CAPÍTULO

17 TÉCNICAS DE PRUEBA DEL SOFTWARE

NUNCA se dará suficiente importancia a las pruebas del software y sus implicaciones en la calidad del software. Citando a Deutsch [DEU79]:

> El desarrollo de sistemas de software implica una serie de actividades de producción en las que las posibilidades de que aparezca el fallo humano son enormes. Los errores pueden empezar a darse desde el primer momento del proceso, en el que los objetivos....pueden estar especificados de forma errónea o imperfecta, así como [en] posteriores pasos de diseño y desarrollo...Debido a la imposibilidad humana de trabajar y comunicarse de forma perfecta, el desarrollo de software ha de ir acompañado de una actividad que garantice la calidad.

Las pruebas del software son un elemento crítico para la garantía de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación.

La creciente percepción del software como un elemento del sistema y la importancia de los «costes» asociados a un fallo del propio sistema, están motivando la creación de pruebas minuciosas y bien planificadas. No es raro que una organización de desarrollo de software emplee entre el 30 y el 40 por ciento del esfuerzo total de un proyecto en las pruebas. En casos extremos, las pruebas del software para actividades críticas (por ejemplo, control de tráfico aéreo, control de reactores nucleares) puede costar ¡de tres a cinco veces más que el resto de los pasos de la ingeniería del software juntos!

VISTAZO RÁPIDO

¿Qué es? Una vez generado el código fuente, el software debe ser probado para descubrir (y corregir) el máximo de errores posibles antes de su entrega al cliente. Nuestro objetivo es diseñar una serie de casos de prueba que tengan una alta probabilidad de encontrar errores —pero ¿cómo conseguirlo?— Aquí es donde aplicamos las técnicas de pruebas del software. Estas técnicas facilitan una guía sistemática para diseñar pruebas que: (1) comprueben la lógica interna de los componentes software, y (2) verifiquen los dominios de entrada y salida del programa para descubrir errores en la funcionalidad, el comportamiento y rendimiento

¿Quién lo hace? Durante las primeras etapas de la prueba, es el ingeniero del software quien realiza todas las pruebas. Sin embargo, conforme progresa el proceso de prueba, los especialistas en pruebas se van incorporando.

¿Por qué es importante? Las revisiones y otras actividades SQA descubren errores, pero no son suficientes. Cada vez que el programa se ejecuta, el cliente lo está probando. Por lo tanto, debemos hacer un intento especial por encontrar y corregir todos los errores antes de entregar el programa al cliente. Con el objetivo de encontrar el mayor número posible de errores, las pruebas deben conducirse sistemáticamente. y los casos de prueba deben diseñarse utilizando técnicas definidas.

¿Cuáles son los pasos? El software debe probarse desde dos perspectivas diferentes: (1) la lógica interna del programa se comprueba utilizando técnicas de diseño de casos de prueba de «caja blanca». Los requisitos del software se comprueban utilizando técnicas de diseño de casos de prueba de «caja negra». En ambos casos, se intenta encontrar el mayor número de errores con la menor cantidad de esfuerzo y tiempo.

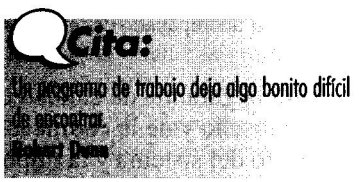
¿Cuál es el producto obtenido? Se define y documenta un conjunto de casos de prueba, diseñados para comprobar la lógica interna y los requisitos externos. Se determinan los resultados esperados y se guardan los resultados realmente obtenidos.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Cuando comienzas la prueba, debes cambiar tu punto de vista. Intenta «romper» con firmeza el software. Diseña casos de prueba de una forma disciplinada y revisa que dichos casos de prueba abarcan todo lo desarrollado.

En este capítulo, veremos los fundamentos de las pruebas del software y las técnicas de diseño de casos de prueba.

17.1 FUNDAMENTOS DE LAS PRUEBAS DEL SOFTWARE

Las pruebas presentan una interesante anomalía para el ingeniero del software. Durante las fases anteriores de definición y de desarrollo, el ingeniero intenta construir el software partiendo de un concepto abstracto y llegando a una implementación tangible. A continuación, llegan las pruebas. El ingeniero crea una serie de casos de prueba que intentan «demoler» el software construido. De hecho, las pruebas son uno de los pasos de la ingeniería del software que se puede ver (por lo menos, psicológicamente) como destructivo en lugar de constructivo.



Los ingenieros del software son, por naturaleza, personas constructivas. Las pruebas requieren que se descarten ideas preconcebidas sobre la «corrección» del software que se acaba de desarrollar y se supere cualquier conflicto de intereses que aparezcan cuando se descubran errores. Beizer [BEI90] describe eficientemente esta situación cuando plantea:

> Existe un mito que dice que si fuéramos realmente buenos programando, no habría errores que buscar. Si tan sólo fuéramos realmente capaces de concentrarnos, si todo el mundo empleara técnicas de codificación estructuradas, el diseño descendente, las tablas de decisión, si los programas se escribieran en un lenguaje apropiado, si tuviéramos siempre la solución más adecuada, entonces no habría errores. Así es el mito. Según el mito, existen errores porque somos malos en lo que hacemos, y si somos malos en lo que hacemos, deberíamos sentirnos culpables por ello. Por tanto, las pruebas, con el diseño de casos de prueba, son un reconocimiento de nuestros fallos, lo que implica una buena dosis de culpabilidad. Y lo tediosas que son las pruebas son un justo castigo a nuestros errores. ¿Castigados por qué? ¿Por ser humanos? ¿Culpables por qué? ¿Por no conseguir una perfección inhumana? ¿Por no poder distinguir entre lo que otro programador piensa y lo que dice? ¿Por no tener telepatía? ¿Por no resolver los problemas de comunicación humana que han estado presentes... durante cuarenta siglos?

¿Deben infundir culpabilidad las pruebas? ¿Son realmente destructivas? La respuesta a estas preguntas es: ¡No! Sin embargo, los objetivos de las pruebas son algo diferentes de lo que podríamos esperar.

17.1.1. Objetivos de las pruebas

En un excelente libro sobre las pruebas del software, Glen Myers [MYE79] establece varias normas que pueden servir acertadamente como objetivos de las pruebas:

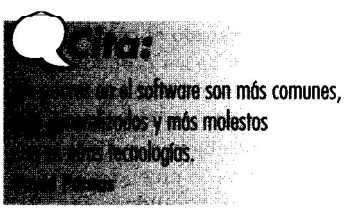
1. La prueba es el proceso de ejecución de un programa con la intención de descubrir un error.
2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

¿Cuál es nuestro primer objetivo cuando probamos el software?

Los objetivos anteriores suponen un cambio dramático de punto de vista. **Nos** quitan la idea que, normalmente, tenemos de que una prueba tiene éxito si no descubre errores.

Nuestro objetivo es diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y de esfuerzo.

Si la prueba se lleva a cabo con éxito (de acuerdo con el objetivo anteriormente establecido), descubrirá errores en el software. Como ventaja secundaria, la prueba demuestra hasta qué punto las funciones del software parecen funcionar de acuerdo con las especificaciones y parecen alcanzarse los requisitos de rendimiento. Además, los datos que se van recogiendo a medida que se lleva a cabo la prueba proporcionan una buena indicación de la fiabilidad del software y, de alguna manera, indican la calidad del software como un todo. Pero, la prueba no puede asegurar la ausencia de defectos; sólo puede demostrar que existen defectos en el software.



17.1.2. Principios de las pruebas

Antes de la aplicación de métodos para el diseño de casos de prueba efectivos, un ingeniero del software deberá entender los principios básicos que guían las pruebas del software. Davis [DAV95] sugiere un conjunto[1] de principios de prueba que se han adaptado para usarlos en este libro:

- ***A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente.*** Como hemos visto, el objetivo de las pruebas del software es descubrir errores. Se entiende que los defectos más graves (desde el punto de vista del cliente) son aquellos que impiden al programa cumplir sus requisitos.

[1] Aquí se presenta sólo un pequeño subconjunto de los principios de ingeniería de pruebas de Davis. Para obtener más información, vea [DAV96].

- *Las pruebas deberían planificarse mucho antes de que empiecen.* La planificación de las pruebas (Capítulo 18) pueden empezar tan pronto como esté completo el modelo de requisitos. La definición detallada de los casos de prueba puede empezar tan pronto como el modelo de diseño se ha consolidado. Por tanto, se pueden planificar y diseñar todas las pruebas antes de generar ningún código.
- *El principio de Pareto es aplicable a la prueba del software.* Dicho de manera sencilla, el principio de Pareto implica que al 80 por 100 de todos los errores descubiertos durante las pruebas se les puede hacer un seguimiento hasta un 20 por 100 de todos los módulos del programa. El problema, por supuesto, es aislar estos módulos sospechosos y probarlos concienzudamente.
- *Las pruebas deberían empezar por «lo pequeño» y progresar hacia «lo grande».* Las primeras pruebas planeadas y ejecutadas se centran generalmente en módulos individuales del programa. A medida que avanzan las pruebas, desplazan su punto de mira en un intento de encontrar errores en grupos integrados de módulos y finalmente en el sistema entero (Capítulo 18).
- *No son posibles las pruebas exhaustivas.* El número de permutaciones de caminos para incluso un programa de tamaño moderado es excepcionalmente grande (vea la Sección 17.2 para un estudio más avanzado). Por este motivo, es imposible ejecutar todas las combinaciones de caminos durante las pruebas. Es posible, sin embargo, cubrir adecuadamente la lógica del programa y asegurarse de que se han aplicado todas las condiciones en el diseño a nivel de componente.
- *Para ser más eficaces, las pruebas deberían ser realizadas por un equipo independiente.* Por «mas eficaces» queremos referirnos a pruebas con la más alta probabilidad de encontrar errores (el objetivo principal de las pruebas). Por las razones que se expusieron anteriormente en este capítulo, y que se estudian con más detalle en el Capítulo 18, el ingeniero del software que creó el sistema no es el más adecuado para llevar a cabo las pruebas para el software.

17.1.3. Facilidad de prueba

En circunstancias ideales, un ingeniero del software diseña un programa de computadora, un sistema o un producto con la «facilidad de prueba» en mente. Esto permite a los encargados de las pruebas diseñar casos de prueba más fácilmente. Pero, ¿qué es la «facilidad de prueba» James Bach[2] describe la facilidad de prueba de la siguiente manera:

La facilidad de prueba del software es simplemente la facilidad con la que se puede probar un programa de computadora. Como la prueba es tan profundamente difícil, merece la pena saber qué se puede hacer para hacerlo más sencillo. A veces los programadores están dispuestos a hacer cosas que faciliten el proceso de prueba y una lista de comprobación de posibles puntos de diseño, características, etc., puede ser útil a la hora de negociar con ellos.

Referencia Web

Un útil documento titulado «Perfeccionando la facilidad de la prueba del software» puede encontrarse en: **www.stlabs.com/newsletters/testnet/docs/testability.htm.**

Existen, de hecho, métricas que podrían usarse para medir la facilidad de prueba en la mayoría de sus aspectos. A veces, la facilidad de prueba se usa para indicar lo adecuadamente que un conjunto particular de pruebas va a cubrir un producto. También es empleada por los militares para indicar lo fácil que se puede comprobar y reparar una herramienta en plenas maniobras. Esos dos significados no son lo mismo que «facilidad de prueba del software». La siguiente lista de comprobación proporciona un conjunto de características que llevan a un software fácil de probar.

Operatividad. «Cuanto mejor funcione, más eficientemente se puede probar.»

- El sistema tiene pocos errores (los errores añaden sobrecarga de análisis y de generación de informes al proceso de prueba).
- Ningún error bloquea la ejecución de las pruebas
- El producto evoluciona en fases funcionales (permite simultanear el desarrollo y las pruebas).

Observabilidad. «Lo que ves es lo que pruebas.»

- Se genera una salida distinta para cada entrada.
- Los estados y variables del sistema están visibles o se pueden consultar durante la ejecución.
- Los estados y variables anteriores del sistema están visibles o se pueden consultar (por ejemplo, registros de transacción).
- Todos los factores que afectan a los resultados están visibles.
- Un resultado incorrecto se identifica fácilmente.
- Los errores internos se detectan automáticamente a través de mecanismos de auto-comprobación.
- Se informa automáticamente de los errores internos.
- El código fuente es accesible.

[2] Los siguientes párrafos son copyright de James Bach, 1994, y se han adaptado de una página de lntemet que apareció inicialmente en el grupo de noticias comp.software-eng. Este material se ha usado con permiso.

PUNTO CLAVE

«La facilidad de prueba» ocurre como el resultado de un buen diseño. El diseño de datos, de la arquitectura, de las interfaces y de los componentes de detalle pueden facilitar la prueba o hacerlo más difícil.

Controlabilidad. «Cuanto mejor podamos controlar el software, más se puede automatizar y optimizar.»

- Todos los resultados posibles se pueden generar a través de alguna combinación de entrada.
- Todo el código es ejecutable a través de alguna combinación de entrada.
- El ingeniero de pruebas puede controlar directamente los estados y las variables del hardware y del software.
- Los formatos de las entradas y los resultados son consistentes y estructurados.
- Las pruebas pueden especificarse, automatizarse y reproducirse convenientemente.

Capacidad de descomposición. «Controlando el ámbito de las pruebas, podemos aislar más rápidamente los problemas y llevar a cabo mejores pruebas de regresión.»

