

Calidade e probas en robótica

2022/23

Pruebas de software



Marcos Boullón Magán

Dept. Electrónica e Computación

Índice

- Introducción
- Pruebas estructurales
 - Criterios de cobertura lógica; Complejidad ciclomática de McCabe
- Pruebas funcionales
 - Partición o clases de equivalencia; Análisis de valores límite; Conjetura de errores; Pruebas aleatorias; Métodos de caja negra basados en grafos
- Enfoque práctico recomendado para el diseño de casos
- Documentación del diseño de pruebas
- Ejecución de las pruebas
- Estrategia de aplicación de las pruebas

Introducción

VERIFICACIÓN

El proceso de evaluación del sistema o de uno de sus componentes para determinar si los productos de una fase dada satisfacen las condiciones impuestas al principio de dicha fase. (IEEE)

- ¿Estamos construyendo correctamente el producto?



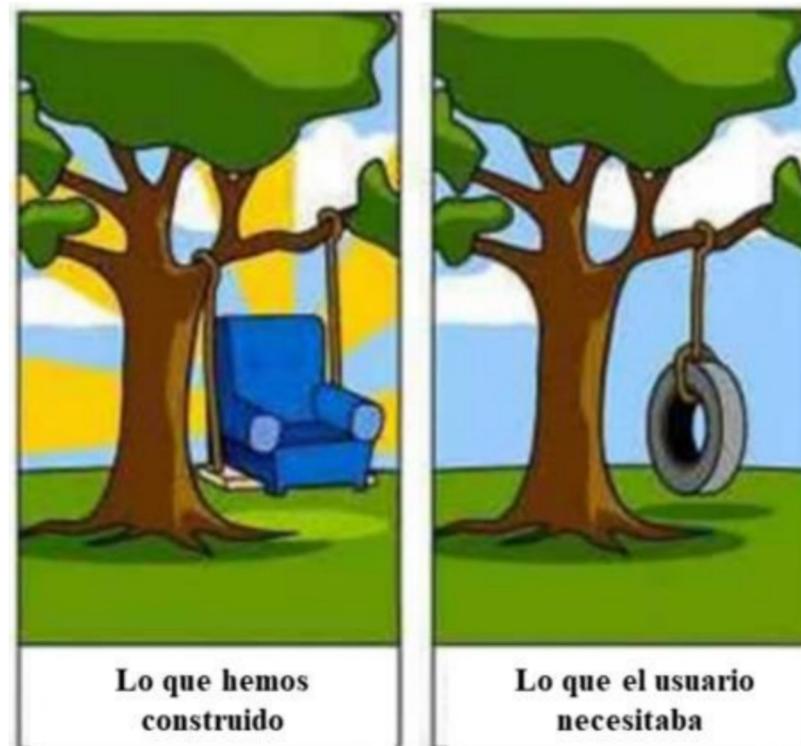
"The readings look good, but just in case, when was
the last time the system was checked for bugs?"

Introducción

VALIDACIÓN

El proceso de evaluación del sistema o de uno de sus componentes durante o al final del desarrollo para determinar si satisface los requisitos especificados. (IEEE)

- ¿Estamos construyendo el producto correcto?



Filosofía de las pruebas

- Es un proceso muy difícil
 - No es físico, no hay leyes de comportamiento, es complejo.
 - La prueba exhaustiva es imposible.
 - Prejuicios y poco tiempo.
- Cambio de mentalidad
 - Desarrollo de software (enfoque constructivo).
 - Pruebas para *demoler* lo construido (enfoque destructivo).
 - Por supuesto, sólo son destructivas en un sentido simbólico.
 - Una prueba tiene éxito si descubre un error nuevo.
 - No hay culpabilidad en el fallo.
 - Prueba clínica.

Objetivos

- Objetivo principal
 - Proyecto. Detectar errores en la menor cantidad de tiempo y con el menor número de recursos posibles.
 - Proceso. Diseño de técnicas que permitan un desarrollo sistemático de pruebas que garanticen ese objetivo.
- Ventajas secundarias
 - Demuestra hasta qué punto se verifican los requisitos.
 - Se generan datos de prueba que informan sobre la fiabilidad.
- No aseguran la ausencia de defectos

Definiciones

PRUEBAS

Una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y se registran, y se realiza una evaluación de algún aspecto.

- Probar es el proceso de ejecutar un programa con el fin de encontrar errores (Myers)



Definiciones

CASO DE PRUEBA

Un conjunto de entradas, condiciones de ejecución, y resultados desarrollados para un objetivo particular como, por ejemplo, ejercitarse en un camino concreto de un programa o verificar el cumplimiento de un requisito.

- Un buen caso de prueba es aquel que tiene una alta probabilidad de detectar un error
- $1 \text{ PRUEBA} \Rightarrow N \text{ CASOS DE PRUEBA}$

Definiciones

FALLO

La incapacidad de un sistema de realizar las funciones requeridas dentro de los requisitos (de rendimiento) especificados.

DEFECTO

Es la incorrección en el software que genera un fallo como, por ejemplo, un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa.

ERROR

Varias acepciones:

- Diferencia entre el valor calculado, observado o medido y el verdadero, especificado o teóricamente correcto.
- Un resultado incorrecto del software (*la definición de FALLO*).
- Un defecto en el software (*la definición de DEFECTO*).
- Una acción humana que conduce a un resultado incorrecto.

Definiciones

Relación entre ERROR, DEFECTO y FALLO



Principios

1. A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente.
2. Las pruebas deberían planificarse mucho antes de que empiecen.
3. El 80% de los errores surgen al hacer un seguimiento del 20% de los módulos del software (*Principio de Pareto*).
4. Las pruebas tendrían que hacerse de lo pequeño hacia lo grande.
5. No son posibles pruebas exhaustivas.
6. Las pruebas, para ser más efectivas, deberían de ser realizadas por un equipo independiente del equipo de desarrollo del software.
 - Además, deberían de hacer en un ambiente controlado diferente al de producción (separado físicamente).
 - Esta máquina independiente de la máquina de producción debería recrear las mismas condiciones que existen en la de producción.

Principios

7. Cada caso de prueba debe definir el resultado de salida esperado.
8. Se debe inspeccionar a conciencia el resultado de cada prueba para poder descubrir síntomas de defectos.
9. Al generar casos de prueba se deben incluir datos válidos y no válidos.
10. Las pruebas deben tener dos objetivos:
 - Probar si el software no hace lo que debe.
 - Probar si el software hace lo que no debe.
11. Se deben evitar los casos desecharables (no documentados).
12. No se deben hacer planes suponiendo que no habrá fallos.
13. Las pruebas son una tarea creativa.

El proceso de prueba



Técnicas de diseño de pruebas

¡Es imposible la prueba exhaustiva!



Técnicas de diseño de pruebas

¡Es imposible la prueba exhaustiva!

- Equilibrio entre confianza en el software y recursos consumidos.
 - Construir los mejores casos de prueba posibles
 - ¿Cómo puede fallar el software?
 - Evitar crear casos redundantes.
 - Ni demasiado sencillo ni demasiado complejo.

Métodos de prueba

PRUEBAS DE CAJA BLANCA

- Conocemos la estructura interna del producto.
 - Pruebas que aseguren que todos los módulos o componentes internos funcionan bien y encajan correctamente.
 - Minucioso examen de los detalles procedimentales.
 - Probaremos los caminos lógicos del software, con casos que ejerciten conjuntos específicos de condiciones y/o bucles.

PRUEBAS DE CAJA NEGRA

- No nos importa cómo está implementada la aplicación.
- Conocemos exactamente qué función debe realizar.
 - Pruebas que demuestren que cada función es completamente operativa.
 - Se llevarán a cabo pruebas sobre la(s) interface(s).

Métodos de prueba

- Prueba exhaustiva de CAJA BLANCA es impracticable:
 - Programa con dos bucles anidados de 0 a 20 en función de las entradas y cuatro condicionales IF-THEN-ELSE en el bucle interior.
 - Hay 10^{14} combinaciones; si cada prueba tarda 1 ms, son 3170 años.
- Prueba exhaustiva de CAJA NEGRA es impracticable:
 - Programa que sume dos números de dos cifras.
 - 10000 posibles combinaciones.
 - Faltan posibles errores:
 - Números negativos
 - Números mayores que 99
 - Letras
 - ...
 - Los enfoques NO SON EXCLUYENTES, sino COMPLEMENTARIOS

Pruebas estructurales. Caja blanca.

- ¿Porqué hacer pruebas de CAJA BLANCA y no sólo de CAJA NEGRA?
 - Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.
 - Los errores tipográficos son aleatorios.
 - Se suele creer que un determinado flujo es poco probable cuando, de hecho, puede ejecutarse regularmente.
 - El flujo lógico de un programa no tiene porqué ser intuitivo, lo que significa que nuestras suposiciones intuitivas sobre el flujo de control y los datos nos pueden llevar a errores de diseño que sólo se descubren en las pruebas del camino.

Pruebas estructurales. Caja blanca.

- Se trata de elegir casos de prueba que ofrezcan una seguridad aceptable de descubrir defectos
- Caso de prueba <=> Camino lógico
- CRITERIOS DE COBERTURA LÓGICA
- Aunque no son imprescindibles se suele utilizar el método gráfico denominado *Grafo de flujo*



Construcción de grafos

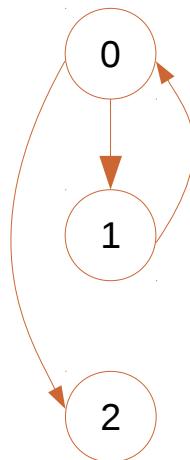
- Señalar sobre el código la condición de cada decisión:
 - *IF, SWITCH, DO, WHILE*
- Agrupar el resto de sentencias situadas entre cada dos condiciones
- Numerar cada nodo del grafo e identificar los nodos condición con una letra indicando en cada arista si ésta se debe a que la condición es verdadera o falsa

Construcción de grafos

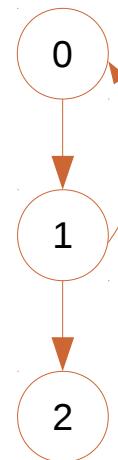
Secuencia



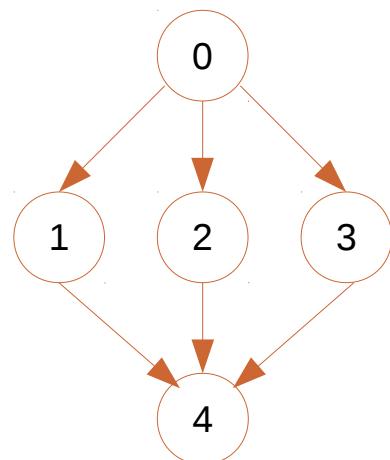
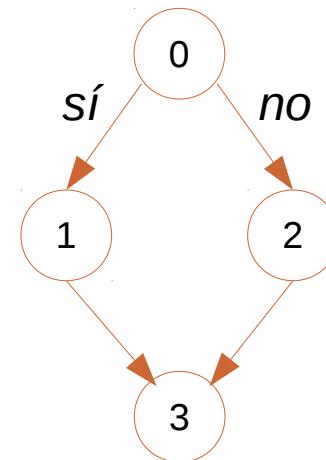
Mientras que



