TÉCNICAS DE EVALUACIÓN DINÁMICA

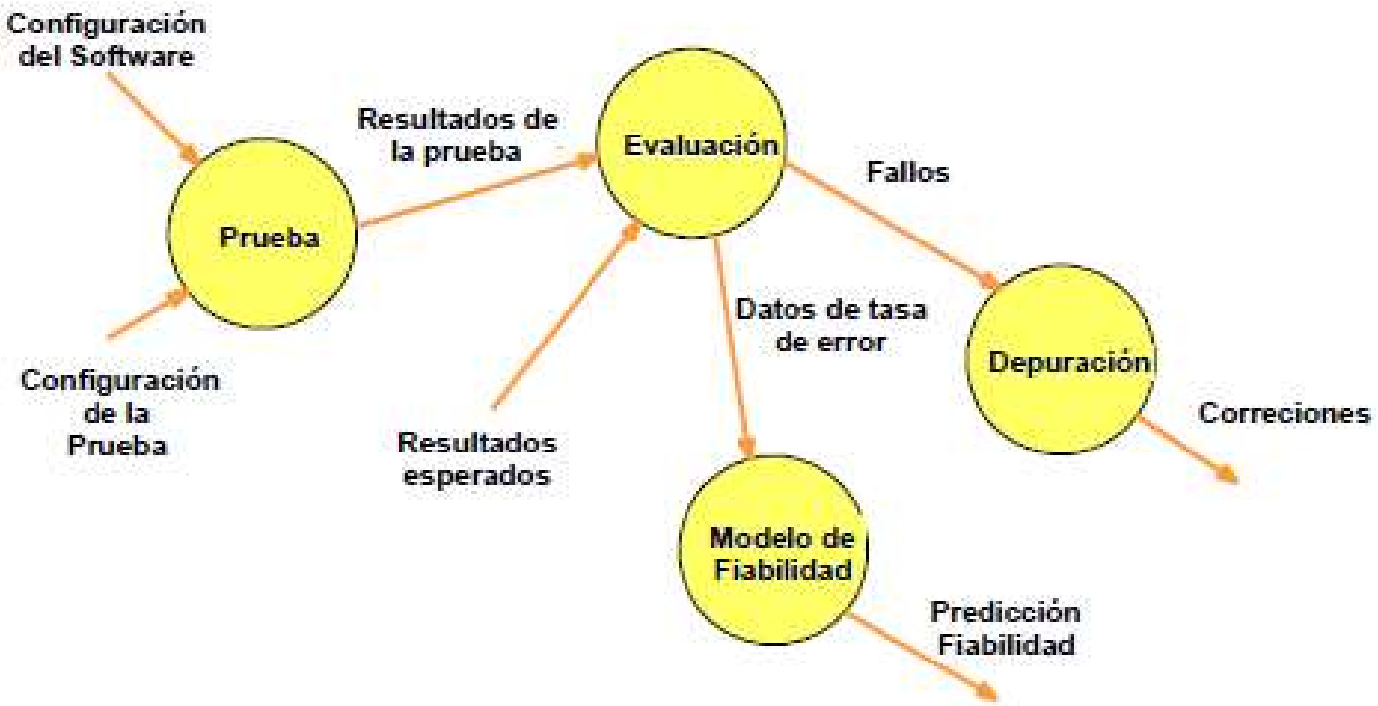
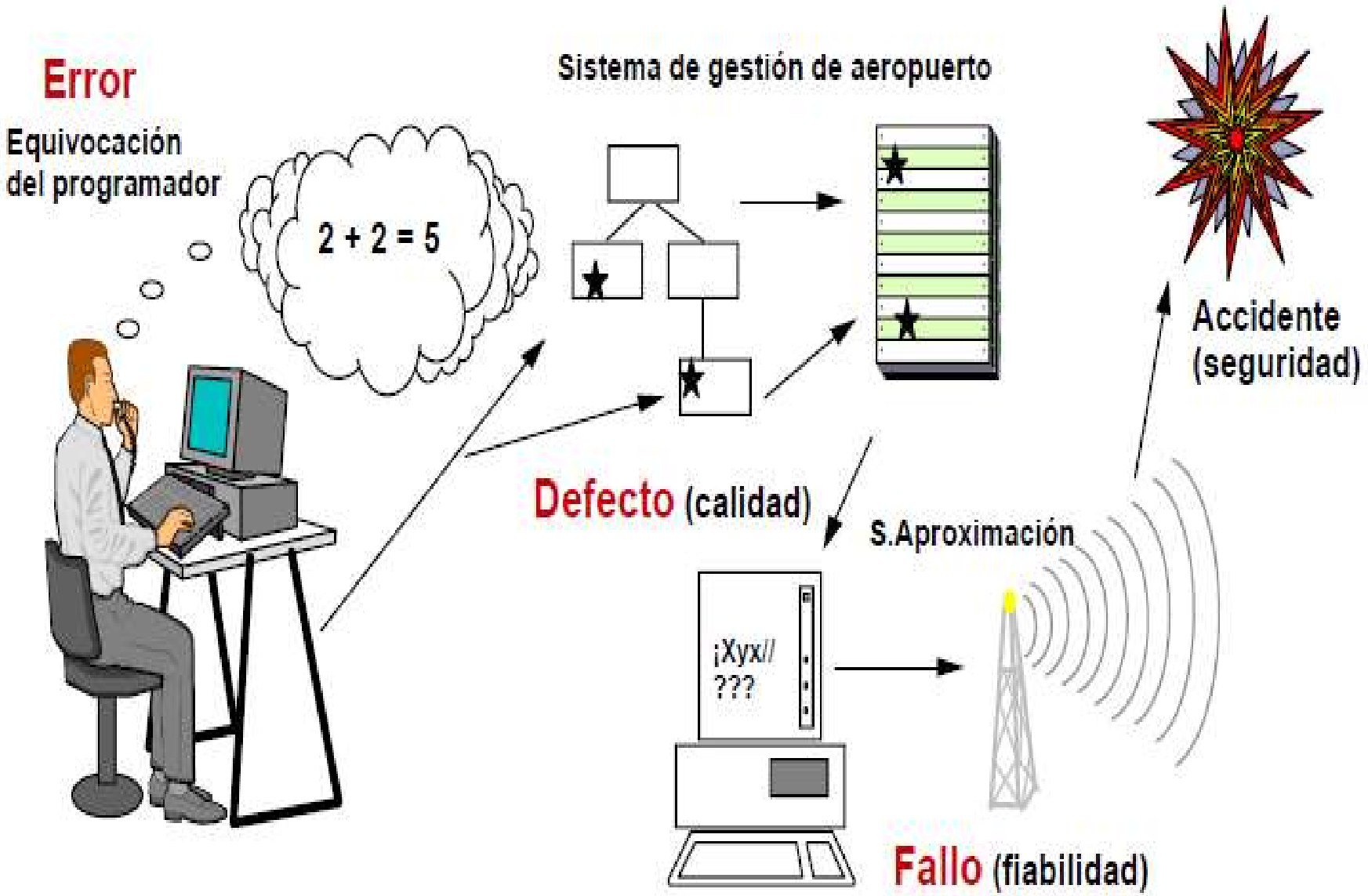


Figura 2. Contexto de la Prueba de Software

Relación entre Error, Defecto y Fallo:



3.Pruebas - Conceptos

- **Ejemplo** de las dificultades que se presentan

```
if (A+B+C)/3 == A
then print ("A, B y C son iguales")
else print ("A, B y C no son iguales")
```

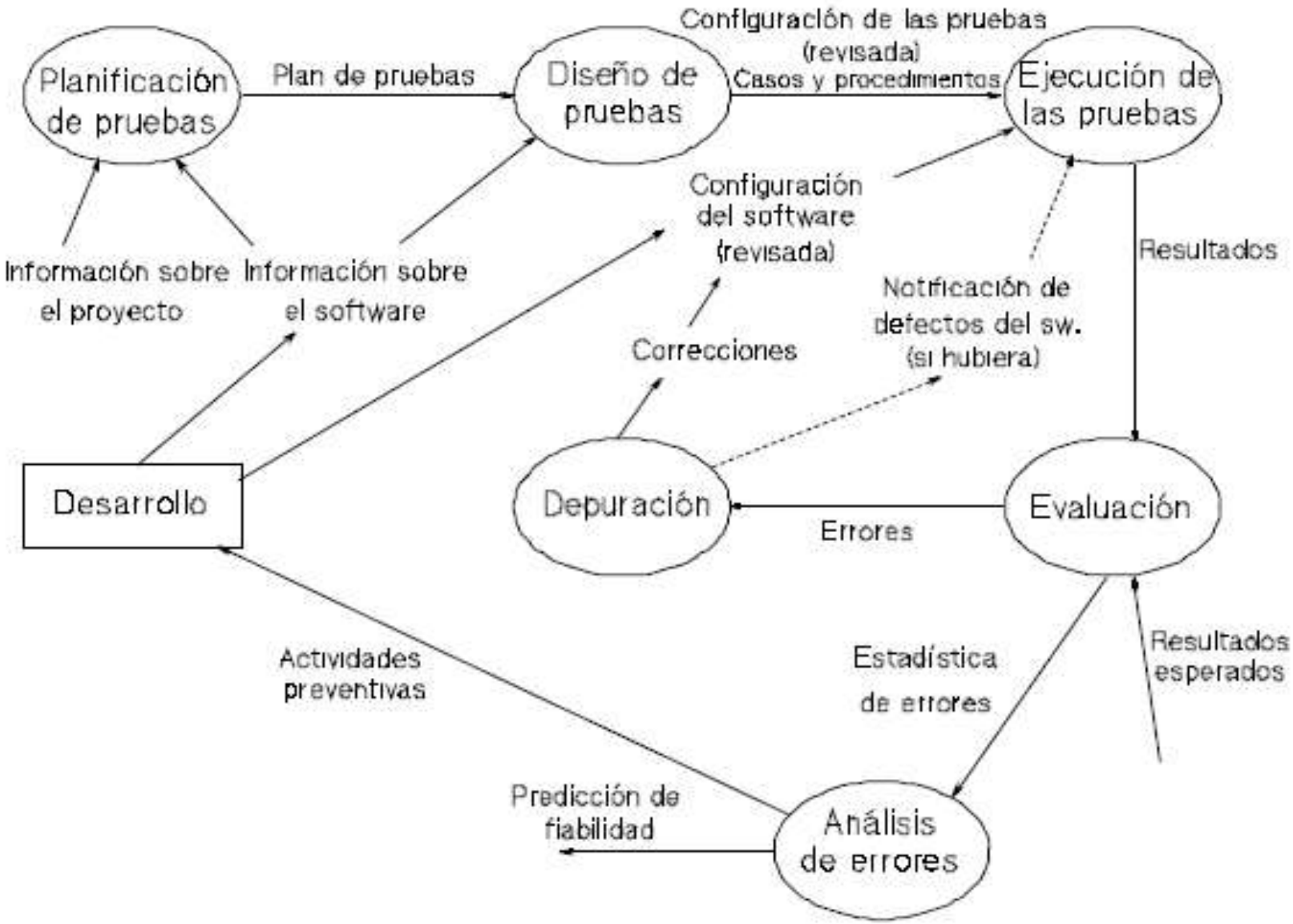
¿Son suficientes estos dos casos de prueba?

Caso 1: A=5; B=5; C=5;

Caso 2: A=2; B=3; C=7;

¿Cuáles otros serían necesarios en caso negativo?

3.Pruebas - Proceso



TÉCNICAS DE EVALUACIÓN DINÁMICA

- El objetivo de las pruebas no es asegurar la ausencia de defectos en un software, únicamente puede demostrar que existen defectos en el software.

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

- Para ser más eficaces (es decir, con más alta probabilidad de encontrar errores), las pruebas deberían ser realizadas por un equipo independiente al que realizó el software.

Características de una buena PRUEBA

- Debe tener una alta probabilidad de encontrar un fallo.
- Debe centrarse en dos objetivos:
  - 1) *probar si el software no hace lo que debe hacer, y*
  - *2) probar si el software hace lo que no debe hacer.*
- No debe ser redundante. El tiempo y los recursos son limitados, así que *todas las pruebas deberían tener un propósito diferente.*
- Se debería emplear la prueba que tenga la *más alta probabilidad de descubrir una clase entera de errores.*
- Una buena prueba no debería ser ni demasiado sencilla ni demasiado compleja, pero si se quieren combinar varias pruebas a la vez se pueden enmascarar errores, por lo que en general, *cada prueba debería realizarse separadamente.*

Tareas a realizar para probar un software

1) *Diseño de las pruebas. Esto es, identificación de la técnica o técnicas de pruebas que se* utilizarán para probar el software.
2) *Generación de los casos de prueba. Los casos de prueba representan los datos que se* utilizarán como entrada para ejecutar el software a probar. Más concretamente los casos de prueba determinan un conjunto de entradas, condiciones de ejecución y resultados esperados para un objetivo particular.
3) *Definición de los procedimientos de la prueba. Esto es, especificación de cómo se va a* llevar a cabo el proceso, quién lo va a realizar, cuándo, …
4) *Ejecución de la prueba, aplicando los casos de prueba generados previamente e* identificando los posibles fallos producidos al comparar los resultados esperados con los resultados obtenidos.
5) *Realización de un informe de la prueba, con el resultado de la ejecución de las pruebas, qué* casos de prueba pasaron satisfactoriamente, cuáles no, y qué fallos se detectaron.

