Técnicas de prueba

El desarrollo de Sistemas de software implica la realización de una serie de actividades predispuestas a incorporar errores (en la etapa de definición de requerimientos, de diseño, de desarrollo, ...).

Debido a que estos errores se deben a nuestra habilidad innata de provocar errores, tenemos que incorporar una actividad que garantice la calidad del software.

En este capítulo estudiaremos:

- Fundamentos de la prueba del software, que definen los objetivos fundamentales de la fase de prueba.
- Diseño de casos de prueba, que se centra en un conjunto de técnicas para que satisfagan los objetivos globales de la prueba.

1. FUNDAMENTOS DE LA PRUEBA DEL SOFTWARE

En la etapa de prueba del software se crean una serie de casos de prueba que intentan "destruir" el software desarrollado.

La prueba requiere que se descarten ideas preconcebidas sobre la "calidad o corrección" del software desarrollado.

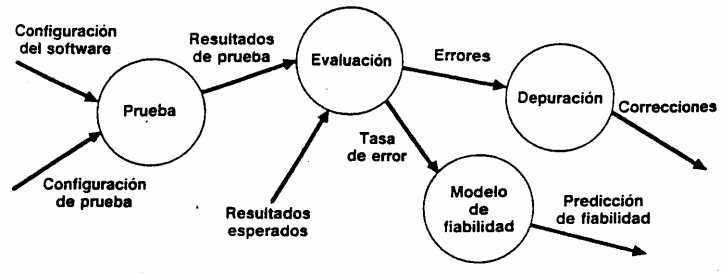
1.1. Objetivos de la prueba

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

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

La prueba no puede asegurar la ausencia de errores; sólo puede demostrar que existen defectos en el software.

1.2. Proceso de prueba



El proceso de prueba tiene dos entradas:

- Configuración del software: Incluye la especificación de requisitos del software, la especificación del diseño y el código fuente
- Configuración de prueba: Incluye un plan y un procedimiento de prueba

Si el funcionamiento del software parece ser correcto y los errores encontrados son fáciles de corregir, podemos concluir con que:

- La calidad y la fiabilidad del software son aceptables, o que
- Las pruebas son inadecuadas para descubrir errores serios

1.3. Diseño de casos de prueba

Se trata de diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y de tiempo.

Cualquier producto de ingeniería se puede probar de dos formas:

- **Pruebas de caja negra:** Realizar pruebas de forma que se compruebe que cada función es operativa.
- **Pruebas de caja blanca:** Desarrollar pruebas de forma que se asegure que la operación interna se ajusta a las especificaciones, y que todos los componentes internos se han probado de forma adecuada.

En la prueba de la caja negra, 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 una salida correcta.

En la prueba de caja blanca se realiza un examen minucioso de los detalles procedimentales, comprobando los caminos lógicos del programa, comprobando los bucles y condiciones, y examinado el estado del programa en varios puntos.

A primera vista, la prueba de caja blanca profunda nos llevaría a tener "programas 100 por cien correctos", es decir:

- Definir todos los caminos lógicos
- Desarrollar casos de prueba para todos los caminos lógicos
- Evaluar los resultados

Pero esto supone un estudio demasiado exhaustivo, que prolongaría excesivamente los planes de desarrollo del software, por lo que se hará un estudio de los caminos lógicos importantes.

2. PRUEBA DE LA CAJA BLANCA

La prueba de la caja blanca es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para derivar los casos de prueba.

Las pruebas de caja blanca intentan garantizar que:

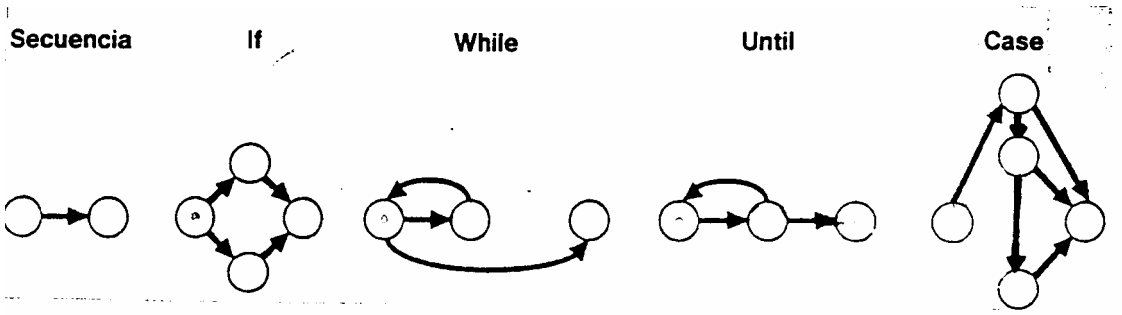
- Se ejecutan al menos una vez todos los caminos independientes de cada módulo
- Se utilizan las decisiones en su parte verdadera y en su parte falsa
- Se ejecuten todos los bucles en sus límites
- Se utilizan todas las estructuras de datos internas