- El sistema software está construido con módulos independientes.
- Los módulos del software se pueden probar independientemente

Simplicidad. «Cuanto menos haya que probar, más rápidamente podremos probarlo.»

- Simplicidad funcional (por ejemplo, el conjunto de características es el mínimo necesario para cumplir los requisitos).
- Simplicidad estructural (por ejemplo, la arquitectura es modularizada para limitar la propagación de fallos).
- Simplicidad del código (por ejemplo, se adopta un estándar de código para facilitar la inspección y el mantenimiento).

Estabilidad. «Cuanto menos cambios, menos interrupciones a las pruebas.»

- Los cambios del software son infrecuentes.
- Los cambios del software están controlados.
- Los cambios del software no invalidan las pruebas existentes.
- El software se recupera bien de los fallos.

Facilidad de comprensión. «Cuanta más información tengamos, más inteligentes serán las pruebas.»

- El diseño se ha entendido perfectamente.
- Las dependencias entre los componentes internos, externos y compartidos se han entendido perfectamente.
- Se han comunicado los cambias del diseño.
- La documentación técnica es instantáneamente accesible.
- La documentación técnica está bien organizada.
- La documentación técnica es específica y detallada.
- La documentación técnica es exacta.

Los atributos sugeridos por Bach los puede emplear el ingeniero del software para desarrollar una configuración del software (por ejemplo, programas, datos y documentos) que pueda probarse.

¿Cuáles son las características de una «buena» prueba?

¿Y qué pasa con las pruebas propias? Kaner, Falk y Nguyen [KAN93] sugieren los siguientes atributos de una «buena» prueba:

1. Una buena prueba tiene una alta probabilidad de encontrar un error. Para alcanzar este objetivo, el responsable de la prueba debe entender el software e intentar desarrollar una imagen mental de cómo podría fallar el software.
2. Una buena prueba no debe ser redundante. El tiempo y los recursos para las pruebas son limitados. No hay motivo para realizar una prueba que tiene el mismo propósito que otra. Todas las pruebas debenan tener un propósito diferente (incluso si es sutilmente diferente). Por ejemplo, un módulo del software de *HogarSeguro* (estudiado en anteriores capítulos) está diseñado para reconocer una contraseña de usuario para activar o desactivar el sistema. En un esfuerzo por descubrir un error en la entrada de la contraseña, el responsable de la prueba diseña una serie de pruebas que introducen una secuencia de contraseñas. Se introducen contraseñas válidas y no válidas (secuencias de cuatro números) en pruebas separadas. Sin embargo, cada contraseña válida/no válida debería analizar un modo diferente de fallo. Por ejemplo, la contraseña no válida 1234 no debería ser aceptada por un sistema programado para reconocer 8080 como la contraseña correcta. Si 1234 es aceptada, tenemos presente un error. Otra prueba, digamos 1235, tendría el mismo propósito que 1234 y es, por tanto, redundante. Sin embargo, la entrada no válida 8081 u 8180 tiene una sutil diferencia, intentando demostrar que existe un error para las contraseñas «parecidas» pero no idénticas a la contraseña correcta.
3. Una buena prueba debería ser «la mejor de la cosecha» [KAN93]. En un grupo de pruebas que tienen propósito similar, las limitaciones de tiempo y recursos pueden abogar por la ejecución de sólo un subconjunto de estas pruebas. En tales casos, se debena emplear la prueba que tenga la más alta probabilidad de descubrir una clase entera de errores.
4. Una buena prueba no debería ser ni demasiado sencilla ni demasiado compleja. Aunque es posible a veces combinar una serie de pruebas en un caso de prueba, los posibles efectos secundarios de este enfoque pueden enmascarar errores. En general, cada prueba debería realizarse separadamente.

17.2. DISEÑO DE CASOS DE PRUEBA

El diseño de pruebas para el software o para otros productos de ingeniería puede requerir tanto esfuerzo como el propio diseño inicial del producto. Sin embargo, los ingenieros del software, por razones que ya hemos tratado, a menudo tratan las pruebas como algo sin importancia, desarrollando casos de prueba que «parezcan adecuados», pero que tienen poca garantía de ser completos. Recordando el objetivo de las pruebas, debemos diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible.

Cita:

Existe una sola regla para el diseño de casos de prueba: cubrir todas las posibilidades, sin hacer demasiados casos de prueba.
Tsuneo Yamaura

Cualquier producto de ingeniería (y de muchos otros campos) puede probarse de una de estas dos formas: (1) conociendo la función específica para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa y, al mismo, tiempo buscando errores en cada función; (2) conociendo el funcionamiento del producto, *se* pueden desarrollar pruebas que aseguren que «todas las piezas encajan», o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada. El primer enfoque de prueba se denomina prueba de caja negra y el segundo, prueba de caja blanca.

Referencia Web

La página de técnicas de prueba es una excelente fuente de información sobre los métodos de prueba:
www.testworks.com/News/TTN-Online/

Cuando se considera el software de computadora, la *prueba de caja negra* se refiere a las pruebas que se llevan a cabo sobre la interfaz del software. O sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce un resultado correcto, así como que la integridad de la información externa (por ejemplo, archivos de datos) se mantiene. Una prueba de caja negra examina algunos aspectos del modelo fundamental del sistema sin tener mucho en cuenta la estructura lógica interna del software.

La *prueba de caja blanca* del software se basa en el minucioso examen de los detalles procedimentales. Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el «estado del programa» en varios puntos para determinar si el estado real coincide con el esperado o mencionado.

PUNTO CLAVE

las pruebas de caja blanca son diseñadas después de que exista un diseño de componente (o código fuente). El detalle de la lógica del programa debe estar disponible.

A primera vista parecería que una prueba de caja blanca muy profunda nos llevaría a tener «programas cien por cien correctos». Todo lo que tenemos que hacer es definir todos los caminos lógicos, desarrollar casos de prueba que los ejerciten y evaluar los resultados, es decir, generar casos de prueba que ejerciten exhaustivamente la lógica del programa. Desgraciadamente, la prueba exhaustiva presenta ciertos problemas logísticos. Incluso para pequeños programas, el número de caminos lógicos posibles puede ser enorme. Por ejemplo, considere un programa de 100 líneas de código en lenguaje C. Después de la declaración de algunos datos básicos, el programa contiene dos bucles que se ejecutan de 1 a 20 veces cada uno, dependiendo de las condiciones especificadas en la entrada. Dentro del bucle interior, se necesitan cuatro construcciones *if-then-else*. ¡Existen aproximadamente 10^{14} caminos posibles que se pueden ejecutar en este programa!

PUNTO CLAVE

No es posible una prueba exhaustiva sobre todos los caminos del programa, porque el número de caminos es simplemente demasiado grande.

Para poner de manifiesto el significado de este número, supongamos que hemos desarrollado un procesador de pruebas mágico («mágico» porque no existe tal procesador) para hacer una prueba exhaustiva. El procesador puede desarrollar un caso de prueba, ejecutarlo y evaluar los resultados en un milisegundo. Trabajando las 24 horas del día, 365 días al año, el procesador trabajaría durante 3170 años para probar el programa. Esto irremediablemente causaría estragos en la mayoría de los planes de desarrollo. La prueba exhaustiva es imposible para los grandes sistemas software.

La prueba de caja blanca, **sin** embargo, no se debe desechar como impracticable. Se pueden elegir y ejercitar una serie de caminos lógicos importantes.

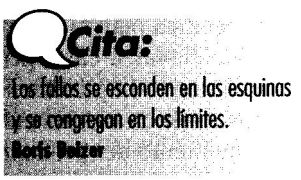
Se pueden comprobar las estructuras de datos más importantes para verificar **su** validez. Se pueden combinar los atributos de la prueba de caja blanca así como los de caja negra, para llegar a un método que valide la interfaz del software y asegure selectivamente que el funcionamiento interno del software es correcto.

17.3 PRUEBA DE CAJA BLANCA

La prueba de caja blanca, denominada a veces *prueba de caja de cristal* es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que **(1)** garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo; (2) ejerciten todas las decisiones lógicas en sus vertientes verdadera **y** falsa; **(3)** ejecuten todos los bucles en sus límites y con sus límites operacionales; y **(4)** ejerciten las estructuras internas de datos para asegurar su validez.

En este momento, se puede plantear un pregunta razonable: ¿Por qué emplear tiempo **y** energía preocupándose de (y probando) las minuciosidades lógicas cuando podríamos emplear mejor el esfuerzo asegurando que se han alcanzado los requisitos del programa? O, dicho de otra forma, ¿por qué no empleamos todas nuestras energías en la prueba de caja negra? La respuesta se encuentra en la naturaleza misma de los defectos del software (por ejemplo, [JON81]):

- Los errores lógicos **y** las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa. Los errores tienden a introducirse en nuestro trabajo cuando diseñamos e implementamos funciones, condiciones o controles que se encuentran fuera de lo normal. El procedimiento habitual tiende a hacerse más comprensible (y bien examinado), mientras que el procesamiento de «casos especiales» tiende a caer en el caos.
- A menudo creemos que un camino lógico tiene **pocas** posibilidades de ejecutarse cuando, de hecho, se puede ejecutar de forma normal. El flujo lógico de un programa a veces no es nada intuitivo, lo que significa que nuestras suposiciones intuitivas sobre el flujo de control **y** los datos nos pueden llevar a tener errores de diseño que sólo se descubren cuando comienza la prueba del camino.
- *Los errores tipográficos son aleatorios.* Cuando se traduce un programa a código fuente en un lenguaje de programación, es muy probable que se den algunos errores de escritura. Muchos serán descubiertos por los mecanismos de comprobación de sintaxis, pero otros permanecerán sin detectar hasta que comience la prueba. Es igual de probable que haya un error tipográfico en un oscuro camino lógico que en un camino principal.



17.4 PRUEBA DEL CAMINO BÁSICO

La *prueba del camino básico* es una técnica de prueba de caja blanca propuesta inicialmente por Tom McCabe [MCC76]. El método del camino básico permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental **y** usar esa medida como guía para la definición de un *conjunto básico* de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa.

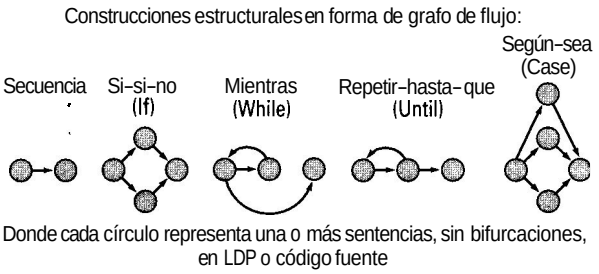


FIGURA 17.1. Notación de grafo **de flujo.**

17.4.1. Notación de grafo de flujo

Antes de considerar el método del camino básico se debe introducir una sencilla notación para la representación del flujo de control, denominada *grafo de flujo* (o *grafo del programa*)[3]. El grafo de flujo representa el flujo de control lógico mediante la notación ilustrada en la Figura 17.1. Cada construcción estructurada (Capítulo 16) tiene su correspondiente símbolo en el grafo del flujo.

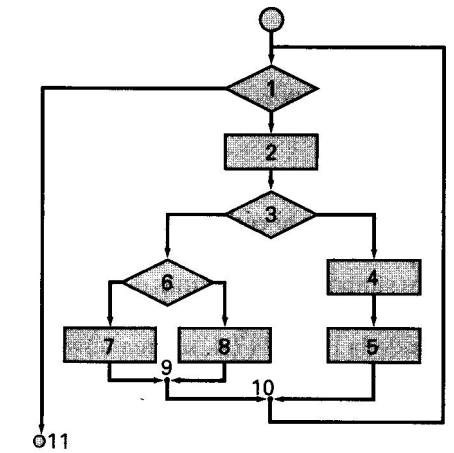
Dibujo un yrofo de flujo cuando lo lógica de lo estructura de control de un módulo seo complejo. El grafo de flujo te permite trazar más fácilmente los cominos del programa.

Para ilustrar el uso de un grafo de flujo, consideremos la representación del diseño procedimental en la Figura 17.2a. En ella, se usa un diagrama de flujo para

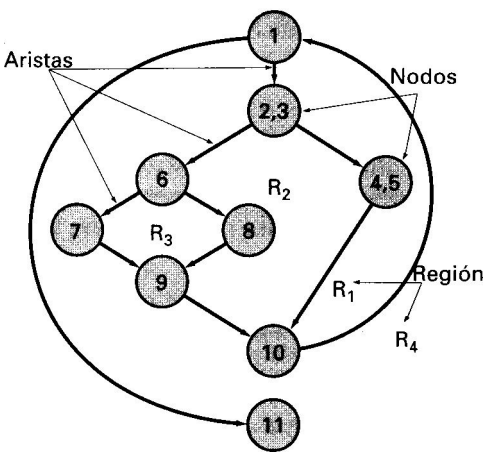
3 Realmente, el método del camino básico se puede llevar a cabo sin usar grafos de flujo. Sin embargo, sirve como herramienta Útil para ilustrar el método.

representar la estructura de control del programa. En la Figura 17.2b se muestra el grafo de flujo correspondiente al diagrama de flujo (suponiendo que no hay condiciones compuestas en los rombos de decisión del diagrama de flujo).