Introducción.

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 inicio del proceso, en el que los objetivos pueden estar especificados de forma errónea o imperfecta, así como dentro de pasos de diseño y desarrollo posteriores..

Introducción.

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

La prueba del software es un elemento crítico para la garantía de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación.

Objetivos de la prueba

Los objetivos principales de realizar una prueba son:

- **Detectar un error.**
- **Tener un buen caso de prueba, es decir que tenga más probabilidad de mostrar un error no descubierto antes.**
- **Descubrir un error no descubierto antes (éxito de la prueba).**

Objetivos de la prueba

Nuestro objetivo, es diseñar pruebas que sistemáticamente descubran diferentes tipos de errores con menor tiempo y esfuerzo.

La prueba no puede asegurar que no existen errores sólo puede mostrar que existen defectos en el software.

Un software fácil de probar tiene las siguientes características:

- **Operatividad**
- **Observatividad**
- **Controlabilidad**
- **Capacidad de descomposición**
- **Simplicidad**
- **Estabilidad**
- **Facilidad de comprensión.**

Principios de la prueba

- **Hacer un seguimiento de las pruebas hasta los requisitos del cliente.**
- **Plantear y diseñar las pruebas antes de generar ningún código.**
- **El 80% de todos los errores se centran sólo en el 20% de los módulos.**

Principios de la prueba

- **Empezar las pruebas en módulos individuales y avanzar hasta probar el sistema entero.**

- **No son posibles las pruebas exhaustivas.**

- **Deben realizarse por un equipo independiente al equipo de desarrollo.**

3.Pruebas - Niveles

- El ámbito o destino de las pruebas del software puede variar en **tres niveles**:
  - Un **módulo único**
    - PRUEBAS UNITARIAS
  - Un **grupo de módulos** (relacionados por propósito, uso, comportamiento o estructura)
    - PRUEBAS DE INTEGRACIÓN
  - Un **sistema completo**
    - PRUEBAS DE SISTEMA

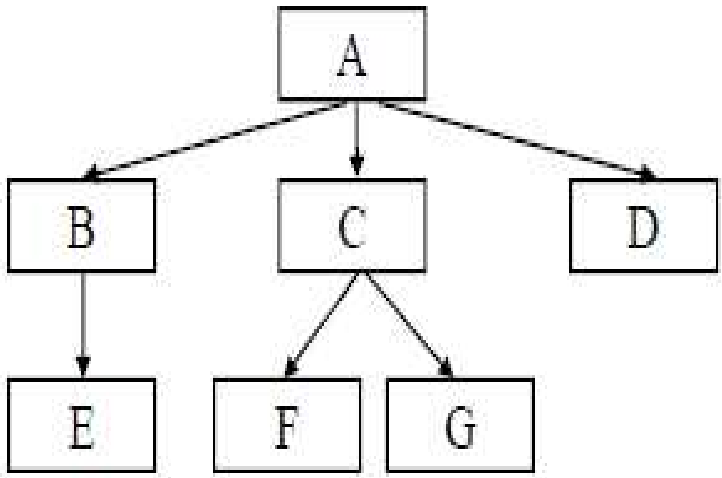
Pruebas Unitarias

- Verifican el **funcionamiento aislado** de piezas de software que pueden ser probadas de forma separada
  - Subprogramas/Módulos individuales
  - Componente que incluye varios subprogramas/módulos
- Estas pruebas suelen llevarse a cabo con:
  - Acceso al código fuente probado
  - Ayuda de herramientas de depuración
  - Participación (opcional) de los programadores que escribieron el código

Pruebas de Integración

- Verifican la **interacción entre componentes** del sistema software
- Estrategias:
  - Guiadas por la arquitectura
    - Los componentes se integran según hilos de funcionalidad
  - Incremental
    - Se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados

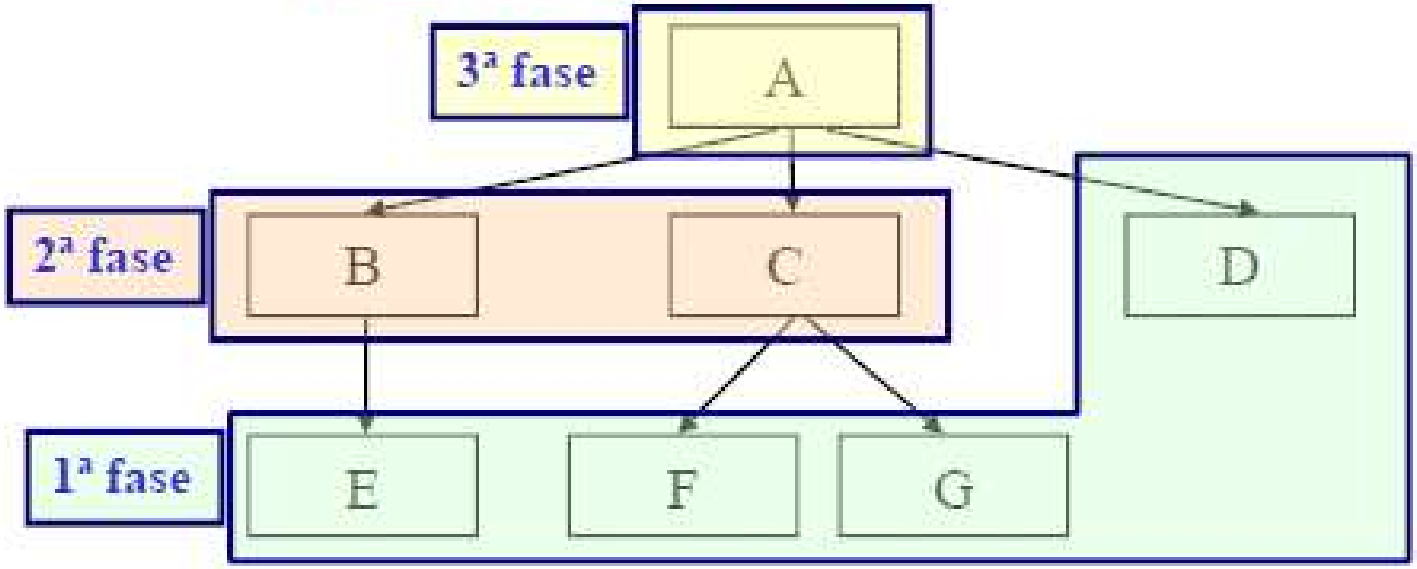
- Incremental Ascendente (Bottom-Up)
  1. Se comienza por los módulos hoja (pruebas unitarias)
  2. Se combinan los módulos según la jerarquía
  3. Se repite en niveles superiores
- Incremental Descendente (Top-Down)
  - Primero en profundidad, completando ramas del árbol
  - Primero en anchura, completando niveles de jerarquía



Pruebas de Integración

Incremental Ascendente (Bottom-Up)

1. **Unitarias de E, F, G y D**
2. **Integración de (B con E), (C con F) y (C con G)**
3. **Integración de (A con B), (A con C) y (A con D)**



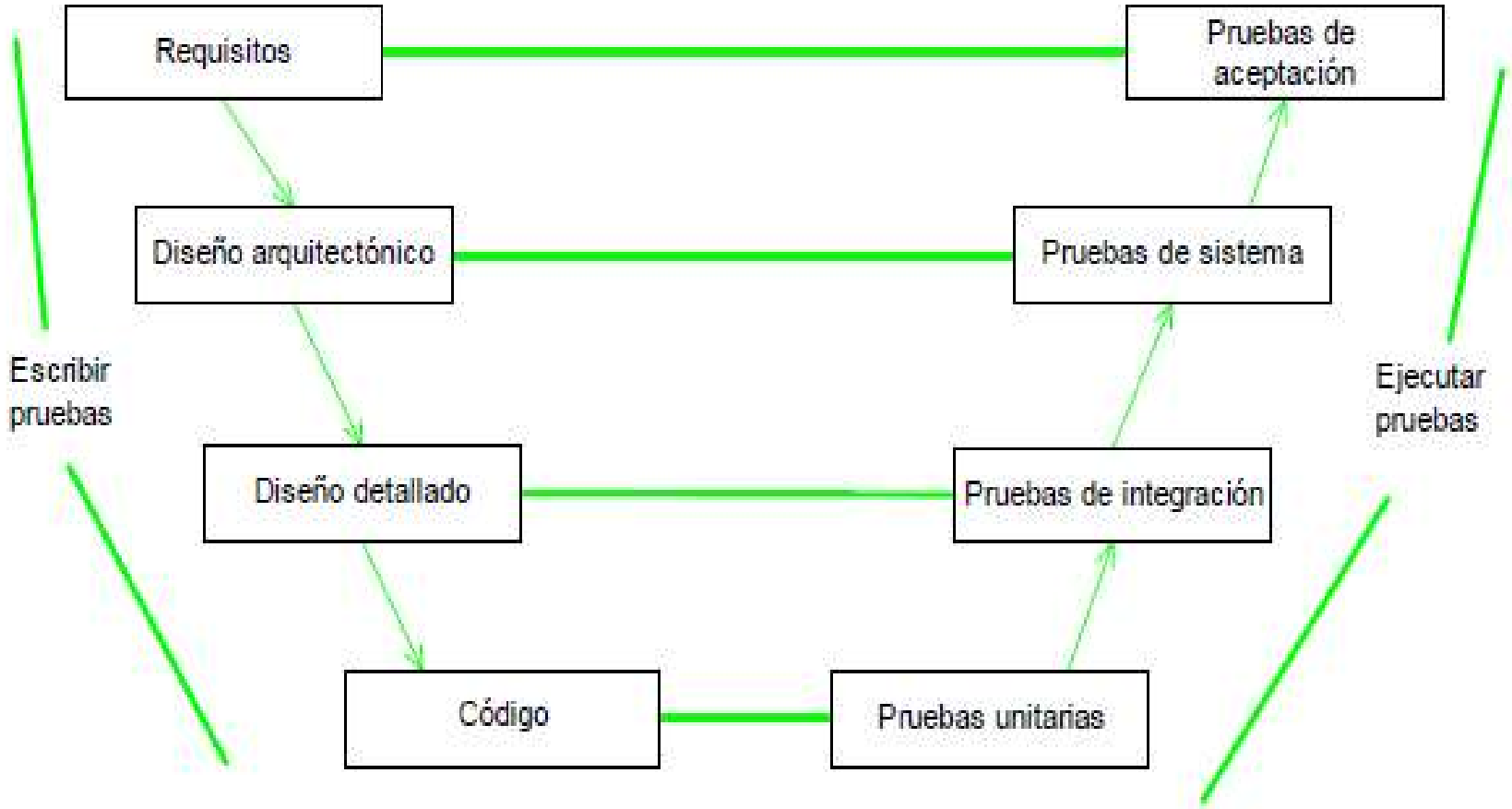
Incremental Descendente (Top-Down)

- Primero en **profundidad**, completando ramas del árbol
  - **(A, B, E, C, F, G, D)**
- Primero en **anchura**, completando niveles de jerarquía
  - **(A, B, C, D, E, F, G)**

Pruebas de Sistema

- Verifican el **comportamiento del sistema** en su conjunto
- Los fallos **funcionales** se suelen detectar en los otros dos niveles anteriores (unitarias e integración)
- Este nivel es más adecuado para comprobar **requisitos no funcionales**
  - Seguridad, Velocidad, Exactitud, Fiabilidad
- También se prueban:
  - Interfaces externos con otros sistemas
  - Utilidades
  - Unidades físicas
  - Entorno operativo

- **Relación entre las Actividades de Desarrollo y las Pruebas**
  - **¿ En qué orden han de escribirse y de realizarse las actividades de pruebas ?**

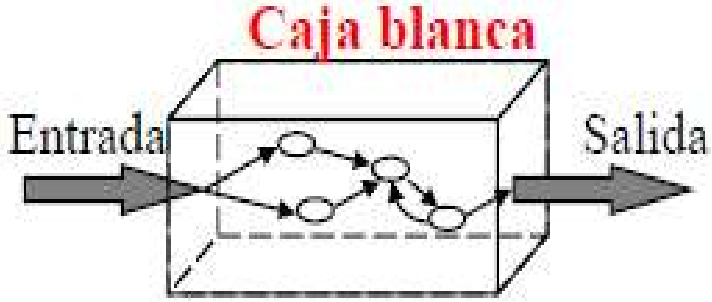


Técnicas de Prueba

- **Principio básico:**
  - Intentar ser lo más **sistemático** posible en identificar un **conjunto representativo de comportamientos** del programa
    - Determinados por subclases del dominio de entradas, escenarios, estados,...
- **Objetivo**
  - "romper" el programa, encontrar el mayor número de fallos posible
- De forma general, existen **dos enfoques** diferentes:
  - **Caja Negra** (**Funcional**)

