despliegan de manera adecuada y que pueden suspenderse sin error y reanudarse sin dificultad.

***Cookies*.** Se requieren pruebas tanto del lado servidor como del lado cliente. En el primero, las pruebas deben garantizar que una *cookie* se construyó adecuadamente (que contiene datos correctos) y que se transmitió de manera adecuada al lado cliente cuando se solicitó contenido o funcionalidad específico. Además, la persistencia adecuada de la *cookie* se prueba para asegurar que su fecha de expiración es correcta. En el lado cliente, las pruebas determinan si la *webapp* liga adecuadamente las *cookies* existentes a una solicitud específica (enviada al servidor).

**Mecanismos de interfaz específicos de aplicación.** Las pruebas se siguen conforme una lista de comprobación de funcionalidad y características que se definen mediante el mecanismo de interfaz. Por ejemplo, Splaine y Jaskiel [Spl01] sugieren la siguiente lista de comprobación para la funcionalidad carro de compras definida para una aplicación de comercio electrónico:

- Prueba de frontera (capítulo 18) del número mínimo y máximo de artículos que pueden colocarse en el carro de compras.
- Prueba de una solicitud de "salida" para un carro de compras vacío.
- Prueba de borrado adecuado de un artículo del carro de compras.
- Prueba para determinar si una compra vacía el contenido del carro.
- Prueba para determinar la persistencia del contenido del carro de compras (esto debe especificarse como parte de los requerimientos del cliente).
- Prueba para determinar si la *webapp* puede recordar el contenido del carro de compras en alguna fecha futura (suponiendo que no se realizó compra alguna).

### 20.4.3 Prueba de la semántica de la interfaz

Una vez que cada mecanismo de interfaz ha sido sometido a prueba de "unidad", la atención de la prueba de interfaz cambia hacia una consideración de la semántica de la interfaz. Esta prueba "evalúa cuán bien cuida el diseño a los usuarios, ofrece instrucciones claras, entrega retroalimentación y mantiene consistencia de lenguaje y enfoque" [Ngu00].

Una revisión profunda del modelo de diseño de interfaz puede proporcionar respuestas parciales a las preguntas implicadas en el párrafo precedente. Sin embargo, cada escenario de caso de uso (para cada categoría de usuario) debe probarse una vez implantada la *webapp*. En esencia, un caso de uso se convierte en la entrada para el diseño de una secuencia de prueba. La intención de la secuencia de prueba es descubrir errores que evitarán que un usuario logre el objetivo asociado con el caso de uso.

Conforme cada caso de uso se prueba, es buena idea mantener una lista de comprobación para asegurar que cada objeto del menú se ejercitó al menos una vez y que se utilizó cada vínculo incrustado dentro de un objeto de contenido. Además, la serie de pruebas debe incluir selección de menú inadecuada y uso de vínculos. La intención es determinar si la *webapp* proporciona manejo y recuperación efectivos del error.

### 20.4.4 Pruebas de usabilidad

**WebRef**

En **www.ahref.com/guides/design/199806/0615jef.html** se encuentra una valiosa guía para las pruebas de usabilidad.

La prueba de usabilidad es similar a la de semántica de interfaz (sección 20.4.3) porque también evalúa el grado en el cual los usuarios pueden interactuar efectivamente con la *webapp* y el grado en el que la *webapp* guía las acciones del usuario, proporciona retroalimentación significativa y refuerza un enfoque de interacción consistente. En lugar de enfocarse atentamente en la semántica de algún objetivo interactivo, las revisiones y pruebas de usabilidad se diseñan para determinar el grado en el cual la interfaz de la *webapp* facilita la vida del usuario.[6]

Invariablemente, el ingeniero en software contribuirá con el diseño de las pruebas de usabilidad, pero las pruebas en sí las realizan los usuarios finales. La siguiente secuencia de pasos es aplicable para tal fin [Spl01]:

1. Definir un conjunto de categorías de prueba de usabilidad e identificar las metas de cada una.
2. Diseñar pruebas que permitirán la evaluación de cada meta.
3. Seleccionar a los participantes que realicen 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 usabilidad de la *webapp*.

La prueba de usabilidad puede ocurrir en varios niveles diferentes de abstracción: 1) puede valorarse la usabilidad de un mecanismo de interfaz específico (por ejemplo, un formulario), 2) puede evaluarse la usabilidad de una página web completa (que abarque mecanismos de interfaz, objetos de datos y funciones relacionadas) y 3) puede considerarse la usabilidad de la *webapp* completa.

El primer paso en la prueba de usabilidad es identificar un conjunto de categorías de usabilidad y establecer los objetivos de la prueba para cada categoría. Las siguientes categorías y objetivos de prueba (escritos en forma de pregunta) ilustran este enfoque:[7]

6 En este contexto se ha usado el término *amigable con el usuario*. Desde luego, el problema es que la percepción de un usuario acerca de una interfaz "amigable" puede ser radicalmente diferente a la de otro.

7 Para información adicional acerca de la usabilidad, vea el capítulo 11.

**¿Qué características de la usabilidad se convierten en el centro de atención de las pruebas y qué objetivos específicos se señalan?**

*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 se colocan de forma que el usuario pueda encontrarlos rápidamente?

*Legibilidad:* ¿El texto está bien escrito y es comprensible?[8] ¿Las representaciones gráficas se entienden con facilidad?

*Estética:* ¿La plantilla, color, fuente y características relacionadas facilitan el uso? ¿Los usuarios "se sienten cómodos" con la apariencia y el sentimiento de la *webapp*?

*Características de despliegue:* ¿La *webapp* usa de manera óptima el tamaño y la resolución de la pantalla?

*Sensibilidad temporal:* ¿Las características, funciones y contenido importantes pueden usarse o adquirir en forma oportuna?

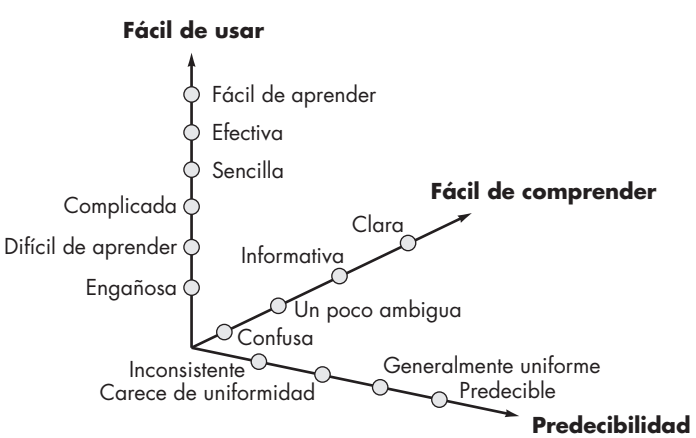
*Personalización:* ¿La *webapp* se adapta a las necesidades específicas de diferentes categorías de usuario o de usuarios individuales?

*Accesibilidad:* ¿La *webapp* es accesible a personas que tienen 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, pueden ejecutarse de nuevo pruebas semánticas de la interfaz, pero en esta instancia las preocupaciones por la usabilidad son primordiales.

Como ejemplo, considere la valoración de usabilidad para los mecanismos de interacción e interfaz. Constantine y Lockwood [Con99] sugieren que debe revisarse la siguiente lista de características de interfaz y probar la usabilidad: animación, botones, color, control, diálogo, campos, formularios, marcos, gráficos, etiquetas, vínculos, menús, mensajes, navegación, páginas, selectores, texto y barras de herramientas. Conforme se valora cada característica, es calificada por los usuarios que realizan la prueba sobre una escala cualitativa. La figura 20.3 muestra un posible conjunto de "calificaciones" de valoración que pueden seleccionar los usuarios, mismas que se aplican a cada característica individualmente, a una página web completa o a la *webapp* como un todo.

**FIGURA 20.3**
**Valoración cualitativa de la usabilidad**




8 Puede usarse el índice de legibilidad FOG y otros para proporcionar una valoración cuantitativa de la legibilidad. Véase **http://developer.gnome.org/documents/usability/usability-readability.html** para más detalles.

### 20.4.5 Pruebas de compatibilidad

Diferentes computadoras, dispositivos de despliegue, sistemas operativos, navegadores y velocidades de conexión de red pueden tener influencia significativa sobre la operación de una *webapp*. Cada configuración de cómputo puede dar como resultado diferencias en velocidades de procesamiento en el lado cliente, en resolución de despliegue y en velocidades de conexión. Los caprichos de los sistemas operativos en ocasiones pueden producir conflictos de procesamiento en la *webapp*. En ocasiones, diferentes navegadores producen resultados ligeramente distintos, sin importar el grado de estandarización HTML dentro de la *webapp*. Los plug-ins requeridos pueden o no conseguirse con facilidad para una configuración particular.

**PUNTO CLAVE**

Las *webapps* se ejecutan dentro de varios entornos en el lado cliente. El objetivo de la prueba de compatibilidad es descubrir errores asociados con un entorno específico (por ejemplo, navegador).

En algunos casos, pequeños conflictos de compatibilidad no representan problemas significativos, pero en otros pueden encontrarse serios errores. Por ejemplo, las velocidades de descarga pueden volverse inaceptables, carecer de un plug-in requerido puede hacer que el contenido no esté disponible, las diferencias de navegador pueden cambiar dramáticamente la plantilla de la página, los estilos de fuente pueden alterarse y volverse ilegibles o los formularios pueden organizarse de manera inadecuada. La *prueba de compatibilidad* busca descubrir dichos problemas antes de que la *webapp* esté en línea.

El primer paso en la prueba de compatibilidad es definir un conjunto de configuraciones de cómputo, y sus variantes, que "se encuentran comúnmente" en el lado cliente. En esencia, se crea una estructura de árbol, identificación de cada plataforma de cómputo, dispositivos de despliegue usuales, sistemas operativos aceptados en la plataforma, navegadores disponibles, probables velocidades de conexión a internet e información similar. A continuación se deriva una serie de pruebas de validación de compatibilidad, con frecuencia adaptadas de pruebas de interfaz existentes, de navegación, de rendimiento y de seguridad. La intención de estas pruebas es descubrir errores o problemas de ejecución que pueden rastrearse para identificar diferencias de configuración.

**CASASEGURA**

#### *Prueba de webapp*

**La escena:** Oficina de Doug Miller.

**Participantes:** Doug Miller (gerente del grupo de ingeniería del software *CasaSegura*) y Vinod Raman (miembro del equipo de ingeniería del software del producto).

**La conversación:**

**Doug:** ¿Qué piensas de la versión 0.0 de la *webapp* de comercio electrónico para **CasaSeguraAsegurada.com**?

**Vinod:** El proveedor subcontratado hizo un buen trabajo. Sharon [gerente de desarrollo del proveedor] me dijo que ahora están haciendo pruebas.

**Doug:** Me gustaría que tú y el resto del equipo hicieran una prueba un poco informal en el sitio de comercio electrónico.

**Vinod (hace muecas):** Creo que vamos a contratar una compañía externa para validar la *webapp*. Todavía nos estamos matando en el intento de poner el producto en línea.

**Doug:** Vamos a contratar a un proveedor de pruebas para las pruebas de rendimiento y seguridad, y nuestro proveedor subcontratado ya está haciendo pruebas. Sólo pienso que otro punto de vista sería útil y, además, nos gustaría conservar los costos en línea, de modo que...

**Vinod (suspira):** ¿Qué buscas?

**Doug:** Quiero estar seguro de que la interfaz y toda la navegación son sólidas.

**Vinod:** Supongo que podemos comenzar con los casos de uso para cada una de las principales funciones de interfaz:

**Aprenda acerca de *CasaSegura*.**
**Especifique el sistema *CasaSegura* que necesita.**
**Compre un sistema *CasaSegura*.**
**Obtenga soporte técnico.**

**Doug:** Bien. Pero sigan las rutas de navegación durante todo el trayecto hasta su conclusión.

**Vinod (observa un cuaderno de casos de uso):** Sí, cuando seleccionas Especifique el sistema CasaSegura que necesita, eso te llevará hacia:

**Seleccione componentes CasaSegura.**

**Obtenga recomendaciones de componentes *CasaSegura***

Podemos ejercitar la semántica de cada ruta.

**Doug:** Mientras estás ahí, verifica el contenido que aparece en cada nodo de navegación.

**Vinod:** Desde luego... y los elementos funcionales también. ¿Quién prueba la usabilidad?

**Doug:** Oh... el proveedor examinador coordinará la prueba de usabilidad. Contratamos una firma de investigación de mercado a fin de alinear 20 usuarios comunes para al estudio de usabilidad, pero si tus chicos descubren algún conflicto de usabilidad...

**Vinod:** Ya sé, pásenlos de largo.

**Doug:** Gracias, Vinod.

## 20.5 Prueba en el nivel de componente

La *prueba en el nivel de componente*, también llamada *prueba de función*, se enfoca en un conjunto de pruebas que intentan descubrir errores en funciones de las *webapps*. Cada función de una *webapp* es un componente de software (implantado en uno de varios lenguajes de programación o lenguajes de guiones) y puede probarse usando técnicas de caja negra (y en algunos casos de caja blanca), como se estudió en el capítulo 18.

Los casos de prueba en el nivel de componente con frecuencia se derivan de la entrada a formularios. Una vez definidos los datos de los formularios, el usuario selecciona un botón u otro mecanismo de control para iniciar la ejecución. Son usuales los siguientes métodos de diseño de caso de prueba (capítulo 18):

- *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 casos de prueba. El formulario de entrada se valora para determinar cuáles clases de datos son relevantes para la función. Los casos de prueba para cada clase de entrada se derivan y ejecutan, mientras que otras clases de entrada se mantienen constantes. Por ejemplo, una aplicación de comercio electrónico puede implantar una función que calcule los cargos de embarque. Entre una variedad de información de embarque proporcionada mediante un formulario, está el código postal del usuario. Los casos de prueba se diseñan con la intención 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 incorrecto, un código postal inexistente, un formato de código postal erróneo).
- *Análisis de valor de frontera.* Los datos de los formularios se prueban en sus fronteras. Por ejemplo, la función de cálculo de embarque anotada anteriormente solicita el número máximo de días requeridos para la entrega del producto. En el formulario se anota un mínimo de 2 días y un máximo de 14. Sin embargo, las pruebas de valor de frontera pueden ingresar valores de 0, 1, 2, 13, 14 y 15 para determinar cómo reacciona la función de datos en y afuera de las fronteras de entrada válida.[9]
- *Prueba de rutas.* Si la complejidad lógica de la función es alta,[10] puede usarse la prueba de rutas (un método de diseño de casos de prueba de caja blanca) para garantizar que se ejercitó cada ruta independiente en el programa.

Además de estos métodos de diseño de casos de prueba, se usa una técnica llamada *prueba de error forzado* [Ngu01] para derivar casos de prueba que a propósito conducen al componente

9 En este caso, un mejor diseño de entrada puede eliminar errores potenciales. El número máximo de días podría seleccionarse de un menú desplegable, lo que impide al usuario especificar entrada fuera de fronteras.

10 La complejidad lógica puede determinarse al calcular la complejidad ciclomática del algoritmo. Vea el capítulo 18 para detalles adicionales.

web a una condición de error. El propósito es descubrir los errores que ocurren durante la manipulación del error (por ejemplo, mensajes de error incorrectos o inexistentes, falla de la *webapp* como consecuencia del error, salida errónea activada por entrada errónea, efectos colaterales que se relacionan con el procesamiento de componentes).

Cada caso de prueba en el nivel componente especifica todos los valores de entrada y salida que se espera que proporcione el componente. La salida real producida como consecuencia de la prueba se registra para futuras referencias durante el soporte y el mantenimiento.

En muchas situaciones, la ejecución correcta de la función de una *webapp* se liga a la interfaz adecuada con una base de datos que puede ser externa a la *webapp*. Por tanto, la prueba de base de datos se convierte en parte integral del régimen de prueba de componente.

## 20.6 PRUEBA DE NAVEGACIÓN

Un usuario viaja a través de una *webapp* en forma muy parecida a como un visitante camina a través de una tienda o de un museo. Existen muchas rutas que pueden tomarse, muchas paradas que pueden realizarse, muchas cosas que aprender y mirar, actividades por iniciar y decisiones por tomar. Este proceso de navegación es predecible porque cada 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 usual conforme el objetivo original. La labor de la prueba de navegación es 1) garantizar que son funcionales todos los mecanismos que permiten al usuario de la *webapp* recorrerla y 2) validar que cada unidad semántica de navegación (USN) pueda lograr la categoría de usuario apropiada.

### 20.6.1 Prueba de sintaxis de navegación

**Cita:**

"No estamos perdidos. Tenemos un desafío posicional."

**John M. Ford**

La primera fase de la prueba de navegación en realidad comienza durante la prueba de interfaz. Los mecanismos de navegación se prueban para asegurarse de que cada interfaz realiza la función que se le ha encargado. Splaine y Jaskiel [Spl01] sugieren que debe probarse cada uno de los siguientes mecanismos de navegación:

- *Vínculos de navegación:* estos mecanismos incluyen vínculos internos dentro de la *webapp*, vínculos externos hacia otras *webapps* y anclas dentro de una página web específica. Cada vínculo debe ser probado para asegurarse de que se alcanza el contenido o funcionalidad adecuados cuando se elige el vínculo.
- *Redirecciones:* estos vínculos entran en juego cuando un usuario solicita una URL inexistente o cuando selecciona un vínculo cuyo contenido se removió o cuyo nombre cambió. Se despliega un mensaje para el usuario y la navegación se redirige hacia otra página (por ejemplo, la página de inicio). Los redireccionamientos deben probarse al solicitar vínculos internos incorrectos o URL externas y debe valorarse cómo maneja la *webapp* estas solicitudes.
- *Marcas de página (favoritos, bookmarks):* aunque las marcas de página son función del navegador, la *webapp* debe probarse para garantizar la extracción de un título de página significativo conforme se crea la marca.
- *Marcos y framesets:* cada marco incluye el contenido de una página web específica; un frameset contiene múltiples marcos y habilita el despliegue de múltiples páginas web al mismo tiempo. Puesto que es posible anidar marcos y framesets unos dentro de otros, estos mecanismos de navegación y despliegue deben probarse para que tengan el contenido correcto, la plantilla y tamaño adecuados, rendimiento de descargas y compatibilidad de navegador.

- *Mapas de sitio:* un mapa de sitio proporciona una tabla de contenido completa para todas las páginas web. Cada entrada del mapa de sitio debe probarse para garantizar que los vínculos llevan al usuario al contenido o funcionalidad adecuados.
- *Motores de búsqueda internos:* las *webapps* complejas con frecuencia contienen cientos o incluso miles de objetos de contenido. Un motor de búsqueda interno permite al usuario realizar una búsqueda de palabra clave dentro de la *webapp* para encontrar el contenido necesario. La prueba del motor de búsqueda valida la precisión y completitud de la búsqueda, las propiedades de manejo de error del motor de búsqueda y las características de búsqueda avanzadas (por ejemplo, el uso de operadores booleanos en el campo de búsqueda).

Algunas de las pruebas anotadas pueden realizarse mediante herramientas automatizadas (por ejemplo, comprobación de vínculos), mientras que otras se diseñan y ejecutan manualmente. La intención de principio a fin es garantizar que los errores en la mecánica de navegación se encuentran antes de que la *webapp* entre en línea.

### 20.6.2 Prueba de la semántica de navegación

**¿Qué preguntas deben plantearse y responderse conforme se prueba cada USN?**

En el capítulo 13, una unidad semántica de navegación (USN) se define como "un conjunto de estructuras de información y navegación relacionada que colaboran en el cumplimiento de un subconjunto de requerimientos de usuario relacionados" [Cac02]. Cada USN se define mediante un conjunto de trayectorias de navegación (llamadas "rutas de navegación") que conectan los nodos de navegación (por ejemplo, páginas web, objetos de contenido o funcionalidad). Considerado como un todo, cada USN permite a un usuario lograr requerimientos específicos definidos por uno o más casos de uso para una categoría de usuario. La prueba de navegación ejercita cada USN para asegurarse de que dichos requerimientos pueden lograrse. Es necesario responder las siguientes preguntas conforme se prueba cada USN:

- ¿La USN se logra en su totalidad sin error?
- ¿Todo nodo de navegación (definido por una USN) se alcanza dentro del contexto de las rutas de navegación definidas por la USN?
- Si la USN puede lograrse usando más de una ruta de navegación, ¿se probó cada ruta relevante?
- Si la interfaz de usuario proporciona una guía para auxiliar en la navegación, ¿las instrucciones son correctas y comprensibles conforme avanza la navegación?
- ¿Existe un mecanismo (distinto a la flecha "retroceso" del navegador) para regresar al nodo de navegación anterior y al comienzo de la ruta de navegación?
- ¿Los mecanismos de navegación dentro de un gran nodo de navegación (es decir, una página web grande) funcionan de manera adecuada?
- Si una función debe ejecutarse en un nodo y el usuario elige no proporcionar entrada, ¿el resto de la USN puede completarse?
- Si una función se ejecuta en un nodo y ocurre un error en el procesamiento de la función, ¿la USN puede completarse?
- ¿Existe alguna forma para descontinuar la navegación antes de que todos los nodos se hayan alcanzado, pero luego regresar a donde se descontinuó la navegación y avanzar desde ahí?
- ¿Todo nodo es alcanzable desde el mapa de sitio? ¿Los nombres de nodo son significativos para los usuarios finales?

**CONSEJO**

*Si no se han creado USN como parte del análisis o el diseño de la webapp, puede aplicar casos de uso para el diseño de casos de prueba de navegación. El mismo conjunto de preguntas se plantean y responden.*

- Si un nodo dentro de una USN se alcanza desde alguna fuente externa, ¿es posible avanzar hacia el nodo siguiente en la ruta de navegación? ¿Es posible regresar al nodo anterior 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 interfaz y usabilidad, debe realizarse por tantos departamentos como sea posible. Usted tiene la responsabilidad durante las primeras etapas de la prueba de navegación, pero las etapas posteriores las deben realizar otros participantes en el proyecto, un equipo de prueba independiente y, a final de cuentas, usuarios no técnicos. La intención es ejercitar la navegación de la *webapp* a profundidad.

## 20.7 PRUEBA DE CONFIGURACIÓN

La variabilidad y la inestabilidad de la configuración son factores importantes que hacen de la prueba de *webapps* un desafío. El hardware, los sistemas operativos, navegadores, capacidad de almacenamiento, velocidades de comunicación de red y varios otros factores en el lado cliente son difíciles de predecir para cada usuario. Además, la configuración para un usuario dado puede cambiar de manera regular [por ejemplo, actualizaciones del sistema operativo (OS), nuevos ISP y velocidades de conexión]. El resultado puede ser un entorno lado cliente que es proclive a errores sutiles y significativos. La impresión que un usuario tiene de la *webapp* y la forma en la que interactúa con ella pueden diferir significativamente de la experiencia de otro usuario si ambos usuarios no trabajan dentro de la misma configuración en el lado cliente.

La labor de la prueba de configuración no es ejercitar toda configuración posible en el lado cliente. En vez de ello, es probar un conjunto de probables configuraciones en los lados cliente y servidor para garantizar que la experiencia del usuario será la misma en todos ellos y que aislará los errores que puedan ser específicos de una configuración particular.

### 20.7.1 Conflictos en el lado servidor

En el lado servidor, los casos de prueba de configuración se diseñan para verificar que la configuración servidor proyectada [es decir, servidor *webapp*, servidor de base de datos, sistemas operativos, software de firewall (cortafuegos), aplicaciones concurrentes] pueden soportar la *webapp* sin error. En esencia, la *webapp* se instaló dentro del entorno del lado servidor y se probó para asegurar que opera sin error.

**? ¿Qué preguntas deben plantearse y responderse conforme se realiza la prueba de configuración en el lado servidor?**

Conforme se diseñan las pruebas de configuración del lado servidor, debe considerarse cada componente de la configuración del servidor. Entre las preguntas que deben plantearse y responderse durante la prueba de configuración del lado servidor se encuentran:

- ¿La *webapp* es completamente compatible con el servidor OS?
- ¿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, *firewalls* o encriptado) permiten a la *webapp* ejecutarse y atender a los usuarios sin interferencia o degradación del rendimiento?
- ¿La *webapp* se probó con la configuración de servidor distribuido[11] (si existe alguno) que se eligió?

11 Por ejemplo, puede usarse un servidor de aplicación separado de un servidor de base de datos. La comunicación entre las dos máquinas ocurre a través de una conexión de red.

- ¿La *webapp* se integró adecuadamente con el 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 servidor se ejecutan adecuadamente?
- ¿Los errores del administrador del sistema se examinaron en sus efectos sobre las operaciones de la *webapp*?
- Si se usan servidores proxy, ¿las diferencias en su configuración se abordaron con pruebas en sitio?

### 20.7.2 Conflictos en el lado cliente

En el lado cliente, las pruebas de configuración se enfocan con más peso en la compatibilidad de la *webapp* con las configuraciones que contienen una o más permutas de los siguientes componentes [Ngu01]:

- *Hardware:* CPU, memoria, almacenamiento y dispositivos de impresión
- *Sistemas operativos:* Linux, Macintosh OS, Microsoft Windows, un OS móvil
- *Software navegador:* Firefox, Safari, Internet Explorer, Opera, Chrome y otros
- *Componentes de interfaz de usuario:* Active X, Java applets y otros
- *Plug-ins:* QuickTime, RealPlayer y muchos otros
- *Conectividad:* cable, DSL, módem regular, T1, WiFi

Además de estos componentes, otras variables incluyen software de redes, caprichos de la ISP y aplicaciones que corren de manera concurrente.

Para diseñar pruebas de configuración en el lado cliente, debe reducir el número de variables de configuración a un número manejable.[12] Para lograr esto, cada categoría de usuario se valora para determinar las probables configuraciones que pueden encontrarse dentro de la categoría. Además, pueden usarse datos de participación de mercado para predecir las combinaciones de componentes más probables. Entonces la *webapp* se prueba dentro de estos entornos.

## 20.8 PRUEBA DE SEGURIDAD

**Cita:**

"Internet es un lugar riesgoso para realizar negocios o almacenar valores. Hackers, crackers, snoops, spoofers... vándalos, lanzadores de virus y proveedores de programas maliciosos corren a sus anchas."

**Dorothy y Peter Denning**

La seguridad de la *webapp* es un tema complejo que debe comprenderse por completo antes de que pueda lograrse una prueba de seguridad efectiva.[13] Las *webapps* y los entornos en los lados cliente y servidor donde se albergan representan un blanco atractivo para *hackers* externos, empleados descontentos, competidores deshonestos y para quien quiera robar información sensible, modificar contenido maliciosamente, degradar el rendimiento, deshabilitar la funcionalidad o avergonzar a una persona, organización o negocio.

Las pruebas de seguridad se diseñan para sondear las vulnerabilidades del entorno lado cliente, las comunicaciones de red que ocurren conforme los datos pasan de cliente a servidor y viceversa, y el entorno del lado servidor. Cada uno de estos dominios puede atacarse, y es tarea del examinador de seguridad descubrir las debilidades que puedan explotar quienes tengan intención de hacerlo.

En el lado cliente, las vulnerabilidades con frecuencia pueden rastrearse en errores preexistentes en navegadores, programas de correo electrónico o software de comunicación. Nguyen [Ngu01] describe un hueco de seguridad común:

12 Aplicar pruebas en toda combinación posible de componentes de configuración consume demasiado tiempo.

13 Los libros de Cross y Fischer [Cro07], Andrews y Whittaker [And06] y Trivedi [Tri03] proporcionan información útil acerca del tema.


*Si la webapp es crucial para el negocio, mantiene datos sensibles o es un blanco probable de los hackers, es buena idea subcontratar pruebas de seguridad con un proveedor que se especialice en ellas.*

Uno de los errores comúnmente mencionados es el desbordamiento de buffer, que permite que código malicioso se ejecute en la máquina cliente. Por ejemplo, ingresar una URL en un navegador que es mucho más larga que el tamaño de buffer asignado 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 experimentado puede explotar astutamente este error al escribir una URL larga con código que se va a ejecutar y que puede hacer que el navegador derribe o altere las configuraciones de seguridad (de alto a bajo) o, peor aún, corromper datos del usuario.

Otra vulnerabilidad potencial en el lado cliente es el acceso no autorizado a las *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 riesgo la privacidad del usuario o, peor aún, que montan el escenario para el robo de identidad.

Los datos comunicados entre el cliente y el servidor son vulnerables al *spoofing* (engaño). El *spoofing* ocurre cuando un extremo de la ruta de comunicación se trastorna por una entidad con intenciones maliciosas. Por ejemplo, un usuario puede ser engañado por un sitio malicioso que actúa como si fuese el servidor de la *webapp* legítimo (apariencia y sensación idénticas). La intención es robar contraseñas, información personal o datos de crédito.

En el lado servidor, las vulnerabilidades incluyen ataques de negación de servicio y guiones maliciosos que pueden pasar hacia el lado cliente o usarse para deshabilitar operaciones del servidor. Además, puede accederse sin autorización a las bases de datos en el lado servidor (robo de datos).


Las pruebas de seguridad deben diseñarse para ejercitar *firewalls*, autenticación, encriptado y autorización.

Para proteger contra éstas (y muchas otras) vulnerabilidades, se implanta uno o más de los siguientes elementos de seguridad [Ngu01]:

- *Firewall:* mecanismo de filtrado, que es una combinación de hardware y software que examina cada paquete de información entrante para asegurarse de que proviene de una fuente legítima y que bloquea cualquier dato sospechoso.
- *Autenticación:* mecanismo de verificación que valida la identidad de todos los clientes y servidores, y permite que la comunicación ocurra solamente cuando ambos lados se verifican.
- *Encriptado:* mecanismo de codificación que protege los datos sensibles al modificarlos de forma que hace imposible leerlos por quienes tienen intenciones maliciosas. El encriptado se fortalece usando *certificados digitales* que permiten al cliente verificar el destino al que se transmiten los datos.
- *Autorización:* mecanismo de filtrado que permite el acceso al entorno cliente o servidor sólo a aquellos individuos con códigos de autorización apropiados (por ejemplo, ID de usuario y contraseña).

Las pruebas de seguridad deben diseñarse para sondear cada una de estas tecnologías de seguridad con la intención de descubrir huecos en la seguridad.

El diseño real de las pruebas de seguridad requiere conocimiento profundo del trabajo interno de cada elemento de seguridad y amplia comprensión de una gran gama de tecnologías de redes. En muchos casos, la prueba de seguridad se subcontrata con firmas que se especializan en dichas tecnologías.

## 20.9 PRUEBA DE RENDIMIENTO

Nada es más frustrante que una *webapp* que tarda minutos en cargar contenido cuando sitios de la competencia descargan contenido similar en segundos. Nada es más exasperante que

intentar ingresar a una *webapp* y recibir un mensaje de "servidor ocupado", con la sugerencia de que se intente de nuevo más tarde. Nada es más desconcertante que una *webapp* que responde instantáneamente en algunas situaciones y luego en otras parece caer en un estado de espera infinita. Estos eventos suceden en la web todos los días y todos ellos se relacionan con el rendimiento.

Las *pruebas de rendimiento* se usan para descubrir problemas de rendimiento que pueden ser resultado de: falta de recursos en el lado servidor, red con ancho de banda inadecuada, capacidades de base de datos inadecuadas, capacidades de sistema operativo deficientes o débiles, funcionalidad de *webapp* pobremente diseñada y otros conflictos de hardware o software que pueden conducir a rendimiento cliente-servidor degradado. La intención es doble: 1) comprender cómo responde el sistema conforme aumenta la *carga* (es decir, número de usuarios, número de transacciones o volumen de datos global) y 2) recopilar mediciones que conducirán a modificaciones de diseño para mejorar el rendimiento.

### 20.9.1 Objetivos de la prueba de rendimiento


*Algunos aspectos del rendimiento de la webapp, al menos como los percibe el usuario final, son difíciles de poner a prueba. La carga de la red, los caprichos del hardware de interfaz con la red y conflictos similares no son fáciles de poner a prueba en la webapp.*

Las pruebas de rendimiento se diseñan para simular situaciones de carga del mundo real. Conforme aumenta el número de usuarios simultáneos de la *webapp* o el número de transacciones en línea o la cantidad de datos (descargados o subidos), las pruebas de rendimiento ayudarán a responder las siguientes preguntas:

- ¿El tiempo de respuesta del servidor se degrada a un punto donde es apreciable e inaceptable?
- ¿En qué punto (en términos de usuarios, transacciones o carga de datos) el rendimiento se vuelve inaceptable?
- ¿Qué componentes del sistema son responsables de la degradación del rendimiento?
- ¿Cuál es el tiempo de respuesta promedio para los usuarios bajo diversas condiciones de carga?
- ¿La degradación del rendimiento tiene impacto sobre la seguridad del sistema?
- ¿La confiabilidad o precisión de la *webapp* resulta afectada conforme crece la carga sobre el sistema?
- ¿Qué sucede cuando se aplican cargas que son mayores que la capacidad máxima del servidor?
- ¿La degradación del rendimiento tiene impacto sobre los ingresos de la compañía?

Para desarrollar respuestas a estas preguntas, se realizan dos tipos diferentes de pruebas de rendimiento: 1) la *prueba de carga* examina la carga del mundo real en varios niveles de carga y en varias combinaciones, y 2) la *prueba de esfuerzo* fuerza a aumentar la carga hasta el punto de rompimiento para determinar cuánta capacidad puede manejar el entorno de la *webapp*. Cada una de estas estrategias de prueba se considera en las secciones siguientes.

### 20.9.2 Prueba de carga

La intención de la prueba de carga es determinar cómo responderán las *webapps* y su entorno del lado servidor a varias condiciones de carga. Conforme avanzan las pruebas, las permutas de las siguientes variables definen un conjunto de condiciones de prueba:

*N,* número de usuarios concurrentes

*T,* número de transacciones en línea por unidad de tiempo

*D,* carga de datos procesados por el servidor en cada transacción


*Si una webapp usa múltiples servidores para proporcionar una capacidad significativa, la prueba de carga debe realizarse en un entorno multiservidor.*

En todo caso, dichas variables se definen dentro de fronteras operativas normales del sistema. Conforme se aplica cada condición de prueba, se recopila una o más de las siguientes medidas: respuesta de usuario promedio, tiempo promedio para descargar una unidad estandarizada de datos y tiempo promedio para procesar una transacción. Estas medidas deben examinarse para determinar si una disminución abrupta en el rendimiento puede rastrearse en una combinación específica de *N*, *T* y *D*.

La prueba de carga también puede usarse para valorar las velocidades de conexión recomendadas para los usuarios de la *webapp*. El rendimiento global, *P*, se calcula de la forma siguiente:

$$P = N \times T \times D$$

Tome como ejemplo un sitio popular de noticias deportivas. En un momento dado, 20 000 usuarios concurrentes envían una solicitud (una transacción, *T*) una vez cada 2 minutos en promedio. Cada transacción requiere que la *webapp* descargue un nuevo artículo que promedia 3 Kb de longitud. Por tanto, el rendimiento global puede calcularse como:

$$P = [20\,000 \times 0.5 \times 3\text{Kb}]/60 = 500 \text{ Kbytes/s}$$
$$= 4 \text{ megabits por segundo}$$

Por ende, la conexión de la red para el servidor tendría que soportar esta tasa de datos y debería ponerse a prueba para asegurarse de que lo hace.

### 20.9.3 Prueba de esfuerzo


El objetivo de la prueba de esfuerzo es comprender de mejor manera como falla un sistema a medida que es forzado más allá de sus límites operacionales.

La *prueba de esfuerzo* es una continuación de la prueba de carga, pero en esta instancia las variables *N*, *T* y *D* se fuerzan a satisfacerse y luego se superan los límites operativos. La intención de estas pruebas es responder a cada una de las siguientes preguntas:

- ¿El sistema se degrada "suavemente" o el servidor se apaga conforme la capacidad se supera?
- ¿El software servidor genera mensajes "servidor no disponible"? De manera más general, ¿los usuarios están conscientes de que no pueden llegar al servidor?
- ¿El servidor pone en cola los recursos solicitados y vacía la cola una vez que disminuye la demanda de capacidad?
- ¿Las transacciones se pierden conforme la capacidad se excede?
- ¿La integridad de los datos resulta afectada conforme la capacidad se excede?
- ¿Qué valores de *N*, *T* y *D* fuerzan el fallo del entorno servidor? ¿Cómo se manifiesta la falla? ¿Se envían notificaciones automáticas al personal de apoyo técnico en el sitio servidor?
- Si el sistema falla, ¿cuánto tiempo tardará en regresar en línea?
- ¿Ciertas funciones de la *webapp* (por ejemplo, funcionalidad de cálculo intenso, capacidades de transmisión de datos) quedan descontinuadas conforme la capacidad alcanza el nivel de 80 o 90 por ciento?

A una variación de las pruebas de esfuerzo en ocasiones se le conoce como *prueba pico/rebote* (*spike/bounce*) [Spl01]. En este régimen de pruebas, la carga alcanza un pico de capacidad, luego se baja rápidamente a condiciones operativas normales y después alcanza de nuevo un pico. Al rebotar la carga del sistema, es posible determinar cuán bien el servidor puede ordenar los recursos para satisfacer una demanda muy alta y entonces liberarlos cuando reaparecen condiciones normales (de modo que esté listo para el siguiente pico).

HERRAMIENTAS DE SOFTWARE

### *Taxonomía de herramientas para prueba de webapp*

En su ensayo acerca de las pruebas de los sistemas de comercio electrónico, Lam [Lam01] presenta una útil taxonomía de herramientas automáticas que tienen aplicabilidad directa para probar en un contexto de ingeniería web. Se anexan las herramientas representativas en cada categoría.[14]

Las **herramientas de configuración y gestión de contenido** gestionan la versión y cambian el control de los objetos de contenido web y los componentes funcionales.
*Herramientas representativas:* Una lista amplia se encuentra en **www.daveeaton.com/scm/CMTools.html**

Las **herramientas de rendimiento de base de datos** miden el rendimiento de la base de datos, como el tiempo para realizar consultas seleccionadas en bases de datos. Dichas herramientas facilitan la optimización de la base de datos.
*Herramientas representativas:* BMC Software (**www.bmc.com**)

Los **depuradores** son herramientas de programación usuales que encuentran y resuelven defectos de software en el código. Son parte de la mayoría de los entornos modernos de desarrollo de aplicaciones.
*Herramientas representativas:*
Accelerated Technology (**www.acceleratedtechnology.com**)
Apple Debugging Tools (**developer.apple.com/tools/performance/**)
IBM VisualAge Environment (**www.ibm.com**)
Microsoft Debugging Tools (**www.microsoft.com**)

Los **sistemas de gestión de defecto** registran los defectos y rastrean su estado y resolución. Algunos incluyen herramientas de reporte para ofrecer información de gestión acerca de la dispersión del defecto y tasas de resolución del mismo.
*Herramientas representativas:*
EXCEL Quickbigs (**www.excelsoftware.com**)
ForeSoft BugTrack (**www.bugtrack.net**)
McCabe TRUETrack (**www.mccabe.com**)

Las **herramientas de monitoreo de red** vigilan el nivel de tráfico en la red. Son útiles para identificar cuellos de botella en la red y probar el vínculo entre sistemas frontales y traseros.
*Herramientas representativas:*
Una lista exhaustiva se encuentra en **www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html**

Las **herramientas de prueba de regresión** almacenan casos de prueba y datos de prueba, y pueden volver a aplicar los casos de prueba después de cambios sucesivos de software.
*Herramientas representativas:*
Compuware QARun (**www.compuware.com/products/qacenter/qarun**)
Rational VisualTest (**www.rational.com**)
Seque Software (**www.seque.com**)

Las **herramientas de monitoreo de sitio** monitorean el rendimiento de un sitio, con frecuencia desde la perspectiva del usuario. Se usan para recopilar estadísticas tales como tiempo de respuesta extremo a extremo y rendimiento global, y para comprobar periódicamente la disponibilidad de un sitio.
*Herramientas representativas:*
Keynote Systems (**www.keynote.com**)

Las **herramientas de esfuerzo** ayudan a los desarrolladores a explorar el comportamiento del sistema bajo niveles altos de uso operativo y encuentran los puntos de ruptura de un sistema.
*Herramientas representativas:*
Mercury Interactive (**www.merc-int.com**)
Herramientas de prueba de fuente abierta (**www.opensourcetesting.org/performance .php**)
Web Performance Load Tester (**www.webperformanceinc.com**)

Los **monitores de recursos del sistema** son parte de la mayoría de los servidores OS y software servidor web; monitorean recursos tales como espacio de disco, uso de CPU y memoria.
*Herramientas representativas:*
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 la generación de datos de prueba.
*Herramientas representativas:*
Una lista exhaustiva se encuentra en **www.softwareqatest.com/qatweb1.html**

Los **comparadores de resultado de prueba** ayudan a comparar los resultados de un conjunto de pruebas con los de otro conjunto. Se usan para comprobar que los cambios de código no han introducido cambios adversos en el comportamiento del sistema.
*Herramientas representativas:*
Una lista útil se encuentra en **www.aptest.com/resources.html**

Los **monitores de transacción** miden el rendimiento de los sistemas de procesamiento de alto volumen de transacciones.
*Herramientas representativas:*
QuotiumPro (**www.quotium.com**)
Software Research eValid (**www.soft.com/eValid/index.html**)

Las **herramientas de seguridad website** ayudan a detectar potenciales problemas de seguridad. Con frecuencia puede establecerse un sondeo de seguridad y herramientas de monitoreo para correr sobre una base calendarizada.
*Herramientas representativas:*
Una lista amplia se encuentra en **www.timberlinetechnologies.com/products/www.html**

14 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que existen en esta categoría. Además, los nombres de las herramientas son marcas registradas de las compañías señaladas.

## 20.10 RESUMEN

La meta de la prueba de las *webapps* es ejercitar cada una de las muchas dimensiones de calidad de la *webapp* con la intención de encontrar errores o descubrir conflictos que puedan conducir a fallas en la calidad. Las pruebas se enfocan en contenido, función, estructura, usabilidad, navegabilidad, rendimiento, compatibilidad, interacción, capacidad y seguridad. Estas pruebas incorporan revisiones que ocurren conforme se diseña la *webapp* y pruebas que se llevan a cabo una vez que se implanta la misma.

La estrategia de prueba de *webapps* ejercita cada dimensión de calidad al examinar inicialmente "unidades" de contenido, funcionalidad o navegación. Una vez validadas las unidades individuales, la atención se cambia hacia pruebas que ejercitan la *webapp* como un todo. Para lograr esto, muchas pruebas se derivan desde la perspectiva del usuario y se activan mediante la información contenida en casos de uso. Se desarrolla un plan de prueba de *webapps* y se identifican los pasos de la prueba, productos de trabajo (por ejemplo, casos de prueba) y mecanismos para la evaluación de los resultados de la prueba. El proceso de prueba abarca siete tipos diferentes de pruebas.

La prueba de contenido (y las revisiones) se enfocan en varias categorías de contenido. La intención es descubrir errores semánticos y sintácticos que afectan la precisión del contenido o la forma en la que se presenta al usuario final. La prueba de interfaz ejercita los mecanismos de interacción que permiten al usuario comunicarse con la *webapp* y valida los aspectos estéticos de la interfaz. La intención es descubrir errores que resultan a partir de mecanismos de interacción pobremente implantados o de omisiones, inconsistencias o ambigüedades en la semántica de la interfaz.

Las pruebas de navegación aplican casos de uso, derivados de la actividad de modelado, en el diseño de casos de prueba que ejercitan cada escenario de uso contra el diseño de navegación. Los mecanismos de navegación se ponen a prueba para garantizar que cualquier error que impida completar un caso de uso se identifique y corrija. La prueba de componente ejercita las unidades de contenido y funcionales dentro de la *webapp*.

La prueba de configuración intenta descubrir errores y/o problemas de compatibilidad que son específicos de un entorno cliente o servidor particular. Entonces se realizan pruebas para descubrir errores asociados con cada posible configuración. Las pruebas de seguridad incorporan una serie de pruebas diseñadas para explotar las vulnerabilidades en la *webapp* y su entorno. La intención es encontrar huecos de seguridad. La prueba de rendimiento abarca una serie de pruebas que se diseñan para valorar el tiempo de respuesta y la confiabilidad de la *webapp* conforme aumenta la demanda en la capacidad de recursos en el lado servidor.

## PROBLEMAS Y PUNTOS POR EVALUAR

**20.1.** ¿Existen algunas situaciones en las que la prueba de *webapps* deba descartarse por completo?

**20.2.** Con sus palabras, analice los objetivos de las pruebas en un contexto *webapp*.

**20.3.** La compatibilidad es una importante dimensión de la calidad. ¿Qué debe probarse para garantizar que existe compatibilidad para una *webapp*?

**20.4.** ¿Cuáles errores tienden a ser más serios: los que hay en el lado cliente o los del lado servidor? ¿Por qué?

**20.5.** ¿Qué elementos de la *webapp* pueden recibir "prueba de unidad"? ¿Qué tipos de pruebas deben realizarse sólo después de que se integran los elementos de la *webapp*?

**20.6.** ¿Siempre es necesario desarrollar un plan de prueba escrito formalmente? Explique.

**20.7.** ¿Es justo decir que la estrategia de prueba de *webapps* global comienza con los elementos visibles para el usuario y que avanza hacia los elementos tecnológicos? ¿Existen excepciones a esta estrategia?

**20.8.** ¿La prueba de contenido *realmente* es una prueba en el sentido convencional? Explique.

**20.9.** Describa los pasos asociados con la prueba de base de datos para una *webapp*. ¿La prueba de base de datos es predominantemente una actividad en el lado cliente o en el lado servidor?

**20.10.** ¿Cuál es la diferencia entre las pruebas que se asocian con los mecanismos de interfaz y las que abordan la semántica de la interfaz?

**20.11.** Suponga que desarrolla una farmacia en línea (**FarmaciaDelaEsquina.com**) que abastece a adultos mayores. 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 advertir de potenciales interacciones medicamentosas. Analice algunas pruebas de usabilidad especiales para esta *webapp*.

**20.12.** Suponga que establece una función de comprobación de interacción medicamentosa para **FarmaciaDelaEsquina.com** (problema 20.11). Analice los tipos de las pruebas en el nivel de componente que deberían realizarse para garantizar que esta función opera adecuadamente. [Nota: tendría que usar una base de datos para establecer 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 toda configuración que una *webapp* probablemente encuentre en el lado servidor? ¿Es posible hacerlo en el lado cliente? Si no, ¿cómo selecciona un conjunto significativo de pruebas de configuración?

**20.15.** ¿Cuál es el objetivo de la prueba de seguridad? ¿Quién realiza esta prueba?

**20.16.** **FarmaciaDelaEsquina.com** (problema 20.11) se volvió muy exitosa y el número de usuarios aumentó dramáticamente en los primeros dos meses de operación. Dibuje una gráfica que muestre el probable tiempo de respuesta como función del número de usuarios para un conjunto fijo de recursos en el lado servidor. Etiquete la gráfica para indicar los puntos de interés en la "curva de respuesta".

**20.17.** En respuesta a su éxito, **FarmaciaDelaEsquina.com** (problema 20.11) implementó un servidor especial exclusivamente para manejar el reabastecimiento de recetas. En promedio, 1 000 usuarios concurrentes envían una solicitud de reabastecimiento una vez cada dos minutos. En respuesta, la *webapp* descarga un bloque de datos de 500 bytes. ¿Cuál es al rendimiento global aproximado requerido para este servidor en megabits por segundo?

**20.18.** ¿Cuál es la diferencia entre prueba de carga y prueba de esfuerzo?

## Lecturas adicionales y fuentes de información

La literatura acerca de pruebas de *webapps* continúa evolucionando. Los libros de Andrews y Whittaker (*How to Break Web Software,* Addison-Wesley, 2006), Ash (*The Web Testing Companion,* Wiley, 2003), Nguyen *et al*. (*Testing Applications for the Web,* 2a. ed., Wiley, 2003), Dustin *et al*. (*Quality Web Systems,* Addison-Wesley, 2002) y Splaine y Jaskiel [Spl01] están entre los tratamientos más completos del tema publicados hasta la fecha. Mosley (*Client-Server Software Testing on the Desktop and the Web*, Prentice Hall, 1999) aborda los temas de prueba en los lados cliente y servidor.

Información útil acerca de las estrategias y métodos de prueba de *webapps*, así como un valioso análisis de las herramientas de prueba automáticas, se presenta en Stottlemeyer (*Automated Web Testing Toolkit,* Wiley, 2001). Graham *et al*. (*Software Test Automation,* Addison-Wesley, 1999) presentan material adicional acerca de herramientas automáticas.