En la Figura 17.2b, cada círculo, denominado *nodo* del grafo de flujo, representa una o más sentencias procedimentales. Un solo nodo puede corresponder a una secuencia de *cuadros* de proceso y a un rombo de decisión. Las flechas del grafo de flujo, denominadas *aristas* o *enlaces*, representan flujo de control y son análogas a las flechas del diagrama de flujo. Una arista debe terminar en un nodo, incluso aunque el nodo no represente ninguna sentencia procedimental (por ejemplo, véase el símbolo para la construcción Si–entonces–si–no). Las áreas delimitadas por aristas y nodos se denominan *regiones*. Cuando contabilizamos las regiones incluimos el área exterior del grafo, contando como otra región más[4].



a) Diagrama de flujo



b) Grafo de flujo

FIGURA 17.2.

Cuando en un diseño procedimental se encuentran condiciones compuestas, la generación del grafo de flujo se hace un poco más complicada. Una condición compuesta se da cuando aparecen uno o más operadores (OR, AND, NAND, NOR lógicos) en una sentencia condicional. En la Figura 17.3 el segmento en LDP se traduce en el grafo de flujo anexo. Se crea un nodo aparte por cada una de las condiciones *a* y *b* de la sentencia **SI** *a* O *b*. Cada nodo que contiene una condición se denomina *nodo predicado* y está caracterizado porque dos o más aristas emergen de él.

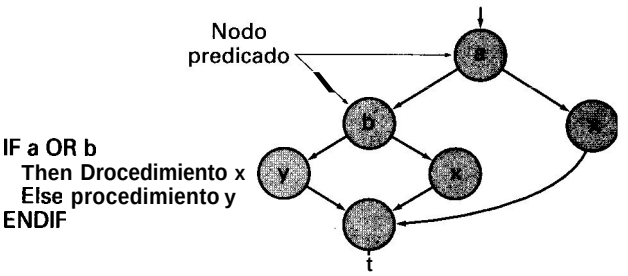


FIGURA 17.3. Lógica compuesta.

17.4.2. Complejidad ciclomática

La *complejidad ciclomática* es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Cuando se usa en el contexto del método de prueba del camino básico, el valor calculado como complejidad ciclomática define el número de *caminos independientes* del *conjunto básico* de un programa y nos da un límite superior para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez.

Un *camino independiente* es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una nueva condición. En términos del grafo de flujo, un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino. Por ejemplo, para el grafo de flujo de la Figura 17.2b, un conjunto de caminos independientes sería:

camino 1: 1-11
camino 2: 1-2-3-4-5-10-1-11
camino 3: 1-2-3-6-8-9-10-1-11
camino 4: 1-2-3-6-7-9-10-1-11

Fíjese que cada nuevo camino introduce una nueva arista. El camino

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

no se considera un camino independiente, ya que es simplemente una combinación de caminos ya especificados y no recorre ninguna nueva arista.

[4] En la Sección 17.6.1 viene un estudio mas detallado de los grafos y su empleo en las pruebas.

CONSEJO

La complejidad ciclomática es una métrica útil para predecir las módulos que san más propensas a error. Puede ser usada tanto para planificar pruebas cama para diseñar casas de prueba.

Los caminos 1,2, 3 y 4 definidos anteriormente componen un *conjunto básico* para el grafo de flujo de la Figura 17.2b. Es decir, si se pueden diseñar pruebas que fuercen la ejecución de esos caminos (un conjunto básico), se garantizará que se habrá ejecutado al menos una vez cada sentencia del programa y que cada condición se habrá ejecutado en sus vertientes verdadera y falsa.

Se debe hacer hincapié en que el conjunto básico no es Único. De hecho, de un mismo diseño procedimental se pueden obtener varios conjuntos básicos diferentes.

¿Cómo se cálcula la complejidad titlomátita?

¿Cómo sabemos cuántos caminos hemos de buscar? El cálculo de la complejidad ciclomática nos da la respuesta. La complejidad ciclomática está basada en la teoría de grafos y nos da una métrica del software extremadamente Útil. La complejidad se puede calcular de tres formas:

1. El número de regiones del grafo de flujo coincide con la complejidad ciclomática.
2. La complejidad ciclomática, *V(G)*, de un grafo de flujo *G* se define como.

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

   donde *A* es el número de aristas del grafo de flujo y *N* es el número de nodos del mismo.
3. La complejidad ciclomática, *V(G)*, de un grafo de flujo *G* también se define como

   $$V(G) = P + 1$$

   donde *P* es el número de nodos predicado contenidos en el grafo de flujo *G*.

PUNTO CLAVE

La complejidad ciclomática determina el número de casos de prueba que deben ejecutarse para garantizar que todas las sentencias de un componente han sido ejecutadas al menos una vez.

Refiriéndonos de nuevo al grafo de flujo de la Figura 17.2b, la complejidad ciclomática se puede calcular mediante cualquiera de los anteriores algoritmos:

1. el grafo 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 grafo de flujo de la Figura 17.2b es 4.

Es más, el valor de *V(G)* nos da un límite superior para el número de caminos independientes que componen el conjunto básico y, consecuentemente, un valor límite superior para el número de pruebas que se deben diseñar y ejecutar para garantizar que se cubren todas las sentencias del programa.

17.4.3. Obtención de casos de prueba

El método de prueba de camino básico se puede aplicar a un diseño procedimental detallado o a un código fuente. En esta sección, presentaremos la prueba del camino básico como una serie de pasos. Usaremos el procedimiento **media,** representado en LDP en la Figura 17.4, como ejemplo para ilustrar todos los pasos del método de diseño de casos de prueba. Se puede ver que **media,** aunque con un algoritmo extremadamente simple, contiene condiciones compuestas y bucles.

Cita:

Errar es humano, encontrar un fallo, divino.
Robert Dunn

1. **Usando el diseño o el código como base, dibujamos el correspondiente grafo de flujo.** Creamos un grafo usando los símbolos y las normas de construcción presentadas en la Sección 16.4.1. Refiriéndonos al LDP de **media** de la Figura 17.4, se crea un grafo de flujo numerando las sentencias de LDP que aparecerán en los correspondientes nodos del grafo de flujo. El correspondiente grafo de flujo aparece en la Figura 17.5.

```
PROCEDURE media;
    * Este procedimiento calcula la media de 100 o menos
      números que se encuentren entre unos limites;
      también calcula el total de entradas y el total
      de números válidos.

      INFERFACE RETURNS media, total. entrada, total. válido;
      INTERFACE ACEPTS valor, mínimo, máximo;

      TYPE valor [1:100] IS SCALAR ARRAY;
      TYPE media, total. entrada, total. válido;
         Mínimo, máximo, suma IS SCALAR:
      TYPE i IS INTEGER;
      i = 1;
1     total, entrada =total, válido = 0;          2
      suma =0;
      DO WHILE VALOR [i] < > -999 and total entrada < 100   3
4  ->    Incrementar total entrada en 1;
         IF valor [i] > = mínimo  AND  valor [i] < = maximo   6
5  /        THEN incrementar total.válido en 1;
   7           suma = suma +valor [i];
            ELSE ignorar
   8     ENDIF
         Incrementar i en 1;
   9  ENODO
      If total valido > 0   10
         THEN media = suma/total valido,
12 ->    ELSE media = -999,
   13 ENDIF
END media
```

FIGURA 17.4. LDP para diseño de pruebas con nodos no identificados.

2. **Determinamos la complejidad ciclomática del grafo de flujo resultante.** La complejidad ciclomática, *V(G),* se determina aplicando uno de los algoritmos descritos en la Sección 17.5.2.. Se debe tener en cuenta que se puede determinar *V(G)* sin desarrollar el grafo de flujo, contando todas las sentencias condicionales del LDP (para el procedimiento **media** las condiciones compuestas cuentan como 2) y añadiendo 1.

   En la Figura 17.5,

   V(G) = 6 regiones

   V(G) = 17 aristas – 13 nodos +2 = 6

   V(G) = *5* nodos predicado + 1 = 6

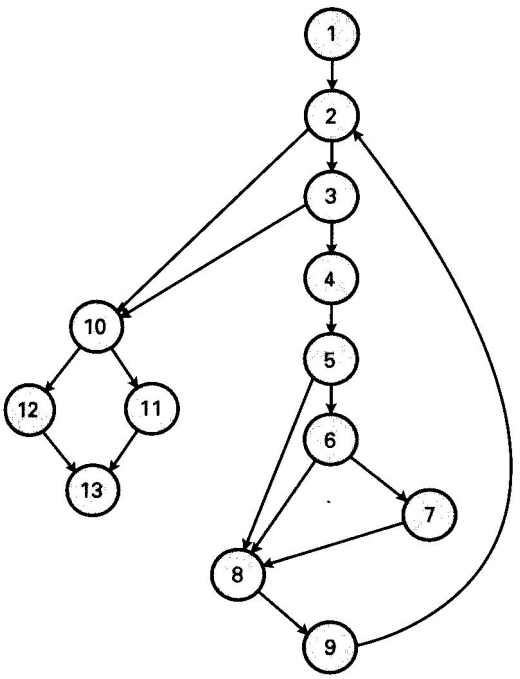


FIGURA 17.5. Grafo de flujo del procedimiento media.

3. **Determinamos un conjunto básico de caminos linealmente independientes.** El valor de V(G) nos da el número de caminos linealmente independientes de la estructura de control del programa. En el caso del procedimiento **media,** hay que especificar seis caminos:

   camino 1: 1-2-10-11-13

   camino 2: 1-2-10-12-13

   camino 3: 1-2-3-10-11-13

   camino 4: 1-2-3-4-5-8-9-2-...

   camino *5:* 1-2-3-4-5-6-8-9-2-...

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

   Los puntos suspensivos (...) que siguen a los caminos 4, 5 y 6 indican que cualquier camino del resto de la estructura de control es aceptable. Normalmente merece la pena identificar los nodos predicado para que sea más fácil obtener los casos de prueba. En este caso, los nodos 2, 3, *5*, 6 **y** 10 son nodos predicado.

4. **Preparamos los casos de prueba que forzarán la ejecución de cada camino del conjunto básico.** Debemos escoger los datos de forma que las condiciones de los nodos predicado estén adecuadamente establecidas, con el fin de comprobar cada camino. Los casos de prueba que satisfacen el conjunto básico previamente descrito son:

Caso de prueba del camino 1:

valor ***(k)*** = entrada válida, donde ***k*** < i definida a continuación

valor *(i)* = –999, donde 2 <= i <= 100

resultados esperados: media correcta sobre los ***k*** valores y totales adecuados.

Nota: el camino 1 no se puede probar por **sí** solo; debe ser probado como parte de las pruebas de los caminos 4, 5 y 6.

Cita:
Los ingenieros del software subestiman bastante el número de pruebas necesarias para verificar un programa simple.
Martyn Ould y Charles Unwin

Caso de prueba del camino 2:

valor (1) = –999

resultados esperados: media = –999; otros totales con sus valores iniciales

Caso de prueba del camino 3:

intento de procesar 101 *o* más valores

los primeros 100 valores deben ser válidos

resultados esperados: igual que en el caso de prueba 1

Caso de prueba del camino 4:

valor *(i)* = entrada válida donde i < 100

valor ***(k)*** < mínimo, para ***k*** < i

resultados esperados: media correcta sobre los **k** valores y totales adecuados

Caso de prueba del camino 5:

valor *(i)* = entrada válida donde i < 100

valor ***(k)*** > máximo, para ***k*** <= i

resultados esperados: media correcta sobre los *n* valores y totales adecuados

Caso de prueba del camino 6:

valor *(i)* = entrada válida donde i < 100

resultados esperados: media correcta sobre los *n* valores y totales adecuados

Ejecutamos cada caso de prueba y comparamos los resultados obtenidos con los esperados. Una vez terminados todos los casos de prueba, el responsable de la prueba podrá estar seguro de que todas las sentencias del programa se han ejecutado por lo menos una vez.

Es importante darse cuenta de que algunos caminos independientes (por ejemplo, el camino 1 de nuestro ejemplo) no se pueden probar de forma aislada. Es decir, la combinación de datos requerida para recorrer el camino no se puede conseguir con el flujo normal del programa. En tales casos, estos caminos se han de probar como parte de otra prueba de camino.

17.4.4. Matrices de grafos

El procedimiento para obtener el grafo de flujo, e incluso la determinación de un conjunto de caminos básicos, es susceptible de ser mecanizado. Para desarrollar una herramienta software que ayude en la prueba del camino básico, puede ser bastante Útil una estructura de datos denominada *matriz de grafo.*

Una matriz de grafo es una matriz cuadrada cuyo tamaño (es decir, el número de filas y de columnas) es igual al número de nodos del grafo de flujo. Cada fila y cada columna corresponde a un nodo específico y las entradas de la matriz corresponden a las *conexiones* (aristas) entre los nodos. En la Figura 17.6 se muestra un sencillo ejemplo de un grafo de flujo con su correspondiente matriz de grafo [BEI90].

En la figura, cada nodo del grafo de flujo está identificado por un número, mientras que cada arista lo está por su letra. Se sitúa una entrada en la matriz por cada conexión entre dos nodos. Por ejemplo, el nodo 3 está conectado con el nodo 4 por la arista b.

¿Qué es una matriz de grafos y cómo aplicarla en la prueba?

