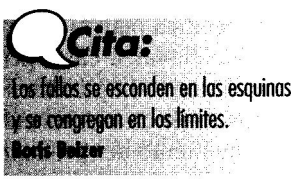
17.3 PRUEBA DE CAJA BLANCA

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.

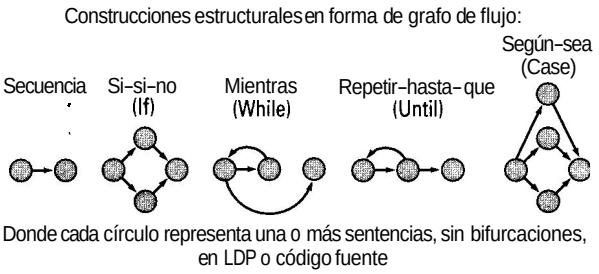


FIGURA 17.1. Notación de grafo **de flujo.**

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.

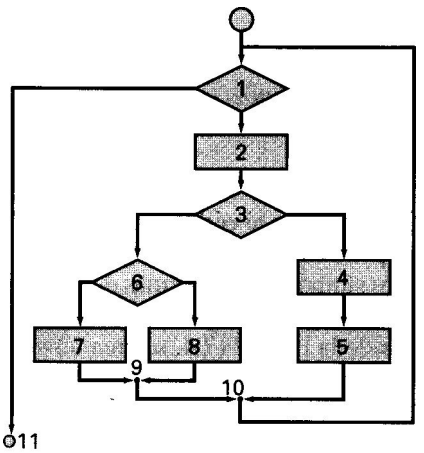
Dibujo un yrofo de flujo cuando lo lógica de lo estructura de control de un módulo seo complejo. El grafo de flujo te permite trazar más fácilmente los cominos del programa.

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

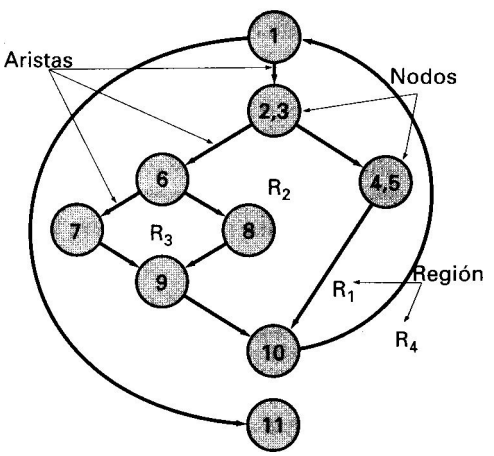
3 Realmente, el método del camino básico se puede llevar a cabo sin usar grafos de flujo. Sin embargo, sirve como herramienta Útil para ilustrar el método.

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.

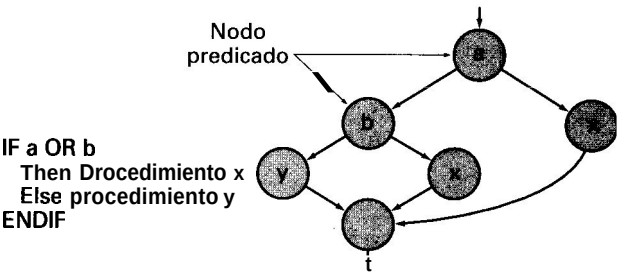


FIGURA 17.3. Lógica compuesta.

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

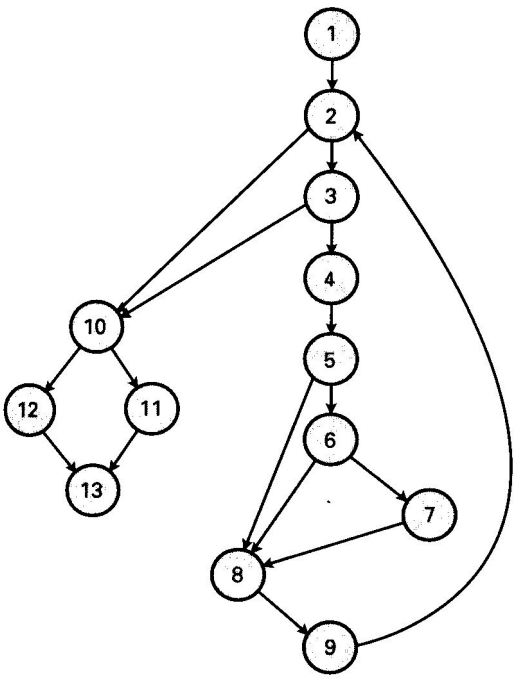


FIGURA 17.5. Grafo de flujo del procedimiento media.