Microsoft (*Performance Testing Guidance for Web Applications,* Microsoft Press, 2008) y Subraya *(Integrated Approach to Web Performance Testing,* IRM Press, 2006) exponen tratamientos detallados acerca de la prueba de rendimiento para *webapps*. Chirillo (*Hack Attacks Revealed,* 2a. ed., Wiley, 2003), Splaine (*Testing Web Security,* Wiley, 2002), Klevinsky *et al*. (*Hack I.T.: Security through Penetration Testing,* Addison-Wesley, 2002) y Skoudis (*Counter Hack,* Prentice Hall, 2001) proporcionan mucha información útil para quienes deben diseñar pruebas de seguridad. Además, los libros que abordan la prueba de seguridad para software en general pueden ofrecer una guía importante para quienes deben poner a prueba *webapps*. Los títulos representativos incluyen: Basta y Halton (*Computer Security and Penetration Testing,* Thomson Delmar Learning, 2007), Wy-

sopal *et al*. (*The Art of Software Security Testing,* Addison-Wesley, 2006) y Gallagher *et al*. (*Hunting Security Bugs,* Microsoft Press, 2006).

En internet está disponible gran variedad de fuentes de información acerca de pruebas de *webapps*. En el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm** puede encontrarse una lista actualizada de referencias que hay en la World Wide Web y que son relevantes para las pruebas de *webapps*.

CAPÍTULO

# 21 MODELADO Y VERIFICACIÓN FORMAL

CONCEPTOS CLAVE



A diferencia de las revisiones y pruebas que comienzan una vez desarrollados los modelos y códigos de software, el modelado y la verificación formales incorporan métodos de modelado especializados que se integran con enfoques de verificación prescritos. Sin el enfoque de modelado adecuado, la verificación no puede lograrse.

En este capítulo se analizan dos métodos de modelado y verificación formales: el *método de ingeniería del software de cuarto limpio* y los métodos *formales*. Ambos requieren un enfoque de especificación especializado y cada uno aplica un método de verificación único. Los dos son bastante rigurosos y la comunidad de ingeniería del software no usa ninguno de ellos ampliamente. Pero si intenta construir software a prueba de balas, dichos métodos pueden ayudarle de manera inconmensurable. Vale la pena aprenderlos.

La *ingeniería del software de cuarto limpio* es un enfoque que enfatiza la necesidad de construir con exactitud el software conforme éste se desarrolla. En lugar de análisis, diseño, código, prueba y ciclo de depuración clásicos, el enfoque de cuarto limpio sugiere un punto de vista diferente [Lin94b]:

> La filosofía que hay detrás de la ingeniería del software de cuarto limpio consiste en evitar la dependencia de costosos procesos de remoción de defectos, al escribir incrementos de código justo la primera vez y verificar su exactitud antes de examinarlo. Su modelo de proceso incorpora la certificación de calidad estadística de los incrementos de código conforme se acumulan en un sistema.

## UNA MIRADA RÁPIDA

**¿Qué es?** ¿Cuántas veces ha escuchado a alguien decir "Hazlo bien desde la primera vez"? Si esto se lograra en el software, habría considerablemente menos esfuerzo empleado en lo innecesario. Dos métodos avanzados de ingeniería del software (ingeniería del software de cuarto limpio y métodos formales) ayudan al equipo de software a "hacerlo bien desde la primera vez" al proporcionar un enfoque basado en matemáticas para programar el modelado y la capacidad de verificar que el modelo es correcto. La ingeniería del software de cuarto limpio enfatiza la verificación matemática de la exactitud antes de que comience la construcción del programa y la certificación de la confiabilidad del software como parte de la actividad de prueba. Los métodos formales usan teoría de conjuntos y notación lógica para crear un enunciado claro de los hechos (requerimientos) que pueden analizarse para mejorar (o incluso probar) la exactitud y la consistencia. La línea de base para ambos métodos es la creación de software con tasas de falla extremadamente bajas.

**¿Quién lo hace?** Un ingeniero del software especialmente capacitado.

**¿Por qué es importante?** Los errores obligan a que haya revisiones. Las revisiones toman tiempo y aumentan los costos. ¿No sería bueno si pudiera reducir dramáticamente el número de errores (bugs) introducidos mientras el software se diseña y construye? Ésa es la premisa del modelado y de la verificación formales.

**¿Cuáles son los pasos?** Los modelos de requerimientos y de diseño se crean usando notación especializada que es susceptible de verificación matemática. La ingeniería de software de cuarto limpio usa representación de estructura de cajas que encapsulan el sistema (o algún aspecto del mismo) en un nivel específico de abstracción. La verificación de la exactitud se aplica cuando está completo el diseño de la estructura de caja. Una vez verificada la exactitud para cada estructura de caja, comienza la prueba de uso estadístico. Los métodos formales traducen los requerimientos de software en una representación más formal al aplicar la notación y la heurística de conjuntos a fin de definir el invariante de datos, estados y operaciones para una función de sistema.

**¿Cuál es el producto final?** Se desarrolla un modelo formal especializado de requerimientos. Se registran los resultados de las pruebas de exactitud y de uso estadístico.

**¿Cómo me aseguro de que lo hice bien?** La prueba de exactitud formal se aplica al modelo de requerimientos. La prueba de uso estadístico ejercita los escenarios de uso para garantizar que se descubren y corrigen los errores en la funcionalidad del usuario.

En muchas formas, el enfoque de cuarto limpio eleva la ingeniería del software a otro nivel, al enfatizar la necesidad de *probar* la exactitud.

Los modelos desarrollados que usan *métodos formales* se describen mediante una sintaxis y semántica formales que especifican el funcionamiento y el comportamiento del sistema. La especificación consiste en matemática en forma (por ejemplo, puede usarse cálculo de predicados como la base para un lenguaje de especificación formal). En su introducción a los métodos formales, Anthony Hall [Hal90] hace un comentario que se aplica igualmente a los métodos de cuarto limpio:

> Los métodos formales [la ingeniería del software de cuarto limpio] son controversiales. Sus defensores afirman que pueden revolucionar el desarrollo [del software]. Sus detractores creen que son imposiblemente difíciles. Mientras tanto, para la mayoría de la gente, los métodos formales [y la ingeniería del software de cuarto limpio] son tan poco corrientes que es difícil juzgar las afirmaciones rivales.

En este capítulo se exploran los métodos de modelado y verificación formales y se examina su impacto potencial sobre la ingeniería del software en los años por venir.

## 21.1 Estrategia de cuarto limpio

**Cita:**

"La única forma de que ocurran errores en un programa es que el autor los ponga ahí. No se conocen otros mecanismos [...] La práctica correcta se dirige a evitar la inserción de errores y, al fallar esto, a removerlos antes de probarlo o de cualquier otra forma de poner en marcha el programa."

**Harlan Mills**

La ingeniería del software de cuarto limpio usa una versión especializada del modelo de software incremental que se introdujo en el capítulo 2. Pequeños equipos de software independientes desarrollan "una tubería de incrementos de software" [Lin94b]. Conforme cada incremento se certifica, se integra en el todo. Por tanto, la funcionalidad del sistema crece con el tiempo.

La secuencia de las tareas de cuarto limpio para cada incremento se ilustra en la figura 21.1. Dentro de la tubería de incrementos de cuarto limpio, ocurren las siguientes tareas:

**Planeación del incremento.** Se desarrolla un plan de proyecto que adopte la estrategia incremental. Se crea la funcionalidad de cada incremento, su tamaño proyectado y un calendario de desarrollo de cuarto limpio. Debe tenerse especial cuidado para garantizar que los incrementos certificados se integrarán en forma oportuna.

**Recopilación de requerimientos.** Se desarrolla una descripción más detallada de los requerimientos del cliente (para cada incremento), con el uso de técnicas similares a las introducidas en el capítulo 5.

**Especificación de estructura de caja.** Se usa un método de especificación que utilice las *estructuras de caja* para describir la especificación funcional. Las estructuras de caja "se aíslan, y separan la definición creativa de comportamiento, datos y procedimientos en cada nivel de refinamiento" [Hev93].

**Diseño formal.** Al usar el enfoque de estructura de cajas, el diseño de cuarto limpio es una extensión natural y sin costuras de la especificación. Aunque es posible hacer una distinción clara entre las dos actividades, las especificaciones (llamadas *cajas negras*) se refinan iterativamente (dentro de un incremento) para convertirse en análogas de los diseños arquitectónicos y en el nivel de componente (llamados *cajas de estado* y *cajas claras*, respectivamente).

**Verificación de exactitud.** El equipo de cuarto limpio realiza una serie de rigurosas actividades de verificación de exactitud sobre el diseño y, luego, el código. La verificación (sección 21.3.2) comienza con la estructura de caja (especificación) de nivel más alto y avanza hacia el detalle y el código de diseño. El primer nivel de la verificación de exactitud ocurre al aplicar un conjunto de "preguntas de exactitud" [Lin88]. Si esto no demuestra que la especificación es correcta, se usan métodos más formales (matemáticos) para la verificación.

**WebRef**

En **www.cleansoft.com** puede encontrarse una excelente fuente de información y recursos para ingeniería del software de cuarto limpio.

**Cita:**

"La ingeniería del software de cuarto limpio logra control de calidad estadístico sobre el desarrollo del software al separar estrictamente el proceso de diseño del proceso de prueba en una tubería de desarrollo incremental de software."

**Harlan Mills**

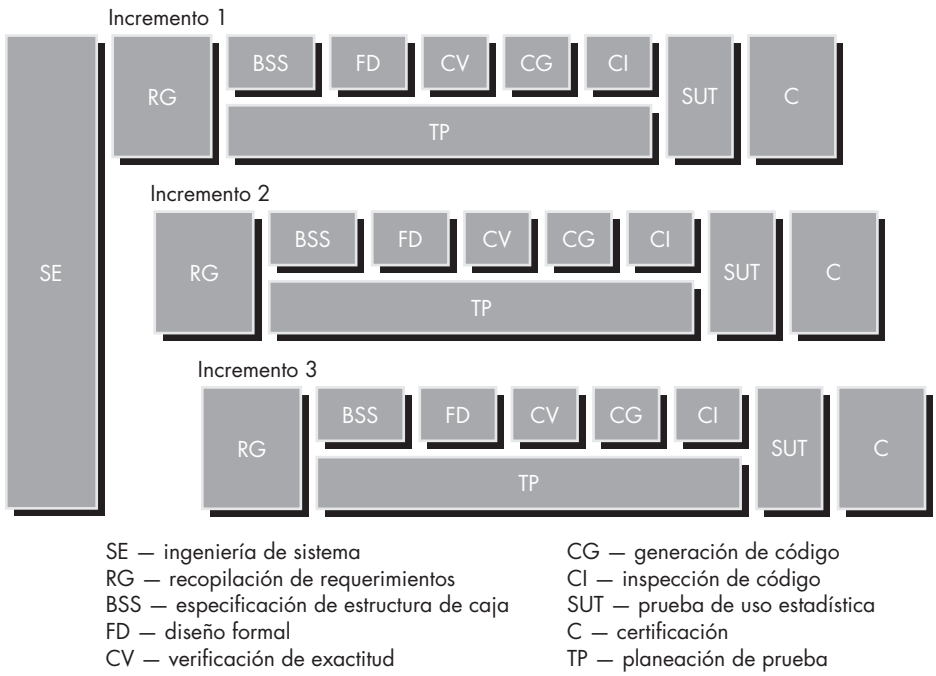



**FIGURA 21.1** Modelo de proceso de cuarto limpio

**Generación, inspección y verificación de código.** Las especificaciones de estructura de caja, representadas en un lenguaje especializado, se traducen al lenguaje de programación adecuado. Las revisiones técnicas (capítulo 15) se usan entonces para asegurar la conformidad semántica del código y las estructuras de código, así como la exactitud sintáctica del código. Luego se realiza la verificación de exactitud para el código fuente.

**Planeación de prueba estadística.** Se analiza el uso proyectado del software y se planea y diseña (sección 21.4) una suite de casos de prueba que ejercitan una "distribución de probabilidad" de uso. En la figura 21.1, esta actividad de cuarto limpio se realiza en paralelo con la especificación, la verificación y la generación de código.

**Prueba de uso estadístico.** Si se recuerda que la prueba exhaustiva del software de computadora es imposible (capítulo 18), siempre es necesario diseñar un número finito de casos de prueba. Las técnicas de uso estadístico [Poo88] ejecutan una serie de pruebas derivadas de una muestra estadística (la distribución de probabilidad anotada anteriormente) de todas las posibles ejecuciones de programa efectuadas por todos los usuarios de una población objetivo (sección 21.4).


*El cuarto limpio enfatiza las pruebas que ejercitan la manera en la que el software se usa realmente. Los casos de uso proporcionan entrada al proceso de planeación de prueba.*

**Certificación.** Una vez completadas la verificación, inspección y prueba de uso (y todos los errores corregidos), el incremento se certifica como listo para su integración.

Las primeras cuatro actividades en el proceso de cuarto limpio establecen el escenario para las actividades normales de verificación que siguen. Por esta razón, el estudio del enfoque de cuarto limpio comienza con las actividades de modelado, que son esenciales para la verificación formal que se va a aplicar.

## 21.2 ESPECIFICACIÓN FUNCIONAL

El enfoque de modelado en la ingeniería del software de cuarto limpio usa un método llamado *especificación de estructura de caja*. Una "caja" encapsula el sistema (o algún aspecto del mismo)

**Cita:**

"Hay algo divertido en la vida: si rechazas aceptar todo menos lo mejor, con mucha frecuencia lo conseguirás."

**W. Somerset Maugham**

en algún nivel de detalle. A través de un proceso de elaboración o refinamiento por pasos, las cajas se refinan en una jerarquía donde cada caja tiene transparencia referencial. Es decir: "la información contenida en cada especificación de caja es suficiente para definir su refinamiento, sin depender de la implementación de alguna otra caja" [Lin94b]. Esto permite al analista dividir un sistema jerárquicamente y avanzar de la representación esencial en la parte superior al detalle específico de la implementación en el fondo. Para ello, se usan tres tipos de cajas:

**¿Cómo se logra el refinamiento como parte de una especificación de estructura de caja?**

**Caja negra.** La caja negra especifica el comportamiento de un sistema o de una parte de un sistema. El sistema (o su parte) responde a estímulos específicos (eventos) al aplicar un conjunto de reglas de transición que mapean el estímulo en una respuesta.

**Caja de estado.** La caja de estado encapsula los datos y servicios (operaciones) de estado en una forma análoga a los objetos. En esta visión de especificación, se representan las entradas (estímulos) y salidas (respuestas) a la caja de estado. La caja de estado también representa la "historia de estímulos" de la caja negra, es decir, los datos encapsulados en la caja de estado que deben conservarse entre las transiciones implicadas.

**Caja clara.** Las funciones de transición que se implican mediante la caja de estado se definen en la caja clara. Dicho de manera simple, una caja clara contiene el diseño de procedimientos para la caja de estado.

La figura 21.2 ilustra el enfoque de refinamiento, usando la especificación de estructura de cajas. Una caja negra ($BB_1$) define las respuestas a un conjunto completo de estímulos. $BB_1$ puede refinarse en un conjunto de cajas negras, $BB_{1.1}$ a $BB_{1.n}$, cada una de las cuales enfoca una clase de comportamiento. El refinamiento continúa hasta que se identifica una clase cohesiva de comportamiento (por ejemplo, $BB_{1.1.1}$). Entonces se define una caja de estado ($SB_{1.1.1}$) para la caja negra ($BB_{1.1.1}$). En este caso, $SB_{1.1.1}$ contiene todos los datos y servicios requeridos para implementar el comportamiento definido por $BB_{1.1.1}$. Finalmente, $SB_{1.1.1}$ se refina en cajas claras ($CB_{1.1.1.n}$) y se especifican los detalles del diseño de procedimientos.

**PUNTO CLAVE**

El refinamiento de estructura de cajas y la verificación de exactitud ocurren simultáneamente.

Conforme ocurre cada uno de estos pasos de refinamiento, también se presenta la verificación de la exactitud. Las especificaciones de caja de estado se verifican para asegurar que cada una se conforma de acuerdo con el comportamiento definido por la especificación de la caja negra padre. De igual modo, las especificaciones de caja clara se verifican contra la caja de estado padre.

**FIGURA 21.2**

**Refinamiento de estructura de cajas**

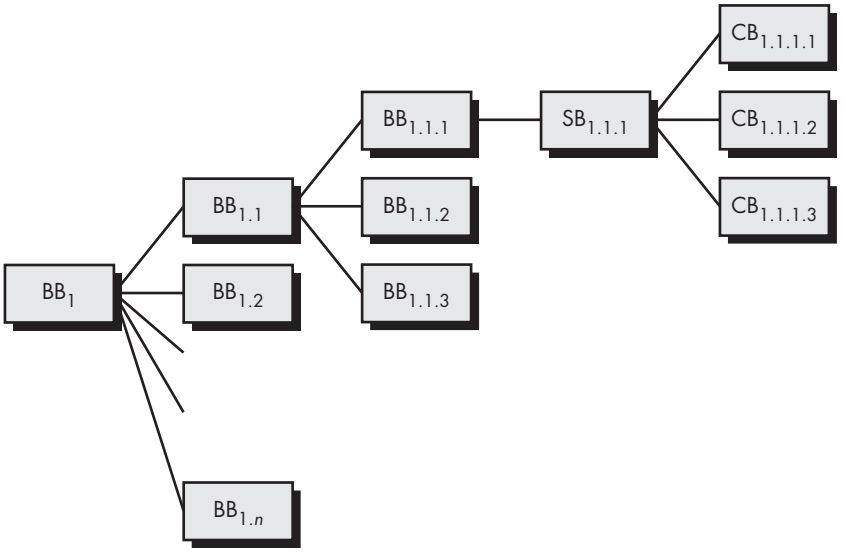
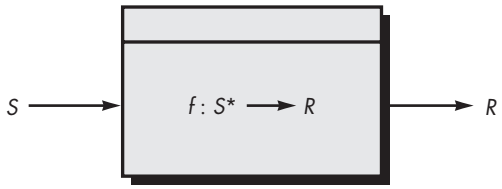


**FIGURA 21.3**
**Especificación de caja negra**




### 21.2.1 Especificación de caja negra

Una especificación de *caja negra* describe una abstracción, estímulos y respuesta, usando la notación que se muestra en la figura 21.3 [Mil88]. La función $f$ se aplica a una secuencia $S^*$ de entradas (estímulos) $S$ y las transforma en una salida (respuesta) $R$. Para componentes de software simples, $f$ puede ser una función matemática, pero, en general, $f$ se describe usando lenguaje natural (o un lenguaje de especificación formal).

Muchos de los conceptos introducidos para los sistemas orientados a objetos también son aplicables para la caja negra. Las abstracciones de datos y las operaciones que manipulan dichas abstracciones se encapsulan mediante la caja negra. Como una jerarquía de clases, la especificación de caja negra puede mostrar jerarquías de uso en las que las cajas de nivel inferior heredan las propiedades de las cajas superiores que hay en la estructura del árbol.

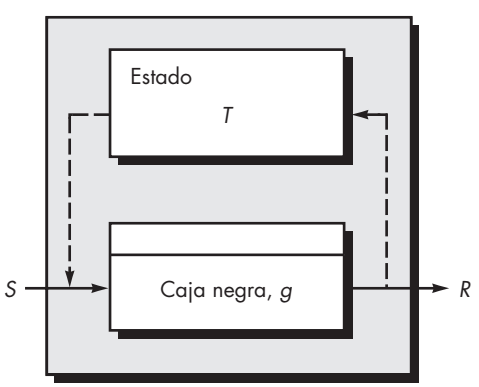
### 21.2.2 Especificación de caja de estado

La *caja de estado* es "una simple generalización de una máquina de estado" [Mil88]. Si se recuerda el análisis acerca del modelado de comportamiento y de los diagramas de estado estudiados en el capítulo 7, un estado es un modo observable de comportamiento del sistema. Conforme ocurre el procesamiento, un sistema responde a los eventos (estímulos), haciendo una transición desde el estado actual hasta algún estado nuevo. Conforme se realiza la transición, puede ocurrir una acción. La caja de estado usa una abstracción de datos para determinar la transición al siguiente estado y la acción (respuesta) que ocurrirá como consecuencia de la transición.

En la figura 21.4, la caja de estado incorpora una caja negra $g$. El estímulo $S$ que es entrada a la caja negra llega desde alguna fuente externa y desde 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:

$$g : S^* \times T^* \rightarrow R \times T$$

**FIGURA 21.4**
**Una especificación de caja de estado**



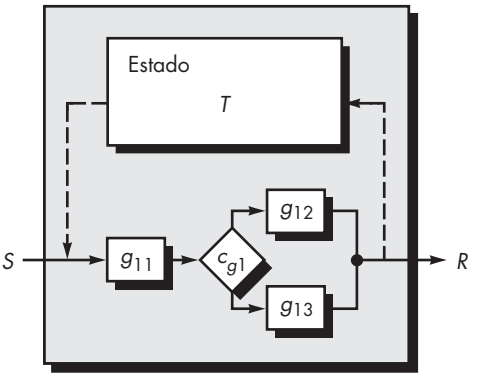



**FIGURA 21.5**
**Una especificación de caja clara**

donde *g* es una subfunción que se liga a un estado específico *t*. Cuando se consideran de manera colectiva, los pares de subfunción de estado (*t, g*) definen la función *f* de la caja negra.

### 21.2.3 Especificación de caja clara

La especificación de caja clara está cercanamente alineada con el diseño procedural y con la programación estructurada. En esencia, la subfunción *g* dentro de la caja de estado se sustituye con los constructos de programación estructurada que implementan *g*.

Como ejemplo, considere la caja clara que se muestra en la figura 21.5. La caja negra *g*, que se muestra en la figura 21.3, se sustituye por un constructo secuencia que incorpora un condicional. Éste, a su vez, puede refinarse en las cajas claras de nivel inferior conforme avanza el refinamiento por pasos.

Es importante observar que puede probarse que la especificación descrita en la jerarquía de caja clara es correcta. Este tema se considera en la sección 21.3.

## 21.3 DISEÑO DE CUARTO LIMPIO

La ingeniería del software de cuarto limpio utiliza mucho la filosofía de programación estructurada (capítulo 10). Pero, en este caso, la programación estructurada se aplica de manera mucho más rigurosa.

Las funciones de procesamiento básico (descritas durante los primeros refinamientos de la especificación) se refinan usando una "expansión en pasos 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 enunciarse directamente en el lenguaje de programación utilizado para la implementación" [Dye92].

El enfoque de programación estructurada puede usarse de manera efectiva para refinar funciones. Pero ¿qué hay del diseño? Aquí entran en juego algunos conceptos de diseño fundamentales (capítulo 8). Los datos de programa se encapsulan como un conjunto de abstracciones que son atendidas por subfunciones. Los conceptos de encapsulamiento de datos, ocultamiento de información y escritura de datos se usan para crear el diseño de datos.

### 21.3.1 Refinamiento de diseño

Cada especificación de caja clara representa el diseño de un procedimiento (subfunción) requerido para lograr una transición de caja de estado. Dentro de la caja clara se usan constructos de programación estructurada y refinamiento por pasos para representar detalles procedurales. Por ejemplo, una función de programa *f* se refina en una secuencia de subfunciones *g* y *h*. Éstas a

su vez se refinan en constructos condicionales (por ejemplo, if-then-else y do-while). Un mayor refinamiento continúa hasta que hay suficiente detalle procedural para crear el componente en cuestión.

En cada nivel de refinamiento, el equipo de cuarto limpio[1] realiza una *verificación formal de exactitud*. Para lograr esto, a los constructos de programación estructurada se une un conjunto de condiciones de exactitud genéricas. Si una función *f* se expande en una secuencia *g* y *h*, la condición de exactitud para toda entrada a *f* es

**¿Qué condiciones se aplican para probar la exactitud de los constructos estructurados?**

- ¿*g* seguida de *h* hace *f*?

Cuando una función *p* se refina en una condicional de la forma "if <*c*> then *q*, else *r*", la condición de exactitud para toda entrada a *p* es

- 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 ciclo, las condiciones de exactitud para toda entrada a *m* son

- ¿Está garantizada la finalización?
- Siempre que <*c*> es verdadera, ¿*n* seguida por *m* hace *m*?; y siempre que <*c*> es falsa, ¿saltar el ciclo todavía hace *m*?

Cada vez que una caja clara se refina al siguiente nivel de detalle, se aplican dichas condiciones de exactitud.

### 21.3.2 Verificación de diseño

**CONSEJO**

*Si se limita sólo a constructos estructurados mientras desarrolla un diseño procedural, la prueba de exactitud es directa. Si viola los constructos, las pruebas de exactitud son difíciles o imposibles.*

Debe observar que el uso de los constructos de programación estructurada restringen el número de pruebas de exactitud que deben realizarse. Una sola condición se verifica para secuencias; dos condiciones se prueban para if-then-elseo y tres condiciones se verifican para ciclo.

Para ilustrar la verificación de exactitud para un diseño procedural, use un ejemplo simple, introducido por primera vez por Linger, Mills y Witt [Lin79]. La intención es diseñar y verificar un pequeño programa que encuentre la parte entera *y* de una raíz cuadrada de un entero dado *x*. El diseño procedural se representa usando el diagrama de flujo de la figura 21.6.[2]

Para verificar la exactitud de este diseño, las condiciones de entrada y salida se agregan como se muestra en la figura 21.6. La condición de entrada observa que *x* debe ser mayor que o igual a 0. La condición de salida requiere que *x* permanezca invariable y que *y* satisfaga la expresión anotada en la figura. Para probar que el diseño es correcto, es necesario probar que las condiciones *init*, *loop*, *cont*, *yes* y *exit*, que se muestran en la figura 21.6, son verdaderas en todos los casos. En ocasiones a esto se le conoce como *subpruebas*.

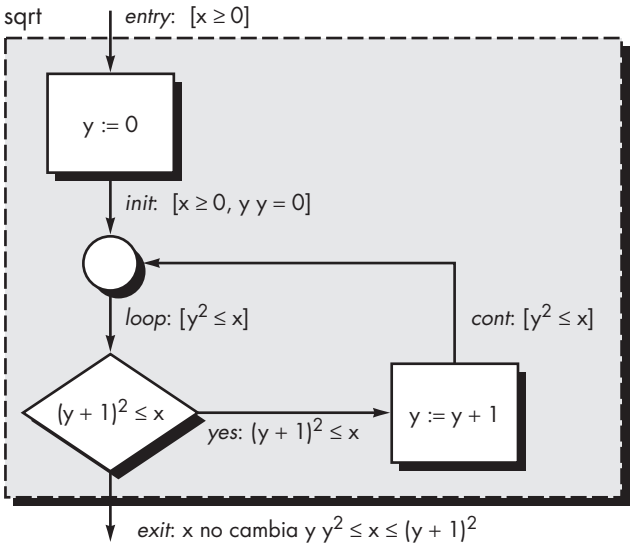
1. La condición *init* demanda que [$x \geq 0$ y $y = 0$]. Con base en los requerimientos del problema, la condición de entrada se supone correcta.[3] Por tanto, se satisface la primera parte de la condición *init*, $x \geq 0$. En el diagrama de flujo, el enunciado inmediatamente anterior a la condición *init* establece $y = 0$. Por tanto, la segunda parte de la condición *init* también se satisface. En consecuencia, *init* es verdadera.
2. La condición *loop* puede encontrarse en una de dos formas: 1) directamente de *init* (en este caso, la condición *loop* se satisface directamente) o por medio del flujo de control

1 Puesto que todo el equipo está involucrado en el proceso de verificación, es menos probable que se cometa un error al realizar la verificación en sí.

2 La figura 21.6 se adaptó de [Lin94]. Utilizada con permiso.

3 En este contexto no tiene validez un valor negativo para la raíz cuadrada.

**FIGURA 21.6**
**Cálculo de la parte entera de una raíz cuadrada**
**Fuente: [Lin79].**




**PUNTO CLAVE**
Para probar que un diseño es correcto, primero deben identificarse todas las condiciones y luego probar que cada una toma el valor booleano adecuado. A éstas se les llama *subpruebas*.

que pasa a través de la condición *cont*. Dado que la condición *cont* es idéntica a la condición *loop*, ésta es verdadera sin importar la trayectoria de flujo que conduce a ella.

3. La condición *cont* se encuentra solamente después de que el valor de $y$ aumenta en 1. Además, la ruta de flujo de control que conduce a *cont* puede invocarse sólo si la condición *yes* también es verdadera. Por tanto, si $(y + 1)^2 \leq x$ se sigue que $y^2 \leq x$. Se satisface la condición *cont*.
4. La condición *yes* se prueba en la lógica condicional que se muestra. Por ende, la condición *yes* debe ser verdadera cuando el flujo de control se mueve a lo largo de la trayectoria mostrada.
5. La condición *exit* demanda primero que $x$ permanezca invariable. Un examen del diseño indica que $x$ no aparece a la izquierda de un operador de asignación. No hay llamadas de función que usen $x$. En consecuencia, es invariable. Puesto que la prueba condicional $(y + 1)^2 \leq x$ debe fallar para alcanzar la condición *exit*, se sigue que $(y + 1)^2 \leq x$. Además, la condición *loop* debe incluso ser verdadera (es decir, $y^2 \leq x$). Por tanto, $(y + 1)^2 > x$ y $y^2 \leq x$ pueden combinarse para satisfacer la condición *exit*.

Adicionalmente, debe asegurarse de que el ciclo termina. Un examen de la condición *loop* indica que, dado que $y$ se incrementa y $x \geq 0$, el ciclo finalmente debe terminar.

Los cinco pasos recién señalados son una prueba de la exactitud del diseño del algoritmo anotado en la figura 21.6. Entonces se está seguro de que el diseño, de hecho, calculará la parte entera de una raíz cuadrada.

Es posible un enfoque matemático más riguroso para verificar el diseño. Sin embargo, un estudio de este tema está más allá del ámbito de este libro. Si tiene interés en ello, consulte [Lin79].

## 21.4 PRUEBAS DE CUARTO LIMPIO

La estrategia y tácticas de las pruebas de cuarto limpio son fundamentalmente diferentes a las de los enfoques de prueba convencionales (capítulos del 17 al 20). Los métodos convencionales derivan un conjunto de casos de prueba para descubrir errores de diseño y codificación. La meta

**Cita:**

"La calidad no es un acto, es un hábito."

**Aristóteles**

de la prueba de cuarto limpio es validar los requerimientos de software al demostrar que una muestra estadística de casos de uso (capítulo 5) se ejecuta exitosamente.

### 21.4.1 Pruebas de uso estadístico

El usuario de un programa de computadora rara vez necesita entender los detalles técnicos del diseño. El comportamiento del programa visible al usuario se activa con entradas y eventos que con frecuencia son producidos por el usuario. Pero en los sistemas complejos, el posible espectro de entrada y eventos (es decir, los casos de uso) puede ser extremadamente amplio. ¿Qué subconjunto de casos de uso verificará de manera adecuada el comportamiento del programa? Ésta es la primera pregunta que enfoca las pruebas de uso estadístico.

**CONSEJO**

*Incluso si se decide no usar el enfoque de cuarto limpio, vale la pena considerar las pruebas de uso estadístico como parte integral de la estrategia de pruebas.*

La prueba de uso estadístico "equivale a examinar el software de la forma en la que los usuarios pretenden usarlo" [Lin94b]. Para lograr esto, los equipos de prueba de cuarto limpio (también llamados *equipos de certificación*) deben determinar una distribución de probabilidad de uso para el software. La especificación (caja negra) para cada incremento del software se analiza a fin de definir un conjunto de estímulos (entradas o eventos) que hacen que el software cambie su comportamiento. En la creación de escenarios de uso y en una comprensión general del dominio de aplicación, a cada estímulo se le asigna una probabilidad de uso con base en entrevistas con usuarios potenciales.

Para cada conjunto de estímulos[4] se generan casos de prueba de acuerdo con la distribución de probabilidad de uso. Para ilustrar lo anterior, considere el sistema *CasaSegura* que se estudió anteriormente en este libro. La ingeniería del software de cuarto limpio se usó para desarrollar un incremento de software que gestiona la interacción del usuario con el teclado del sistema de seguridad. Para este incremento se identificaron cinco estímulos. Los análisis indican la distribución de probabilidad porcentual de cada estímulo. Para seleccionar con facilidad los casos de prueba, dichas probabilidades se mapean en intervalos numerados entre 1 y 99 [Lin94] y se ilustran en la siguiente tabla:

| Estímulo del programa | Probabilidad | Intervalo |
|---|---|---|
| Armar/desarmar (AD) | 50% | 1–49 |
| Establecer zona (EZ) | 15% | 50–63 |
| Consulta (C) | 15% | 64–78 |
| Prueba (P) | 15% | 79–94 |
| Alarma de pánico | 5% | 95–99 |

Para generar una secuencia de casos de prueba de uso que se ajusten a la distribución de probabilidad de uso, se generan números aleatorios entre 1 y 99. Cada número aleatorio corresponde a un intervalo en la distribución de probabilidad precedente. Por tanto, la secuencia de casos de prueba de uso se define al azar, pero corresponde a la probabilidad adecuada de ocurrencia del estímulo. Por ejemplo, suponga 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 adecuados con base en el intervalo de distribución que se muestra en la tabla, se derivan los siguientes casos de uso:

4 Puede usar herramientas automatizadas para lograr esto. Para mayor información, consulte [Dye92].

AD–T–AD–AD–AD–ZS
T–AD–AD–AD–Q–AD–AD
AD–AD–ZS–T–T–AD

El equipo de prueba los ejecuta, y verifica el comportamiento del software contrastándolo con la especificación para el sistema. La temporización de las pruebas se registra de modo que puedan determinarse los intervalos de tiempo. Al usar intervalos de tiempo, el equipo de certificación puede calcular el tiempo medio hasta el fallo (TMHF). Si una larga secuencia de pruebas se realiza sin fallas, el TMHF es bajo y la confiabilidad del software puede suponerse alta.

### 21.4.2 Certificación

Las técnicas de verificación y prueba analizadas anteriormente en este capítulo conducen a componentes de software (e incrementos completos) que pueden certificarse. Dentro del contexto del enfoque de ingeniería del software de cuarto limpio, la *certificación* implica que la confiabilidad (medida por el TMHF) puede especificarse para cada componente.

El impacto potencial de los componentes de software certificables va más allá de un solo proyecto de cuarto limpio. Los componentes de software reutilizables pueden almacenarse junto con sus escenarios de uso, estímulos de programa y distribuciones de probabilidad. Cada componente tendría una confiabilidad certificada bajo el escenario de uso y el régimen de pruebas descritos. Esta información es invaluable para otros que quieran usar los componentes.

**? ¿Cómo se certifica un componente de software?**

El enfoque de certificación involucra cinco pasos [Woh94]: 1) se crean escenarios de uso, 2) se especifica un perfil de uso, 3) se generan casos de prueba a partir del perfil, 4) las pruebas se ejecutan y los datos de fallo se registran y analizan, y 5) se calcula la confiabilidad y se certifica. Los pasos del 1 al 4 se analizaron en una sección anterior. La certificación para la ingeniería del software de cuarto limpio requiere la creación de tres modelos [Poo93]:

**Modelo de muestreo.** La prueba de software ejecuta *m* casos de prueba aleatorios y se certifica si no ocurren fallos o un número específico de ellos. El valor de *m* se deriva matemáticamente para asegurar que se logra la confiabilidad requerida.

**Modelo de componente.** Se certifica un sistema compuesto de *n* componentes. El modelo de componentes permite al analista determinar la probabilidad de que el componente *i* fallará antes de su conclusión.

**Modelo de certificación.** La confiabilidad global del sistema se proyecta y se certifica.

Al completar las pruebas de uso estadístico, el equipo de certificación tiene la información requerida para entregar el software que tenga un TMHF certificado, usando cada uno de estos modelos. Si tiene más interés, vea [Cur86], [Mus87] o [Poo93], para detalles adicionales.

## 21.5 Conceptos de métodos formales

*The Encyclopedia of Software Engineering* [Mar01] define los métodos formales en la forma siguiente:

> Los métodos formales utilizados para desarrollar sistemas de cómputo son técnicas con base matemática para describir las propiedades del sistema. Tales métodos formales proporcionan marcos conceptuales dentro de los cuales las personas pueden especificar, desarrollar y verificar los sistemas en forma sistemática más que *ad hoc*.

Las propiedades deseadas de una especificación formal (consistencia, completitud y falta de ambigüedad) son los objetivos de todos los métodos de especificación. Sin embargo, el lenguaje de especificación con base matemática que se utiliza para los métodos formales da como resultado una probabilidad mucho mayor de lograr dichas propiedades. La sintaxis formal de un

**Cita:**

"Los métodos formales tienen un tremendo potencial para mejorar la claridad y precisión de las especificaciones de los requerimientos, y para encontrar errores importantes y sutiles."

**Steve Easterbrook *et al.***

lenguaje de especificación (sección 21.7) permite que los requerimientos o el diseño se interpreten sólo en una forma, lo que elimina la ambigüedad que con frecuencia ocurre cuando un lector debe interpretar un lenguaje natural (por ejemplo, inglés) o una notación gráfica (por ejemplo, UML). Las facilidades descriptivas de la teoría de conjuntos y la notación lógica permiten un enunciado claro de los requerimientos. Para ser consistente, los requerimientos enunciados en un lugar dentro de una especificación no deben contradecirse en otro lugar. La consistencia se logra[5] al probar matemáticamente que los hechos iniciales pueden mapearse formalmente (usando reglas de inferencia) en los enunciados ulteriores dentro de la especificación.

Para presentar los conceptos de los métodos formales básicos, considere algunos ejemplos simples a fin de ilustrar el uso de la especificación matemática, sin empantanarse en demasiados detalles matemáticos.

**Ejemplo 1: una tabla simbólica.** Un programa se usa para mantener una tabla simbólica. Dicha tabla se utiliza frecuentemente en muchos tipos diferentes de aplicaciones. Consiste de una colección de ítems sin duplicación alguna. En la figura 21.7 se muestra un ejemplo de una tabla simbólica típica. En ella se representa la tabla que utiliza un sistema operativo para contener los nombres de los usuarios del sistema. Otros ejemplos de tablas incluyen la colección de nombres del personal en un sistema de nómina, la colección de nombres de las computadoras en un sistema de comunicaciones de red y la colección de los destinos en un sistema para producir horarios de transportes.

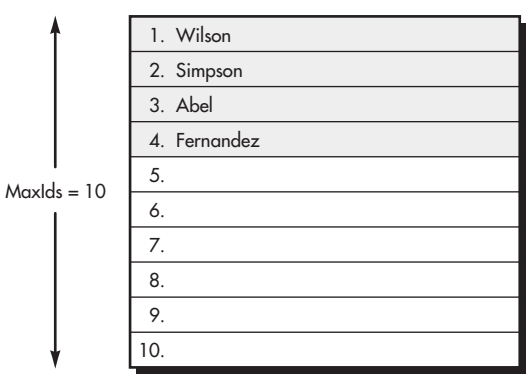
**PUNTO CLAVE**

Una invariante de datos es un conjunto de condiciones que son verdaderas a lo largo de la ejecución del sistema que contiene una colección de datos.

Suponga que la tabla que se presenta en este ejemplo contiene no más de *MaxIds* nombres. Esta afirmación, que coloca una restricción sobre la tabla, es un componente de una condición conocida como *invariante de datos*: una condición que es verdadera a lo largo de la ejecución de un sistema que contiene una colección de datos. La invariante de datos que se sostiene para la tabla simbólica recién analizada 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 simbólica, esto significa que en cualquier momento en el que se examine la tabla simbólica durante la ejecución del sistema, siempre contendrá no más de *MaxIds* nombres y no contendrá duplicados.

Otro concepto importante es el de *estado*. Muchos lenguajes formales, como el OCL (sección 21.7.1), usan la noción de estado que se estudió en el capítulo 7, es decir, un sistema puede

**FIGURA 21.7**
**Una tabla simbólica**




5 En realidad, la completitud es difícil de garantizar, aun cuando se usen métodos formales. Algunos aspectos de un sistema pueden quedar indefinidos conforme se cree la especificación; otras características pueden omitirse a propósito a fin de permitir a los diseñadores cierta libertad para escoger un enfoque de implementación; y, finalmente, es imposible considerar todo escenario operativo en un sistema grande y complejo. Las cosas pueden omitirse simplemente por equivocación.

CONSEJO

*Otra forma de apreciar la noción de estado es señalar que los datos determinan el estado. Es decir, puede examinar los datos para ver en qué estado se encuentra el sistema.*

estar en uno de muchos estados y cada uno representa un modo de comportamiento observable de manera externa. Sin embargo, una definición diferente para el término *estado* se usa en el lenguaje Z (sección 21.7.2). En Z (y en lenguajes relacionados), el estado de un sistema se representa por los datos almacenados del sistema (por ende, Z sugiere un número mucho mayor de estados, lo que representa cada posible configuración de los datos). Al usar la última definición en el ejemplo del programa de tabla simbólica, el estado es la tabla simbólica.

El concepto final es el de *operación*. Ésta es una acción que tiene lugar dentro de un sistema y que lee o escribe datos. Si el programa de tabla simbólica tiene que ver con agregar y remover nombres de la tabla simbólica, entonces se asociará con dos operaciones: una operación para *add()* (agregar) un nombre específico a la tabla simbólica y otra para *remove()* (remover) un nombre existente de la tabla.[6] Si el programa proporciona la facilidad para comprobar si un nombre específico está contenido en la tabla, entonces habría una operación que regresaría alguna indicación acerca de si el nombre está en la tabla.

Pueden asociarse tres tipos de condiciones con las operaciones: invariantes, precondiciones y poscondiciones. Una *invariante* define lo que se garantiza que no cambia. Por ejemplo, la tabla simbólica tiene una invariante que afirma que el número de elementos siempre es menor que 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 agrega un nombre a una tabla simbólica de identificadores de personal es válida sólo si el nombre que se agrega no está contenido en la tabla y también si hay menos de *MaxIds* identificadores de personal en ella. La *poscondición* de una operación define lo que se garantiza que es verdadero hasta completar una operación. Esto se define por su efecto sobre los datos. Para la operación *add( )*, la poscondición especificaría matemáticamente que la tabla aumentó con el nuevo identificador.

**Ejemplo 2: un manipulador de bloques.** Una de las partes más importantes de un sistema operativo simple es el subsistema que mantiene los archivos creados por los usuarios. Parte del subsistema de llenado es el *manipulador de bloques*. El almacén de archivos está compuesto de bloques de almacenamiento que se mantienen en un dispositivo de almacenamiento de archivos. Durante la operación de la computadora, se crearán y borrarán archivos, lo que requiere adquisición y liberación de bloques de almacenamiento. Para poder lidiar con esto, el subsistema de llenado mantendrá un reservorio de bloques no utilizados (libres) y seguirá la pista de los bloques que estén en uso actual. Cuando los bloques se liberan de un archivo borrado, por lo general se agregan a una fila de bloques que esperan para incorporarse al reservorio de bloques no utilizados. Esto se muestra en la figura 21.8, donde se presentan algunos componentes: el reservorio de bloques no utilizados, los bloques que en la actualidad constituyen los archivos administrados por el sistema operativo y los bloques que esperan agregarse al reservorio. Los bloques que esperan se mantienen en una fila en la que cada elemento contiene un conjunto de bloques de un archivo borrado.

CONSEJO

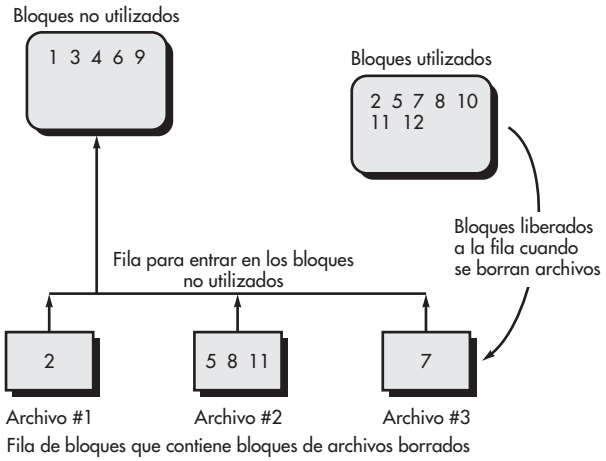
*Las técnicas de lluvia de ideas pueden funcionar bien cuando debe desarrollar una invariante de datos para una función razonablemente compleja. Haga que los miembros del equipo de software escriban enlaces, restricciones y limitaciones para la función; luego, combínelas y edítelas.*

Para este subsistema, el estado es la colección de bloques libres, la colección de bloques usados y la fila de bloques regresados. La invariante de datos, que se expresa en lenguaje natural, es

- Ningún bloque se marcará como no utilizado y usado al mismo tiempo.
- Todos los conjuntos de bloques que se conservan en la fila serán subconjuntos de la colección de los bloques actualmente utilizados.
- Ningún elemento de la fila contendrá el mismo número de bloque.
- La colección de bloques utilizados y bloques que no se usan será la colección total de bloques que constituyen los archivos.

6 Debe observarse que agregar un nombre no puede ocurrir en el estado *full* (lleno) y borrar un nombre es imposible en el estado *empty* (vacío).

**FIGURA 21.8**
**Manipulador de bloques**




- 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 estos datos son: *add()* una colección de bloques al final de la fila, *remove()* una colección de bloques usados del frente de la fila y colocarlos en la colección de bloques no utilizados, y *check()* (comprobar) si la fila de bloques está vacía.

La precondición de *add()* es que los bloques que se van a agregar deben estar en la colección de bloques usados. La poscondición es que la colección de bloques ahora se encuentra al final de la fila. La precondición de *remove()* es que la fila debe tener al menos un ítem. La poscondición es que los bloques deben agregarse a la colección de bloques no utilizados. La operación *check()* no tiene precondición. Esto significa que la operación siempre está definida, sin importar qué valor tenga el estado. La poscondición entrega el valor *true* (verdadero) si la fila está vacía y *false* (falso) de otro modo.

En los ejemplos anotados en esta sección, se introducen los conceptos clave de especificación formal, pero sin enfatizar las matemáticas que se requieren para hacer formal la especificación. En la sección 21.6, se considera cómo puede usarse la notación matemática para especificar de manera formal algún elemento del sistema.

## 21.6 APLICACIÓN DE NOTACIÓN MATEMÁTICA[7] PARA ESPECIFICACIÓN FORMAL

Para ilustrar el uso de la notación matemática en la especificación formal de un componente de software, revise el ejemplo del manipulador de bloques que se presentó en la sección 21.5. A fin de revisarlos, un importante componente del sistema operativo de una computadora mantiene los archivos que crearon los usuarios. El manipulador de bloques mantiene un reservorio de bloques sin utilizar y también seguirá la pista a los bloques que estén en uso en el momento. Cuando los bloques se liberan de un archivo borrado, por lo general se agregan a una fila de bloques que esperan para agregarse al reservorio de bloques no utilizados. Esto se muestra de manera esquemática en la figura 21.8.

7 Esta sección se escribió suponiendo que el lector está familiarizado con la notación matemática asociada con conjuntos y secuencias, y con la notación lógica que se usa en el cálculo de predicados. Si necesita un repaso, en el sitio web de la 7a. edición de este libro se presenta una breve revisión como recurso complementario. Para información más detallada, vea [Jec06] o [Pot04].

**¿Cómo pueden representarse estados e invariantes de datos usando un conjunto y operadores lógicos?**

Un conjunto llamado *BLOCKS* (bloques) consistirá de todo número de bloques. *AllBlocks* (todos los bloques) es un conjunto de bloques que se encuentra entre 1 y *MaxBlocks* (máximo de bloques). El estado se modelará mediante dos conjuntos y una secuencia. Los dos conjuntos son *used* (utilizado) y *free* (libre). Ambos contienen bloques: el conjunto *used* contiene los que actualmente se usan en los archivos y el conjunto *free* los que están disponibles para nuevos archivos. La secuencia contendrá conjuntos de bloques que están listos para ser liberados de los archivos que se borraron. El estado puede describirse como

$$used, free: \mathbb{P}\ BLOCKS$$
$$BlockQueue: seq\ \mathbb{P}\ BLOCKS$$

Esto es muy parecido a la declaración de las variables de programa. Se afirma que *used* y *free* serán conjuntos de bloques y que *BlockQueu* (fila de bloques) será una secuencia, cada elemento de la cual será un conjunto de bloques. La invariante de datos puede escribirse como

$$used \cap free = \varnothing \land$$
$$used \cup free = AllBlocks \land$$
$$\forall\ i: \text{dom}\ BlockQueue \bullet BlockQueue\ i \subseteq used \land$$
$$\forall\ i, j: \text{dom}\ BlockQueue \bullet i \neq j = BlockQueue\ i \cap BlockQueue\ j = \varnothing$$

**WebRef**

Amplia información de los métodos formales se puede encontrar en **www.afm.sbu .ac.uk**.