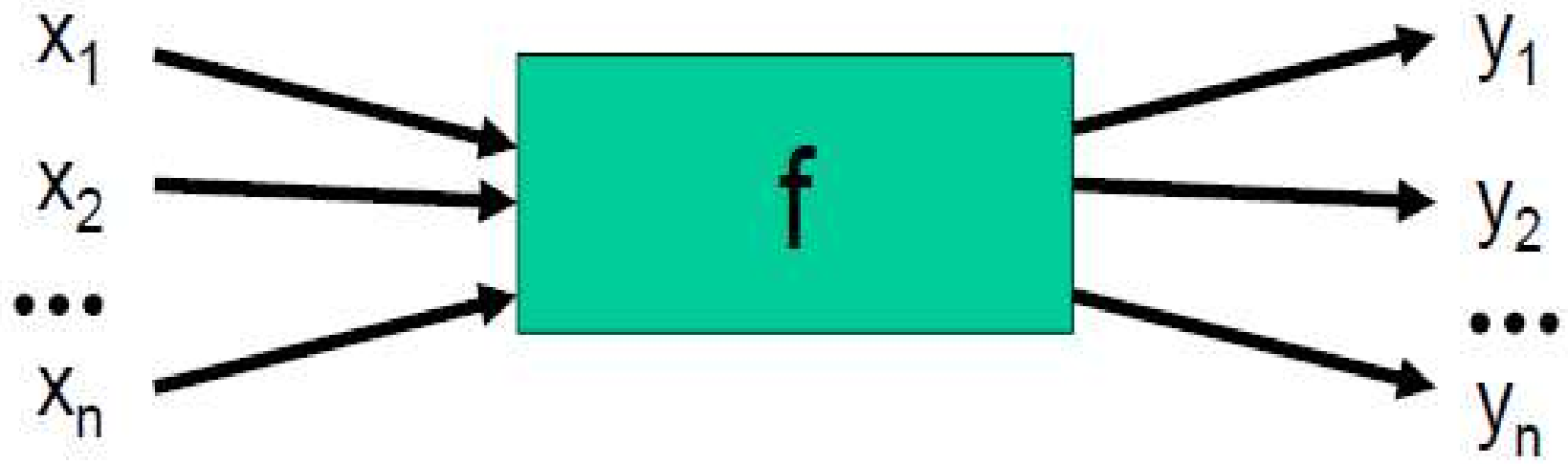
    Los casos de prueba se basan sólo en el comportamiento de entrada/salida
  - **Caja Blanca** (**Estructural**)

    Basadas en información sobre cómo el software ha sido diseñado o codificado



Caja Negra

- Orientadas a los requisitos funcionales



¿$y_i = f(x_i)$, para todo *i*?

Buscan asegurar que:

- Se ha ingresado toda clase de entrada
- Que la salida observada = esperada

Particionamiento Equivalente

- El dominio de las **entradas** se divide en subconjuntos equivalentes respecto de una relación especificada (**clases de equivalencia**):
  - La prueba de un valor representativo de una clase permite suponer «razonablemente» que el resultado obtenido será el mismo que para otro valor de la clase
- Se realiza un conjunto representativo de casos de prueba para cada clase de equivalencia

Particionamiento Equivalente

- **Ejemplo**: Entrada de un Programa
  - Código de Área: Número de tres cifras que no comienza ni por 0 ni por 1
  - Nombre: Seis caracteres
  - Orden: cheque, depósito, pago factura, retirada de fondos

Condición de entrada	Clases válidas	Clases inválidas
Código área		
Nombre para identificar la operación		
Orden		

Particionamiento Equivalente

- **Ejemplo**: Entrada de un Programa
  - Código de Área: Número de tres cifras que no comienza ni por 0 ni por 1
  - Nombre: Seis caracteres
  - Orden: cheque, depósito, pago factura, retirada de fondos

Condición de entrada	Clases válidas	Clases inválidas
Código área	(1) 200≤código≤999	(2) código <200 (3) código >999 (4) no es número
Nombre para identificar la operación	(5) seis caracteres	(6) menos de 6 caracteres (7) más de 6 caracteres
Orden	(8) «cheque» (9) «depósito» (10) «pago factura» (11) «retirada de fondos»	(12) ninguna orden válida

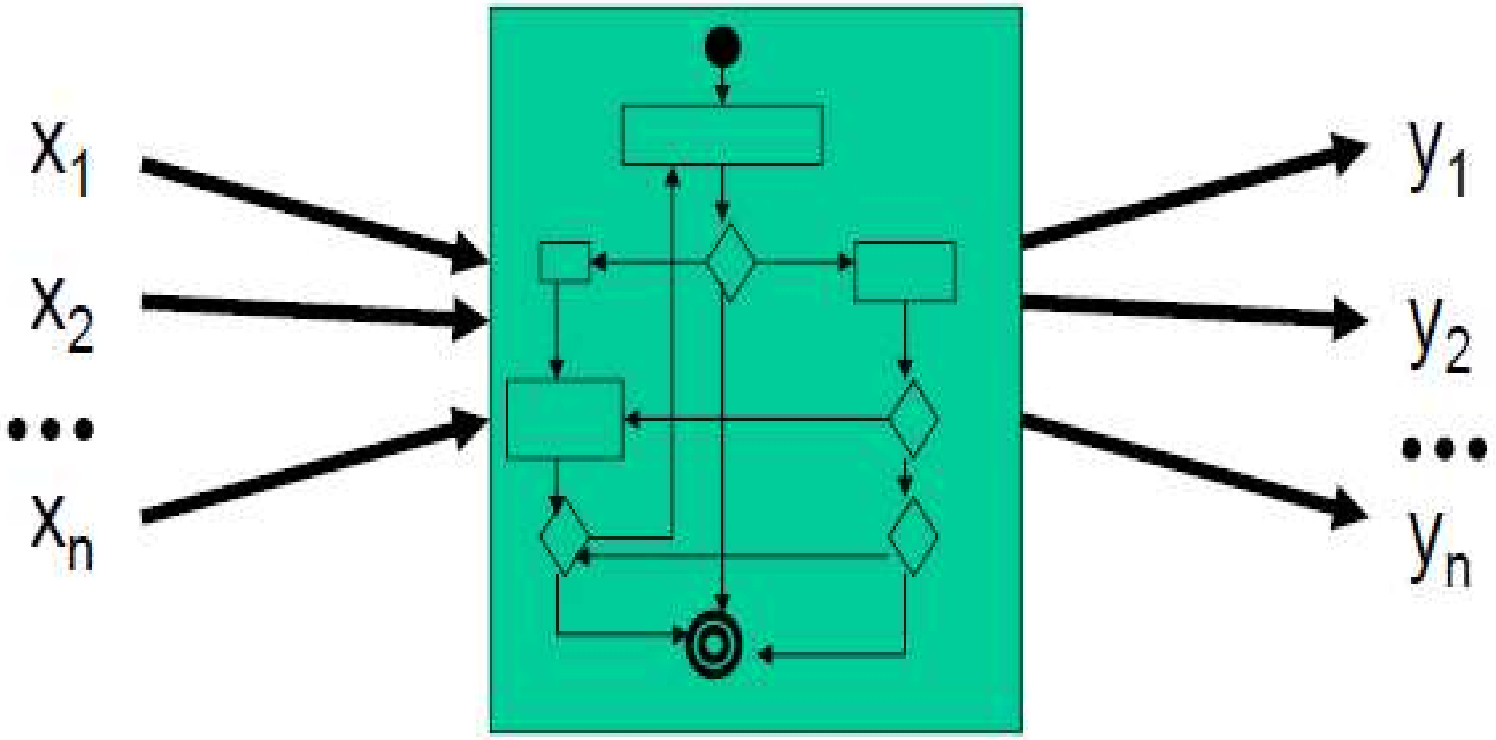
Análisis de Valores Límite

- Similar a la anterior pero los valores de entrada de los casos de prueba se eligen en las cercanías de los **límites** de los dominios de **entrada** de las variables
- **Suposición**: Muchos defectos tienden a concentrarse cerca de los valores extremos de las entradas
- Extensión: **Pruebas de Robustez**, con casos fuera de los dominios de entrada

- Si aplicamos AVL en el ejemplo anterior ¿qué valores para los casos de prueba seleccionaríamos?

Caja Blanca

- De alguna manera, me interesa conocer la **cobertura** alcanzada por mis casos de prueba dentro de la unidad de prueba

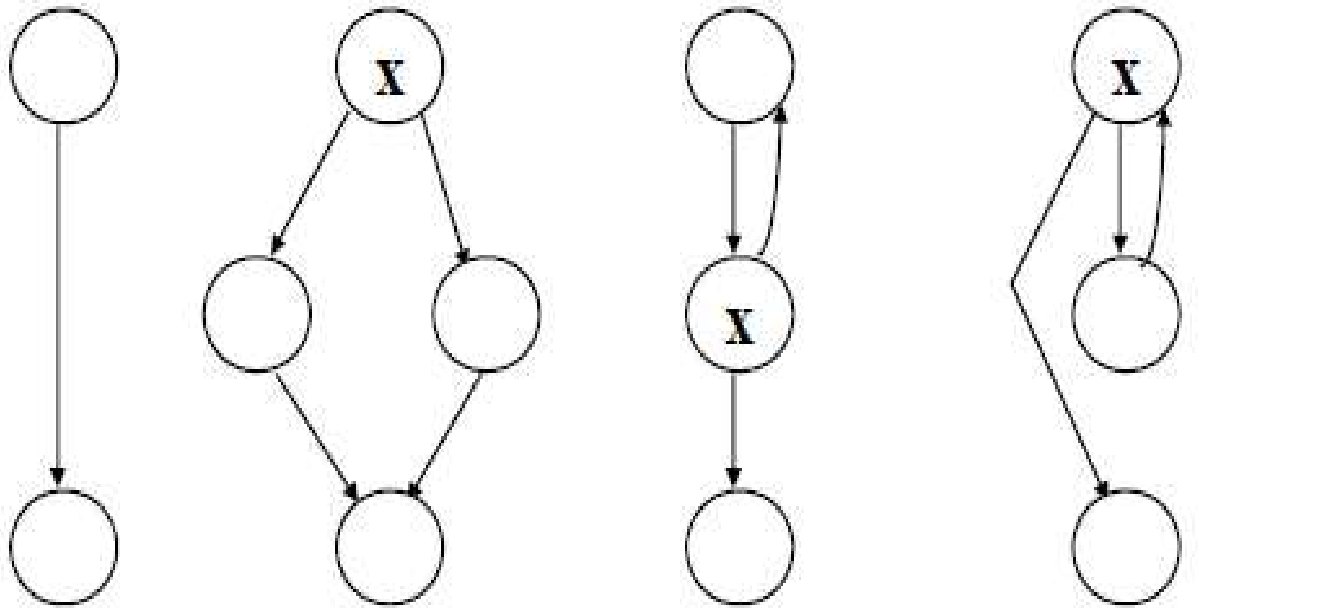


Pruebas de caja blanca

- Prueba de camino básico
  - Notación de grafo de flujo
  - Complejidad ciclomática
  - Obtención de casos de prueba
  - Matrices de grafos
- Prueba de la estructura de control
  - Prueba de condición
  - Prueba de flujo de datos
  - Prueba de bucles

- **Grafos de Flujo**
  - Utilizados para la representación del flujo de control
  - La estructura de control sirve de base para obtener los casos de prueba

*El diseño de casos de prueba tiene que estar basado en la elección de **caminos** importantes que ofrezcan una seguridad aceptable de que se descubren defectos*



Secuencia **Si x entonces... (If x then...else...)** **Hacer... hasta x (Do...until x)** **Mientras x hacer... (While x do...)**

Prueba del camino básico

Es una técnica de prueba de caja blanca que nos permite obtener una medida de complejidad lógica.

Con la medida de complejidad se genera un conjunto básico de caminos que se ejecutan por lo menos una vez durante la ejecución del programa.

Prueba del camino básico

Para representar el flujo de control de un programa se usa un grafo de flujo.

Para determinar la complejidad lógica de un programa, se calcula la complejidad ciclomática que define el número de caminos independientes del conjunto básico.

Un *camino independiente* es cualquier camino del programa que incluye nuevas instrucciones de un proceso o una nueva condición.

Obtención de casos de prueba

1. **Dibujar el grafo correspondiente**
2. **Determinar la complejidad.**
3. **Determinar un conjunto básico de caminos linealmente independientes**
4. **Preparar casos de prueba que forzarán a la ejecución de cada camino básico.**

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.