Hasta aquí, la matriz de grafo no es nada más que una representación tabular del grafo de flujo. Sin embargo, añadiendo un *peso de enlace* a cada entrada de la matriz, la matriz de grafo se puede convertir en una potente herramienta para la evaluación de la estructura de control del programa durante la prueba. El peso de enlace nos da información adicional sobre el flujo de control. En su forma más Sencilla, el peso de enlace es 1 (existe una conexión) ó 0 (no existe conexión). A los pesos de enlace se les puede asignar otras propiedades más interesantes:

- la probabilidad de que un enlace (arista) sea ejecutado;
- el tiempo de procesamiento asociado al recorrido de un enlace;
- la memoria requerida durante el recorrido de un enlace; y
- los recursos requeridos durante el recorrido de un enlace.

Para ilustrarlo, usaremos la forma más simple de peso, que indica la existencia de conexiones (0 ó 1). La matriz de grafo de la Figura 17.6 se rehace tal y como se muestra en la Figura 17.7. Se ha reemplazado cada letra por un 1, indicando la existencia de una conexión (se han excluido los ceros por claridad). Cuando se representa de esta forma, la matriz se denomina ***matriz de conexiones.***

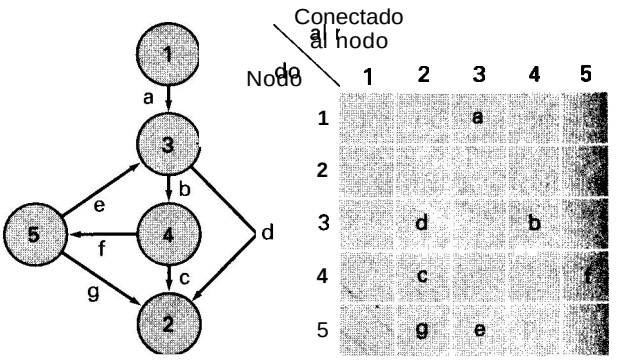


FIGURA 17.6. Matriz del grafo.

En la Figura 17.7, cada fila con dos o más entradas representa un nodo predicado. Por tanto, los cálculos aritméticos que se muestran a la derecha de la matriz de conexiones nos dan otro nuevo método de determinación de la complejidad ciclomática (Sección 17.4.2).

Beizer [BEI90] proporciona un tratamiento profundo de otros algoritmos matemáticos que se pueden aplicar a las matrices de grafos. Mediante estas técnicas, el análisis requerido para el diseño de casos de prueba se puede automatizar parcial o totalmente.

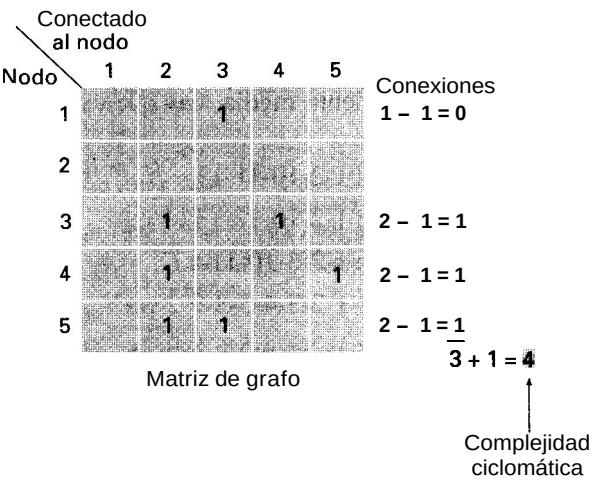


FIGURA 17.7. Matriz de conexiones.

17.5 PRUEBA DE LA ESTRUCTURA DE CONTROL

La técnica de prueba del camino básico descrita en la Sección 17.4 es una de las muchas técnicas para la *prueba de la estructura de control*. Aunque la prueba del camino básico es sencilla y altamente efectiva, no es suficiente por **sí** sola. En esta sección se tratan otras variantes de la prueba de estructura de control. Estas variantes amplían la cobertura de la prueba y mejoran la calidad de la prueba de caja blanca.

17.5.1. Prueba de condición[5]

los errores son mucho más comunes en el entorno de los condiciones lógicas que en los sentencias de proceso secuencial.

La prueba de condición es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa. Una condición simple es una variable lógica o una expresión relacional, posiblemente precedida con un operador NOT («¬»). Una expresión relacional toma la siguiente forma

$$E_1 < \textit{operador-relacional} > E_2$$

donde E_1 y E_2 son expresiones aritméticas y <*operador-relacional*> puede ser alguno de los siguientes: «<», «<=», «=», «≠» («=» desigualdad), «>», o «>=». Una *condición compuesta* está formada por dos o más condiciones simples, operadores lógicos y paréntesis. Suponemos que los operadores lógicos permitidos en una condición compuesta incluyen OR («|»), **AND** («&») y NOT («¬»). A una condición sin expresiones relacionales se la denomina *Expresión lógica*.

Por tanto, los tipos posibles de componentes en una condición pueden ser: un operador lógico, una variable lógica, un par de paréntesis lógicos (que rodean a una condición simple o compuesta), un operador relacional o una expresión aritmética.

Si una condición es incorrecta, entonces es incorrecto al menos un componente de la condición. Así, los tipos de errores de una condición pueden ser los siguientes:

- error en operador lógico (existencia de operadores lógicos incorrectos/desaparecidos/sobrantes)
- error en variable lógica
- error en paréntesis lógico
- error en operador relacional
- error en expresión aritmética.

El método de la *prueba de condiciones* se centra en la prueba de cada una de las condiciones del programa. Las estrategias de prueba de condiciones (tratadas posteriormente en este capítulo) tienen, generalmente, dos ventajas. La primera, que la cobertura de la prueba de una condición es sencilla. La segunda, que la cobertura de la prueba de las condiciones de un programa da una orientación para generar pruebas adicionales del programa.

El propósito de la prueba de condiciones es detectar, no sólo los errores en las condiciones de un programa, sino también otros errores en dicho programa. Si un conjunto de pruebas de un programa P es efectivo para detectar errores en las condiciones que se encuentran en P, es probable que el conjunto de pruebas también sea efectivo para detectar otros errores en el programa P. Además, si una estrategia de prueba es efectiva para detectar errores en una condición, entonces es probable que dicha estrategia también sea efectiva para detectar errores en el programa.

Se han propuesto una serie de estrategias de prueba de condiciones. La *prueba de ramificaciones* es, posiblemente, la estrategia de prueba de condiciones más sencilla. Para una condición compuesta C, es necesario ejecutar al menos una vez las ramas verdadera y falsa de C y cada condición simple de C [MYE79].

Cada vez que decides efectuar una prueba de condición, deberás evaluar cada condición en un intento por descubrir errores. ¡Este es un escondrijo ideal para los errores!

La *prueba del dominio* [WHI80] requiere la realización de tres o cuatro pruebas para una expresión relacional. Para una expresión relacional de la forma

$$E_1 < \textit{operador-relacional} > E_2$$

se requieren tres pruebas, para comprobar que el valor de E_1 es mayor, igual o menor que el valor de E_2, respectivamente [HOW82]. Si el <*operador-relacional*> es incorrecto y E_1 y E_2 son correctos, entonces estas tres pruebas garantizan la detección de un error del operador relacional. Para detectar errores en E_1 y E_2, la prueba que haga el valor de E_1 mayor o menor que el de E_2, debe hacer que la diferencia entre estos dos valores sea lo más pequeña posible.

Para una expresión lógica con n variables, habrá que realizar las 2^n pruebas posibles ($n > 0$). Esta estrategia puede detectar errores de un operador, de una variable y de un paréntesis lógico, pero sólo es práctica cuando el valor de n es pequeño.

[5] Las secciones 17.5.1. y 17.5.2. se han adaptado de [TAI89] con permiso del profesor K.C. Tai.

También se pueden obtener pruebas sensibles a error para expresiones lógicas [FOS84, TAI87]. Para una expresión lógica singular (una expresión lógica en la cual cada variable lógica sólo aparece una vez) con n variables lógicas ($n > 0$), podemos generar fácilmente un conjunto de pruebas con menos de 2" pruebas, de tal forma que ese grupo de pruebas garantice la detección de múltiples errores de operadores lógicos **y** también sea efectivo para detectar otros errores.

Tai [TAI89] sugiere una estrategia de prueba de condiciones que se basa en las técnicas destacadas anteriormente. La técnica, denominada BRO* *(prueba del operador relacional y de ramificación),* garantiza la detección de errores de operadores relacionales **y** de ramificaciones en una condición dada, en la que todas las variables lógicas **y** operadores relacionales de la condición aparecen sólo una vez **y** no tienen variables en común.

La estrategia BRO utiliza restricciones de condición para una condición C. Una restricción de condición para C con n condiciones simples se define como (D_1, D_2,...,D_n), donde D_i ($0 < i \leq n$) es un símbolo que especifica una restricción sobre el resultado de la i-ésima condición simple de la condición C. Una restricción de condición D para una condición C se cubre o se trata en una ejecución de C, si durante esta ejecución de C, el resultado de cada condición simple de C satisface la restricción correspondiente de D.

Para una variable lógica B, especificamos una restricción sobre el resultado de B, que consiste en que B tiene que ser verdadero (v) o falso (f). De forma similar, para una expresión relacional, se utilizan los símbolos >, = **y** < para especificar restricciones sobre el resultado de la expresión.

Como ejemplo, consideremos la condición

$$C_1: \qquad B_1 \,\&\, B_2$$

donde B_1 **y** B_2 son variables lógicas. La restricción de condición para C_1 es de la forma (D_1, D_2), donde D_1 y D_2 son «v» o «f». El valor (v, f) es una restricción de condición para C_1 **y** se cubre mediante la prueba que hace que el valor de B_1 sea verdadero **y** el valor de B_2 sea falso. La estrategia de prueba BRO requiere que el conjunto de restricciones {(v, v), (f, v), (v, f)} sea cubierto mediante las ejecuciones de C_1. Si C_1 es incorrecta por uno o más errores de operador lógico, por lo menos un par del conjunto de restricciones forzará el fallo de C_1.

Como segundo ejemplo, consideremos una condición de la forma

$$C_2: \qquad B_1 \,\&\, (E_3 = E_4)$$

donde B_1 es una expresión lógica y E_3, **y** E_4 son expresiones aritméticas. Una restricción de condición para C_2 es de la forma (D_1, D_2), donde D_1 es «v» o «f» **y** D_2 es >, = o <. Puesto que C_2 es igual que C_1, excepto en que la segunda condición simple de C_2 es una expresión relacional, podemos construir un conjunto de restricciones para C_2 mediante la modificación del conjunto de restricciones {(v, v), (f, v), (v, f)} definido para C_1. Obsérvese que «v» para ($E_3 = E_4$) implica «=» **y** que «f» para ($E_3 = E_4$) implica «< »o «>». Al reemplazar (v, v) **y** (f, v) por (v, =) **y** (f, =), respectivamente **y** reemplazando (v, f) por (v, <) **y** (v,>), el conjunto de restricciones resultante para C, es {(v, =), (f, =), (v, <), (v, >)}. La cobertura del conjunto de restricciones anterior garantizará la detección de errores del operador relacional o lógico en C_2.

Como tercer ejemplo, consideremos una condición de la forma

$$C_3: \qquad (E_1 > E_2) \,\&\, (E_3 \doteq E_4)$$

donde E_1, E_2, E_3 y E_4 son expresiones aritméticas. Una restricción de condición para C, es de la forma (D_1, D_2), donde todos los D_i **y** D_2, son >, = o <. Puesto **que** C_3 es igual que C_1 excepto en que la primera condición simple de C_1 es una expresión relacional, podemos construir un conjunto de restricciones para C_3 mediante la modificación del conjunto de restricciones para C_1, obteniendo

$$\{(>, =), (=, =), (<, =), (>, >), (>. <)\}$$

La cobertura de este conjunto de restricciones **garantizará** la detección de errores de operador relacional en C_1.

17.5.2. Prueba del flujo de datos

El método de *prueba de flujo de datos* selecciona caminos de prueba de un programa de acuerdo con la ubicación de las definiciones **y** los usos de las variables del programa. Se han estudiado **y** comparado varias estrategias de prueba de flujo de datos (por ejemplo, [FRA88], [NTA88], [FRA93]).

Para ilustrar el enfoque de prueba de flujo de datos, supongamos que a cada sentencia de un programa se le asigna un número único de sentencia **y** que las funciones no modifican sus parámetros o las variables globales. Para una sentencia con S como número de sentencia,

DEF(S) = {X | la sentencia **S** contiene una definición de **X** }

USO(S) = {X | la sentencia **S** contiene un uso de X}

Si la sentencia S es una sentencia *if* o de bucle, su conjunto DEF estará vacío y su conjunto USE estará basado en la condición de la sentencia S. La definición de una variable X en una sentencia S se dice que está *viva* en una sentencia S' si existe un camino de la sentencia S a la sentencia S' que no contenga otra definición de X.

Es para realista asumir que la prueba del flujo de datas puede ser usada ampliamente cuando probamos grandes sistemas. En cualquier rasa, puede ser utilizada en areas del software que sean sospechosos.