2.1. Prueba del camino básico

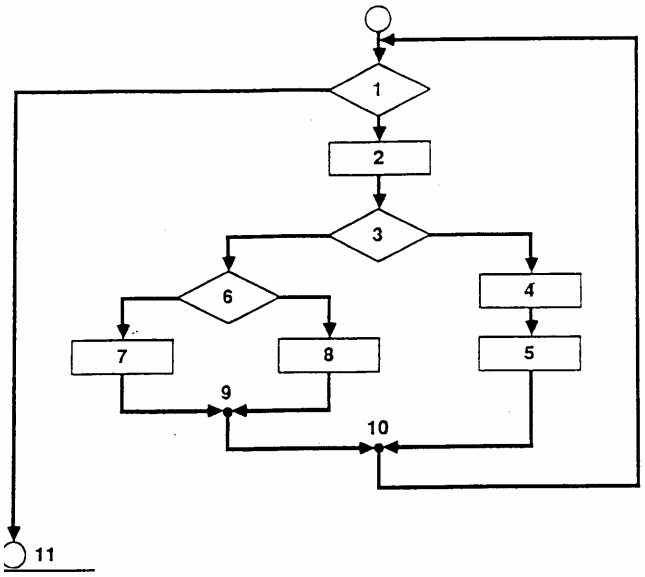
El método del camino básico (propuesto por McCabe) permite obtener una medida de la complejidad de un diseño procedimental, y utilizar esta medida como guía para la definición de una serie de caminos básicos de ejecución, diseñando casos de prueba que garanticen que cada camino se ejecuta al menos una vez.

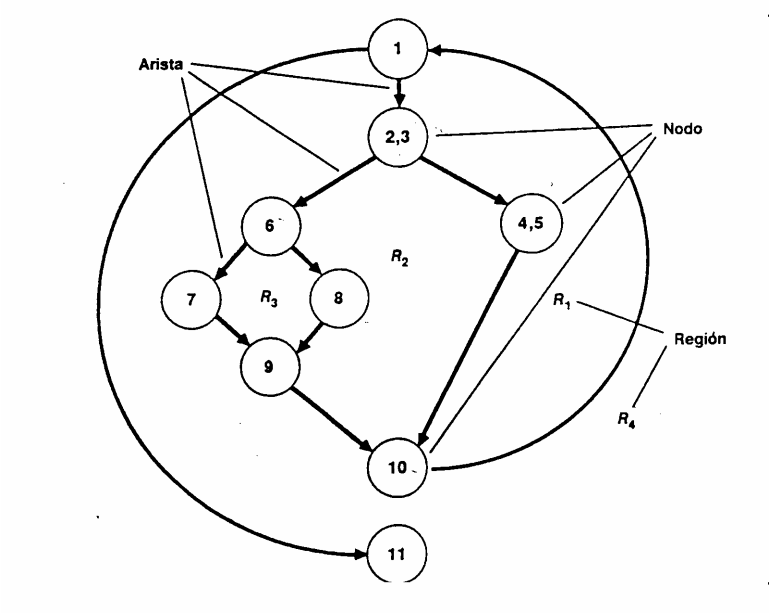
2.1.1. Notación del grafo de flujo o grafo del programa

Representa el flujo de control lógico con la siguiente notación:



A continuación se muestra un ejemplo basado en un diagrama de flujo que representa la estructura de control del programa.



