Temario

1. Introducción a la Ingeniería del Software
2. Ingeniería del software: Los procesos
3. Ingeniería del software: Los ciclos de vida
4. Análisis de requisitos
5. Modelado del Análisis Estructurado
6. **Pruebas del software**

Tema 6. Pruebas del software

INTRODUCCIÓN

- **VERIFICACIÓN:**
  - El proceso de evaluación de un 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? (Boehm)
- **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? (Boehm)

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 (Constructivo)
  - Pruebas para "demoler" lo construido (Destructivo)
    - No son destructivas en el sentido estricto del vocablo
  - No hay culpabilidad en el fallo
    - Prueba clínica
  - Una prueba tiene éxito si descubre un error nuevo

OBJETIVO

- Objetivo
  - 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 dicho objetivo.
- Ventajas Secundarias
  - Demuestra hasta que punto se verifican los requisitos
  - Se generan datos de prueba que informan sobre la fiabilidad
- No aseguran la ausencia de defectos.

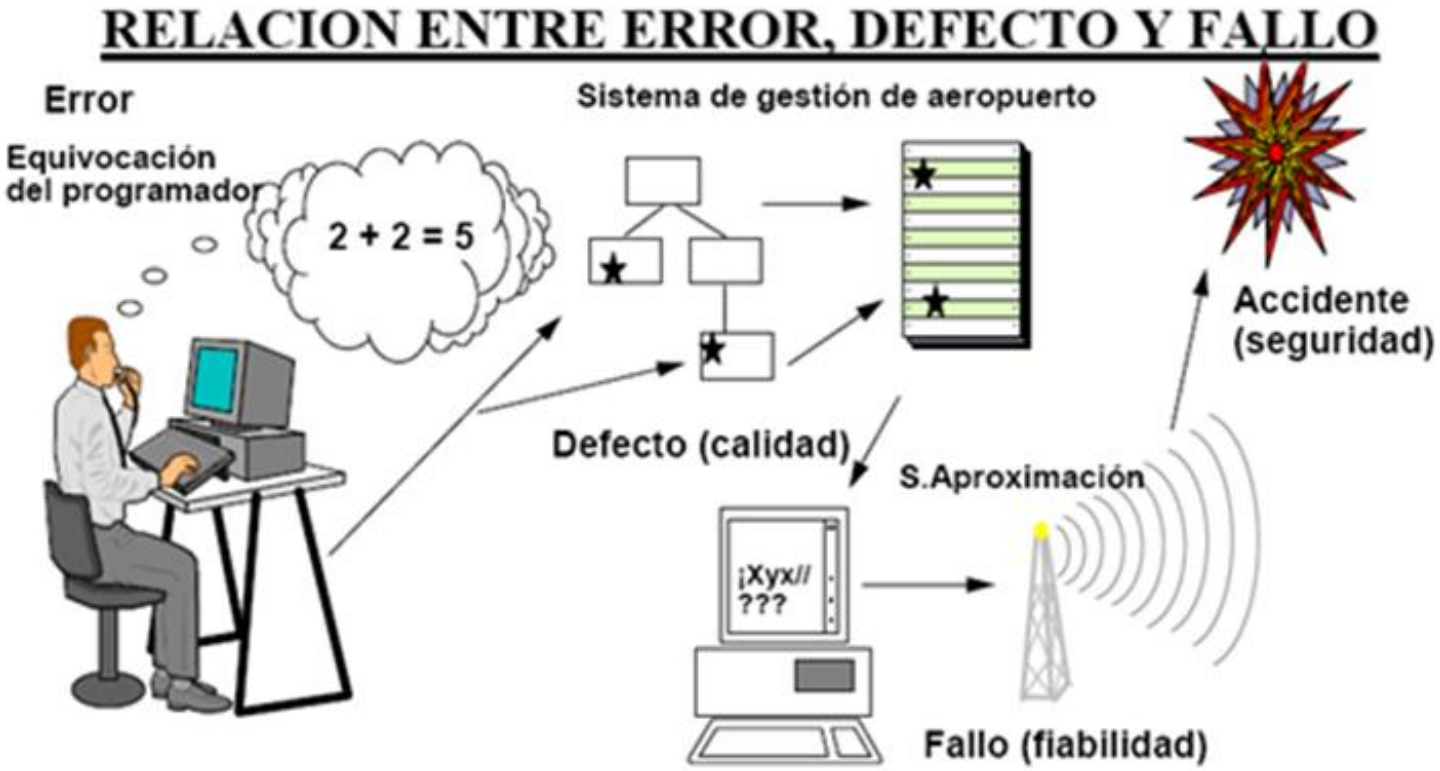
DEFINICIONES