Los componentes matemáticos de la invariante de datos coinciden con cuatro de las características de los componentes de lenguaje natural descritos anteriormente. La primera línea de la invariante de datos afirma que no habrá bloques comunes en la colección utilizada y en las colecciones libres de bloques. La segunda afirma que la colección de bloques utilizados y de bloques libres siempre será igual a la colección completa 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 utilizados. La línea final afirma que, para cualesquiera dos elementos de la fila de bloques que no son el mismo, no habrá bloques comunes en dichos elementos. Los dos componentes de lenguaje natural finales de la invariante de datos se implementan en virtud del hecho de que *used* y *free* son conjuntos y, por tanto, no contendrán duplicados.

La primera operación por definir es aquella que remueve un elemento de la cabeza de la fila de bloques. La precondición es que debe haber al menos un ítem en la fila:

$$\#\text{BlockQueue} > 0,$$

La poscondición es que la cabeza de la fila debe removerse y colocarse en la colección de bloques libres y la fila debe ajustarse para mostrar la remoción:

$$used' = used \setminus head\ BlockQueue \land$$
$$free' = free \cup head\ BlockQueue \land$$
$$BlockQueue' = tail\ BlockQueue$$

Una convención que se usa en muchos métodos formales es que el valor de una variable después de una operación es prima. En consecuencia, el primer componente de la expresión precedente afirma que los nuevos bloques usados (*used'*) serán iguales a los antiguos bloques usados menos los bloques que se removieron. El segundo componente señala que los nuevos bloques libres (*free'*) serán los antiguos bloques libres, con el agregado de la cabeza de la fila de bloques. El tercer componente afirma que la nueva fila de bloques será igual a la fila del valor antiguo de la fila de bloques, es decir, todos los elementos en la fila menos el primero. Una segunda operación agrega una colección de bloques, *Ablocks*, a la fila de bloques. La precondición es que *Ablocks* es en la actualidad un conjunto de bloques utilizados:

**¿Cómo se representan las precondiciones y las poscondiciones?**

$$Ablocks \subseteq used$$

La poscondición es que el conjunto de bloques se agrega al final de la fila de bloques y el conjunto de bloques usados y libres permanece invariable:

$$BlockQueue' = BlockQueue \frown \langle Ablocks \rangle \land$$
$$used' = used \land$$
$$free' = free$$

No hay duda de que la especificación matemática de la fila de bloques es considerablemente más rigurosa que una narrativa de lenguaje natural o que un modelo gráfico. El rigor adicional requiere esfuerzo, pero los beneficios obtenidos de la consistencia mejorada y de la completitud pueden justificarse para algunos dominios de aplicación.

## 21.7 LENGUAJES DE ESPECIFICACIÓN FORMAL

Un lenguaje de especificación formal por lo general se compone de tres componentes primarios: 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 usarán para describir el sistema y 3) un conjunto de relaciones que definen las reglas que indican cuáles objetos satisfacen adecuadamente la especificación.

El dominio sintáctico de un lenguaje de especificación formal con frecuencia se basa en una sintaxis que se deriva de la notación estándar de la teoría de conjuntos y del cálculo de predicados. El *dominio semántico* de un lenguaje de especificación indica cómo representa el lenguaje los requerimientos del sistema.

Es posible usar diferentes abstracciones semánticas para describir el mismo sistema en formas distintas. En los capítulos 6 y 7, se hizo esto de manera menos formal. Se representaron información, función y comportamiento. Para representar el mismo sistema, puede usarse una notación de modelado diferente. La semántica de cada representación proporciona visiones complementarias del sistema. Para ilustrar este enfoque cuando se usan métodos formales, suponga que utiliza un lenguaje de especificación formal para describir el conjunto de eventos que hacen que ocurra un estado particular en un sistema. Otra relación formal muestra todas las funciones que ocurren dentro de un estado determinado. La intersección de estas dos relaciones proporciona un indicio de los eventos que producirán funciones específicas.

En la actualidad se usan varios lenguajes de especificación formales. OCL [OMG03b], Z [ISO02], LARCH [Gut93] y VDM [Jon91] son lenguajes de especificación formal representativos que muestran las características anotadas anteriormente. En este capítulo se presenta un breve estudio de OCL y Z.

### 21.7.1 Lenguaje de restricción de objeto (OCL)[8]

El *lenguaje de restricción de objeto* (OCL) es una notación formal desarrollada de modo que los usuarios de UML puedan agregar más precisión a sus especificaciones. En el lenguaje está disponible todo el poder de la lógica y de la matemática discreta. Sin embargo, los diseñadores de OCL decidieron que, en los enunciados OCL, sólo deberían usarse caracteres ASCII (en lugar de la notación matemática tradicional). Esto hace que el lenguaje sea más amistoso para las personas que tienen menos inclinación matemática y que la computadora lo procese más fácilmente. Pero también hace al OCL un poco farragoso en algunos lugares.

Para usar OCL, comience con uno o más diagramas UML: los diagramas de clase, estado o actividad más comunes (apéndice 1). Se agregan expresiones OCL y hechos de estado acerca de elementos de los diagramas. Dichas expresiones se llaman *restricciones*; cualquier implementación derivada del modelo debe asegurar que cada una de las restricciones siempre sigue siendo verdadera.

8 Esta sección es aportación del profesor Timothy Lethbridge, de la Universidad de Ottawa, y se presenta aquí con permiso.

**TABLA 21.1** RESUMEN DE NOTACIÓN OCL CLAVE

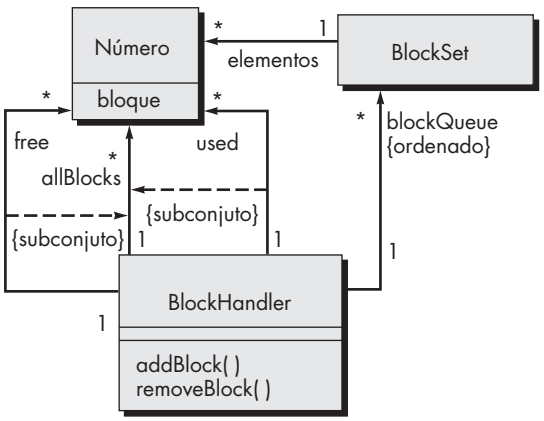
| | |
|---|---|
| x.y | Obtiene la propiedad y del objeto x. Una propiedad puede ser un atributo, el conjunto de objetos al final de una asociación, el resultado de evaluar una operación y otras cosas, dependiendo del tipo de diagrama UML. Si x es un Conjunto, entonces y se aplica a cada elemento de x; los resultados se recopilan en un nuevo Conjunto. |
| c−>f() | Aplica la operación f interna de OCL a la Colección c en sí (en oposición a cada uno de los objetos en c). A continuación se mencionan ejemplos de operaciones internas. |
| and, or, =, <> | And lógica, or lógica, igual, no es igual. |
| p implica q | Verdadero si q es verdadero o p es falso. |
| ***Muestra de operaciones sobre colecciones (incluidos conjuntos y secuencias)*** | |
| C−>size() | El número de elementos en la Colección c. |
| C−>isEmpty() | Verdadero si c no tiene elementos, falso de otro modo. |
| c1−>includesAll(c2) | Verdadero si cada elemento de c2 se encuentra en c1. |
| c1−>excludesAll(c2) | Verdadero si ningún elemento de c2 se encuentra en c1. |
| C−>forAll(elem \| boolexpr) | Verdadero si boolexpr es verdadera cuando se aplica a cada elemento de c. Conforme se evalúa un elemento, se enlaza a la variable elem, que puede usarse en boolexpr. Esto implementa cuantificación universal, que se estudió anteriormente. |
| C−>forAll(elem1, elem2 \| boolexpr) | Igual que el anterior, excepto que boolexpr se evalúa para cada posible *par* de elementos tomados de c, incluidos casos donde el par tiene el mismo elemento. |
| C−>isUnique(elem \| expr) | Verdadero si expr evalúa un valor diferente cuando se aplica a cada elemento de c. |
| ***Muestra de operaciones específicas para 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 la omisión del objeto x. |
| ***Muestra de operación específica a secuencias*** | |
| Seq−>first() | El objeto que es el primer elemento en la secuencia seq. |

Como un lenguaje de programación orientado a objeto, una expresión OCL involucra operadores que operan sobre 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 **Collection** OCL, de la cual **Set** (conjunto) y **Sequence** (secuencia) son dos subclases.

El objeto **self** es el elemento del diagrama UML en cuyo contexto se evaluará la expresión OCL. Al *navegar* usando el símbolo. (punto) del objeto **self** pueden obtenerse otros objetos. Por ejemplo:

- Si **self** es la clase **C**, con atributo a, entonces **self.a** evalúa al objeto almacenado en **a**.
- Si **C** tiene una asociación uno a muchos llamada *assoc* con otra clase **D**, entonces **self.assoc** evalúa un **Set** cuyos elementos son del tipo **D**.
- Finalmente (y un poco más sutilmente), si **D** tiene el atributo **b**, entonces la expresión **self.assoc.b** evalúa al conjunto de todos los b que pertenecen a todos los **D**.

**FIGURA 21.9**
**Diagrama de clase para un manipulador de bloques**




OCL proporciona operaciones internas que implementan operadores de conjunto y lógicos, especificación constructiva y matemáticas relacionadas. En la tabla 21.1 se presenta una pequeña muestra de aquéllas.

Para ilustrar el uso de OCL en especificación, se reexamina el ejemplo de manipulador de bloques, que se introdujo en la sección 21.5. El primer paso es desarrollar un modelo UML (figura 21.9). Este diagrama de clase especifica muchas relaciones entre los objetos involucrados. Sin embargo, las expresiones OCL se agregan para que los implementadores del sistema puedan conocer con más precisión que debe permanecer verdadero conforme corre el sistema.

Las expresiones OCL que complementan el diagrama de clase corresponden a las seis partes de la invariante que se estudió en la sección 21.5. En el ejemplo que sigue, la invariante se repite en castellano y luego se escribe la correspondiente expresión OCL. Se considera buena práctica proporcionar texto en lenguaje natural junto con la lógica formal; hacerlo así ayuda a entender la lógica, y también ayuda a los revisores a descubrir errores, por ejemplo, situaciones donde el lenguaje natural y la lógica no corresponden.

1. *Ningún bloque se marcará como no utilizado y usado al mismo tiempo.*

```
context BlockHandler inv:
   (self.used2.intersection(self.free)) 2.isEmpty()
```

Observe que cada expresión comienza con la palabra clave ***context***. Esto indica el elemento del diagrama UML que restringe la expresión. De manera alternativa, podría colocar la restricción directamente en el diagrama UML, encerrada entre llaves. Aquí la palabra clave **self** se refiere a la instancia de **BlockHandler**; en lo que sigue, como es permisible en OCL, se omitirá el **self**.

2. *Todos los conjuntos de bloques que se conservan en la fila serán subconjuntos de la colección de los bloques actualmente utilizados.*

```
context BlockHandler inv:
   blockQueue2.forAll(aBlockSet | used2.includesAll(aBlockSet ))
```

3. *Ningún elemento de la fila contendrá el mismo número de bloque.*

```
context BlockHandler inv:
   blockQueue2.forAll(blockSet1, blockSet2 |
      blockSet1 ,. blockSet2 implies
      blockSet1.elements.number2.excludesAll(blockSet2elements.number))
```

La expresión antes de **implies** es necesaria para garantizar que se ignoran los pares donde ambos elementos son el mismo bloque.

4. *La colección de bloques utilizados y bloques que no se utilizan será la colección total de bloques que constituyen los archivos.*

```
context BlockHandler inv:
   allBlocks 5 used2.union(free)
```

5. *La colección de bloques no utilizados no tendrá números de bloque duplicados.*

```
context BlockHandler inv:
   free2.isUnique(aBlock | aBlock.number)
```

6. *La colección de bloques utilizados no tendrá números de bloque duplicados.*

```
context BlockHandler inv:
   used2.isUnique(aBlock | aBlock.number)
```

OCL también puede usarse para especificar precondiciones y poscondiciones de operaciones. Por ejemplo, lo siguiente describe operaciones que remueven y agregan conjuntos de bloques a la fila. Observe que la notación **x@pre** indica el objeto **x** como existe *antes* de la operación; esto es opuesto a la notación matemática estudiada anteriormente, donde la **x** está *después* de la operación que se designa especialmente (como **x′**).

```
context BlockHandler::removeBlocks()
    pre: blockQueue2.size() .0
    post: used 5 used@pre-blockQueue@pre2.first() and
        free = free@pre2.union(blockQueue@pre2.first()) and
        blockQueue 5 blockQueue@pre2.excluding(blockQueue@pre2.first)

context BlockHandler::addBlocks(aBlockSet :BlockSet)
   pre: used2.includesAll(aBlockSet.elements)
   post: (blockQueue.elements 5 blockQueue.elements@pre
      2. append (aBlockSet.elements) and
      used 5 used@pre and
      free 5 free@pre
```

OCL es un lenguaje de modelado, pero tiene todos los atributos de un lenguaje formal. OCL permite la expresión de varias restricciones, precondiciones y poscondiciones, guardias y otras características que se relacionan con los objetos representados en varios modelos UML.

### 21.7.2 El lenguaje de especificación Z

Z es un lenguaje de especificación que se usa ampliamente dentro de la comunidad de métodos formales. El lenguaje Z se aplica a conjuntos escritos, relaciones y funciones dentro del contexto de la lógica de predicados de primer orden para construir *esquemas*, un medio para estructurar la especificación formal.

**WebRef**

En **www.users.cs.york.ac.uk/~susan/abs/z.htm**, puede encontrarse información detallada acerca del lenguaje Z.

Las especificaciones Z se organizan como un conjunto de esquemas, una estructura de lenguaje que introduce variables y que especifica la relación entre dichas variables. Un esquema es en esencia la especificación formal análoga del componente de lenguaje de programación. Los esquemas se usan para estructurar una especificación formal en la misma forma en la que los componentes se usan para estructurar un sistema.

Un esquema describe los datos almacenados a los que accede y que altera un sistema. En el contexto de Z, esto se llama "estado". Este uso del término *estado* en Z es ligeramente diferente

del uso de la palabra en el resto de este libro.[9] Además, el esquema identifica las operaciones que se aplican para cambiar el estado y las relaciones que ocurren dentro del sistema. La estructura genérica de un esquema toma la forma:

esquemaNombre
declaraciones
---
invariante

donde las declaraciones identifican las variables que abarca el estado del sistema y la invariante impone restricciones en la forma en la que puede evolucionar el estado. En la tabla 21.2 se presenta un resumen de la notación del lenguaje Z.

El siguiente ejemplo de esquema describe el estado del manipulador de bloques y la invariante de datos:

BlockHandler
*used, free* : ℙ *BLOCKS*
*BlockQueue* : seq ℙ *BLOCKS*
*used* ∩ *free* = ∅ ∧
*used* ∪ *free* = *AllBlocks* ∧
∀*i*: **dom** *BlockQueue* • *BlockQueue i* ⊆ *used* ∧
∀*i*, *j*: dom *BlockQueue* • *i* ≠ *j* => *BlockQueue i* ∩ *BlockQueue j* = ∅

Como se anotó, el esquema consiste de dos partes. La parte que está arriba de la línea central representa las variables del estado, mientras que la parte de abajo de la línea central describe la invariante de datos. Siempre que el esquema especifique operaciones que cambian al estado, se precede con el símbolo Δ. El siguiente ejemplo de esquema describe la operación que remueve un elemento de la fila de bloques:

RemoveBlocks
Δ *BlockHandler*
---
#BlockQueue > 0,
*used'* = *used* \ *head BlockQueue* ∧
*free'* = *free* ∪ *head BlockQueue* ∧
*BlockQueue'* = *tail BlockQueue*

La inclusión de Δ *BlockHandler* da como resultado todas las variables que constituyen el estado disponible para el esquema *RemoveBlocks* y garantiza que la invariante de datos se sostendrá antes y después de ejecutar la operación.

La segunda operación, que agrega una colección de bloques al final de la fila, se representa como

AddBlocks
Δ *BlockHandler*
Ablocks? : BLOCKS

9 Recuerde que, en otros capítulos, *estado* se usó para identificar un modo de comportamiento observable en el exterior para un sistema.

**TABLA 21.2** RESUMEN DE NOTACIÓN Z

La notación Z se basa en la teoría de conjuntos descrita 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 variables con una lista de predicados que restringen el posible valor de una variable. En Z, el esquema X se define mediante la forma

```
———X———————————————————
   declaraciones
———————————————————————
   predicados
———————————————————————
```

Las funciones globales y las constantes se definen mediante 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* se declara como un conjunto de *X*s. |
| $X \in S$ | x es miembro de *S*. |
| $x \notin S$ | x no es miembro de *S*. |
| $S \subseteq T$ | *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 tanto de *S* como de *T*. |
| $S \setminus T$ | La diferencia de *S* y *T*: contiene todo miembro de *S* excepto aquellos que también están en *T*. |
| $\varnothing$ | Conjunto vacío: no contiene miembros. |
| {x} | Conjunto de un solo elemento: sólo contiene a x. |
| $\mathbb{N}$ | Conjunto de los números naturales 0, 1, 2, .... |
| $S : \mathbb{F}\, X$ | *S* se declara como un conjunto finito de *X*s. |
| max (*S*) | El máximo del conjunto no vacío de números *S*. |

**Funciones:**

| | |
|---|---|
| $f : X \rightarrowtail Y$ | *f* se declara como una inyección parcial de *X* a *Y*. |
| dom *f* | El dominio de f: el conjunto de valores *x* para los cuales se define *f*(*x*). |
| ran *f* | El rango de f: el conjunto de valores tomados por *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 mapea a *y*. |
| $\{x\} \lhd\!\!\!- f$ | Una función como *f*, excepto que *x* se remueve de su dominio. |

**Lógica:**

| | |
|---|---|
| $P \wedge Q$ | *P* y *Q*: es verdadero si tanto *P* como *Q* son verdaderos. |
| $P \Rightarrow Q$ | *P* implica a *Q*: es verdadero si *Q* es verdadero o *P* es falso. |
| $\theta\, S' = \theta\, S$ | Ningún componente del esquema *S* cambia en una operación. |

$$
\begin{aligned}
&Ablocks? \subseteq used \\
&BlockQueue' = BlockQueue \frown \langle \text{Ablocks?} \rangle \wedge \\
&used' = used \wedge \\
&free' = free
\end{aligned}
$$

———————————————————————

Por convención en Z, una variable de entrada que se lea, pero que no forme parte del estado, termina con un signo de interrogación. Por tanto, Ablocks?, que actúa como un parámetro de entrada, termina con un signo de interrogación.

HERRAMIENTAS DE SOFTWARE

### *Métodos formales*

**Objetivo:** El objetivo de las herramientas de métodos formales es auxiliar al equipo de software en la verificación de especificación y de exactitud.

**Mecánica:** La mecánica de las herramientas varía. En general, las herramientas auxilian a probar la especificación y la exactitud de la automatización, que por lo general se define mediante un lenguaje especializado para probar los teoremas. Muchas herramientas no se comercializan y se desarrollaron con propósitos de investigación.

**Herramientas representativas:**[10]

*ACL2*, desarrollada en la Universidad de Texas (**www.cs.utexas.edu/users/moore/acl2/**), es "tanto un lenguaje de programación en el que pueden modelarse sistemas de cómputo como una herramienta para ayudarle a probar las propiedades de dichos modelos".

*EVES*, desarrollado por ORA Canadá (**www.ora.on.ca/eves.html**), implementa el lenguaje Verdi para especificación formal y un generador de prueba automático.

En **http://vl.fmnet.info/** puede encontrar una extensa lista de más de 90 herramientas de métodos formales.

## 21.8 RESUMEN

La ingeniería del software de cuarto limpio es un enfoque formal del desarrollo de software que puede conducir a software con una calidad notablemente alta. Usa la especificación de estructura de cajas para el análisis y el modelado del diseño, y enfatiza la verificación de la exactitud, en lugar de las pruebas, como el mecanismo primario para encontrar y remover errores. La prueba de uso estadístico se aplica para desarrollar la información de tasa de fallos necesaria para certificar la confiabilidad del software entregado.

El enfoque de cuarto limpio comienza con los modelos de análisis y diseño que usan una representación de estructura de cajas. Una "caja" encapsula el sistema (o algún aspecto de él) en un nivel específico de abstracción. Las cajas negras se usan para representar el comportamiento externamente observable de un sistema. Las cajas de estado encapsulan los datos y operaciones de estado. Una caja clara se usa para modelar el diseño procedural que se implica mediante los datos y operaciones de una caja de estado.

La verificación de exactitud se aplica una vez que está completo el diseño de estructura de cajas. El diseño procedural para un componente de software se divide en una serie de subfunciones. Para probar la exactitud de las subfunciones, se definen 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 que está completa la verificación de exactitud, comienza la prueba de uso estadístico. A diferencia de las pruebas convencionales, la ingeniería del software de cuarto limpio no enfatiza las pruebas de unidad o de integración. En vez de ello, el software se prueba al definir un conjunto de escenarios de uso, al determinar la probabilidad de uso para cada escenario y luego definir pruebas aleatorias que se conformen con las probabilidades. Los registros de error que resultan se combinan con modelos de muestreo, componentes y certificación para habilitar el cálculo matemático de la confiabilidad proyectada para el componente de software.

Los métodos formales usan las facilidades descriptivas de la teoría de conjuntos y la notación lógica para permitir que un ingeniero de software cree un enunciado claro de los hechos (requerimientos). Los conceptos subyacentes que gobiernan los métodos formales son: 1) la invariante de datos, una condición verdadera a lo largo de la ejecución del sistema que contiene una co-

10 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

lección de datos; 2) el estado, una representación del modo de comportamiento observable externamente a un sistema o (en lenguajes Z o relacionados) los datos almacenados a los que un sistema accede o que altera; y 3) la operación, una acción que tiene lugar en un sistema y que lee o escribe datos a un estado. Una operación se asocia con dos condiciones: una precondición y una poscondición.

¿Alguna vez la ingeniería del cuarto limpio o los métodos formales se usarán ampliamente? La respuesta es "probablemente no". Son más difíciles de aprender que los métodos de ingeniería del software convencional y representan un significativo "choque cultural" para algunos profesionales del software. Pero la siguiente vez que escuche a alguien lamentarse: "¿Por qué este software no puede quedar bien desde la primera vez?", sabrá que hay técnicas que le ayudan a hacer exactamente eso.

## Problemas y puntos por evaluar

**21.1.** Si tuviera que elegir un aspecto de la ingeniería del software de cuarto limpio que la haga radicalmente diferente de los enfoques de ingeniería del software convencional o de la orientada a objeto, ¿cuál sería?

**21.2.** ¿Cómo trabajan en conjunto un modelo de proceso incremental y la certificación para producir software de alta calidad?

**21.3.** Con la especificación de estructura de cajas, desarrolle modelos de análisis y diseño de "primer paso" para el sistema *CasaSegura*.

**21.4.** Un algoritmo de ordenamiento en burbujas (*bubble-sort*) se define de la manera siguiente:

```
procedure bubblesort;
var i, t, integer;
begin
repeat until t5a[1]
        t:5a[1];
        for j:5 2 to n do
            if a[j-1] . a[j] then begin
                t:5a[j-1];
                a[j-1]:5a[j];
                a[j]:5t;
                end
endrep
end
```

Divida el diseño en subfunciones y defina un conjunto de condiciones que le permitirían probar que este algoritmo es correcto.

**21.5.** Documente una prueba de verificación de exactitud para el ordenamiento en burbujas estudiado en el problema 21.4.

**21.6.** Seleccione un programa que usted use regularmente (por ejemplo, manejador de correo electrónico, procesador de palabra, programa de hoja de cálculo). Cree un conjunto de escenarios de uso para el programa. Defina la probabilidad de uso para cada escenario y luego desarrolle una tabla de estímulos del programa y distribución de probabilidad, similar a la que se muestra en la sección 21.4.1.

**21.7.** Para la tabla de estímulos del programa y distribución de probabilidad que desarrolló en el problema 21.6, use un generador de números aleatorios para desarrollar un conjunto de casos de prueba para usar en la prueba de uso estadístico.

**21.8.** Con sus palabras, describa la intención de la certificación en el contexto de la ingeniería del software de cuarto limpio.

**21.9.** Al lector lo asignan a un equipo que desarrolla software para un fax módem. Su labor es desarrollar la porción de "directorio" de la aplicación. La función directorio permite el almacenamiento de hasta *MaxNombres* personas junto con los nombres de compañía asociados, números de fax y otra información relacionada. Use lenguaje natural para definir

a) La invariante de datos.
b) El estado.
c) Las probables operaciones.

**21.10.** Al lector se le asigna a un equipo de software que desarrolla software, llamado MemoriaDuplicador, que proporciona a una PC mayor memoria aparente que la memoria física. Esto se logra al identificar, recopilar y reasignar bloques de memoria que se asignaron a una aplicación existente, pero que no se utilizan. Los bloques no utilizados se reasignan a aplicaciones que requieren memoria adicional. Realice las suposiciones adecuadas y use lenguaje natural para definir

a) La invariante de datos.
b) El estado.
c) Las probables operaciones.

**21.11.** Use la notación OCL o la Z que se presentó en las tabla 21.1 o 21.2, seleccione alguna parte del sistema de seguridad *CasaSegura* descrito anteriormente en este libro e intente especificarla con OCL o con Z.

**21.12.** Use una o más de las fuentes de información anotadas en las referencias o en *Lecturas adicionales y fuentes de información* de este capítulo, para desarrollar una presentación de media hora acerca de la sintaxis y la semántica básicas de un lenguaje de especificación formal distinto a OCL o a Z.

## Lecturas adicionales y fuentes de información

En años recientes, se han publicado relativamente pocos libros acerca de las técnicas de especificación y verificación avanzadas. Sin embargo, vale la pena considerar algunas de las nuevas adiciones a la literatura. Un libro editado por Gabbar (*Modern formal Methods and Applications*, Springer, 2006) presenta tanto fundamentos, como nuevos desarrollos y aplicaciones avanzadas. Jackson (*Software Abstractions*, the MIT Press, 2006) presenta todos los fundamentos y un enfoque que él llama "métodos formales ligeros". Monin y Hinchey (*Understanding formal Methods*, Springer, 2003) ofrecen una excelente introducción a la materia. Butler *et al.* (*Integrated Formal Methods*, Springer, 2002) presentan varios artículos acerca del tema de los métodos formales.

Además de los libros indicados en este capítulo, Prowell *et al.* (*Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999) proporcionan un tratamiento a profundidad de todos los aspectos importantes del enfoque de cuarto limpio. Poore y Trammell (*Cleanroom Software Engineering: A Reader*, Blackweel Publishing, 1996) editaron útiles análisis de temas de cuarto limpio. Becker y Whittaker (*Cleanroom Software Engineering Practies*, Idea Group Publishing, 1996) presentan un excelente panorama para quienes no están familiarizados con las prácticas de cuarto limpio.

El Cleanroom Pamphlet (Software Technology Support Center, Hill AF Base, abril de 1995) contiene reimpresiones de algunos artículos importantes. El Data and Analysis Center for Software (DACS) (**www.dacs.dtic.mil**) proporciona muchos ensayos útiles, manuales y otras fuentes de información acerca de la ingeniería del software de cuarto limpio.

La verificación de diseño mediante prueba de exactitud se encuentra en el centro del enfoque de cuarto limpio. Los libros de Cupillari (*The Nuts and Bolts of Proofs*, 3a ed., Academic Press, 2005), Solow (*How to Read and Do Proofs,* 4a. ed., Wiley, 2004), Eccles (*An Introduction to Mathematical Reasoning,* Cambridge University Press, 1998), proporcionan excelentes introducciones a los principios matemáticos. Stavely (*Toward Zero-Defect Software,* Addison-Wesley, 1998), Baber (*Error-Free Software,* Wiley, 1991) y Schulmeyer (*Zero Defect Software,* McGraw-Hill, 1990) estudian la prueba de exactitud con detalle considerable.

En el dominio de los métodos formales, los libros de Casey (*A Programming Approach to Formal Methods,* McGraw-Hill, 2000), Hinchey y Bowan (*Industrial Strength Formal Methods,* Springer-Verlag, 1999), Hussmann (*Formal Foundations for Software Engineering Methods,* Springer-Verlag, 1997) y Sheppard (*An Introduction to Formal Specification with Z and VDM,* McGraw-Hill, 1995) ofrecen guías útiles. Además, libros específicos sobre lenguaje, como los de Warmer y Kleppe (*Object Constraint Language,* Addison-Wesley, 1998), Jacky (*The Way of Z: Practical Programming with Formal Methods,* Cambridge University Press, 1997), Harry (*Formal Methods Fact File: VDM and Z,* Wiley, 1997) y Cooper y Barden (*Z in Practice,* Prentice-Hall, 1995) proporcionan introducciones útiles a los métodos formales, así como a varios lenguajes de modelado.

En internet está disponible una gran variedad de fuentes de información acerca de ingeniería del software de cuarto limpio y de los métodos formales. Una lista actualizada de referencias en la World Wide Web que son relevantes para el modelado formal y la verificación puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

# 22

# ADMINISTRACIÓN DE LA CONFIGURACIÓN DEL SOFTWARE

**CONCEPTOS CLAVE**



Cuando se construye software de computadoras, el cambio es inevitable. Y el cambio aumenta el nivel de confusión cuando los miembros de un equipo de software trabajan en un proyecto. La confusión surge cuando los cambios no se analizan antes de que se realicen, cuando no se registran antes de que se implanten, si no se reportan a quienes tienen necesidad de conocerlos o si no se controlan en forma que mejore la calidad y se reduzca el error. Babich [Bab86] analiza esto cuando afirma:

> El arte de coordinar el desarrollo de software para minimizar [...] la confusión se llama administración de la configuración, que es el arte de identificar, organizar y controlar las modificaciones que se hacen al software que construirá un equipo de programación. La meta es maximizar la productividad al minimizar los errores.

La administración de la configuración del software (ACS) es una actividad sombrilla que se aplica a lo largo del proceso de software. Puesto que el cambio puede ocurrir en cualquier momento, se desarrollan actividades ACS para 1) identificar el cambio, 2) controlar el cambio, 3) garantizar que el cambio se implementó de manera adecuada y 4) reportar los cambios a otros que puedan estar interesados.

Es importante hacer una distinción clara entre el apoyo al software y la administración de la configuración del software. El apoyo es un conjunto de actividades de ingeniería de software que ocurren después de que éste se entregó al cliente y de que se puso en operación. La administración de la configuración del software es un conjunto de actividades de rastreo y control que inicia cuando comienza un proyecto de ingeniería de software y sólo termina cuando el software se retira de la operación.

## UNA MIRADA RÁPIDA

**¿Qué es?** Cuando se construye software de computadora, ocurren cambios. Y puesto que ocurren, es necesario administrarlos de manera efectiva. La administración de la configuración del software (ACS), también llamada gestión del cambio, es un conjunto de actividades diseñadas para administrar el cambio mediante la identificación de los productos de trabajo que es probable que cambien, el establecimiento de relaciones entre ellos, la definición de mecanismos para administrar diferentes versiones de dichos productos de trabajo y el control de los cambios impuestos, así como la auditoría y reporte de los cambios realizados.

**¿Quién lo hace?** Todos los involucrados en el proceso de software se relacionan en cierta medida con la gestión del cambio, pero en ocasiones se crean posiciones de apoyo especializadas para administrar el proceso ACS.

**¿Por qué es importante?** Si no se controla el cambio, éste lo controla a uno. Y eso nunca es bueno. Es muy fácil que un torrente de cambios descontrolados convierta en caos un proyecto de software bien estructurado. Como consecuencia, la calidad del software se reduce y la entrega se demora. Por dicha razón, la gestión del cambio es parte esencial de la administración de la calidad.

**¿Cuáles son los pasos?** Puesto que muchos productos de trabajo se realizan cuando el software se construye, cada uno debe identificarse de manera única. Una vez logrado, pueden establecerse mecanismos para control de versión y de cambio. Para garantizar que la calidad se mantiene conforme se realizan cambios, audite el proceso; y para asegurarse que quienes deben conocerlos estén informados acerca de los cambios, realice reportes.

**¿Cuál es el producto final?** Un Plan de Administración de la Configuración del Software define la estrategia del proyecto para la gestión del cambio. Además, cuando se invoca ACS formal, el proceso de control de cambio produce solicitudes de cambio de software, reportes y órdenes de cambio de ingeniería.

**¿Cómo me aseguro de que lo hice bien?** Cuando todo producto de trabajo pueda explicarse, rastrearse y controlarse; cuando todo cambio pueda rastrearse y analizarse; cuando todos los que deben saber acerca de un cambio están informados, entonces la gestión del cambio se hizo correctamente.

Una meta principal de la ingeniería de software es mejorar la facilidad con la que los cambios pueden acomodarse y reducir la cantidad de esfuerzo empleado cuando deban realizarse cambios. En este capítulo se estudian las actividades específicas que permiten administrar el cambio.

## 22.1 Administración de la configuración del software

La salida del proceso de software es información que puede dividirse en tres categorías amplias: 1) programas de cómputo (tanto en el nivel de fuente como en formatos ejecutables), 2) productos de trabajo que describen los programas de cómputo (dirigidos a varios participantes) y 3) datos o contenido (incluidos dentro del programa o externos a él). Los ítems que comprenden toda la información producida como parte del proceso de software se llaman colectivamente *configuración del software*.

**Cita:**

"No hay nada permanente, excepto el cambio."

**Heráclito, 500 a.C.**

Conforme avanza el trabajo de ingeniería de software, se crea una jerarquía de *ítems de configuración del software* (ICS): un elemento de información nominado que puede ser tan pequeño como un solo diagrama UML o tan grande como el documento de diseño completo. Si cada ICS simplemente conduce a otros ICS, dará como resultado poca confusión. Por desgracia, en el proceso entra otra variable: *el cambio*, que puede ocurrir en cualquier momento, por cualquier razón. De hecho, la *Primera Ley de la Ingeniería de Sistemas* [Ber80] establece: "Sin importar dónde se esté en el ciclo de vida del sistema, el sistema cambiará, y el deseo por cambiar persistirá a lo largo del ciclo de vida."

¿Cuál es el origen de estos cambios? La respuesta a esta pregunta es tan variada como los mismos cambios. Sin embargo, existen cuatro fuentes fundamentales de cambio:

**¿Cuál es el origen de los cambios que se solicitan para el software?**

- Nuevas condiciones empresariales o de mercado dictan los cambios en los requerimientos del producto o en las reglas empresariales.
- Nuevas necesidades de los accionistas demandan modificación a los datos producidos por los sistemas de información, a la funcionalidad que entregan los productos o a los servicios que ofrece un sistema basado en computadora.
- La reorganización o crecimiento/reducción de la empresa produce cambios en las prioridades proyectadas o en la estructura del equipo de ingeniería de software.
- Restricciones presupuestales o de calendario causan una redefinición del sistema o del producto.

La administración de la configuración del software es un conjunto de actividades que se desarrollaron para administrar el cambio a lo largo del ciclo de vida del software de computadora. La ACS puede verse como una actividad que garantiza la calidad del software y que se aplica a lo largo del proceso de software. En las siguientes secciones se describen las principales tareas ACS y los conceptos importantes que pueden ayudar a gestionar el cambio.

### 22.1.1 Un escenario ACS[1]

Un escenario operativo de administración del cambio (AC) típico involucra a un gerente de proyecto que está a cargo de un grupo de software, a un gerente de configuración responsable de los procedimientos y políticas AC, a los ingenieros de software encargados de desarrollar y mantener el producto de software y al cliente que usa el producto. En el escenario, se supone que el producto es pequeño y que involucra a alrededor de 15 000 líneas de código desarrollado por un equipo de seis personas. (Observe que es posible que existan otros escenarios de equipos

1 Esta sección se extrajo de [Dar01]. El permiso especial para reproducir "Spectrum of functionality in CM System", de Susan Dart [Dar01], © 2001 de Carnegie Mellon University, se obtuvo del Software Engineering Institute.

más pequeños o más grandes, pero, en esencia, hay temas genéricos que cada uno de estos proyectos enfrenta con respecto a la AC).

**¿Cuáles son las metas de y las actividades realizadas por cada uno de los elementos constituyentes involucrados en la administración del cambio?**

En el nivel operativo, el escenario involucra varios roles y tareas. Para el gerente de proyecto, la meta es garantizar que el producto se desarrolla dentro de cierto marco temporal. Por tanto, monitorea el progreso del desarrollo y reconoce y reacciona ante los problemas. Esto se hace mediante la generación y el análisis de reportes acerca del estado del sistema de software y al realizar revisiones al sistema.

Las metas del gerente de configuración son garantizar que se sigan los procedimientos y políticas para crear, cambiar y probar el código, así como hacer accesible la información acerca del proyecto. Para implantar técnicas a fin de mantener el control sobre los cambios de código, este gerente introduce mecanismos para: realizar peticiones oficiales de cambios, evaluarlos (mediante un Consejo de Control de Cambios que sea responsable de aprobar los cambios al sistema de software) y autorizarlos. El gerente elabora y difunde la lista de tareas para los ingenieros y básicamente crea el contexto del proyecto. Además, recopila estadísticas acerca de los componentes que hay en el sistema de software, tales como la información que determina cuáles componentes del sistema son problemáticos.

**PUNTO CLAVE**

Debe existir un mecanismo para asegurar que los cambios simultáneos hechos al mismo componente se rastreen, gestionen y ejecuten de manera adecuada.

Para los ingenieros de software, la meta es trabajar eficazmente. Esto significa que los ingenieros no deben interferir innecesariamente unos con otros en la creación y prueba del código y en la producción de productos operativos de apoyo. Pero, al mismo tiempo, deben intentar comunicarse y coordinarse de manera eficiente. Específicamente, los ingenieros usan herramientas que ayudan a construir un producto de software consistente. Se comunican y coordinan al notificarse unos con otros las tareas requeridas y las tareas completadas. Los cambios se propagan a través del trabajo mutuo mediante fusión de archivos. Existen mecanismos para garantizar que, para componentes que experimentan cambios simultáneos, hay alguna forma de resolver los conflictos y la fusión de cambios. Se conserva una historia de la evolución de todos los componentes del sistema, una bitácora con las razones de los cambios y un registro de lo que realmente cambió. Los ingenieros tienen su propio espacio de trabajo para crear, cambiar, poner a prueba e integrar código. En cierto punto, el código se convierte en una línea de referencia desde la cual continúan mayores desarrollos y se realizan variantes para otras máquinas objetivo.

El cliente usa el producto. Puesto que éste se encuentra bajo control AC, el cliente sigue procedimientos formales para solicitar cambios y para indicar errores en el producto.

De manera ideal, un sistema AC utilizado en este escenario debe apoyar todos estos roles y tareas, es decir, los roles determinan la funcionalidad requerida de un sistema AC. El gerente de proyecto ve la AC como un mecanismo de auditoría; el gerente de configuración la considera un mecanismo de control, rastreo y generación de políticas; el ingeniero de software, como un mecanismo de control de cambio, construcción y acceso; y el cliente la ve como un camino para garantizar la calidad.

### 22.1.2 Elementos de un sistema de administración de la configuración

En su exhaustivo artículo acerca de la administración de la configuración del software, Susan Dart [Dar01] identifica cuatro elementos importantes que deben existir cuando se desarrolla un sistema de administración de la configuración:

- *Elementos componentes:* conjunto de herramientas acopladas dentro de un sistema de administración de archivos (por ejemplo, base de datos) que permite el acceso a cada ítem de configuración del software, así como su gestión.
- *Elementos de proceso:* colección de acciones y tareas que definen un enfoque efectivo de la gestión del cambio (y actividades relacionadas) para todos los elementos constituyentes involucrados en la administración, ingeniería y uso del software.

- *Elementos de construcción:* conjunto de herramientas que automatizan la construcción de software al asegurarse de que se ensambló el conjunto adecuado de componentes validados (es decir, la versión correcta).
- *Elementos humanos:* conjunto de herramientas y características de proceso (que abarcan otros elementos AC) utilizados por el equipo de software para implementar ACS efectiva.

Estos elementos (que se estudiarán con más detalle en secciones posteriores) no son mutuamente excluyentes. Por ejemplo, los elementos componentes trabajan en conjunción con los elementos de construcción conforme evoluciona el proceso de software. Los elementos de proceso guían muchas actividades humanas que se relacionan con la ACS y, por tanto, también pueden considerarse como elementos humanos.

### 22.1.3 Líneas de referencia


*La mayoría de los cambios de software se justifican, así que no hay razón para quejarse por su presencia. En su lugar, asegúrese de que tiene los mecanismos para lidiar con ellos.*

El cambio es un hecho de vida en el desarrollo de software. Los clientes quieren modificar los requerimientos. Los desarrolladores quieren cambiar el enfoque técnico. Los gerentes quieren modificar la estrategia del proyecto. ¿Por qué todas estas modificaciones? La respuesta realmente es muy simple. Conforme pasa el tiempo, todos los elementos constituyentes saben más (acerca de lo que necesitan, sobre qué enfoque sería mejor, cómo realizarlo y todavía obtener dinero). Este conocimiento adicional es la fuerza motora que hay detrás de la mayoría de los cambios y que conduce a un enunciado de hechos que es difícil de aceptar para muchos profesionales de la ingeniería de software: *¡la mayoría de los cambios se justifican!*

Una *línea de referencia* es un concepto de administración de la configuración del software que le ayuda a controlar el cambio sin impedir seriamente cambios justificados. El IEEE (IEEE Std. No. 610.12-1990) define una línea de referencia como:

> Una especificación o producto que se revisó formalmente y con el que se estuvo de acuerdo, que a partir de entonces sirve como base para un mayor desarrollo y que puede cambiar sólo a través de procedimientos de control de cambio formal.

Antes de que un ítem de configuración del software se convierta en línea de referencia, los cambios pueden realizarse rápida e informalmente. No obstante, una vez establecida la línea de referencia, pueden realizarse cambios, pero debe aplicarse un procedimiento formal específico para evaluar y verificar cada uno de ellos.

En el contexto de la ingeniería de software, una línea de referencia es un hito en el desarrollo del software. Una línea de referencia se marca al entregar uno o más ítems de configuración del software que se aprobaron como consecuencia de una revisión técnica (capítulo 15). Por ejemplo, los elementos de un modelo de diseño se documentaron y revisaron. Se encontraron y corrigieron errores. Una vez que todas las partes del modelo se revisaron, corrigieron y luego aprobaron, el modelo de diseño se convierte en línea de referencia. Los cambios adicionales a la arquitectura del programa (documentada en el modelo de diseño) pueden realizarse sólo después de que cada uno se evalúa y aprueba. Aunque las líneas de referencia pueden definirse en cualquier nivel de detalle, en la figura 22.1 se muestran las líneas de referencia de software más comunes.


*Asegúrese de que la base de datos del proyecto se mantenga en una ubicación centralizada controlada.*

En la figura 22.1 también se ilustra la progresión de eventos que conducen a una línea de referencia. Las tareas de la ingeniería de software producen uno o más ICS. Después de revisar y aprobar los ICS, se colocan en una *base de datos del proyecto* (también llamada *librería de proyecto* o *repositorio de software*, y que se estudia en la sección 22.2). Cuando un miembro de un equipo de ingeniería de software quiere hacer una modificación a un ICS que se ha convertido en línea de referencia, se copia de la base de datos del proyecto en el espacio de trabajo privado del ingeniero. Sin embargo, este ICS extraído puede modificarse solamente si se siguen controles ACS (que se estudian más adelante en este capítulo). Las flechas de la figura 22.1 ilustran la ruta de modificación de un ICS convertido en línea de referencia.

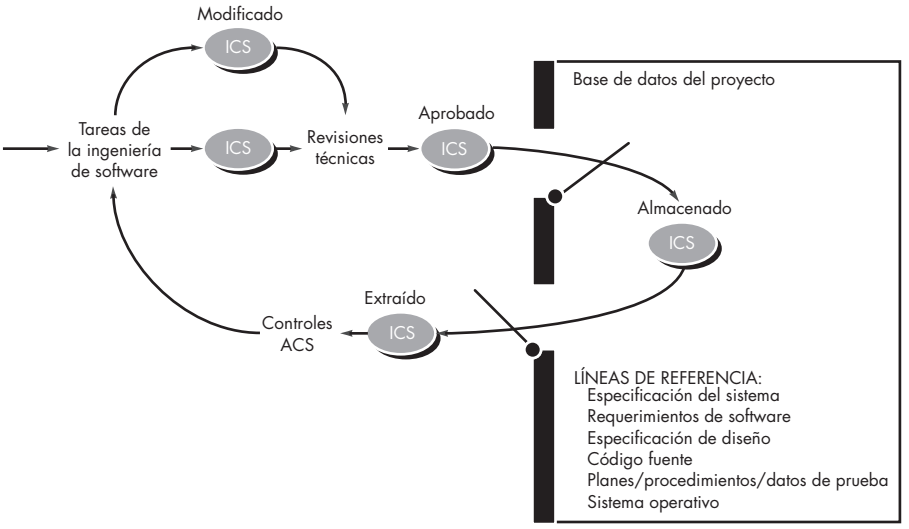



**FIGURA 22.1**
**ICS como línea de referencia y base de datos del proyecto**

### 22.1.4 Ítems de configuración del software

Ya se definió un ítem de configuración del software como la información que se crea como parte del proceso de ingeniería de software. En última instancia, un ICS podría considerarse como una sola sección de una gran especificación o como un caso de prueba en una gran suite de pruebas. De manera más realista, un ICS es todo o parte de un producto de trabajo (por ejemplo, un documento, toda una suite de casos de prueba o un componente de programa nominado).

Además de los ICS que se derivan de los productos de trabajo de software, muchas organizaciones de ingeniería de software también colocan las herramientas de software bajo control de configuración, es decir, versiones específicas de editores, compiladores, navegadores y otras herramientas automatizadas se "congelan" como parte de la configuración del software. Puesto que dichas herramientas se usaron para producir documentación, código fuente y datos, deben estar disponibles cuando tengan que realizarse cambios a la configuración del software. Aunque los problemas son raros, es posible que una nueva versión de una herramienta (por ejemplo, un compilador) pueda producir resultados diferentes que la versión original. Por esta razón, las herramientas, como el software que ayudan a producir, pueden convertirse en líneas de referencia como parte de un proceso amplio de administración de la configuración.

En realidad, los ICS se organizan para formar objetos de configuración que puedan catalogarse con un solo nombre en la base de datos del proyecto. Un *objeto de configuración* tiene un nombre y atributos, y está "conectado" con otros objetos mediante relaciones. En la figura 22.2, los objetos de configuración **DesignSpecification** (especificación de diseño), **DataModel** (modelo de datos), **ComponentN** (componente n), **SourceCode** (código fuente) y **TestSpecification** (especificación de prueba) se definen cada uno por separado. Sin embargo, cada uno de los objetos se relaciona con los demás, como se muestra mediante las flechas. Una flecha curva indica una relación composicional, es decir, **DataModel** y **ComponentN** son parte del objeto **DesignSpecification**. Una flecha con doble punta indica una interrelación. Si se realizara un cambio al objeto **SourceCode**, las interrelaciones permiten determinar qué otros objetos (e ICS) pueden resultar afectados.[2]

2 Esas relaciones se definen dentro de la base de datos. La estructura de la base de datos (repositorio) se estudia con mayor detalle en la sección 22.2.

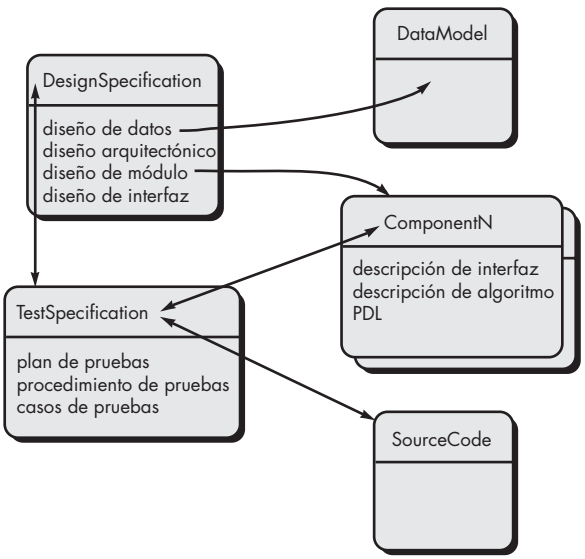



**FIGURA 22.2**
**Objetos de configuración**

## 22.2 EL REPOSITORIO ACS

En los primeros días de la ingeniería de software, los ítems de configuración del software se mantenían como documentos en papel (¡o tarjetas perforadas!), colocadas en carpetas de papel o de anillos, y se almacenaban en archiveros metálicos. Este mecanismo era problemático por muchas razones: 1) con frecuencia era difícil encontrar un ítem de configuración cuando se necesitaba, 2) era muy desafiante determinar cuáles ítems cambiaban, cuándo y por quién, 3) construir una nueva versión de un programa existente consumía mucho tiempo y era proclive al error y 4) describir relaciones detalladas y complejas entre los ítems de configuración era virtualmente imposible.