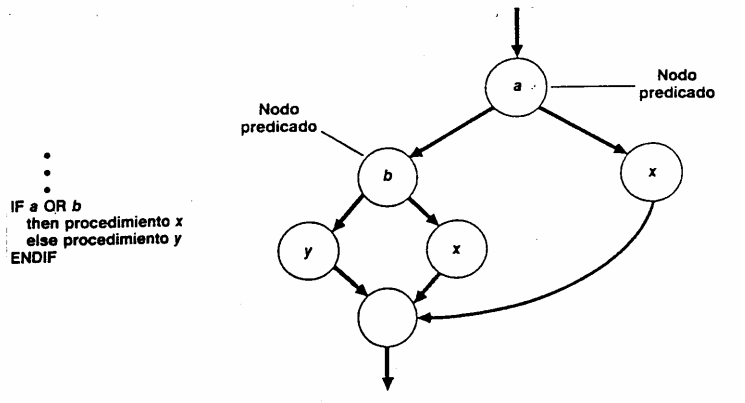


En el grafo de flujo

- Cada nodo representa una o más sentencias procedimentales
- Un solo nodo puede corresponder a una secuencia de pasos del proceso y a una decisión
- Las flechas (aristas) representan el flujo de control

Cualquier representación del diseño procedimental se puede traducir a un grafo de flujo.

Si en el diseño procedimental se utilizan condiciones compuestas, la generación del grafo de flujo tiene que descomponer las condiciones compuestas en condiciones sencillas, tal y como muestra la figura siguiente.



2.1.2. Complejidad ciclomática

Es una medida que proporciona una idea de la complejidad lógica de un programa.

- La complejidad ciclomática coincide con el número de regiones del grafo de flujo
- La complejidad ciclomática, *V(G)*, de un grafo de flujo *G*, se define como

  V(G) = Aristas - Nodos + 2
- La complejidad ciclomática, *V(G)*, de un grafo de flujo *G*, también se define como

  V(G) = Nodos de predicado + 1

A partir del grafo de flujo de la figura 4, la complejidad ciclomática sería:

- Como el grafo tiene cuatro regiones, *V(G) = 4*
- Como el grafo tiene 11 aristas y 9 nodos, *V(G) = 11 - 9 - 2 = 4*
- Como el grafo tiene 3 nodos predicado, *V(G) = 3 + 1 = 4*

A partir del valor de la complejidad ciclomática obtenemos el número de caminos independientes, que nos dan un valor límite para el número de pruebas que tenemos que diseñar.

En el ejemplo, el número de caminos independientes es 4, y los caminos independientes son:

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

2.1.3. Pasos del diseño de pruebas mediante el camino básico

- Obtener el grafo de flujo, a partir del diseño o del código del módulo
- Obtener la complejidad ciclomática del grafo de flujo
- Definir el conjunto básico de caminos independientes
- Determinar los casos de prueba que permitan la ejecución de cada uno de los caminos anteriores
- Ejecutar cada caso de prueba y comprobar que los resultados son los esperados

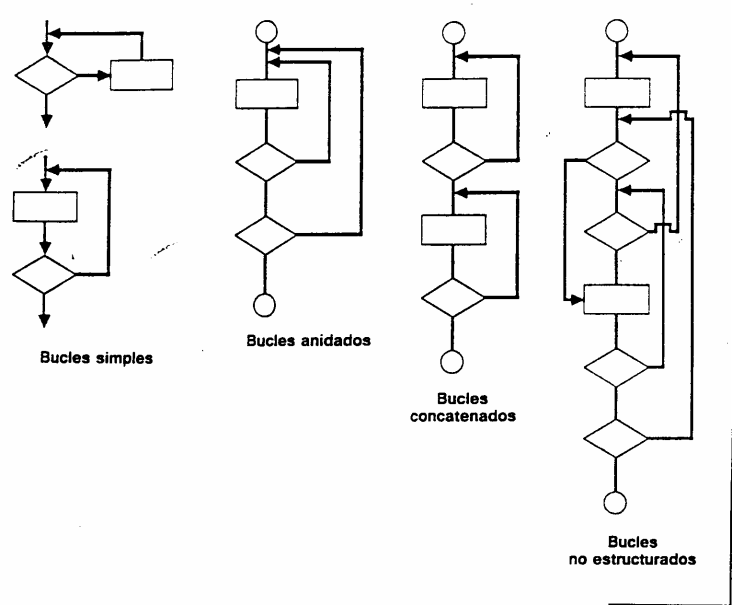
2.2. Prueba de bucles

Los bucles son la piedra angular de la inmensa mayoría de los algoritmos implementados en software, por lo que tenemos que prestarles una atención especial a la hora de realizar la prueba del software.