- ***Pruebas***: Una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran se realiza una evaluación de algún aspecto.
  - Probar es el proceso de ejecutar un programa con el fin de encontrar errores. (MYERS)
- ***Caso de prueba***: Un conjunto de entradas, condiciones de ejecución, y resultados desarrollados para un objetivo particular como, por ejemplo, ejercitar 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 Prueba ⇒ N Casos_de_Prueba**

DEFINICIONES

- ***Fallo***: La incapacidad de un sistema para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados.
- ***Defecto***: Es una 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
  1. Diferencia entre un valor calculado, observado o medido y el verdadero, especificado o teóricamente correcto.
  2. Un resultado incorrecto del software
  3. Un defecto en el software
  4. Una acción humana que conduce a un resultado incorrecto.

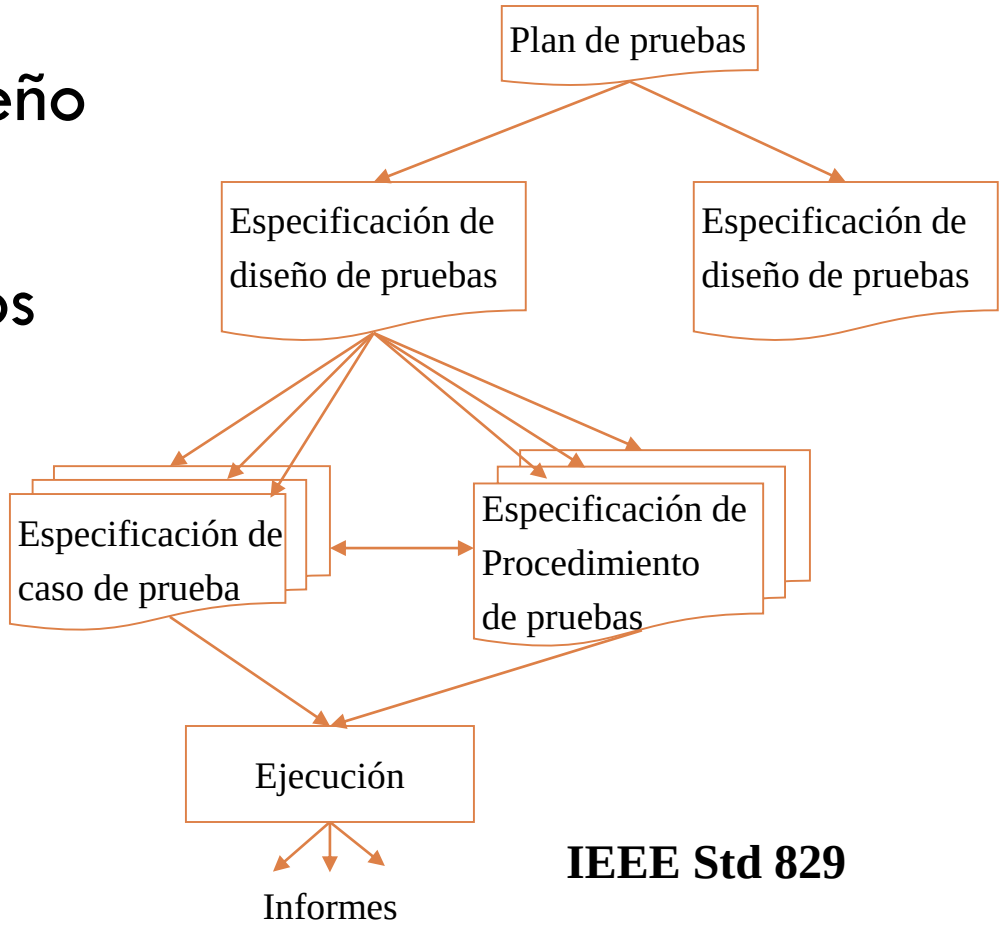
DEFINICIONES



Error (del programador) —Se Plasma→ **Defecto** (en el software) —Da lugar a→ **Fallo** (el sistema no hace lo que debe) —Que provoca→ Efectos negativos (dependiendo de la criticidad del sistema)