Repetir hasta que



Salto condicional

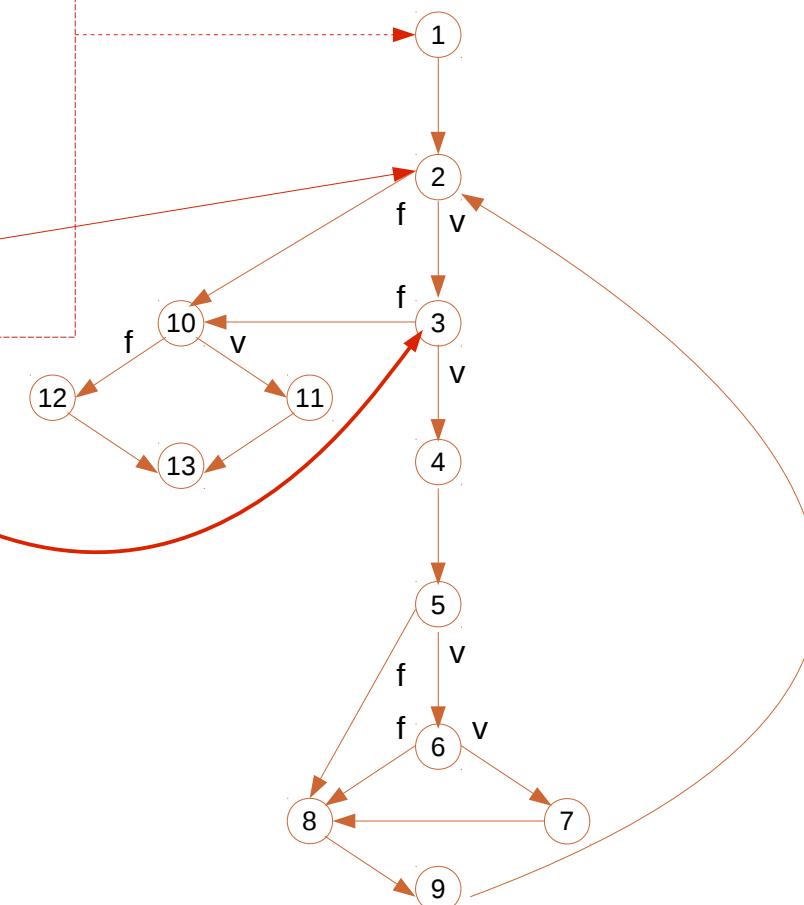


Según sea
(switch case)

Construcción de grafos

Procedimiento que recibe un array de valores decimales, un mínimo y un máximo, y calcula la media de los valores válidos, entre mín y máx.

```
PROCEDURE media
INTERFACE DEVUELVE media, total.entrada, total.válido
INTERFACE ACEPTE valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY DECIMAL
TYPE media, total.entrada, total.válido ES DECIMAL
TYPE mínimo, máximo, suma ES DECIMAL
TYPE i ES ENTERO
i = 1
total.entrada = total.válido = 0
suma = 0
WHILE valor[i] <> -999 AND total.entrada < 100
    total.entrada += 1
    IF valor[i] >= mínimo AND valor[i] <= máximo
        THEN
            total.válido += 1
            suma += valor[i]
        ELSE
            ignorar
        END IF
    i += 1
END WHILE
IF total.válido > 0
THEN
    media = suma/total.válido
ELSE
    media = -999
END IF
```



Construcción de grafos

- En un DIAGRAMA DE FLUJO representamos la secuencia de actividades de un proceso.
- En un GRAFO, una secuencia consecutiva se representa con un único nodo independientemente del número de actividades o su complejidad.

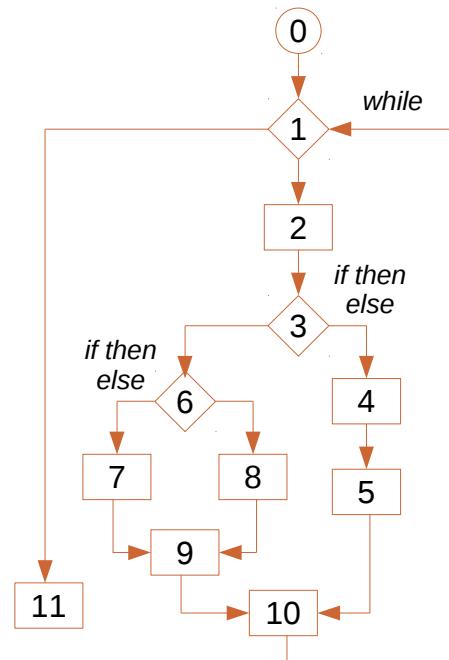
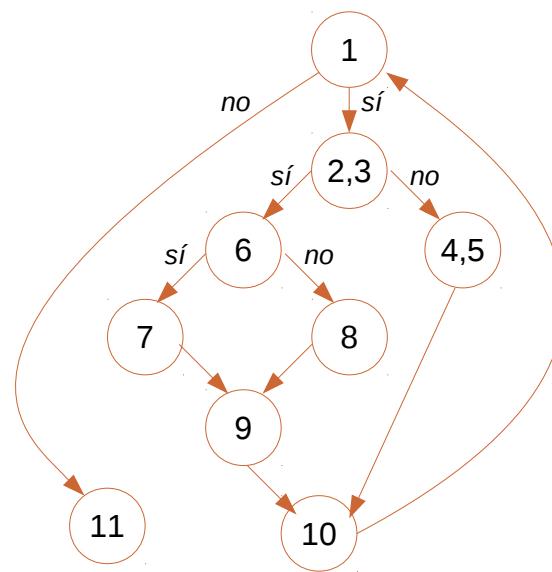


Diagrama de flujo



Grafo de flujo

Criterios de cobertura lógica

- **Cobertura de Sentencias.** Se trata de generar CASOS DE PRUEBA para que cada sentencia se ejecute una vez.
- **Cobertura de Decisiones.** Existen suficientes CP como para que cada decisión tenga, al menos una vez, un resultado verdadero y otro falso. En general, garantiza la *Cobertura de sentencia*.
- **Cobertura de Condición.** Cada condición de una decisión, adopta al menos una vez un resultado verdadero y otro falso. No garantiza la *Cobertura de decisión*.
- **Cobertura de Decisión/Condición.** Consiste en exigir los dos criterios anteriores simultáneamente.
- **Criterio de Condición Múltiple.** Descompone cada decisión múltiples en una secuencia de condiciones unicondicionales, y luego se exige que cada combinación posible de resultados en cada condición se ejecute al menos una vez.

Descomposición de condiciones compuestas

Condiciones compuestas

```
IF a OR b  
THEN  
    procedimiento X  
ELSE  
    procedimiento Y  
END IF
```

C. Decisión

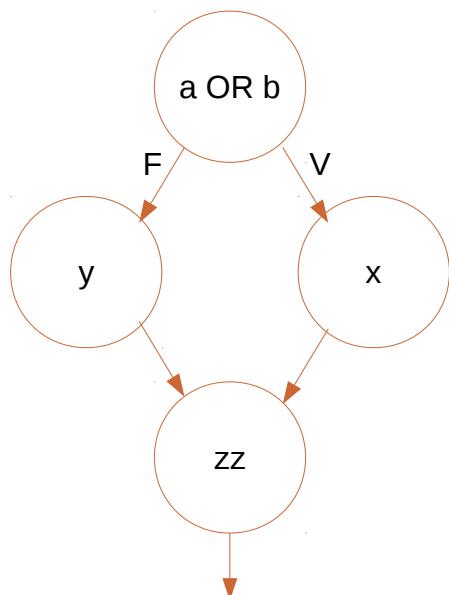
a	b	
1	0	xx
0	0	y

C. Condición

a	b	
1	0	xx
0	1	xx

C. Condición/Decisión

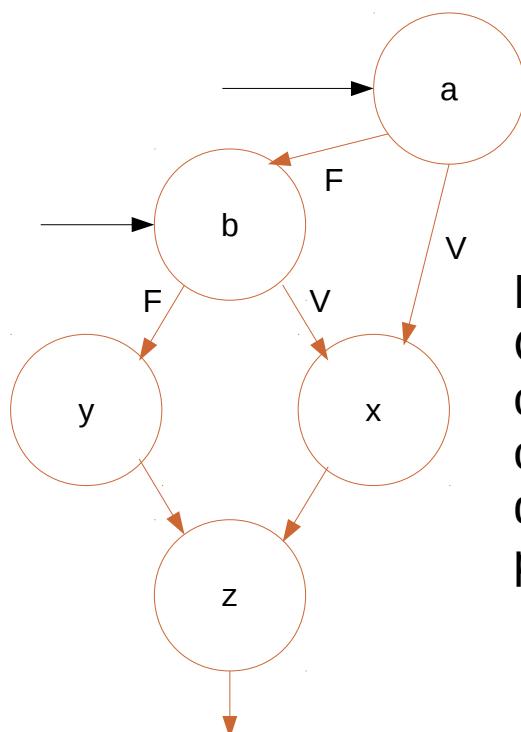
a	b	
1	1	xx
0	0	y



Descomposición de condiciones compuestas

Condiciones compuestas

```
IF a OR b  
THEN  
    procedimiento X  
ELSE  
    procedimiento Y  
END IF
```



Nodo predicado:
Contiene una
condición y se
caracteriza porque
dos o más aristas
parten de él

C. Decisión

a	b	
1	0	x
0	0	y

C. Condición

a	b	
1	0	x
0	1	x

C. Condición/Decisión

a	b	
1	1	x
0	0	y

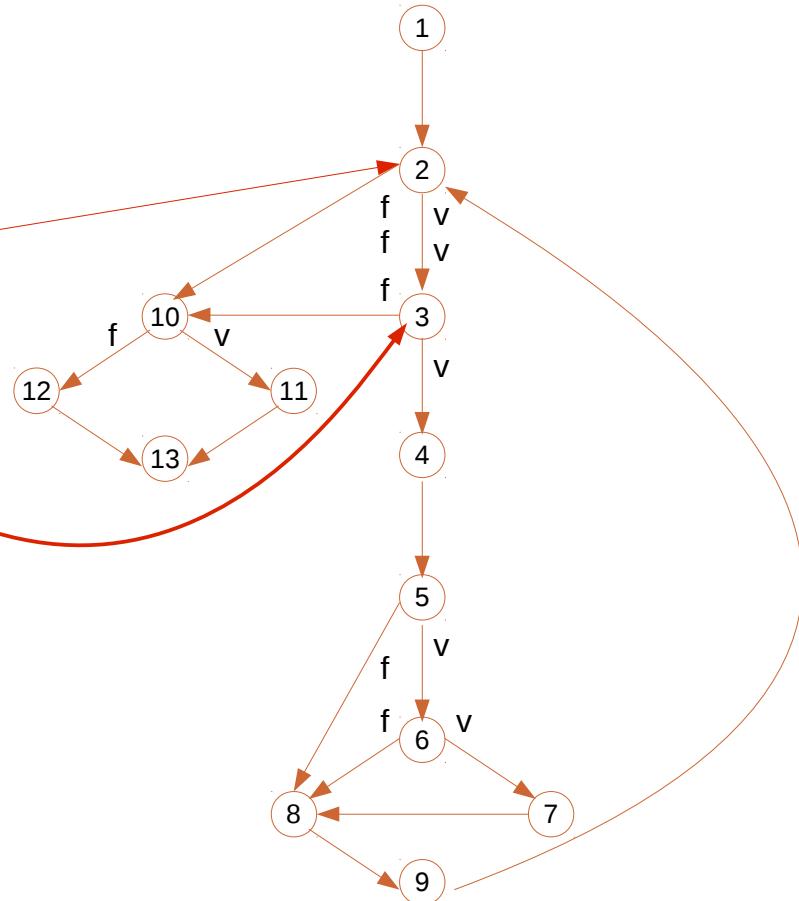
C. Condición Múltiple

a	b	
1	0	x
0	1	x
0	0	y

Descomposición de condiciones compuestas

- ¿Qué criterio de cobertura cumple el siguiente ejemplo?

```
PROCEDURE media
INTERFACE DEVUELVE media, total.entrada, total.válido
INTERFACE ACEPTE valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY DECIMAL
TYPE media, total.entrada, total.válido ES DECIMAL
TYPE mínimo, máximo, suma ES DECIMAL
TYPE i ES ENTERO
i = 1
total.entrada = total.válido = 0
suma = 0
WHILE valor[i] <> -999 AND total.entrada < 100
    total.entrada += 1
    IF valor[i] >= mínimo AND valor[i] <= máximo
        THEN
            total.válido += 1
            suma += valor[i]
        ELSE
            ignorar
        END IF
    i += 1
END WHILE
IF total.válido > 0
THEN
    media = suma/total.válido
ELSE
    media = -999
END IF
```