* En ingles, Branch arid Relational Operator

Una ***cadena de definición-uso (o*** cadena DU) de una variable *X* tiene la forma *[X,S,* S'], donde **S** y **S'** son números de sentencia, *X* está en DEF(*S*) y en USO(*S'*) y la definición de *X* en la sentencia **S** está viva en la sentencia **S'.**

Una sencilla estrategia de prueba de flujo de datos se basa en requerir que se cubra al menos una vez cada cadena DU. Esta estrategia se conoce como ***estrategia*** de ***prueba*** DU. Se ha demostrado que la prueba DU no garantiza que se cubran todas las ramificaciones de un programa. Sin embargo, solamente no se garantiza el cubrimiento de una ramificación en situaciones raras como las construcciones ***if-then-else*** en las que la parte ***then*** no tiene ninguna definición de variable y no existe la parte ***else.*** En esta situación, la prueba DU no cubre necesariamente la rama ***else*** de la sentencia $superior.

Las estrategias de prueba de flujo de datos son Útiles para seleccionar caminos de prueba de un programa que contenga sentencias *if o* ***de bucles*** anidados. Para ilustrar esto, consideremos la aplicación de la prueba DU para seleccionar caminos de prueba para el LDP que sigue:

```
proc X
    B1:
    do while C1
       if C2
          then
            if C4
               then B4;
               else B5;
            endif;
          else
            if C3
               then b2;
               else B3;
            endif;
       endif;
     enddo;
      B6;
  end proc;
```

Para aplicar la estrategia de prueba DU para seleccionar caminos de prueba del diagrama de flujo de control, necesitamos conocer las definiciones y los usos de las variables de cada condición o cada bloque del LDP. Asumimos que la variable X es definida en la Última sentencia de los bloques B1, B2, B3, B4 y B5, y que se usa en la primera sentencia de los bloques B2, B3, B4, B5 y B6. La estrategia de prueba DU requiere una ejecución del camino más corto de cada B_i, $0 < i \leq 5$, a cada B_j, $1 < j \leq 6$. (Tal prueba también cubre cualquier uso de la variable *X* en las condiciones C1, C2, C3 y C4). Aunque hay veinticinco cadenas DU de la variable *X*, sólo necesitamos cinco caminos para cubrir la cadena DU. La razón es que se necesitan cinco caminos para cubrir la cadena DU de X desde B_i, $0 < i \leq 5$, hasta B6, y las otras cadenas DU se pueden cubrir haciendo que esos cinco caminos contengan iteraciones del bucle.

Dado que las sentencias de un programa están relacionadas entre **sí** de acuerdo con las definiciones de las variables, el enfoque de prueba de flujo de datos es efectivo para la protección contra errores. Sin embargo, los problemas de medida de la cobertura de la prueba y de selección de caminos de prueba para la prueba de flujo de datos son más difíciles que los correspondientes problemas para la prueba de condiciones.

17.5.3. Prueba de bucles

Los bucles son la piedra angular de la inmensa mayoría de los algoritmos implementados en software. **Y** sin embargo, les prestamos normalmente poca atención cuando llevamos a cabo las pruebas del software.

Las estructuras de bucles complejos es otro lugar propenso a errores. Por tonto, es muy valioso realizar diseños de pruebas que ejerciten completamente los estructuras bucle.

La prueba de bucles es una técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de bucles. Se pueden definir cuatro clases diferentes de bucles [BEI90]: ***bucles simples, bucles concatenados, bucles anidados*** y ***bucles no estructurados*** (Fig. 17.8).

Bucles simples. A los bucles simples se les debe aplicar el siguiente conjunto de pruebas, donde ***n*** es el número máximo de pasos permitidos por el bucle:

1. pasar por alto totalmente el bucle
2. pasar una sola vez por el bucle
3. pasar dos veces por el bucle
4. hacer ***m*** pasos por el bucle con ***m < n***
5. hacer ***n – 1***, ***n*** y ***n+l*** pasos por el bucle

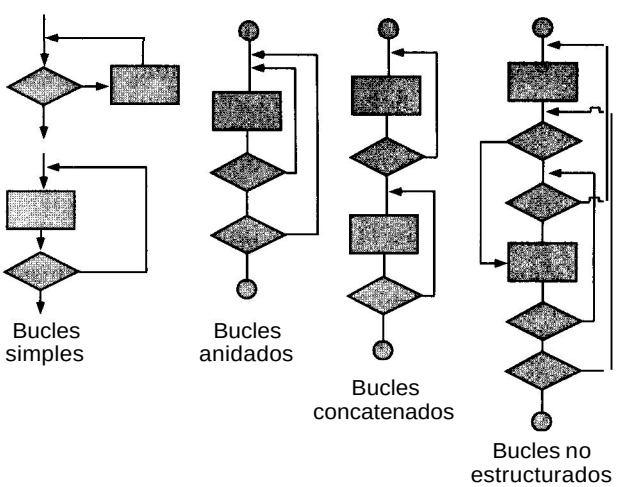


FIGURA 17.8. Clases de **bucles.**

Bucles anidados. Si extendiéramos el enfoque de prueba de los bucles simples a los bucles anidados, el número de posibles pruebas aumentaría geométricamente a medida que aumenta el nivel de anidamiento. Esto llevaría a un número impracticable de pruebas. Beizer [BEI90] sugiere un enfoque que ayuda a reducir el número de pruebas:

1. Comenzar por el bucle más interior. Establecer o configurar los demás bucles con sus valores mínimos.
2. Llevar a cabo las pruebas de bucles simples para el bucle más interior, mientras se mantienen los parámetros de iteración (por ejemplo, contador del bucle) de los bucles externos en sus valores mínimos. Añadir otras pruebas para valores fuera de rango o excluidos.
3. Progresar hacia fuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo todos los bucles externos en sus valores mínimos y los demás bucles anidados en sus valores «típicos».
4. Continuar hasta que se hayan probado todos los bucles.

Bucles concatenados. Los bucles concatenados se pueden probar mediante el enfoque anteriormente definido para los bucles simples, mientras cada uno de los bucles sea independiente del resto. Sin embargo, si hay dos bucles concatenados y se usa el controlador del bucle 1 como valor inicial del bucle *2*, entonces los bucles no son independientes. Cuando los bucles no son independientes, se recomienda usar el enfoque aplicado para los bucles anidados.

No debes probar los bucles no estructurados. Rediséñalos.

Bucles no estructurados. Siempre que sea posible, esta clase de bucles se deben *rediseñar* para que se ajusten a las construcciones de programación estructurada (Capítulo 16).

17.6 PRUEBA DE CAJA NEGRA

Las pruebas de caja negra, también denominada *prueba de comportamiento,* se centran en los requisitos funcionales del software. O sea, la prueba de caja negra permite al ingeniero del software obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa. La prueba de caja negra no es una alternativa a las técnicas de prueba de caja blanca. **Más** bien se trata de un enfoque complementario que intenta descubrir diferentes tipos de errores que los métodos de caja blanca.

La prueba de caja negra intenta encontrar errores de las siguientes categorías: **(1)** funciones incorrectas o ausentes, (2) errores de interfaz, **(3)** errores en estructuras de datos o en accesos a bases de datos externas, **(4)** errores de rendimiento y *(5)* errores de inicialización y de terminación.

A diferencia de la prueba de caja blanca, que se lleva a cabo previamente en el proceso de prueba, la prueba de caja negra tiende a aplicarse durante fases posteriores de la prueba (véase el Capítulo 18). Ya que la prueba de caja negra ignora intencionadamente la estructura de control, centra su atención en el campo de la información. Las pruebas se diseñan para responder a las siguientes preguntas:

- ¿Cómo se prueba la validez funcional?
- ¿Cómo se prueba el rendimiento y el comportamiento del sistema?
- ¿Qué clases de entrada compondrán unos buenos casos de prueba?
- ¿Es el sistema particularmente sensible a ciertos valores de entrada?
- ¿De qué forma están aislados los límites de una clase de datos?
- ¿Qué volúmenes y niveles de datos tolerará el sistema?
- ¿Qué efectos sobre la operación del sistema tendrán combinaciones específicas de datos?

Mediante las técnicas de prueba de caja negra se obtiene un conjunto de casos de prueba que satisfacen los siguientes criterios [MYE79]: (1) casos de prueba que reducen, en un coeficiente que es mayor que uno, el número de casos de prueba adicionales que se deben diseñar para alcanzar una prueba razonable y *(2)* casos de prueba que nos dicen algo sobre la presencia o ausencia de clases de errores en lugar de errores asociados solamente con la prueba que estamos realizando.

17.6.1. Métodos de prueba basados en grafos

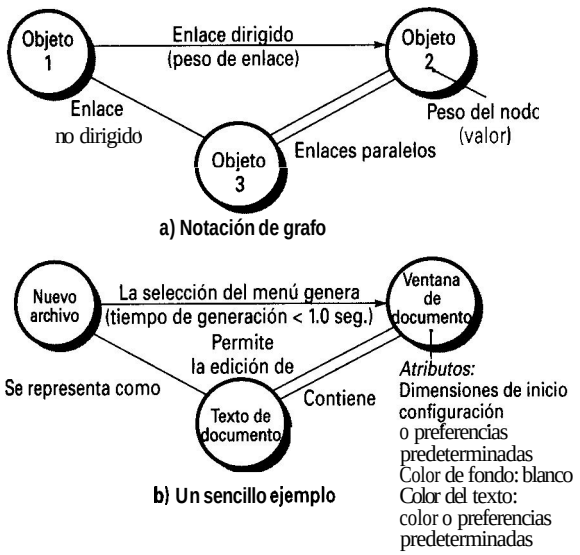
El primer paso en la prueba de caja negra es entender los objetos[6] que se modelan en el software y las relaciones que conectan a estos objetos. Una vez que se ha llevado a cabo esto, el siguiente paso es definir una serie de pruebas que verifiquen que «todos los objetos tienen entre ellos las relaciones esperadas» [BEI95]. Dicho de otra manera, la prueba del software empieza creando un grafo de objetos importantes y sus relaciones, y después

[6] En este contexto, el término «objeto» comprende los objetos de datos que se estudiaron en los Capítulos 11 y 12 así como objetos de programa tales como módulos o colecciones de sentencias del lenguaje de programación.

diseñando una serie de pruebas que cubran el grafo de manera que se ejerciten todos los objetos y sus relaciones para descubrir los errores.

PUNTO CLAVE

Un grafo representa la relación entre objeto dato y objeto programa, permitiéndonos derivar casos de prueba que buscan errores asociados con estas relaciones.



RGURA 17.9.

Para llevar a cabo estos pasos, el ingeniero del software empieza creando un *grafo* —una colección de *nodos* que representan objetos; *enlaces* que representan las relaciones entre los objetos; *pesos de nodos* que describen las propiedades de un nodo (por ejemplo, un valor específico de datos o comportamiento de estado) y *pesos de enlaces* que describen alguna característica de un enlace—[7].

En la Figura 17.9a se muestra una representación simbólica de un grafo. Los nodos se representan como círculos conectados por enlaces que toman diferentes formas. Un *enlace dirigido* (representado por una flecha) indica que una relación se mueve sólo en una dirección. Un *enlace bidireccional,* también denominado *enlace simétrico,* implica que la relación se aplica en ambos sentidos. Los enlaces paralelos se usan cuando se establecen diferentes relaciones entre los nodos del grafo.

Como ejemplo sencillo, consideremos una parte de un grafo de una aplicación de un procesador de texto (Fig. 17.9b) donde:

objeto n.º 1 = **selección en el menú de archivo nuevo**
objeto n.º 2 = **ventana del documento**
objeto n.º 3 = **texto del documento**

Como se muestra en la figura, una selección del menú en **archivo nuevo** genera una **ventana del documento.** El peso del nodo de **ventana del documento** proporciona una lista de los atributos de la ventana que se esperan cuando se genera una ventana. El peso del enlace indica que la ventana se tiene que generar en menos de **1.0** segundos. Un enlace no dirigido establece una relación simétrica entre **selección en el menú de archivo nuevo** y **texto del documento,** y los enlaces paralelos indican las relaciones entre la **ventana del documento** y el **texto del documento.** En realidad, se debería generar un grafo bastante más detallado como precursor al diseño de casos de prueba. El ingeniero del software obtiene entonces casos de prueba atravesando el grafo y cubriendo cada una de las relaciones mostradas. Estos casos de prueba están diseñados para intentar encontrar errores en alguna de las relaciones.

Beizer [BEI95] describe un número de métodos de prueba de comportamiento que pueden hacer uso de los grafos:

Modelado del flujo de transacción. Los nodos representan los pasos de alguna transacción (por ejemplo, los pasos necesarios para una reserva en una línea aérea usando un servicio en línea), y los enlaces representan las conexiones lógicas entre los pasos (por ejemplo, *vuelo.información.entrada* es seguida de *validación/disponibilidad.procesamiento*). El diagrama de flujo de datos (Capítulo 12) puede usarse para ayudar en la creación de grafos 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 telefonista coge una petición por teléfono), y los enlaces representan las transiciones que ocurren para moverse de estado a estado (por ejemplo, *petición-información se verifica durante inventario-disponibilidad-búsqueda* y es seguido por *cliente-factura-información-entrada).* El diagrama estado-transición (Capítulo 12) puede usarse para ayudar en la creación de grafos de este tipo.

Modelado del flujo de datos. Los nodos son objetos de datos y los enlaces son las transformaciones que ocurren para convertir un objeto de datos en otro. Por ejemplo, el nodo **FICA.impuesto.retenido** (FIR) se calcula de **brutos.sueldos** (BS) usando la relación FIR = 0,62 **X** BS.

Modelado de planificación. Los nodos son objetos de programa y los enlaces son las conexiones secuenciales entre esos objetos. Los pesos de enlace se usan para especificar los tiempos de ejecución requeridos al ejecutarse el programa.