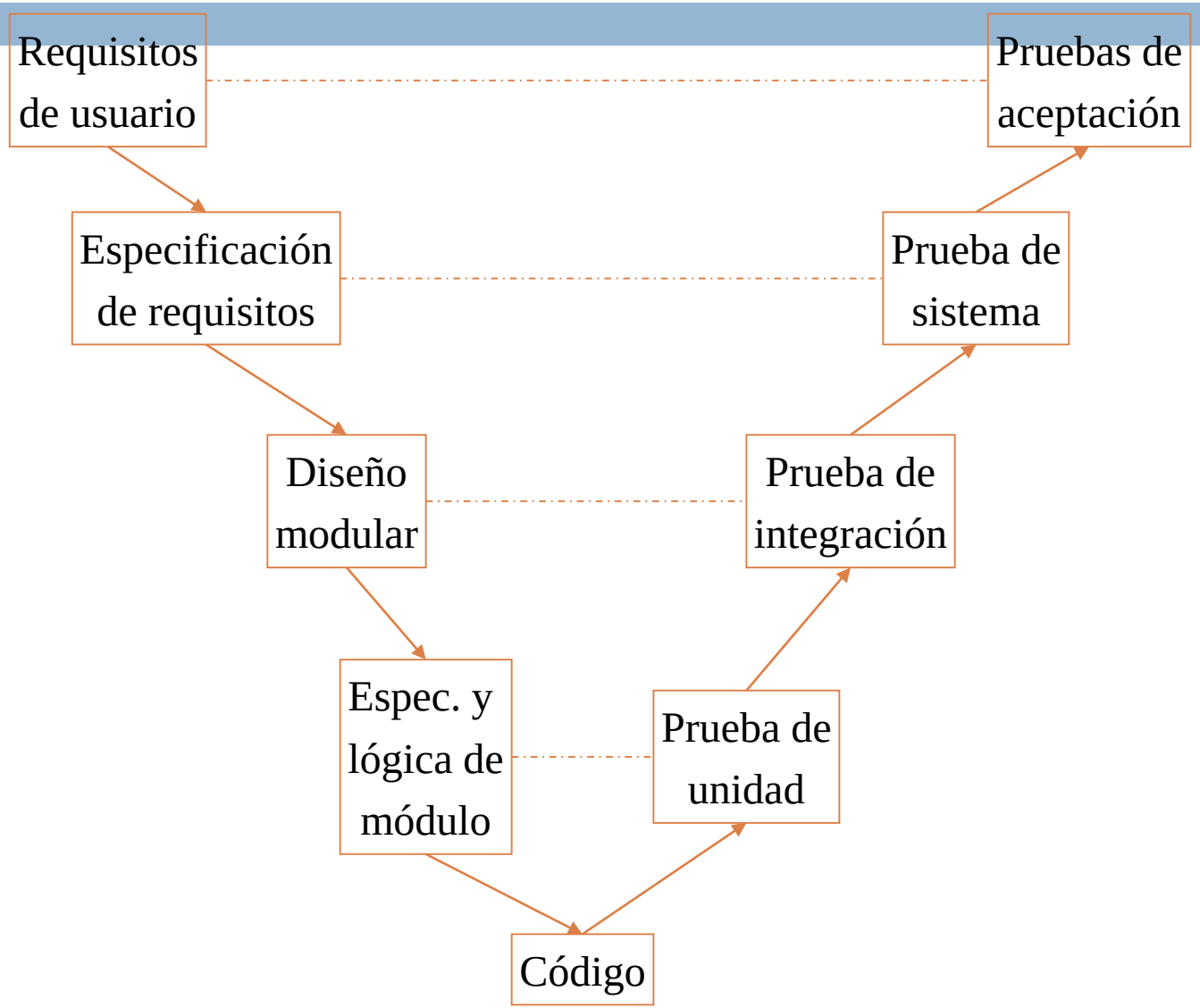
Documentación de diseño de pruebas

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

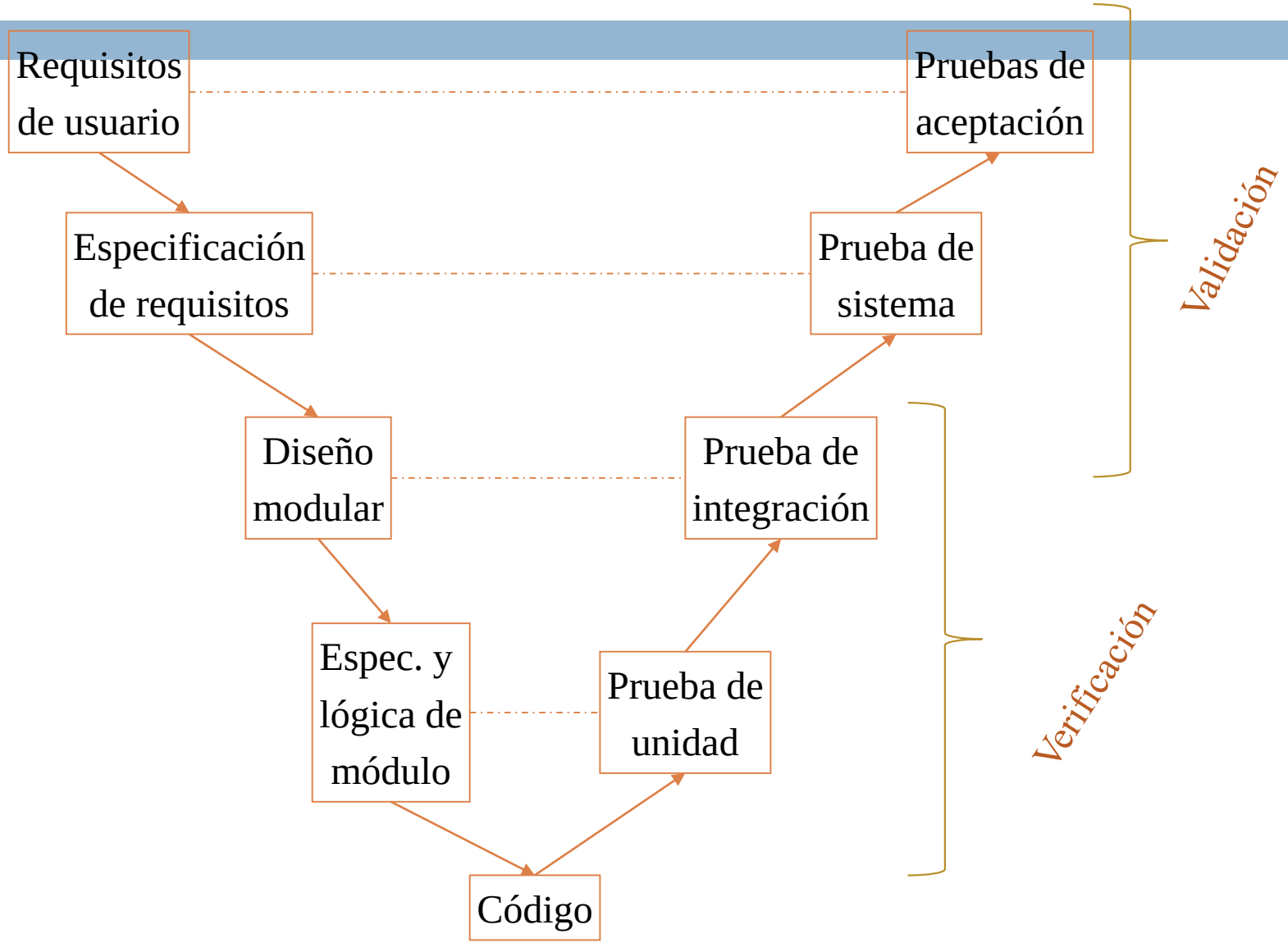


IEEE Std 829

Estrategia de aplicación de pruebas



Estrategia de aplicación de pruebas



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

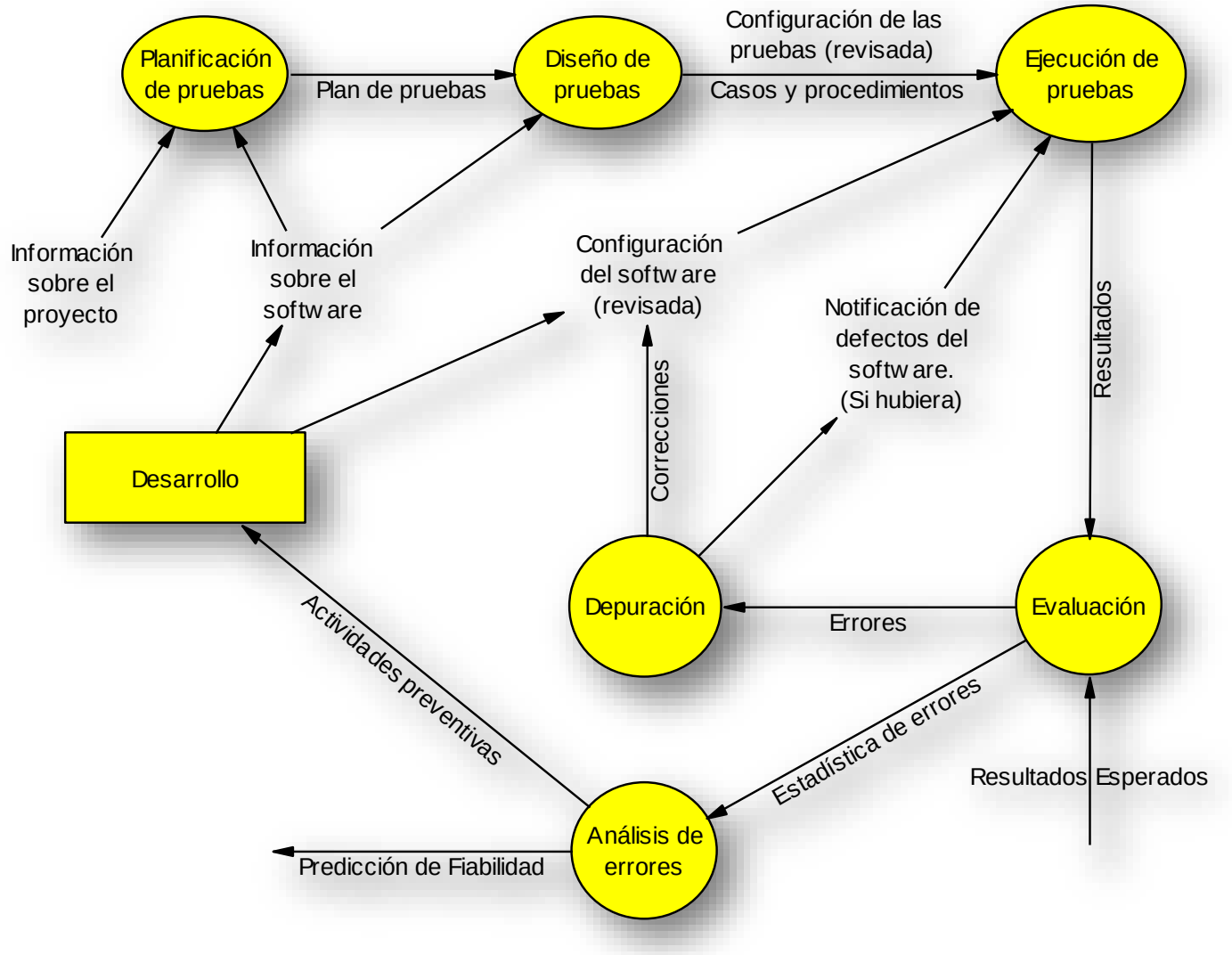
Davis [DAV95] – Pressman pp282

PRINCIPIOS

1. Cada caso de prueba debe definir el resultado de salida esperado.
2. Se debe inspeccionar a conciencia el resultado de cada prueba para poder descubrir síntomas de defectos
3. Al generar casos de prueba se deben incluir datos válidos y no válidos
4. Las pruebas deben tener dos objetivos
    1. Probar si el software no hace lo que debe
    2. Probar si el software hace lo que no debe
5. Se debe evitar los casos desechables (No documentados)
6. No se deben hacer planes asumiendo que no habrá fallos.
7. Las pruebas son una tarea creativa.
8. Añade además el Principio de Pareto e independencia del equipo de pruebas (3 y 6 de Davis)

Myers [Myers 79] – Piattini 422

El Proceso de Prueba



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?.
    - Casos no redundantes.
    - Ni demasiado sencillo ni demasiado complejo.