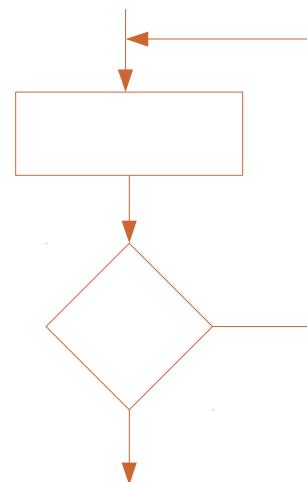
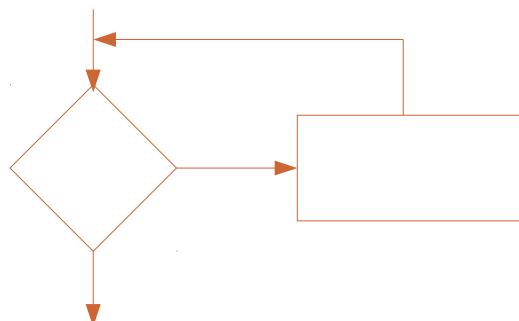
Criterios de cobertura lógica

- **Cobertura de caminos.** Es el criterio de cobertura más riguroso. Exige que cada camino del programa se ejecute al menos una vez.
- Problema: los bucles
 - Los bucles generan el mayor número de problemas con la cobertura de caminos. Bucles anidados o con condiciones que varían el número de repeticiones.
 - Camino de prueba. Un camino que atraviesa, como máximo, una vez el interior de cada bucle que se encuentra.
 - Camino de prueba insuficiente. Los bucles se deben recorrer 0, 1, 2 veces.

Tratamiento de bucles

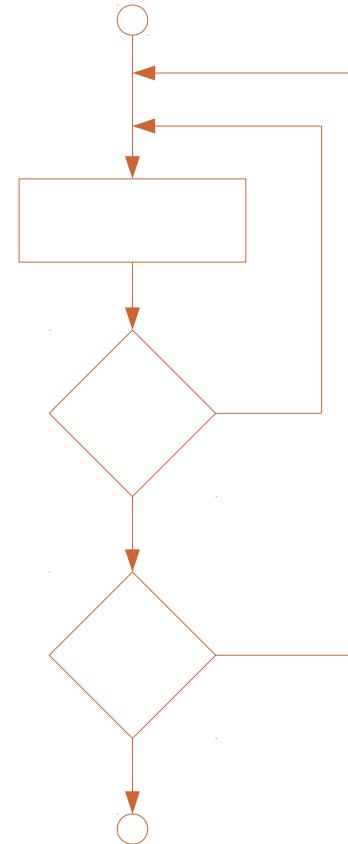
- **Bucles simples**, con n el número máximo de pasos permitidos para el bucle:

- Pasar totalmente por alto el bucle.
- Pasar una sola vez por el bucle.
- Pasar dos veces por el bucle.
- Hacer $m < n$ pasos por el bucle.
- Hacer $n-1$, n y $n+1$ pasos por el bucle.



Tratamiento de bucles

- **Bucles anidados.** No es posible extender la prueba del bucle simple.
 - Comenzar por el bucle más interior con los otros en sus valores mínimos.
 - Llevar a cabo la prueba de bucle simple al más interior.
 - Progresar hacia afuera llevando a cabo pruebas para el siguiente bucle y manteniendo los exteriores en sus valores mínimos y los interiores en sus valores típicos.
 - Continuar hasta probar todos los bucles.



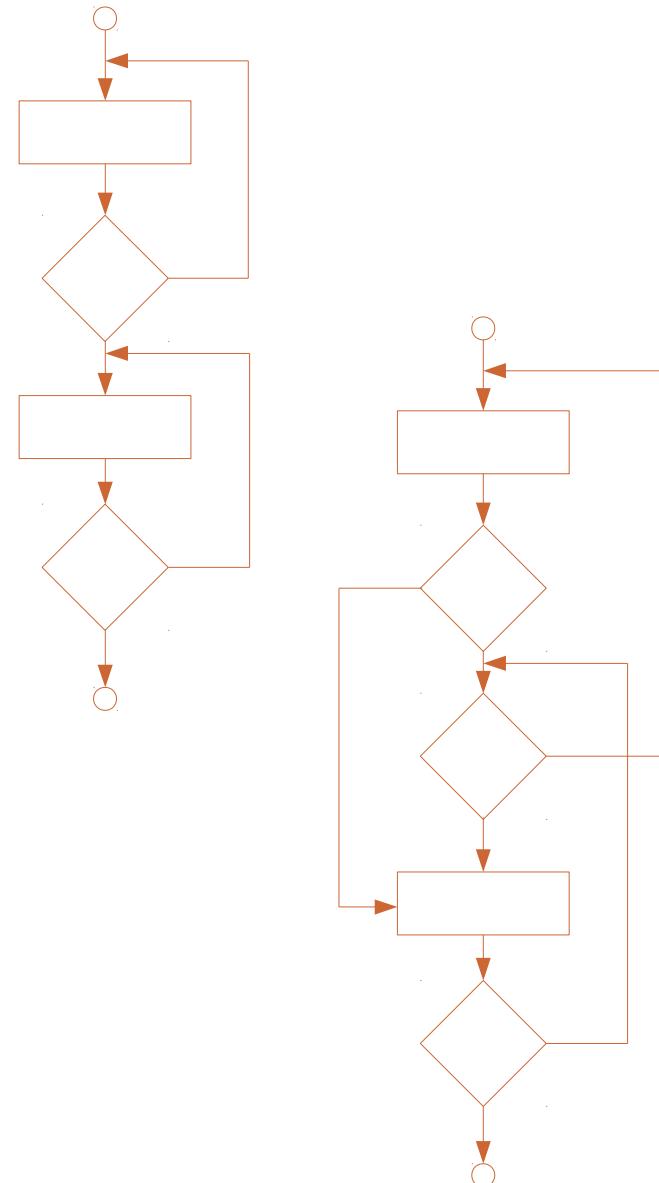
Tratamiento de bucles

- **Bucles concatenados.**

- Como los bucles simples si son independientes.
- Como los bucles anidados si no son independientes, por ejemplo, los valores del bucle A se usan como iniciales del bucle B.

- **Bucles no estructurados.**

- Siempre que sea posible deben reconstruirse como estructurados.



Complejidad ciclomática de McCabe

- Dado un grafo, ¿cuántos caminos son precisos para probarlo?
 - McCabe propone una métrica que indica el número de caminos independientes que existen en un grafo.
 - Esta métrica nos proporciona una medida cuantitativa de la complejidad lógica del módulo software que estamos probando.
 - Propone como un buen criterio de prueba la ejecución de un conjunto de caminos independientes cuyo número indica la métrica.
 - Nos permite establecer un conjunto básico de caminos de ejecución que aseguren que todo el código se ejecuta, al menos, una vez.
 - Este criterio se propone como equivalente a la cobertura de decisión.

Complejidad ciclomática de McCabe

- Cálculo de la complejidad ciclomática

- $V(G) = a - n + 2$

- a = número de arcos o aristas del grafo

- n = número de nodos

- $V(G) = c + 1$

- c = número de nodos de condición

- una condición de n arcos de salida se contabiliza como $n-1$

- $V(G) = r$

- r = el número de regiones cerradas del grafo

Complejidad ciclomática de McCabe

- En el número de regiones hay que tener presente que las fórmulas de McCabe sólo pueden aplicarse a grafos fuertemente conexos, donde siempre existe un camino entre cada dos nodos.
- En programas con un nodo inicial y otro final esto no se verifica, por lo que para calcular las regiones se deben unir estos nodos o contabilizar la región externa.

$$V(G) = A - N + 2 = 11 - 9 + 2 = 4$$

$$V(G) = P + 1 = 3 + 1 = 4$$

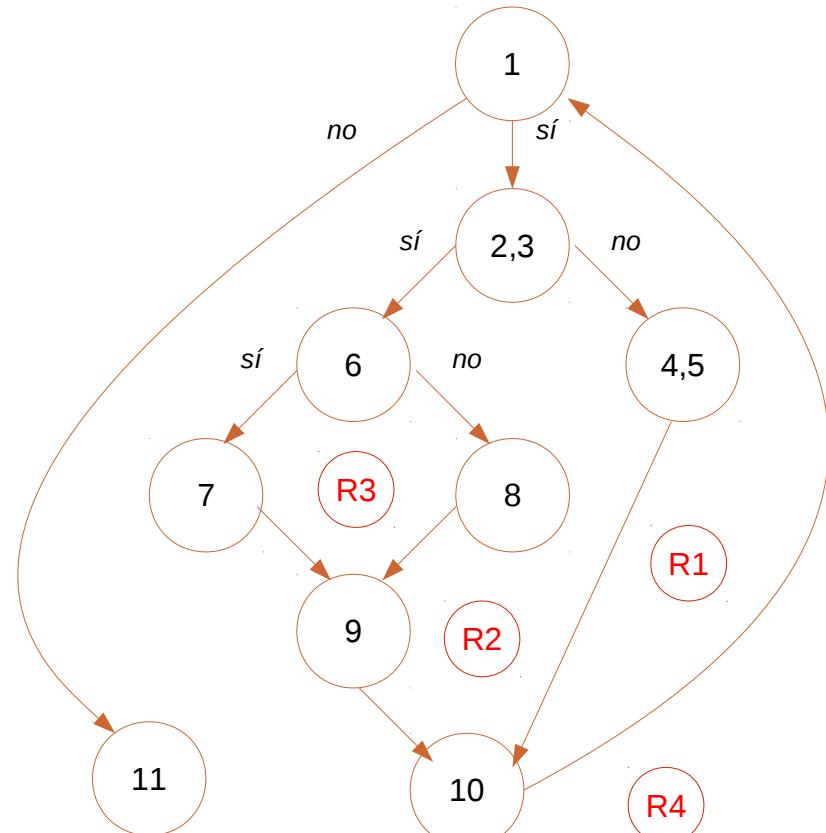
$$V(G) = R = 4$$

R1

R2

R3

R4



Complejidad ciclomática de McCabe

- El criterio de prueba de McCabe implica elegir tantos caminos de un grafo como caminos independientes haya.
- $V(G)$ constituye un límite superior que asegura la cobertura de sentencia y sería equivalente a la cobertura de decisiones.
- Cuando $V(G)$ es mayor que 10 la probabilidad de defectos en el módulo es alta (excepto que el número sea debido a sentencias CASE).

Complejidad ciclomática de McCabe

- Método del camino básico facilita la selección de los caminos independientes presentes en un grafo.
 - Selección de un camino de prueba típico o básico.
 - Crear variaciones sobre ese camino de forma que cada variación se distinga al menos en una arista de las demás.
 - Conviene tener presente que algunos caminos no se pueden ejecutar solos y necesitan de la ejecución de algún otro.
- Seleccionados los caminos debemos analizar el código para determinar las entradas que los fuerzan.
 - Es posible que éstas no existan encontrándonos ante un camino imposible que debe ser sustituido por otro que también permita satisfacer el criterio de McCabe.
 - A partir de las entradas debemos revisar la especificación para predecir las salidas.

