UNIDAD 7: PRUEBAS. IMPLEMENTACION. MANTENIMIENTO

Concepto y fundamentos de las pruebas. Defectos y fallas del software. Prueba unitaria. Prueba de integración. Prueba de caja blanca. Prueba de caja negra. Prueba funcional. Prueba de rendimiento. Planificación de la prueba. Diseño de casos de prueba. Implementación: Elección del entorno. Elección de la tecnología. Pautas para la programación. Esquema de implantación. Documentación. Entrenamiento. Mantenimiento: La naturaleza del mantenimiento. Los problemas del mantenimiento. Técnicas y herramientas para el mantenimiento. Reingeniería. Ingeniería inversa. Reestructuración

Un **error** de programación se puede dar al momento en que el desarrollador asigna 2 valores a una misma variable, o cometido en la lógica de programación. En el momento en que se compila el código, se arma la versión y se instala en un ambiente, ese software contiene **defectos**, ¿Cuáles?, no lo sabemos si no hasta ejecutar nuestras pruebas, en el momento en que el sistema falla, se manifiesta mediante un mensaje de error el cual capturamos para reportar un **fallo**.

```
void MinMax (int Min, int Max)
{
        int Help;
                if (Min>Max)
                {
                        Max = Help;
                        Max = Min;
                        Help = Min;
                }
End MinMax;
```

Error ("Error"):

- Acción humana que produce un resultado incorrecto.
- **Ejemplo: Un error de programación**

Defecto ("Defect"):

- Desperfecto en un componente o sistema que puede causar que el componente o sistema falle en desempeñar las funciones requeridas.
- **Ejemplo: Una sentencia o una definición de datos incorrectas.**

Fallo ("Failure"):

- Manifestación física o funcional de un defecto.
- **Ejemplo: Desviación de un componente o sistema respecto de la prestación, servicio o resultado esperados.**

En resumen, un **error** introduce un **defecto** en el software que a su vez causa un **fallo** al momento de ejecutar pruebas.

Es común ver como algunas personas utilizan de manera indistinta los términos Defecto, Falla y Error. Sin embargo, cada uno de ellos tiene un significado diferente:

- **Defecto:** Un defecto se encuentra en un artefacto y puede definirse como una diferencia entre la versión correcta del artefacto y una versión incorrecta. Coincide con la definición de diccionario, **"imperfección"**.
- **Falla:** En terminología IEEE, una falla es la **discrepancia visible** que se produce al ejecutar un programa con un defecto, el cual es incapaz de funcionar correctamente (no sigue su curso normal).
- **Error:** Es una equivocación cometida por el desarrollador. Algunos ejemplos de errores son: un error de digitación, una malinterpretación de un requerimiento o de la funcionalidad de un método. El estándar 829 de la IEEE coincide con la definición de diccionario de error como **"una idea falsa o equivocada".** Por tal razón un programa no puede tener o estar en un error, ya que los programas no tienen ideas; las ideas las tienen la gente.

Ampliando el tema de los errores, estos errores ocurren cuando cualquier aspecto de un producto de software es incompleto, inconsistente o incorrecto. Las tres grandes clases clasificaciones de errores del software son los de **requisitos,** de **diseño** y de **implantación**. En esta sección hablaremos de dos de ellos:

- Errores de diseño: Se introducen por fallas al traducir los requisitos en estructuras de solución correctas y completas, por inconsistencias tanto dentro de las especificaciones de diseño y como entre las especificaciones de diseño y los requisitos. Un error de requisitos o un error de diseño, que no se descubre sino hasta las pruebas de código fuente, puede ser muy costoso de corregir. De modo que es importante que la calidad de los requisitos y de los documentos del diseño se valoren pronto y con frecuencia.
- Los errores de instrumentación: Son los cometidos al traducir las especificaciones de diseño en código fuente. Estos errores pueden ocurrir en las declaraciones de datos, en las referencias a los datos, en la lógica del flujo de control, en expresiones computacionales, en interfaces entre subprogramas y en operaciones de entrada/salida. La calidad de los productos de trabajo generados durante el análisis y el diseño se puede estimar y mejorar utilizando procedimientos sistemáticos de control de calidad, mediante recorridos e inspecciones y por medio de verificaciones automatizadas para supervisar que sea consistente y que esté completo. Las técnicas para estimar y mejorar la calidad del código fuente incluyen los procedimientos sistemáticos de control de calidad, los recorridos inspecciones, el análisis estático, la ejecución simbólica, las pruebas de unidad y las pruebas de integración sistemáticas. Las técnicas de verificación formal se pueden usar para mostrar, de manera rigurosa, que un programa fuente se conforma con sus requisitos; la verificación formal también puede servir para guiar la síntesis sistemática de los programas fuente.