[7] Si los conceptos anteriores suenan vagamente familiares, recordemos que los grafos se usaron también en la Sección 17.4.1 para crear un grafo del programa para el método de la prueba del camino básico. Los nodos del grafo del programa contenían instrucciones (objetos de programa) caracterizados como representaciones de diseño procedimental o como código fuente y los enlaces dirigidos indicaban el flujo de control entre estos objetos del programa. **Aquí** se extiende el uso de los grafos para incluir la prueba de caja negra.

Un estudio detallado de cada uno de estos métodos de prueba basados en grafos está más allá del alcance de este libro. El lector interesado debería consultar [BEI95] para ver un estudio detallado. Merece la pena, sin embargo, proporcionar un resumen genérico del enfoque de pruebas basadas en grafos.

¿Cuáles son las actividades generales requeridas durante la prueba basada en un grafo?

Las pruebas basadas en grafos empiezan con la definición de todos los nodos y pesos de nodos. O sea, se identifican los objetos y los atributos. El modelo de datos (Capítulo **12)** puede usarse como punto de partida, pero es importante tener en cuenta que muchos nodos pueden ser objetos de programa (no representados explícitamente en el modelo de datos). Para proporcionar una indicación de los puntos de inicio y final del grafo, es útil definir unos nodos de entrada y salida.

Una vez que se han identificado los nodos, se deberían establecer los enlaces y los pesos de enlaces. En general, conviene nombrar los enlaces, aunque los enlaces que representan el flujo de control entre los objetos de programa no es necesario nombrarlos.

En muchos casos, el modelo de grafo puede tener bucles (por ejemplo, un camino a través del grafo en el que se encuentran uno o más nodos más de una vez). La prueba de bucle (Sección 17.5.3) se puede aplicar también a nivel de comportamiento (de caja negra). El grafo ayudará a identificar aquellos bucles que hay que probar.

Cada relación es estudiada separadamente, de manera que se puedan obtener casos de prueba. La *transitividad* de relaciones secuenciales es estudiada para determinar cómo se propaga el impacto de las relaciones a través de objetos definidos en el grafo. La transitividad puede ilustrarse considerando tres objetos **X, Y** y Z. Consideremos las siguientes relaciones:

X *es necesaria para calcular* **Y**
Y *es necesaria para calcular* **Z**

Por tanto, se ha establecido una relación transitiva entre **X** y **Z:**

X *es necesaria para calcular* **Z**

Basándose en esta relación transitiva, las pruebas para encontrar errores en el cálculo de **Z** deben considerar una variedad de valores para **X** e **Y.**

La *simetría* de una relación (enlace de grafo) es también una importante directriz para diseñar casos de prueba. Si un enlace es bidireccional (simétrico), es importante probar esta característica. La característica *UNDO* [BEI95] (deshacer) en muchas aplicaciones para ordenadores personales implementa una limitada simetría. Es decir, *UNDO* permite deshacer una acción después de haberse completado. Esto debería probarse minuciosamente y todas las excepciones (por ejemplo, lugares donde no se puede usar *UNDO)* deberían apuntarse. Finalmente, todos los nodos del grafo deberían tener una relación que los lleve de vuelta a ellos mismos; en esencia, un bucle de «no acción» o «acción nula». Estas relaciones *reflexivas* deberían probarse también.

Cuando empieza el diseño de casos de prueba, el primer objetivo es conseguir la *cobertura de nodo.* Esto significa que las pruebas deberían diseñarse para demostrar que ningún nodo se ha omitido inadvertidamente y que los pesos de nodos (atributos de objetos) son correctos.

A continuación, se trata la *cobertura de enlaces.* Todas las relaciones se prueban basándose en sus propiedades. Por ejemplo, una relación simétrica se prueba para demostrar que es, de hecho, bidireccional. Una relación transitiva se prueba para demostrar que existe transitividad. Una relación reflexiva se prueba para asegurarse de que hay un bucle nulo presente. Cuando se han especificado los pesos de enlace, se diseñan las pruebas para demostrar que estos pesos son válidos. Finalmente, se invocan las pruebas de bucles (Sección 17.5.3).

17.6.2. Partición equivalente

La *partición equivalente* es un método de prueba de caja negra que divide el campo de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba. Un caso de prueba ideal descubre de forma inmediata una clase de errores (por ejemplo, proceso incorrecto de todos los datos de carácter) que, de otro modo, requerirían la ejecución de muchos casos antes de detectar el error genérico. La partición equivalente se dirige a la definición de casos de prueba que descubran clases de errores, reduciendo así el número total de casos de prueba que hay que desarrollar.

las clases de entrada san conocidas relativamente temprano en el procesa de software. Por esto razón, comenzamos pensando en la partición equivalente una vez el diseña ha sido creada.

El diseño de casos de prueba para la partición equivalente se basa en una evaluación de las clases de equivalencia para una condición de entrada. Mediante conceptos introducidos en la sección anterior, si un conjunto de objetos puede unirse por medio de relaciones simétricas, transitivas y reflexivas, entonces existe una clase de equivalencia [BEI95]. Una *clase de equivalencia* representa un conjunto de estados válidos o no válidos para condiciones de entrada. Típicamente, una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición lógica. Las clases de equivalencia se pueden definir de acuerdo con las siguientes directrices:

1. Si una condición de entrada especifica un *rango,* se define una clase de equivalencia válida y dos no válidas.

2. Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos no válidas.
3. Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y una no válida.
4. Si una condición de entrada es lógica, se define una clase de equivalencia válida y una no válida.

Como ejemplo, consideremos los datos contenidos en una aplicación de automatización bancaria. El usuario puede «llamar» al banco usando su ordenador personal, dar su contraseña de seis dígitos y continuar con una serie de órdenes tecleadas que desencadenarán varias funciones bancarias. El software proporcionado por la aplicación bancaria acepta datos de la siguiente forma:

Código de área: en blanco o un número de tres dígitos
Prefijo: número de tres dígitos que no comience por 0 o 1
Sufijo: número de cuatro dígitos
Contraseña: valor alfanumérico de seis dígitos
Órdenes: «comprobar», «depositar», «pagar factura», etc.

Las condiciones de entrada asociadas con cada elemento de la aplicación bancaria se pueden especificar como:

Código de área:	condición de entrada, *lógica*—el código de área puede estar o no presente condición de entrada, *rango*—valores definidos entre 200 y 999, con excepciones específicas
Prefijo:	condición de entrada, rango —valor especificado > 200 sin dígitos 0
Sufijo:	condición de entrada, valor —longitud de cuatro dígitos
Contraseña:	condición de entrada, lógica —la palabra clave puede estar o no presente; condición de entrada, valor —cadena de seis caracteres
Orden:	condición de entrada, *conjunto*—contenida en las órdenes listadas anteriormente

Aplicando las directrices para la obtención de clases de equivalencia, se pueden desarrollar y ejecutar casos de prueba para cada elemento de datos del campo de entrada. Los casos de prueba se seleccionan de forma que se ejercite el mayor número de atributos de cada clase de equivalencia a la vez.

17.6.3. Análisis de valores límite

Por razones que no están del todo claras, los errores tienden a darse más en los límites del campo de entrada que en el «centro». Por ello, se ha desarrollado el *análisis* de valores límites (AVL) como técnica de prueba. El análisis de valores límite nos lleva a una elección de casos de prueba que ejerciten **los** valores límite.

AVL amplía la partición equivalente para fijarse sobre datos en el «límite» de una clase de equivalencia.

El análisis de valores límite es una técnica de diseño de casos de prueba que complementa a la partición equivalente. En lugar de seleccionar cualquier elemento de una clase de equivalencia, el **AVL** lleva a la elección de casos de prueba en los «extremos» de la clase. En lugar de centrarse solamente en las condiciones de entrada, el AVL obtiene casos de prueba también para el campo de salida [MYE79].

¿Cómo se crean casos de prueba para el AVL?

Las directrices de AVL son similares en muchos aspectos a las que proporciona la partición equivalente:

1. Si una condición de entrada especifica un rango delimitado por los valores a y ***b,*** se deben diseñar casos de prueba para los valores a y *b,* y para los valores justo por debajo y justo por encima de a y *b,* respectivamente.
2. Si una condición de entrada especifica un número de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo. También se deben probar los valores justo por encima y justo por debajo del máximo y del mínimo.
3. Aplicar las directrices 1 y 2 a las condiciones de salida. Por ejemplo, supongamos que se requiere una tabla de «temperatura / presión» como salida de un programa de análisis de ingeniería. Se deben diseñar casos de prueba que creen un informe de salida que produzca el máximo **(y** el mínimo) número permitido de entradas en la tabla.
4. Si las estructuras de datos internas tienen límites preestablecidos (por ejemplo, una matriz que tenga un límite definido de 100 entradas), hay que asegurarse de diseñar un caso de prueba que ejercite la estructura de datos en sus límites.

La mayoría de los ingenieros del software llevan a cabo de forma intuitiva alguna forma de AVL. Aplicando las directrices que se acaban de exponer, la prueba de límites será más completa y, por tanto, tendrá una mayor probabilidad de detectar errores.

17.6.4. Prueba de comparación

Hay situaciones (por ejemplo, aviónica de aeronaves, control de planta nuclear) en las que la fiabilidad del software es algo absolutamente crítico. En ese tipo de

aplicaciones, a menudo se utiliza hardware y software redundante para minimizar la posibilidad de error. Cuando se desarrolla software redundante, varios equipos de ingeniería del software separados desarrollan versiones independientes de una aplicación, usando las mismas especificaciones. En esas situaciones, se deben probar todas las versiones con los mismos datos de prueba, para asegurar que todas proporcionan una salida idéntica. Luego, se ejecutan todas las versiones en paralelo y se hace una comparación en tiempo real de los resultados, para garantizar la consistencia.

Con las lecciones aprendidas de los sistemas redundantes, los investigadores (por ejemplo, [BRI87]) han sugerido que, para las aplicaciones críticas, se deben desarrollar versiones de software independientes, incluso aunque sólo se vaya a distribuir una de las versiones en el sistema final basado en computadora. Esas versiones independientes son la base de una técnica de prueba de caja negra denominada *prueba de comparación o prueba mano a mano* [KNI89].

Cuando se han producido múltiples implementaciones de la misma especificación, a cada versión del software se le proporciona como entrada los casos de prueba diseñados mediante alguna otra técnica de caja negra (por ejemplo, la partición equivalente). Si las salidas producidas por las distintas versiones son idénticas, se asume que todas las implementaciones son correctas. Si la salida es diferente, se investigan todas las aplicaciones para determinar el defecto responsable de la diferencia en una o más versiones. En la mayoría de los casos, la comparación de las salidas se puede llevar a cabo mediante una herramienta automática.

17.6.5. Prueba de la tabla ortogonal

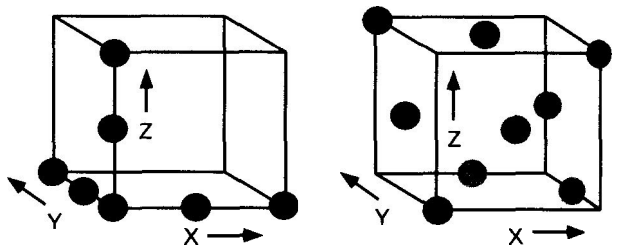
Hay muchas aplicaciones en que el dominio de entrada es relativamente limitado. Es decir, el número de parámetros de entrada es pequeño y los valores de cada uno de los parámetros está claramente delimitado. Cuando estos números son muy pequeños (por ejemplo, 3 parámetros de entrada tomando 3 valores diferentes), es posible considerar cada permutación de entrada y comprobar exhaustivamente el proceso del dominio de entrada. En cualquier caso, cuando el número de valores de entrada crece y el número de valores diferentes para cada elemento de dato se incrementa, la prueba exhaustiva se hace impracticable o imposible.

PUNTO CLAVE

La prueba de la tabla ortogonal permite diseñar casos de prueba que facilitan una cobertura máxima de prueba con un número razonable de casos de prueba.

La prueba de la tabla ortogonal puede aplicarse a problemas en que el dominio de entrada es relativamente pequeño pero demasiado grande para posibilitar pruebas exhaustivas. El método de prueba de la tabla ortogonal es particularmente útil al encontrar errores asociados con fallos localizados —una categoría de error asociada con defectos de la lógica dentro de un componente software—.

Para ilustrar la diferencia entre la prueba de la tabla ortogonal y una aproximación más convencional «un elemento de entrada distinto cada vez», considerar un sistema que tiene tres elementos de entrada, **X,** Y y Z. Cada uno de estos elementos de entrada tiene tres valores diferentes. Hay $3^3 = 27$ posibles casos de prueba. Phadke [PHA97] sugiere una visión geométrica de los posibles casos de prueba asociados con X, Y y Z, según se ilustra en la Figura 17.10. Observando la figura, cada elemento de entrada en un momento dado puede modificarse secuencialmente en cada eje de entrada.



Un elemento de entrada a la vez — Tabla ortogonal L9

FIGURA 17.10. Una vista geométrica de los casos.

Esto da como resultado un alcance relativamente limitado al dominio de entrada (representado en la figura por el cubo de la izquierda).