Complejidad ciclomática de McCabe

- Ejemplo

CAMINOS INDEPENDIENTES

Camino 1: 1-11

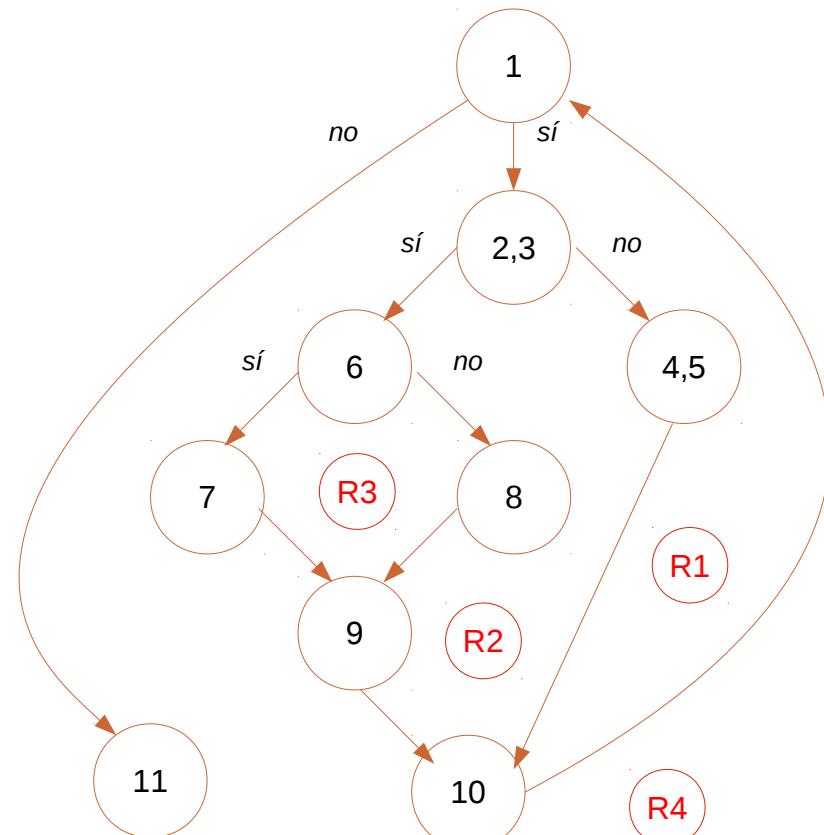
Camino 2: 1-2-3-4-5-10-1-11

Camino 3: 1-2-3-6-8-9-10-1-11

Camino 4: 1-2-3-6-7-9-10-1-11

OTROS CAMINOS

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



Complejidad ciclomática de McCabe

- Ejemplo completo

```
PROCEDURE media
INTERFACE DEVUELVE media, total.entrada, total.válido
INTERFACE ACEPTE valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY DECIMAL
TYPE media, total.entrada, total.válido ES DECIMAL
TYPE mínimo, máximo, suma ES DECIMAL
TYPE i ES ENTERO
i = 1
total.entrada = total.válido = 0
suma = 0
WHILE valor[i] <> -999 AND total.entrada < 100
    total.entrada += 1
    IF valor[i] >= mínimo AND valor[i] <= máximo
        THEN
            total.válido +=1
            suma += valor[i]
        ELSE
            ignorar
        END IF
        i += 1
    END WHILE
    IF total.válido > 0
        THEN
            media = suma/total.válido
        ELSE
            media = -999
    END IF
```

Complejidad ciclomática de McCabe

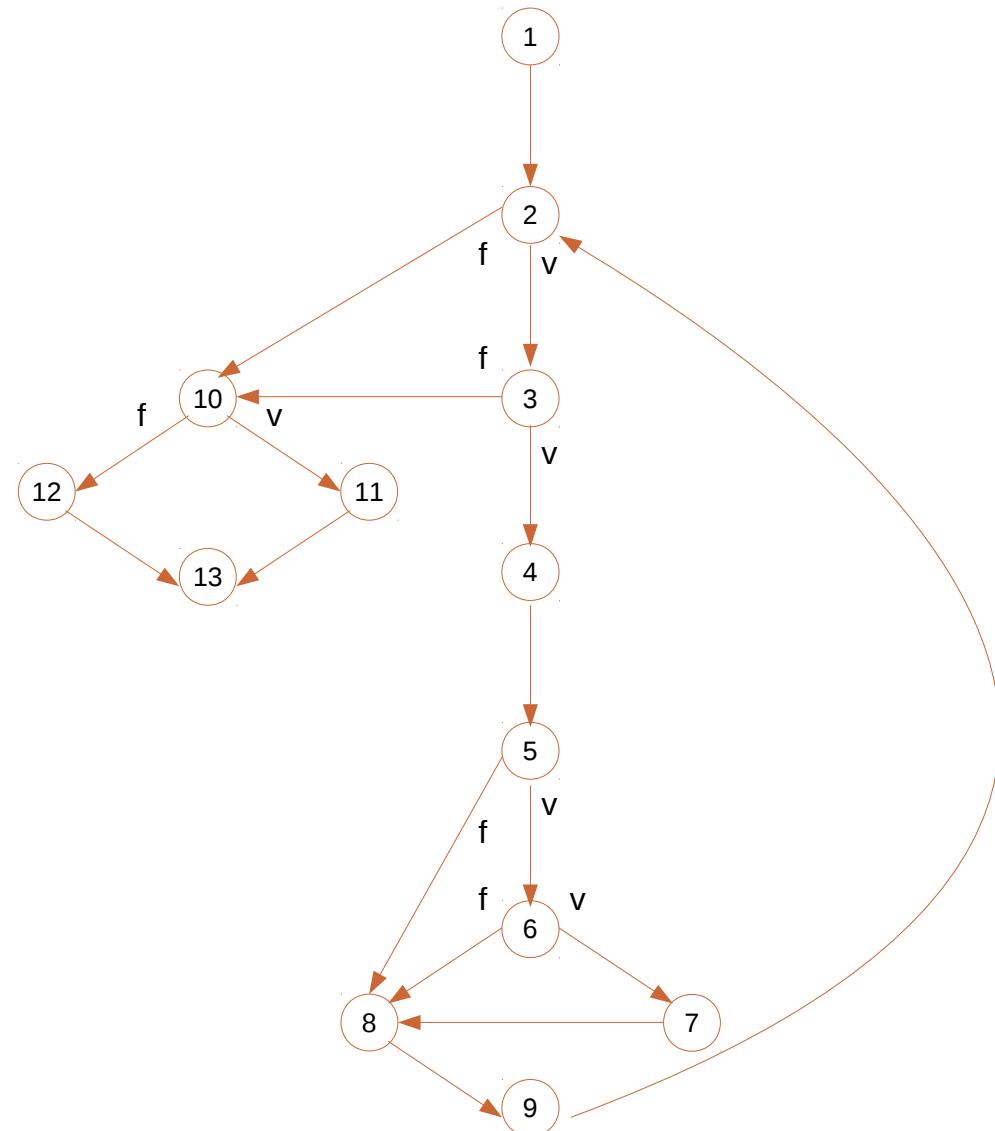
- Ejemplo completo (2)

```
PROCEDURE media
INTERFACE DEVUELVE media, total.entrada, total.válido
INTERFACE ACEPTA valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY DECIMAL
TYPE media, total.entrada, total.válido ES DECIMAL
TYPE mínimo, máximo, suma ES DECIMAL
TYPE i ES ENTERO
i = 1
total.entrada = total.válido = 0
suma = 0 1
WHILE valor[i] <> -999 2 AND total.entrada < 100 3
    total.entrada += 1 4
    IF valor[i] >= mínimo 5 AND valor[i] <= máximo 6
        THEN
            total.válido +=1
            suma += valor[i] 7
        ELSE
            ignorar
        END IF
        i += 1 8
    END WHILE 9
    IF total.válido > 0 10
        THEN
            media = suma/total.válido 11
        ELSE
            media = -999 12
    END IF 13
```

Complejidad ciclomática de McCabe

- Ejemplo completo (3)

```
PROCEDURE media
INTERFACE DEVUELVE media, total.entrada, total.válido
INTERFACE ACEPTE valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY DECIMAL
TYPE media, total.entrada, total.válido ES DECIMAL
TYPE mínimo, máximo, suma ES DECIMAL
TYPE i ES ENTERO
i = 1
total.entrada = total.válido = 0
suma = 0 1
WHILE valor[i] <> -999 2 AND total.entrada < 100 3
    total.entrada += 1 4
    IF valor[i] >= mínimo 5 AND valor[i] <= máximo 6
        THEN
            total.válido +=1
            suma += valor[i] 7
        ELSE
            ignorar
        END IF
        i += 1 8
    END WHILE 9
    IF total.válido > 0 10
        THEN
            media = suma/total.válido 11
        ELSE
            media = -999 12
    END IF 13
```



Complejidad ciclomática de McCabe

- Ejemplo completo (4)

COMPLEJIDAD CICLOMÁTICA

$$V(G) = a - n + 2 = 17 - 13 + 2 = 6$$

$$V(G) = r = 6$$

$$V(G) = p + 1 = 5 + 1 = 6$$

CONJUNTO BÁSICO DE CAMINOS

1: 1-2-10-11-13

2: 1-2-10-12-13

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

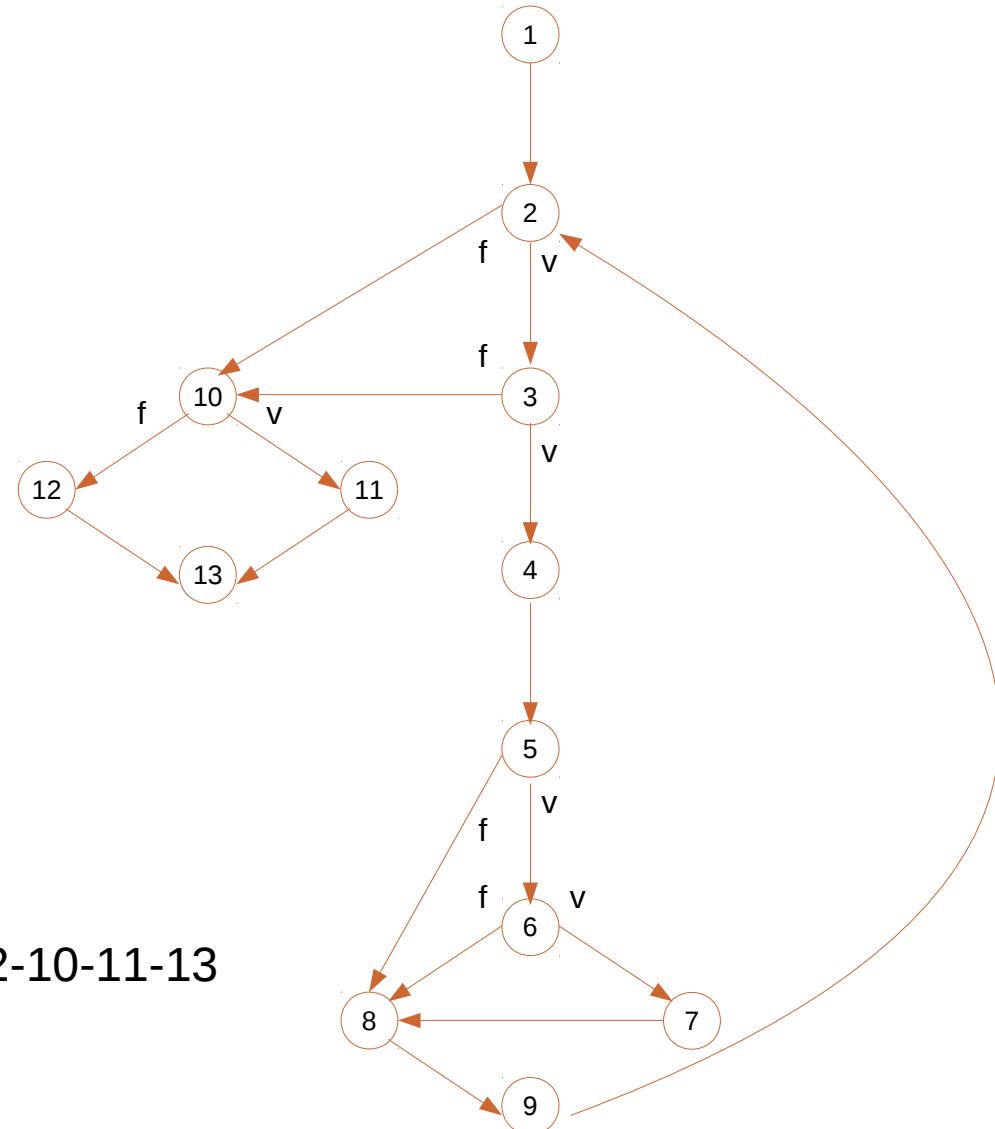
4: 1-2-3-4-5-8-9-2...