Relacion entre Defecto Falla Error

~~Error (error):~~
Es una equivocación cometida por un desarrollador. Algunos ejemplos de errores son: un error de tipeo, una malinterpretación de un requerimiento o de la funcionalidad de un método. El estándar 829 de la IEEE coincide con la definición de diccionario de error como “una idea falsa o equivocada”. Por ende un programa no puede tener o estar en un error, ya que los programas no tienen ideas; las ideas las tienen la gente.

~~Defecto (fault, defect):~~
Un error puede conducir a uno o más defectos. Un defecto se encuentra en un artefacto y puede definirse como una diferencia entre la versión correcta del artefacto y una versión incorrecta. De nuevo coincide con la definición de diccionario, “imperfección”. Por ejemplo, un defecto es haber utilizado el operador “<” en vez de “<=“.

Falla (failure):
En terminología IEEE, una falla es la discrepancia visible que se produce al ejecutar un programa con un defecto, respecto a la ejecución del programa correcto.
Es decir,una falla es el síntoma de un defecto.

ESTRATEGIAS DE PRUEBA DE SOFTWARE

17

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

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.

17.1.1 Verificación y validación

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

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.

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.

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.

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

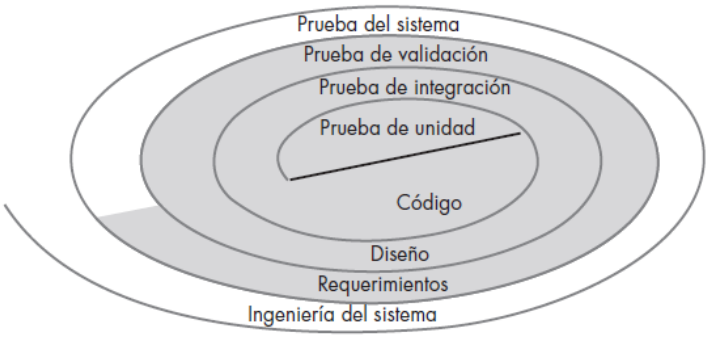
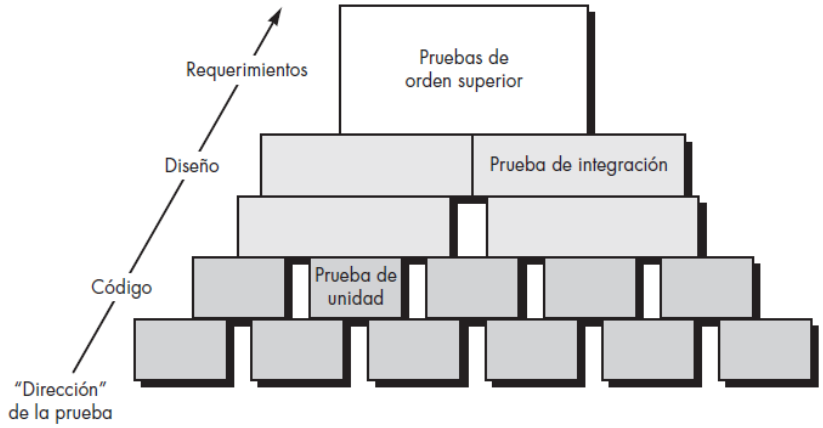


FIGURA 17.2
Pasos de la prueba del software



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.

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?

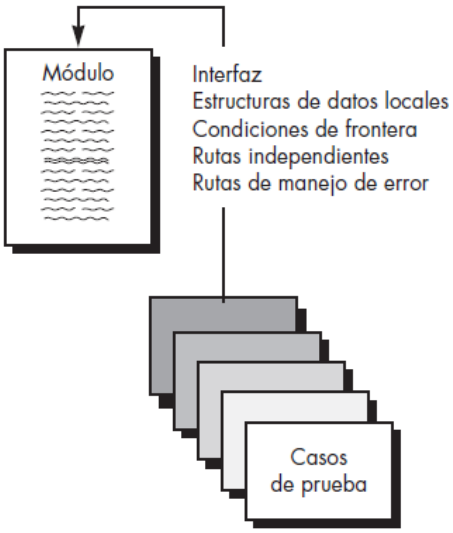
Una estrategia de prueba que eligen la mayoría de los equipos de software se coloca entre los dos extremos. Toma una visión incremental de las pruebas, comenzando con la de unidades de programa individuales, avanza hacia pruebas diseñadas para facilitar la integración de las unidades y culmina con pruebas 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.