En la actualidad, los ICS se mantienen en una base de datos del proyecto, o repositorio. El *Diccionario Webster* define la palabra *repositorio* como "cualquier cosa o persona que se considera como centro de acumulación o almacenamiento". Durante la historia temprana de la ingeniería de software, de hecho el repositorio era una persona: el programador que debía recordar la ubicación de toda la información relevante para un proyecto de software, quien debía recuperar la información que nunca se escribió y reconstruir la información perdida. Tristemente, usar a una persona como "el centro para acumulación y almacenamiento" (aun conforme con la definición del Webster) no funciona muy bien. En la actualidad, el repositorio es una "cosa": una base de datos que actúa como el centro de acumulación y de almacenamiento de la información de ingeniería de software. El papel de la persona (el ingeniero del software) es interactuar con el repositorio, usando las herramientas que se integran con él.

### 22.2.1 El papel del repositorio

El repositorio ACS es el conjunto de mecanismos y estructuras de datos que permiten a un equipo de software administrar el cambio en forma efectiva. Proporciona las funciones obvias de un moderno sistema de administración de base de datos, al asegurar integridad, posibilidad de compartir e integración de datos. Además, el repositorio ACS proporciona un centro para la integración de herramientas de software, es fundamental en el flujo del proceso de software y puede reforzar la estructura y el formato uniforme para los productos que son resultado de la ingeniería de software.

Para lograr estas capacidades, el repositorio se define como un metamodelo. El *metamodelo* determina cómo se almacena la información en el repositorio, cómo pueden acceder las herramientas a los datos y cómo pueden verlas los ingenieros de software, cuán bien pueden mantenerse la seguridad y la integridad de los datos y cuán fácilmente puede extenderse el modelo existente para alojar nuevas necesidades.

### 22.2.2 Características y contenido generales

Las características y el contenido del repositorio se entienden mejor al observarlo desde dos perspectivas: qué debe almacenarse en él y qué servicios específicos proporciona. En la figura 22.3 se presenta un desglose detallado de los tipos de representaciones, documentos y otros productos de trabajo.

**WebRef**

En **www.oracle.com/technology/products/repository/index.html** puede obtenerse un ejemplo de repositorio disponible en el mercado.

Un repositorio robusto proporciona dos clases de servicios diferentes: 1) los mismos tipos de servicios que pueden esperarse de cualquier sistema sofisticado de administración de base de datos y 2) los servicios que son específicos del entorno de ingeniería de software.

Un repositorio que sirve a un equipo de ingeniería de software también debe: 1) integrarse con o directamente apoyar las funciones de administración del proceso, 2) apoyar reglas específicas que gobiernan la función ACS y los datos mantenidos dentro del repositorio, 3) proporcionar una interfaz hacia otras herramientas de ingeniería de software y 4) acomodar almacenamiento de objetos de datos sofisticados (por ejemplo, texto, gráficos, video, audio).

**PUNTO CLAVE**

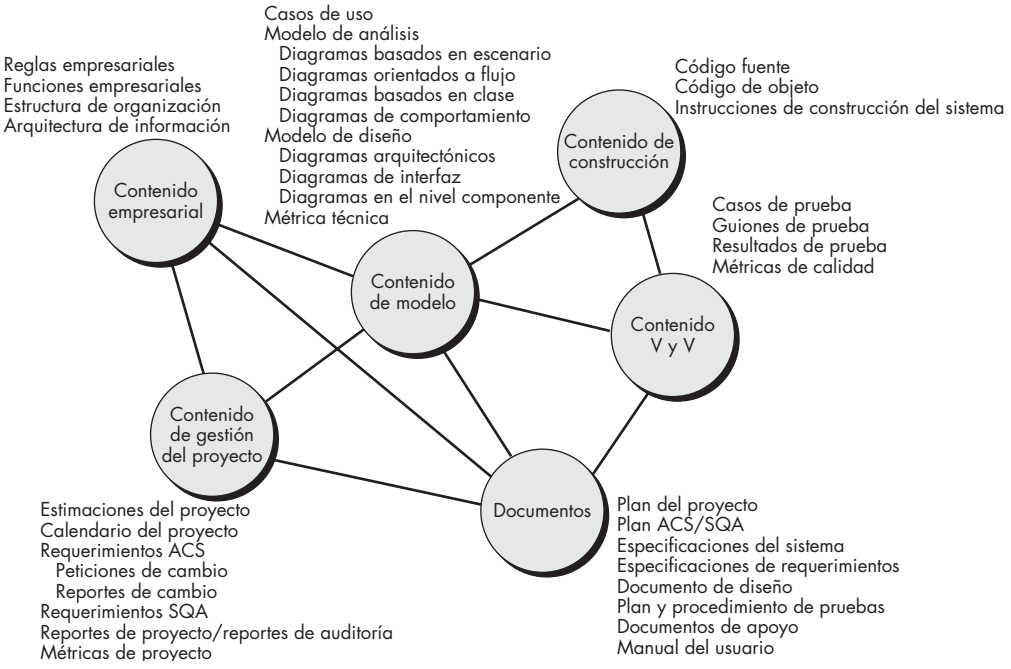
El repositorio debe mantener relacionados los ICS con muchas diferentes versiones del software. Más importante, debe proporcionar los mecanismos para ensamblar dichos ICS en una configuración específica de una versión.

### 22.2.3 Características ACS

Para apoyar el ACS, el repositorio debe tener un conjunto de herramientas que proporcionan apoyo a las siguientes características:

**Versiones.** Conforme avanza el proyecto, se crearán muchas versiones (sección 22.3.2) de productos resultantes individuales. El repositorio debe guardar todas estas versiones para permitir la administración efectiva de los productos liberados y, a los desarrolladores, regresar a versiones anteriores durante las pruebas y la depuración.

**FIGURA 22.3**
**Contenido del repositorio**



El repositorio debe controlar una amplia variedad de tipos de objeto, incluidos texto, gráficos, mapas de bits, documentos complejos y objetos únicos, como definiciones de pantalla y reportes, archivos objeto, datos de prueba y resultados. Un repositorio maduro rastrea versiones de objetos con niveles arbitrarios de granularidad; por ejemplo, puede rastrearse una sola definición de datos o un grupo de módulos.

**Rastreo de dependencia y gestión del cambio.** El repositorio administra una amplia variedad de relaciones entre los elementos de datos almacenados en él. En éstos 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 de la empresa, entre elementos de diseño y entregables, etcétera. Algunas de estas relaciones son meras asociaciones, y otras son dependencias o relaciones obligatorias.

La capacidad de seguir la pista de todas estas relaciones es vital para la integridad de la información almacenada en el repositorio y para la generación de entregables con base en él, y es una de las aportaciones más importantes del concepto de repositorio para la mejora del proceso de software. Por ejemplo, si un diagrama de clase UML se modifica, el repositorio puede detectar si clases relacionadas, descripciones de interfaz y componentes de código también requieren modificación y si pueden llevar los ICS afectados a la atención del desarrollador.

**Rastreo de requerimientos.** Esta función especial depende de la administración de vínculos y ofrece la capacidad de rastrear todos los componentes de diseño y construcción, así como entregables que resulten de una especificación de requerimientos determinada (rastreo hacia adelante). Además, proporciona la capacidad de identificar qué requisito genera algún producto de trabajo determinado (rastreo hacia atrás).

**Administración de la configuración.** Una instalación de administración de la configuración sigue la pista a una serie de configuraciones que representa hitos de proyecto específicos o liberaciones de producción.

**Ensayos de auditoría.** Un ensayo de auditoría establece información adicional acerca de cuándo, por qué y quién realiza los cambios. La información acerca de la fuente de los cambios puede ingresarse como atributos de objetos específicos en el repositorio. Un mecanismo de activación de repositorio es útil para que siempre que se modifique un elemento de diseño, se avise al desarrollador, o a la herramienta que se utilice, el inicio de la entrada de la información de auditoría (como la razón para un cambio).

## 22.3 El proceso ACS

**Cita:**

"Cualquier cambio, incluso para mejorar, está acompañado de inconvenientes e incomodidades."

**Arnold Bennett**

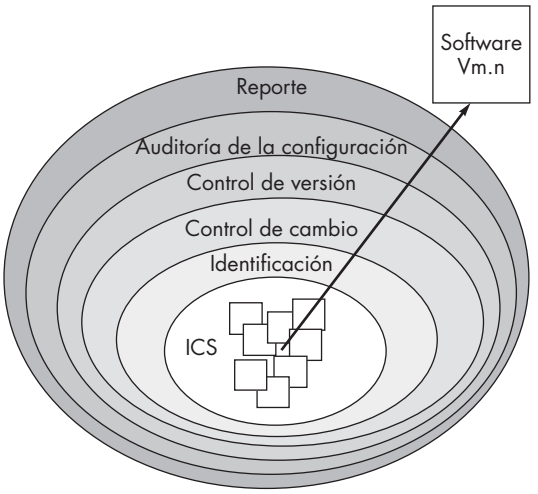
El proceso de administración de la configuración del software define una serie de tareas que tienen cuatro objetivos principales: 1) identificar todos los ítems que de manera colectiva definen la configuración del software, 2) administrar los cambios a uno o más de estos ítems, 3) facilitar la construcción de diferentes versiones de una aplicación y 4) garantizar que la calidad del software se conserva conforme la configuración evoluciona con el tiempo.

Un proceso que logra dichos objetivos no necesita ser burocrático o pesado, pero debe caracterizarse de forma que permita a un equipo de software desarrollar respuestas a un conjunto de preguntas complejas:

**¿Qué preguntas debe responder el proceso ACS?**

- ¿Cómo identifica un equipo de software los elementos discretos de una configuración de software?
- ¿Cómo gestiona una organización las muchas versiones existentes de un programa (y su documentación) de manera que permita que el cambio se acomode eficientemente?

**FIGURA 22.4**
**Capas del proceso ACS**




- ¿Cómo controla una organización los cambios antes y después de que el software se libera a un cliente?
- ¿Quién tiene la responsabilidad de aprobar y clasificar los cambios solicitados?
- ¿Cómo puede garantizarse que los cambios se realizaron adecuadamente?
- ¿Qué mecanismo se usa para enterar a otros acerca de los cambios que se realizaron?

Estas preguntas conducen a la definición de las cinco tareas ACS (identificación, control de versión, control de cambio, auditoría de la configuración y reporte) que se ilustran en la figura 22.4.

En la figura, las tareas ACS pueden visualizarse como capas concéntricas. Los ICS fluyen hacia afuera a través de estas capas a lo largo de su vida útil, y a final de cuentas se vuelven parte de la configuración del software de una o más versiones de una aplicación o sistema. Conforme un ICS se mueve a través de una capa, las acciones que implica cada tarea ACS pueden o no ser aplicables. Por ejemplo, cuando se crea un nuevo ICS, debe identificársele. Sin embargo, si no se solicitan cambios para el ICS, la capa de control de cambio no se aplica. El ICS se asigna a una versión específica del software (entran en juego los mecanismos de control de versión). Un registro del ICS (su nombre, fecha de creación, designación de la versión, etcétera) se conserva con propósitos de auditoría de la configuración y se reporta a quienes tienen necesidad de conocerlo. En las secciones siguientes se examina con más detalle cada una de estas capas del proceso ACS.

### 22.3.1 Identificación de objetos en la configuración del software

Para controlar y administrar ítems de configuración del software, cada uno debe nombrarse por separado y luego organizarse usando 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 que se crea durante el análisis, el diseño, el código o la prueba. Por ejemplo, un objeto básico puede ser una sección de una especificación de requerimientos, parte de un modelo de diseño, código fuente para un componente o una suite de casos de prueba que se utilice para ejercitar el código. Un *objeto agregado* es una colección de objetos básicos y de otros objetos agregados.

3 El concepto de objeto agregado [Gus89] se ha propuesto como un mecanismo para representar una versión completa de una configuración de software.

Por ejemplo, un **DesignSpecification** es un objeto agregado. Conceptualmente, puede vérsele como una lista nominada (identificada) de punteros que especifican objetos agregados, como **ArchitecturalModel** y **DataModel**, y *objetos básicos*, como **ComponentN** y **UMLClassDiagramN**.

Las interrelaciones establecidas por los objetos de configuración le permiten valorar el impacto del cambio.

Cada objeto tiene un conjunto de características distintivas que lo identifican de manera única: un nombre, una descripción, una lista de recursos y una "realización". El nombre del objeto es una cadena de caracteres que identifica el objeto sin ambigüedades. La descripción del objeto es una lista de ítems de datos que identifican el tipo ICS (por ejemplo, elemento de modelo, programa, datos) representado por el objeto, un identificador de proyecto e información de cambio y/o versión. Los recursos son "entidades que se proporcionan, procesan, referencian o que de algún modo el objeto las requiere" [Cho89]. Por ejemplo, los tipos de datos, las funciones específicas o incluso los nombres de variable pueden considerarse como recursos del objeto. La realización es un puntero hacia la "unidad de texto" para un objeto básico, y nulo para un objeto agregado.

La identificación del objeto de configuración también puede considerar las relaciones que existen entre los objetos nominados. Por ejemplo, al usar la notación simple

```
Diagrama clase <parte de> modelo requerimientos;
Modelo requerimientos <parte de> especificación requerimientos;
```

se puede crear una jerarquía de ICS.

*Incluso si la base de datos del proyecto proporciona la capacidad para establecer dichas relaciones, éstas consumen mucho tiempo para su establecimiento y son difíciles de mantener actualizadas. Aunque son muy útiles para el análisis de impacto, no son esenciales para la administración de cambio global.*

En muchos casos, los objetos se interrelacionan a través de ramas de la jerarquía de objetos. Estas relaciones transestructurales pueden representarse en la forma siguiente:

```
ModeloDatos <interrelacionado> ModeloFlujoDatos
ModeloDatos <interrelacionado> CasoPruebaClaseM
```

En el primer caso, la interrelación se efectúa 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 objetos de software debe reconocer que los objetos evolucionan a lo largo del proceso de software. Antes de que un objeto se convierta en línea de referencia, puede cambiar muchas veces; incluso, después de establecer una línea de referencia, los cambios pueden ser bastante frecuentes.

### 22.3.2 Control de versión

El control de versión combina procedimientos y herramientas para administrar diferentes versiones de objetos de configuración que se crean durante el proceso de software. Un sistema de control de versión implementa o se integra directamente con cuatro grandes capacidades: 1) una base de datos de proyecto (repositorio) que almacena todos los objetos de configuración relevantes, 2) una capacidad de *administración de versión* que almacena todas las versiones de un objeto de configuración (o que permite la construcción de cualquier versión usando diferencias de las versiones pasadas) y 3) una *facilidad para elaboración* que le permite recopilar todos los objetos de configuración relevantes y construir una versión específica del software. Además, los sistemas de control de versión y de control de cambio con frecuencia implementan una capacidad de *rastreador de conflictos* (también llamado *rastreador de errores*) que permite al equipo registrar y rastrear el estado de todos los conflictos sobresalientes asociados con cada objeto de configuración.

Algunos sistemas de control de versión establecen un *conjunto de cambio*, una colección de todos los cambios (en relación con cierta configuración de referencia) que se requieren para crear una versión específica del software. Dart [Dar91] observa que un conjunto de cambio

"captura todos los cambios habidos en todos los archivos que hay en la configuración, junto con la razón de los cambios y detalles de quién y cuándo hizo los cambios".

Algunos conjuntos de cambio nominados pueden identificarse para una aplicación o sistema. Esto permite construir una versión del software al especificar los conjuntos de cambios (por nombre) que deben aplicarse a la configuración de referencia. Para lograr esto, se aplica un enfoque de *modelado de sistema*. El modelo del sistema contiene: 1) una *plantilla* que incluye una jerarquía de componente y un "orden de construcción" para los componentes que describen cómo debe construirse el sistema, 2) reglas de construcción y 3) reglas de verificación.[4]

Durante las décadas pasadas se propusieron algunos enfoques automatizados diferentes para el control de versión. La diferencia principal en los enfoques es la sofisticación de los atributos que se usan para construir versiones específicas y variantes de un sistema y la mecánica del proceso para su construcción.

**HERRAMIENTAS DE SOFTWARE**

### *El Sistema de Versiones Concurrentes (SVC)*

El uso de herramientas para lograr el control de versión es esencial para la administración efectiva del cambio. El *sistema de versiones concurrentes* (SVC) es una herramienta ampliamente utilizada para el control de versiones. Originalmente diseñada para código fuente, pero útil para cualquier archivo basado en texto, el SVC 1) establece un repositorio simple, 2) mantiene todas las versiones de un archivo en un solo archivo nominado al almacenar sólo las diferencias entre versiones progresivas del archivo original y 3) protege a un archivo contra los cambios simultáneos al establecer diferentes directorios para cada desarrollador, lo que, por tanto, aísla uno de otros. El SVC mezcla los cambios cuando cada desarrollador completa su trabajo.

Es importante observar que el SVC no es un sistema "de construcción", es decir, no construye una versión específica del software. Deben integrarse otras herramientas (por ejemplo, *Makefile*) con el SVC para lograr esto. El SVC no implanta un proceso de control de cambio (por ejemplo, solicitudes de cambio, reportes de cambio, rastreo de errores).

Incluso con estas limitaciones, el SVC "es un dominante sistema de control de versión, de red transparente y código abierto, [que] es útil para todos, desde desarrolladores individuales hasta grandes equipos distribuidos" [CVS07]. Su arquitectura cliente-servidor permite a los usuarios acceder a los archivos mediante conexiones de internet, y su filosofía de código abierto lo vuelve disponible para la mayoría de las plataformas más usadas.

SVC está disponible sin costo alguno para entornos Windows, Mac OS, LINUX y UNIX. Vea [CVS07] para más detalles.

### 22.3.3 Control de cambio

**Cita:**

**"El arte de avanzar es preservar el orden en medio del cambio y preservar el cambio en medio del orden."**

**Alfred North Whitehead**

La realidad del control de cambio en un contexto moderno de ingeniería de software la resume bellamente James Bach [Bac98]:

> El control del cambio es vital. Pero las fuerzas que lo hacen necesario también lo hacen desconcertante. Nos preocupamos por el cambio porque una pequeña perturbación en el código puede crear una gran falla en el producto. Pero también puede corregir un gran fallo o permitir maravillosas nuevas capacidades. Nos preocupamos por el cambio porque un solo desarrollador granuja podría hundir el proyecto, aunque en las mentes de dichos granujas se originan ideas brillantes y un abrumador proceso de control del cambio podría efectivamente desalentarlos de hacer trabajo creativo.

Bach reconoce que se está frente a un acto de equilibrio. Mucho control del cambio y se crearán problemas. Muy poco y se crearán otros problemas.

Para un gran proyecto de software, el cambio descontrolado conduce rápidamente al caos. Para tales proyectos, el control del cambio combina procedimientos humanos y herramientas

4 También es posible consultar el modelo del sistema para valorar cómo impactará un cambio en un componente a otros componentes.

**FIGURA 22.5**
**El proceso de control del cambio**


**PUNTO CLAVE**
Cabe señalar que un número de peticiones de cambios se pueden combinar para resultar en una OCI única y esas OCI típicamente provocan cambios en los objetos de configuración múltiple.

automatizadas a fin de proporcionar un mecanismo para el control del cambio. El proceso de control del cambio se ilustra de manera esquemática en la figura 22.5. Una *petición de cambio* se envía y evalúa para valorar el mérito técnico, 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 *reporte de cambio*, que utiliza una *autoridad de control del cambio* (ACC), es decir, una persona o un grupo que toma una decisión final acerca del estatus y la prioridad del cambio. Por cada cambio aprobado se genera una *orden de cambio de ingeniería* (OCI). La OCI describe el cambio que se va a realizar, las restricciones que deben respetarse y los criterios para revisar y auditar.

El objeto que se va a cambiar puede colocarse en un directorio que controlan exclusivamente los ingenieros de software que realizan el cambio. Un sistema de control de versión (vea la barra lateral SVC) actualiza el archivo original una vez que se realiza el cambio. Como alternativa, el objeto que se va a cambiar puede "sacarse" de la base de datos del proyecto (repositorio), realizarse el cambio y aplicarse las actividades adecuadas de SQA. Luego, el objeto "entra" a la base de datos y se usan mecanismos de control de versión adecuados (sección 22.3.2) para crear la siguiente versión del software.

Dichos mecanismos de control de versión, integrados dentro del proceso de control de cambio, implementan dos importantes elementos de la gestión del cambio: control del acceso y control de la sincronización. El *control del acceso* determina qué ingenieros de software tienen la autoridad para acceder y modificar un objeto de configuración particular. El *control de la sincronización* ayuda a garantizar que cambios paralelos, realizados por dos personas diferentes, no se sobreescriban mutuamente.

Acaso se sienta incómodo con el nivel de burocracia que implica la descripción del proceso de control de cambio que se muestra en la figura 22.5. Este sentimiento no es raro. Sin salvaguardas adecuadas, el control del cambio puede retardar el progreso y crear burocracia innecesaria. La mayoría de los desarrolladores de software que tienen mecanismos de control del cambio (por desgracia, muchos no los tienen) han creado algunas capas de control para auxiliarse y evitar los problemas que se mencionan aquí.


*Opte por un poco más de control de cambio del que crea que necesitará. Es probable que demasiado sea la cantidad correcta.*

Antes de que un ICS se convierta en referencia, sólo es necesario aplicar *control de cambio informal*. El desarrollador del objeto de configuración (ICS) en cuestión puede hacer cualquier cambio que sea justificado por el proyecto y por los requerimientos técnicos, en tanto los cambios no afecten requerimientos más amplios del sistema que se encuentren afuera del ámbito de trabajo del desarrollador. Una vez que el objeto experimenta revisión técnica y se aprueba, puede crearse una línea de referencia.[5] Cuando el ICS se convierte en referencia, se implementa un *control de cambio en el nivel del proyecto*. Entonces, para hacer un cambio, el desarrollador debe obtener la aprobación del gerente del proyecto (si el cambio es "local") o del ACC (si el cambio afecta a otros ICS). En algunos casos, la generación formal de peticiones de cambio, reportes de cambio y OCI se otorgan en conjunto. Sin embargo, se realiza la valoración de cada cambio y todos los cambios se rastrean y revisan.

Cuando el producto de software se libera a los clientes, se instituye el *control de cambio formal*. El procedimiento de control de cambio formal se delineó en la figura 22.5.

**Cita:**

"El cambio es inevitable, excepto para las máquinas expendedoras."

**Letrero en un parachoques**

La autoridad de control de 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 componerse de una persona (el gerente de proyecto) o de algunas personas (por ejemplo, representantes de software, hardware, ingeniería de base de datos, apoyo, mercadotecnia). El papel de la ACC es adoptar una visión global, es decir, valorar el impacto del cambio más allá del ICS en cuestión. ¿Cómo afectará el cambio al hardware? ¿Cómo lo hará en el desempeño? ¿Cómo modificará la percepción de los clientes acerca del producto? ¿Cómo afectará la calidad y confiabilidad del producto? La ACC debe abordar éstas y muchas otras preguntas.

**CASASEGURA**

### *Conflictos ACS*

**La escena:** Oficina de Doug Miller en el momento de comenzar el proyecto de software *CasaSegura*.

**Participantes:** Doug Miller (gerente del equipo de ingeniería de software *CasaSegura*), Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

**La conversación:**

**Doug:** Ya sé que es temprano, pero tenemos que hablar acerca de la gestión del cambio.

**Vinod (ríe):** Difícilmente. Mercadotecnia llamó esta mañana con algunas "segundas opiniones". Nada importante, pero sólo es el comienzo.

5 Una línea de referencia también puede crearse por otras razones. Por ejemplo, cuando se crean "construcciones diarias", todos los componentes verificados en un momento determinado se convierten en la línea de referencia para el trabajo del día siguiente.

**Jamie:** En proyectos anteriores fuimos bastante informales acerca de la administración del cambio.

**Doug:** Lo sé, pero éste es más grande y más visible y, según recuerdo...

**Vinod (cabecea):** Nos matamos con cambios incontrolables en el proyecto de control de iluminación de la casa... recuerdo las demoras que...

**Doug (frunce el ceño):** Una pesadilla que prefiero no revivir.

**Jamie:** Así que, ¿qué hacemos?

**Doug:** Según veo, tres cosas. Primero, tenemos que desarrollar, o pedir prestado, un proceso de control de cambio.

**Jamie:** Quieres decir: ¿cómo solicitan las personas los cambios?

**Vinod:** Sí, pero también cómo se evalúa el cambio, cómo se decide cuándo hacerlo (si eso es lo que se decide) y cómo se conservan registros de lo que afecta el cambio.

**Doug:** Segundo, debemos conseguir una herramienta ACS realmente buena para el control de cambios y de versión.

**Jamie:** Podemos construir una base de datos para todos los productos de trabajo.

**Vinod:** En este contexto se llaman ICS, y la mayoría de las buenas herramientas proporcionan cierto soporte para eso.

**Doug:** Ése es un buen comienzo, ahora tenemos que...

**Jamie:** Sí, Doug, dijiste que había tres cosas...

**Doug (sonríe):** Tercero: todos debemos comprometernos en seguir el proceso de administración del cambio y usar las herramientas, sin importar cuáles sean, ¿está claro?

### 22.3.4 Auditoría de configuración

La identificación, control de versión y control del cambio ayudan a conservar el orden en lo que de otro modo sería una situación caótica y fluida. Sin embargo, incluso los más exitosos mecanismos de control rastrean un cambio sólo hasta que se genera una OCI. ¿Cómo puede un equipo de software asegurarse de que el cambio se implementó adecuadamente? La respuesta es doble: 1) revisiones técnicas y 2) auditoría a la configuración del software.

La revisión técnica (capítulo 15) se enfoca en la exactitud técnica del objeto de configuración que se modificó. Los revisores valoran el ICS para determinar la consistencia con otros ICS, así como omisiones o potenciales efectos colaterales. Una revisión técnica debe realizarse para todos los cambios, salvo los más triviales.

Una *auditoría de configuración del software* complementa la revisión técnica al valorar un objeto de configuración acerca de las características que por lo general no se consideran durante la revisión. La auditoría hace y responde las siguientes preguntas:

**? ¿Cuáles son las preguntas principales que se plantean durante una auditoría de configuración?**

1. ¿Se realizó el cambio especificado en la OCI? ¿Se incorporó alguna modificación adicional?
2. ¿Se llevó a cabo una revisión técnica para valorar la exactitud técnica?
3. ¿Se siguió el proceso del software y se aplicaron adecuadamente los estándares de ingeniería de software?
4. El cambio se "resaltó" en el ICS? ¿Se especificaron la fecha del cambio y el autor del cambio? ¿Los atributos del objeto de configuración reflejan el cambio?
5. ¿Se siguieron los procedimientos ACS para anotar, registrar y reportar el cambio?
6. ¿Los ICS relacionados se actualizaron adecuadamente?

En algunos casos, las preguntas de la auditoría se plantean como parte de una revisión técnica. No obstante, cuando la ACS es una actividad formal, la auditoría de la configuración la realiza por separado el grupo de aseguramiento de la calidad. Tales auditorías formales de la configuración también garantizan que los ICS correctos (por versión) se incorporan en una construcción específica, y que toda la documentación se actualizó y es consistente con la versión que se construyó.

### 22.3.5 Reporte de estado

El *reporte del estado de la configuración* (en ocasiones llamado *contabilidad de estado*) es una tarea ACS que responde las siguientes preguntas: 1) ¿Qué ocurrió? 2) ¿Quién lo hizo? 3) ¿Cuándo ocurrió? 4) ¿Qué más se afectará?


*Desarrolle una lista de "se necesita saber" para cada objeto de configuración y manténgala actualizada. Cuando se realice un cambio, asegúrese de notificar a todos a través de la lista.*

El flujo de información para el reporte del estado de la configuración (REC) se ilustra en la figura 22.5. Cada vez que se le asigna a un ICS una nueva identificación o que se le actualiza, se hace una entrada REC. Cada ocasión que el ACC aprueba un cambio (es decir, emite una OCI), se hace una entrada REC. Cada vez que se lleva a cabo una auditoría de la configuración, los resultados se reportan como parte de la tarea REC. La salida del REC puede colocarse en una base de datos en línea o en un sitio web, de modo que los desarrolladores de software o el personal de apoyo puedan acceder a la información del cambio mediante categorías de palabras clave. Además, regularmente se genera un reporte REC y se tiene la intención de mantener al tanto de los cambios importantes a los gerentes y profesionales.

**HERRAMIENTAS DE SOFTWARE**

#### *Apoyo a ACS*

**Objetivo:** Las herramientas ACS proporcionan apoyo a una o más de las actividades del proceso que se estudian en la sección 22.3.

**Mecánica:** La mayoría de las modernas herramientas ACS trabajan en conjunto con un repositorio (un sistema de base de datos) y ofrecen mecanismos para identificación, control de versión y de cambio, auditoría y reportes.

**Herramientas representativas:**[6]

*CCC/Harvest*, distribuida por Computer Associates (**www.cai.com**), es un sistema ACS multiplataforma.

*ClearCase*, desarrollada por Rational, proporciona una familia de funciones ACS (**www-306.ibm.com/software/awdtools/clearcase/index.html**).

*Serena ChangeMan ZMF*, distribuida por Serena (**www.serena.com/US/products/zmf/index.aspx**), proporciona un juego completo de herramientas ACS que son aplicables tanto para software convencional como para *webapps*.

*SourceForge*, distribuida por VA Software (**sourceforge.net**), ofrece administración de versión, capacidades de construcción, rastreo de conflictos/errores y muchas otras características de administración.

*SurroundSCM*, desarrollada por Seapine Software, proporciona capacidades completas de administración del cambio (**www.seapine.com**).

*Vesta*, distribuida por Compac, es un sistema ACS de dominio público que puede soportar tanto proyectos pequeños (< 10 KLOC) como grandes (10 000 KLOC) (**www.vestasys.org**).

Una gran lista de herramientas y entornos ACS comerciales puede encontrarse en **www.cmtoday.com/yp/commercial.html**.

## 22.4 ADMINISTRACIÓN DE LA CONFIGURACIÓN PARA *WEBAPPS*

Anteriormente, en este libro, se estudió la naturaleza especial de las *webapps* y los métodos especializados (llamados métodos de *ingeniería web*[7]) que se requieren para construirlas. Entre las muchas características que diferencian a las *webapps* del software tradicional se encuentra la naturaleza siempre presente del cambio.

**¿Qué impacto tiene sobre una *webapp* el cambio descontrolado?**

Los desarrolladores de *webapps* con frecuencia usan un modelo de proceso iterativo incremental que aplica muchos principios derivados del desarrollo de software ágil (capítulo 3). Al usar este método, un equipo de ingeniería con frecuencia desarrolla un incremento de *webapp* en un periodo muy corto, usando un enfoque impulsado por el cliente. Los incrementos posteriores agregan contenido y funcionalidad adicionales, y es probable que cada uno implemente

6 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

7 Vea [Pre08] para una discusión amplia de los métodos de ingeniería web.

cambios que conduzcan a aumento de contenido, mejor usabilidad, estética mejorada, mejor navegación, desempeño aumentado y seguridad más fuerte. Por tanto, en el mundo ágil de las *webapps*, el cambio se ve de manera un poco diferente.

Si usted es miembro de un equipo *webapp*, debe abrazar el cambio. Más aún, un equipo ágil típico se abstiene de todas las cosas que parecen ser pesadas, burocráticas y formales. La administración de la configuración del software con frecuencia se ve (aunque incorrectamente) como poseedora de estas características. Esta aparente contradicción se remedia no al rechazar los principios, prácticas y herramientas ACS, sino, más bien, al modelarlos para satisfacer las necesidades especiales de los proyectos *webapp*.

### 22.4.1 Conflictos dominantes

Conforme las *webapps* se vuelven cada vez más importantes para la supervivencia y el crecimiento empresarial, crece la necesidad de la administración de la configuración. ¿Por qué? Porque sin controles efectivos, los cambios inapropiados para una *webapp* (recuerde que la inmediatez y la evolución continua son los atributos dominantes de muchas *webapps*) pueden conducir a: publicación no autorizada de información de productos nuevos, funcionalidad errónea o pobremente probada que frustra a los visitantes de un sitio web, huecos en la seguridad que ponen en peligro los sistemas internos de la compañía y otras consecuencias económicamente desagradables o incluso desastrosas.

Las estrategias generales para la administración de la configuración del software (ACS) descritas en este capítulo son aplicables, pero las tácticas y las herramientas deben adaptarse para conformarse a la naturaleza única de las *webapps*. Cuando se desarrollan tácticas para la administración de la configuración de una *webapp*, deben considerarse cuatro conflictos [Dar99].

**Contenido.** Una *webapp* típica contiene un arreglo muy amplio de contenido: texto, gráficos, applets, guiones, archivos audio/video, elementos de página activos, tablas, transmisión de datos, y muchos otros. El reto es organizar este mar de contenido en un conjunto racional de objetos de configuración (sección 22.1.4) y luego establecer mecanismos de control de la configuración adecuados para dichos objetos. Un camino es modelar el contenido de la *webapp* usando técnicas de modelado de datos convencionales (capítulo 6), unidas a un conjunto de propiedades especializadas de cada objeto. La naturaleza estática/dinámica de cada objeto y su longevidad proyectada (objeto temporal, de existencia fija o permanente) son ejemplos de propiedades que se requieren para establecer un enfoque ACS efectivo. Por ejemplo, si un ítem de contenido cambia cada hora, tiene longevidad temporal. Los mecanismos de control para este ítem serán diferentes (menos formales) a los aplicados para un componente de formulario, que es un objeto permanente.

**Personas.** Puesto que un porcentaje significativo de desarrollo *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 antecedentes de ingeniería de software y son completamente ajenos a las necesidades de la administración de la configuración. Como consecuencia, la aplicación crece y cambia en forma descontrolada.

**Escalabilidad.** Las técnicas y los controles aplicados a una *webapp* pequeña no escalan bien hacia arriba. No es raro que una *webapp* simple crezca significativamente conforme se implementan interconexiones con sistemas de información, bases de datos, almacenes de datos y puertas a portales existentes. Conforme crecen el tamaño y la complejidad, pequeños cambios pueden tener efectos de largo alcance y no intencionados que pueden ser problemáticos. Por tanto, el rigor de los mecanismos de control de la configuración debe ser directamente proporcional a la escala de aplicación.

**Políticas.** ¿Quién "posee" la *webapp* ? Esta pregunta se plantea en compañías grandes y pequeñas, y su respuesta tiene un impacto significativo sobre las actividades de administración y control. En algunas instancias, los desarrolladores web se alojan fuera del área de IT de la organización y crean potenciales dificultades de comunicación. Dart [Dar99] sugiere las siguientes preguntas para ayudar a entender las políticas asociadas con la ingeniería web:

**¿Cómo se determina quién tiene la responsabilidad de la AC de la *webapp*?**

- ¿Quién asume la responsabilidad por la precisión de la información en el sitio web?
- ¿Quién garantiza que los procesos de control de calidad se siguieron antes de que la información se publique en el sitio?
- ¿Quién es responsable por la realización de 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 administración de la configuración para *webapps*.

La administración de la configuración para *webapps* continúa evolucionando (por ejemplo, [Ngu06]). Un proceso ACS convencional puede ser demasiado engorroso, pero en años pasados surgió una nueva generación de *herramientas de administración de contenido*, específicamente diseñadas para ingeniería web. Dichas herramientas establecen un proceso que adquiere información existente (de un amplio arreglo de objetos de *webapps*), gestiona los cambios a los objetos, los estructura en forma tal que es posible presentarlos a un usuario final y luego los ofrece al entorno del lado cliente para su despliegue.

### 22.4.2 Objetos de configuración de *webapps*

Las *webapps* abarcan un amplio rango de objetos de configuración: objetos de contenido (por ejemplo, texto, gráficos, imágenes, video, audio), componentes funcionales (guiones, applets) y objetos de interfaz (COM o CORBA). Los objetos de la *webapp* pueden identificarse (con nombres de archivo asignados) en cualquier forma que sea adecuada para la organización. Sin embargo, se recomiendan las siguientes convenciones para asegurar la conservación de la compatibilidad entre plataformas: los nombres de archivo deben limitarse a 32 caracteres de longitud, deben evitarse nombres con mayúsculas mezcladas o todos con mayúsculas, así como subrayar los nombres de los archivos. Además, las referencias URL (vínculos) dentro de un objeto de configuración siempre deben usar rutas relativas (por ejemplo,../products/alarmsensors.html).

Todo el contenido de la *webapp* tiene formato y estructura. Los formatos de archivo interno los dicta el entorno de computación en el que se almacena el contenido. Sin embargo, el *formato renderizado* (con frecuencia llamado *formato de despliegue*) se define mediante el estilo estético y las reglas de diseño establecidas para la *webapp*. La *estructura del contenido* define una arquitectura de contenido, es decir, la forma en la que se ensamblan los objetos de contenido para presentar información significativa a un usuario final. Boiko [Boi04] define la estructura como "mapas que se tienden sobre un conjunto de trozos de contenido [objetos] para organizarlos y hacerlos accesibles a las personas que los necesitan".

### 22.4.3 Administración de contenido

**Cita:**

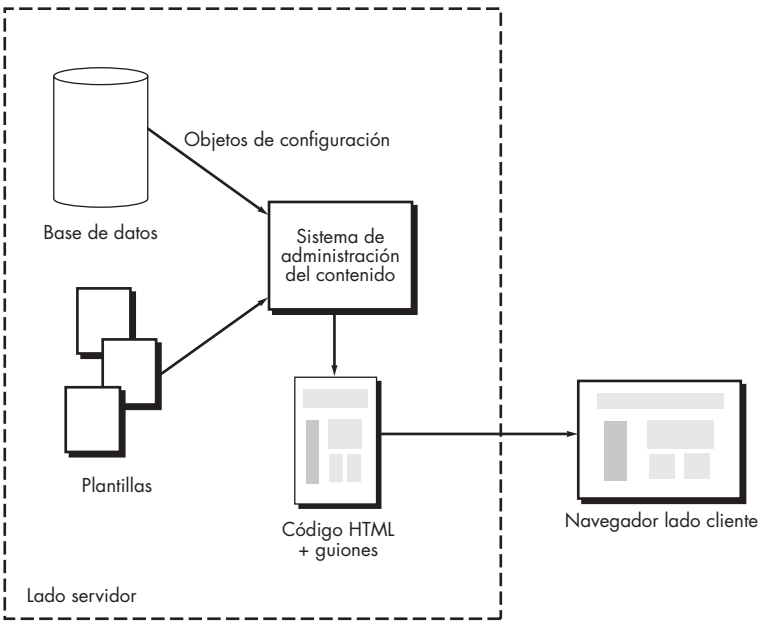
"La administración del contenido es un antídoto al frenesí informativo de la actualidad."

**Bob Boiko**

La *administración de contenido* se relaciona con la administración de la configuración en tanto un sistema de administración de contenido (SAC) establece un proceso, apoyado por herramientas adecuadas, en el que éstas adquieren contenido existente (de un amplio arreglo de objetos de configuración de la *webapp*), que los estructura en forma que les permite presentarse a un usuario final y luego ofrecerlo al entorno del lado cliente para su despliegue.

El uso más común de un sistema de administración de contenido ocurre cuando se construye una *webapp* dinámica. Las *webapps* dinámicas crean páginas web "al vuelo", es decir, el usuario

**FIGURA 22.6**
**Sistema de administración del contenido**




por lo general consulta la *webapp* para requerir 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 discos compactos para su venta. Cuando un usuario solicita un disco compacto o su equivalente en música electrónica, se consulta una base de datos y se descarga información variada acerca del artista, el disco compacto (por ejemplo, su portada o gráficos), el contenido musical y audio de muestra y se le configura en una plantilla de contenido estándar. La página web resultante se construye en el lado servidor y pasa al navegador en el lado cliente para su examen por parte del usuario final. En la figura 22.6 se muestra una representación genérica de esto.

En el sentido más general, un SAC "configura" el contenido para el usuario final al invocar tres subsistemas integrados: un subsistema de recopilación, un subsistema de gestión y un sistema de publicación [Boi04].

PUNTO CLAVE

El subsistema de recopilación abarca todas las acciones requeridas para crear, adquirir y/o convertir el contenido en una forma que pueda presentarse en el lado cliente.

**El subsistema de recopilación.** El contenido se deriva de datos e información que debe crearse o adquirirse por un desarrollador de contenido. El *subsistema de recopilación* abarca todas las acciones requeridas para crear y/o adquirir contenido y las funciones técnicas que se necesitan para 1) convertir el contenido en una forma que pueda representarse mediante un lenguaje de marcaje (por ejemplo, HTML, XML) y 2) organizar el contenido en paquetes que puedan desplegarse efectivamente en el lado cliente.

La creación y adquisición de contenido (con frecuencia llamado *autoría*) usualmente ocurre en paralelo con otras actividades de desarrollo de *webapps* y con frecuencia la realizan desarrolladores de contenido no técnicos. Esta actividad combina elementos de creatividad e investigación y se apoya con herramientas que permiten al autor del contenido caracterizar a éste en forma que puede estandarizarse para su uso dentro de la *webapp*.

Una vez que existe el contenido, debe convertirse para adecuarse a los requerimientos de un SAC. Esto implica desnudar el contenido bruto de cualquier información innecesaria (por ejemplo, representaciones gráficas redundantes), formatear el contenido para adecuarse a los requerimientos del SAC y mapear los resultados en una estructura de información que le permitirá manejarse y publicarse.

PUNTO CLAVE

El subsistema de administración implementa un repositorio para todo el contenido. La administración de la configuración se realiza dentro de este subsistema.

**El subsistema de administración.** Una vez que existe el contenido, debe almacenarse en un repositorio, catalogarse para adquisición y uso posterior y etiquetarse para definir: 1) el estado actual (por ejemplo, ¿el objeto de contenido está completo o en desarrollo?), 2) la versión adecuada del objeto de contenido y 3) los objetos de contenido relacionados. Por tanto, el *subsistema de administración* implementa un repositorio que abarca los siguientes elementos:

- *Base de datos de contenido:* la estructura de información que se establece para almacenar todos los objetos de contenido
- *Capacidades de la base de datos:* funciones que permiten al SAC buscar objetos de contenido específicos (o categorías de objetos), almacenar y recuperar objetos, y administrar la estructura de archivos que se establece para el contenido
- *Funciones de administración de configuración:* los elementos funcionales y el flujo de trabajo asociado que apoyan la identificación del objeto de contenido, el control de la versión, la administración del cambio, la auditoría del cambio y los reportes.

Además de estos elementos, el subsistema de administración implementa una función de administración que abarca los metadatos y las reglas que controlan la estructura global del contenido y la forma en la que se soporta.

PUNTO CLAVE

El subsistema de publicación extrae el contenido del repositorio y lo entrega a navegadores en el lado cliente.

**El subsistema de publicación.** El contenido debe extraerse del repositorio, convertirse a una forma que sea manejable para su publicación y formatearse de modo que pueda transmitirse a navegadores en el lado cliente. El subsistema de publicación logra estas tareas usando una serie de plantillas. Cada *plantilla* es una función que construye una publicación usando uno de tres componentes diferentes [Boi04]:

- *Elementos estáticos:* texto, gráficos, medios audiovisuales y guiones que no requieren más procesamiento se transmiten directamente al lado cliente.
- *Servicios de publicación:* la función invoca servicios de recuperación y formateo específicos que personalizan el contenido (usando reglas predefinidas), realizan conversión de datos y construyen vínculos de navegación adecuados.
- *Servicios externos:* acceso a infraestructura de información corporativa externa, como datos empresariales o aplicaciones de "cuarto trasero".

Un sistema de administración del contenido que abarque cada uno de estos subsistemas es aplicable para grandes proyectos web. Sin embargo, la filosofía y funcionalidad básicas asociadas con un SAC son aplicables a todas las *webapps* dinámicas.

HERRAMIENTAS DE SOFTWARE

### *Administración de contenido*

**Objetivo:** Auxiliar a los ingenieros de software y desarrolladores de contenido en la administración del contenido que se incorpora en las *webapps*.

**Mecánica:** Las herramientas en esta categoría permiten a los ingenieros web y proveedores de contenido actualizar el contenido *webapp* en forma controlada. La mayoría establece un sistema de gestión de archivos simple que asigna permisos de actualización y edición página por página para varios tipos de contenido *webapp*. Algunos mantienen un sistema de versiones, de modo que una versión previa de contenido pueda archivarse con propósitos históricos.

**Herramientas representativas:**[8]

*Vignette Content Management*, desarrollado por Vignette (**www.vignette.com/us/Products**), es una suite de herramientas de administración de contenido empresarial.

8 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

*ektron-CMS300*, desarrollada por ektron (**www.ektron.com**), es una suite de herramientas que proporciona capacidades de administración de contenido y herramientas de desarrollo web.
*OmniUpdate*, desarrollada por WebsiteASP, Inc. (**www.omniupdate.com**), es una herramienta que permite a proveedores de contenido autorizados desarrollar actualizaciones controladas a contenido web específico.

En los siguientes sitios web puede encontrarse información adicional acerca de ACS y herramientas de administración de 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**).

### 22.4.4 Administración del cambio

El flujo de trabajo asociado con el control del cambio para software convencional (sección 22.3.3) por lo general es muy pesado para el desarrollo de *webapps*. Es improbable que la secuencia petición de cambio, reporte de cambio y orden de cambio de ingeniería pueda lograrse en forma ágil y aceptable para la mayoría de los proyectos de desarrollo web. Entonces, ¿cómo se gestiona un torrente continuo de cambios solicitados por el contenido y la funcionalidad de la *webapp*?

Para implementar administración de cambio efectiva dentro de la filosofía "codifica y ve" que continúa dominando el desarrollo web debe modificarse el proceso de control de cambio convencional. Cada cambio debe categorizarse en una de cuatro clases:

*Clase 1:* un cambio de contenido o función que corrige un error o aumenta el contenido o funcionalidad locales

*Clase 2:* un cambio de contenido o función que tiene un impacto sobre otros objetos de contenido o componentes funcionales

*Clase 3:* un cambio de contenido o función que tiene un amplio impacto a través de una *webapp* (por ejemplo, extensión de funcionalidad trascendental, significativo aumento o reducción en contenido, grandes cambios requeridos en navegación)

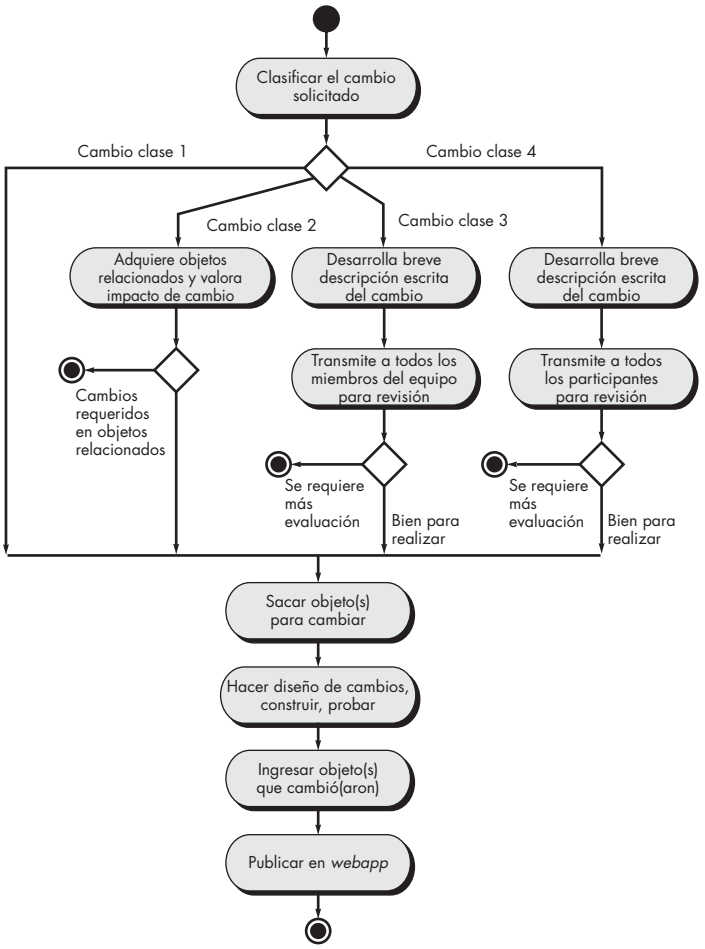
*Clase 4:* un gran cambio de diseño (por ejemplo, un cambio en diseño de interfaz o enfoque de navegación) que inmediatamente será notable para una o más categorías de usuario

Una vez categorizado el cambio solicitado, puede procesarse en concordancia con el algoritmo que se muestra en la figura 22.7.