La prueba de bucles es una técnica de prueba de caja blanca que se centra en la validez de las construcciones de los bucles.

Se pueden definir cuatro tipos de bucles diferentes:

- Bucles simples
- Bucles concatenados
- Bucles anidados
- Bucles no estructurados



2.2.1. Bucles simples

A los bucles simples (de *n* iteraciones) se les tiene que aplicar el conjunto de pruebas siguientes:

- Saltar el bucle
- Pasar sólo una vez por el bucle
- Pasar dos veces por el bucle
- Hacer *m* pasos del bucle con $m < n$
- Hacer *n-1, n* y *n+1* pasos por el bucle

2.2.2. Bucles anidados

Si extendiésemos el conjunto de pruebas de los bucles simples a los bucles anidados, el número de pruebas crecería geométricamente, por lo que Beizer sugiere el siguiente conjunto de pruebas para bucles anidados:

- Comenzar con el bucle más interno, estableciendo los demás bucles a los valores mínimos
- Llevar a cabo las pruebas de bucles simples para el bucle más interno, conservando los valores de iteración de los bucles más externos a los valores mínimos
- Progresar hacia fuera en el siguiente bucle más externo, y manteniendo los bucles más externos a sus valores mínimos
- Continuar hasta que se hayan probado todos los bucles

2.2.3. Bucles concatenados

Probar los bucles concatenados mediante las técnicas de prueba para bucles simples, considerándolos como bucles independientes.

2.2.4. Bucles no estructurados

Rediseñar estos bucles para que se ajusten a las construcciones de la programación estructurada.

Ejemplo

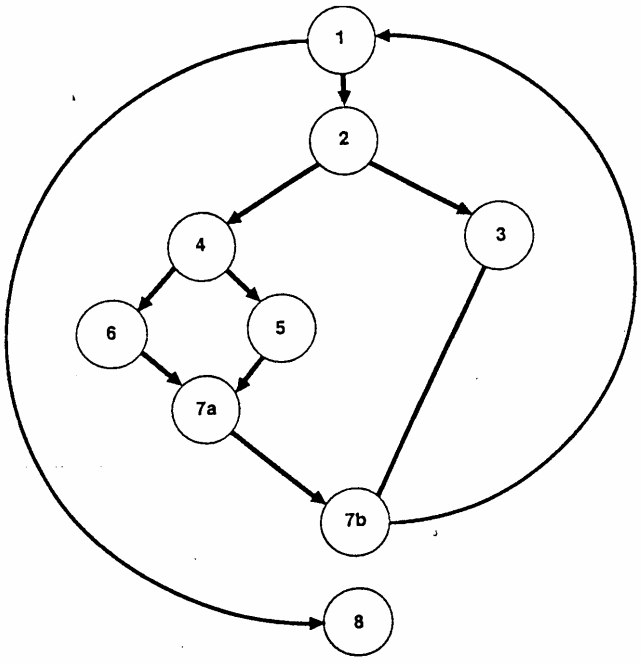
Construir el Grafo de Flujo correspondiente a la siguiente especificación del software en LDP.

```
                    LDP

      procedimiento: ordenar

1:      do while queden registros
          leer registros;
2:        if campo 1 del registro = 0
3:          then procesar registro;
                 guardar en buffer;
                 incrementar contador;
4:          elseif campo 2 del registro = 0
5:            then reinicializar contador;
6:            else procesar registro;
                 guardar en archivo;
7a:         endif
        endif
7b:   enddo
8:0: end
```