Cuando se realiza la prueba de la tabla ortogonal, se crea una *tabla ortogonal **L9*** de casos de prueba. La tabla ortogonal L9 tiene una «propiedad de equilibrio)) [PHA97]. Es decir, los casos de prueba (representados por puntos negros en la figura) están «uniformemente dispersos en el dominio de prueba», según se ilustra en el cubo de la derecha de la Figura 17.10. El alcance de la prueba por todo el dominio de entrada es más completo.

Para ilustrar el uso de la tabla ortogonal L9, considerar la función *enviar* para una aplicación de un fax. Cuatro parámetros, P1, P2, P3 y P4 se pasan a la función *enviar*. Cada uno tiene tres valores diferentes. Por ejemplo, P1 toma los valores:

P1 = 1, enviar ahora
P2 = 2, enviar dentro de una hora
P3 = 3, enviar después de media noche

P2, P3, y P4 podrán tomar también los valores 1, 2 y 3, representando otras funciones *enviar*.

Si se elige la estrategia de prueba «un elemento de entrada distinto cada vez», se especifica la siguiente secuencia de pruebas (P1,P2,P3,P4): (1,1,1,1), (2,1,1,1),

(3,1,1,1), (1,2,1,1), (1,3,1,1), (1,1,2,1), (1,1,3,1), (1,1,1,2), y (1,1,1,3). Phadke [PHA97] valoraestos casos de prueba de la siguiente manera:

> Cada uno de estos casos de prueba son Útiles, Únicamente cuando estos parámetros de prueba no se influyen mutuamente. Pueden detectar fallos lógicos cuando el valor de un parámetro produce un mal funcionamiento del software. Estos fallos pueden llamarse fallos de modalidad simple. El método no puede detectar fallos lógicos que causen un mal funcionamiento cuando dos o más parámetros simultáneamente toman determinados valores; es decir, no se pueden detectar interacciones. **Así,** esta habilidad para detectar fallos es limitada.

Dados un número relativamente pequeño de parámetros de entrada y valores diferentes, es posible realizar una prueba exhaustiva. El número de pruebas requeridas es 3–81 —grande pero manejable—. Todos los fallos asociados con la permutación de los datos serán encontrados, pero el esfuerzo requerido es relativamente alto.

Pruebas	Herramienta de prueba			
	P_1	P_2	P_3	P_4
	1	1	1	1
	1	2	2	2
	1	3	3	3
	2	1	2	3
	2	2	3	1
	2	3	1	2
	3	1	3	2
	3	2	1	3
	3	3	2	1

FIGURA 17.11. Una tabla ortogonal L9.

La prueba de la tabla ortogonal nos permite proporcionar una buena cobertura de prueba con bastantes menos casos de prueba que en la estrategia exhaustiva. Una tabla ortogonal L9 para la función de envío del fax se describe en la Figura 17.11.

Phadke [PHA97] valora el resultado de las pruebas utilizando la tabla ortogonal de la siguiente manera:

> **Detecta y aísla todos los fallos de modalidad simple.** Un fallo de modalidad simple es un problema que afecta a un solo parámetro. Por ejemplo, si todos los casos de prueba del factor P 1 = 1 causan una condición de error, nos encontramos con el fallo de modalidad simple. En los ejemplos de prueba 1, 2 y *3* [Fig. 17.111 se encontrarán errores. Analizando la información en que las pruebas muestran errores, se puede identificar que valores del parámetro causan el error. En este ejemplo, anotamos que las pruebas 1, 2 y *3* causan un error, lo que permite aislar [el proceso lógico asociado con «enviar ahora» (P1 = 1)] la fuente del error. **El** aislamiento del fallo es importante para solucionar el error.
>
> **Detecta todos los fallos de modalidad doble. Si** existe un problema donde están afectados dos parámetros que intervienen conjuntamente, se llama fallo de modalidad doble. En efecto, un fallo de modalidad doble es una indicación de incompatibilidad o de imposibilidad de interacción entre dos parámetros.
>
> **Fallos multimodales.** Las tablas ortogonales [del tipo indicado] pueden asegurar la detección Únicamente de fallos de modalidad simple o doble. Sin embargo, muchos fallos en modalidad múltiple pueden ser detectados a través de estas pruebas.

Se puede encontrar un estudio detallado sobre la prueba de tabla ortogonal en [PHA89].

17.7 PRUEBA DE ENTORNOS ESPECIALIZADOS, ARQUITECTURAS Y APLICACIONES

A medida que el software de computadora se ha hecho más complejo, ha crecido también la necesidad de enfoques de pruebas especializados. Los métodos de prueba de caja blanca y de caja negra tratados en las Secciones 17.5 y 17.6 son aplicables a todos los entornos, arquitecturas y aplicaciones pero a veces se necesitan unas directrices y enfoques Únicos para las pruebas. En esta sección se estudian directrices de prueba para entornos, arquitecturas y aplicaciones especializadas que pueden encontrarse los ingenieros del software.

17.7.1. Prueba de interfaces gráficas de usuario (IGUs)

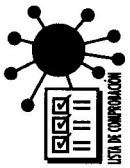
Las interfaces gráficas de usuario (IGUs) presentan interesantes desafíos para los ingenieros del software. Debido a los componentes reutilizables provistos como parte de los entornos de desarrollo de las GUI, la creación de la interfaz de usuario se ha convertido en menos costosa en tiempo y más exacta. Al mismo tiempo, la complejidad de las IGUs ha aumentado, originando más dificultad en el diseño y ejecución de los casos de prueba.

Referencia cruzada
Una guía para el diseño de IGU se presenta en el Capítulo 15.

Dado que las IGUs modernas tienen la misma apariencia y filosofía, se pueden obtener una serie de pruebas estándar. Los grafos de modelado de estado finito (Sección 16.6.1) pueden ser utilizados para realizar pruebas que vayan dirigidas sobre datos específicos y programas objeto que sean relevantes para las IGUs.

Considerando el gran número de permutaciones asociadas con las operaciones IGU, sería necesario para

probar el utilizar herramientas automáticas. Una amplía lista de herramientas de prueba de IGU han aparecido en el mercado en los Últimos años. Para profundizar en el tema, ver el Capítulo 31.



Prueba IGU.

17.7.2. Prueba de arquitectura cliente/servidor

La arquitectura cliente/servidor (C/S) representa un desafío significativo para los responsables de la pruebas del software. La naturaleza distribuida de los entornos cliente/servidor, los aspectos de rendimiento asociados con el proceso de transacciones, la presencia potencial de diferentes plataformas hardware, las complejidades de las comunicaciones de red, la necesidad de servir a múltiples clientes desde una base de datos centralizada (o en algunos casos, distribuida) y los requisitos de coordinación impuestos al servidor se combinan todos para hacer las pruebas de la arquitectura C/S y el software residente en ellas, considerablemente más difíciles que la prueba de aplicaciones individuales. De hecho, estudios recientes sobre la industria indican un significativo aumento en el tiempo invertido y los costes de las pruebas cuando se desarrollan entornos C/S.

Referencia cruzada

Lo ingeniería del software cliente/servidor se presenta en el Capítulo 28.

17.7.3. Prueba de la documentación y facilidades de ayuda

El término apruebas del software» hace imaginarnos gran cantidad de casos de prueba preparados para ejecutar programas de computadora y los datos que manejan. Recordando la definición de software presentada en el primer capítulo de este libro, es importante darse cuenta de que la prueba debe extenderse al tercer elemento de la configuración del software —la documentación —.

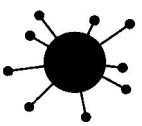
Los errores en la documentación pueden ser tan destructivos para la aceptación del programa, como los errores en los datos o en el código fuente. Nada es más frustrante que seguir fielmente el manual de usuario y obtener resultados o comportamientos que no coinciden con los anticipados por el documento. Por esta razón, la prueba de la documentación debería ser una parte importante de cualquier plan de pruebas del software.

La prueba de la documentación se puede enfocar en dos fases. La primera fase, *la revisión e inspección* (Capítulo 8), examina el documento para comprobar la claridad editorial. La segunda fase, *la prueba en vivo,* utiliza la documentación junto al uso del programa real.

Es sorprendente, pero la prueba en vivo de la documentación se puede enfocar usando técnicas análogas a muchos de los métodos de prueba de caja negra estudiados en la Sección 17.6. Se pueden emplear pruebas basadas en grafos para describir el empleo del programa; se pueden emplear el análisis de la partición equivalente o de los valores límites para definir varias clases de entradas e interacciones asociadas.

17.7.4. Prueba de sistemas de tiempo-real

La naturaleza asíncrona y dependiente del tiempo de muchas aplicaciones de tiempo real, añade un nuevo y potencialmente difícil elemento a la complejidad de las pruebas - e l tiempo —. El responsable del diseño de casos de prueba no sólo tiene que considerar los casos de prueba de caja blanca y de caja negra, sino también el tratamiento de sucesos (por ejemplo, procesamiento de interrupciones), la temporización de los datos y el paralelismo de las tareas (procesos) que manejan los datos. En muchas situaciones, los datos de prueba proporcionados al sistema de tiempo real cuando se encuentra en un determinado estado darán un proceso correcto, mientras que al proporcionárselos en otro estado llevarán a un error.


Sistemas de tiempo real.

Por ejemplo, un software de tiempo real que controla una moderna fotocopiadora puede aceptar interrupciones del operador (por ejemplo, el operador de la máquina pulsa teclas de control tales como «inicialización» u «oscurecimiento») sin error cuando la máquina se encuentra en el estado de hacer fotocopias (estado de «copia»). Esas mismas interrupciones del operador, cuando se producen estando la máquina en estado de «atasco», pueden producir un código de diagnóstico que indique la situación del atasco (un error).

Además, la estrecha relación que existe entre el software de tiempo real y su entorno de hardware también puede introducir problemas en la prueba. Las pruebas del software deben considerar el impacto de los fallos del hardware sobre el proceso del software. Puede ser muy difícil simular de forma realista esos fallos.

Todavía han de evolucionar mucho los métodos generales de diseño de casos de prueba para sistemas de tiempo real. Sin embargo, se puede proponer una estrategia en cuatro pasos:

Prueba de tareas. El primer paso de la prueba de sistemas de tiempo real consiste en probar cada tarea independientemente. Es decir, se diseñan pruebas de caja blanca y de caja negra y se ejecutan para cada

tarea. Durante estas pruebas, cada tarea se ejecuta independientemente. La prueba de la tarea ha de descubrir errores en la lógica y en el funcionamiento, pero no los errores de temporización o de comportamiento.

Referencia Web

El Forum de Discusión de la Prueba del Software presenta temas de interés a los profesionales que efectúan la prueba: **www.ondaweb.com /HyperNews/get.cgi/forums/sti.html**.

Prueba de comportamiento. Utilizando modelos del sistema creados con herramientas CASE, es posible simular el comportamiento del sistema en tiempo real y examinar su comportamiento como consecuencia de sucesos externos. Estas actividades de análisis pueden servir como base del diseño de casos de prueba que se llevan a cabo cuando se ha construido el software de tiempo real. Utilizando una técnica parecida a la partición equivalente (Sección 17.6.1), se pueden categorizar los sucesos (por ejemplo, interrupciones, señales de control, datos) para la prueba. Por ejemplo, los sucesos para la fotocopiadora pueden ser interrupciones del usuario (por ejemplo, reinicialización del contador), interrupciones mecánicas (por ejemplo, atasco del papel), interrupciones del sistema (por ejemplo, bajo nivel de tinta) y modos de fallo (por ejemplo, rodillo excesivamente caliente). Se prueba cada uno de esos sucesos individualmente y se examina el comportamiento del sistema ejecutable para detectar errores que se produzcan como consecuencia del proceso asociado a esos sucesos. Se puede comparar el comportamiento del modelo del sistema (desarrollado durante el análisis) con el software ejecutable para ver si existe concordancia. Una vez que se ha probado cada clase de sucesos, al sistema se le presentan sucesos en un orden aleatorio y con una frecuencia aleatoria. Se examina el comportamiento del software para detectar errores de comportamiento.

Prueba intertareas. Una vez que se han aislado los errores en las tareas individuales y en el comportamiento del sistema, la prueba se dirige hacia los errores relativos al tiempo. Se prueban las tareas asíncronas que se sabe que comunican con otras, con diferentes tasas de datos y cargas de proceso para determinar si se producen errores de sincronismo entre las tareas. Además, se prueban las tareas que se comunican mediante colas de mensajes o almacenes de datos, para detectar errores en el tamaño de esas áreas de almacenamiento de datos.

Prueba del sistema. El software y el hardware están integrados, por lo que se lleva a cabo una serie de pruebas completas del sistema (Capítulo 18) para intentar descubrir errores en la interfaz software/hardware.

La mayoría de los sistemas de tiempo real procesan interrupciones. Por tanto, es esencial la prueba del manejo de estos sucesos lógicos. Usando el diagrama estado-transición y la especificación de control (Capítulo 12), el responsable de la prueba desarrolla una lista de todas las posibles interrupciones y del proceso que ocurre como consecuencia de la interrupción. Se diseñan después pruebas para valorar las siguientes características del sistema:

- ¿Se han asignado y gestionado apropiadamente las prioridades de interrupción?
- ¿Se gestiona correctamente el procesamiento para todas las interrupciones?
- ¿Se ajusta a los requisitos el rendimiento (por ejemplo, tiempo de proceso) de todos los procedimientos de gestión de interrupciones?
- ¿Crea problemas de funcionamiento o rendimiento la llegada de un gran volumen de interrupciones en momentos críticos?