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

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

OTROS CAMINOS

7: 1-2-3-4-5-6-7-8-9-2-3-4-5-6-7-8-9-2-10-11-13



Complejidad ciclomática de McCabe

- Ejemplo completo (5)

```
PROCEDURE media
INTERFACE DEVUELVE media, total.entrada, total.válido
INTERFACE ACEPTE valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY DECIMAL
TYPE media, total.entrada, total.válido ES DECIMAL
TYPE mínimo, máximo, suma ES DECIMAL
TYPE i ES ENTERO
i = 1
total.entrada = total.válido = 0
suma = 0 1
WHILE valor[i] <> -999 2 AND total.entrada < 100 3
    total.entrada += 1 4
    IF valor[i] >= mínimo 5 AND valor[i] <= máximo 6
        THEN
            total.válido +=1
            suma += valor[i] 7
        ELSE
            ignorar
        END IF
        i += 1 8
    END WHILE 9
    IF total.válido > 0 10
        THEN
            media = suma/total.válido 11
        ELSE
            media = -999 12
    END IF 13
```

Camino 1: 1-2-10-11-13

Valor(1) = dato válido

Valor(i) = -999, $2 \leq i \leq 100$

Media = Valor(1)

Camino 1, alternativo:

1-2-3-4-5-6-7-8-9-2-10-11-13

Caso de prueba

Estado:

mínimo = 5

máximo = 20

Entrada:

Valor(1) = 10

Valor(i) = -999, $2 \leq i \leq 100$

Salida:

Media = 10

Complejidad ciclomática de McCabe

- Ejemplo completo (6)

```
PROCEDURE media
INTERFACE DEVUELVE media, total.entrada, total.válido
INTERFACE ACEPTE valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY DECIMAL
TYPE media, total.entrada, total.válido ES DECIMAL
TYPE mínimo, máximo, suma ES DECIMAL
TYPE i ES ENTERO
i = 1
total.entrada = total.válido = 0
suma = 0 1
WHILE valor[i] <> -999 2 AND total.entrada < 100 3
    total.entrada += 1 4
    IF valor[i] >= mínimo 5 AND valor[i] <= máximo 6
        THEN
            total.válido +=1
            suma += valor[i] 7
        ELSE
            ignorar
        END IF
        i += 1 8
    END WHILE 9
    IF total.válido > 0 10
        THEN
            media = suma/total.válido 11
        ELSE
            media = -999 12
    END IF 13
```

Camino 1: 1-2-10-11-13

Valor(1) = dato válido
Valor(i) = -999, $2 \leq i \leq 100$
Media = Valor(1)

Camino 2: 1-2-10-12-13

Valor(1) = -999
Media = -999

Caso de prueba

Estado:
mínimo = 5
máximo = 20
Entrada:
Valor(i) = -999, $1 \leq i \leq 100$
Salida:
Media = -999

Complejidad ciclomática de McCabe

- Ejemplo completo (7)

```
PROCEDURE media
INTERFACE DEVUELVE media, total.entrada, total.válido
INTERFACE ACEPTE valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY DECIMAL
TYPE media, total.entrada, total.válido ES DECIMAL
TYPE mínimo, máximo, suma ES DECIMAL
TYPE i ES ENTERO
i = 1
total.entrada = total.válido = 0
suma = 0 1
WHILE valor[i] <> -999 2 AND total.entrada < 100 3
    total.entrada += 1 4
    IF valor[i] >= mínimo 5 AND valor[i] <= máximo 6
        THEN
            total.válido +=1
            suma += valor[i] 7
        ELSE
            ignorar
    END IF
    i += 1 8
END WHILE 9
IF total.válido > 0 10
THEN
    media = suma/total.válido 11
ELSE
    media = -999 12
END IF 13
```

Camino 1: 1-2-10-11-13

Valor(1) = dato válido
Valor(i) = -999, $2 \leq i \leq 100$
Media = Valor(1)

Camino 2: 1-2-10-12-13

Valor(1) = -999
Media = -999

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

Intentamos procesar 100 valores, con
los 100 primeros <> -999
Media = valor medio de los 100

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

Valor(i) = dato válido, $i < 100$
Valor(k) < mínimo, $k < i$
Media = valor correcto sobre los datos > mínimo

Complejidad ciclomática de McCabe

- Ejemplo completo (8)

```
PROCEDURE media
INTERFACE DEVUELVE media, total.entrada, total.válido
INTERFACE ACEPTE valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY DECIMAL
TYPE media, total.entrada, total.válido ES DECIMAL
TYPE mínimo, máximo, suma ES DECIMAL
TYPE i ES ENTERO
i = 1
total.entrada = total.válido = 0
suma = 0 1
WHILE valor[i] <> -999 2 AND total.entrada < 100 3
    total.entrada += 1 4
    IF valor[i] >= mínimo 5 AND valor[i] <= máximo 6
        THEN
            total.válido +=1
            suma += valor[i] 7
        ELSE
            ignorar
        END IF
        i += 1 8
    END WHILE 9
    IF total.válido > 0 10
        THEN
            media = suma/total.válido 11
        ELSE
            media = -999 12
    END IF 13
```

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

Valor(i) = dato válido, $i < 100$

Valor(k) > máximo, $k < i$

Media = valor correcto sobre los datos menores que máximo

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

Valor(i) = dato válido, $i < 100$

Media: valor correcto sobre los n valores totales y adecuados

Pruebas funcionales. Caja negra.

- Trataremos de encontrar errores en las siguientes categorías:
 1. Funciones incorrectas o ausentes.
 - $\text{resta}(a, b) \Rightarrow n-a$
 2. Errores de interfaz.
 - Devuelve información incorrecta o incompleta.
 3. Errores en estructuras de datos o acceso a la BBDD.
 4. Errores de rendimiento.
 5. Errores de inicialización y terminación.
 - En el ejemplo de CAJA BLANCA no es posible configurar los valores máx y mín entre los que se calcula la media

Pruebas funcionales. Caja negra.

- La prueba funcional o de caja negra se centra en el estudio de las especificaciones.
 - Conocidas las entradas, qué salidas esperamos.
 - No es posible la prueba exhaustiva.

Pruebas funcionales. Caja negra.

- Selección de casos de prueba
 - Enfoque sistemático
 - Búsqueda de buenos casos de prueba
 - Un buen caso de prueba es aquel que tiene una alta probabilidad de detectar un nuevo error
 - El caso ejecuta el mayor número de posibilidades de entrada diferentes para así reducir el total de casos.
 - Cada entrada cubre un conjunto extenso de otras
 - Enfoque aleatorio
 - Pruebas aleatorias

Pruebas funcionales. Caja negra.

- Selección de casos de prueba
 - Enfoque sistemático
 - Partición o clases de equivalencia
 - Análisis de valores límite (*AVL*)
 - Conjetura de errores
 - Enfoque aleatorio
 - Pruebas aleatorias

Enfoque sistemático

- Busca utilizar casos de prueba bien elegidos.
- Definiciones:
 - El que reduce el número de casos necesarios para que la prueba sea razonable. Explota al máximo las posibilidades de combinaciones de entradas.
 - El que cubre un conjunto extenso de casos posibles. Da información sobre la ausencia o presencia de defectos con las entradas probadas, pero también sobre un conjunto de otras entradas similares no probadas.

Partición o clases de equivalencia

- Dividimos el dominio de valores de entradas en un número finito de clases que cumplan:
 - La prueba de un valor representativo de una clase permite suponer razonablemente que el resultado obtenido será el mismo que el obtenido probando cualquier otro valor de la clase.

Partición o clases de equivalencia

- Método de diseño de casos:
 - Identificar las clases de equivalencia.
 - Crear los casos de prueba correspondientes.

Partición o clases de equivalencia

- Identificar las clases de equivalencia.
 - Identificación de las condiciones de las entradas del programa (restricciones de formato y valores).
 - Identificación de las clases de equivalencia:
 - Datos válidos
 - Datos no válidos o erróneos

Partición o clases de equivalencia

- Reglas de identificación de clases:
 - **R1.** Rango tipo $5 < n < 7$
 - Una clase válida y dos no válidas
 - **R2.** Lista de valores de tamaño variable (por ejemplo, titulares de una cuenta bancaria, que pueden ser 1, 2...4)
 - Una clase válida y dos no válidas
 - **R3.** Situación tipo *A DEBE SER B* (por ejemplo, n debe ser un número)
 - Una clase válida y otra no válida
 - **R4.** Distintos valores admitidos, cada uno con un comportamiento distinto (por ejemplo, pago con tarjeta; cada tipo de tarjeta es único)
 - Una clase válida para cada una de ellos, y otra no válida
 - **R5.** Si es necesario, se dividen las clases en otras menores

Partición o clases de equivalencia

- Método de diseño de casos:
 - Identificar las clases de equivalencia.
 - Crear los casos de prueba correspondientes.
 - Pasos a seguir:
 - Numeramos las clases de equivalencia
 - Especificar casos de prueba que cubran el mayor número posible de clases de equivalencia válidas aún no cubiertas, hasta que todas las clases de equivalencia válidas estén cubiertas.
 - Especificar casos de prueba únicos por cada clase de equivalencia no válida sin cubrir hasta que todas ellas sean cubiertas.
 - **Contraejemplo**
 - Pregunta “introducir un valor entre 5 y 25”, respuesta 200A
 - El primero error enmascara al segundo, por lo que no se puede detectar con un único caso de prueba

Partición o clases de equivalencia

- Ejemplo
 - Leemos de fichero dos tipos de registros:
 - *registro_participantes*
 - *tipo_registro*; definido en el dominio numérico sin signo, de todos los posibles valores sólo toma el valor 0
 - *ganador*; definido en el dominio lógico, toma el valor TRUE si el participante es ganador y FALSE en caso contrario
 - *participante*; cadena numérica
 - *registro_preguntas*
 - *tipo_registro*; definido en un dominio numérico sin signo, valor 1
 - *pregunta*; definido en el dominio numérico, valores entre 1 y 99
 - *respuesta*; dominio alfanumérico

Partición o clases de equivalencia

Información	Tipo de dato	Regla	Clase válida	Clase no válida
			(ID) Dominio	(ID) Dominio
tipo_registro	valores válidos	4	(1) 0	
			(2) 1	(3) >1
	numérico sin signo	5, 3	>=0	(4) no es número positivo
ganador	booleana	4	(5) TRUE	(6) no existe
			(7) FALSE	
participante	txt	3	(8) existe	(9) no existe
pregunta	1-99	1	(10) 1-99	(11) <1
				(12) >99
respuesta	txt	3	(13) existe	(14) no existe

Partición o clases de equivalencia

- Además, de manera implícita,
 - Regla R2 para identificar errores en el número de participantes (casos 0, 1, 2) y el número de respuestas entregadas, 99 (casos <99, 99, >99)
 - Regla 2 para verificar el número de caracteres permitido para el tamaño de los textos: el nombre del participante y la respuesta a la pregunta
 - Regla R3 para determinar que el tipo de registro DEBE SER positivo (0, -1) y DEBE SER numérico (0, A)

Análisis de valores límites

- Técnica que complementa a las clases de equivalencia, con dos diferencias:
 - Seleccionamos elementos en la frontera de cada clase
 - Examinamos también el dominio de salida

Análisis de valores límites

- Reglas de construcción:
 - **R1.** Si una condición de entrada especifica un intervalo cerrado de valores, tipo [-1.0, 1.0], examinaremos exactamente los valores extremos del intervalo, en este caso -1.0 y 1.0, y los casos no válidos justo fuera del intervalo, en este caso -1.1 y 1.1 (si sólo se admite un decimal).
 - **R2.** Si una condición especifica un número de valores de entrada, por ejemplo “el fichero tendrá de 1 a 255 registros”, diseñaremos casos con los valores máximo y mínimo, en este caso 0, 1, 255 y 256.