En la figura, los cambios en las clases 1 y 2 se tratan de manera informal y se manejan en forma ágil. Para un cambio de clase 1, se evaluaría el impacto del cambio, pero no se requiere revisión o documentación externa. Conforme se realiza el cambio, los procedimientos de entrada y salida estándar se refuerzan mediante herramientas de configuración de repositorio. Para los cambios de clase 2, se revisa el impacto del cambio sobre objetos relacionados (o se pide hacerlo a otros desarrolladores responsables de dichos objetos). Si el el cambio puede hacerse sin requerir cambios significativos a otros objetos, la modificación ocurre sin revisión o documentación adicional. Si se requieren cambios sustantivos, es necesario más evaluación y planificación.

Los cambios de clases 3 y 4 también se tratan en forma ágil, pero se requiere alguna documentación descriptiva y más procedimientos de revisión formal. Para los cambios de clase 3, se desarrolla una *descripción del cambio* que describe el cambio y proporciona una breve valoración del impacto del mismo. La descripción se distribuye a todos los miembros del equipo, quienes revisan el cambio para valorar mejor su impacto. Para los cambios de clase 4, también se desarrolla una descripción del cambio, pero en este caso la realizan todos los participantes.

**FIGURA 22.7**
**Administración de cambios para *webapps***




**HERRAMIENTAS DE SOFTWARE**

### *Gestión del cambio*

**Objetivo:** Auxiliar a los ingenieros web y a desarrolladores de contenido a administrar los cambios en objetos de configuración *webapp* conforme estos se realizan.

**Mecánica:** En esta categoría, las herramientas se desarrollaron originalmente para software convencional, pero pueden adaptarse para su uso por parte de ingenieros web y desarrolladores de contenido a fin de realizar cambios controlados a *webapps*. Soportan entrada y salida automatizadas, control y reconstrucción de versiones, reporte y otras funciones ACS.

**Herramientas representativas:**[9]

*ChangeMan WCM*, desarrollada por Serena (**www.serena.com**), es una suite de herramientas de administración del cambio que proporcionan capacidades ACS completas.

*ClearCase*, desarrollada por Rational (**www-306.ibm.com/software/rational/sw-atoz/indexC.html**), es una suite de herramientas que proporciona todas las capacidades de administración de configuración para *webapps*.

*Source Integrity*, desarrollada por mks (**www.mks.com**), es una herramienta ACS que puede integrarse con entornos de desarrollo seleccionados.

9 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

### 22.4.5 Control de versión

Conforme una *webapp* evoluciona a través de una serie de incrementos, pueden existir al mismo tiempo algunas versiones diferentes. Una versión (la *webapp* operativa actual) está disponible mediante internet para usuarios finales; otra (el siguiente incremento de *webapp*) puede estar en las etapas finales de prueba antes de su implementación; una tercera versión está en desarrollo y representa una gran actualización en contenido, estética de interfaz y funcionalidad. Los objetos de configuración deben definirse con claridad, de modo que cada uno pueda asociarse con la versión adecuada. Además, deben establecerse mecanismos de control. Dreilinger [Dre99] analiza la importancia del control de versiones (y de cambios) cuando escribe:

> En un sitio *descontrolado*, donde múltiples autores tienen acceso para editar y contribuir, surge el potencial para conflictos y problemas, más aún si dichos autores trabajan desde diferentes oficinas en diferentes momentos del día y de la noche. Usted puede pasar el día mejorando el archivo *index.html* para un cliente. Después de que usted realiza sus cambios, otro desarrollador que trabaja en casa después de las horas de oficina, o en otra oficina, puede pasar la noche actualizando su propia versión recientemente revisada del archivo *index.html*, ¡y sobreescribir por completo su trabajo sin que haya forma de recuperarlo!

Es probable que usted haya experimentado una situación similar. Para evitarla, se requiere un proceso de control de versiones.

1. *Debe establecerse un repositorio central para el proyecto web*, que contendrá las versiones actuales de todos los objetos de configuración *webapp* (contenido, componentes funcionales y otros).
2. *Cada ingeniero web crea su propia carpeta de trabajo*, que contiene aquellos objetos que se crean o cambian en cualquier momento.
3. *Los relojes en todas las estaciones de trabajo de los desarrolladores deben estar sincronizados* para evitar conflictos de sobreescritura cuando dos desarrolladores realizan actualizaciones que están muy cercanas en el tiempo.
4. *Conforme se desarrollan nuevos objetos de configuración o se cambian los objetos existentes, deben importarse hacia el repositorio central.* La herramienta de control de versión (vea la discusión de SVC en la barra lateral) gestionará todas las funciones de entrada y salida de las carpetas de trabajo de cada desarrollador web. Cuando se hagan cambios al repositorio, la herramienta también proporcionará actualizaciones automáticas por correo electrónico a todas las partes interesadas.
5. *Conforme los objetos se importen al o se exporten del repositorio, se elabora un mensaje de bitácora automático con marca de tiempo.* Esto proporciona información útil para auditar y puede volverse parte de un esquema de reporte efectivo.

La herramienta de control de versión conserva diferentes versiones de la *webapp* y puede revertir una de ellas a una versión más antigua si se requiere.

### 22.4.6 Auditoría y reporte

Con la intención de obtener agilidad, las funciones de auditoría y reporte no tienen mucho énfasis en el trabajo de ingeniería web.[10] Sin embargo, no se eliminan por completo. Todos los objetos que entran o salen del repositorio se registran en una bitácora que puede revisarse en

10 Esto empieza a cambiar. Hay un creciente énfasis en la ACS como un elemento de la seguridad de la *webapp* [Sar06]. Al proporcionar un mecanismo para rastrear y reportar todo cambio hecho a cada objeto Web, una herramienta de administración del cambio puede proporcionar valiosa protección contra cambios maliciosos.

cualquier momento. Es posible crear un reporte de bitácora completo de modo que todos los miembros del equipo web tengan una cronología de los cambios hechos en un periodo definido. Además, puede enviarse una notificación automatizada por correo electrónico (dirigida a aquellos desarrolladores y participantes que tengan interés) cada vez que un objeto entre o salga del repositorio.

**INFORMACIÓN**

### *Estándares ACS*

La siguiente lista de estándares ACS (extraída en parte de www.12207.com) es razonablemente exhaustiva:

| | |
|---|---|
| **Estándares IEEE** | **standards.ieee.org/catalog/olis/** |
| IEEE 828 | Software para planes de administración de la configuración |
| IEEE 1042 | Software para administración de la configuración |
| **Estándares ISO** | **www.iso.ch/iso/en/ISOOnline.frontpage** |
| ISO 10007-1995 | Administración de calidad, guía para AC |
| ISO/IEC 12207 | Software de tecnología de la información-Procesos de ciclo de vida |
| ISO/IEC TR 15271 | Guía para ISO/IEC 12207 |
| ISO/IEC TR 15846 | Ingeniería de software-Proceso de ciclo de vida de software-Orden de software para administración de la configuración |
| **Estándares EIA** | **www.eia.org/** |
| EIA 649 | Estándar de consenso nacional para administración de la configuración |
| EIA CMB4-1A | Definiciones de administración de la configuración para programas de cómputo digitales |
| EIA CMB4-2 | Identificación de configuración para programas de cómputo digitales |
| EIA CMB4-3 | Librerías de software de cómputo |
| EIA CMB4-4 | Control de cambio de configuración para programas de cómputo digitales |
| EIA CMB6-1C | Orden de referencias de administración de configuración y datos |
| EIA CMB6-3 | Identificación de configuración |
| EIA CMB6-4 | Control de la configuración |
| EIA CMB6-5 | Libro de texto para contabilidad de estado de la configuración |
| EIA CMB7-1 | Intercambio electrónico de datos de administración de la configuración |
| **Estándares militares EUA** | **Información de estándares MIL: www-library.itsi.disa.mil** |
| DoD MIL STD-973 | Administración de la configuración |
| MIL-HDBK-61 | Guía para administración de la configuración |
| **Otros estándares:** | |
| DO-178B | Lineamientos para el desarrollo de software de aviación |
| ESA PSS-05-09 | Guía para administración de la configuración del software |
| AECL CE-1001-STD rev.1 | Estándar para ingeniería de software crucial para seguridad |
| DOE SCM checklist: | http://cio.doe.gov/ITReform/sqse/download/cmcklst.doc |
| BS-6488 | British Std., Administración de la configuración de sistemas basados en computadoras |
| Best Practice—UK | Oficina de comercio gubernamental: **www.ogc.gov.uk** |
| CMII | Instituto de mejores prácticas AC: **www.icmhq.com** |

*Configuration Management Resource Guide* (*Guía de recursos de administración de la configuración*) proporciona información complementaria para los interesados en procesos y práctica AC. Está disponible en **www.quality.org/config/cm-guide.html**.

## 22.5 RESUMEN

La administración de la configuración del software es una actividad sombrilla que se aplica a lo largo del proceso de software. La ACS identifica, controla, audita y reporta las modificaciones que invariablemente ocurren mientras el software se desarrolla y después de que se libera a un cliente. Todos los productos de trabajo creados como fases de la ingeniería de software se vuelven parte de una configuración del software. La configuración se organiza de manera que permite el control ordenado del cambio.

La configuración del software se compone de un conjunto de objetos interrelacionados, también llamados ítems de configuración del software (ICS), que se producen como resultado de alguna actividad de ingeniería de software. Además de documentos, programas y datos, el en-

torno de desarrollo que se usa para crear software también puede colocarse bajo control de la configuración. Todos los ICS se almacenan dentro de un repositorio que implementa un conjunto de mecanismos y estructuras de datos para asegurar la integridad de los datos, proporcionar apoyo de integración para otras herramientas de software (información de apoyo que comparte entre todos los miembros del equipo de software) e implementar funciones de apoyo al control de versiones y de cambios.

Una vez desarrollado y revisado el objeto de configuración, se convierte en línea de referencia. Los cambios a un objeto convertido en línea de referencia dan como resultado la creación de una nueva versión de dicho objeto. La evolución de un programa puede rastrearse al examinar la historia de revisión de todos los objetos de configuración. El control de versiones es el conjunto de procedimientos y herramientas que sirven para administrar el uso de dichos objetos.

El control de cambios es una actividad procedimental que garantiza la calidad y la consistencia conforme se realizan cambios a un objeto de configuración. El proceso de control de cambios comienza con una petición de cambios, conduce a una decisión para hacer o rechazar la petición del cambio y culmina con una actualización controlada del ICS que debe cambiarse.

La auditoría de la configuración es una actividad SQA que ayuda a garantizar que la calidad se conserva conforme se realizan cambios. El reporte de estado proporciona información acerca de cada cambio a quienes necesitan conocerla.

La administración de la configuración para *webapps* es similar en muchos aspectos a la ACS para software convencional. Sin embargo, cada una de las tareas núcleo ACS debe dinamizarse para hacerla tan magra como sea posible y deben implementarse provisiones especiales para la administración del contenido.

## Problemas y puntos por evaluar

**22.1.** ¿Por qué es verdadera la primera ley de la ingeniería de sistemas? Ofrezca ejemplos específicos para cada una de las cuatro razones fundamentales para el cambio.

**22.2.** ¿Cuáles son los cuatro elementos que existen cuando se implementa un sistema ACS efectivo? Analice cada uno brevemente.

**22.3.** Explique con sus palabras las razones para las líneas de referencia.

**22.4.** Suponga que usted es el gerente de un proyecto pequeño. ¿Qué líneas de referencia definiría para el proyecto y cómo las controlaría?

**22.5.** Diseñe un sistema de base de datos de proyecto (repositorio) que permitiría a un ingeniero del software almacenar, poner referencias cruzadas, rastrear, actualizar y cambiar todos los ítems de configuración de software importantes. ¿Cómo manejaría la base de datos diferentes versiones del mismo programa? ¿El código fuente se manejaría de manera diferente a la documentación? ¿Cómo se prohibiría a dos desarrolladores hacer diferentes cambios al mismo tiempo a un mismo ICS?

**22.6.** Investigue una herramienta ACS existente y describa cómo implementa control para versiones, variantes y objetos de configuración en general.

**22.7.** Las relaciones <parte de> e <interrelacionado> representan relaciones simples entre objetos de configuración. Describa cinco relaciones adicionales que puedan ser útiles en el contexto de un repositorio ACS.

**22.8.** Investigue acerca de una herramienta ACS existente y describa cómo implementa la mecánica de control de versiones. De manera alternativa, lea dos o tres ensayos acerca de ACS y describa las diferentes estructuras de datos y mecanismos de referencia que se usan para el control de versiones.

**22.9.** Desarrolle una lista de verificación para usar durante las auditorías de configuración.

**22.10.** ¿Cuál es la diferencia entre una auditoría ACS y una revisión técnica? ¿Su función puede plegarse en una revisión? ¿Cuáles son los pros y los contras?

**22.11.** Describa brevemente las diferencias entre ACS para software convencional y ACS para *webapps*.

**22.12.** ¿Qué es la administración del contenido? Use la web para investigar las características de una herramienta de administración del contenido y ofrezca un resumen breve.

## Lecturas adicionales y fuentes de información

Entre las propuestas de lecturas sobre ACS más recientes se encuentran Leon (*Software Configuration Management Handbook,* 2a. ed., Artech House Publishers, 2005), Maraia (*The Build Master: Microsoft's Software Configuration Management Best Practices,* Addison-Wesley, 2005), Keyes (*Software Configuration Management,* Auerbach, 2004) y Hass (*Configuration Management Principles and Practice,* Addison-Wesley, 2002). Cada uno de estos libros presenta todo el proceso ACS con detalle sustancial. Maraia (*Software Configuration Management Implementation Roadmap*, Wiley, 2004) ofrece una guía única de "cómo hacer" para quienes deben implementar ACS dentro de una organización. Lyon (*Practical CM*, Raven Publishing, 2003, disponible en **www.configuration.org**) escribió una guía exhaustiva para el profesional AC, que incluye lineamientos pragmáticos para implementar cada aspecto de un sistema de administración de la configuración (se actualiza anualmente). White y Clemm (*Software Configuration Management Strategies and Rational ClearCase*, Addison-Wesley, 2000) presentan ACS dentro del contexto de una de las herramientas ACS más populares.

Berczuk y Appleton (*Software Configuration Management Patterns*, Addison-Wesley, 2002) proponen varios patrones útiles que ayudan a comprender la ACS y a implementar de manera efectiva sistemas ACS. 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 ACS y luego consideran sus remedios. Bays (*Software Release Methodology*, Prentice Hall, 1999) se enfoca en la mecánica de "liberación exitosa de producto", un importante complemento a la ACS efectiva.

Conforme las *webapps* se vuelven más dinámicas, la administración del contenido se ha convertido en un tema esencial para los ingenieros web. Los libros de White (*The Content Management Handbook,* Curtin University Books, 2005), Jenkins *et al*. (*Enterprise Content Management Methods,* Open Text Corporation, 2005), Boiko [Boi04], Mauthe y Thomas (*Professional Content Management Systems,* Wiley, 2004), Addey *et al*. (*Content Management Systems,* Glasshaus, 2003), Rockley (*Managing Enterprise Content,* New Riders Press, 2002), Hackos (*Content Management for Dynamic Web Delivery,* Wiley, 2002), y Nakano (*Web Content Management,* Addison-Wesley, 2001) presentan tratamientos valiosos del tema.

Además de discusiones genéricas del tema, Lim *et al*. (*Enhancing Microsoft Content Management Server with ASP.NET 2.0,* Packt Publishing, 2006), Ferguson (*Creating Content Management Systems in Java,* Charles River Media, 2006), IBM Redbooks (*IBM Workplace Web Content Management for Portal 5.1 and IBM Workplace Web Content Management 2.5,* Vivante, 2006), Fritz *et al*. (*Typo3: Enterprise Content Management,* Packt Publishing, 2005) y Forta (*Reality ColdFusion: Intranets and Content Management,* Pearson Education, 2002) abordan la administración del contenido dentro del contexto de herramientas y lenguajes específicos.

En internet está disponible una gran variedad de fuentes de información acerca de ingeniería de administración de la configuración del software y de administración del contenido. Una lista actualizada de referencias en la World Wide Web que son relevantes para la administración de la configuración del software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compICS/pressman/professional/olc/ser.htm**.

CAPÍTULO

# 23 MÉTRICAS DE PRODUCTO

Conceptos clave



Un elemento clave de cualquier proceso de ingeniería es la medición. Pueden usarse medidas para entender mejor los atributos de los modelos que se crean y para valorar la calidad de los productos o sistemas sometidos a ingeniería que se construyen. Pero, a diferencia de otras disciplinas de la ingeniería, la del software no está asentada en las leyes cuantitativas de la física. Mediciones directas, como voltaje, masa, velocidad o temperatura, son raras en el mundo del software. Puesto que las mediciones y métricas del software con frecuencia son indirectas, están abiertas a debate. Fenton [Fen91] aborda este conflicto cuando afirma:

> Medición es el proceso mediante el cual se asignan números o símbolos a los atributos de las entidades en el mundo real, de manera que se les define de acuerdo con reglas claramente determinadas [...] En ciencias físicas, medicina, economía y más recientemente en ciencias sociales, ahora es posible medir atributos que anteriormente se consideraban inmensurables [...] Desde luego, tales mediciones no son tan refinadas como muchas mediciones en las ciencias físicas [...], pero existen [y con base en ellas se toman decisiones importantes]. Sentimos que la obligación de intentar "medir lo inmensurable" para mejorar la comprensión de entidades particulares es tan poderosa en la ingeniería del software como en cualquiera otra disciplina.

Pero algunos miembros de la comunidad del software continúan argumentando que el software es "inmensurable" o que los intentos por medir deben posponerse hasta comprender mejor el software y los atributos que deben usarse para describirlo. Esto es un error.

## Una mirada rápida

**¿Qué es?** Por su naturaleza, la ingeniería es una disciplina cuantitativa. Las métricas de producto ayudan a los ingenieros de software a obtener comprensión acerca del diseño y la construcción del software que elaboran, al enfocarse en atributos mensurables específicos de los productos de trabajo de la ingeniería del software.

**¿Quién lo hace?** Los ingenieros de software usan métricas de proyecto para auxiliarse en la construcción de software de mayor calidad.

**¿Por qué es importante?** Siempre habrá un elemento cualitativo en la creación del software para computadoras. El problema es que la valoración cualitativa tal vez no sea suficiente. Se necesitan criterios objetivos que ayuden a guiar el diseño de datos, arquitectura, interfaces y componentes. Cuando se prueban, es necesaria la guía cuantitativa que ayuda en la selección de los casos de prueba y de sus objetivos. Las métricas de producto proporcionan una base desde donde el análisis, el diseño, la codificación y las pruebas pueden realizarse de manera más objetiva y valorarse de modo más cuantitativo.

**¿Cuáles son los pasos?** El primer paso en el proceso de medición es derivar las mediciones y métricas del software que sean adecuadas para la presentación del software que se está construyendo. A continuación, se recolectan los datos requeridos para derivar las métricas formuladas. Una vez calculadas, las métricas adecuadas se analizan con base en lineamientos preestablecidos y en datos anteriores. Los resultados del análisis se interpretan para obtener comprensión acerca de la calidad del software, y los resultados de la interpretación conducen a modificación de requerimientos y modelos de diseño, código fuente o casos de prueba. En algunas instancias, también puede conducir a modificación del proceso de software en sí.

**¿Cuál es el producto final?** Las métricas de producto que se calculan a partir de los datos recolectados de los modelos de requerimientos y de diseño, código fuente y casos de prueba.

**¿Cómo me aseguro de que lo hice bien?** Debe establecer los objetivos de la medición antes de comenzar la recolección de datos y debe definir cada métrica de producto sin ambigüedades. Defina sólo algunas métricas y luego úselas para obtener comprensión acerca de la calidad de un producto de trabajo de ingeniería del software.

Aunque las métricas de producto para el software de computadora son imperfectas, pueden proporcionar una forma sistemática de valorar la calidad con base en un conjunto de reglas claramente definidas. También proporcionan comprensión inmediata, en lugar de hacerlo después de los hechos. Esto permite descubrir y corregir potenciales problemas antes de que se conviertan en defectos catastróficos.

En este capítulo se presentan las mediciones que pueden usarse para valorar la calidad del producto conforme se somete a ingeniería. Estas mediciones de atributos internos del producto ofrecen una indicación en tiempo real de la eficacia de los modelos de requerimientos, diseño y código, así como de la efectividad de los casos de prueba y de la calidad global del software que se va a construir.

## 23.1 Marco conceptual para las métricas de producto

Como se anotó en la introducción, la medición asigna números o símbolos a atributos de entidades en el mundo real. Para lograr esto, se 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, [Zus97]) son temas que están más allá del ámbito de este libro, vale la pena establecer un marco conceptual fundamental y un conjunto de principios básicos que guíen la definición de las métricas de producto para el software.

**Cita:**

"Una ciencia es tan madura como sus herramientas de medición."

**Louis Pasteur**

### 23.1.1 Medidas, métricas e indicadores

Aunque los términos *medida, medición* y *métrica* con frecuencia se usan de modo intercambiable, es importante observar las sutiles diferencias entre ellos. En el contexto de la ingeniería del software, una *medida* proporciona un indicio cuantitativo de la extensión, cantidad, dimensión, capacidad o tamaño de algún atributo de un producto o proceso. La *medición* es el acto de determinar una medida. El *IEEE Standard Glosary of Software Engineering Terminology* [IEE93b] define *métrica* como "una medida cuantitativa del grado en el que un sistema, componente o proceso posee un atributo determinado".

**¿Cuál es la diferencia entre una medida y una métrica?**

Cuando se ha recolectado un solo punto de datos (por ejemplo, el número de errores descubiertos dentro de un solo componente de software), se establece una medida. La medición ocurre como resultado de la recolección de uno o más puntos de datos (por ejemplo, algunas revisiones de componente y pruebas de unidad se investigan para recolectar medidas del número de errores de cada uno). Una métrica de software relaciona en alguna forma las medidas individuales (por ejemplo, el número promedio de errores que se encuentran por revisión o el número promedio de errores que se encuentran por unidad de prueba).

**PUNTO CLAVE**

Un indicador es una o varias métricas que proporcionan comprensión acerca del proceso, el producto o el proyecto.

Un ingeniero de software recolecta medidas y desarrolla métricas de modo que se obtengan indicadores. Un *indicador* es una métrica o combinación de métricas que proporcionan comprensión acerca del proceso de software, el proyecto de software o el producto en sí. Un indicador proporciona comprensión que permite al gerente de proyecto o a los ingenieros de software ajustar el proceso, el proyecto o el producto para hacer mejor las cosas.

### 23.1.2 El reto de la métrica de producto

Durante las cuatro décadas pasadas, muchos investigadores intentaron desarrollar una sola métrica que proporcionara una medida abarcadora de la complejidad del software. Fenton [Fen94] caracteriza esta investigación como una búsqueda del "imposible Santo Grial". Aunque se han propuesto decenas de medidas de complejidad [Zus90], cada una toma una visión un poco diferente de lo que es la complejidad y de qué atributos de un sistema conducen a la complejidad. Por analogía, considere una métrica para evaluar un automóvil atractivo. Algunos observadores pueden enfatizar el diseño de la carrocería; otros pueden considerar característi-

**Cita:**

"Así como la medición de temperatura comenzó con un dedo índice... y creció hasta escalas, herramientas y técnicas sofisticadas, de igual modo madura la medición del software."

**Shari Pfleeger**

cas mecánicas; otros más pueden destacar el costo o el rendimiento o el uso de combustibles alternativos o la capacidad para reciclar cuando el carro sea chatarra. Dado que cualquiera de estas características puede estar en desacuerdo con otras, es difícil derivar un solo valor para el "atractivo". El mismo problema ocurre con el software de computadora.

Aunque existe la necesidad de medir y de controlar la complejidad del software, y si bien un solo valor de esta métrica de calidad es difícil de derivar, es posible desarrollar medidas de diferentes atributos internos de programa (por ejemplo, modularidad efectiva, independencia funcional y otros atributos estudiados en el capítulo 8). Estas medidas y las métricas derivadas de ellas pueden usarse como indicadores independientes de la calidad de los modelos de requerimientos y de diseño. Pero aquí de nuevo surgen problemas. Fenton [Fen94] observa esto cuando afirma: "el peligro de intentar encontrar medidas que caracterizan tantos atributos diferentes es que, inevitablemente, las medidas tienen que satisfacer objetivos en conflicto. Esto es contrario a la teoría representacional de las mediciones". Aunque el enunciado de Fenton es correcto, muchas personas argumentan que la medición de producto realizada durante las primeras etapas del proceso de software brinda a los ingenieros de software un mecanismo consistente y objetivo para valorar la calidad.

**WebRef**

Horst Zuse compiló voluminosa información acerca de la métrica de producto en **irb.cs.tu-berlin.de/~zuse/**

Sin embargo, es justo preguntar cuán válidas son las métricas de producto, es decir, ¿cuán cercanamente se alinean las métricas de producto con la confiabilidad a largo plazo y con la calidad de un sistema basado en computadora? Fenton [Fen91] aborda esta pregunta en la forma siguiente:

> A pesar de las conexiones intuitivas entre la estructura interna de los productos de software [métricas de producto] y sus atributos externos de producto y proceso, en realidad ha habido pocos intentos científicos para establecer relaciones específicas. Existen algunas razones por las que esto es así; la más comúnmente citada es lo impráctico que resulta realizar experimentos relevantes.

Cada uno de los "retos" anotados aquí es un motivo de precaución, pero no es razón para desechar las métricas de producto.[1] La medición es esencial si debe lograrse la calidad.

### 23.1.3 Principios de medición

Antes de presentar una serie de métricas de producto que 1) auxilien en la evaluación de los modelos de análisis y diseño, 2) proporcionen un indicio de la complejidad de los diseños procedimentales y del código fuente y 3) faciliten el diseño de pruebas más efectivas, es importante comprender los principios de medición básicos. Roche [Roc94] sugiere un proceso de medición que puede caracterizarse mediante cinco actividades:

**? ¿Cuáles son los pasos de un proceso de medición efectivo?**

- *Formulación.* La derivación de medidas y métricas de software apropiadas para la representación del software que se está construyendo.
- *Recolección.* Mecanismo que se usa para acumular datos requeridos para derivar las métricas formuladas.
- *Análisis.* El cálculo de métricas y la aplicación de herramientas matemáticas.
- *Interpretación.* Evaluación de las métricas resultantes para comprender la calidad de la representación.
- *Retroalimentación.* Recomendaciones derivadas de la interpretación de las métricas del producto, transmitidas al equipo de software.

1 Aunque las críticas de métricas específicas son comunes en la literatura, muchos críticos se enfocan en conflictos particulares y pierden el objetivo principal de las métricas en el mundo real: ayudar al ingeniero de software a establecer una vía sistemática y objetiva para obtener comprensión de su trabajo y mejorar la calidad del producto resultante.

Las métricas de software serán útiles sólo si se caracterizan efectivamente y si se validan de manera adecuada. Los siguientes principios [Let03b] son representativos de muchos que pueden proponerse para la caracterización y validación de métricas:


*En realidad, muchas métricas de producto actualmente en uso no concuerdan con dichos principios tanto como debieran. Pero eso no significa que no tengan valor; sólo tenga cuidado cuando los use y entienda que tienen la intención de proporcionar comprensión, no estricta verificación científica.*

- *Una métrica debe tener propiedades matemáticas deseables,* es decir, el valor de la métrica debe estar en un rango significativo (por ejemplo, 0 a 1, donde 0 realmente significa ausencia, 1 indica el valor máximo y 0.5 representa el "punto medio"). Además, una métrica que intente estar en una escala racional no debe constituirse con componentes que sólo se miden en una escala ordinal.
- *Cuando una métrica representa una característica de software que aumenta cuando ocurren rasgos positivos o que disminuye cuando se encuentran rasgos indeseables, el valor de la métrica debe aumentar o disminuir en la misma forma.*
- *Cada métrica debe validarse de manera empírica en una gran variedad de contextos antes de publicarse o utilizarse para tomar decisiones.* Una métrica debe medir el factor de interés, independientemente de otros factores. Debe "escalar" a sistemas más grandes y funcionar en varios lenguajes de programación y dominios de sistema.

Aunque la formulación, caracterización y validación son cruciales, la recolección y el análisis son las actividades que impulsan el proceso de medición. Roche [Roc94] sugiere los siguientes principios para dichas actividades: 1) siempre que sea posible, la recolección y el análisis de datos deben automatizarse; 2) deben aplicarse técnicas estadísticas válidas para establecer relaciones entre atributos de producto internos y características de calidad externas (por ejemplo, si el nivel de complejidad arquitectónica se correlaciona con el número de defectos reportados en el uso de producción), y 3) para cada métrica deben establecerse lineamientos y recomendaciones interpretativos.

### 23.1.4 Medición de software orientado a meta

**WebRef**

En **www.thedacts.com/GoldPractices/practices/gqma.html** puede encontrar un útil foro de discusión sobre el MPM.

El paradigma *Meta/Pregunta/Métrica* (MPM) fue desarrollado por Basili y Weiss [Bas84] como una técnica para identificar métricas significativas para cualquier parte del proceso de software. MPM enfatiza la necesidad de: 1) establecer una *meta* de medición explícita que sea específica para la actividad del proceso o para la característica del producto que se quiera valorar, 2) definir un conjunto de *preguntas* que deban responderse con la finalidad de lograr la meta y 3) identificar *métricas* bien formuladas que ayuden a responder dichas preguntas.

Para definir cada meta de medición, puede usarse una *plantilla de definición de meta* [Bas94]. La plantilla toma la forma:

> **Analizar** {el nombre de la actividad o atributo que se va a medir} **con el propósito de** {el objetivo global del análisis[2]} **con respecto a** {el aspecto de la actividad o atributo que se considera} **desde el punto de vista de** {las personas que tienen interés en la medición} **en el contexto de** {el entorno en el que tiene lugar la medición}.

Como ejemplo, considere una plantilla de definición de meta para *CasaSegura*:

> **Analizar** la arquitectura del software *CasaSegura* **con el propósito de** evaluar los componentes arquitectónicos **con respecto a** la capacidad de hacer *CasaSegura* más extensible **desde el punto de vista de** los ingenieros de software que realizan el trabajo **en el contexto de** mejora del producto durante los próximos tres años.

2 Van Solingen y Berghout [Sol90] sugieren que el objeto casi siempre es "comprender, controlar o mejorar" la actividad del proceso o el atributo del producto.

Con una meta de medición definida de manera explícita, se desarrolla un conjunto de preguntas. Las respuestas ayudarán al equipo de software (o a otros participantes) a determinar si se logró la meta de medición. Entre las preguntas que pueden plantearse se encuentran:

$P_1$: ¿Los componentes arquitectónicos se caracterizan de forma que compartimentalizan la función y los datos relacionados?

$P_2$: ¿La complejidad de cada componente dentro de las fronteras facilitará la modificación y la extensión?

Cada una de estas preguntas debe responderse de manera cuantitativa, usando una o más medidas y métricas. Por ejemplo, una métrica que proporciona un indicio de la cohesión (capítulo 8) de un componente arquitectónico puede ser útil para responder $P_1$. Las métricas que se estudian más adelante en este capítulo pueden proporcionar comprensión para $P_2$. En todo caso, las métricas que se eligen (o derivan) deben corresponderse con los principios de medición analizados en la sección 23.1.3 y con los atributos de medición expuestos en la sección 23.1.5.

### 23.1.5 Atributos de las métricas de software efectivas

Se han propuesto cientos de métricas para el software de computadora, pero no todas brindan apoyo práctico al ingeniero de software. Algunas demandan medición demasiado compleja, otras son tan particulares que pocos profesionales del mundo real tienen alguna esperanza de entenderlas y otras más violan las nociones intuitivas básicas de lo que realmente es el software de alta calidad.

Ejiogu [Eji91] define un conjunto de atributos que deben abarcar las métricas de software efectivas. La métrica derivada y las medidas que conducen a ella deben ser:

**¿Cómo se valora la calidad de una métrica de software propuesta?**

- *Simple y calculable.* Debe ser relativamente fácil aprender cómo derivar la métrica y su cálculo no debe demandar esfuerzo o tiempo excesivo.
- *Empírica e intuitivamente convincente.* Debe satisfacer las nociones intuitivas del ingeniero acerca del atributo de producto que se elabora (por ejemplo, una métrica que mide la cohesión del módulo debe aumentar en valor conforme aumenta el nivel de cohesión).
- *Congruente y objetiva.* Siempre debe producir resultados que no tengan ambigüedades. Una tercera parte independiente debe poder derivar el mismo valor de métrica usando la misma información acerca del software.
- *Constante en su uso de unidades y dimensiones.* El cálculo matemático de la métrica debe usar medidas que no conduzcan a combinaciones extrañas de unidades. Por ejemplo, multiplicar personas en los equipos de proyecto por variables de lenguaje de programación en el programa da como resultado una mezcla sospechosa de unidades que no son intuitivamente convincentes.
- *Independiente del lenguaje de programación.* Debe basarse en el modelo de requerimientos, el modelo de diseño o la estructura del programa en sí. No debe depender de los caprichos de la sintaxis o de la semántica del lenguaje de programación.
- *Un mecanismo efectivo para retroalimentación de alta calidad.* Debe proporcionar información que pueda conducir a un producto final de mayor calidad.

**CONSEJO**

*La experiencia indica que una métrica de producto sólo se usará si es intuitiva y fácil de calcular. Si se tienen que hacer decenas de "conteos" y se requieren cálculos complejos, es improbable que la métrica se adopte ampliamente.*

Aunque la mayoría de las métricas de software satisfacen estos atributos, algunas métricas de uso común pueden fracasar para satisfacer uno o dos de ellos. Un ejemplo es el punto de función (que se estudia en la sección 23.2.1), una medida de la "funcionalidad" entregada por el software. Puede argumentarse[3] que el atributo de congruente y objetiva fracasa porque una tercera

3 Puede plantearse un contrargumento igualmente vigoroso. Tal es la naturaleza de las métricas del software.

parte independiente no puede ser capaz de derivar el mismo valor del punto de función que un colega que use la misma información acerca del software. Por tanto, ¿debe rechazarse la medida PF? La respuesta es "¡desde luego que no!". El PF proporciona comprensión útil y, en consecuencia, ofrece distinto valor, incluso si falla en satisfacer un atributo a la perfección.

**CASASEGURA**

### *Debate acerca de las métricas de producto*

**La escena:** Cubículo de Vinod.

**Participantes:** Vinod, Jamie y Ed, miembros del equipo de ingeniería del software *CasaSegura*, quienes continúan trabajando en el diseño en el nivel de componentes y en el diseño de casos de prueba.

**La conversación:**

**Vinod:** Doug [Doug Miller, gerente de ingeniería del software] me dijo que todos debemos usar métricas de producto, pero fue muy vago. También dijo que no presionaría... que usarlas era asunto nuestro.

**Jamie:** Eso está bien porque no hay forma de que yo tenga tiempo para comenzar esa cosa de las medidas. Estamos peleando por mantener el calendario como está.

**Ed:** Estoy de acuerdo con Jamie. Estamos en contra, aquí... no tenemos tiempo.

**Vinod:** Sí, lo sé, pero probablemente hay algún mérito en usarlas.

**Jamie:** No lo discuto, Vinod, es cuestión de tiempo... y, en lo que a mí respecta, no tengo para perderlo.

**Vinod:** Pero, ¿y si las mediciones te ahorran tiempo?

**Ed:** Estás mal, requieren tiempo, y, como dijo Jamie...

**Vinod:** No, espera... ¿y si nos ahorra tiempo?

**Jamie:** ¿Cómo?

**Vinod:** Volver a trabajar... así es cómo. Si una medida que usemos nos ayuda a evitar un problema grande o incluso moderado, y esto evita que tengamos que volver a trabajar una parte del sistema, ahorramos tiempo. ¿O no?

**Ed:** Es posible, supongo, ¿pero puedes garantizarnos que alguna métrica de producto nos ayudará a encontrar un problema?

**Vinod:** ¿Puedes garantizarme que no lo hará?

**Jamie:** ¿Y qué es lo que propones?

**Vinod:** Creo que debemos seleccionar algunas métricas de diseño, probablemente orientadas a clase, y usarlas como parte de nuestro proceso de revisión para cada componente que desarrollemos.

**Ed:** No estoy familiarizado con las métricas orientadas a clase.

**Vinod:** Yo pasaré algo de tiempo revisándolas y haré una recomendación... ¿está bien para ustedes?

[Ed y Jamie asienten sin mucho entusiasmo.]

## 23.2 MÉTRICAS PARA EL MODELO DE REQUERIMIENTOS

El trabajo técnico en la ingeniería del software comienza con la creación del modelo de requerimientos. En esta etapa se derivan los requerimientos y se establece un cimiento para el diseño. Por tanto, son deseables métricas de producto que proporcionen comprensión acerca de la calidad del modelo de análisis.

Aunque en la literatura han aparecido relativamente pocas métricas de análisis y especificación, es posible adaptar las métricas que se usan frecuentemente para estimación de proyectos y aplicarlas en este contexto. Dichas métricas examinan el modelo de requerimientos con la intención de predecir el "tamaño" del sistema resultante. En ocasiones (mas no siempre), el tamaño es un indicador de la complejidad del diseño y casi siempre es un indicador creciente de codificación, integración y esfuerzo de pruebas.

### 23.2.1 Métrica basada en funciones

La *métrica de punto de función* (PF) puede usarse de manera efectiva como medio para medir la funcionalidad que entra a un sistema.[4] Al usar datos históricos, la métrica PF puede entonces usarse para: 1) estimar el costo o esfuerzo requerido para diseñar, codificar y probar el software;

4 Acerca de las métricas PF se han escrito cientos de libros, ensayos y artículos. En [IFP05] puede encontrar una valiosa bibliografía.

**WebRef**

En **www.ifpug.org** y en **www.functionpoints.com** puede obtenerse mucha información útil acerca de los puntos de función.

2) predecir el número de errores que se encontrarán durante las pruebas, y 3) prever el número de componentes y/o de líneas fuente proyectadas en el sistema implementado.

Los puntos de función se derivan usando una relación empírica basada en medidas contables (directas) del dominio de información del software y en valoraciones cualitativas de la complejidad del software. Los valores de dominio de información se definen en la forma siguiente:[5]

**Número de entradas externas (EE).** Cada *entrada externa* se origina de un usuario o se transmite desde otra aplicación, y proporciona distintos datos orientados a aplicación o información de control. Con frecuencia, las entradas se usan 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* es datos derivados dentro de la aplicación que ofrecen información al usuario. En este contexto, salida externa se refiere a reportes, pantallas, mensajes de error, etc. Los ítems de datos individuales dentro de un reporte no se cuentan por separado.

**Número de consultas externas (CE).** Una *consulta externa* se define como una entrada en línea que da como resultado la generación de alguna respuesta de software inmediata en la forma de una salida en línea (con frecuencia recuperada de un ALI).

**Número de archivos lógicos internos (ALI)**. Cada *archivo lógico interno* es un agrupamiento lógico de datos que reside dentro de la frontera de la aplicación y se mantiene mediante entradas externas.

**Número de archivos de interfaz externos (AIE).** Cada *archivo de interfaz externo* es un agrupamiento lógico de datos que reside fuera de la aplicación, pero que proporciona información que puede usar la aplicación.

Una vez recolectados dichos datos, la tabla de la figura 23.1 se completa y un valor de complejidad se asocia con cada conteo. Las organizaciones que usan métodos de punto de función desarrollan criterios para determinar si una entrada particular es simple, promedio o compleja. No obstante, la determinación de complejidad es un tanto subjetiva.

Para calcular puntos de función (PF), se usa la siguiente relación:

$$\text{PF} = \text{conteo total} \times [0.65 + 0.01 \times \Sigma (F_i)] \tag{23.1}$$

donde conteo total es la suma de todas las entradas PF obtenidas de la figura 23.1.

Los $F_i$ ($i$ = 1 a 14) son *factores de ajuste de valor* (FAV) con base en respuestas a las siguientes preguntas [Lon02]:

**FIGURA 23.1**
**Cálculo de puntos de función**

| Valor de dominio de información | Conteo | | Factor ponderado Simple | Promedio | Complejo | | |
|---|---|---|---|---|---|---|---|
| Entradas externas (EE) | | × | 3 | 4 | 6 | = | |
| Salidas externas (SE) | | × | 4 | 5 | 7 | = | |
| Consultas externas (CE) | | × | 3 | 4 | 6 | = | |
| Archivos lógicos internos (ALI) | | × | 7 | 10 | 15 | = | |
| Archivos de interfaz externos (AIE) | | × | 5 | 7 | 10 | = | |
| Conteo total | → | | | | | | |

5 En realidad, la definición de valores de dominio de información y la forma en la que se cuentan son un poco más complejos. Para más detalles, el lector interesado debe consultar [IFP01].

**PUNTO CLAVE**

Los factores de ajuste de valor se usan para proporcionar un indicio de la complejidad del problema.

1. ¿El sistema requiere respaldo y recuperación confiables?
2. ¿Se requieren comunicaciones de datos especializadas para transferir información hacia o desde la aplicación?
3. ¿Existen funciones de procesamiento distribuidas?
4. ¿El desempeño es crucial?
5. ¿El sistema correrá en un entorno operativo existente enormemente utilizado?
6. ¿El sistema requiere entrada de datos en línea?
7. ¿La entrada de datos en línea requiere que la transacción de entrada se construya sobre múltiples pantallas u operaciones?
8. ¿Los ALI se actualizan en línea?
9. ¿Las entradas, salidas, archivos o consultas son complejos?
10. ¿El procesamiento interno es complejo?
11. ¿El código se diseña para ser reutilizable?
12. ¿La conversión y la instalación se incluyen en el diseño?
13. ¿El sistema se diseña para instalaciones múltiples en diferentes organizaciones?
14. ¿La aplicación se diseña para facilitar el cambio y su uso por parte del usuario?

**WebRef**

En **irb.cs.uni-mogdeburg.de/sw-eng/us/java/fp/** puede encontrar una calculadora PF en línea.

Cada una de estas preguntas se responde usando una escala que varía de 0 (no importante o aplicable) a 5 (absolutamente esencial). Los valores constantes en la ecuación (23.1) y los factores ponderados que se aplican a los conteos de dominio de información se determinan de manera empírica.

Para ilustrar el uso de la métrica PF en este contexto, considere la representación simple de modelo de análisis que se ilustra en la figura 23.2. En la figura, se representa un diagrama de flujo de datos (capítulo 7) para una función dentro del software *CasaSegura*.

La función gestiona la interacción del usuario, acepta la contraseña de éste para activar o desactivar el sistema y permite consultas sobre el estado de las zonas de seguridad y de varios sensores de seguridad. La función despliega una serie de mensajes de advertencia y envía señales de control adecuadas a varios componentes del sistema de seguridad.

El diagrama de flujo de datos se evalúa para determinar un conjunto de medidas de dominio de información clave que son requeridas para calcular la métrica de 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 configuración sistema**) y también están presentes dos salidas externas

**FIGURA 23.2**

**Modelo de flujo de datos para el software *CasaSegura***

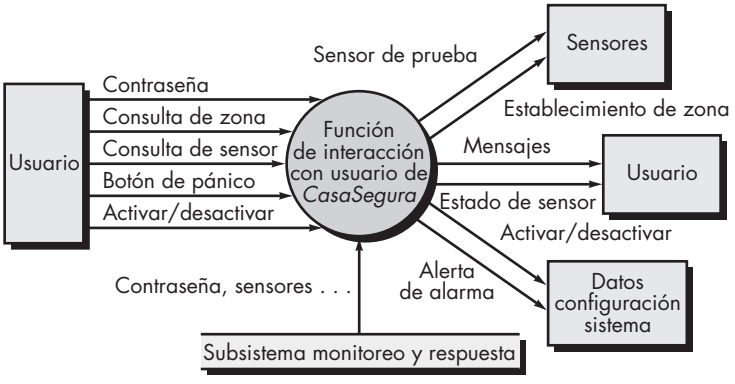


**FIGURA 23.3**
Cálculo de puntos de función

| Valor dominio de información | Conteo | | Factor ponderado: 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 lógicos internos (ALI) | 1 | × | 7 | 10 | 15 | = | 7 |
| Archivos de interfaz externos (AIE) | 4 | × | 5 | 7 | 10 | = | 20 |
| Conteo total | | | | | | | 50 |

(**mensajes** y **estado de sensor**) y cuatro AIE (**sensor de prueba**, **establecimiento de zona**, **activar/desactivar** y **alerta de alarma**). En la figura 23.3 se muestran estos datos, junto con la complejidad adecuada.

El conteo total que se muestra en la figura 23.3 debe ajustarse usando la ecuación (23.1). Para los propósitos de este ejemplo, suponga que $\Sigma(F_i)$ es 46 (un producto moderadamente complejo). Por tanto,

$$PF = 50 \times [0.65 + (0.01 \times 46)] = 56$$

**Cita:**

"En lugar de sólo meditar acerca de cuál 'nueva métrica' aplicar [...] también debemos plantearnos la pregunta más básica: ¿qué haremos con las métricas?"

**Michael Mah y Larry Putnam**

Con base en el valor PF proyectado, derivado del modelo de requerimientos, el equipo del proyecto puede estimar el tamaño global implementado de la función de interacción del usuario de *CasaSegura*. Suponga que los datos anteriores indican que un PF se traduce en 60 líneas de código (se usará un lenguaje orientado a objeto) y que se producen 12 PF por cada persona-mes de esfuerzo. Estos datos históricos ofrecen al gerente de proyecto información importante de planificación que se basa en el modelo de requerimientos y no en estimaciones preliminares. Suponga aún más, que en los proyectos pasados se encontró un promedio de tres errores por punto de función durante las revisiones de requerimientos y diseño, y cuatro errores por punto de función durante las pruebas de unidad e integración. A final de cuentas, dichos datos pueden ayudarlo a valorar lo completo de sus actividades de revisión y pruebas.

Uemura *et al.* [Uem99] sugieren que los puntos de función también pueden calcularse a partir de clases UML y diagramas de secuencia. Si tiene más interés, consulte detalles en [Uem99].

### 23.2.2 Métricas para calidad de la especificación

Davis *et al.* [Dav93] proponen una lista de características que pueden usarse para valorar la calidad del modelo de requerimientos y la correspondiente especificación de requerimientos: *especificidad* (falta de ambigüedad), *completitud*, *corrección*, *comprensibilidad*, *verificabilidad*, *consistencia interna y externa*, *factibilidad*, *concisión*, *rastreabilidad*, *modificabilidad*, *precisión* y *reusabilidad*. Además, los autores observan que las especificaciones de alta calidad se almacenan electrónicamente, son ejecutables o al menos interpretables, se anotan mediante importancia relativa, son estables, tienen versión, se organizan, cuentan con referencia cruzada y se especifican en el nivel correcto de detalle.

Aunque muchas de estas características parecen ser cualitativas por naturaleza, Davis *et al.* [Dav93] sugieren que cada una puede representarse usando una o más métricas. Por ejemplo, se supone que existen $n_r$ requerimientos en una especificación, tales que

$$n_r = n_f + n_{nf}$$

donde $n_f$ es el número de requerimientos funcionales y $n_{nf}$ es el número de requerimientos no funcionales (por ejemplo, rendimiento).

**PUNTO CLAVE**

Al medir las características de la especificación, es posible obtener comprensión cuantitativa acerca de la especificidad y de la completitud.

Para determinar la *especificidad* (falta de ambigüedad) de los requerimientos, Davis *et al.*, sugieren una métrica que se basa en la consistencia de la interpretación de los revisores de cada requisito:

$$Q_1 = \frac{n_{ui}}{n_r}$$

donde $n_{ui}$ es el número de requerimientos para los cuales todos los revisores tienen interpretaciones idénticas. Mientras más cercano a 1 esté el valor de $Q$, menor será la ambigüedad de la especificación.

La *completitud* de los requerimientos funcionales puede determinarse al calcular la razón

$$Q_2 = \frac{n_u}{n_i \times n_s}$$

donde $n_u$ es el número de requerimientos funcionales únicos, $n_i$ es el número de entradas (estímulos) definidas o implicadas por la especificación y $n_s$ es el número de estados especificados. La razón $Q_2$ mide el porcentaje de funciones necesarias que se especificaron para un sistema. Sin embargo, no aborda requerimientos no funcionales. Para incorporar éstos en una métrica global de completitud, se debe considerar el grado en el que se validaron los requerimientos:

$$Q_3 = \frac{n_c}{n_c + n_{nv}}$$

donde $n_c$ es el número de requerimientos que se validaron como correctos y $n_{nv}$ es el número de requerimientos que no se han validado.