Notación de grafo de flujo

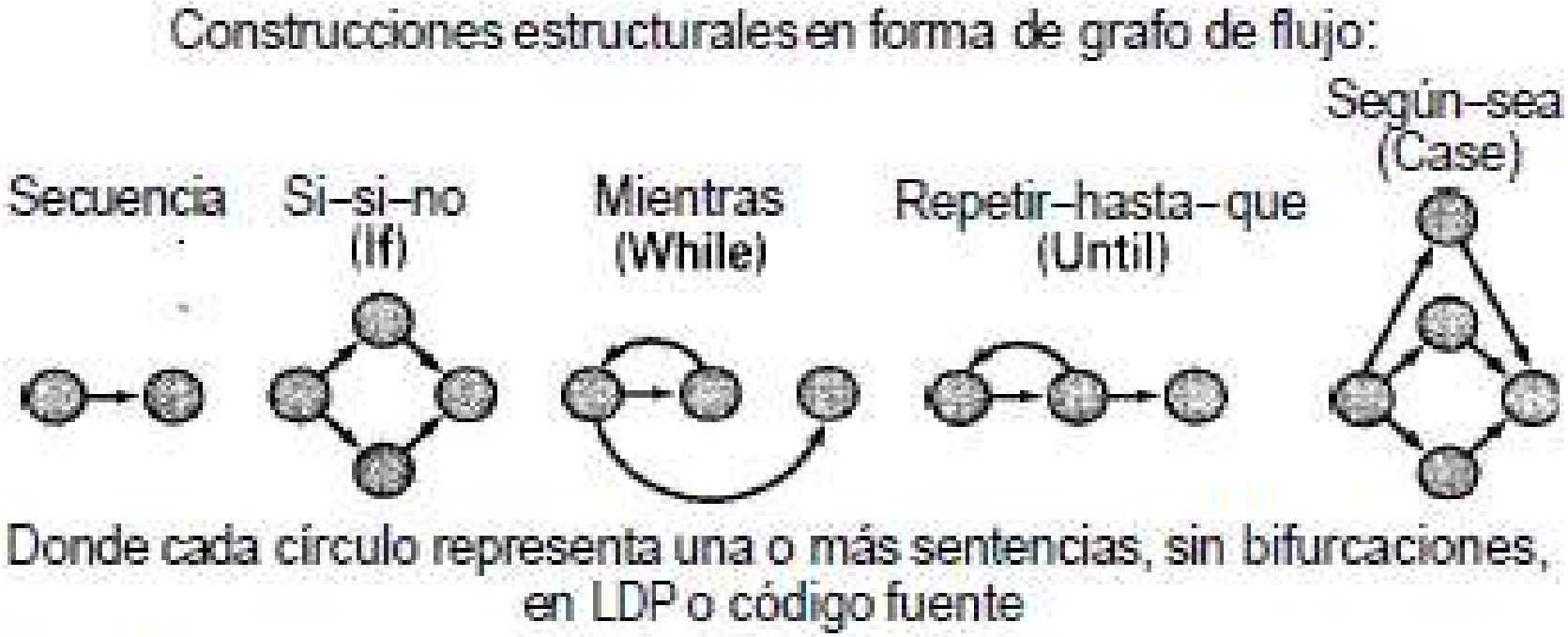
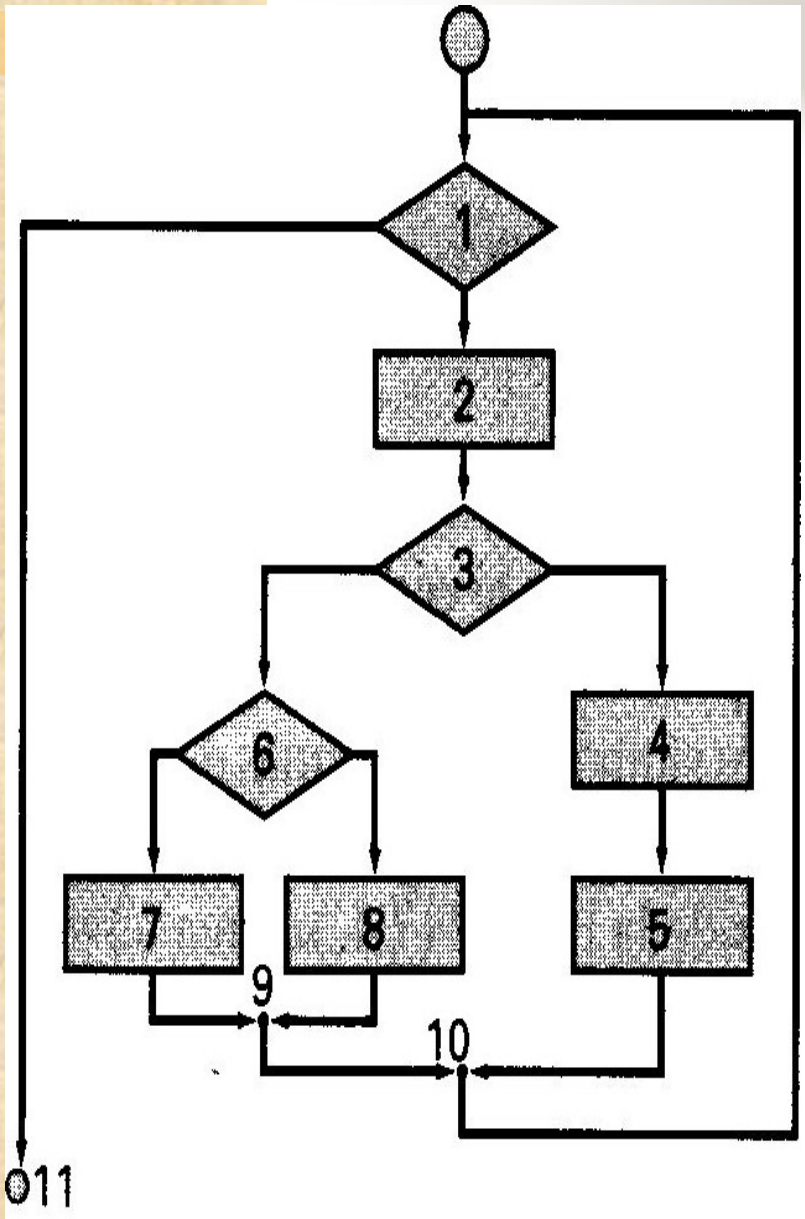
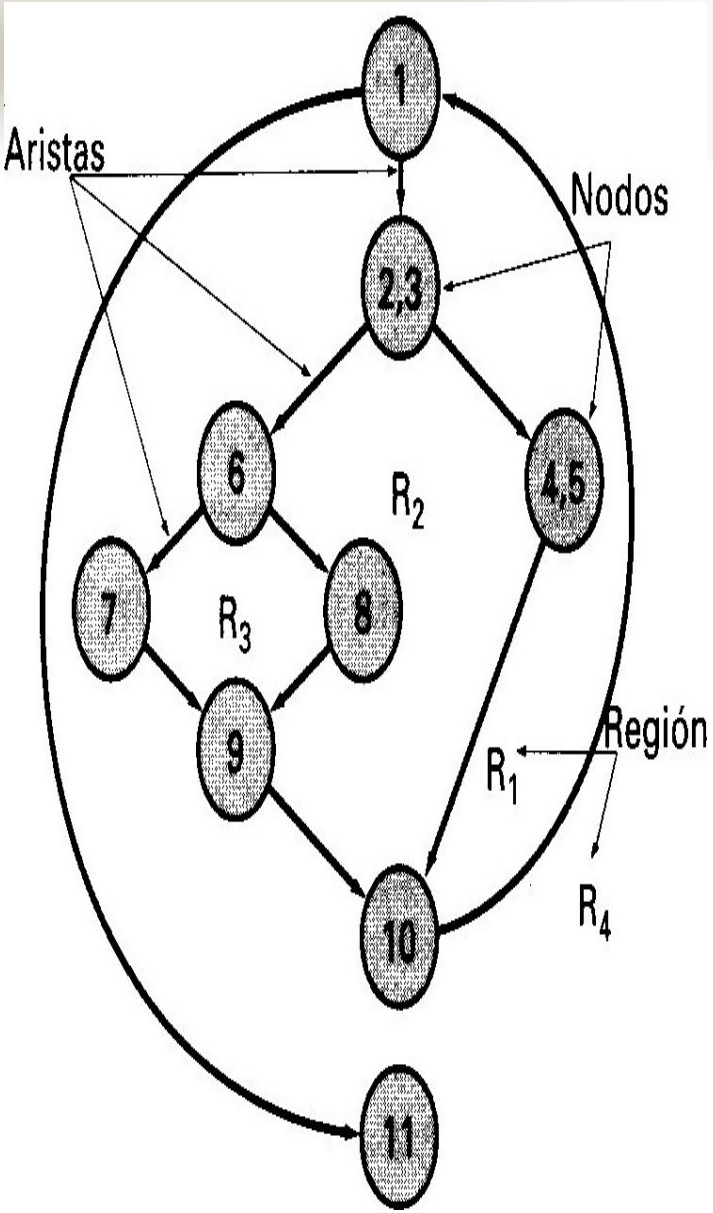


FIGURA 17.1. Notación de grafo de flujo.

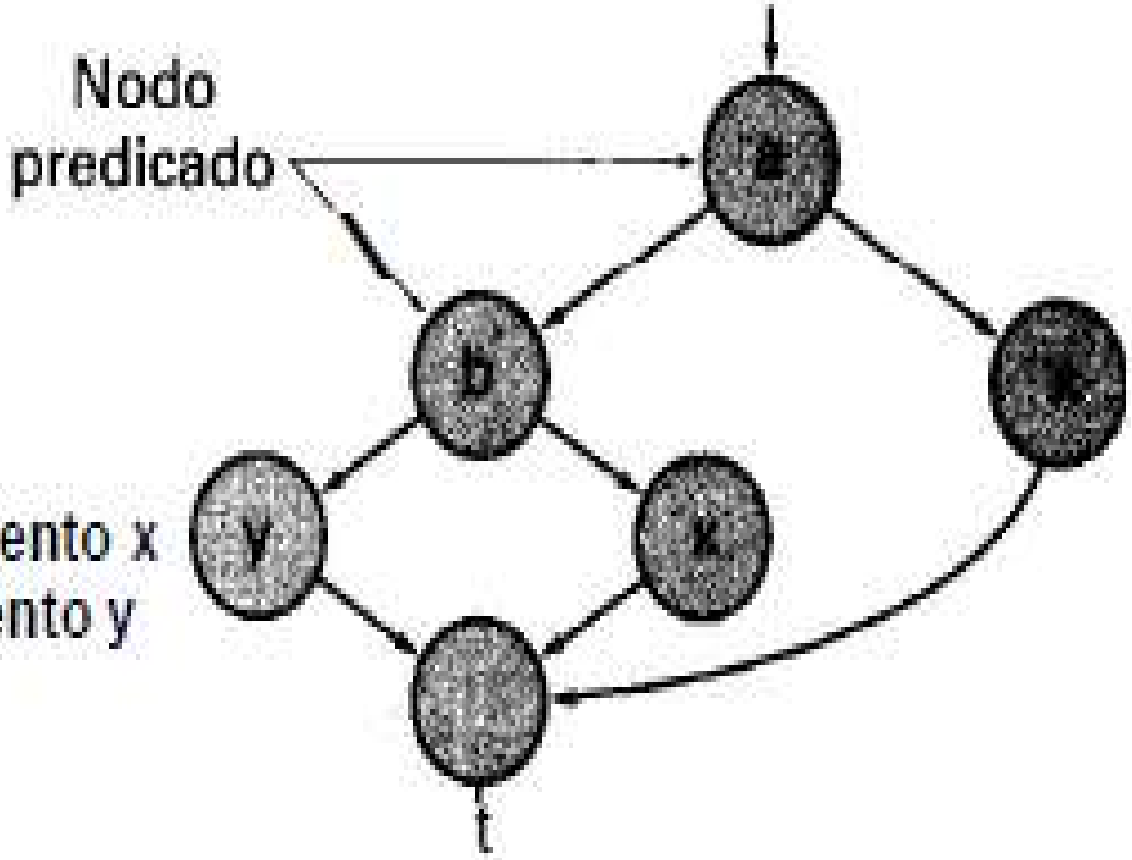
El grafo de flujo representa el flujo de control lógico. Cada construcción estructurada tiene su correspondiente símbolo en el grafo del flujo.


1
2
3
6
4
7
8
5
9
10
11


Aristas
Nodos
1
2,3
6
4,5
7
8
9
10
11
R1
R2
R3
R4
Región

Lógica compuesta

IF a OR b
  Then Drocedimiento x
  Else procedimiento y
ENDIF



Complejidad ciclomática

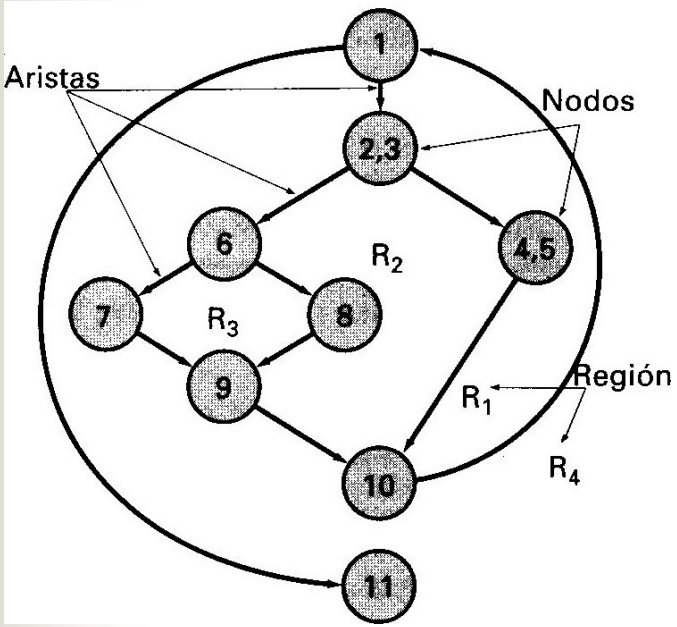
- La *complejidad ciclomática es una métrica del software* que proporciona una medición cuantitativa de la complejidad lógica de un programa.

- 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.

Complejidad ciclomática

- 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.



caminos independientes (1,2,3 **y 4) componen** un *conjunto básico para el grafo de flujo*

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.