Análisis de valores límites

- Reglas de construcción:

- **R3.** Usar la regla R1 para la condición de salida. Por ejemplo, “*el descuento aplicable será del 6% al 50%*”; intentaremos obtener descuentos para 5.99%, 6%, 50% y 50.01%.
- **R4.** Usar la regla R2 para cada condición de salida. Por ejemplo, “*el programa puede generar de 1 a 4 informes*”; trataremos de generar 0, 1, 4 y 5 informes.
- **R5.** Si la entrada o salida del programa es una colección ordenada de objetos (tabla, fichero secuencial...), nos centramos en el primer y último elemento.

Análisis de valores límites

- Consideraciones sobre las reglas R3 y R4:
 1. Los valores límite de entrada no siempre nos dan valores límites de salida.
$$y = \sin(x)$$
$$x \in [0, 2\pi] \Rightarrow y \in [-1, 1]$$
 2. No siempre se pueden obtener resultados fuera del rango de salida, pero es importante considerar esta posibilidad.

Conjetura de errores

- Se trata de hacer una lista de equivocaciones que pueda cometer un programador y a continuación diseñar un caso de prueba para cada elemento de la lista:
 - Valor 0, entrada y salida
 - En listas de valores, concentrarse en el caso de que no haya valores, que haya 1 o que sean todos iguales
 - Intentar imaginar qué se puede malinterpretar de las especificaciones
 - Considerar que el usuario no es muy hábil o es malintencionado

Pruebas aleatorias

- El sistema funciona correctamente con datos válidos
 - Eventualmente se deberían crear todas las posibles entradas del sistema. Podemos apoyarnos en métodos estadísticos para conseguir la misma distribución de entrada o reglas de comportamiento que deben seguir los datos.
- Test de esfuerzo
 - Se trata de simular la entrada al programa en la secuencia y la frecuencia con la que pueden aparecer en realidad los datos, de forma continua y sin parar.

Métodos de caja negra basados en grafos

- Son posibles distintos modelos
 - Modelo de estados finitos
 - Modelo de Transacción
 - Modelo de flujo de datos
- Todos ellos son modelos de CAJA NEGRA:
 - No modelan cómo está programado el software
 - En realidad no son los verdaderos estados del software ni sus transacciones ni sus flujos de datos
 - Modelan su comportamiento visible, pantallas por las que pasa, o el contexto del problema, cómo al hacer una reserva (con o sin software)

Métodos de caja negra basados en grafos

- Modelado de estados finitos
 - Los nodos son estados del software observables por el usuario y los enlaces representan las transiciones que ocurren para moverse de uno a otro.
 - Ejemplo: Al realizar una determinada función, el usuario ve una serie de pantallas en las que puede realizar distintas acciones que llevan a distintas pantallas:
 - Los NODOS reflejan las PANTALLAS
 - Las TRANSICIONES, ACCIONES que cambian la pantalla
 - *Pressman, pp.295-296*

Métodos de caja negra basados en grafos

- Modelado del flujo de transacciones
 - Los nodos representan los pasos de alguna transacción y los enlaces son las conexiones lógicas entre ellos. Podríamos partir del diagrama de flujo de datos.
 - Ejemplo: Los pasos requeridos para hacer una reserva en un hotel o una aerolínea. Son pasos del proceso, se use software o no.
 - *Pressman, pp.295-296*

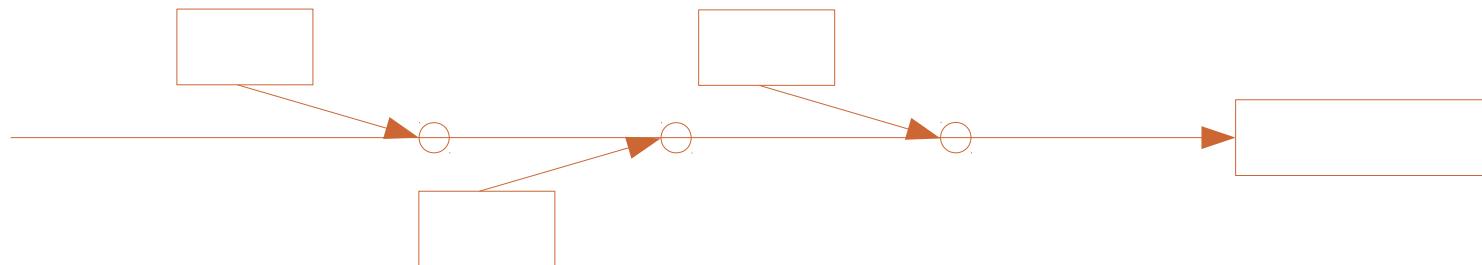
Métodos de caja negra basados en grafos

- Modelado del flujo de datos
 - Los nodos son objetos de datos y los enlaces son las transformaciones que sufren para pasar de ser un objeto a ser otro.
 - Ejemplo: Cálculo de hipoteca. A partir del valor de hipoteca y plazo de amortización resultan unos gastos, mensualidades e intereses que pueden llevar a cambiar el valor o el plazo.
 - *Pressman, pp.295-296*

Enfoque recomendado

1. Si la especificación contiene combinaciones de condiciones de entrada, comenzar formando grafos causa-efecto.

Hacer las pruebas en la misma secuencia que en el diagrama



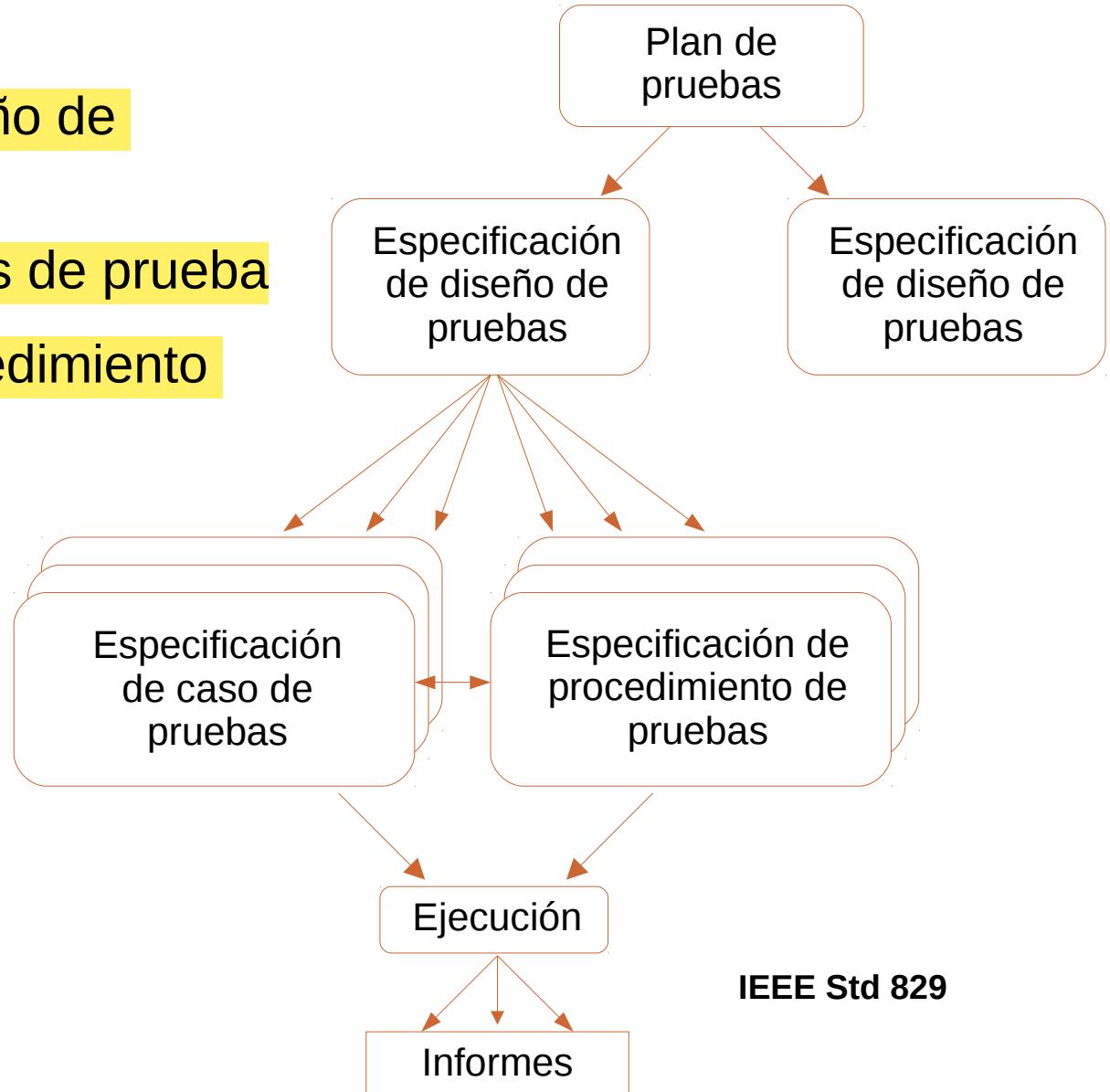
2. Identificar clases de equivalencia válidas y no válidas para la entrada y la salida.
3. En todos los casos usar análisis de valores límites para añadir casos de prueba.
4. Utilizar conjetura de errores para añadir nuevos casos.

Enfoque recomendado

5. Ejecutar los casos generados hasta el momento y analizar la cobertura obtenida.
6. Elegir casos precisos de caja blanca si en el punto anterior (5) no se ha cumplido el criterio de cobertura lógica elegido.
 - Aunque éste es el enfoque para una *prueba razonable*, en la práctica nos influencia la etapa (diagrama en V) en que estemos trabajando:
 - Por ejemplo, en pruebas de aceptación ya no tenemos que pensar en pruebas de caja blanca
 - Por ejemplo, en pruebas unitarias sobre un módulo el criterio de cobertura puede cumplirse sin recurrir a pruebas de caja negra

Documentación de diseño de pruebas

- Plan de pruebas
- Especificación del diseño de prueba
- Especificación de casos de prueba
- Especificación de procedimiento de pruebas
- Ejecución



IEEE Std 829

Documentación de diseño de pruebas

PLAN DE PRUEBAS

Planificación general del esfuerzo a realizar:

- ¿Dónde focalizamos las pruebas? Elementos y características
- ¿Qué estrategia seguimos? Enfoque
- ¿Cómo decidimos si el software pasa las pruebas? Paso/fallo/suspensión
- Recursos humanos, hardware y software
- Restricciones de tiempo y coste
- Riesgos: Qué puede bloquear o hacer ineficientes las pruebas
- Descripción de tiempos
- ¿Qué entregamos?

Documentación de diseño de pruebas

- Ejemplo (Enfoque):
 - *Las pruebas se centrarán en las funcionalidades vitales que incluirán casos no válidos. Las funcionalidades importantes sólo se probarán con casos válidos y las deseables por conjetaura de error*
 - *Se partirá de las pruebas de aceptación para identificar qué pruebas debe pasar el sistema. A partir de estas pruebas identificadas se identificarán las clases que colaboran en proporcionarlas y sobre éstas se centrarán las pruebas unitarias*
 - ...