**Cita:**

"Medir lo que es mensurable y lo que no es mensurable, hace [lo] mensurable."

**Galileo**

## 23.3 MÉTRICAS PARA EL MODELO DE DISEÑO

**PUNTO CLAVE**

Las métricas pueden proporcionar comprensión acerca de los datos estructurales y de la complejidad del sistema asociada con el diseño arquitectónico.

Es inconcebible que el diseño de una nueva aeronave, un nuevo chip de computadora o un nuevo edificio de oficinas se realizara sin definir medidas de diseño, ni determinar las métricas para varios aspectos de la calidad del diseño ni usarlos como indicadores para guiar la forma en la que evoluciona el diseño. Y aún así, con frecuencia el diseño de los sistemas complejos basados en software procede virtualmente sin medición. La ironía de esto es que están disponibles métricas del diseño para el software, pero la gran mayoría de los ingenieros del software continúan sin percatarse de su existencia.

Las métricas de diseño para software de computadora, al igual que todas las demás métricas de software, no son perfectas. El debate continúa acerca de su eficacia y sobre la forma en la que deben aplicarse. Muchos expertos argumentan que se requiere más experimentación antes de poder usar las medidas de diseño, aunque el diseño sin medición es una alternativa inaceptable.

En las siguientes secciones se examinan algunas de las métricas de diseño más comunes para software de computadora. Cada una puede proporcionarle comprensión mejorada y todas pueden ayudar a que el diseño evolucione hacia un mayor nivel de calidad.

### 23.3.1 Métricas del diseño arquitectónico

Las métricas del diseño arquitectónico se enfocan en características de la arquitectura del programa (capítulo 9) con énfasis en la estructura arquitectónica y en la efectividad de los módulos o componentes dentro de la arquitectura. Dichas métricas son "caja negra" en tanto no requieren conocimiento alguno del funcionamiento interior de un componente de software particular.

Card y Glass [Car90] definen tres medidas de complejidad del diseño de software: complejidad estructural, complejidad de datos y complejidad del sistema.

Para arquitecturas jerárquicas (por ejemplo, arquitecturas de petición y retorno), la *complejidad estructural* de un módulo *i* se define de la forma siguiente:

$$S(i) = f^2_{\text{out}}(i)$$

donde $f_{\text{out}}(i)$ es el *fan-out*[6] del módulo *i*.

La *complejidad de datos* ofrece un indicio de la complejidad que hay en la interfaz interna para un módulo *i* y se define como

$$D(i) = \frac{v(i)}{f_{\text{out}}(i) + 1}$$

donde $v(i)$ es el número de variables de entrada y salida que pasan hacia y desde el módulo *i*.

Finalmente, la *complejidad del sistema* se define como la suma de las complejidades estructural y de datos y se especifica como

$$C(i) = S(i) + D(i)$$

Conforme aumenta el valor de cada una de estas complejidades, la complejidad arquitectónica global del sistema también aumenta. Esto conduce a una mayor probabilidad de que también aumenten el esfuerzo de integración y el de pruebas.

Fenton [Fen91] sugiere algunas métricas simples de morfología (es decir, de forma) que permiten la comparación de diferentes arquitecturas de programa usando un conjunto de dimensiones directas. Con respecto a la arquitectura de llamado y retorno de la figura 23.4, puede definirse la siguiente métrica:

Tamaño = $n + a$

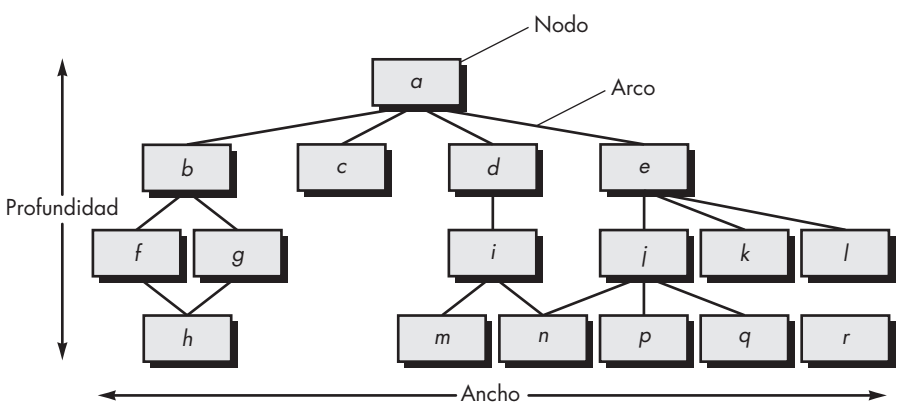
donde *n* es el número de nodos y *a* es el número de arcos. Para la arquitectura que se muestra en la figura 23.4,

Tamaño = 17 + 18 = 35
Profundidad = trayectoria más larga desde el nodo raíz (superior) hasta un nodo hoja. Para la arquitectura que se muestra en la figura 23.4, profundidad = 4.
Ancho = número máximo de nodos en cualquier nivel de la arquitectura. Para la arquitectura que se muestra en la figura 23.4, ancho = 6.

La razón arco a nodo, $r = a/n$, mide la densidad de conectividad de la arquitectura y puede proporcionar un indicio simple del acoplamiento de la arquitectura. Para la arquitectura que se muestra en la figura 23.4, $r = 18/17 = 1.06$.

La comandancia de los sistemas de la fuerza aérea estadounidense [USA87] desarrolló algunos indicadores de calidad del software que se basan en las características de diseño mensura-

**FIGURA 23.4**
**Métricas de morfología**




6 *Fan-out* (cargabilidad o abanico de salida) se define como el número de módulos inmediatamente subordinados al módulo *i*, es decir, el número de módulos que invoca directamente el módulo *i*.

bles de un programa de computadora. Al usar conceptos similares a los propuestos en IEEE Std. 982.1-1988 [IEE94], la fuerza aérea usa la información obtenida de los diseños de datos y arquitectónico para derivar un *índice de calidad de la estructura del diseño* (ICED) que varía de 0 a 1. Es necesario averiguar los siguientes valores para calcular el ICED [Cha89]:

$S_1$ = número total de módulos definidos en la arquitectura del programa
$S_2$ = número de módulos cuya función correcta depende de la fuente de entrada de datos o que produce los datos que se van a utilizar en alguna otra parte (en general, los módulos de control, entre otros, no se contarían como parte de $S_2$)
$S_3$ = número de módulos cuya función correcta depende del procesamiento previo
$S_4$ = número de ítems de base de datos (incluidos objetos de datos y todos los atributos que definen objetos)
$S_5$ = número total de ítems de base de datos únicos
$S_6$ = número de segmentos de base de datos (diferentes registros u objetos individuales)
$S_7$ = número de módulos con una sola entrada y salida (el procesamiento de excepción no se considera como una salida múltiple)

**Cita:**

"La medición puede verse como una desviación. Ésta es necesaria porque los humanos básicamente no son capaces de tomar decisiones claras y objetivas [sin apoyo cuantitativo]".

**Horst Zuse**

Una vez determinados los valores de $S_1$ a $S_7$ para un programa de cómputo, pueden calcularse los siguientes valores intermedios:

*Estructura del programa:* $D_1$, donde $D_1$ se define del modo siguiente: si el diseño arquitectónico se desarrolló usando un método distinto (por ejemplo, diseño orientado a flujo de datos o diseño orientado a objeto), entonces $D_1 = 1$, de otro modo $D_1 = 0$.

*Independencia de módulo:* $D_2 = 1 - \frac{S_2}{S_1}$

*Módulos no dependientes del procesamiento previo:* $D_3 = 1 - \frac{S_3}{S_1}$

*Tamaño de base de datos:* $D_4 = 1 - \frac{S_5}{S_4}$

*Compartimentalización de base de datos:* $D_5 = 1 - \frac{S_6}{S_4}$

*Entrada/salida de módulo característico:* $D_6 = 1 - \frac{S_7}{S_1}$

Con la determinación de estos valores intermedios, el ICED se calcula en la forma siguiente:

$$\text{ICED} = \Sigma\, w_i D_i$$

donde $i = 1$ a 6, $w_i$ es el peso relativo de la importancia de cada uno de los valores intermedios y $\Sigma w_i = 1$ (si todos los $D_i$ pesan igual, entonces $w_i = 0.167$).

El valor del ICED para diseños anteriores puede determinarse y compararse con un diseño que actualmente esté en desarrollo. Si el ICED es significativamente menor que el promedio, se indican más trabajo de diseño y revisión. De igual modo, si se hacen grandes cambios a un diseño existente puede calcularse el efecto de dichos cambios en el ICED.

### 23.3.2 Métricas para diseño orientado a objetos

Hay mucho de subjetivo en el diseño orientado a objetos: un diseñador experimentado "sabe" cómo caracterizar un sistema OO de modo que implemente de manera efectiva los requerimientos del cliente. Pero, conforme un modelo de diseño OO crece en tamaño y complejidad, una visión más objetiva de las características del diseño puede beneficiar tanto al diseñador experimentado (quien adquiere comprensión adicional) como al novato (quien obtiene un indicio de la calidad que de otro modo no tendría disponible).

En un tratamiento detallado de las métricas de software para sistemas OO, Whitmire [Whi97] describe nueve características distintas y mensurables de un diseño OO:

**¿Qué características pueden medirse cuando se valora un diseño OO?**

**Tamaño.** El tamaño se define en función de cuatro visiones: población, volumen, longitud y funcionalidad. La *población* se mide al realizar un conteo estático de entidades OO, tales como clases u operaciones. Las medidas de *volumen* son idénticas a las medidas de población, pero se recolectan de manera dinámica: en un instante de tiempo 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 un indicio indirecto del valor entregado al cliente por una aplicación OO.

**Cita:**

"Muchas de las decisiones en las cuales tuve que confiar en el folclore y el mito, ahora se pueden resolver haciendo uso de datos cuantitativos."

Scott Whitmire

**Complejidad.** Como el tamaño, existen muchas visiones diferentes de la complejidad del software [Zus97]. Whitmire ve la complejidad en términos de características estructurales al examinar cómo se relacionan mutuamente las clases de un diseño OO.

**Acoplamiento.** Las conexiones físicas entre elementos del diseño OO (por ejemplo, el número de colaboraciones entre clases o el de mensajes que pasan entre los objetos) representan el acoplamiento dentro de un sistema OO.

**Suficiencia.** Whitmire define *suficiencia* como "el grado en el que una abstracción posee las características requeridas de él o en el que un componente de diseño posee características en su abstracción, desde el punto de vista de la aplicación actual". Dicho de otra forma, se pregunta: "¿Qué propiedades debe poseer esta abstracción (clase) para serme útil?" [Whi97]. En esencia, un componente de diseño (por ejemplo, una clase) es *suficiente* si refleja por completo todas las propiedades del objeto de dominio de aplicación que se modela, es decir, si la abstracción (clase) posee sus características requeridas.

**Completitud.** La única diferencia entre completitud y suficiencia es "el conjunto de características contra las cuales se compara la abstracción o el componente de diseño" [Whi97]. La suficiencia compara la abstracción desde el punto de vista de la aplicación actual. La *completitud* considera múltiples puntos de vista, y plantea la pregunta: "¿qué propiedades se requieren para representar por completo al objeto de dominio problema?". Puesto que el criterio para completitud considera diferentes puntos de vista, tiene una implicación indirecta en el grado en el que puede reutilizarse la abstracción o el componente de diseño.

**Cohesión.** Como su contraparte en software convencional, un componente OO debe diseñarse de manera que tenga todas las operaciones funcionando en conjunto para lograr un solo propósito bien definido. La cohesividad de una clase se determina al examinar el grado en el que "el conjunto de propiedades que posee es parte del problema o dominio de diseño" [Whi97].

**Primitivismo.** Una característica que es similar a la simplicidad, el primitivismo (aplicado tanto a operaciones como a clases), es el grado en el 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 encapsula sólo operaciones primitivas.

**Similitud.** El grado en el que dos o más clases son similares en términos de su estructura, función, comportamiento o propósito se indica mediante esta medida.

**Volatilidad.** Como se menciona muchas veces en este libro, los cambios en el diseño pueden ocurrir cuando se modifican los requerimientos o cuando ocurren modificaciones en otras partes de una aplicación, lo que da como resultado la adaptación obligatoria del componente de diseño en cuestión. La volatilidad de un componente de diseño OO mide la probabilidad de que ocurrirá un cambio.

En realidad, las métricas de producto para sistemas OO pueden aplicarse no sólo al modelo de diseño, sino también al de requerimientos. En las siguientes secciones se estudian las métricas

que proporcionan un indicio de la calidad en el nivel de clase OO y en el de operación. Además, también se exploran las métricas aplicables a la administración del proyecto y a las pruebas.

### 23.3.3 Métricas orientadas a clase: la suite de métricas CK

La clase es la unidad fundamental de un sistema OO. Por tanto, las medidas y métricas para una clase individual, la jerarquía de clase y las colaboraciones de clase serán invaluables cuando se requiera valorar la calidad del diseño OO. Una clase encapsula datos y la función que los manipula. Con frecuencia es el "padre" de las subclases (en ocasiones llamadas hijos) que heredan sus atributos y operaciones. Usualmente colabora con otras clases. Cada una de estas características puede usarse como la base para la medición.[7]

Chidamber y Kemerer propusieron uno de los conjuntos de métricas de software OO de mayor referencia [Chi94]. En ocasiones llamada *suite de métricas CK*, los autores proponen seis métricas de diseño basadas en clase para sistemas OO.[8]

**Métodos ponderados por clase (MPC).** Suponga que $n$ métodos de complejidad $c_1$, $c_2$, ..., $c_n$ se definen para una clase **C**. La métrica de complejidad específica que se elige (por ejemplo, complejidad ciclomática) debe normalizarse de modo que la complejidad nominal para un método tome un valor de 1.0.

$$\text{MPC} = \Sigma c_i$$

para $i = 1$ hasta $n$. El número de métodos y su complejidad son indicadores razonables de la cantidad de esfuerzo requerido para implementar y probar una clase. Además, mientras más grande sea el número de métodos, más complejo será el árbol de herencia (todas las subclases heredan los métodos de sus padres). Finalmente, conforme el número de métodos crece para una clase determinada, es probable que se vuelva cada vez más específica su aplicación y, por tanto, limite la potencial reutilización. Por todas estas razones, MPC debe mantenerse tan bajo como sea razonable.

Aunque parecería relativamente directo desarrollar un conteo para el número de métodos en una clase, el problema en realidad es más complejo de lo que parece. Debe desarrollarse un enfoque de conteo consistente [Chu95].

**Profundidad del árbol de herencia (PAH).** Esta métrica es "la máxima longitud desde el nodo hasta la raíz del árbol" [Chi94]. En la figura 23.5, el valor de la PAH para la jerarquía de clase que se muestra es 4. Conforme crece la PAH, es probable que las clases de nivel inferior hereden muchos métodos. Esto conduce a potenciales dificultades cuando se intenta predecir el comportamiento de una clase. Una jerarquía de clase profunda (PAH grande) también conduce a mayor complejidad de diseño. En el lado positivo, grandes valores de PAH implican que muchos métodos pueden reutilizarse.

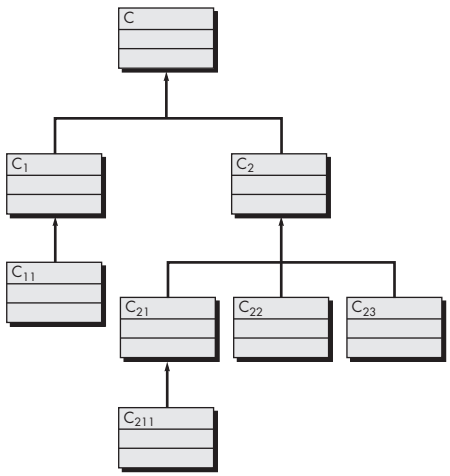
**Número de hijos (NDH).** Las subclases que son inmediatamente subordinadas a una clase en la jerarquía de clase se denominan hijos. En la figura 23.5, la clase $\mathbf{C_2}$ tiene tres hijos: subclases $\mathbf{C_{21}}$, $\mathbf{C_{22}}$ y $\mathbf{C_{23}}$. Conforme crece el número de hijos, el reuso aumenta, pero también, como el NDH aumenta, la abstracción representada por la clase padre puede diluirse si algunos de los hijos no son miembros adecuados de la clase padre. Conforme el NDH aumenta, la cantidad de pruebas (requeridas para ejercitar cada hijo en su contexto operativo) también aumentará.

---

7 Debe observarse que, en la literatura técnica, actualmente está en debate la validez de algunas de las métricas estudiadas en este capítulo. Quienes defienden la teoría de mediciones demandan un grado de formalismo que algunas métricas OO no proporcionan. Sin embargo, es razonable afirmar que las métricas mencionadas ofrecen comprensión útil para el ingeniero de software.

8 Chidamber, Darcy y Kemerer usan el término *métodos* en lugar de *operaciones*. El uso de este término se refleja en esta sección.

**FIGURA 23.5**
**Una jerarquía de clase**





*Los conceptos de acoplamiento y cohesión se aplican tanto a software convencional como a OO. Mantenga el acoplamiento de clase bajo y la cohesión de clase y operación alta.*

**Acoplamiento entre clases de objetos (ACO).** El modelo CRC (capítulo 6) puede usarse para determinar el valor para el ACO. En esencia, ACO es el número de colaboraciones citadas para una clase en su tarjeta índice CRC.[9] Conforme el ACO aumenta, es probable que la reusabilidad de una clase disminuya. Valores altos de ACO también complican las modificaciones y las pruebas que sobrevienen cuando se realizan modificaciones. En general, los valores de ACO para cada clase deben mantenerse tan bajos como sea razonable. Esto es consistente con el lineamiento general para reducir el acoplamiento en el software convencional.

**Respuesta para una clase (RPC).** Respuesta para una clase es "un conjunto de métodos que potencialmente pueden ejecutarse en respuesta a un mensaje recibido por un objeto de dicha clase" [Chi94]. RPC es el número de métodos en el conjunto respuesta. Conforme aumenta la RPC, también lo hace el esfuerzo requerido para probar, pues la secuencia de pruebas (capítulo 19) crece. Igualmente, se sigue que, conforme la RPC aumenta, la complejidad de diseño global de la clase aumenta.

**Falta de cohesión en métodos (FCOM).** Cada método dentro de una clase **C** accede a uno o más atributos (también llamados variables de instancia). FCOM es el número de métodos que acceden a uno o más de los mismos atributos.[10] Si ningún método accede a los mismos atributos, entonces la FCOM = 0. Para ilustrar el caso donde la FCOM ≠ 0, considere una clase con seis métodos. Cuatro de ellos tienen uno o más atributos en común (es decir, acceden a atributos comunes). Por tanto, la FCOM = 4. Si la FCOM es alta, los métodos pueden acoplarse unos con otros mediante atributos. Esto aumenta la complejidad del diseño de clase. Aunque hay casos en los que un valor alto de la FCOM es justificable, es deseable mantener alta la cohesión, es decir, mantener baja la FCOM.[11]

9 Si las tarjetas índice CRC se desarrollan manualmente, completitud y consistencia deben valorarse antes de que el ACO pueda determinarse de manera confiable.

10 La definición formal es un poco más compleja. Vea [Chi94] para detalles.

11 La métrica FCOM proporciona útil comprensión en algunas situaciones, pero puede confundir en otras. Por ejemplo, mantener en acoplamiento encapsulado dentro de una clase aumenta la cohesión del sistema como un todo. Por tanto, en al menos un sentido importante, la FCOM más alta en realidad sugiere que una clase puede tener mayor cohesión, no menor.

**CasaSegura**

### *Aplicación de métricas CK*

**La escena:** Cubículo de Vinod.

**Participantes:** Vinod, Jamie, Shakira y Ed, miembros del equipo de ingeniería del software *CasaSegura*, quienes continúan trabajando en el diseño en el nivel de componentes y en el diseño de casos de prueba.

**La conversación:**

**Vinod:** ¿Alguno de ustedes tuvo oportunidad de leer la descripción de la suite de métricas CK que envié el miércoles e hizo las mediciones?

**Shakira:** No fue muy complicado. Regresé a mi clase UML y a diagramas de secuencia, como sugeriste, y obtuve conteos burdos para PAH, RPC y FCOM. No pude encontrar el modelo CRC, así que no conté ACO.

**Jamie (sonríe):** No pudiste encontrar el modelo CRC porque yo lo tengo.

**Shakira:** Por eso adoro a este equipo: comunicación soberbia.

**Vinod:** Yo hice mis conteos... ¿ustedes desarrollaron números para las métricas CK?

[Jamie y Ed asienten.]

**Jamie:** Dado que yo tenía las tarjetas CRC, eché un vistazo al ACO y parecía bastante uniforme a través de la mayoría de las clases. Hubo una excepción, la cual anoté.

**Ed:** Hay algunas clases donde la RPC es muy alta, en comparación con los promedios... tal vez debamos echar un vistazo para simplificarlos.

**Jamie:** Quizá sí, quizá no. Todavía estoy preocupado por el tiempo, y no quiero componer cosas que en realidad no están rotas.

**Vinod:** Estoy de acuerdo. Tal vez debas buscar clases que tengan malos números en al menos dos o más de las métricas CK. Cuestión de dos strikes y estás modificado.

**Shakira [observa la lista de clases de Ed con alta RPC]:** Mira, ve estas clases, tienen alta FCOM así como alta RPC. ¿Dos strikes?

**Vinod:** Sí, eso creo... será difícil implementar debido a la complejidad y dificultad para probar por la misma razón. Probablemente valdría la pena diseñar dos clases separadas para lograr el mismo comportamiento.

**Jamie:** ¿Crees que modificarlo nos ahorrará tiempo?

**Vinod:** A largo plazo, sí.

### 23.3.4 Métricas orientadas a clase: La suite de métricas MOOD

**Cita:**

"Analizar el software OO para evaluar su calidad se ha vuelto cada vez más importante conforme el paradigma [OO] continúa aumentando en popularidad."

**Rachel Harrison *et al.***

Harrison, Counsell y Nithi [Har98b] proponen un conjunto de métricas para diseño orientado a objeto que proporciona indicadores cuantitativos para características de diseño OO. A continuación se presenta un muestreo de métricas MOOD.

**Factor de herencia de método (FHM).** El grado en el que la arquitectura de clase de un sistema OO utiliza la herencia tanto para métodos (operaciones) como para atributos se define como

$$\text{FHM} = \frac{\Sigma M_i (C_i)}{\Sigma M_a (C_i)}$$

donde la suma ocurre sobre $i = 1$ hasta TC. TC 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

$M_a(C_i)$ = número de métodos que pueden invocarse en asociación con $C_i$
$M_d(C_i)$ = número de métodos declarados en la clase $C_i$
$M_i(C_i)$ = número de métodos heredados (y no invalidados) en $C_i$

El valor de FHM [el factor de herencia de atributo (FHA) se define de forma análoga] ofrece un indicio del impacto de la herencia sobre el software OO.

**Factor de acoplamiento (FA).** Anteriormente, en este capítulo, se dijo que el acoplamiento es un indicio de las conexiones entre elementos del diseño OO. La suite de métricas MOOD define el acoplamiento en la forma siguiente:

$$\text{FA} = \Sigma_i \Sigma_j \, is\_client \, \frac{(C_i, C_j)}{T_c^2 - T_c}$$

donde las sumas ocurren sobre $i = 1$ hasta $T_c$ y $j = 1$ hasta $T_c$. La función

$$\begin{aligned} \textit{is\_client} \text{ (es cliente)} &= 1, \text{ si y sólo si existe una relación entre la clase cliente } C_c \text{ y la clase servidor } C_s, \text{ y } C_c \neq C_s \\ &= 0, \text{ de otro modo} \end{aligned}$$

Aunque muchos factores afectan la complejidad, comprensibilidad y mantenimiento del software, es razonable concluir que, conforme el valor de FA aumenta, la complejidad del software OO también aumentará, y como resultado pueden sufrir la comprensibilidad, el mantenimiento y el potencial de reuso.

Harrison *et al.* [Har98b] presentan un análisis detallado de FHM y FA junto con otras métricas, y examinan su validez para usarlas en la valoración de la calidad del diseño.

### 23.3.5 Métricas OO propuestas por Lorenz y Kidd

En su libro acerca de métricas OO, Lorenz y Kidd [Lor94] dividen las métricas basadas en clase en cuatro amplias categorías; cada una tiene una relación en el diseño en el nivel de componentes: tamaño, herencia, internos y externos. Las métricas orientadas a tamaño para una clase de diseño OO se enfocan en conteos de atributos y operaciones para una clase individual y en valores promedio para el sistema OO como un todo. Las métricas basadas en herencia se enfocan en la forma en la que las operaciones se reutilizan a lo largo de la jerarquía de clases. Las métricas para interiores de clase se fijan en la cohesión (sección 23.3.3) y en los conflictos orientados a código. Y las métricas externas examinan el acoplamiento y el reuso. Un ejemplo de las métricas propuestas por Lorenz y Kidd es:

**Tamaño de clase (TDC).** El tamaño global de una clase puede determinarse usando las siguientes medidas:

- El número total de operaciones (tanto heredadas como operaciones de instancia privada) que se encapsulan dentro de la clase.
- El número de atributos (tanto heredados como de instancia privada) que encapsula la clase.


*Durante la revisión del modelo de análisis, las tarjetas índice CRC proporcionarán un indicio razonable de los valores esperados para TDC. Si encuentra una clase con un gran número de responsabilidades, considere dividirla.*

La métrica MPC propuesta por Chidamber y Kemerer (sección 23.3.3) también es una medida ponderada del tamaño de clase. Como se indicó anteriormente, grandes valores para TDC indican que una clase puede tener demasiada responsabilidad. Esto reducirá la reutilización de la clase y complicará la implementación y las pruebas. En general, las operaciones y atributos heredados o públicos deben ponderarse más para determinar el tamaño de clase [Lor94]. Las operaciones y atributos privados permiten la especialización y están más localizadas en el diseño. También pueden calcularse promedios para el número de atributos y operaciones de clase. Mientras más bajos sean los valores promedio para el tamaño, hay más probabilidad de que las clases dentro del sistema puedan reutilizarse ampliamente.

### 23.3.6 Métricas de diseño en el nivel de componente

Las métricas de diseño en el nivel de componente para componentes de software convencional se enfocan en las características internas de un componente de software e incluyen medidas de cohesión de módulo, acoplamiento y complejidad. Dichas medidas pueden ayudarlo a juzgar la calidad de un diseño en el nivel de componente.

Las métricas de diseño en el nivel de componente pueden aplicarse una vez desarrollado el diseño procedural y son "cajas de cristal" en tanto requieren conocimiento del funcionamiento interior del módulo en el que se está trabajando. Alternativamente, pueden demorarse hasta que el código fuente esté disponible.

PUNTO CLAVE
Es posible calcular medidas de la independencia funcional (acoplamiento y cohesión) de un componente y usarlas para valorar la calidad de un diseño.

**Métricas de cohesión.** Bieman y Ott [Bie94] definen una colección de métricas que proporcionan un indicio de la cohesión (capítulo 8) de un módulo. Las métricas se definen mediante cinco conceptos y medidas:

*Rebanada de datos* (*data slice*). Dicho de manera simple, una rebanada de datos es una marcha hacia atrás a través de un módulo que busca valores de datos que afecten la ubicación del módulo donde comenzó la marcha. Debe observarse que es posible definir tanto las rebanadas de programa (que se enfocan en enunciados y condiciones) como las rebanadas de datos.

*Símbolos de datos* (*data tokens*). Las variables definidas por un módulo pueden definirse como *tokens* de datos para el módulo.

*Símbolos pegamento* (*glue tokens*). Este conjunto de *tokens* de datos se encuentra en una o más rebanadas de datos.

*Símbolos superpegamento* (*superglue tokens*). Estos *tokens* de datos son comunes a cada rebanada de datos en un módulo.

*Pegajosidad* (*stickiness*). La pegajosidad relativa de un *token* pegamento es directamente proporcional al número de rebanadas de datos que enlaza.

Bieman y Ott desarrollan métricas para *cohesión funcional fuerte* (CFF), *cohesión funcional débil* (CFD) y *adhesividad* (el grado relativo en el que los *tokens* pegamento enlazan rebanadas de datos). Un análisis detallado de las métricas de Bieman y Ott, se deja a los autores [Bie94].

**Métricas de acoplamiento.** El acoplamiento de módulo proporciona un indicio de cuán "conectado" está un módulo con otros módulos, con datos globales y con el entorno exterior. En el capítulo 9 se estudió el acoplamiento en términos cualitativos.

Dhama [Dha95] propuso una métrica para acoplamiento de módulo que abarca acoplamiento de datos y flujo de control, acoplamiento global y acoplamiento ambiental. Las medidas requeridas para calcular acoplamiento de módulo se definen en función de cada uno de los tres tipos de acoplamiento anotados anteriormente.
Para acoplamiento de datos y flujo de control,

$d_i$ = número de parámetros de datos de entrada
$c_i$ = número de parámetros de control de entrada
$d_o$ = número de parámetros de datos de salida
$c_o$ = número de parámetros de control de salida

Para acoplamiento global,

$g_d$ = número de variables globales usadas como datos
$g_c$ = número de variables globales usadas como control

Para acoplamiento de entorno,

$w$ = número de módulos llamados (*fan-out*)
$r$ = número de módulos que llaman al módulo bajo consideración (*fan-in*)

Al usar estas medidas, un indicador de acoplamiento de módulo $m_c$ se define de la forma siguiente:

$$m_c = \frac{k}{M}$$

donde $k$ es una constante de proporcionalidad y

$$M = d_i + (a \times c_i) + d_o + (b \times c_o) + g_d + (c \times g_c) + w + r$$

Los valores para $k$, $a$, $b$ y $c$ deben derivarse de manera empírica.

Conforme el valor de $m_c$ crece, el acoplamiento de módulo global disminuye. Con la finalidad de que la métrica de acoplamiento se mueva hacia arriba conforme el grado de acoplamiento aumenta, una métrica de acoplamiento revisada puede definirse como

$$C = 1 - m_c$$

donde el grado de acoplamiento aumenta conforme el valor de *M* aumenta.

**Métricas de complejidad.** Para determinar la complejidad del flujo de control de programa, pueden calcularse varias métricas de software. Muchas de éstas se basan en el gráfico de flujo. Un gráfico (capítulo 18) es una representación compuesta de nodos y ligas (también llamadas aristas). Cuando las ligas (aristas) se dirigen, el gráfico de flujo es un gráfico dirigido.

McCabe y Watson [McC94] identifican algunos usos importantes para las métricas de complejidad:

> Las métricas de complejidad pueden usarse para predecir información crucial acerca de la confiabilidad y el mantenimiento de los sistemas de software a partir de análisis automáticos de código fuente [o información de diseño procedimental]. Las métricas de complejidad también proporcionan retroalimentación durante el proyecto de software para ayudar a controlar la [actividad de diseño]. Durante las pruebas y el mantenimiento, proporcionan información detallada acerca de los módulos de software para ayudar a destacar áreas de potencial inestabilidad.

La métrica de complejidad para software de computadora más ampliamente usada es la complejidad ciclomática, originalmente desarrollada por Thomas McCabe [McC76] y que se estudió en detalle en el capítulo 18.

PUNTO CLAVE
La complejidad ciclomática es sólo una entre un gran número de métricas de complejidad.

Zuse ([Zus90], [Zus97]) presenta una discusión enciclopédica de no menos de 18 diferentes categorías de métricas de complejidad de software. El autor expone las definiciones básicas para las métricas en cada categoría (por ejemplo, existen algunas variaciones en la métrica de complejidad ciclomática) y luego analiza y critica cada una. El trabajo de Zuse es el más exhaustivo publicado a la fecha.

### 23.3.7 Métricas orientadas a operación

Puesto que la clase es la unidad dominante en los sistemas OO, se han propuesto menos métricas para operaciones que residen dentro de una clase. Churcher y Shepperd [Chu95] analizan esto cuando afirman: "Los resultados de estudios recientes indican que los métodos tienden a ser pequeños, tanto en términos de número de enunciados como en complejidad lógica [Wil93], lo que sugiere que la estructura de conectividad de un sistema puede ser más importante que el contenido de los módulos individuales." Sin embargo, puede obtenerse algo de comprensión al examinar las características promedio para los métodos (operaciones). Tres métricas simples, propuestas por Lorenz y Kidd [Lor94], son apropiadas:

**Tamaño promedio de operación ($TO_{prom}$).** El tamaño puede determinarse al contar el número de líneas de código o el de mensajes enviados por la operación. Conforme aumenta el número de mensajes enviados por una sola operación, es probable que las responsabilidades no se hayan asignado bien dentro de una clase.

**Complejidad de la operación (CO).** La complejidad de una operación puede calcularse usando cualquiera de las métricas de complejidad propuestas para software convencional [Zus90]. Puesto que las operaciones deben limitarse a una responsabilidad específica, el diseñador debe luchar por mantener la CO tan baja como sea posible.

**Número promedio de parámetros por operación ($NP_{prom}$).** Mientras más grande sea el número de parámetros de operación, más compleja es la colaboración entre objetos. En general, el $NP_{prom}$ debe mantenerse tan bajo como sea posible.

### 23.3.8 Métricas de diseño de interfaz de usuario

**Cita:**

"Es posible aprender al menos un principio del diseño de interfaz de usuario al cargar una lavadora de platos. Si apila muchos en ella, nada quedará muy limpio."

**Autor desconocido**

Aunque hay considerable literatura acerca del diseño de interfaces hombre/computadora (capítulo 11), se ha publicado relativamente poca información acerca de las métricas que proporcionarían comprensión de la calidad y de la usabilidad de la interfaz.

Sears [Sea93] sugiere que la *corrección de la plantilla* (CP) es una métrica de diseño valioso para las interfaces hombre/computadora. Una GUI típica usa entidades de plantilla (íconos gráficos, texto, menús, ventanas y similares) para auxiliar al usuario a completar tareas. Para lograr una tarea dada usando una GUI, el usuario debe moverse de una entidad de plantilla a la siguiente. La posición absoluta y relativa de cada entidad de plantilla, la frecuencia con la que se usa y el "costo" de la transición desde una entidad de plantilla a la siguiente contribuirán a la corrección de la interfaz.

Un estudio de métricas de página web [Ivo01] indica que las características simples de los elementos de la plantilla también pueden tener un impacto significativo sobre la calidad percibida del diseño GUI. El número de palabras, vínculos, gráficos, colores y fuentes (entre otras características) contenidas en una página web afectan la complejidad percibida y la calidad de dicha página.

Es importante observar que la selección de un diseño GUI puede guiarse con métricas como CP, pero el árbitro final debe ser la entrada del usuario con base en los prototipos GUI. Nielsen y Levy [Nie94] reportan que "uno tiene una oportunidad razonablemente grande de triunfar si se elige entre [diseños de] interfaz basadas exclusivamente en opiniones de los usuarios. El rendimiento de tarea promedio de los usuarios y su satisfacción subjetiva con una GUI están enormemente correlacionados".

## 23.4 MÉTRICAS DE DISEÑO PARA *WEBAPPS*

Un útil conjunto de medidas y métricas para *webapps* proporciona respuestas cuantitativas a las siguientes preguntas:

- ¿La interfaz de usuario promueve usabilidad?
- ¿La estética de la *webapp* es apropiada para el dominio de aplicación y agrada al usuario?
- ¿El contenido se diseñó de tal forma que imparte más información con menos esfuerzo?
- ¿La navegación es eficiente y directa?
- ¿La arquitectura de la *webapp* se diseñó para alojar 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 se diseñaron de manera que se reduce la complejidad procedimental y se mejora la exactitud, la confiabilidad y el desempeño?

**CONSEJO**

*Muchas de estas métricas son aplicables a todas las interfaces de usuario y deben considerarse en conjunto con las que se presentaron en la sección 23.3.8.*

En la actualidad, cada una de estas preguntas puede abordarse sólo de manera cualitativa, porque todavía no existe una suite validada de métricas que proporcionen respuestas cuantitativas.

En los siguientes párrafos se presenta una muestra representativa de métricas de diseño *webapp* que se han propuesto en la literatura. Es importante observar que muchas de ellas todavía no se validan, por lo que deben usarse juiciosamente.

**Métricas de interfaz.** Para *webapps* pueden considerarse las siguientes medidas:

| *Métrica sugerida* | *Descripción* |
|---|---|
| Corrección de plantilla | Ver sección 23.3.8 |
| Complejidad de plantilla | Número de regiones[12] distintas definidas por una interfaz |
| Complejidad de región de plantilla | Número promedio de distintos vínculos por región |
| Complejidad de reconocimiento | Número promedio de distintos ítems que el usuario debe buscar antes de realizar una navegación o decidir la entrada de datos |
| Tiempo de reconocimiento | Tiempo promedio (en segundos) que tarda un usuario en seleccionar la acción adecuada para una tarea determinada |
| Esfuerzo de escritura | Número promedio de golpes de tecla requeridos para una función específica |
| Esfuerzo de toma de ratón | Número promedio de tomas de ratón por función |
| Complejidad de selección | Número promedio de vínculos que pueden seleccionarse por página |
| Tiempo de adquisición de contenido | Número promedio de palabras de texto por página web |
| Carga de memoria | Número promedio de distintos ítems de datos que el usuario debe recordar para lograr un objetivo específico |

**Métricas estéticas (diseño gráfico).** Por su naturaleza, el diseño estético se apoya en el juicio cualitativo y por lo general no es sensible a la medición ni a las métricas. Sin embargo, Ivory *et al.* [Ivo01] proponen un conjunto de medidas que pueden ser útiles para valorar el impacto del diseño estético:

| *Métrica sugerida* | *Descripción* |
|---|---|
| Conteo de palabra | Número total de palabras que aparecen en una página |
| Porcentaje de texto de cuerpo | Porcentaje de palabras que son cuerpo frente a texto de despliegue (es decir, títulos) |
| % texto cuerpo enfatizado | Porción de texto de cuerpo que se enfatiza (por ejemplo, negrillas, mayúsculas) |
| Conteo de posicionamiento de texto | Cambios en posición de texto desde el alineado a la izquierda |
| Conteo de grupo de texto | Áreas de texto resaltadas con color, regiones con bordes, reglas o listas |
| Conteo de vínculos | Vínculos totales en una página |
| Tamaño de página | Bytes totales para la página, así como elementos, gráficos y hojas de estilo |
| Porcentaje gráfico | Porcentaje de bytes de página que son usados para gráficos |
| Conteo gráfico | Gráficos totales en una página (no incluye gráficos especificados en guiones, applets y objetos) |
| Conteo de color | Total de colores empleados |
| Conteo de fuente | Total de fuentes empleadas (es decir, tipo + tamaño + negrilla + itálica) |

**Métricas de contenido.** Las métricas en esta categoría se enfocan en la complejidad del contenido y en los grupos de objetos de contenido que se organizan en páginas [Men01].

| *Métrica sugerida* | *Descripción* |
|---|---|
| Espera de página | Tiempo promedio requerido para que una página se descargue a diferentes velocidades de conexión |
| Complejidad de página | Número promedio de tipos diferentes de medios usados en la página, no incluido el texto |

12 Una región distinta es un área dentro de la plantilla de despliegue que logra cierto conjunto específico de funciones relacionadas (por ejemplo, una barra de menú, un despliegue gráfico estático, un área de contenido, un despliegue animado).

| | |
|---|---|
| Complejidad gráfica | Número promedio de medios gráficos por página |
| Complejidad de audio | Número promedio de medios de audio por página |
| Complejidad de video | Número promedio de medios de video por página |
| Complejidad de animación | Número promedio de animaciones por página |
| Complejidad de imagen escaneada | Número promedio de imágenes escaneadas por página |

**Métricas de navegación.** Las métricas en esta categoría abordan la complejidad del flujo de navegación [Men01]. En general, son útiles sólo para aplicaciones web estáticas, que no incluyen vínculos y páginas generados de manera dinámica.

| *Métrica sugerida* | *Descripción* |
|---|---|
| Complejidad de vinculación de página | Número de vínculos por página |
| Conectividad | Número total de vínculos internos, no incluidos vínculos generados de manera dinámica |
| Densidad de conectividad | Conectividad dividida por conteo de página |

Usar un subconjunto de las métricas sugeridas puede servir para derivar relaciones empíricas que permiten a un equipo de desarrollo *webapp* valorar la calidad técnica y predecir el esfuerzo con base en las estimaciones de complejidad estimadas. En esta área todavía queda mucho trabajo por hacer.

HERRAMIENTAS DE SOFTWARE

### *Métricas técnicas para* webapps

**Objetivo:** Auxiliar a los ingenieros web a desarrollar métricas *webapp* significativas que proporcionen comprensión acerca de la calidad global de una aplicación.

**Mecánica:** La mecánica de las herramientas varía.

**Herramientas representativas:**[13]

*Netmechanic Tools*, desarrollada por Netmechanic (**www.netmechanic.com**), es una colección de herramientas que ayudan a mejorar el desempeño de sitios web y que se enfocan en conflictos específicos de implementación.

*NIST Web Metrics Testbed*, desarrollada por The National Institute of Standards and Technology (**zing.ncsl.nist.gov/WebTools/**), abarca la siguiente colección de útiles herramientas que están disponibles para descarga:

*Web Static Analyzer Tool (WebSAT):* comprueba el HTML de la página web contra lineamientos de usabilidad típicos.

*Web Category Analysis Tool (WebCAT):* permite que el ingeniero de usabilidad construya y realice un análisis de categoría web.

*Web Variable Instrumenter Program (WebVIP):* instrumenta un sitio web para capturar una bitácora de interacción con el usuario.

*Framework for Logging Usability Data (FLUD):* implementa un formateador de archivo y analizador gramatical para representación de las bitácoras de interacción del usuario.

*VisVIP Tool:* produce una visualización 3D 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.

## 23.5 MÉTRICAS PARA 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 software

13 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría.

14 Debe observarse que las "leyes" de Halstead generaron gran controversia, y muchos creen que la teoría subyacente tiene fallas. Sin embargo, se ha realizado verificación experimental para lenguajes de programación seleccionados (por ejemplo, [Fel89]).

**Cita:**

"El cerebro humano sigue un conjunto de reglas más rígido [para desarrollo de algoritmos] del que se tiene conocimiento."

**Maurice Halstead**

de computadora, usando un conjunto de medidas primitivas que pueden derivarse después de generar el código o de que el diseño esté completo. Las medidas son:

$n_1$ = número de operadores distintos que aparecen en un programa
$n_2$ = número de operandos distintos que aparecen en un programa
$N_1$ = número total de ocurrencias de operador
$N_2$ = número total de ocurrencias de operando

Halstead usa estas medidas primitivas para desarrollar: expresiones para la longitud de programa global, volumen mínimo potencial para un algoritmo, volumen real (número de bits requeridos para especificar un programa), nivel del programa (una medida de complejidad del software), nivel del lenguaje (una constante para un lenguaje determinado) y otras características, como esfuerzo de desarrollo, tiempo de desarrollo e incluso el número proyectado de fallas en el software.

Halstead muestra que la longitud $N$ puede estimarse

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

y el volumen del programa puede definirse

$$V = N \log_2 (n_1 + n_2)$$

Debe observarse que $V$ variará con el lenguaje de programación y representa el volumen de información (en bits) requerido para especificar un programa.

**CONSEJO**

*Los operadores incluyen todo el flujo de constructos de control, condicionales y operaciones matemáticas. Los operandos abarcan todas las variables y constantes de programa.*

Teóricamente, debe existir un volumen mínimo para un algoritmo particular. Halstead define una razón de volumen $L$ como la razón del volumen de la forma más compacta de un programa al volumen del programa real. En realidad, $L$ siempre debe ser menor que 1. En términos de medidas primitivas, la razón de volumen puede expresarse como

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

El trabajo de Halstead es sensible a la verificación experimental y se ha llevado a cabo un gran trabajo para investigar la ciencia del software. Un análisis de este trabajo está más allá del ámbito de este libro. Para mayor información, vea [Zus90], [Fen91] y [Zus97].

## 23.6 MÉTRICAS PARA PRUEBAS

**PUNTO CLAVE**

Las métricas de prueba se ubican en dos amplias categorías: 1) métricas que intentan predecir el número probable de pruebas requeridas en varios niveles de prueba y 2) métricas que se enfocan en la cobertura de pruebas para un componente determinado.

Aunque se ha escrito mucho acerca de las métricas de software para pruebas (por ejemplo, [Het93]), la mayoría de las métricas proponen enfocarse en el proceso de las pruebas, no en las características técnicas de las pruebas en sí. En general, los examinadores deben apoyarse en las métricas de análisis, diseño y código para guiarlos en el diseño y la ejecución de los casos de prueba.

Las métricas del diseño arquitectónico proporcionan información acerca de la facilidad o dificultad asociada con las pruebas de integración (sección 23.3) y de la necesidad de software de pruebas especializado (por ejemplo, resguardos y controladores). La complejidad ciclomática (una métrica de diseño en el nivel de componente) yace en el centro de la prueba de ruta base, un método de diseño de casos de prueba que se presentó en el capítulo 18. Además, la complejidad ciclomática puede usarse para dirigirse a módulos como candidatos para prueba de unidad extensa. Los módulos con alta complejidad ciclomática tienen más probabilidad de ser proclives al error que los módulos donde su complejidad ciclomática es menor. Por esta razón, debe emplear esfuerzo por arriba del promedio para descubrir errores en tales módulos antes de que se integren en un sistema.

### 23.6.1 Métricas de Halstead aplicadas para probar

El esfuerzo de prueba puede estimarse usando métricas derivadas de las medidas de Halstead (sección 23.5). Al usar las definiciones para volumen de programa *V* y nivel de programa *PL*, el esfuerzo *e* de Halstead puede calcularse como

$$PL = \frac{1}{(n_1/2) \times (N_2/n_2)} \tag{23.2a}$$

$$e = \frac{V}{PL} \tag{23.2b}$$

El porcentaje de esfuerzo de prueba global que se va a asignar a un módulo *k* puede estimarse usando la siguiente relación:

$$\text{Porcentaje de esfuerzo de prueba } (k) = \frac{e(k)}{\Sigma e(i)} \tag{23.3}$$

donde $e(k)$ se calcula para el módulo *k* usando las ecuaciones (23.2), y la suma en el denominador de la ecuación (23.3) es la suma del esfuerzo de Halstead a través de todos los módulos del sistema.

### 23.6.2 Métricas para pruebas orientadas a objetos

Las métricas del diseño OO anotadas en la sección 23.3 proporcionan un indicio de la calidad del diseño. También ofrecen un indicio general de la cantidad de esfuerzo de prueba requerido para ejercitar un sistema OO. Binder [Bin94b] sugiere un amplio arreglo de métricas de diseño que tienen influencia directa sobre la "comprobabilidad" de un sistema OO. Las métricas consideran aspectos de encapsulación y herencia.

**Falta de cohesión en métodos (FCOM).**[15] Mientras más alto sea el valor de la FCOM, más estados deben ponerse a prueba para garantizar que los métodos no generan efectos colaterales.


*Las pruebas OO pueden ser bastante complejas. Las métricas pueden ayudarle a dirigir los recursos de prueba en hebras, escenarios y paquetes de clases que son "sospechosas" con base en las características medidas. Úselas.*

**Porcentaje público y protegido (PPP).** Los atributos públicos se heredan de otras clases y, por tanto, son visibles para dichas clases. Los atributos protegidos son accesibles a los métodos en las subclases. Esta métrica indica el porcentaje de los atributos de clase que son públicos o protegidos. Valores altos de PPP aumentan la probabilidad de efectos colaterales entre las clases porque los atributos públicos y protegidos conducen a alto potencial para acoplamiento.[16] Las pruebas deben diseñarse para garantizar el descubrimiento de tales efectos colaterales.

**Acceso público a miembros de datos (APD).** Esta métrica indica el número de clases (o métodos) que pueden acceder a otros atributos de clase, una violación de la encapsulación. Valores altos de APD conducen al potencial de efectos colaterales entre clases. Las pruebas deben diseñarse para garantizar el descubrimiento de tales efectos colaterales.

**Número de clases raíz (NCR).** Esta métrica es un conteo de las distintas jerarquías de clase que se describen en el modelo de diseño. Deben desarrollarse las suites de prueba para cada clase raíz y la correspondiente jerarquía de clase. Conforme el NCR aumenta, también aumenta el esfuerzo de prueba.

***Fan-in* (FIN).** Cuando se usa en el contexto OO, el *fan-in* (abanico de entrada) en la jerarquía de herencia es un indicio de herencia múltiple. FIN > 1 indica que una clase hereda sus atributos y operaciones de más de una clase raíz. FIN > 1 debe evitarse cuando sea posible.