Complejidad ciclomática

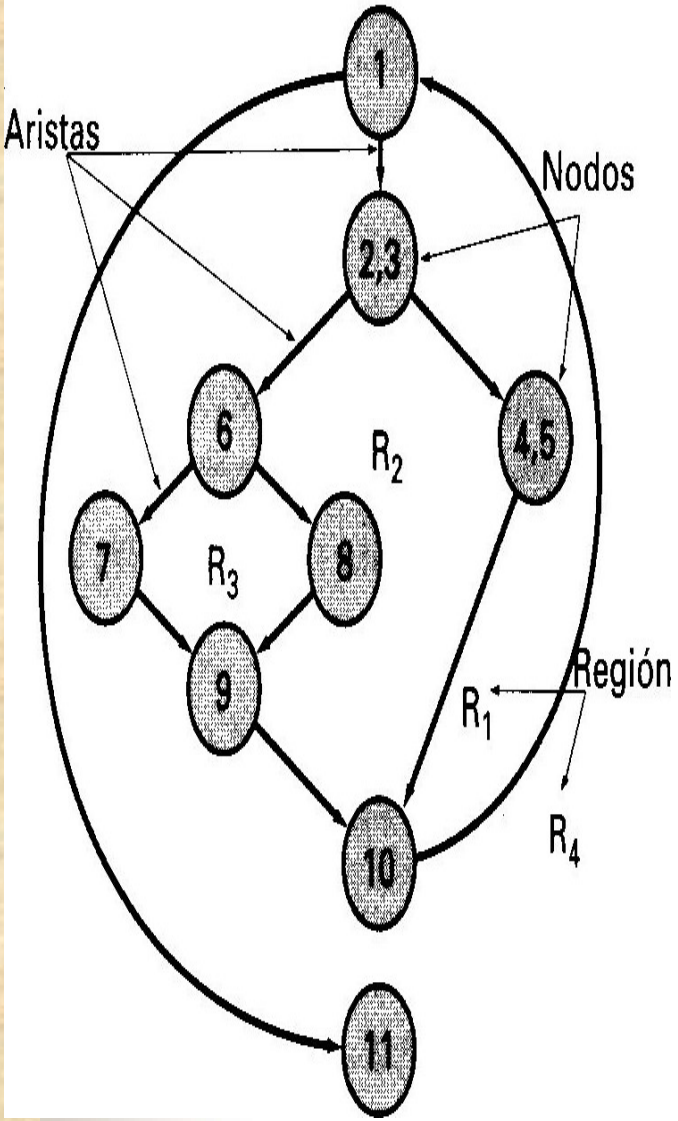
¿Cómo sabemos cuántos caminos hemos de buscar?

- 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:

  - El número de regiones del grafo de flujo coincide con la complejidad ciclomatica.
  - 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.***
  - 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.*

Complejidad ciclomática



La complejidad ciclomática se puede calcular mediante cualquiera de los anteriores algoritmos:

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

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.

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. Usaremos el procedimiento **media, representado en LDP**, 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.**

```
PROCEDURE media;
    * Este procedimiento calcula la media de 100 o menos
      números que se encuentren entre unos límites;
      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;
      suma = 0;
      DOWHILE VALOR [i] <> -999 and total entrada < 100   3
4        Incrementar total entrada en 1;
         IF valor [i] >= mínimo  AND  valor [i] <= máximo   6
5           THEN incrementar total.válido en 1;
7               suma = suma + valor [i];
            ELSE ignorar
8        ENDIF
         Incrementar i en 1;
9     ENDDO
      IF total válido > 0   10
         THEN media = suma/total válido,
12       ELSE media = -999;
13    ENDIF
END media
```

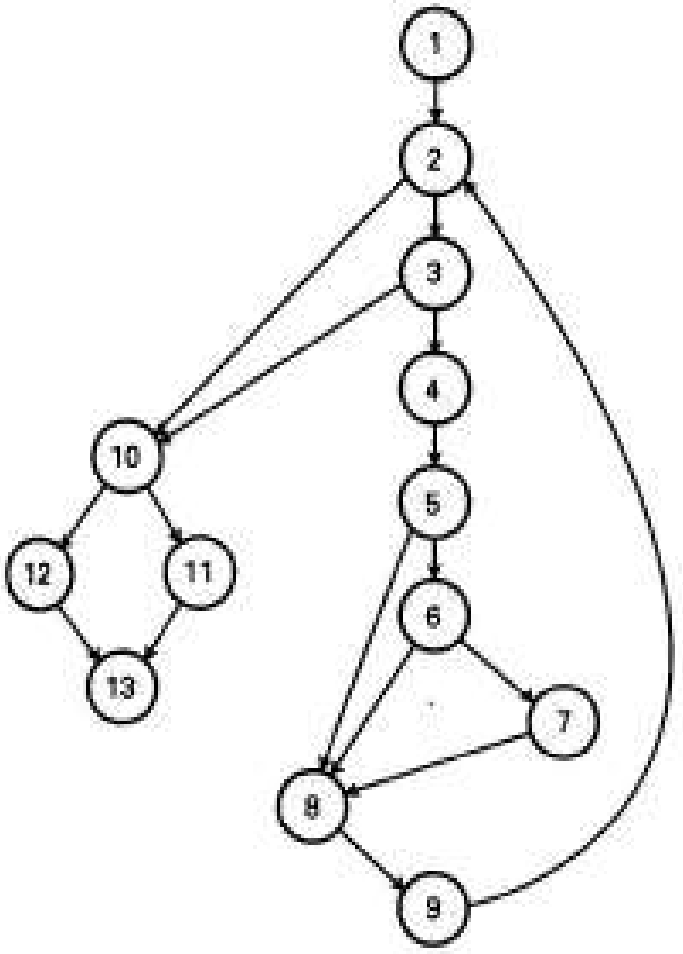
Determinamos la complejidad ciclomática del grafo de flujo resultante.