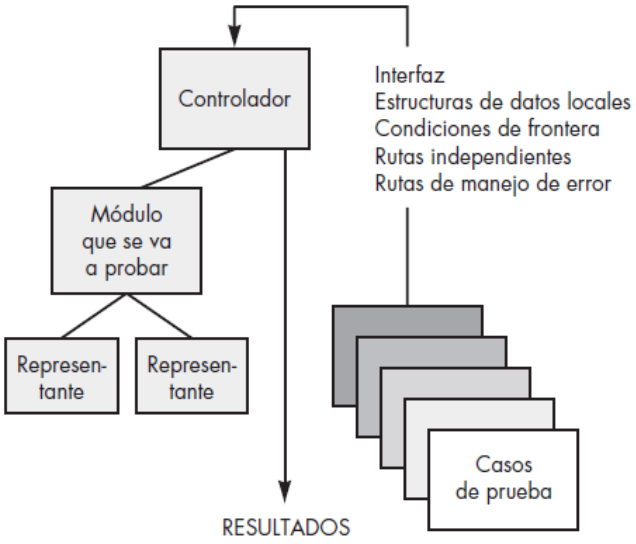
casos de
le
un
a garantizar
digo que

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-



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.



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.

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.

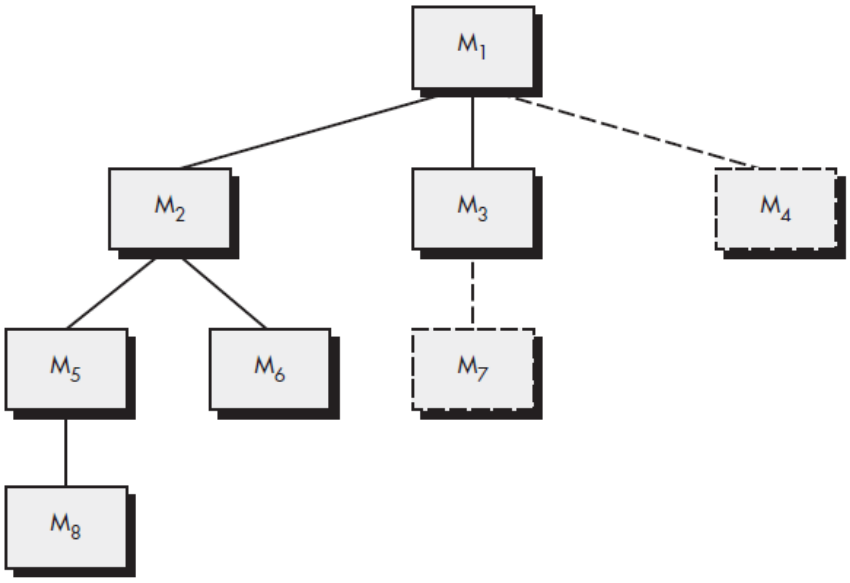
Las pruebas de integración son una técnica sistemática para construir la arquitectura del software mientras se llevan a cabo pruebas para descubrir errores asociados con la interfaz. El objetivo es tomar los componentes probados de manera individual y construir una estructura de programa que se haya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

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.

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-



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:

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.

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:

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.

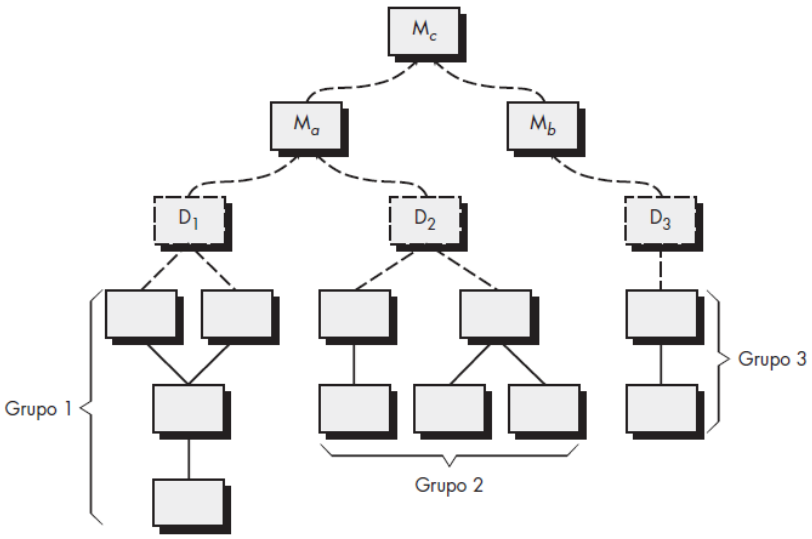
La integración sigue el patrón que se ilustra en la figura 17.6. Los componentes se combinan para formar los grupos 1, 2 y 3. Cada uno de ellos se prueba usando un controlador (que se muestra como un bloque rayado). Los componentes en los grupos 1 y 2 se subordinan a M_a. Los controladores D_1 y D_2 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.