MÉTODOS DE PRUEBA

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

MÉTODOS DE PRUEBA

- Prueba exhaustiva de **Caja Negra** impracticable. (Ej. Piattini)
  - Programa que sume 2 números de 2 cifras.
  - 10.000 posibles combinaciones
  - Faltan posibles errores.
    - Números negativos
    - Números mayores que 100
    - Introducir letras
    - ...
- Prueba exhaustiva de **Caja Blanca** impracticable. (Ej. Pressman)
  - Programa con dos bucles anidados de 0 a 20 en función de las entradas y 4 if-then-else en el bucle interior
  - Hay $10^{14} \Rightarrow$ 1 prueba por 1 ms. $\Rightarrow$ 3170 años.
- Los enfoques no son excluyentes.
  - SON COMPLEMENTARIOS.

Tema 6. Pruebas del software

Pruebas Estructurales. Caja Blanca

- ¿Por qué hacer pruebas de caja blanca y no solo 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

Pruebas Estructurales. Caja Blanca

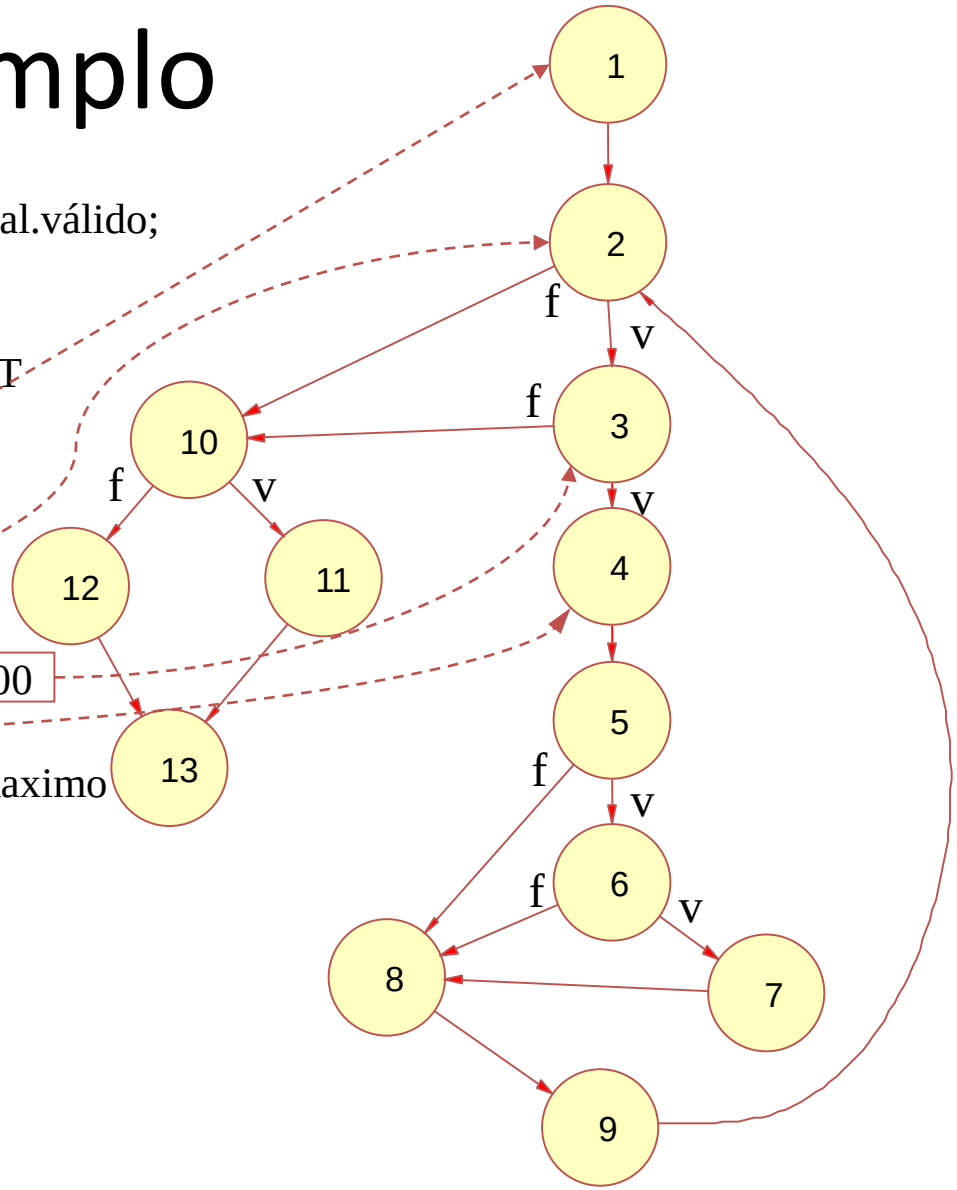
- Se trata de elegir casos de prueba que ofrezcan una seguridad aceptable de descubrir defectos.
  - Caso de prueba ≡ Camino lógico.
  - CRITERIOS DE COBERTURA LÓGICA
  - Aunque no son imprescindibles se suele utilizar el método gráfico denominado Grafo de flujo.