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

      INTERFACE 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;
      suma =0;
      DOWHILE VALOR [i] <> -999 and total entrada < 100   3
4       Incrementar total entrada en 1;
        IF valor [i] >= mínimo  AND  valor [i] <= máximo   6
5         THEN incrementar total.válido en 1;
7             suma = suma +valor [i];
          ELSE ignorar
8       ENDIF
        Incrementar i en 1;
9     ENDDO
      IF total válido > 0   10
          THEN media =suma/total válido,
12        ELSE media = -999;
13    ENDIF
END media
```

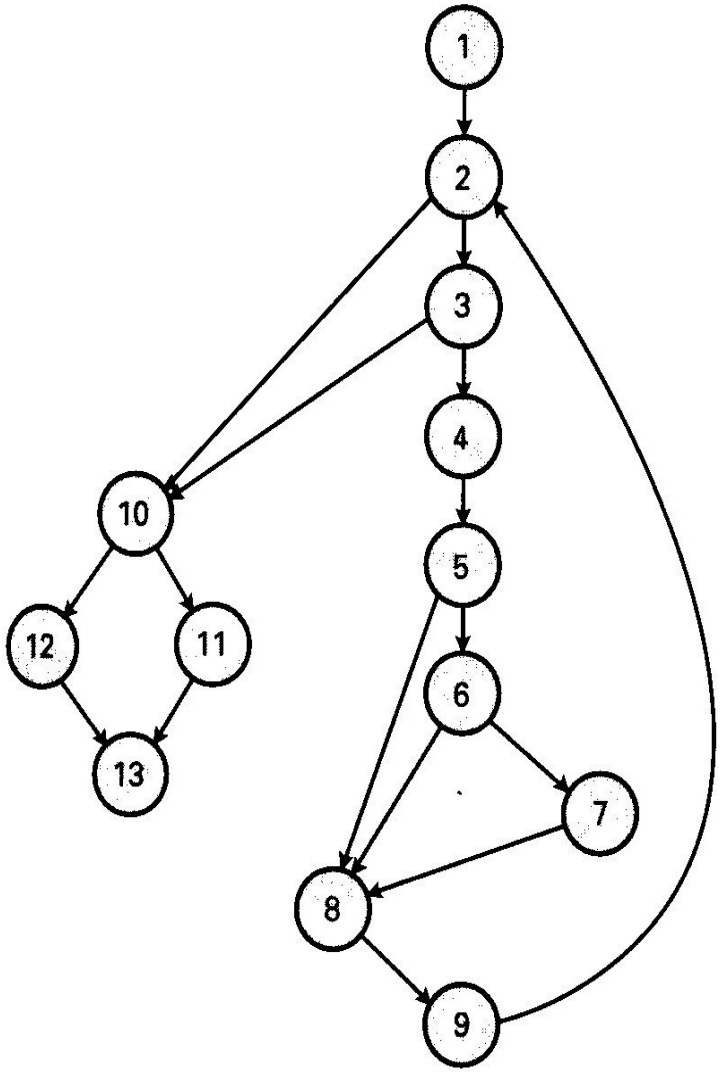
V(G) = 6 regiones

V(G) = 17 aristas – 13 nodos + 2 = 6

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



Determinamos un conjunto básico de caminos linealmente independientes.



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..

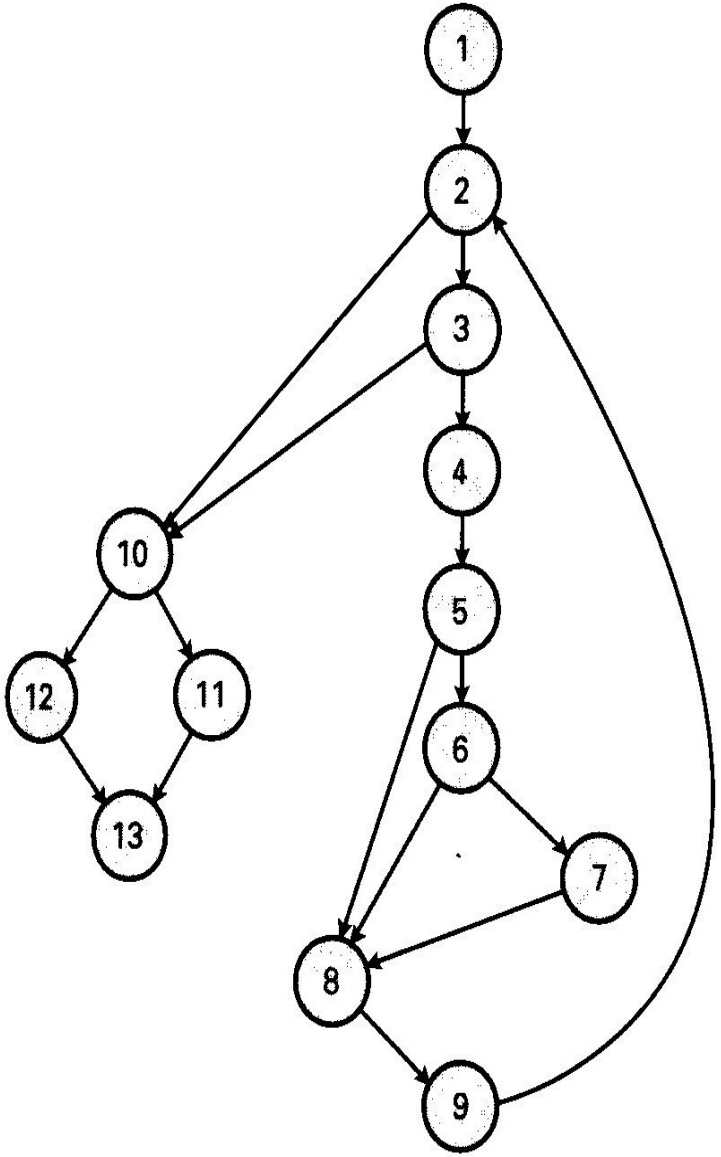
Normalmente merece la pena identificar los nodos predicado para que sea más fácil obtener los casos de prueba. En este caso, los nodos 2, 3, *5, 6 y 10 son* nodos predicado.

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:

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



Caso de prueba del camino 1:

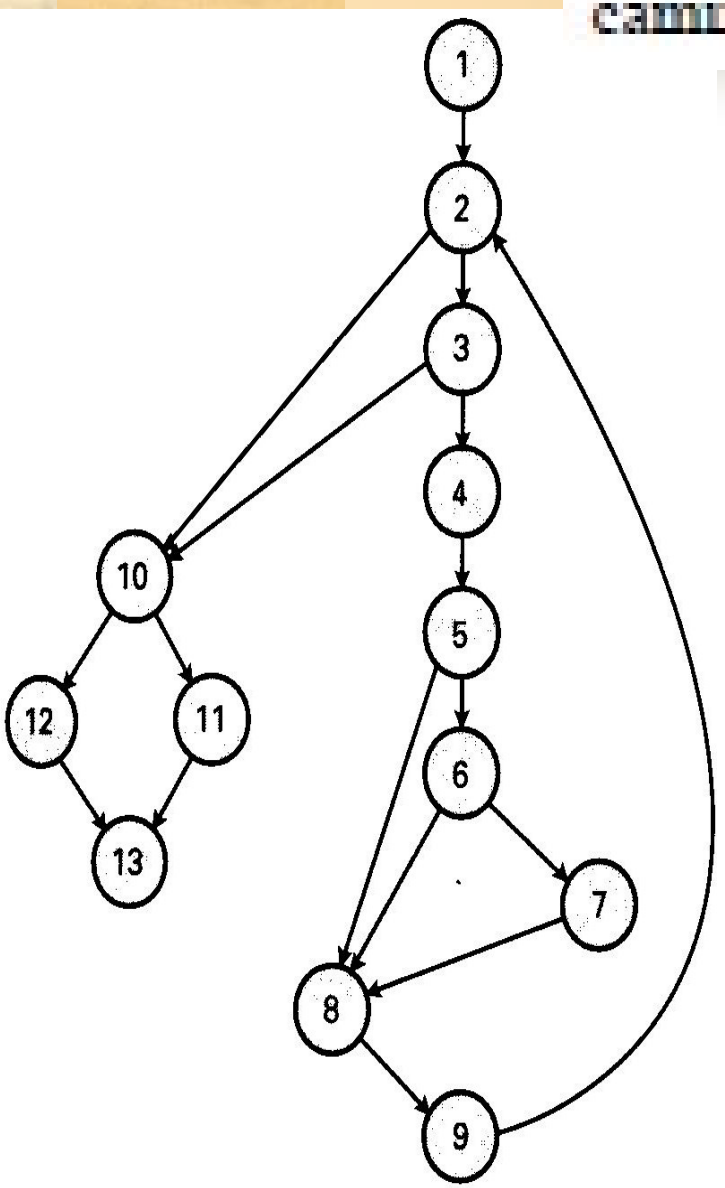
valor (*k*) = entrada válida, donde *k* < i definida a continuación

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.

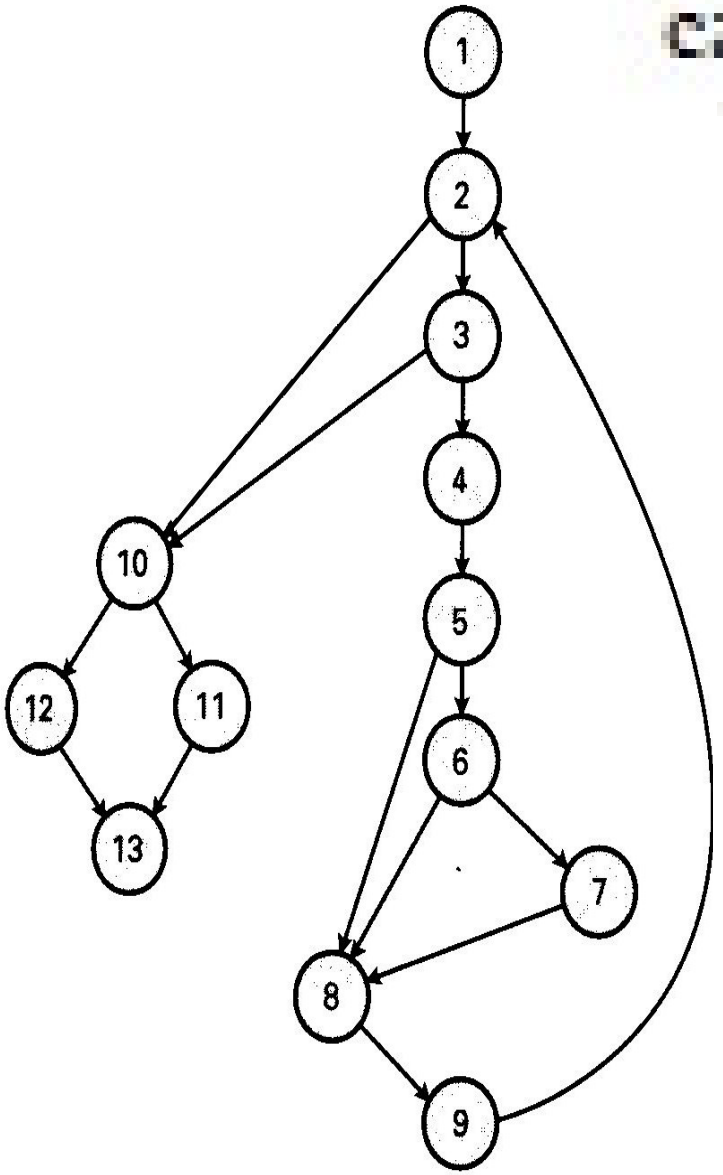
camino 2: 1-2-10-12-13



Caso de prueba del camino 2:

valor (1) = -999

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



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

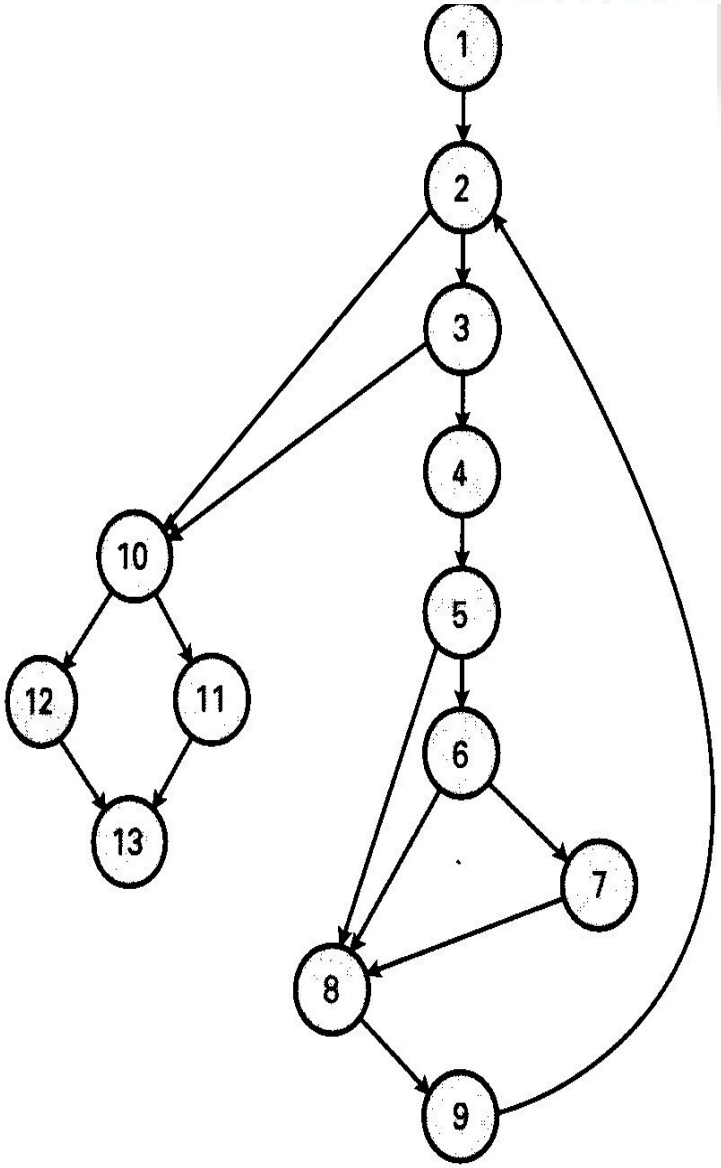
Caso de prueba del camino 3:

intento de procesar 101 o más valores

los primeros 100 valores deben ser válidos

resultados esperados: igual que en el caso de prueba 1

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



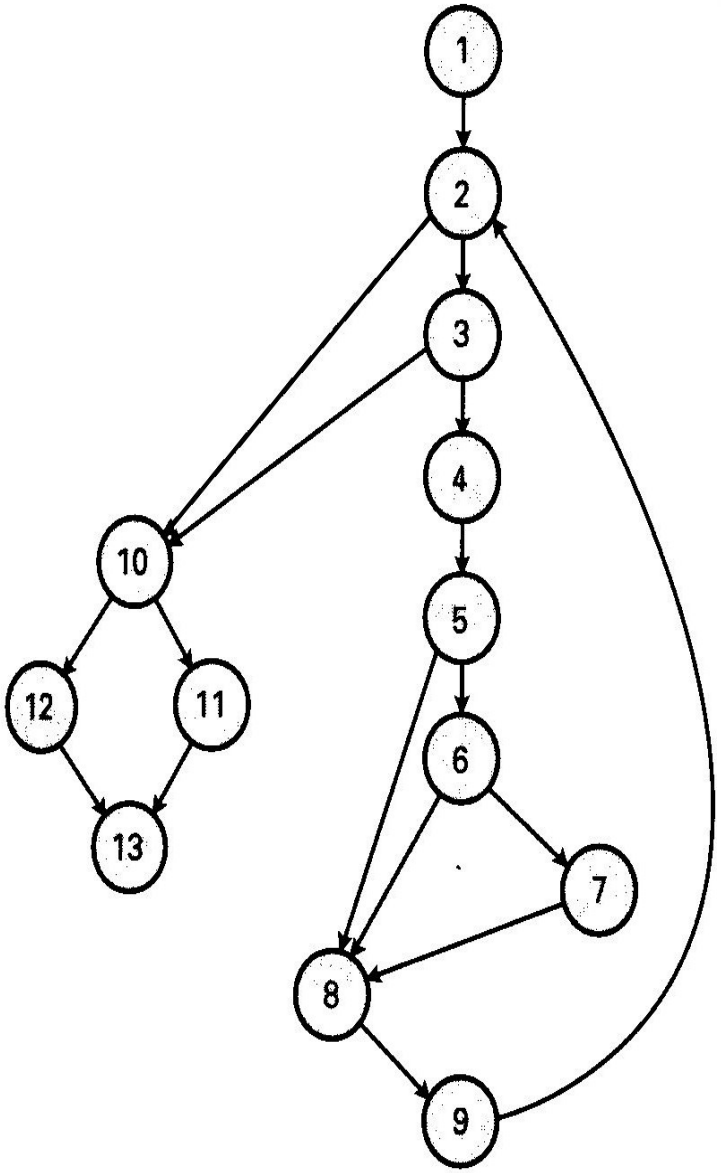
Caso de prueba del camino 4:

valor (i) = entrada válida donde i < 100

valor (k) < mínimo, para k < i

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

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



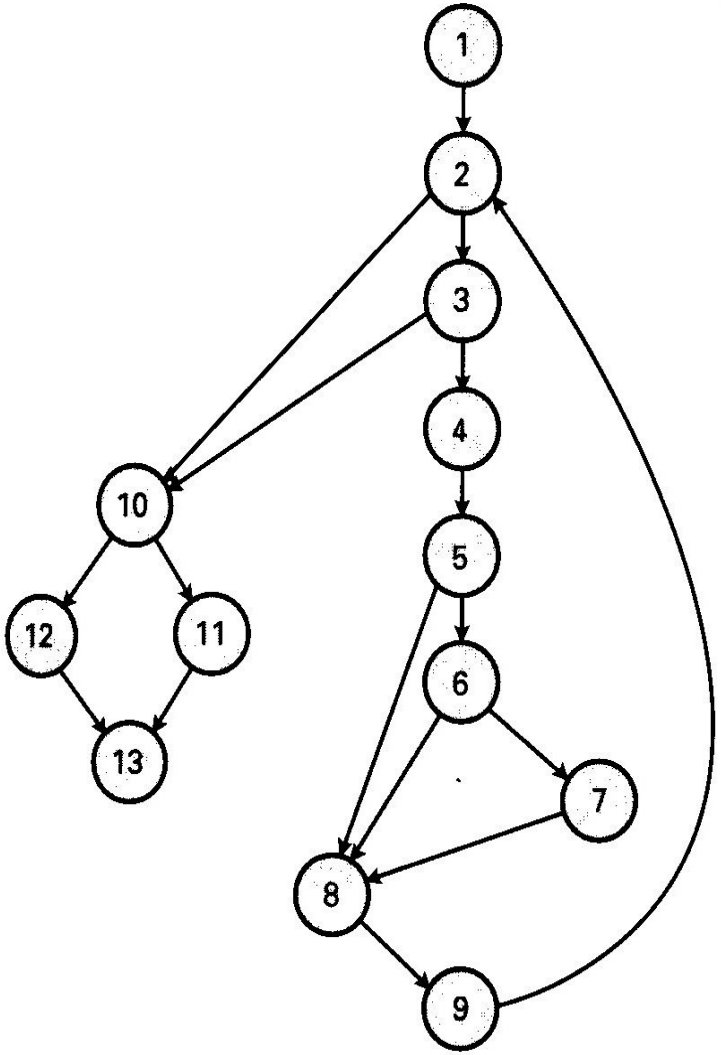
Caso de prueba del camino 5:

valor (i) = entrada válida donde i < 100

valor (k) > máximo, para k <= i

resultados esperados: media correcta sobre los i valores y totales adecuados

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



Caso de prueba del camino 6:

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

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

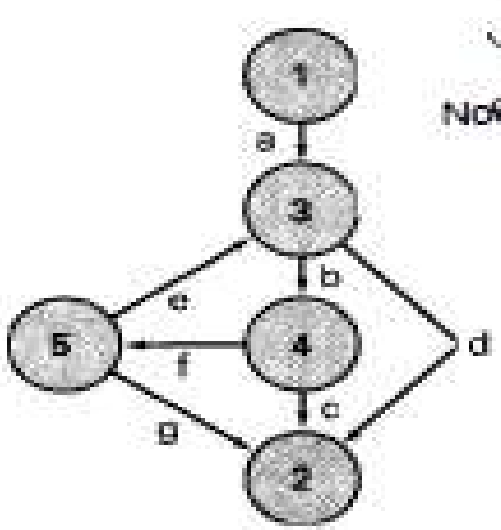
- Ejecutamos cada caso de prueba y comparamos los resultados obtenidos con los esperados.

- 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.

Matrices de grafos

- 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.



Grafo de flujo

Nodo \ Conectado al nodo	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Matriz del grafo

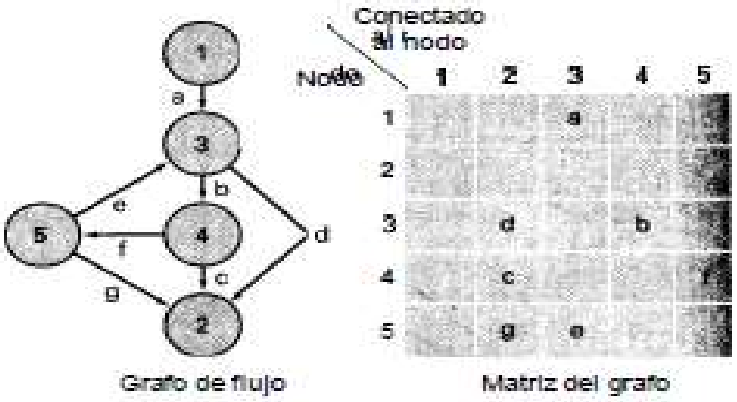
Matrices de grafos

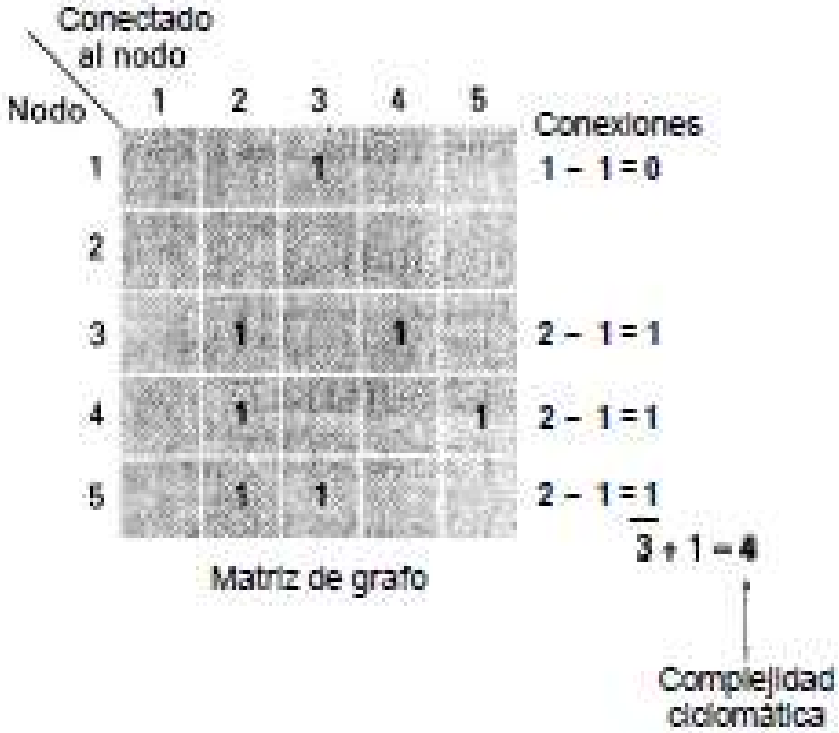
- Hasta aquí, la matriz de grafo no es nada más que una representación tabular del grafo de flujo. Sin embargo añadiendose un ***peso de enlace*** a cada entrada de la matriz, la matriz de grafo se puede convertir en una potente herramienta para la evaluación de la estructura de control del programa 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 O (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

Matrices de grafos





Pruebas de la estructura de control

Existen tres pruebas de estructura de condición, que son:

1. **Prueba de condición,**
2. **Prueba de estructura de datos y**
3. **Pruebas de estructuras de control (bucles).**

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 .

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

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

Si una condición es incorrecta, entonces es incorrecta al menos un componente de la condición.

Prueba de condición

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 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.

Prueba de condición

- 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*.
- 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

- Criterios de Cobertura:

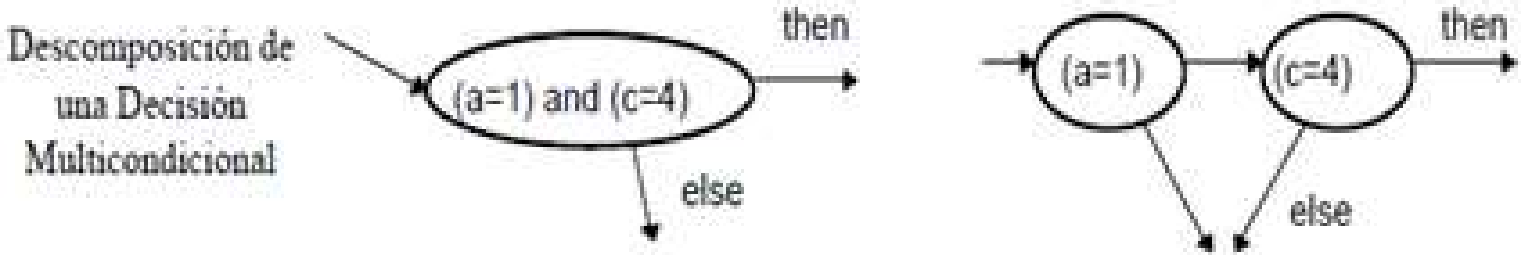
☑ **Cobertura de sentencias**. Que cada sentencia se ejecute al menos una vez.

☑ **Cobertura de decisiones**. Que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso.

☑ **Cobertura de condiciones**. Que cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez.

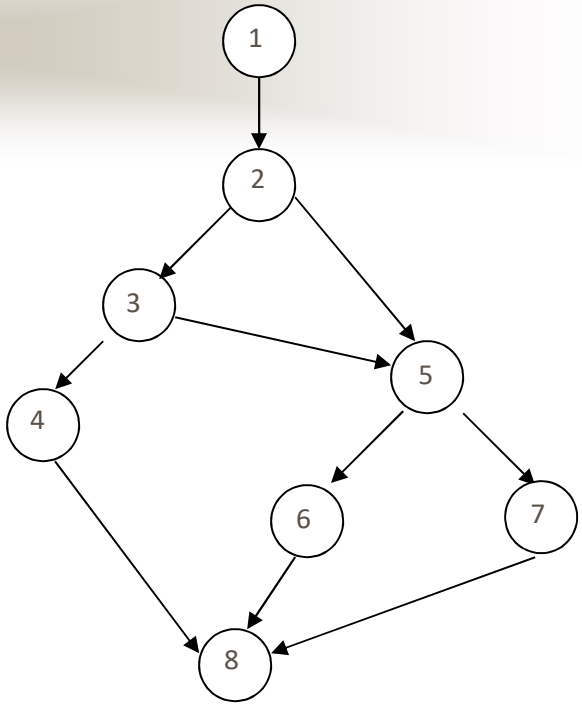
☑ **Criterio de decisión/condición**. Que se cumplan a la vez el criterio de condiciones y el de decisiones.

☑ **Criterio de condición múltiple**. La evaluación de las condiciones de cada decisión no se realiza de forma simultánea.



Cobertura de sentencias

- Camino 1 ➔ 1 - 2 – 3 – 4 – 8
- Camino 2 ➔ 1 – 2 – 3 – 5 – 6 –8
- Camino 3 ➔ 1 – 2 – 5 – 6 – 8
- Camino 4 ➔ 1 – 2 – 5 – 7 - 8



- Se trata de ejecutar con los casos de prueba cada sentencia e instrucción al menos una vez.
- En este caso con ejecutar los caminos 1, 2 y 4 nos vale:

Camino	Características	Caso de Prueba		
		x	y	z
Camino 1	x > y , x > z	10	3	3
Camino 2	y < x < z	5	2	10
Camino 4	x < y , z < y	5	10	5

Cobertura de decisiones

- Escribimos los casos suficientes para que cada condición tenga al menos una resultado verdadero y otro falso. Utilizando los mismos caminos y casos de prueba que en la cobertura de sentencias cubriremos también en este caso la cobertura de decisiones

Camino	Características	Caso de Prueba		
		x	y	z
Camino 1	x > y , x > z	10	3	3
Camino 2	y < x < z	5	2	10
Camino 4	x < y , z < y	5	10	5

Cobertura de condiciones

- Se trata de escribir los casos suficientes para que cada condición de cada decisión adopte el valor verdadero y el falso al menos una vez. Los casos de prueba en este caso serán los mismos que en la cobertura de decisiones

Cobertura de decisión /condición

- Es el cumplimiento de la cobertura de condiciones y de decisiones. Elegiremos los caminos 1,2 y 4, con los casos de prueba vistos anteriormente.

- Camino 1 → 1 - 2 – 3 – 4 – 8
- Camino 2 → 1 – 2 – 3 – 5 – 6 –8
- Camino 3 → 1 – 2 – 5 – 6 – 8
- Camino 4 → 1 – 2 – 5 – 7 - 8

Criterio de condición múltiple

- Si tenemos decisiones multicondicionales las descompondremos en decisiones unicondicionales, ejecutando todas las combinaciones posibles de resultados.