15 Vea la sección 23.3.3 para una descripción de FCOM.

16 Algunas personas promueven diseños sin que alguno de los atributos sea público o privado, es decir, PPP = 0. Esto implica que todos los atributos deben valorarse en otras clases mediante métodos.

**Número de hijos (NDH) y profundidad del árbol de herencia (PAH).**[17] Como se mencionó en el capítulo 19, los métodos de superclase tendrán que volverse a probar para cada subclase.

## 23.7 MÉTRICAS PARA MANTENIMIENTO

Todas las métricas de software presentadas en este capítulo pueden usarse para el desarrollo de nuevo software y para el mantenimiento del software existente. Sin embargo, se han propuesto métricas diseñadas explícitamente para actividades de mantenimiento.

IEEE Std. 982.1-1988 [IEE93] sugiere un *índice de madurez de software* (IMS) que proporcione un indicio de la estabilidad de un producto de software (con base en cambios que ocurran para cada liberación del producto). Para ello, se determina la siguiente información:

$M_T$ = número de módulos en la liberación actual
$F_c$ = número de módulos en la liberación actual que cambiaron
$F_a$ = número de módulos en la liberación actual que se agregaron
$F_d$ = número de módulos de la liberación anterior que se borraron en la liberación actual

El índice de madurez del software se calcula de la forma siguiente:

$$\text{IMS} = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

Conforme el IMS tiende a 1.0, el producto comienza a estabilizarse. El IMS también puede usarse como una métrica para planificar actividades de mantenimiento de software. El tiempo medio para producir una liberación de un producto de software puede correlacionarse con el IMS, y es posible desarrollar modelos empíricos para esfuerzo de mantenimiento.

### HERRAMIENTAS DE SOFTWARE

#### *Métricas de producto*

**Objetivo:** Auxiliar a los ingenieros del software a desarrollar métricas significativas que valoren los productos operativos producidos durante el modelado de análisis y diseño, la generación de código fuente y las pruebas.

**Mecánica:** Las herramientas en esta categoría abarcan un amplio abanico de métricas y se implementan como una aplicación independiente o (más comúnmente) como funcionalidad que existe dentro de las herramientas para análisis y diseño, codificación o pruebas. En la mayoría de los casos, las herramientas de métricas analizan una representación del software (por ejemplo, un modelo UML o código fuente) y desarrollan como resultado una o más métricas.

**Herramientas representativas:**[18]

*Krakatau Metrics*, desarrollada por Power Software (**www.power-software.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 una variedad de métricas arquitectónicas, de diseño y orientadas a código, así como métricas orientadas a proyecto.

*Rational Rose*, distribuida por IBM (**www.304.ibm.com/jct03001c/software/awdtools/developer/rose/**), es un amplio conjunto de herramientas para modelado UML que incorpora algunas características de análisis de métricas.

*RSM*, desarrollada por M-Squared Technologies (**msquaredtechnologies.com/m2rsm/index.html**), calcula una amplia variedad de métricas orientadas a código para C, C++ y Java.

*Understand*, desarrollada por Scientific Toolworks, Inc. (**www.scitools.com**), calcula métricas orientadas a código para varios lenguajes de programación.

17 Vea la sección 23.3.3 para una descripción del NDH y de la PAH.

18 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

## 23.8 Resumen

Las métricas de software proporcionan una forma cuantitativa para valorar la calidad de los atributos internos de producto y, por tanto, permiten valorar la calidad antes de construir el producto. Las métricas proporcionan la comprensión necesaria para crear modelos efectivos de requerimientos y diseño, código sólido y pruebas amplias.

Para ser útil en un contexto de mundo real, una métrica de software debe ser simple y calculable, convincente, congruente y objetiva. Debe ser independiente del lenguaje de programación y ofrecer retroalimentación efectiva.

Las métricas para el modelo de requerimientos se enfocan en los tres componentes del modelo: la función, los datos y el comportamiento. Las métricas para diseño consideran arquitectura, diseño en el nivel de componentes y conflictos en el diseño de interfaz. Las métricas de diseño arquitectónico consideran los aspectos estructurales del modelo de diseño. Las métricas de diseño en el nivel de componente proporcionan un indicio de la calidad del módulo al establecer medidas indirectas para cohesión, acoplamiento y complejidad. Las métricas de diseño de interfaz de usuario ofrecen un indicio de la facilidad con la que puede usarse una GUI. Las métricas *webapp* consideran aspectos de la interfaz de usuario, así como estética, contenido y navegación de la *webapp*.

Las métricas para los sistemas OO se enfocan en mediciones que pueden aplicarse a las características de clase y de diseño (localización, encapsulación, ocultamiento de información, herencia y técnicas de abstracción de objeto) que hacen única a la clase. La suite de métricas CK define seis métricas de software orientado a clase que se enfocan en la clase y en la jerarquía de clase. La suite de métricas también desarrolla métricas para valorar las colaboraciones entre clases y la cohesión en métodos que residen dentro de una clase. En un nivel orientado a clase, la suite de métricas CK puede aumentarse con las métricas propuestas por Lorenz y Kidd y con la suite de métricas MOOD.

Halstead proporciona un interesante conjunto de métricas en el nivel del código fuente. Al usar el número de operadores y operandos presentes en el código, la ciencia del software proporciona una variedad de métricas que pueden usarse para valorar la calidad del programa.

Pocas métricas de producto se han propuesto para uso directo en las pruebas de software y en el mantenimiento. Sin embargo, muchas otras métricas de producto pueden usarse para guiar el proceso de pruebas y como mecanismo para valorar la capacidad de mantenimiento de un programa de cómputo. Para valorar la comprobabilidad de un sistema OO, se ha propuesto una gran variedad de métricas OO.

## Problemas y puntos por evaluar

**23.1.** La teoría de la medición es un tema avanzado que tiene un fuerte engranaje con las métricas de software. Con [Zus97], [Fen91], [Zus90] o fuentes en la web, escriba un breve ensayo que resalte las tesis principales de la teoría de la medición. Proyecto individual: desarrolle una presentación acerca del tema y preséntela en clase.

**23.2.** ¿Por qué no es posible desarrollar una sola métrica exhaustiva para la complejidad de programa o calidad de programa? Intente encontrar una medida o métrica de la vida diaria que viole los atributos de las métricas de software efectivos definidos en la sección 23.1.5.

**23.3.** Un sistema tiene 12 entradas externas, 24 salidas externas, presenta 30 diferentes consultas externas, gestiona 4 archivos lógicos internos y tiene interfaz con 6 diferentes sistemas legados (6 AIE). Todos estos datos son de complejidad promedio y el sistema global es relativamente simple. Calcule PF para el sistema.

**23.4.** El software para System X tiene 24 requerimientos funcionales individuales y 14 requerimientos no funcionales. ¿Cuál es la especificidad de los requerimientos y cuál es la completitud?

**23.5.** Un gran sistema de información tiene 1 140 módulos. Existen 96 módulos que realizan funciones de control y coordinación y 490 módulos cuya función depende del procesamiento previo. El sistema procesa aproximadamente 220 objetos de datos, cada uno de los cuales tiene un promedio de tres atributos. Existen 140 ítems de base de datos únicos y 90 diferentes segmentos de base de datos. Finalmente, 600 módulos tienen puntos de entrada y salida únicos. Calcule el ICED para este sistema.

**23.6.** Una clase **X** tiene 12 operaciones. La complejidad ciclomática se calcula para todas las operaciones en el sistema OO y el valor promedio de la complejidad de módulo es 4. Para la clase **X**, la complejidad para las operaciones de la 1 a la 12 es 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectivamente. Calcule los métodos ponderados por clase.

**23.7.** Desarrolle una herramienta de software que calcule la complejidad ciclomática para un módulo de lenguaje de programación. Puede elegir el lenguaje.

**23.8.** Desarrolle una pequeña herramienta de software que realice análisis de Halstead sobre el código fuente del lenguaje de programación de su elección.

**23.9.** Un sistema legado tiene 940 módulos. La última liberación requirió el cambio de 90 de dichos módulos. Además, se agregaron 40 nuevos módulos y se removieron 12 módulos antiguos. Calcule el índice de madurez de software para el sistema.

## Lecturas y fuentes de información adicionales

Existe un número sorprendentemente grande de libros que se dedican a las métricas de software, aunque la mayoría se enfocan en las métricas de proceso y proyecto, con la exclusión de métricas de producto. Lanza *et al.* (*Object-Oriented Metrics in Practice*, Springer, 2006) analizan métricas OO y su uso para valorar la calidad de un diseño. Genero (*Metrics for Software conceptual Models*, Imperial College Press, 2005) y Ejiogu (*Software Metrics*, BookSurge Publishing, 2005) presentan una amplia variedad de métricas técnicas para casos de uso, modelos UML y otras representaciones de modelado. Hutcheson (*Software Testing fundamentals: Methods and Metrics*, Wiley, 2003) presenta un conjunto de métricas para prueba. Kan (*Metrics and Models in Software Quality Engineering,* Addison-Wesley, 2a. ed., 2002), Fenton y Pfleeger (*Software Metrics: A Rigorous and Practical Approach,* Brooks-Cole Publishing, 1998), y Zuse [Zus97] escribieron tratamientos profundos de las métricas de producto.

Los libros de Card y Glass [Car90], Zuse [Zus90], Fenton [Fen91], Ejiogu [Eji91], Moeller y Paulish (*Software Metrics,* Chapman y Hall, 1993), y Hetzel [Het93] abordan métricas de producto con cierto detalle. Oman y Pfleeger (*Applying Software Metrics*, IEEE Computer Society Press, 1997) editaron una antología de importantes ensayos acerca de las métricas de software.

Ebert *et al.* (*Best Practices in Software Measurement*, Springer, 2004) consideran los métodos para establecer un programa de métricas y los principios subyacentes para la medición del software. Shepperd (*Foundations of Software Measurement*, Prentice-Hall, 1996) también aborda la teoría de medición con cierto detalle. La investigación actual se presenta en los *Proceedings of the Symposium on Software Metrics* (IEEE, publicación anual).

En [IEE93] se presenta un amplio resumen de decenas de métricas de software útiles. En general, un análisis de cada métrica se ha separado en las "primitivas" (medidas) esenciales requeridas para calcular la métrica y las relaciones apropiadas para efectuar el cálculo. Se proporciona análisis y muchas referencias en un apéndice.

Whitmire [Whi97] presenta un tratamiento amplio y matemáticamente sofisticado de las métricas OO. Lorenz y Kidd [Lor94] y Hendersen-Sellers (*Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, 1996) proporcionan tratamientos que se dedican a las métricas OO.

En internet, está disponible una gran variedad de fuentes de información acerca de las métricas del software. Una lista actualizada de referencias en la World Wide Web que son relevantes para la métrica del software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

PARTE

Cuatro

# ADMINISTRACIÓN DE PROYECTOS DE SOFTWARE

En esta parte de *Ingeniería del software. Un enfoque práctico,* aprenderá sobre las técnicas de administración requeridas para planificar, organizar, monitorear y controlar proyectos de software. En los capítulos que siguen se abordan preguntas como las siguientes:

- ¿Cómo debe administrarse el personal, el proceso y el problema durante un proyecto de software?
- ¿Cómo pueden usarse las métricas del software para administrar un proyecto y el proceso de software?
- ¿Cómo genera un equipo de software estimaciones confiables de esfuerzo, costo y duración del proyecto?
- ¿Qué técnicas pueden usarse para valorar los riesgos que pueden tener impacto sobre el éxito del proyecto?
- ¿Cómo selecciona un gerente de proyecto de software un conjunto de tareas laborales para los ingenieros del software?
- ¿Cómo se crea un calendario de proyecto?
- ¿Por qué el mantenimiento y la reingeniería son importantes para los gerentes de ingeniería de software y para los profesionales?

Una vez respondidas estas preguntas, estará mejor preparado para administrar proyectos de software en una forma que lo conducirá a la entrega oportuna de un producto de alta calidad.

CAPÍTULO

# 24 Conceptos de administración de proyecto



En el prefacio de su libro acerca de la administración de proyectos de software, Meiler Page-Jones [pág. 85] hace una afirmación que pueden compartir muchos consultores de ingeniería de software:

> He visitado decenas de tiendas comerciales, tanto buenas como malas, y he observado tableros de datos que procesan los administradores, de nuevo, buenos y malos. Con mucha frecuencia, observé con horror cómo estos administradores luchan infructuosamente a través de proyectos de pesadilla, se retuercen bajo fechas límite imposibles o sistemas entregados que contrarían a sus usuarios y que se dedican a devorar enormes trozos de tiempo de mantenimiento.

Lo que Page-Jones describe son síntomas extraídos de una selección de problemas administrativos y técnicos. Sin embargo, si a los proyectos se les realizara un análisis postmortem, es muy probable que se encuentre un tema común: la administración del proyecto era débil.

En éste y en los siguientes capítulos, se presentan los conceptos clave que conducen a la administración efectiva del proyecto de software. Este capítulo considera los conceptos y principios básicos de la administración de proyectos de software. El capítulo 25 presenta las métricas de proceso y proyecto, base para la toma de decisiones administrativas efectivas. Las técnicas que se usan para estimar el costo se analizan en el capítulo 26. El capítulo 27 ayuda a definir un calendario de proyecto realista. Las actividades administrativas que conducen a mo-

## Una mirada rápida

**¿Qué es?** Aunque muchas personas (en sus momentos más oscuros) toman la visión de Dilbert de la "administración", ésta sigue siendo una actividad muy necesaria cuando se construyen sistemas y productos basados en computadora. La administración del proyecto involucra planificación, monitoreo y control del personal, procesos y acciones que ocurren conforme el software evoluciona desde un concepto preliminar hasta su despliegue operativo completo.

**¿Quién lo hace?** Todo mundo "administra" en cierta medida, pero el ámbito de las actividades administrativas varía entre las personas involucradas en un proyecto de software. Un ingeniero del software administra sus actividades cotidianas, planifica, monitorea y controla las tareas técnicas. Los gerentes de proyecto planifican, monitorean y controlan el trabajo de un equipo de ingenieros de software. Los gerentes ejecutivos coordinan la interfaz entre la empresa y los profesionales del software.

**¿Por qué es importante?** Construir software de computadora es una labor compleja, particularmente si involucra a muchas personas que trabajan durante un tiempo relativamente largo. Por eso es necesario administrar los proyectos de software.

**¿Cuáles son los pasos?** Comprender las cuatro P: personal, producto, proceso y proyecto. El personal debe organizarse para realizar el trabajo de software de manera efectiva. La comunicación con el cliente y con otros participantes debe ocurrir de modo que el ámbito del producto y los requerimientos sean comprensibles. Debe seleccionarse un proceso que sea adecuado para el personal y el producto. El proyecto debe planificarse, estimándose el esfuerzo y el cronograma necesarios para concluir las tareas: definición de los productos operativos, establecimiento de los puntos de verificación de calidad, identificación de mecanismos para monitorear y control del trabajo definido por el plan.

**¿Cuál es el producto final?** En cuanto inician las actividades de administración, se produce un plan de proyecto que define el proceso y las tareas que se van a realizar, el personal que hará el trabajo y los mecanismos que se emplearán para valorar riesgos, controlar el cambio y evaluar la calidad.

**¿Cómo me aseguro de que lo hice bien?** Nunca se está completamente seguro de que el plan del proyecto es correcto, hasta que se entrega un producto de alta calidad a tiempo y dentro del presupuesto. Sin embargo, un gerente de proyecto lo hace bien cuando alienta al personal del software a trabajar en conjunto, como un equipo efectivo, y cuando enfoca su atención en las necesidades del cliente y en la calidad del producto.

nitoreo, mitigación y administración efectiva del riesgo se presentan en el capítulo 28. Finalmente, el capítulo 29 considera el mantenimiento y la reingeniería, y estudia los conflictos administrativos que se encontrarán cuando lidie con sistemas heredados.

## 24.1 El espectro administrativo

La administración efectiva de un proyecto de software se enfoca en las cuatro P: personal, producto, proceso y proyecto. El orden no es arbitrario. El gerente que olvida que el trabajo de la ingeniería del software es una empresa intensamente humana nunca triunfará en la administración del proyecto. Un gerente que fracase en alentar una comunicación comprensiva con los participantes durante las primeras etapas de la evolución de un producto se arriesga a construir una solución elegante para el problema equivocado. El gerente que ponga poca atención al proceso corre el riesgo de insertar métodos y herramientas técnicos competentes pero en el vacío. Aquel que se embarque sin un plan sólido pone en peligro el éxito del proyecto.

### 24.1.1 El personal

Desde la década de 1960 se estudia la formación de personal de software motivado y enormemente calificado. De hecho, el "factor humano" es tan importante que el Software Engineering Institute desarrolló un *Modelo de madurez de capacidades del personal* (People-CMM, por sus siglas en inglés), en reconocimiento al hecho de que "toda organización requiere mejorar continuamente su habilidad para atraer, desarrollar, motivar, organizar y conservar la fuerza de trabajo necesaria a fin de lograr sus objetivos empresariales estratégicos" [Cur01].

El People-CMM define las siguientes áreas prácticas clave para el personal de software: plantilla, comunicación y coordinación, ambiente de trabajo, desempeño administrativo, capacitación, compensación, análisis y desarrollo de competencias, desarrollo profesional, desarrollo de grupo de trabajo y desarrollo de equipo/cultura, entre otros. Las organizaciones que conforme a este modelo logran altos niveles de madurez de capacidades de personal tienen una probabilidad muy elevada de alcanzar la implementación de prácticas administrativas efectivas en los proyectos de software.

El People-CMM es un compañero de la *Integración del modelo de madurez de capacidades del software* (capítulo 30), que guía a las organizaciones en la creación de un proceso de software maduro. Los conflictos asociados con la administración del personal y con la estructura para los proyectos de software se consideran más adelante, en este capítulo.

### 24.1.2 El producto

Antes de poder planear un proyecto, deben establecerse los objetivos y el ámbito del producto, considerarse soluciones alternativas e identificar las restricciones técnicas y administrativas. Sin esta información, es imposible definir estimaciones razonables (y precisas) del costo, una valoración efectiva del riesgo, una descomposición realista de las tareas del proyecto y un calendario de proyecto manejable que proporcione en cada momento un indicio significativo del progreso.

Como desarrolladores de software, todos los participantes deben reunirse 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 empresarial y continúa como el primer paso en la ingeniería de requerimientos del software (capítulo 5). Los objetivos identifican las metas globales para el producto (desde el punto de vista de los participantes) sin considerar cómo se lograrán estas metas. El ámbito identifica los datos, funciones y comportamientos principales que caracterizan al producto y, más importante, intenta ligar dichas características en forma cuantitativa.

Una vez comprendidos los objetivos y el ámbito del producto, se consideran soluciones alternativas. Aunque se analizan muy pocos detalles, las alternativas permiten a los gerentes y profesionales seleccionar un "mejor" enfoque, dadas las restricciones impuestas por fechas de entrega, restricciones presupuestales, disponibilidad de personal, interfaces técnicas y muchos otros factores.

### 24.1.3 El proceso


*Quienes se adhieren a la filosofía de proceso ágil (capítulo 3) argumentan que su proceso es más esbelto que otros. Esto puede ser cierto, pero todavía tienen un proceso, y la ingeniería de software ágil todavía requiere disciplina.*

Un proceso de software (capítulos 2 y 3) proporciona el marco conceptual desde el cual puede establecerse un plan completo para el desarrollo de software. Un pequeño número de actividades de marco conceptual se aplica a todos los proyectos de software, sin importar su tamaño o complejidad. Algunos conjuntos de diferentes tareas (tareas, hitos, productos operativos y puntos de aseguramiento de calidad) permiten que las actividades del marco conceptual se adapten a las características del proyecto de software y a los requerimientos del equipo del proyecto. Finalmente, las actividades sombrilla (como el aseguramiento de la calidad del software, la administración de configuración del software y las mediciones) recubren el modelo de proceso. Las actividades sombrilla son independientes de cualquier actividad del marco conceptual y ocurren a lo largo del proceso.

### 24.1.4 El proyecto

Los proyectos de software se planean y controlan debido a una razón principal: es la única forma conocida para manejar la complejidad. E incluso así, los equipos de software todavía batallan. En un estudio de 250 grandes proyectos de software desarrollados entre 1998 y 2004, Capers Jones [Jon04] encontró que "alrededor de 25 se consideraron exitosos por haber logrado sus objetivos de calendario, costo y calidad. Aproximadamente 50 tuvieron demoras o excesos por abajo de 35 por ciento, mientras que más o menos 175 experimentaron grandes demoras y excesos, o se dieron por concluidos sin completarse". Aunque actualmente la tasa de éxito para los proyectos de software puede haber mejorado un poco, la tasa de falla de proyecto sigue siendo mucho más alta de lo que debiera.[1]

Para evitar el fracaso del proyecto, un gerente de proyecto de software y los ingenieros de software que construyan el producto deben evitar un conjunto de señales de advertencia comunes, entender los factores de éxito cruciales que conducen a una buena administración del proyecto y desarrollar un enfoque de sentido común para planificar, monitorear y controlar el proyecto. Cada uno de estos temas se estudia en la sección 24.5 y en los capítulos que siguen.

## 24.2 El personal

En un estudio publicado por el IEEE [Cur88], se preguntó a los vicepresidentes de ingeniería de tres grandes compañías tecnológicas cuál era el elemento más importante para el éxito de un proyecto de software. Ellos respondieron de la siguiente manera:

**VP 1:** Supongo que, si tienes que elegir una cosa que sea la más importante en nuestro ambiente, diría que no son las herramientas que usamos, es el personal.

**VP 2:** El ingrediente más importante que fue exitoso en este proyecto fue tener gente inteligente [...] en mi opinión, muy pocas cosas más importan [...] La cosa más importante que

1 Con estas estadísticas, es razonable preguntar cómo sigue creciendo exponencialmente el impacto de las computadoras. Parte de la respuesta es que un número sustancial de estos proyectos "fallidos" estuvieron mal concebidos desde el inicio. Los clientes pierden interés rápidamente (porque lo que pidieron en realidad no era tan importante como pensaron la primera vez) y los proyectos se cancelan.

haces para un proyecto es seleccionar al personal [...] El éxito de la organización de desarrollo de software está muy, muy asociada con la habilidad para reclutar buen personal.

**VP 3:** La única regla que tengo en la administración es asegurarme de que tengo buen personal, gente realmente buena, y que hago crecer gente buena y que proporciono un ambiente en el que la gente buena puede producir.

De hecho, éste es un testimonio convincente acerca de la importancia del personal en el proceso de ingeniería de software. Y aún así, para la mayoría de las personas, desde los vicepresidentes ejecutivos de ingeniería hasta el profesional en el nivel más bajo, con frecuencia dan por hecho al personal. Los administradores argumentan (como lo hizo el grupo anterior) que las personas son lo importante, pero sus acciones en ocasiones contradicen sus palabras. En esta sección se examina a las personas que participan en el proceso de software y la forma en la que se organizan para realizar ingeniería de software efectiva.

### 24.2.1 Los participantes

El proceso de software (y todo proyecto de software) está poblado de participantes, quienes pueden organizarse en alguna de las siguientes áreas:

1. *Gerentes ejecutivos*, quienes definen los temas empresariales que con frecuencia tienen una influencia significativa sobre el proyecto.
2. *Gerentes de proyecto (técnicos)*, quienes deben planificar, motivar, organizar y controlar a los profesionales que hacen el trabajo de software.
3. *Profesionales* que aportan las habilidades técnicas que se necesitan para someter a ingeniería un producto o aplicación.
4. *Clientes* que especifican los requerimientos para el software que se va a fabricar, así como otros participantes que tienen un interés periférico en el resultado.
5. *Usuarios finales*, quienes interactúan con el software una vez que se libera para su uso productivo.

Todo proyecto de software está poblado con personas que están dentro de esta taxonomía.[2] Para ser efectivo, el equipo de software debe organizarse de manera que maximice las habilidades y capacidades de cada persona. Y ésta es labor del líder del equipo.

### 24.2.2 Líderes de equipo

La administración del proyecto es una actividad que implica mucho trato con la gente; por esta razón, los profesionales competentes tienen con frecuencia pobre desempeño como líderes de equipo. Simplemente, no tienen la mezcla justa de habilidades personales. Y aún así, como Edgemon afirma: "Por desgracia, y por muy frecuente que parezca, los individuos simplemente se topan con el papel de gerente de proyecto y se convierten en gerentes accidentales de proyecto" [Edg95].

En un excelente libro acerca del liderazgo técnico, Jerry Weinberg [Wei86] sugiere un modelo MOI de liderazgo:

**? ¿Qué se busca cuando se elige a alguien como líder de un proyecto de software?**

**Motivación.** Habilidad para alentar (mediante "empuje o jalón") al personal técnico a producir a su máxima capacidad.

**Organización.** Habilidad para moldear los procesos existentes (o inventar nuevos) que permitirán que el concepto inicial se traduzca en un producto final.

2 Cuando se desarrollan *webapps*, personal no técnico puede involucrarse en la creación de contenido.

**Ideas o innovación.** Habilidad para alentar a las personas a crear y sentirse creativas, aun cuando deban trabajar dentro de fronteras establecidas para un producto o aplicación de software particular.

Weinberg sugiere que los líderes de proyecto exitosos aplican un estilo administrativo de resolución de problemas. Es decir, un gerente de proyecto de software debe concentrarse en comprender el problema que se va a resolver, en administrar el flujo de ideas y, al mismo tiempo, en dejar que todos en el equipo sepan (por medio de palabras y, mucho más importante, con acciones) que la calidad cuenta y que no se comprometerá.

**Cita:**

"En términos más simples, un líder es aquel que sabe a dónde quiere ir, y se levanta y marcha."

**John Erskine**

Otra visión [Edg95] de las características que definen a un gerente de proyecto eficaz enfatiza cuatro rasgos clave:

**Resolución de problemas.** Un gerente de proyecto de software eficaz puede diagnosticar los conflictos técnicos y organizativos que son más relevantes, estructura sistemáticamente una solución o motiva adecuadamente a otros profesionales para desarrollarla, aplica lecciones aprendidas de proyectos pasados a situaciones nuevas y sigue siendo suficientemente flexible para cambiar de dirección si los intentos por resolver el problema son infructuosos.

**Identidad administrativa.** Un buen gerente de proyecto debe hacerse cargo del mismo. Debe tener la confianza para asumir el control cuando sea necesario y asegurarse de permitir que el buen personal técnico siga sus instintos.

**Logro.** Un gerente competente debe recompensar la iniciativa y el logro para optimizar la productividad de un equipo de proyecto. Debe demostrar mediante sus acciones que no se castigará del correr riesgos de manera controlada.

**Influencia y construcción del equipo.** Un gerente de proyecto eficaz debe poder "leer" a la gente; debe poder comprender las señales verbales y no verbales, y reaccionar ante las necesidades de las personas que envían estas señales. El gerente debe permanecer bajo control en situaciones de alto estrés.

### 24.2.3 El equipo de software

**Cita:**

"No todo grupo es un equipo y no todo equipo es eficaz."

**Glenn Parker**

Existen casi tantas estructuras organizativas humanas para el desarrollo del software como organizaciones que lo desarrollan. Para bien o para mal, la estructura organizativa no puede modificarse fácilmente. La preocupación por las consecuencias prácticas y por las políticas del cambio organizativo no está dentro del ámbito de responsabilidad del gerente del proyecto de software. Sin embargo, la organización de las personas directamente involucradas en un nuevo proyecto de software está dentro del campo de acción del gerente del proyecto.

La "mejor" estructura de equipo depende del estilo administrativo de la organización, del número de personas que formarán el equipo y de sus niveles de habilidad, así como de la dificultad global del problema. Mantei [Man81] describe siete factores de proyecto que deben considerarse cuando se planee la estructura de los equipos de ingeniería de software:

**? ¿Qué factores deben considerarse cuando se elige la estructura de un equipo de software?**

- Dificultad del problema que se va a resolver.
- "Tamaño" del programa resultante en líneas de código o puntos de función.
- Tiempo que el equipo permanecerá unido (vida del equipo).
- Grado en el que puede dividirse en módulos el problema.
- Calidad y confiabilidad requeridas por el sistema que se va a construir.
- Rigidez de la fecha de entrega.
- Grado de sociabilidad (comunicación) requerido para el proyecto.

Constantine [Con93] sugiere cuatro "paradigmas organizacionales" para los equipos de ingeniería de software:

**¿Qué opciones se tienen cuando se define la estructura de un equipo de software?**

1. Un *paradigma cerrado* estructura un equipo conforme a una jerarquía de autoridad tradicional. Tales equipos pueden trabajar bien cuando producen software muy similar al de esfuerzos anteriores, pero será menos probable que sean innovadores cuando trabajen dentro de este paradigma.
2. Un *paradigma aleatorio* estructura un equipo de manera holgada y depende de la iniciativa individual de los miembros del equipo. Cuando se requiere innovación o avance tecnológico, destacarán los equipos que siguen este paradigma, pero pueden batallar cuando se requiera "desempeño ordenado".
3. Un *paradigma abierto* intenta estructurar un equipo de manera que logre algunos de los controles asociados con el paradigma cerrado, pero también mucha de la innovación que ocurre cuando se usa el paradigma aleatorio. El trabajo se realiza de manera colaboradora; la gran comunicación y la toma de decisiones consensuadas constituyen las características de los equipos de paradigma abierto. Las estructuras de equipo de este paradigma son muy adecuadas para la solución de problemas complejos, pero pueden no desempeñarse tan eficazmente como otros equipos.
4. Un *paradigma síncrono* se apoya en la compartimentalización natural de un problema y organiza a los miembros del equipo para trabajar en trozos del problema con poca comunicación activa entre ellos.

**Cita:**

"Si quieres ser incrementalmente mejor: sé competitivo. Si quieres ser exponencialmente mejor: sé cooperativo."

**Autor desconocido**

Como acotación histórica, cabe decir que una de las primeras organizaciones de equipo de software fue una estructura de paradigma cerrado originalmente llamado *equipo de programador jefe*. Esta estructura la propuso por primera ocasión Harlan Mills y la describió Baker [Bak72]. El núcleo del equipo estaba compuesto de: un *ingeniero ejecutivo* (el programador jefe), quien planeaba, coordinaba y revisaba todas las actividades técnicas del equipo; *personal técnico* (por lo general de dos a cinco personas), quienes realizaban análisis y desarrollaban actividades; y un *ingeniero de respaldo*, quien apoyaba al ingeniero ejecutivo en sus actividades y podía sustituirlo con mínima pérdida en la continuidad del proyecto. El programador jefe puede auxiliarse con uno o más especialistas (por ejemplo, experto en telecomunicaciones, diseñador de bases de datos), personal de apoyo (por ejemplo, escritores técnicos, oficinistas) y un bibliotecario de software.

Como contrapunto a la estructura del equipo del programador jefe, el paradigma aleatorio de Constantine [Con93] sugiere un equipo de software con independencia creativa cuyo enfoque para trabajar pueda denominarse de mejor manera como *anarquía innovadora*. Aunque el enfoque de espíritu libre en el trabajo de software es atractivo, canalizar la energía creativa hacia un equipo de alto rendimiento debe ser una meta central de una organización de ingeniería de software. Para lograr un equipo de alto rendimiento:

- Los miembros del equipo deben tenerse confianza entre sí.
- La distribución de habilidades debe ser adecuada para el problema.
- Es posible que tenga que excluirse del equipo a los inconformes si debe mantenerse la cohesión del equipo.

Sin importar el tipo de organización del equipo, el objetivo para todo gerente de proyecto es ayudar a crear un equipo que muestre cohesión. En su libro *Peopleware*, DeMarco y Lister [DeM98] analizan este tema:

> Tendemos a usar la palabra equipo con mucha holgura en el mundo empresarial, y llamamos así a cualquier grupo de personas asignadas para trabajar juntas. Pero muchos de estos grupos simple-

**¿Qué es un equipo "cuajado"?**

mente no parecen equipos. No tienen una definición común de éxito o algún espíritu de equipo identificable. Lo que falta es un fenómeno que llamamos *cuajar*.

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 cuajarse, la probabilidad de éxito va hacia arriba. El equipo puede volverse imparable, una fuerza arrasadora para el éxito [...] No necesita ser administrado en la forma tradicional y, ciertamente, no necesita ser motivado. Adquiere cantidad de movimiento.

DeMarco y Lister sostienen que los miembros de los equipos cuajados son significativamente más productivos y más motivados que el promedio. Comparten una meta común, una cultura común y en muchos casos un "sentido de élite" que los hace únicos.

**¿Por qué fallan los equipos cuando deben cuajar?**

Pero no todos los equipos cuajan. De hecho, muchos equipos sufren de lo que Jackman [Jac98] llama "toxicidad de equipo". Ella define cinco factores que "fomentan un ambiente de equipo potencialmente tóxico": 1) una atmósfera de trabajo frenético, 2) alta frustración que causa fricción entre los miembros del equipo, 3) un proceso de software "fragmentado o pobremente coordinado", 4) una definición poco clara de los roles en el equipo de software y 5) "continua y repetida exposición al fracaso".

**Cita:**

"Hacer o no hacer. No hay intento."

Yoda, de *La guerra de las galaxias*

Para evitar un ambiente de trabajo frenético, el gerente del proyecto debe estar seguro de que el equipo tiene acceso a toda la información requerida para hacer el trabajo y de que las metas y objetivos principales, una vez definidos, no deben modificarse a menos que sea absolutamente necesario. Un equipo de software puede evitar la frustración si se le da tanta responsabilidad para la toma de decisiones como sea posible. Un proceso inadecuado (por ejemplo, tareas innecesarias o abrumadoras o productos operativos pobremente elegidos) puede evitarse al entender el producto que se va a construir y a las personas que hacen el trabajo, así como al permitir al equipo seleccionar el modelo de proceso. El equipo mismo debe establecer sus propios mecanismos de responsabilidad (las revisiones técnicas[3] son una excelente forma de lograr esto) y definir una serie de enfoques correctos cuando un miembro del equipo tiene fallos en el desempeño. Finalmente, la clave para evitar una atmósfera de fracaso radica en establecer técnicas basadas en equipo para retroalimentarse y resolver problemas.

Además de las cinco toxinas descritas por Jackman, un equipo de software con frecuencia batalla con los diferentes rasgos humanos de sus miembros. Algunos son extrovertidos; otros, introvertidos. Algunas personas reúnen información de manera intuitiva y separan los conceptos abarcadores de los hechos dispares. Otras procesan la información de manera lineal, y reúnen y organizan detalles minúsculos de los datos proporcionados. Ciertos miembros del equipo se sienten cómodos al tomar decisiones sólo cuando se presenta un argumento lógico y ordenado. Otros son intuitivos y quieren tomar decisiones con base en "corazonadas". Algunos profesionales quieren calendarios detallados poblados de tareas organizadas que les permitan lograr el cierre para algún elemento de un proyecto. Otros prefieren un ambiente más espontáneo en el que los temas abiertos sean bien vistos. Algunos trabajan duro para hacer que las cosas estén listas mucho antes de una fecha final y, por tanto, evitan el estrés conforme la fecha se aproxima, mientras que otros se sienten energizados por la adrenalina que produce una entrega de último minuto. Una discusión detallada de la psicología de estos rasgos y de las formas en las que el líder del equipo habilidoso puede ayudar a las personas con rasgos opuestos para trabajar en conjunto está más allá del ámbito de este libro.[4] Sin embargo, es importante observar que el reconocimiento de las diferencias humanas es el primer paso hacia la creación de equipos que cuajen.

3 Las revisiones técnicas se estudian con detalle en el capítulo 15.

4 Una excelente introducción a estos temas, en su relación con los equipos de proyecto de software, puede encontrarse en [Fer98].

### 24.2.4 Equipos ágiles

Durante las décadas pasadas, el desarrollo de software ágil (capítulo 3) se ha sugerido como antídoto a muchos de los problemas que plagan el trabajo en un proyecto de software. Cabe recordar que la filosofía ágil alienta la satisfacción del cliente y la entrega incremental temprana del software, así como pequeños equipos de proyecto enormemente motivados, métodos informales, mínimos productos operativos de ingeniería de software y simplicidad de desarrollo global.

El pequeño equipo de trabajo enormemente motivado, también llamado *equipo ágil*, adopta la mayoría de las características de los equipos de proyecto de software exitosos que se estudiaron en la sección anterior y evita muchas de las toxinas que crean problemas. No obstante, la filosofía ágil subraya la competencia individual (miembro de equipo), acoplada con la colaboración grupal como factores de éxito vitales para el equipo. Cockburn y Highsmith [Coc01a] observan esto cuando escriben:

> Si el personal del proyecto es suficientemente bueno, puede usar casi cualquier proceso y lograr esta asignación. Si no lo es, ningún proceso reparará su inadecuación: "personal mata proceso" es una forma de decirlo. Sin embargo, la falta de apoyo de usuarios y ejecutivos puede matar al proyecto: "política mata personal". El apoyo inadecuado puede impedir que incluso el personal bueno logre esta tarea.


Un equipo ágil es un equipo autoorganizado que tiene autonomía para planificar y tomar decisiones técnicas.

Para hacer uso efectivo de las competencias de cada miembro del equipo y fomentar la colaboración efectiva a través de un proyecto de software, los equipos ágiles son *autoorganizados*. Un equipo autoorganizado no necesariamente mantiene una sola estructura de equipo, sino que usa elementos de los paradigmas aleatorio, abierto y síncrono de Constantine, estudiados en la sección 24.2.3.

Muchos modelos de proceso ágil (por ejemplo, Scrum) dan al equipo ágil significativa autonomía para tomar las decisiones administrativas y técnicas del proyecto necesarias para hacer que el trabajo se cumpla. La planificación se mantiene al mínimo y al equipo se le permite seleccionar su propio enfoque (por ejemplo, proceso, métodos, herramientas), restringido únicamente por los requerimientos empresariales y los estándares de la organización. Conforme avanza el proyecto, el equipo se autoorganiza para enfocarse en la competencia individual, de manera que ésta sea más benéfica para el proyecto en un momento determinado. Para lograr esto, un equipo ágil puede realizar reuniones grupales diarias para coordinar y sincronizar el trabajo que debe realizarse en ese día.

Con base en la información obtenida durante dichas reuniones, el equipo adapta su enfoque para lograr un incremento de trabajo. Conforme transcurre cada día, la autoorganización y la colaboración continuas mueven al equipo hacia un incremento de software completo.

### 24.2.5 Conflictos de coordinación y comunicación


"La propiedad colectiva no es más que una ilustración de la idea de que los productos deben atribuirse al equipo [ágil], no a los individuos que constituyen el equipo."

**Jim Highsmith**

Existen muchas razones por las que los proyectos de software tienen problemas. 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 da como resultado un torrente continuo de cambios que detienen al equipo de proyecto. La interoperabilidad se ha convertido en una característica clave de muchos sistemas. El software nuevo debe comunicarse con el software existente y ajustarse a las restricciones predefinidas impuestas por el sistema o por el producto.

Tales características del software moderno (escala, incertidumbre e interoperabilidad) son hechos de la vida. Para lidiar con ellos de manera efectiva, deben implantarse métodos efectivos a fin de coordinar al personal que hace el trabajo. Esto se logra estableciendo mecanismos para la comunicación formal e informal entre los miembros del equipo y entre los distintos equipos. La comunicación formal se consigue mediante "comunicación escrita, reuniones estructuradas y otros canales de comunicación relativamente no interactivos e impersonales" [Kra95]. La co-

municación 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.

**CASASEGURA**

### *Estructura del equipo*

**La escena:** Oficina de Doug Miller antes de iniciar el proyecto de software *CasaSegura*.

**Personajes**: Doug Miller (gerente del equipo de ingeniería de software *CasaSegura*) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

**La conversación:**

**Doug:** ¿Han tenido oportunidad de consultar la información preliminar que preparó mercadotecnia acerca de *CasaSegura*?

**Vinod (asiente y observa a sus compañeros de equipo):** Sí. Pero tenemos muchas preguntas.

**Doug:** Dejemos eso por un momento. Me gustaría hablar acerca de cómo vamos a estructurar el equipo, quién es responsable de qué...

**Jamie:** Yo estoy totalmente a favor de la filosofía ágil, Doug. Creo que debemos ser un equipo autoorganizado.

**Vinod:** Estoy de acuerdo. Dada la apretada línea de tiempo y algo de la incertidumbre, así como el hecho de que todos somos realmente competentes [risas], ésta parece ser la forma correcta de avanzar.

**Doug:** Está bien por mí, pero ustedes conocen las instrucciones.

**Jamie (sonríe y habla como si recitara algo):** "Tomamos decisiones tácticas acerca de quién hace qué y cuándo, pero es nuestra responsabilidad sacar el producto a tiempo."

**Vinod:** Y con calidad.

**Doug:** Exactamente. Pero recuerden que hay restricciones. Mercadotecnia define los incrementos del software que se va a producir... consultándonos, desde luego.

**Jamie:** ¿Y?

**Doug:** Y vamos a usar UML como nuestro enfoque de modelado.

**Vinod:** Pero mantendremos la documentación extraña en un mínimo absoluto.

**Doug:** ¿Quién es el enlace conmigo?

**Jamie:** Decidimos que Vinod sea el líder técnico; tiene más experiencia, así que Vinod es tu enlace, pero siéntete en libertad de hablar con cualquiera de nosotros.

**Doug (ríe):** No se preocupen. Lo haré.

## 24.3 EL PRODUCTO

Un gerente de proyecto de software se enfrenta con un dilema en el comienzo mismo de un proyecto de software. Se requieren estimaciones cuantitativas y un plan organizado, pero no hay información sólida disponible. Un análisis detallado de los requerimientos del software proporcionaría la información necesaria para las estimaciones, pero el análisis usualmente tarda semanas o incluso meses en completarse. Peor aún, los requerimientos pueden ser fluidos y cambiar con regularidad conforme avanza el proyecto. Y, sin embargo, ¡se necesita un plan "ahora"!

Le guste o no al gerente de proyecto, debe examinar el producto; se pretende que el problema se resuelva desde el principio mismo del proyecto; cuando menos, debe establecer y acotar el ámbito del producto.

### 24.3.1 Ámbito del software

La primera actividad en la administración del proyecto de software es determinar el *ámbito del software*, que se define al responder las siguientes preguntas:


*Si no puede acotar una característica del software que intenta construir, mencione la característica como un riesgo del proyecto (capítulo 25).*

**Contexto.** ¿Cómo encaja en un sistema, producto o contexto empresarial más grande el software que se va a construir y qué restricciones se imponen como resultado del contexto?

**Objetivos de información.** ¿Qué objetos de datos visibles para el cliente se producen como salida del software? ¿Qué objetos de datos se requieren como entrada?

**Función y desempeño.** ¿Qué función realiza el software para transformar los datos de entrada en salida? ¿Existe alguna característica de desempeño especial que deba abordarse?

El ámbito del proyecto de software no debe tener ambigüedades ni ser incomprensible en los niveles administrativo y técnico. Debe acotar un enunciado del ámbito del software; es decir, los datos cuantitativos (por ejemplo, número de usuarios simultáneos, entorno objetivo, máximo tiempo de respuesta permisible) se enuncian de manera explícita, se anotan las restricciones y/o limitaciones (por ejemplo, el costo del producto restringe el tamaño de la memoria) y se describen los factores mitigantes (por ejemplo, los algoritmos deseados están bien entendidos y disponibles en Java).

### 24.3.2 Descomposición del problema


*Para desarrollar un plan de proyecto razonable, debe descomponer el problema. Esto puede lograrse usando una lista de funciones o con casos de uso.*

La descomposición del problema, en ocasiones llamada *división* o *elaboración del problema*, es una actividad que se asienta en el centro del análisis de requerimientos del software (capítulos 6 y 7). Durante la actividad de determinación del ámbito, no se hacen intentos por descomponer completamente el problema. En vez de ello, la descomposición se aplica en dos áreas principales: 1) la funcionalidad y el contenido (información) que deben entregarse y 2) el proceso que se usará para entregarlo.

Los seres humanos tienden a aplicar una estrategia de "divide y vencerás" cuando se enfrentan a un problema complejo. Dicho de manera simple, un problema complejo se divide en problemas más pequeños que son más manejables. Ésta es la estrategia que se aplica conforme comienza la planeación del proyecto. Las funciones del software, descritas en el enunciado del ámbito, se evalúan y refinan para proporcionar más detalle antes de comenzar la estimación (capítulo 26). Puesto que tanto las estimaciones de costo como las de calendario se orientan funcionalmente, con frecuencia es útil cierto grado de descomposición. De igual modo, los principales objetos de contenido o datos se descomponen en sus partes constituyentes, lo que proporciona una comprensión razonable de la información que se va a producir con el software.

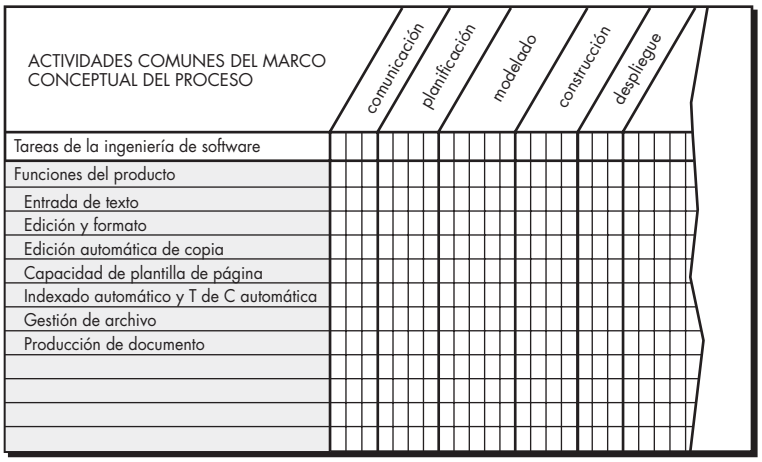
Tome como ejemplo un proyecto que construirá un nuevo producto de procesamiento de palabras. Entre las características únicas del producto, están la entrada continua de voz, así como un teclado virtual a través de una pantalla táctil, características extremadamente sofisticadas de "edición de copia automática", capacidad de plantilla de página, indexado automático y tabla de contenidos automática. El gerente del proyecto debe establecer primero un enunciado del ámbito que acote dichas características (y otras funciones más comunes, tales como edición, gestión de archivos y producción de documentos). Por ejemplo, ¿la entrada continua de voz requiere que el usuario "entrene" al producto? Específicamente: ¿qué capacidades ofrecerá el editor de copia? ¿Cuán sofisticadas serán las capacidades de la plantilla de página?, ¿éstas abarcarán las capacidades que requiere una pantalla táctil?

Conforme avanza la determinación de ámbito, ocurre de manera natural un primer nivel de división. El equipo de proyecto aprende que el departamento de mercadotecnia habló con los clientes potenciales y descubrió que las siguientes funciones deben ser parte de la edición automática de copia: 1) corrector de vocabulario, 2) corrector gramatical, 3) comprobación de referencias para documentos grandes (por ejemplo, ¿una referencia a una entrada bibliográfica se encuentra en la lista de entrada en la bibliografía?), 4) implementación de una característica de hoja de estilo que imponga consistencia a través de un documento y 5) validación de referencias de sección y capítulo para documentos grandes. Cada una de estas características representa una subfunción por implementar en el software. Cada una puede refinarse aún más si la descomposición hace más sencilla la planificación.

## 24.4 El proceso

Las actividades del marco conceptual (capítulo 2) que caracterizan al proceso de software son aplicables a todos los proyectos de software. El problema es seleccionar el modelo de proceso que sea adecuado para el software que el equipo del proyecto someterá a ingeniería.

**FIGURA 24.1**
**Fusión de problema y proceso**




El equipo debe decidir qué modelo de proceso es más adecuado: 1) para los clientes que solicitaron el producto y el personal que hará el trabajo, 2) para las características del producto en sí y 3) para el entorno de proyecto donde trabaja el equipo de software. Cuando se selecciona un modelo de proceso, el equipo define entonces un plan de proyecto preliminar con base en el conjunto de actividades del marco conceptual del proceso. Una vez establecido el plan preliminar, comienza la descomposición del proceso, es decir, debe crearse un plan completo que refleje las tareas laborales requeridas para poblar las actividades del marco conceptual. Estas actividades se exploran brevemente en las secciones que siguen; en el capítulo 26 se presenta una visión más detallada.

### 24.4.1 Fusión de producto y proceso

La planificación del proyecto comienza con la fusión de producto y proceso. Cada función que se va a someter a ingeniería por parte del equipo debe pasar a través del conjunto de actividades de marco conceptual que defina la organización de software.