Documentación de diseño de pruebas

1. Identificador único del documento PLAN (gestión de configuración)
2. Introducción y resumen de elementos y características a probar
3. Elementos software que se van a probar (módulos, programas...)
4. Características que se van a probar
5. Características que no se prueban
6. Enfoque general de la prueba (actividades, técnicas, herramientas...)
7. Criterios de paso/fallo para cada elemento
8. Criterios de suspensión y requisitos de reanudación
9. Documentación a entregar (como mínimo, los descritos en IEEE Std 829)
10. Actividades de preparación y realización de las pruebas
11. Necesidades de entorno
12. Responsabilidades en la organización y realización de las pruebas
13. Necesidades de personal y formación
14. Esquema de tiempos (tiempos estimados, hitos...)
15. Riesgos asumidos por el plan y planes de contingencia para cada riesgo

Documentación de diseño de pruebas

ESPECIFICACIÓN DEL DISEÑO DE PRUEBA

Detalla el plan de pruebas:

- Características de los elementos a probar
- Detalles como técnicas y métodos de análisis de resultados
- Identificación de pruebas (un identificador para cada prueba)
 - Describe los casos de prueba a utilizar (un identificador para cada caso)
 - Procedimiento a seguir
- Criterios de paso/fallo de la prueba

Documentación de diseño de pruebas

- Ejemplo (Prueba):
 - *RF1. Se crearán CASOS DE PRUEBA con clases de equivalencia, análisis de valores límite, conjetura de error y cobertura de sentencia.*
 - *RF2. Sólo se generan casos válidos con clases de equivalencia*
 - *Se considera que pasan las pruebas si todos los casos válidos funcionan*
 - *Se suspenden las pruebas si los casos válidos no funcionan o si el sistema queda bloqueado y éstas no pueden realizarse*

Documentación de diseño de pruebas

- Diseño de pruebas LTP/LTD
 - Característica a ser probada
 - Lista de métodos concretos
 - Organización de los métodos en
 - Métodos individuales
 - Combinación de métodos
 - Funcionalidad completa
 - Refinamiento del plan
 - Técnicas usadas
 - Estrategia de uso de pruebas según escenarios
 - Pruebas generadas
 - Característica específica bajo prueba
 - Método
 - Combinación de métodos
 - ...
 - Técnicas aplicadas (anexo)
 - Lista de casos de prueba
 - Procedimiento o lista de procedimientos
 - Criterio de paso/fallo de las pruebas
 - ¿Cuándo paramos? La característica está probada: fijado en el MTP

Documentación de diseño de pruebas

ESPECIFICACIÓN DE CASOS DE PRUEBA

Detalla los casos de prueba mencionados en el punto anterior:

- Elementos software y características a probar
- Especificación de las entradas requeridas, y su sincronización
- Especificación de las salidas, y sus características (tiempo de respuesta...)
- Necesidades del entorno (recursos humanos, software y hardware)
- Requisitos especiales
- Dependencias entre casos

Documentación de diseño de pruebas

Detalles:

- Entradas. No tiene sentido ejecutar un proceso para el que no tenemos todas las entradas disponibles. Por ejemplo, el calcular el estado de un paciente antes de recibir todas las variables de entrada: ecografía, análisis de orina...
- Dependencia entre casos. En ocasiones no se podrá probar un caso sin ejecutar otros previamente. Por ejemplo, adquirir datos de un sensor antes de darlo de alta...

Documentación de diseño de pruebas

ESPECIFICACIÓN DE PROCEDIMIENTO DE PRUEBAS

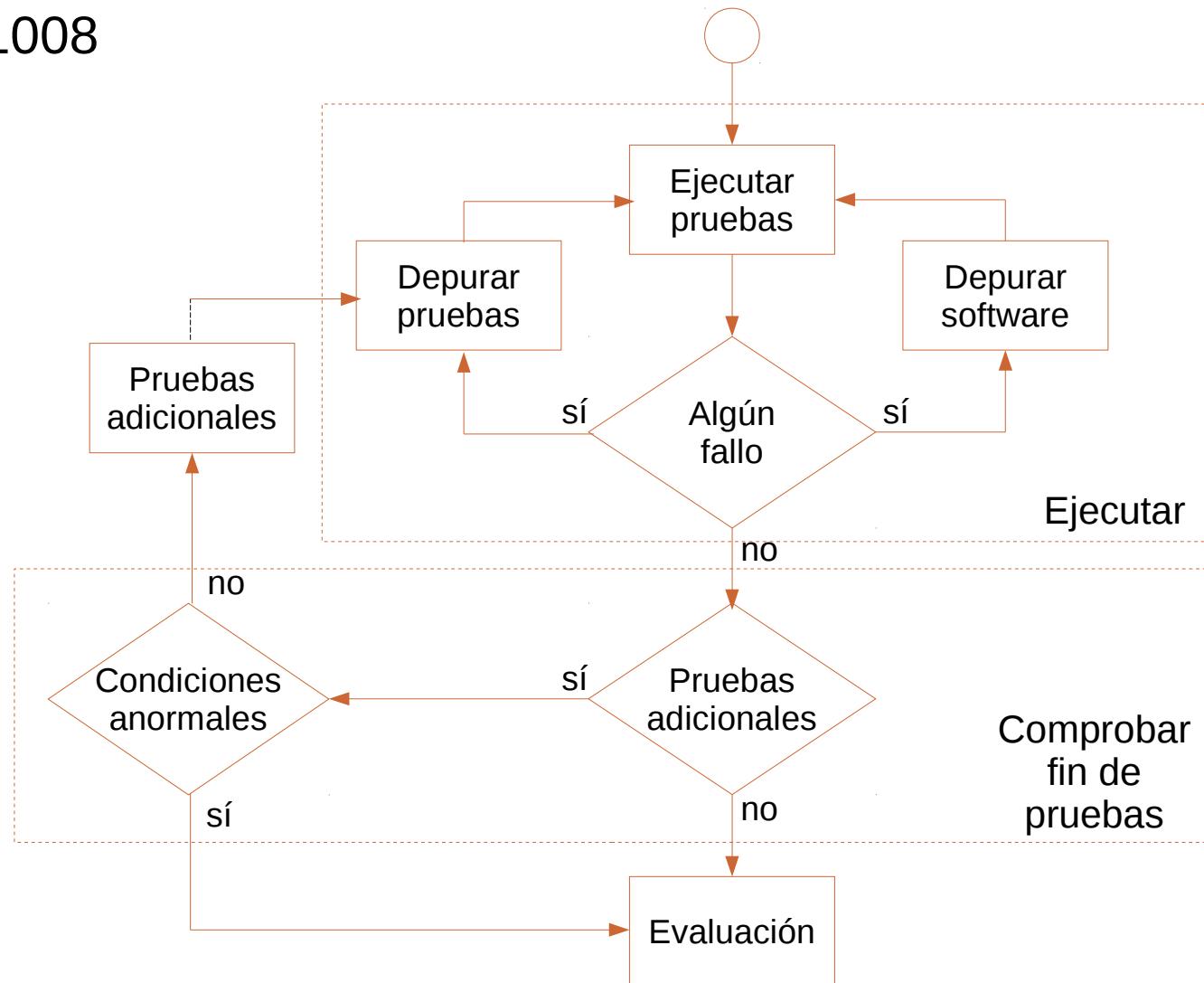
Indica cómo proceder con detalle a la ejecución de los casos

- Objetivos y lista de casos para evaluar un elemento
- Requisitos especiales
- Pasos: Formas de registrar resultados e incidencias
 - Secuencias necesarias para preparar la ejecución
 - Acciones para empezar y continuar la ejecución
 - Cómo realizar las medidas
 - Cómo tratar las incidencias y restaurar el entorno

Ejecución de las pruebas

EJECUCIÓN

IEEE Std 1008



Ejecución de las pruebas

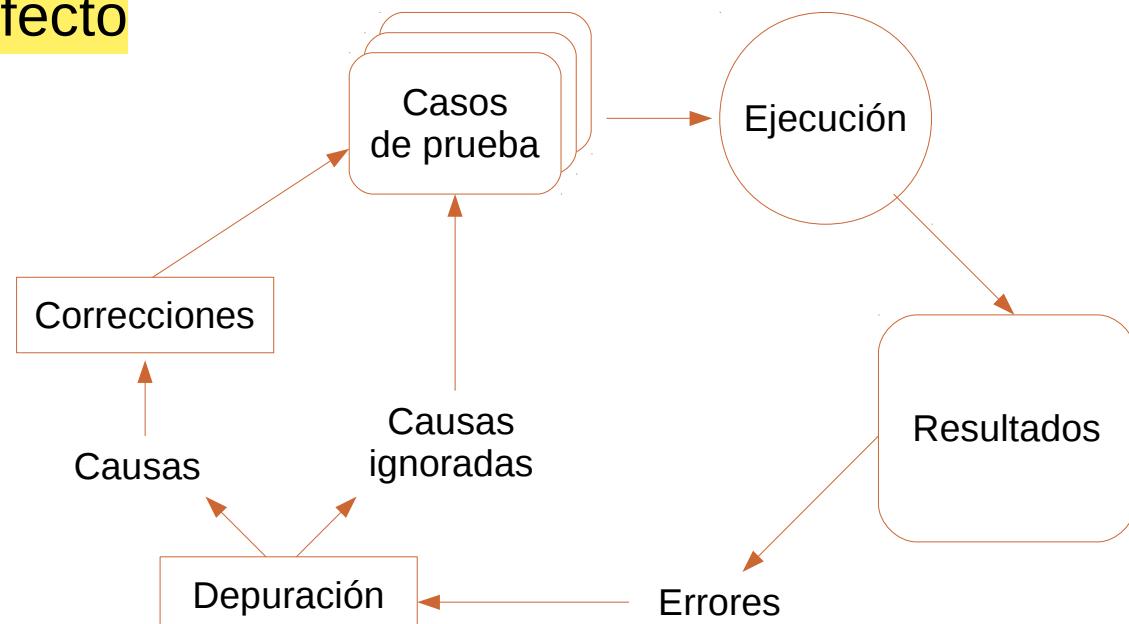
DOCUMENTACIÓN DE LA EJECUCIÓN

- Histórico de pruebas
 - Registro cronológico de la ejecución
 - Fecha, elementos probados y su entorno
 - Referencia al informe de incidencia, si lo hay
- Informe de incidencia
 - Resumen del incidente, análisis del impacto sobre las pruebas
- Informe resumen
 - Resume los resultados de las pruebas y aporta una evaluación del software basado en dichos resultados
 - Variaciones del software y las pruebas con respecto a su especificación de diseño
 - Valoración de la cobertura lógica alcanzada
 - Resumen de las actividades (detalles de recursos)

Ejecución de las pruebas

DEPURACIÓN

- Objetivos
 - Encontrar la causa del error, analizarla y corregirla
 - Si no se encuentra, generar nuevos casos de prueba
- Etapas
 - Localización del defecto
 - Corrección