Además, se deberían probar las áreas de datos globales que se usan para transferir información como parte del proceso de una interrupción para valorar el potencial de generación de efectos colaterales.

RESUMEN

El principal objetivo del diseño de casos de prueba es obtener un conjunto de pruebas que tengan la mayor probabilidad de descubrir los defectos del software. Para llevar a cabo este objetivo, se usan dos categorías diferentes de técnicas de diseño de casos de prueba: prueba de caja blanca y prueba de caja negra.

Las pruebas de caja blanca se centran en la estructura de control del programa. Se obtienen casos de prueba que aseguren que durante la prueba se han ejecutado, por lo menos una vez, todas las sentencias del programa y que se ejercitan todas las condiciones lógicas. La prueba del camino básico, una técnica de caja blanca, hace uso de grafos de programa (o matrices de grafos) para obtener el conjunto de pruebas linealmente independientes que aseguren la total cobertura. La prueba de condiciones y del flujo de datos ejercita más aún la lógica del programa y la prueba de los bucles complementa a otras técnicas de caja blanca, proporcionando un procedimiento para ejercitar bucles de distintos grados de complejidad.

Heztel [HET84] describe la prueba de caja blanca como «prueba a pequeña escala». Esto se debe a que las pruebas de caja blanca que hemos considerado en este capítulo son típicamente aplicadas a pequeños componentes de programas (po ejemplo; módulos o pequeños grupos de módulos). Por otro lado, la prueba de caja negra amplía el enfoque y se puede denominar «prueba a gran escala».

Las pruebas de caja negra son diseñadas para validar los requisitos funcionales sin fijarse en el funcionamiento interno de un programa. Las técnicas de prueba de caja negra se centran en el ámbito de información de un programa, de forma que se proporcione una cobertura completa de prueba. Los métodos de prueba basados en grafos exploran las relaciones entre los objetos del programa y su comportamiento. La partición equivalente divide el campo de entrada en clases de datos que tienden a ejercitar determinadas funciones del software. El análisis de valores límite prueba la habilidad del programa para manejar datos que se encuentran en los límites aceptables. La prueba de la tabla ortogonal suministra un método sistemático y eficiente para probar sistemas con un número reducido de parámetros de entrada.

Los métodos de prueba especializados comprenden una amplia gama de capacidades del software y áreas de aplicación. Las interfaces gráficas de usuario, las arquitecturas cliente/servidor, la documentación y facilidades de ayuda, y los sistemas de tiempo real requieren directrices y técnicas especializadas **para** la prueba del software.

A menudo, los desarrolladores de software experimentados dicen que «la prueba nunca termina, simplemente se transfiere de usted (el ingeniero del software) al cliente: Cada vez que el cliente usa el programa, lleva a cabo una prueba.» Aplicando el diseño de casos de prueba, el ingeniero del software puede conseguir **una** prueba más completa y descubrir y corregir así el mayor número de errores antes de que comiencen las «pruebas del cliente».

REFERENCIAS

[BEI90] Beizer, B., *Software Testing Techniques*, 2.ª ed., Van Nostrand Reinhold, 1990.

[BEI95] Beizer, B., *Black-Box Testing*, Wiley, 1995.

[BRI87] Brilliant, S.S., J.C. Knight, y N.G. Levenson, «The Consistent Comparison Problem in N-Version Software», *ACM Software Engineering Notes*, vol. 12, n.° 1, enero 1987, pp. 29-34.

[DAV95] Davis, A., 201 *Principles of Software Development*, McGraw-Hill, 1995.

[DEU79] Deutsch, M., «Verification and Validation», *Software Engineering*, R. Jensen y C.Tonies (eds.), Prentice-Hall, 1979, pp. 329-408.

[FOS84] Foster, K.A., «Sensitive Test Data for Boolean Expressions», *ACM Software Engineering Notes*, vol. 9, n.° 2, Abril 1984, pp. 120-125.

[FRA88] Frankl, P.G., y E.J. Weyuker, «An Applicable Family of Data Flow Testing Criteria», *IEEE Trans. Software Engineering*, vol. 14, n.° 10, Octubre 1988, pp.1483-1498.

[FRA93] Frankl, P.G., y S. Weiss, «An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow», *IEEE Trans. Software Engineering*, vol. 19, n.° 8, Agosto 1993, pp. 770-787.

[HET84] Hetzel, W., *The Complete Guide to Software Testing*, QED Information Sciences, Inc., Wellesley, MA, 1984.

[HOW82] Howden, W.E., «Weak Mutation Testing and the Completeness of Test Cases», *IEEE Trans. Software Engineering*, vol. SE-8, n.° 4, julio 1982, pp. 371-379.

[JON81] Jones, T.C., *Programming Productivity: Issues for the 80's*, IEEE Computer Society Press, 1981.

[KAN93] Kaner, C., J. Falk y H.Q. Nguyen, *Testing Computer Software*, 2.ª ed., Van Nostrand Reinhold, 1993.

[KNI89] Knight, J., y P. Ammann, «Testing Software Using Multiple Versions», Software Productivity Consortium, Report n.° 89029N, Reston, VA, Junio 1989.

[MCC76] McCabe, T., «A Software Complexity Measure», *IEEE Trans. Software Engineering*, vol. 2, Diciembre 1976, pp. 308-320.

[MYE79] Myers, G., *The Art of Software Testing*, Wiley, 1979.

[NTA88] Ntafos, S.C., «A comparison of Some Structural Testing Strategies», *IEEE Trans. Software Engineering*, vol. 14, n.° 6, Junio 1988, pp. 868-874.

[PHA89] Phadke, M.S., *Quality Engineering Using Robust Design*, Prentice Hall, 1989.

[PHA97] Phadke, M.S., «Planning Efficient Software Tests», *Crosstalk*, vol. 10, n.° 10, Octubre 1997, pp. 278-283.

[TAI87] Tai, K.C., y H.K. Su, «Test Generation for Boolean Expresions», *Proc. COMPSAC'87*, Octubre 1987, pp. 278-283.

[TAI89] Tai, K.C., «What to Do Beyond Branch Testing», *ACM Software Engineering Notes*, vol. 14, n.° **2, Abril** 1989, pp. 58-61.

[WHI80] White, L.J., y E.I. Cohen, «A Domain Strategie **for** Program Testing», *IEEE Trans. Software Engineering*, vol. SE-6, n.° 5, Mayo 1980, pp. 247-257.

PROBLEMAS Y PUNTOS A CONSIDERAR

17.1. Myers [MYE79] usa el siguiente programa como autocomprobación de **su** capacidad para especificar una prueba adecuada: un programa lee tres valores enteros. Los tres valores se interpretan como representación de la longitud de los tres lados de un triángulo. El programa imprime un mensaje indicando si el triángulo es escaleno, isósceles o equilátero. Desarrolle un conjunto de casos de prueba que considere que probará de forma adecuada este programa.

17.2. Diseñe e implemente el programa especificado en el Problema 17.1 (con tratamiento de errores cuando sea necesario). Obtenga un grafo de flujo para el programa y aplique la prueba del camino básico para desarrollar casos de prueba que garanticen la comprobación de todas las sentencias del programa. Ejecute los casos y muestre sus resultados.

17.3. ¿Se le ocurren algunos objetivos de prueba adicionales que no se hayan mencionado en la Sección 17.1.1?

17.4. Aplique la técnica de prueba del camino básico a cualquiera de los programas que haya implementado en los Problemas 17.4 a 17.11.

17.5. Especifique, diseñe e implemente una herramienta de software que calcule la complejidad ciclomática para el lenguaje de programación que quiera. Utilice la matriz de grafos como estructura de datos operativa en el diseño.

17.6. Lea a Beizer [BEI90] y determine cómo puede ampliar el programa desarrollado en el Problema 17.5 para que incluya varios pesos de enlace. Amplíe la herramienta para que procese las probabilidades de ejecución o los tiempos de proceso de enlaces.

17.7. Use el enfoque de prueba de condiciones descrito en la Sección 17.5.1 para diseñar un conjunto de casos de prueba para el programa creado en el Problema 17.2.

17.8. Mediante el enfoque de prueba de flujo de datos descrito en la Sección 17.5.2, cree una lista de cadenas de definición-uso para el programa creado en el Problema 17.2.

17.9. Diseñe una herramienta automática que reconozca los bucles y los clasifique como indica la Sección 17.5.3.

17.10. Amplíe la herramienta descrita en el Problema 17.9 para que genere casos de prueba para cada categoría de bucle, cuando los encuentre. Será necesario llevar a cabo esta función de forma interactiva con el encargado de la prueba.

17.11. Dé por lo menos tres ejemplos en los que la prueba de caja negra pueda dar la impresión de que «todo está bien», mientras que la prueba de caja blanca pudiera descubrir errores. Indique por lo menos tres ejemplos en los que la prueba de caja blanca pueda dar la impresión de que «todo está bien», mientras que la prueba de caja negra pudiera descubrir errores.

17.12. ¿Podría una prueba exhaustiva (incluso si fuera posible para pequeños programas) garantizar que un programa es al 100 por 100 correcto?

17.13. Usando el método de la partición equivalente, obtenga un conjunto de casos de prueba para el sistema *HogarSeguro* descrito anteriormente en el libro.

17.14. Mediante el análisis de valores límite, obtenga un conjunto de casos de prueba para el sistema SSRB descrito en el Problema 12.13.

17.15. Investigue un poco y escriba un breve documento sobre el mecanismo de generación de tablas ortogonales para la prueba de datos.

17.16. Seleccione una IGU específica para un programa con el que esté familiarizado y diseñe una serie de pruebas para ejercitar la IGU.

17.17. Investigue en un sistema Cliente/Servidor que le sea familiar. Desarrolle un conjunto de escenarios de usuario y genere un perfil operacional para el sistema.

17.18. Pruebe un manual de usuario (o una facilidad de ayuda) de una aplicación que utilice frecuentemente. Encuentre al menos un error en la documentación.

LECTURAS Y FUENTES DE INFORMACIÓN

La ingeniería del software presenta tanto desafíos técnicos como de gestión. Los libros de Black (*Managing the Testing Process*, Microsoft Press, 1999), Dustin, Rashka y Paul (*Test process Improvement: Step-By-Step Guide to Structured Testing*, Addison-Wesley, 1999), Peny (*Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach*, Dorset House, 1997), y Kit y Finzi (*Software Testing in the Real World: Improving the Process*, Addison-Wesley, 1995) orientan sobre las necesidades de gestión y de procesos.

Para aquellos lectores que deseen información adicional sobre la tecnología de prueba del software, existen varios libros excelentes. Kaner, Nguyen y Falk (*Testing Computer Software*, Wiley, 1999), Hutcheson (*Software Testing Methods and Metrics: The Most Important Test*, McGraw-Hill, 1997), Marick (*The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing*, Prentice-Hall, 1995), Jorgensen (*Software Testing: A Craftman's Approach*, CRC Press, 1995) presentan estudios sobre los métodos y estrategias de prueba.

Myers [MYE79] permanece como un texto clásico, cubriendo con considerable detalle las técnicas de caja negra. Beizer [BEI90] da una amplia cobertura de las técnicas de caja blanca, introduciendo cierto nivel de rigor matemático que a menudo se echa en falta en otros tratados sobre la prueba. Su último libro [BEI95] presenta un tratado conciso de métodos importantes. Perry (*Effective Methods for Software Testing*, Wiley-QED, 1995), y Freeman y Voas (*Software Assesment: Reliability, Sfety, Testability*, Wiley, 1995) presentan buenas introducciones a las estrategias y tácticas de las pruebas. Mosley (*The Handbook of MIS Application Software Testing*, Prentice-Hall, 1993) estudia aspectos de las pruebas para grandes sistemas de información, y Marks (*Testing Very Big Systems*, McGraw-Hill, 1992) estudia los aspectos especiales que deben considerarse cuando se prueban grandes sistemas de programación.

La prueba del software es un recurso en continua actividad. Es por esta razón por lo que muchas organizaciones automatizan parte de los procesos de prueba. Los libros de Dustin, Rashka y Poston (*Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley, 1999) y Poston (*Automating Specification-Based Software Testing*, IEEE Computer Society, 1996) hablan sobre herramientas, estrategias y métodos para automatizar la prueba. Una excelente fuente de información sobre herramientas automatizadas de prueba la encontramos en *Testing Tools Reference Guide* (Software Quality Engineering, Inc., Jacksonville, FL, actualizada anualmente). Este directorio contiene descripciones de cientos de herramientas de prueba, clasificadas por tipo de prueba, plataforma hardware y soporte software.

Algunos libros tratan los métodos y estrategias de prueba en áreas de aplicación especializada. Gardiner (*Testing Safety-Related Software: A Practica1 Handbook*, Springer Verlag, 1999) ha editado un libro que trata la prueba en sistemas de seguridad crítica. Mosley (*Client/Server Software Testing on the Desk Top and the Web*, Prentice Hall, 1999) trata la prueba para clientes, servidores y componentes en red. Rubin (*Handbook of Usability Testing*, Wiley, 1994) ha escrito una guía útil para lo que necesitan manejar interfaces humanas.

Una amplia variedad de fuentes de información sobre pruebas del software y elementos relacionados están disponibles en internet. Una lista actualizada de referencias a páginas web que son relevantes sobre los conceptos de prueba, métodos y estrategias se pueden encontrar en **http://www.pressman5.com**.