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.***

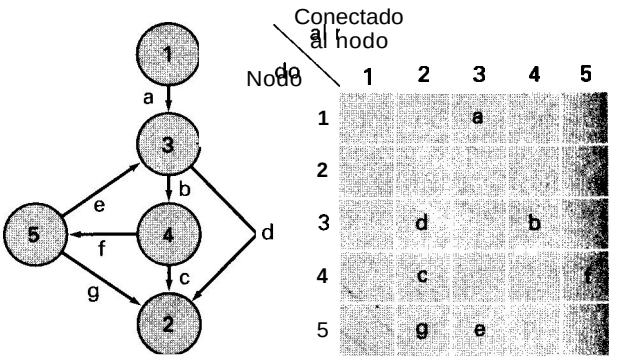


FIGURA 17.6. Matriz del grafo.

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.

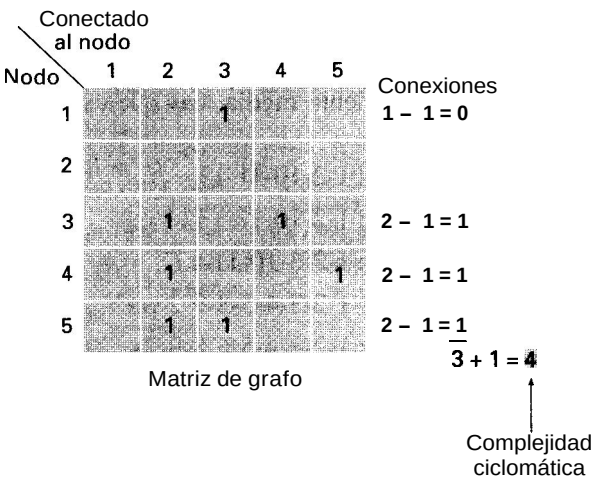


FIGURA 17.7. Matriz de conexiones.

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_2 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_3 es de la forma (D_1, D_2), donde todos los D_1 **y** D_2 son >, = o <. Puesto **que** C_3 es igual que C_2 excepto en que la primera condición simple de C_3 es una expresión relacional, podemos construir un conjunto de restricciones para C_3 mediante la modificación del conjunto de restricciones para C_2, obteniendo

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

La cobertura de este conjunto de restricciones **garantizará** la detección de errores de operador relacional en C_3.

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 $n+1$ pasos por el bucle

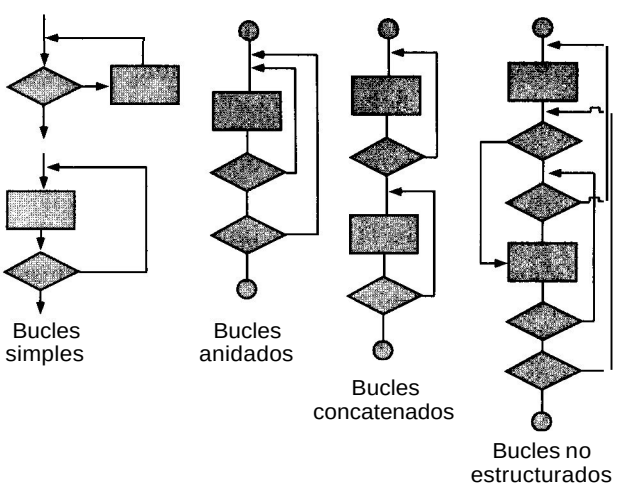


FIGURA 17.8. Clases de **bucles.**

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.