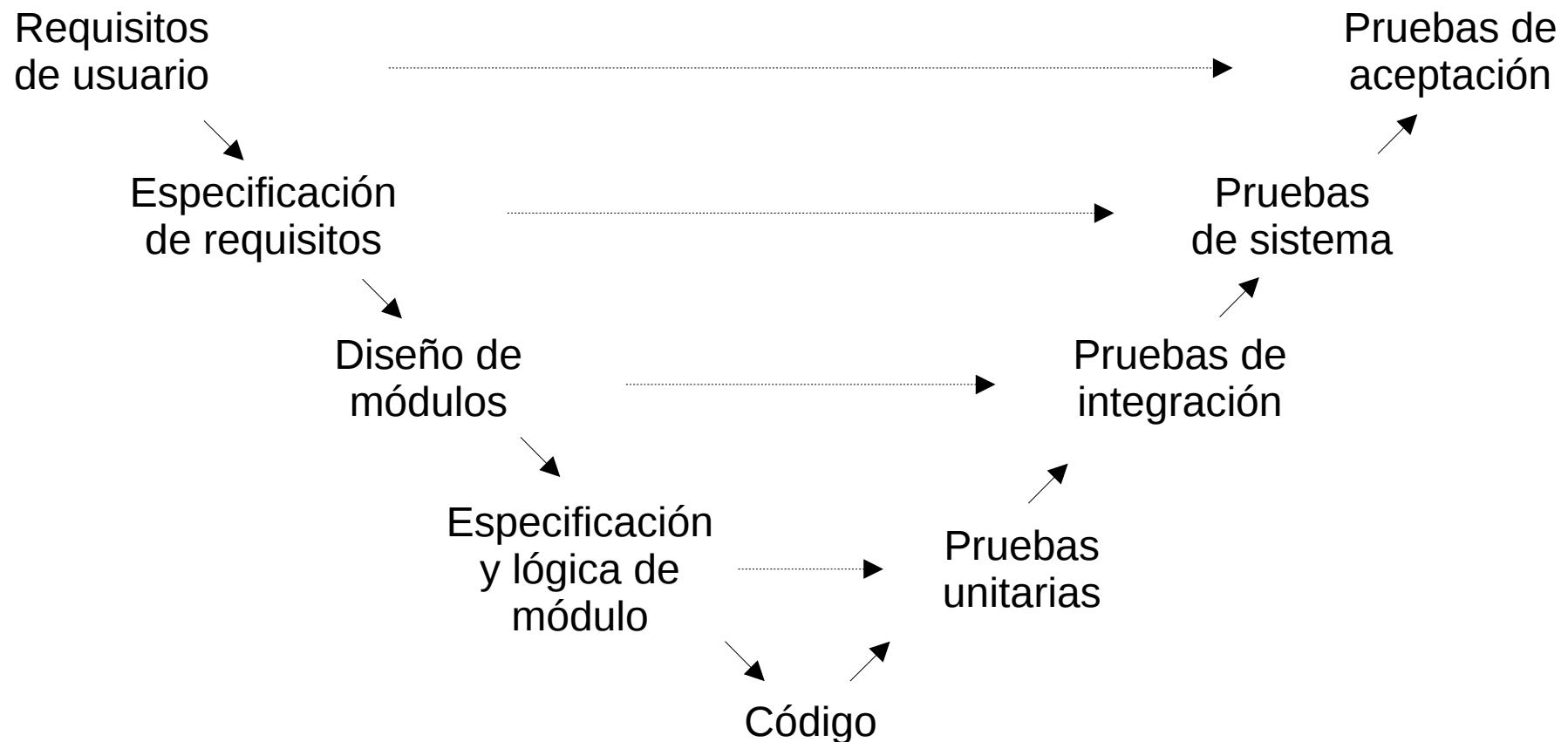
Ejecución de las pruebas

- Consejos de depuración:
 - Proceso mental de solución de un problema
 - Analizar la información
 - No explorar aleatoriamente
 - No experimentar cambiando el programa
 - Al llegar a un punto muerto
 - Pasar a otra cosa
 - Describir el problema a otra persona
 - Se deben atacar los problemas individualmente
 - Se debe fijar la atención también en los datos y no sólo en la lógica del proceso

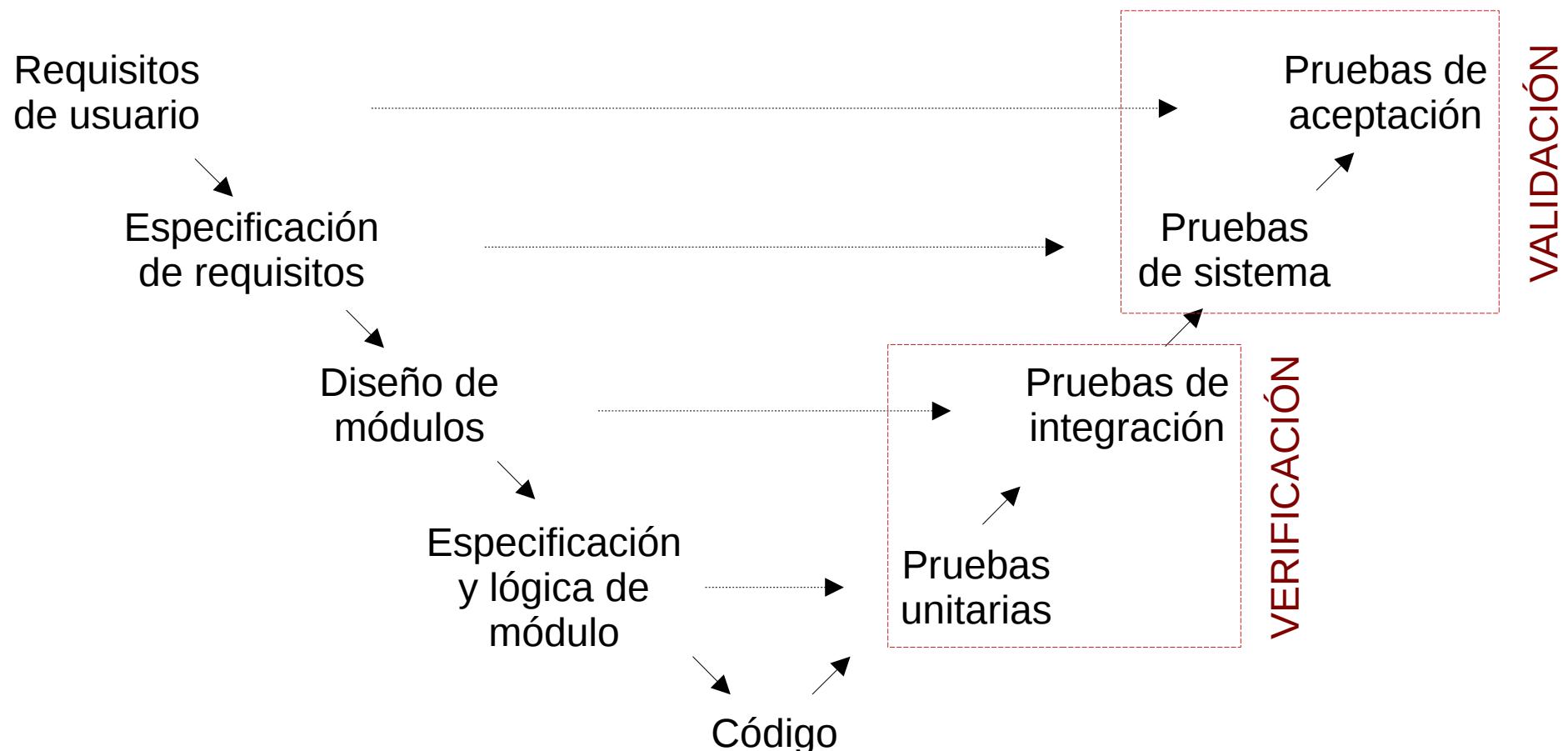
Ejecución de las pruebas

- Corrección del error:
 - Donde hay un defecto suele haber más
 - Debe corregirse el defecto, no sus síntomas
 - La probabilidad de corregir un defecto perfectamente no es del 100%
 - ¡Cuidado con crear otros nuevos!
 - La corrección nos sitúa temporalmente en la fase de diseño

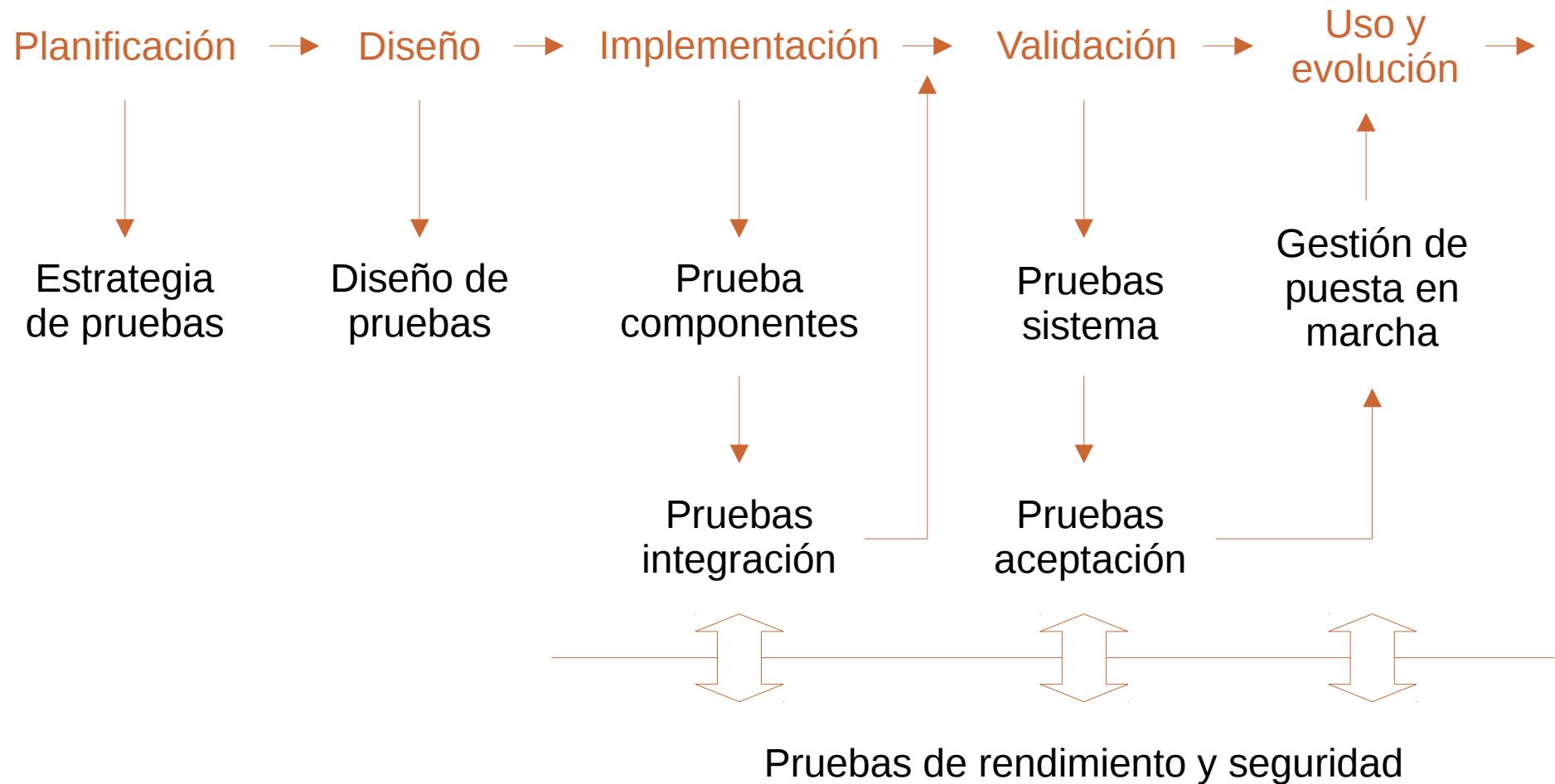
Estrategia de aplicación de las pruebas



Estrategia de aplicación de las pruebas



Estrategia de aplicación de las pruebas



Bibliografía

- Barry W. Boehm.; "Characteristics of Software Quality"; North-Holland Publishing Company, 1978.
- Glenford J. Myers, Corey Sandler, Tom Badgett; "The Art of Software Testing"; Ed. Wiley; 2011.
- Roger S. Pressman; "Ingeniería del Software. Un enfoque práctico". Ed. Mc Graw Hill, España, 2005.
- Ian Sommerville, "Ingeniería de software", Ed. Addison Wesley, 2011.