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:

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.

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:

- *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.

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.

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.

pasos de
in intenta
al
en **los**
re las cosas
nte aparentes

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

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.

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.

17.7 PRUEBAS DEL SISTEMA

s impues-

itables".

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?".

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 procedimientos 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.

18.3 Prueba de caja blanca

Cita:

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

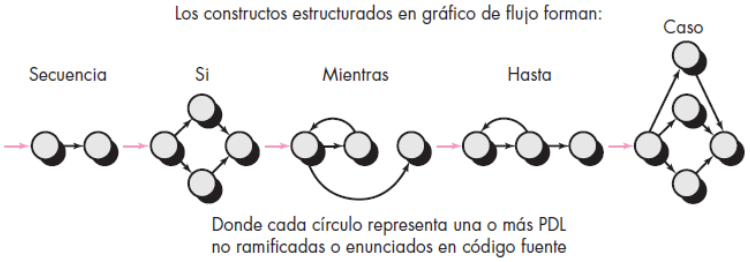
Boris Beizer

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 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.

Figura 18.1 Notación de gráfico de flujo



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.

ero encon-
o."

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:

tas
s pruebas
?

- ¿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.

Prácticas de mantenimiento

Su objetivo como analista de sistemas debe ser instalar o modificar sistemas que tengan una vida útil razonable. Lo ideal es crear un sistema cuyo diseño sea integral y con una suficiente visión a futuro suficiente para atender las necesidades actuales y proyectadas de los usuarios durante los años por venir. Recurra a su experiencia para anticipar esas necesidades y después agregar tanto flexibilidad como capacidad de adaptación al sistema. Entre mejor sea el diseño del sistema, más fácil será mantenerlo y menos dinero tendrá que invertir la empresa en ello.

Reducir los costos de mantenimiento es primordial, ya que el mero mantenimiento del software puede devorar hasta el 50 por ciento del presupuesto total de procesamiento de datos de una empresa. Los costos excesivos de mantenimiento son responsabilidad del diseñador del sistema, ya que cerca del 70 por ciento de los errores de software se atribuyen a un diseño de software inapropiado. Desde la perspectiva de sistemas, tiene sentido el hecho de que detectar y corregir los errores de diseño de software lo antes posible sea menos costoso que dejar que pasen desapercibidos hasta que sea necesario el mantenimiento.

La mayor parte de las veces el mantenimiento se realiza para mejorar el software existente y no para responder a una crisis o falla del sistema. El mantenimiento también se realiza para actualizar software en respuesta a los cambios en la organización. Este trabajo no es tan sustancial como mejorar el software, pero hay que hacerlo. El mantenimiento de emergencia y adaptativo representa menos de la mitad de todo el mantenimiento del sistema.

Parte del trabajo del analista de sistemas es asegurar que se implementen los canales y procedimientos adecuados para permitir la retroalimentación sobre las necesidades de mantenimiento y las acciones para satisfacerlas. Para los usuarios debe ser fácil comunicar los problemas y las sugerencias a los encargados de dar mantenimiento al sistema. Las soluciones son proveer a los usuarios el acceso vía correo electrónico al soporte técnico, así como permitirles descargar actualizaciones de productos o parches a través de la Web.

Auditoría

La auditoría es otra forma de asegurar la calidad de la información que contiene el sistema. En términos generales, la auditoría se refiere al proceso de hacer que un experto que no esté involucrado en el proceso de establecer o usar un sistema examine la información para evaluar su confiabilidad. Sin importar que la información resulte confiable o no, el hallazgo sobre su confiabilidad se comunica a los demás con el fin de que actúen en consecuencia.

Para los sistemas de información, en general hay dos tipos de auditores: internos y externos. El hecho de determinar si ambos son necesarios para el sistema que usted diseñe dependerá del tipo de sistema. Los auditores internos trabajan para la misma organización que posee el sistema de información, mientras que los auditores externos (también llamados independientes) se contratan del exterior.

Los auditores externos se utilizan cuando el sistema de información procesa datos que influyen en los estados financieros de la empresa; los externos realizan una auditoría sobre el sistema para asegurar la imparcialidad de los estados financieros que se producen. También se pueden usar cuando ocurre algo fuera de lo normal en el que hay empleados de la empresa involucrados, como una sospecha de fraude por computadora o malversación de fondos.

Los auditores internos estudian los controles que se utilizan en el sistema de información para asegurarse de que sean adecuados y que estén realizando la función esperada. También evalúan la conveniencia de los controles de seguridad. Aunque trabajan para la misma organización, los auditores internos no se reportan con la persona responsable del sistema al que están auditando. Con frecuencia, el trabajo de los auditores internos es más profundo que el de los auditores externos.