- Tenemos la decisión multicondicional x>y && x>z y la decisión unicondicional z>y. Combinándolas nos damos cuenta que para que se cumpla el criterio de condición múltiple debemos ejecutar los cuatro caminos independientes:

Camino	Características	Caso de Prueba		
		x	y	z
Camino 1	x > y , x > z	10	3	3
Camino 2	y < x < z	5	2	10
Camino 3	x < y < z	2	5	8
Camino 4	x < y , z < y	5	10	5

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.
- 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 I la sentencia S contiene una definición de X }
USO(S) = { X I la sentencia S contiene un uso de X)

- Si la sentencia S es una sentencia i*f 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*
- 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' ..

Prueba del flujo de datos

- Una sencilla estrategia de prueba de flujo de datos se basa en requerir que se cubra al menos una vez cada cadena DU.
- 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*.

En resumen …

Método *dataflow*:

- Localizar qué puntos en el programa dan valor a una variable y cuáles la consumen
- Para cada variable x definir conjuntos de instrucciones Def(x) (donde se asigna) y Use(x) (donde se lee)
- Para cada instrucción s, definir Def(s) y Use(s) a partir de los Def(x) y Use(x)
- Definir cadenas DU (Def-Use):

  (x,s1,s2) si x está en Def(s1) y en Use(s2) y la definición de x en s1 aún es válida en s2
- Necesario cubrir al menos una vez cada cadena DU

Prueba de bucles

- Los bucles son la piedra angular .
- 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:
  - ***bucles simples;***
  - ***bucles concatenados;***
  - ***bucles anidados;***
  - ***bucles no estructurados.***

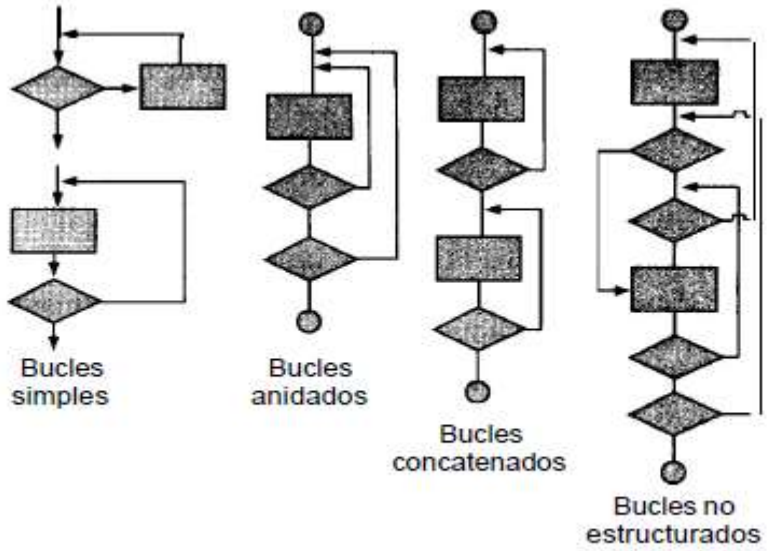


FIGURA 17.8. Clases de bucles.

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*

Bucles anidados

- Si extendiéramos el enfoque de prueba de los bucles simples a los bucles anidados, el número de posibles pruebas aumentaria geométricamente a medida que aumenta el nivel de anidamiento. Esto llevaria a un número impracticable de pruebas. Beizer 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.

Bucles no estructurados

- Siempre que sea posible, esta clase de bucles se deben *rediseñar para que* se ajusten a las construcciones de programación estructurada.

Estrategias de prueba del software

- **Estrategias de prueba del software:**
  - **Prueba de unidad,**
  - **Prueba de integración,**
  - **Prueba de validación,**
  - **Prueba del sistema.**

Estrategias de prueba del software

- **Proporcionan un plano o guía para el desarrollador del software, para la organización de control de calidad y para el cliente .**
- **Es una guía que describe los pasos a llevar a cabo como parte de la prueba, cuándo se deben planificar y realizar esos pasos, y cuánto esfuerzo, tiempo y recursos se van a requerir.**

  Por lo tanto, cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de los casos de prueba, la ejecución de las pruebas y la agrupación y evaluación de los datos resultantes.

Etapas de un plan de pruebas

a. especificar los objetivos de las pruebas.

b. determinar con precisión los criterios a seguir en su realización.

c. Integrar al personal y los elementos necesarios para el desarrollo de las pruebas.

d. Aplicación de la prueba o pruebas según los criterios seleccionados.

e. evaluación de los resultados y consideraciones para llevar a cabo una nueva serie de pruebas.

Un enfoque estratégico para la prueba del software

Generalmente se proporciona una *plantilla* para la prueba con las siguientes características generales:

- **La prueba comienza en el nivel de módulo y trabaja "hacia fuera", hacia la integración de todo el sistema basado en computadora. Se usa el enfoque “bottom-up”.**
- **En diferentes momentos se utilizarán diferentes técnicas de prueba**
- **La prueba la lleva a cabo el que desarrolla el software y (para grandes proyectos) un grupo de prueba independiente.**
- **La prueba y la depuración son actividades diferentes, pero la depuración se puede incluir en cualquier estrategia de prueba.**

Verificación y validación

La verificación se refiere al conjunto de actividades que aseguran que el software implementa correctamente una función específica.

La validación se refiere a un conjunto diferente de actividades que aseguran que el software construído se ajusta a los requisitos del cliente.

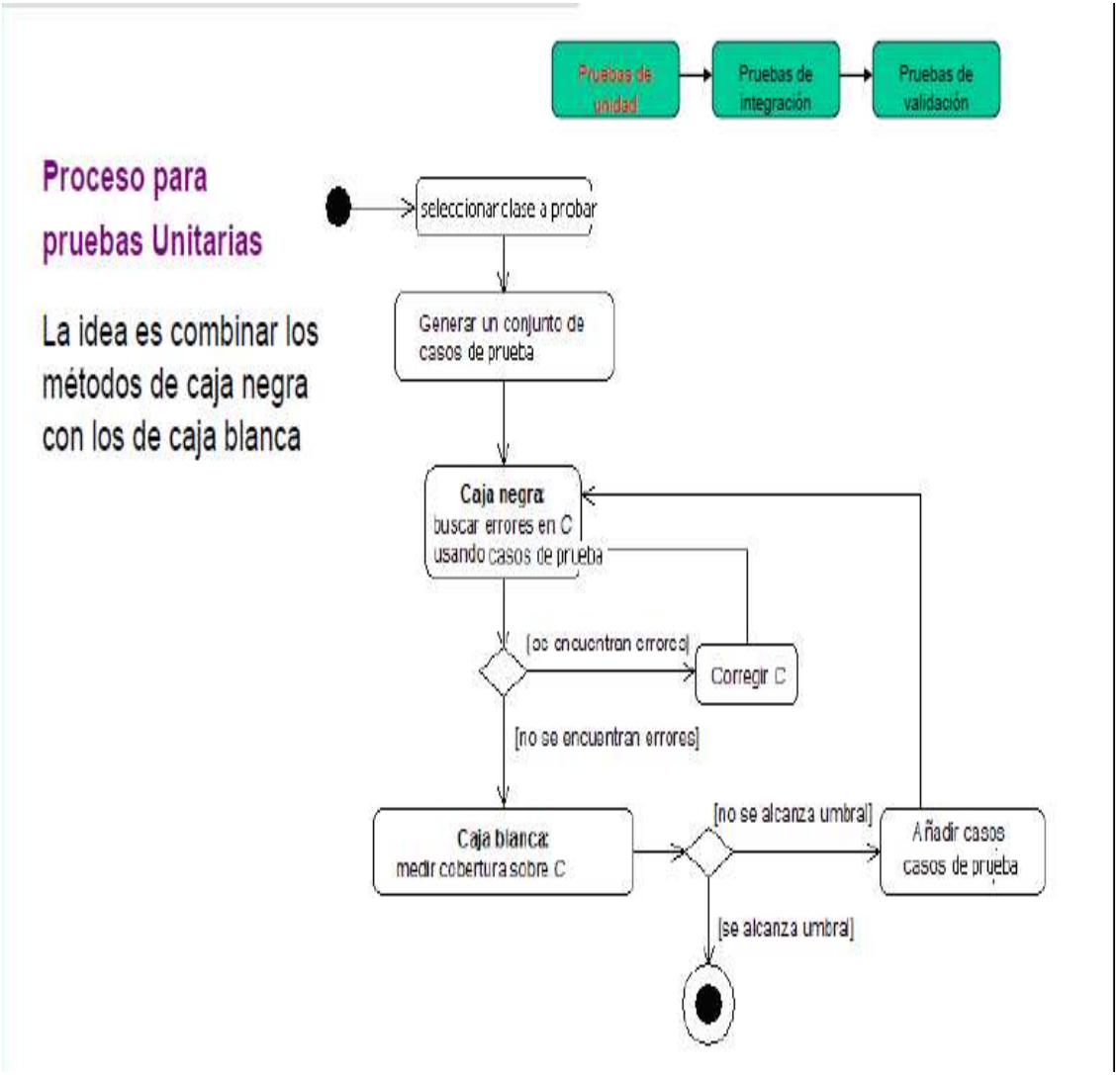
La verificación y la validación comprenden un amplio rango de actividades de SQA que incluyen :

- **Revisiones técnicas formales,**
- **Auditorias de configuración y calidad,**
- **Supervisión del rendimiento,**
- **Revisión de la base de datos,**
- **Análisis de los algoritmos,**
- **Prueba de desarrollo,**
- **Prueba de calificación,**
- **Prueba de instalación.**

Prueba de unidad

La prueba de unidad

- Centra el proceso de verificación en la menor unidad del diseño del software - el módulo.
- Usando la descripción del diseño detallado como guía, se prueban caminos de control importantes, con el fin de descubrir errores dentro del ámbito del módulo.
- Puede llevarse a cabo en paralelo para múltiples módulos.



Consideraciones sobre la prueba de unidad

Las pruebas que se dan como parte de la prueba de unidad son:

- **Se prueba la *interfaz* del módulo .**
- **Se examinan las *estructuras de datos* locales.**
- **Se prueban las *condiciones límites* .**
- **Se ejercitan todos los *caminos independientes* de la estructura de control.**
- **Y finalmente, se prueban todos los *caminos de manejo de errores.***

Antes de iniciar cualquier otra prueba es preciso probar el flujo de datos de la interfaz del módulo.

Prueba de integración

Es una técnica sistemática para construir la estructura del programa mientras que, al mismo tiempo, se llevan a cabo pruebas para detectar errores asociados con la interacción.

El objetivo es tomar los módulos probados en unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.

Prueba de integración

- **A continuación se deben ensamblar o integrar los módulos para formar el paquete del software completo.**
- **La prueba de integración se dirige a todos los aspectos asociados con el doble problema de verificación y de construcción del programa.**
- **Las técnicas que más prevalecen son las de diseño de casos de prueba de caja negra**

Documentación de la prueba de integración

- ***La especificación de prueba incluye*** **un plan general para la integración del software y una descripción de las pruebas específicas. Es un resultado del proceso de ingeniería del software y forma parte de la configuración del software.**