Solución

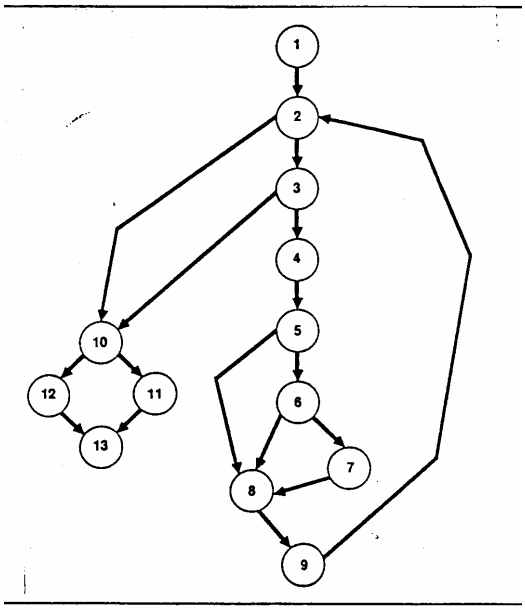


Ejemplo

Construir el Grafo de Flujo correspondiente al siguiente código de un programa

```
PROCEDURE media;
    * Este procedimiento calcula la media de 100 o menos
      números que se encuentren entre límites; también calcula
      el total de entradas y el total de válidos.
    INTERFACE RETURNS media, total.entrada.total.válido;
    INTERFACE ACCEPTS 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;
(1) i= 1;
    total.entrada =total.válido = 0;
    suma = 0;
(2) DO WHILE valor[i] < > -999 and total.entrada < 100
(3) (4) increment total.entrada by 1;
(5)     IF valor[i] > = mínimo AND valor[i] < = máximo
(6)     THEN increment total.válido by 1;
(7)          suma = suma + valor[i];
        ELSE skip
(8)  ENDIF
     increment i by 1;
(9) ENDDO
(10) IF total.válido > 0
(11)    THEN media = suma/total.válido;
(12)    ELSE media = -999;
(13) ENDIF
END media
```

Solución



3. PRUEBA DE LA CAJA NEGRA

Las pruebas de caja negra se llevan a cabo sobre la interfaz del software, obviando el comportamiento interno y la estructura del programa.

Los casos de prueba de la caja negra pretenden demostrar que:

- Las funciones del software son operativas
- La entrada se acepta de forma correcta
- Se produce una salida correcta
- La integridad de la información externa se mantiene

A continuación se derivan conjuntos de condiciones de entrada que utilicen todos los requisitos funcionales de un programa.

Las pruebas de caja negra pretenden encontrar estos tipos de errores:

- Funciones incorrectas o ausentes
- Errores en la interfaz
- Errores en estructuras de datos o en accesos a bases de datos externas
- Errores de rendimento
- Errores de inicialización y de terminación

Los tipos de prueba de cana negra que vamos a estudiar son:

- Prueba de partición equivalente
- Prueba de análisis de valores límites

3.1. Prueba de partición equivalente

Este método de prueba de caja negra divide el dominio de entrada de un programa en clases de datos, a partir de las cuales deriva los casos de prueba.

Cada una de estas clases de equivalencia representa a un conjunto de estados válidos o inválidos para las condiciones de entrada.

3.1.1. Identificación de las clases de equivalencia

Se identifican clases de equivalencia válidas e inválidas con la siguiente tabla

Condiciones externas	Clases de equivalencia válidas	Clases de equivalencia inválidas

A continuación se siguen estas directrices:

- Si una condición de entrada especifica un rango de valores (p.e, entre 1 y 999), se define una CEV (1 <= valor <= 999) y dos CEI (valor < 1 y valor > 999)
- Si una CE requiere un valor específico (p.e., el primer carácter tiene que ser una letra), se define una CEV (una letra) y una CEI (no es una letra)
- Si una CE especifica un conjunto de valores de entrada, se define una CEV para cada uno de los valores válidos, y una CEI (p.e., CEV para "Moto", "Coche" y "Camión", y CEI para "Bicicleta")
- Si una condición de entrada especifica el número de valores (p.e., una casa puede tener uno o dos propietarios), identificar una CEV y dos CEI (0 propietarios y 3 propietarios)

3.1.2. Identificación de casos de prueba

Seguir estos pasos

- Asignar un número único a cada clase de equivalencia
- Escribir casos de prueba hasta que sean cubiertas todas las CEV, intentando cubrir en cada casos tantas CEV como sea posible
- Para cada CEI, escribir un caso de prueba, cubriendo en cada caso una CEI

Ejemplo

Diseñar casos de prueba de partición equivalente para un software que capte estos datos de entrada:

- *Código de área:* En blanco o un número de tres dígitos
- *Prefijo:* Número de tres dígitos que no comiencen por 0 ó 1
- *Sufijo:* Número de cuatro dígitos
- *Ordenes:* "Cheque", "Depósito", "Pago factura"
- *Palabra clave:* Valor alfanumérico de 6 dígitos