Suponga que la organización adoptó las actividades genéricas del marco conceptual que se estudiaron en el capítulo 2: **comunicación**, **planificación**, **modelado**, **construcción** y **despliegue**. Los miembros del equipo que trabajen en una función del producto aplicarán en ella cada una de las actividades del marco conceptual. En esencia, se crea una matriz similar a la que se muestra en la figura 24.1. Cada función de producto principal (las funciones anotadas con números para el software de procesamiento de palabra estudiadas anteriormente) se menciona en la columna izquierda. Las actividades de marco conceptual se mencionan en la fila superior. Las tareas del trabajo de la ingeniería de software (para cada actividad del marco conceptual) ingresarán en la fila siguiente.[5] La labor del gerente de proyecto (y otros miembros del equipo) es estimar los requerimientos de recurso para cada celda de la matriz, fechas de inicio y término de las tareas asociadas con cada celda, y los productos operativos que se van a producir como consecuencia de cada tarea. Dichas actividades se consideran en el capítulo 26.

### 24.4.2 Descomposición del proceso

Un equipo de software debe tener un grado significativo de flexibilidad al elegir el modelo de proceso de software que es mejor para el proyecto y las tareas de la ingeniería de software que

5 Obsérvese que las tareas deben adaptarse a las necesidades específicas del proyecto, con base en algunos criterios de adaptación.

PUNTO CLAVE

El marco conceptual del proceso establece un esqueleto para la planificación del proyecto y se adapta para abarcar un conjunto de tareas que son adecuadas para el proyecto.

pueblen el modelo de proceso una vez elegido. Un proyecto relativamente pequeño que sea similar a esfuerzos anteriores puede lograrse mejor al usar el enfoque secuencial lineal. Si la fecha límite es tan apretada como para que toda la funcionalidad no pueda entregarse razonablemente, puede ser mejor una estrategia incremental. De igual modo, los proyectos con otras características (por ejemplo, requerimientos de incertidumbre, tecnología innovadora, clientes difíciles, significativo potencial de reuso) conducirán a la selección de otros modelos de proceso.[6]

Una vez elegido el modelo de proceso, el marco conceptual del proceso se adapta a él. En todo caso, puede usarse el marco conceptual genérico de proceso que se estudió anteriormente. Funcionará para los modelos lineales, para modelos iterativos e incrementales, para modelos evolutivos e incluso para modelos concurrentes o de ensamble de componentes. El marco conceptual del proceso es invariante y sirve como la base para todo el trabajo que realiza una organización de software.

Pero las tareas del trabajo real sí varían. La descomposición del proceso comienza cuando el gerente de proyecto pregunta: ¿cómo logramos esta actividad del marco conceptual? Por ejemplo, un proyecto simple y relativamente pequeño puede requerir las siguientes tareas para la actividad de comunicación:

1. Desarrollar lista de clarificación de conflictos.
2. Reunirse con los participantes para abordar la clarificación de conflictos.
3. Desarrollar en conjunto un enunciado del ámbito.
4. Revisar el enunciado del ámbito con todos los interesados.
5. Modificar el enunciado del ámbito según se requiera.

Estos eventos pueden ocurrir durante un periodo de menos de 48 horas. Representan una descomposición de proceso que es adecuada para el pequeño proyecto relativamente simple.

Ahora, considere un proyecto más complejo, que tenga un ámbito más amplio e impacto empresarial más significativo. Tal proyecto puede requerir las siguientes tareas para la **comunicación**:

1. Revisar la solicitud del cliente.
2. Planificar y calendarizar una reunión formal facilitada con todos los participantes.
3. Realizar investigación para especificar la solución propuesta y los enfoques existentes.
4. Preparar un "documento de trabajo" y una agenda para la reunión formal.
5. Realizar la reunión.
6. Desarrollar conjuntamente miniespecificaciones que reflejen las características de datos, funcionales y de comportamiento del software. De manera alternativa, desarrollar casos de uso que describan el software desde el punto de vista del usuario.
7. Revisar cada miniespecificación o usar casos de uso para ver su exactitud, consistencia y falta de ambigüedad.
8. Ensamblar las miniespecificaciones en un documento de ámbito.
9. Revisar el documento de ámbito o colección de casos de uso con todos los interesados.
10. Modificar el documento de ámbito o casos de uso según se requiera.

Ambos proyectos realizan la actividad de marco conceptual que se llama **comunicación**, pero el primer equipo de proyecto realiza la mitad de tareas de trabajo de ingeniería de software que el segundo.

---

6 Recuerde que las características del proyecto también tienen mucho apoyo en la estructura del equipo de software (sección 24.2.3).

## 24.5 El proyecto

Para administrar un proyecto de software exitoso, se debe comprender qué puede salir mal, de modo que los problemas puedan evitarse. En un excelente ensayo acerca de los 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 del software no entiende las necesidades del cliente.
2. El ámbito del producto está pobremente definido.
3. Los cambios se gestionan pobremente.
4. Cambia la tecnología elegida.
5. Las necesidades empresariales cambian [o están mal definidas].
6. Las fechas límite son irreales.
7. Los usuarios son resistentes.
8. Pérdida de patrocinio [o nunca obtenido adecuadamente].
9. El equipo del proyecto carece de personal con habilidades adecuadas.
10. Los gerentes [y profesionales] evitan mejores prácticas y lecciones aprendidas.

**Cita:**

"No tenemos tiempo para detenernos por combustible, ya vamos retrasados."

**M. Cleron**

Los profesionales de la industria, hastiados, con frecuencia se refieren 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 tiempo asignados. El último 10 por ciento toma otro 90 por ciento del esfuerzo y tiempo asignados [Zah94]. Las semillas que conducen a la regla 90-90 están contenidas en las señales anotadas en la lista anterior.

Pero, ¡basta de negatividad! ¿Cómo actúa un gerente para evitar los problemas recién anotados? Reel [Ree99] sugiere un enfoque de sentido común de cinco partes en los proyectos de software:

1. *Comenzar con el pie derecho.* Esto se logra al trabajar duro (muy duro) para entender el problema que debe resolverse y luego establecer objetivos y expectativas realistas para todos aquellos que estarán involucrados en el proyecto. Lo anterior se refuerza al construir el equipo correcto (sección 24.2.3) y darle autonomía, autoridad y tecnología necesarias para realizar el trabajo.

**Cita:**

"Un proyecto es como un viaje en carretera. Algunos 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 una camioneta $4 \times 4$ en las montañas y de noche."

**Cem Kaner, James Bach y Bret Pettichord**

2. *Mantener la cantidad de movimiento.* Muchos proyectos parten hacia un buen comienzo y luego lentamente se desintegran. A fin de mantener la cantidad de movimiento, el gerente de proyecto debe proporcionar incentivos para mantener la rotación de personal en un mínimo absoluto, el equipo debe enfatizar la calidad en cada tarea que realice y el administrador ejecutivo debe hacer todo lo posible para permanecer fuera del camino del equipo.[7]
3. *Siga la pista al progreso.* Para un proyecto de software, el progreso se rastrea conforme los productos operativos (por ejemplo, modelos, código fuente, conjuntos de casos de prueba) se producen y aprueban (usando revisiones técnicas) como parte de una actividad que asegure la calidad. Además, pueden recopilarse medidas de proceso de software y proyecto (capítulo 25) y usarse para valorar el progreso contra promedios desarrollados para la organización de desarrollo del software.

7 La implicación de esta afirmación es que la burocracia se reduce al mínimo, las reuniones extrañas se eliminan y se quita el énfasis a la adhesión dogmática a las reglas de proceso y proyecto. El equipo debe ser autoorganizado y autónomo.

4. *Tome decisiones inteligentes.* En esencia, las decisiones del gerente del proyecto y del equipo de software deben "mantenerse simples". Siempre que sea posible, decida usar software comercial de anaquel, o componentes o patrones de software existentes, así como evitar interfaces a la medida cuando estén disponibles enfoques estándar; decida también identificar y luego evitar los riesgos obvios, y asignar más tiempo del que se considere necesario para tareas complejas y riesgosas (necesitará cada minuto).
5. *Realice un análisis postmortem*. Establezca un mecanismo consistente para extraer lecciones aprendidas por cada proyecto. Evalúe los calendarios planeado y real, recopile y analice métricas de proyecto de software, consiga retroalimentación de los miembros del equipo y de los clientes, y registre los hallazgos en forma escrita.

## 24.6 El principio W$^5$HH

En un excelente ensayo acerca del proceso de software y los proyectos, Barry Boehm [Boe96] afirma: "necesita un principio de organización que reduzca la escala a fin de proporcionar planes [de proyecto] simples para proyectos simples". Boehm sugiere un enfoque que aborda los objetivos del proyecto, hitos y calendarios, responsabilidades, enfoques administrativos y técnicos, y recursos requeridos. Él lo llama *principio W$^5$HH*, por una serie de preguntas que conducen a una definición de las características clave del proyecto y al plan de proyecto resultante:

**¿Cómo se definen las características clave del proyecto?**

*¿Por qué* (why) *se desarrollará el sistema?* Todos los participantes deben valorar la validez de las razones empresariales para el trabajo de software. ¿El propósito de la empresa justifica el gasto de personal, tiempo y dinero?

*¿Qué* (what) *se hará?* Defina el conjunto de tareas requeridas para el proyecto.

*¿Cuándo* (when) *se hará?* El equipo establece un calendario de proyecto al identificar cuándo se realizarán las tareas del proyecto y cuándo se alcanzarán los hitos.

*¿Quién* (who) *es responsable de cada función?* Defina el papel y la responsabilidad de cada miembro del equipo de software.

*¿Dónde* (where) *se ubicarán en la organización?* No todos los roles y responsabilidades residen dentro de los profesionales del software. Clientes, usuarios y otros participantes también tienen responsabilidades.

*¿Cómo* (how) *se hará el trabajo, técnica y organizativamente?* Una vez establecido el ámbito del producto, debe definirse una estrategia técnica para el proyecto.

*¿Cuánto* (how much) *se necesita de cada recurso?* La respuesta a esta pregunta se deriva al desarrollar estimaciones (capítulo 26) con base en las respuestas a las preguntas anteriores.

El principio W$^5$HH de Boehm es aplicable sin importar el tamaño o complejidad de un proyecto de software. Las preguntas anotadas ofrecen un excelente esbozo de la planificación.

## 24.7 Prácticas cruciales

El Airlie Council[8] desarrolló una lista de "prácticas de software cruciales para administración basada en desempeño". Dichas prácticas "las usan consistentemente, y las consideran cruciales,

8 El Airlie Council incluyó un equipo de expertos en ingeniería de software contratados por el Departamento de Defensa estadounidense para ayudar a desarrollar lineamientos para mejores prácticas en administración de proyectos de software e ingeniería de software. Para conocer más acerca de mejores prácticas, vea **www.swqual.com/newsletter/vol1/no3/vol1no3.html**

proyectos y organizaciones enormemente exitosas cuya 'línea base' para el desempeño es consistentemente mucho mejor que el promedio industrial" [Air99].

Las prácticas cruciales[9] incluyen: administración del proyecto basada en métrica (capítulo 25), estimación empírica de costo y calendario (capítulos 26 y 27), rastreo del valor ganado (capítulo 27), rastreo de defecto contra metas de calidad (capítulos del 14 al 16) y administración consciente del personal (sección 24.2). Cada una de estas prácticas cruciales se aborda a lo largo de las partes 3 y 4 de este libro.

HERRAMIENTAS DE SOFTWARE

### *Herramientas de software para gerentes de proyecto*

Las "herramientas" citadas aquí son genéricas y se aplican a un amplio rango de actividades realizadas por los gerentes de proyecto. En capítulos finales se consideran las herramientas específicas de administración de proyecto (por ejemplo, herramientas de calendarización, de estimación, de análisis de riesgo).

**Herramientas representativas:**[10]

El Software Program Manager's Network (**www.spmn.com**) desarrolló una herramienta simple llamada *Project Control Panel*, que brinda a los gerentes de proyecto un indicio directo del estado del proyecto. La herramienta tiene "calibradores" muy parecidos a un tablero y se implementa con Microsoft Excel. Está disponible para descarga en **www.spmn.com/products_software.html**

*Gantthead.com* (**www.gantthead.com/**) desarrolló un conjunto de útiles *listas de comprobación para gerentes de proyecto*.

*Ittoolkit.com* (**www.ittoolkit.com**) proporciona "una colección de guías de planificación, plantillas de proceso y hojas de trabajo inteligentes" disponible en CD-ROM.

## 24.8 RESUMEN

La administración de proyectos de software es una actividad sombrilla dentro de la ingeniería de software. Comienza antes de iniciar cualquier actividad técnica y continúa a lo largo del modelado, construcción y despliegue del software de cómputo.

Cuatro P tienen influencia sustancial sobre la administración del proyecto de software: personal, producto, proceso y proyecto. El personal debe organizarse en equipos eficaces, motivados para hacer trabajo de software de alta calidad, y coordinarse para lograr comunicación efectiva. Los requerimientos del producto deben comunicarse de cliente a desarrollador, dividirse (descomponerse) en sus partes constitutivas y ubicarse para su trabajo por parte del equipo de software. El proceso debe adaptarse al personal y al producto. Se selecciona un marco conceptual común al proceso, se aplica un paradigma de ingeniería de software adecuado y se elige un conjunto de tareas de trabajo para realizar el trabajo. Finalmente, el proyecto debe organizarse de forma que permita triunfar al equipo de software.

El elemento esencial en todos los proyectos de software es el personal. Los ingenieros del software pueden organizarse en diferentes estructuras de equipo que van desde las jerarquías tradicionales de control hasta los equipos de "paradigma abierto". Para apoyar el trabajo del equipo, pueden aplicarse varias técnicas de coordinación y comunicación. En general, las revisiones técnicas y la comunicación informal persona a persona tienen más valor para los profesionales.

La actividad de administración del proyecto abarca medición y métricas, estimación y calendarización, análisis de riesgos, rastreo y control. Cada uno de estos temas se considera en los capítulos siguientes.

---

9 Aquí sólo se mencionan aquellas prácticas cruciales asociadas con "integridad del proyecto".

10 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

## Problemas y puntos por evaluar

**24.1.** Con base en la información contenida en este capítulo y en su propia experiencia, desarrolle "diez mandamientos" para empoderar a los ingenieros del software, es decir, elabore una lista de 10 lineamientos que conducirán al personal de software a que trabaje a toda su potencia.

**24.2.** El modelo de madurez de capacidades del personal (People-CMM) de The Software Engineering Institute, People-CMM echa un vistazo organizado a "áreas prácticas clave" que cultivan buen personal de software. Su instructor le asignará una APC para análisis y resumen.

**24.3.** Describa tres situaciones de la vida real en las que el cliente y el usuario final sean el mismo. Describa tres situaciones en las que sean diferentes.

**24.4.** Las decisiones tomadas por los administradores ejecutivos pueden tener un impacto significativo sobre la efectividad de un equipo de ingeniería del software. Proporcione cinco ejemplos para ilustrar que esto es cierto.

**24.5.** Revise el libro de Weinberg [Wei86] y escriba un resumen, con una extensión de dos a tres páginas, de los temas que deben considerarse al aplicar el modelo MOI.

**24.6.** Al lector se le asigna una gerencia de proyecto dentro de una organización de sistemas de información. Su labor será construir una aplicación que sea muy similar a otras que su equipo construyó, aunque ésta será más grande y más compleja. Los requerimientos se documentaron ampliamente por parte del cliente. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo de proceso de software elegiría y por qué?

**24.7.** Al lector se le asigna una gerencia de proyecto para una pequeña compañía de productos de software. Su labor será construir un producto innovador que combine hardware de realidad virtual con software de última generación. Puesto que la competencia para el mismo mercado de entretenimiento es intensa, existe una presión significativa para tener listo el trabajo. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo de proceso de software elegiría y por qué?

**24.8.** Al lector se le asigna una gerencia de proyecto para una gran compañía de productos de software. Su labor será administrar el desarrollo de la versión de siguiente generación de su software de procesamiento de palabras ampliamente usado. Puesto que la competencia es intensa, se establecieron y anunciaron apretadas fechas límite. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo de proceso de software elegiría y por qué?

**24.9.** Al lector se le asigna una gerencia de proyecto de software para una compañía que atiende al mundo de la ingeniería genética. Su labor será administrar el desarrollo de un nuevo producto de software que acelerará el ritmo de tipificación genética. El trabajo está orientado a investigación y desarrollo, pero la meta es elaborar un producto dentro del próximo año. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo de proceso de software elegiría y por qué?

**24.10.** Al lector se le ha pedido desarrollar una pequeña aplicación que analice cada curso ofrecido en la universidad y reporte las calificaciones promedio obtenidas en el curso (por un determinado periodo). Exponga el alcance y las limitaciones de este trabajo.

**24.11.** Haga una descomposición funcional de primer nivel de la función de plantilla de página que se estudió brevemente en la sección 24.3.2.

## Lecturas y fuentes de información adicionales

El Project Management Institute (*Guide to the Project management body of Knowledge*, PMI, 2001) abarca todos los aspectos importantes de la administración de proyectos. Bechtold (*Essentials of Software Project Management,* 2a. ed., Management Concepts, 2007), Wysocki (*Effective Software Project Management,* Wiley, 2006), Stellman y Greene (*Applied Software Project Management,* O'Reilly, 2005), y Berkun (*The Art of Project Management,* O'Reilly, 2005) enseñan habilidades básicas y ofrecen lineamientos detallados para todas las tareas de administración de proyectos de software. McConnell (*Professional Software Development*, Addison-Wesley, 2004) ofrece consejo pragmático para lograr "calendarios más cortos, productos de mayor calidad y proyectos más exitosos". Henry (*Software Project Management*, Addison-Wesley, 2003) ofrece consejo del mundo real que es útil para todos los gerentes de proyecto.

Tom DeMarco *et al.* (*Adrenaline Junkies and Template Zombies*, Dorset House, 2008) escribieron un tratamiento comprensivo de los patrones humanos que se encuentran en todo proyecto de software. Una excelente serie de cuatro volúmenes, escrito por Weinberg (*Quality Software Management*, Dorset House, 1992, 1993, 1994, 1996), presenta conceptos de sistemas básicos de pensamiento y administración, explica cómo usar efectivamente las mediciones y aborda la "acción congruente", la habilidad para establecer "ajuste" entre las necesidades del gerente, las necesidades del personal técnico y las necesidades de la empresa. Ello proporcionará información útil a los gerentes novatos y a los expertos. Futrell *et al.* (*Quality Software Project Management*, Prentice-Hall, 2002) presentan un voluminoso tratamiento de la administración de proyectos. Brown *et al.* (*Antipatterns in Project Management*, Wiley, 2000) plantean qué no hacer durante la administración de un proyecto de software.

Brooks (*The Mythical Man-Month*, Anniversary Edition, Addison-Wesley, 1995) actualizó su libro clásico para ofrecer nueva comprensión de los temas de proyecto y administración de software. McConnell (*Software Project Survival Guide*, Microsoft Press, 1997) presenta excelente lineamiento pragmático para quienes deben administrar proyectos de software. Purba y Shah (*How to Manage a Successful Software Project*, 2a. ed., Wiley, 2000) presentan diversos estudios de caso que indican por qué algunos proyectos triunfan y otros fracasan. Bennatan (*On Time Within Budget*, 3a. ed., Wiley, 2000) presenta consejos útiles y lineamientos para gerentes de proyecto de software. Weigers (*Practical Project Initiation*, Microsoft Press, 2007) proporciona lineamientos prácticos para poner en marcha exitosamente un proyecto de software.

Puede argumentarse que el aspecto más importante de la administración del proyecto de software es la administración de personas. Cockburn (*Agile Software Development*, Addison-Wesley, 2002) presenta uno de los mejores análisis del personal de software escrito a la fecha. DeMarco y Lister [DeM98] escribieron el libro definitivo acerca del personal de software y los proyectos del software. Además, en años recientes se publicaron los siguientes libros acerca de la materia y vale la pena examinarlos:

Cantor, M., *Software Leadership: A Guide to Successful Software Development,* Addison-Wesley, 2001.

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.

Garton, C., y K. Wegryn, *Managing Without Walls,* McPress, 2006.

Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process,* Addison-Wesley, 1997.

Humphrey, W. S., *TSP-Coaching Development Teams,* Addison-Wesley, 2006.

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.

Peters, L., *Getting Results from Software Development Teams,* Microsoft Press, 2008.

Whitehead, R., *Leading a Software Development Team,* Addison-Wesley, 2001.

Aun cuando no se relacionan específicamente con el mundo del software, y en ocasiones adolecen de sobresimplificación y gran generalización, los muy vendidos libros de "administración" de Kanter (*Confidence,* Three Rivers Press, 2006), Covy (*The 8th Habit,* Free Press, 2004), 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) enfatizan "nuevas reglas" definidas por una economía rápidamente cambiante. Los títulos más antiguos, como *Who Moved My Cheese?* y *The One-Minute Manager* e *In Search of Excellence*, siguen ofreciendo valiosos elementos que pueden ayudar a administrar personal y proyectos de manera más efectiva.

En internet está disponible una gran variedad de fuentes de información acerca de las métricas de la administración de proyectos de software. Una lista actualizada de referencias existentes en la World Wide Web y que son relevantes para la administración de proyectos de software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

# 25

# MÉTRICAS DE PROCESO Y DE PROYECTO

CONCEPTOS CLAVE



La medición permite ganar comprensión acerca del proceso y del proyecto, al proporcionar un mecanismo de evaluación objetiva. Lord Kelvin dijo alguna vez:

> Cuando puedes medir aquello de lo que hablas y expresarlo en números, sabes algo acerca de ello; pero cuando no puedes medir, cuando no puedes expresarlo en números, tu conocimiento es exiguo e insatisfactorio: puede ser el comienzo del conocimiento; sin embargo, apenas habrás avanzado, en tus pensamientos, hacia la etapa de una ciencia.

La comunidad de la ingeniería del software tomó a pecho las palabras de lord Kelvin. ¡Mas no sin frustración ni con poca controversia!

La medición puede aplicarse al proceso de software con la intención de mejorarlo de manera continua. Puede usarse a través de un proyecto de software para auxiliar en estimación, control de calidad, valoración de productividad y control de proyecto. Finalmente, la medición pueden usarla los ingenieros del software para ayudar en la valoración de la calidad de los productos de trabajo y auxiliar en la toma de decisiones tácticas conforme avanza un proyecto (capítulo 23).

Dentro del contexto del proceso de software y de los proyectos que se realizan usando a aquél, un equipo de software está preocupado principalmente por la productividad y por las métricas de calidad: medidas de "salidas" de desarrollo de software como función del esfuerzo y el tiempo aplicado y medidas de la "aptitud para el uso" de los productos operativos que se producen. Con propósitos de planificación y estimación, el interés es histórico. ¿Cuál fue la productividad en el desarrollo de software en proyectos anteriores? ¿Cuál la calidad del software que se produjo? ¿Cómo pueden extrapolarse al presente los datos de productividad y calidad anteriores? ¿Cómo pueden las mediciones ayudar a planificar y a estimar con más precisión?

En su manual acerca de medición del software, Park, Goethert y Florac [Par96] anotan las razones por las que se mide: 1) para *caracterizar* un esfuerzo y obtener comprensión "de los

## UNA MIRADA RÁPIDA

**¿Qué es?** Las métricas de proceso y proyecto de software son medidas cuantitativas que permiten obtener comprensión acerca de la eficacia del proceso del software y de los proyectos que se realizan, usando el proceso como marco conceptual. Se recopilan datos básicos de calidad y productividad. Luego, se analizan, se comparan con promedios anteriores y se valoran para determinar si han ocurrido mejoras en calidad y productividad. Las métricas también se usan para puntualizar áreas problemáticas, de modo que puedan desarrollarse remedios y el proceso de software pueda mejorarse.

**¿Quién lo hace?** Las métricas del software se analizan y valoran por parte de los gerentes del software. Con frecuencia, los ingenieros del software recopilan las medidas.

**¿Por qué es importante?** Si no se mide, el juicio puede basarse solamente en la evaluación subjetiva. Con medición, pueden marcarse las tendencias (buenas o malas), hacerse mejores estimaciones y, con el tiempo, lograrse verdadera mejoría.

**¿Cuáles son los pasos?** Se define un conjunto limitado de medidas de proceso, proyecto y producto, que son fáciles de recopilar. Dichas medidas con frecuencia se normalizan usando métricas de tamaño o de función. El resultado se analiza y compara con promedios anteriores para proyectos similares realizados dentro de la organización. Las tendencias se valoran y se generan conclusiones.

**¿Cuál es el producto final?** Un conjunto de métricas de software que proporcionan comprensión acerca del proceso y del proyecto.

**¿Cómo me aseguro de que lo hice bien?** Al aplicar un esquema de medición consistente, aunque simple, que nunca debe usarse para valorar, recompensar o castigar el desempeño individual.

procesos, productos, recursos y entornos, y establecer líneas de referencia para comparar con valoraciones futuras"; 2) para *evaluar* y "determinar el estado de avance con respecto a los planes"; 3) para *predecir* al "obtener comprensión de las relaciones entre procesos y productos, y construir modelos 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 administrativa. Si se realiza adecuadamente, ofrece entendimiento al gerente de un proyecto. Y, como resultado, lo auxilia a él y al equipo de software para tomar decisiones que conducirán hacia un proyecto exitoso.

## 25.1 Métricas en los dominios de proceso y proyecto

PUNTO CLAVE

La métrica de proceso tiene impacto a largo plazo. Su intención es mejorar el proceso en sí. La métrica de proyecto usualmente contribuye al desarrollo de la primera.

Las *métricas de proceso* se recopilan a través de todos los proyectos y durante largos espacios de tiempo. Su intención es proporcionar un conjunto de indicadores de proceso que conduzca a mejorar el proceso de software a largo plazo. Las *métricas de proyecto* permiten al gerente de un proyecto de software: 1) valorar el estado de un proyecto en marcha, 2) rastrear riesgos potenciales, 3) descubrir áreas problema antes de que se vuelvan "críticas", 4) ajustar el flujo de trabajo o las tareas y 5) evaluar la habilidad del equipo del proyecto para controlar la calidad de los productos operativos del software.

Las medidas que recopila un equipo de proyecto y que convierte en métricas para uso durante un proyecto también pueden transmitirse a quienes tienen responsabilidad en la mejora del proceso de software (capítulo 30). Por esta razón, muchas de las métricas se usan tanto en los dominios del proceso como en los del proyecto.

### 25.1.1 Las métricas del proceso y la mejora del proceso de software

La única forma racional para mejorar cualquier proceso es medir atributos específicos del mismo, desarrollar un conjunto de métricas significativas con base en dichos atributos y luego usarlas para proporcionar indicadores que conducirán a una estrategia para mejorar (capítulo 30). Pero antes de estudiar las métricas del software y su impacto sobre el mejoramiento del proceso de software, es importante observar que el proceso sólo es uno de varios "factores controlables en el mejoramiento de la calidad del software y del desempeño organizativo" [Pau94].

PUNTO CLAVE

La habilidad y motivación del personal del software que hace el trabajo son los factores más importantes que influyen en la calidad del software.

En la figura 25.1, el proceso se asienta en el centro de un triángulo que conecta tres factores que tienen profunda influencia sobre la calidad del software y en el desempeño de la organización. La habilidad y motivación del personal ha demostrado [Boe81] ser 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 puebla el proceso también tiene un impacto.

Además, existe el triángulo de proceso dentro de un círculo de condiciones ambientales que incluyen entorno de desarrollo (por ejemplo, herramientas de software integradas), condiciones empresariales (fechas límite, reglas empresariales) y características del cliente (facilidad de comunicación y colaboración).

**Cita:**

"Las métricas de software le permiten saber cuándo reír y cuándo llorar."

**Tom Gilb**

La eficacia de un proceso de software sólo puede medirse de manera indirecta. Esto significa que es posible derivar un conjunto de métricas con base en los resultados que pueden derivarse del proceso. Los resultados incluyen medidas de los errores descubiertos antes de liberar el software, defectos entregados a y reportados por usuarios finales, productos operativos entregados (productividad), esfuerzo humano empleado, tiempo calendario consumido, conformidad con la agenda y otras medidas. También pueden derivarse métricas de proceso al medir las características de tareas de ingeniería de software específicas. Por ejemplo, puede medirse el

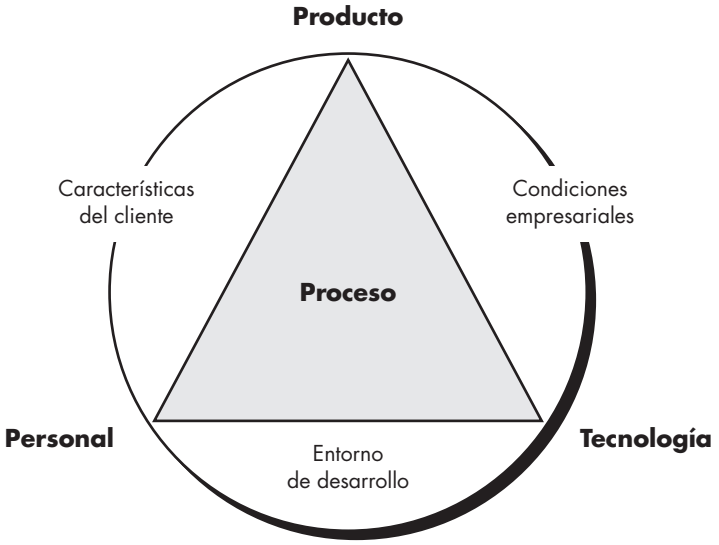



**FIGURA 25.1**
**Determinantes para calidad del software y efectividad organizativa.**
Fuente: Adaptado de [Pau94].

esfuerzo y el tiempo empleados al realizar las actividades sombrilla y las actividades genéricas de ingeniería del software descritas en el capítulo 2.

Grady [Gra92] argumenta que existen usos "privados y públicos" para diferentes tipos de datos de proceso. Puesto que es natural que los ingenieros de software individual puedan ser sensibles al uso de las métricas recopiladas de manera individual, dichos datos deben ser privados para el individuo y funcionar sólo como un indicio para él. Los ejemplos de *métricas privadas* incluyen tasas de defecto (por individuo), tasas de defecto (por componente) y errores que se encuentran durante el desarrollo.

**? ¿Cuál es la diferencia entre usos privado y público para las métricas del software?**

La filosofía de "datos de proceso privados" se conforma bien con el enfoque de proceso de software personal (capítulo 2) propuesto por Humphrey [Hum97], quien reconoce que el mejoramiento en el proceso del software puede y debe comenzar en el nivel individual. Los datos de proceso privado pueden funcionar como un importante motor conforme se trabaja para mejorar el enfoque de la ingeniería del software.

Algunas métricas de proceso son privadas para el equipo de proyecto del software, pero públicas para todos los miembros del equipo. Los ejemplos incluyen defectos reportados por grandes funciones de software (que se desarrollaron por parte de algún número de profesionales), errores encontrados durante las revisiones técnicas y líneas de código o puntos de función por componente o función.[1] El equipo revisa dichos datos para descubrir indicios que puedan mejorar el desempeño del equipo.

Las métricas públicas por lo general asimilan información que originalmente era privada para los individuos y el equipo. Las tasas de defecto en el nivel de proyecto (absolutamente no atribuidas a un individuo), esfuerzo, tiempos calendario y datos relacionados se recopilan y evalúan con la intención de descubrir indicios que puedan mejorar el desempeño del proceso organizativo.

Las métricas de proceso de software pueden proporcionar beneficios significativos conforme una organización trabaja para mejorar su nivel global de madurez de proceso. Sin embargo, como todas las métricas, éstas pueden tener mal uso, lo que crea más problemas de los que resuelven. Grady [Gra92] sugiere una "etiqueta de métrica de software" que sea adecuada tanto para gerentes como para profesionales, conforme instauran un programa de métricas de proceso:

1 Las líneas de código y las métricas de punto de función se estudian en las secciones 25.2.1 y 25.2.2.

- Usar el sentido común y sensibilidad organizacional cuando se interpreten datos de métricas.
- Proporcionar retroalimentación regular a los individuos y equipos que recopilan medidas y métricas.
- No usar métricas para valorar a los individuos.
- Trabajar con los profesionales y con los equipos para establecer metas y métricas claras que se usarán para lograr las primeras.
- Nunca usar métricas para amenazar a los individuos o a los equipos.
- No considerar "negativos" los datos de métricas que indiquen un área problemática. Dichos datos simplemente son un indicio para mejorar el proceso.
- No obsesionarse con una sola métrica ni excluir otras métricas importantes.

**¿Qué lineamientos deben aplicarse cuando se recopilan métricas de software?**

Conforme una organización se siente más cómoda con la recolección y uso de métricas de proceso, la derivación de los indicadores simples da lugar a un enfoque más riguroso llamado *mejora estadística de proceso de software* (MEPS). En esencia, MEPS usa análisis de falla del software para recopilar información acerca de todos los errores y defectos[2] que se encuentren conforme se desarrolle y use una aplicación, sistema o producto.

### 25.1.2 Métricas de proyecto

A diferencia de las métricas de proceso de software que se usaron con propósitos estratégicos, las medidas de proyecto de software son tácticas. Es decir, el gerente de proyecto y un equipo de software usan las métricas de proyecto y los indicadores derivados de ellas para adaptar el flujo de trabajo del proyecto y las actividades técnicas.

La primera aplicación de las métricas de proyecto sobre la mayoría de los proyectos de software ocurre durante la estimación. Las métricas recopiladas de proyectos anteriores se usan como la base desde la cual se hacen estimaciones de esfuerzo y tiempo para el trabajo de software nuevo. Conforme avanza un proyecto, las medidas de esfuerzo y tiempo calendario utilizadas se comparan con las estimaciones originales (y con la agenda del proyecto). El gerente del proyecto usa dichos datos para monitorear y controlar el progreso.

Mientras comienza el trabajo técnico, otras métricas del proyecto empiezan a tener significado. Se miden las tasas de producción representadas en términos de modelos creados, horas de revisión, puntos de función y líneas de fuente entregadas. Además, se rastrean los errores descubiertos durante cada tarea de ingeniería del software. Conforme el software evoluciona desde los requerimientos hasta el diseño, se recopilan métricas técnicas (capítulo 23) a fin de valorar la calidad del diseño y proporcionar indicios que influirán en el enfoque tomado para generación y prueba de código.

**¿Cómo deben usarse las métricas durante el proyecto en sí?**

La intención de las métricas de proyecto es doble. Primero, se usan para minimizar el calendario de desarrollo al hacer los ajustes necesarios para evitar demoras y mitigar potenciales problemas y riesgos. Segundo, se usan para valorar la calidad del producto sobre una base en marcha y, cuando es necesario, modificar el enfoque técnico para mejorar la calidad.

Conforme la calidad mejora, los defectos se minimizan, y conforme el conteo de defectos baja, la cantidad de reelaboración requerida durante el proyecto también se reduce. Esto conduce a una reducción en el costo global del proyecto.

2 En este libro, un *error* se define como un fallo en un producto operativo de la 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 entregar el software al usuario final. Debe destacarse que otros no hacen esta distinción.

**CASASEGURA**

### *Establecimiento de un enfoque de métricas*

**La escena:** Oficina de Doug Miller cuando el proyecto de software *CasaSegura* está a punto de comenzar.

**Participantes:** Doug Miller (gerente del equipo de ingeniería del software *CasaSegura*) y Vinod Raman y Jamie Lazar, miembros del equipo de ingeniería de software del producto.

**La conversación:**

**Doug:** Antes de empezar a trabajar en este proyecto, me gustaría que definieran y recopilaran un conjunto de métricas simples. Para comenzar, tendrán que definir sus metas.

**Vinod (frunce el ceño):** Nunca hemos hecho esto antes y...

**Jamie (interrumpe):** Y con base en la administración de la línea de tiempo de la que hablaste, nunca tendremos el tiempo. De cualquier forma, ¿qué bien hacen las métricas?

**Doug (levanta su mano para detener el embate):** Cálmense y respiren, chicos. El hecho de que nunca lo hayamos hecho antes es la principal razón para comenzar ahora, y el trabajo de las métricas del que hablo de ninguna manera debe tardar mucho tiempo... de hecho, sólo puede ahorrarnos tiempo.

**Vinod:** ¿Cómo?

**Doug:** Miren, vamos a hacer mucho más trabajo interno en ingeniería del software conforme nuestros productos se vuelvan más inteligentes, se habiliten en web, todo eso... y necesitamos entender el proceso que usamos para construir software... y mejorarlo de modo que podamos construir software de mejor manera. La única forma de hacer esto es medir.

**Jamie:** Pero estamos bajo presión de tiempo, Doug. No estoy a favor de más papeleo... necesitamos el tiempo para hacer nuestro trabajo, no para recolectar datos.

**Doug (con calma):** Jamie, el trabajo de un ingeniero involucra recopilar datos, evaluarlos y usar los resultados para mejorar el producto y el proceso. ¿Me equivoco?

**Jamie:** No, pero...

**Doug:** ¿Y si mantenemos el número de medidas que recopilemos 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 no es necesario.

**Doug:** Voy a pedirte que me complazcas en esto. ¿Cuánto saben acerca de las métricas del software?

**Jamie (mira a Vinod):** No mucho.

**Doug:** Aquí hay algunas referencias en la red... pasé algunas horas recuperándolas para avanzar.

**Jamie (sonríe):** Creo que dijiste que esto no tomaría tiempo.

**Doug:** El tiempo que se emplea aprendiendo nunca se desperdicia... háganlo y luego establezcan algunas metas, planteen algunas preguntas y definan la métrica que necesitamos recopilar.

## 25.2 MEDICIÓN DEL SOFTWARE

En el capítulo 23 se indicó que las mediciones en el mundo físico pueden clasificarse en dos formas: medidas directas (por ejemplo, la longitud de un tornillo) y medidas indirectas (por ejemplo, la "calidad" de los tornillos producidos, medidos por conteo de rechazos). Las métricas de software pueden clasificarse de igual modo.

**Cita:**

"No todo lo que puede contarse cuenta y no todo lo que cuenta puede contarse."

**Albert Einstein**

Las *medidas directas* del proceso de software incluyen costo y esfuerzo aplicado. Las medidas directas del producto incluyen líneas de código (LOC) producidas, rapidez de ejecución, tamaño de memoria y defectos reportados sobre cierto espacio de tiempo. Las medidas indirectas del producto incluyen funcionalidad, calidad, complejidad, eficiencia, confiabilidad, capacidad de mantenimiento y muchas otras "habilidades" que se estudiaron en el capítulo 14.

El costo y el esfuerzo requeridos para construir software, el número de líneas de código producidas y otras medidas directas son relativamente sencillos de recolectar, en tanto se establezcan por adelantado convenciones específicas para la medición. Sin embargo, la calidad y funcionalidad del software o su eficiencia o capacidad de mantenimiento son más difíciles de valorar y pueden medirse sólo de manera indirecta.

**CONSEJO**

*Puesto que muchos factores afectan el trabajo de software, no use métricas para comparar individuos o equipos.*

El dominio de la métrica del software se dividió en métricas de proceso, proyecto y producto, y se dijo que las métricas de producto que son privadas para un individuo con frecuencia se combinan para desarrollar métricas de proyecto que son públicas para un equipo de software. Luego las métricas de proyecto se consolidan para crear métricas de proceso que son públicas para la organización del software como un todo. Pero, ¿cómo combina una organización las métricas que vienen de diferentes individuos o proyectos?

**FIGURA 25.2**
**Métricas orientadas a tamaño**

| Proyecto | LOC | Esfuerzo | $(000) | Pp. 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 |
| • | • | • | • | • | • | | |
| • | • | • | • | • | • | | |
| • | • | • | • | • | • | | |

Para ilustrar, considere un ejemplo simple. Los individuos que trabajan en dos equipos de proyecto diferentes registran y categorizan todos los errores que encuentran durante el proceso de software. Las medidas individuales luego se combinan para desarrollar medidas de equipo. El equipo A encuentra 342 errores durante el proceso de software antes de la liberación. El equipo B encuentra 184 errores. Si todas las demás cosas permanecen iguales, ¿cuál equipo es más efectivo para descubrir errores a lo largo del proceso? Dado que no se conoce el tamaño o complejidad de los proyectos, no puede responderse esta pregunta. Sin embargo, si las medidas se normalizan, es posible crear métricas de software que permitan la comparación con promedios organizacionales más amplios.

### 25.2.1 Métricas orientadas a tamaño

Las métricas de software orientadas a tamaño se derivan al normalizar las medidas de calidad y/o productividad para considerar el *tamaño* del software que se produjo. Si una organización de software mantiene registros simples, puede crearse una tabla de medidas orientadas a tamaño, como la que se muestra en la figura 25.2. La tabla menciona cada proyecto de desarrollo de software que se completó durante los años anteriores y que corresponden a medidas para dicho proyecto. En la entrada de la tabla (figura 25.2) para el proyecto alfa: 12 100 líneas de código se desarrollaron con 24 persona-meses de esfuerzo a un costo de US$168 000. Debe observarse que los registros de esfuerzo y código que aparecen en la tabla representan todas las actividades de ingeniería del software (análisis, diseño, código y prueba), no sólo codificación. Más información para el proyecto alfa indica que: se desarrollaron 365 páginas de documentación, se registraron 134 errores antes de liberar el software y se encontraron 29 defectos después de liberarlo al cliente, dentro del primer año de operación. Tres personas trabajaron en el desarrollo del software para el proyecto alfa.

Con la finalidad de desarrollar métricas que puedan asimilarse con métricas similares de otros proyectos, pueden elegirse líneas de códigos como un valor de normalización. A partir de los rudimentarios datos contenidos en la tabla, pueden desarrollarse métricas simples orientadas a tamaño para cada proyecto:

- Errores por KLOC (miles de líneas de código).
- Defectos por KLOC.
- $ por KLOC.
- Páginas de documentación por KLOC.

Además, es posible calcular otras métricas interesantes:

- Errores por persona-mes.
- KLOC por persona-mes.
- $ por página de documentación.

PUNTO CLAVE

Las métricas orientadas a tamaño se usan ampliamente, pero continúa el debate acerca de su validez y aplicabilidad.

Las métricas orientadas a tamaño no se aceptan universalmente como la mejor forma de medir el proceso de software. La mayor parte de la controversia gira en torno del uso de líneas de código como medida clave. Quienes proponen la medida LOC afirman que las LOC son un "artefacto" de todos los proyectos de desarrollo de software y que pueden contarse fácilmente; que muchos modelos existentes de estimación de software usan LOC o KLOC como entrada clave y que ya existe un gran cuerpo de literatura y predicado de datos acerca de LOC. Por otra parte, los opositores argumentan que las medidas LOC dependen del lenguaje de programación; que cuando se considera la productividad, castigan a los programas bien diseñados pero cortos; que no pueden acomodarse con facilidad en lenguajes no procedurales y que su uso en la estimación requiere un nivel de detalle que puede ser difícil de lograr (es decir, el planificador debe estimar las LOC que se van a producir mucho antes de completar el análisis y el diseño).

### 25.2.2 Métricas orientadas a función

Las métricas de software orientadas a función usan una medida de la funcionalidad entregada por la aplicación como un valor de normalización. La métrica orientada a función de mayor uso es el punto de función (PF). El cálculo del punto de función se basa en características del dominio y de la complejidad de información del software. La mecánica del cálculo del PF se estudió en el capítulo 23.[3]

El punto de función, como la medida LOC, es controvertido. Quienes lo proponen afirman que el PF es independiente del lenguaje de programación, lo que lo hace ideal para aplicaciones que usan lenguajes convencionales y no procedurales, 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 estimación. Sus opositores afirman que el método requiere cierta "maña", pues dicho 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.

### 25.2.3 Reconciliación de métricas LOC y PF

La relación entre líneas de código y puntos de función depende del lenguaje de programación que se use para implementar el software y la calidad del diseño. Algunos estudios intentan relacionar las medidas PF y LOC. La tabla[4] [QSM02], que se presenta en la página siguiente, ofrece estimaciones burdas del número promedio de líneas de código requeridas para construir un punto de función en varios lenguajes de programación.

Una revisión de estos datos indica que un LOC de C++ proporciona aproximadamente 2.4 veces la "funcionalidad" (como promedio) que un LOC de C. Más aún, un LOC de Smalltalk proporciona al menos cuatro veces la funcionalidad de un LOC para un lenguaje de programación convencional, como Ada, COBOL o C. Al usar la información contenida en la tabla, es posible "retroactivar" [Jon98] el software existente para estimar el número de puntos de función, una vez conocido el número total de enunciados del lenguaje de programación.

---

3 Vea la sección 23.2.1 para un análisis detallado del cálculo de PF.

4 Usado con permiso de Quantitative Software Management (www.qsm.com), copyright 2002.

| | *LOC por punto de función* | | | |
|---|---|---|---|---|
| **Lenguaje de programación** | **Promedio** | **Mediana** | **Bajo** | **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 | — | — | — |
| Easytrieve+ | 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 |

Las medidas LOC y PF se usan frecuentemente para calcular métricas de productividad. Esto invariablemente conduce a un debate acerca del uso de tales datos. ¿El LOC/persona-mes (o PF/persona-mes) de un grupo debe compararse con datos similares de otro? ¿Los gerentes deben valorar el desempeño individual usando dichas métricas? La respuesta a estas preguntas es un enfático *¡NO!* La razón para esta respuesta es que muchos factores influyen en la producti-

vidad, lo que hace que las comparaciones entre "manzanas y naranjas" se malinterpreten con facilidad.

Es cierto que los puntos de función y las métricas basadas en LOC son predictores relativamente precisos del esfuerzo y del costo en el desarrollo del software. Sin embargo, si se van a usar LOC y PF para estimación (capítulo 26), debe establecerse una línea de referencia de información.

Dentro del contexto de las métricas de proceso y proyecto, la preocupación debe estar centrada principalmente en la productividad y la calidad, medidas de "salida" del desarrollo del software como función del esfuerzo y el tiempo aplicados y medidas de "aptitud para el uso" de los productos operativos que se producen. Con propósitos de mejorar el proceso y la planificación del proyecto, su interés es histórico. ¿Cuál fue la productividad en el desarrollo del software en proyectos anteriores? ¿Cuál fue la calidad del software que se produjo? ¿Cómo pueden extrapolarse al presente los datos de productividad y calidad anteriores? ¿Cómo puede ayudar a mejorar el proceso y la planificación de nuevos proyectos con más precisión?

### 25.2.4 Métricas orientadas a objeto

Las métricas de proyecto de software convencional (LOC o PF) pueden usarse para estimar proyectos de software orientados a objeto. Sin embargo, dichas métricas no proporcionan suficiente granularidad para los ajustes de calendario y esfuerzo que se requieren conforme se repite a través 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 la parte 2 de este libro) es una secuencia detallada de pasos que describen la interacción entre el usuario y la aplicación. Cada guión se organiza en tripletas de la forma

{**iniciador,** *acción,* **participante**}

donde **iniciador** es el objeto que solicita cierto servicio (que inicia un mensaje), *acción* es el resultado de la solicitud y **participante** es el objeto servidor que satisface la solicitud. El número de guiones de escenario se relaciona directamente con el tamaño de la aplicación y con el número de casos de prueba que deben desarrollarse para ejercitar el sistema una vez construido.


*No es raro para guiones de escenario múltiple mencionar la misma funcionalidad u objetos de datos. Por tanto, tenga cuidado cuando use conteo de guión. Muchos guiones en ocasiones pueden reducirse a una sola clase o conjunto de código.*

**Número de clases clave.** Las *clases clave* son los "componentes enormemente independientes" [Lor94] que se definen tempranamente en el análisis orientado a objeto (capítulo 6).[5] Puesto que las clases clave son centrales en el dominio del problema, el número de tales clases es un indicio de la cantidad de esfuerzo requerido para desarrollar el software y también de la cantidad potencial de reuso por aplicar durante el desarrollo del sistema.

**Número de clases de apoyo.** Las *clases de apoyo* se requieren para implementar el sistema, pero no se relacionan de inmediato con el dominio del problema. Los ejemplos pueden ser clases de interfaz de usuario (GUI), clases de acceso y manipulación de base de datos y clases de cálculo. Además, es posible desarrollar clases de apoyo para cada una de las clases clave. Las clases de apoyo se definen de manera iterativa a lo largo de un proceso evolutivo. El número de clases de apoyo es un indicio de la cantidad de esfuerzo requerido para desarrollar el software y también de la potencial cantidad de reuso que se va a aplicar durante el desarrollo del sistema.


*Las clases pueden variar en tamaño y complejidad. Por tanto, vale la pena considerar la clasificación del conteo de clase por tamaño y complejidad.*

**Número promedio de clases de apoyo por clase clave.** En general, las clases clave se conocen tempranamente en el proyecto. Las clases de apoyo se definen a todo lo largo del mismo. Si el número promedio de clases de apoyo por clase clave se conoce para un dominio de problema determinado, la estimación (con base en el número total de clases) se simplificará

5 En la parte 2 de este libro, se hace referencia a clases clave como *clases de análisis*.