- **El *alcance de la prueba* resume las características funcionales, de rendimiento y de diseño interno específicas que van a a ser probadas. Se limita el esfuerzo de prueba, se describen los criterios de terminación de cada fase de prueba y se documentan las limitaciones del plan.**

Documentación de la prueba de integración

- **El *plan de prueba* describe la estrategia general para la integración. La prueba se divide en *fases* y *subfases* dirigidas a específicas características funcionales y del ámbito de información del software.**

En todas las fases de prueba se siguen los siguientes criterios con sus correspondientes pruebas:

- **Integridad de interfaz,**
- **Validez funcional,**
- **Contenido de la información,**
- **Rendimiento.**

Pruebas de Integración

- Involucran varios módulos
- Pueden ser estructurales o funcionales
  - Estructurales: como las de caja blanca, pero analizando llamadas entre módulos
  - Funcionales: como las de caja negra, pero comprobando funcionalidades conjuntas
  - Se siguen utilizando clases de equivalencia **y** análisis de valores frontera
- Las **pruebas finales** consideran todo el sistema, cubriendo plenamente la especificación de requisitos del usuario

Prueba de validación

- **Tras la culminación de la prueba de la integración, el software está completamente ensamblado como un paquete, se han encontrado y corregido los errores de interfaz y puede comenzar una serie final de pruebas del software - La *prueba de validación.***

Criterios para la prueba de validación

La validación del software se consigue mediante una serie de pruebas de la caja negra que demuestran la conformidad con los requisitos.

Un plan de prueba traza las pruebas que se han de llevar a cabo y un procedimiento de prueba define los casos de prueba específicos que se usarán para demostrar la conformidad con los requisitos.

Pruebas de Aceptación

- Pruebas funcionales que realiza el **cliente** antes de poner la aplicación en producción
- Hay errores que sólo el cliente puede detectar...
  - "**el cliente siempre tiene razón**"
- Técnicas:
  - Pruebas alfa: Se invita al cliente al entorno de desarrollo, trabajando sobre un entorno controlado
  - Pruebas beta: Se desarrollan en el entorno del cliente, que se queda sólo con el producto en un entorno sin controlar
  - Ambas pruebas son habituales cuando se va a distribuir el programa a **muchos clientes**

Otros tipos de pruebas

- Solidez *(robustness testing)*
  - ¿Cómo reacciona el sistema ante datos de entrada erróneos?
  - Por ejemplo, probar:
    - Todas las órdenes correctas
    - Órdenes con errores de sintaxis
    - Órdenes correctas, pero fuera de lugar
    - La orden nula
    - Órdenes con datos de más
    - Provocar una interrupción después de introducir una orden
    - Órdenes con delimitadores inapropiados
    - Órdenes con delimitadores incongruentes ( ]

Prueba del sistema

La prueba del sistema, realmente está constituida por una serie de pruebas diferentes cuyo propósito principal es ejercitar profundamente el sistema basado en computadora.

Aunque cada prueba tiene un propósito distinto, todas trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas.

Otros tipos de pruebas

- Recorridos *(walkthroughs)*
  - Se reúne a desarrolladores y críticos: los críticos se leen el código línea a línea y piden explicaciones a los desarrolladores
  - Eficaz para errores de naturaleza local
  - Pésima para localizar fallos en interacciones entre partes alejadas
- Aleatorias *(random testing)*
  - Se basa en que la probabilidad de descubrir un error es similar si se eligen pruebas al azar que si se utilizan criterios de cobertura
  - Razonable empezar las pruebas probando al azar
  - Pero es insuficiente en programas críticos

Prueba de recuperación

La *prueba de recuperación* es una prueba del sistema que fuerza el fallo del software de muchas formas y verifica que la recuperación se lleva a cabo apropiadamente.

Si la recuperación es automática hay que evaluar la correción de reinicialización, de los mecanismos de recuperación del estado del sistema, de la recuperación de datos y del rearranque.

Si la recuperación requiere la intervención humana, hay que evaluar los tiempos medios de reparación para determinar si están dentro de unos límites aceptables.

Prueba de seguridad

La *prueba de seguridad* intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán.

Prueba de resistencia o de carga maxima

Las *pruebas de resistencia* están diseñadas para enfrentar a los programas con situaciones anormales. En esencia, el sujeto que realiza la prueba de resistencia se pregunta: "¿A qué potencia puedo ponerlo a funcionar antes de que falle ?"

Prueba de rendimiento

La *prueba de rendimiento* está diseñada para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado. La prueba de rendimiento se da durante todos los pasos del proceso de prueba. Incluso al nivel de unidad, se debe asegurar el rendimiento de los módulos individuales a medida que se llevan a cabo las pruebas de la caja blanca.

El arte de la depuración

La *depuración* aparece como una consecuencia de una prueba efectiva o sea, cuando un caso de prueba descubre un error, la depuración es el proceso que resulta en la eliminación del error.

Aunque la depuración puede y debe ser un proceso ordendo, sigue teniendo mucho de arte.

El arte de la depuración

El proceso de depuración siempre tiene uno de dos resultados:

- **(1) se encuentra la causa, se corrige y se elimina; o**
- **(2) no se encuentra la causa.**

En este último caso, la persona que realiza la depuración debe sospechar la causa, diseñar un caso de prueba que ayude a validar sus sospechas y el trabajo vuelve hacia atrás a la corrección del error de una forma iterativa.

Bibliografia

- Análisis y diseño de sistemas de información (James A. Senn)
- Análisis y diseño de sistemas (Kendall&Kendall)
- Diseño de sistemas de informacion Teoria y Practica (John G. Burch)
- Piattini (2007). (Cap. 10)
- Sommerville (2005). (Capítulo 22 y 23)
- Jacobson, I., Booch, G., and Rumbaugh, J. (2000): El Proceso
- Unificado de Desarrollo. Addison-Wesley. (Capítulo 11)
- Pressman, R. (2005): Ingeniería del Software: Un Enfoque Práctico. 6º
- Edición. McGraw-Hill. (Capítulos 13 y 14)
- Pfleeger (2002). (Caps. 7, 8 y 9)
- IEEE Computer Society (2004). SWEBOK - Guide to the Software
- Engineering Body of Knowledge, 2004. (Capítulos 4 y 5)
- http://www.swebok.org/