Construcción de grafos

- Señalar sobre el código la condición de cada decisión.
  - IF, SWITCH, DO, WHILE
- Agrupar el resto de las sentencias situadas entre cada dos condiciones según los siguientes esquemas
- 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.

Ejemplo

```
PROCEDURE media;
INTEFACE DEVUELVE media, total.entrada, total.válido;
INTERFACE ACEPTA  valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY FLOAT;
TYPE media, total.entrada, total.válido ES FLOAT
TYPE mínimo, máximo, suma ES FLOAT
TYPE i ES ENTERO
i=1;
total.entrada=total.válido=0;
suma=0;
DO WHILE valor[i]<>-999 AND total.entrada<100
        total.entrada+=1;
        IF valor[i]>=mínimo and valor[i]<= maximo
          THEN total.válido+=1;
                suma+=valor[i];
          ELSE ignorar
        ENDIF
         i+=1;
END DO
IF total.válido >0
        THEN media=suma/total.válido;
        ELSE media=-999;
ENDIF
```

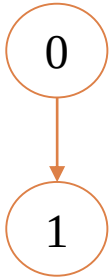


Construcción de grafos

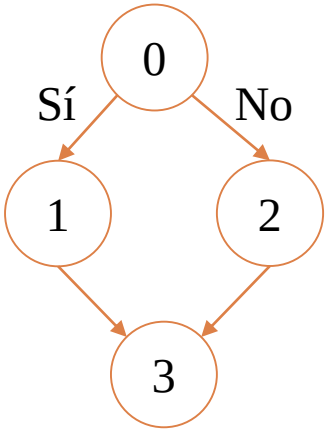
- Señalar sobre el código la condición de cada decisión.
  - IF, SWITCH, DO, WHILE
- Agrupar el resto de las sentencias situadas entre cada dos condiciones según los siguientes esquemas
- Numerar cada nodo del grafo e identificar los nodos condición con una letra indicando en cada arista si ésta se debe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

Construcción de grafos

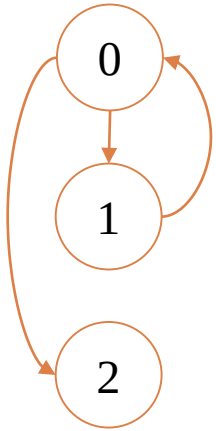
Secuencia



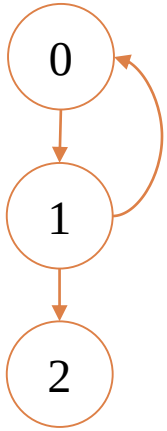
Salto condicional



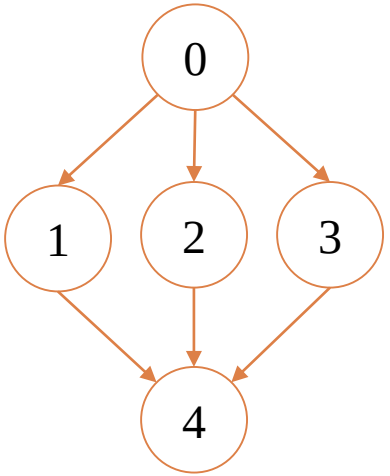
Mientras que



Repetir hasta que



Según sea (Swich Case)



Construcción de grafos

- Señalar sobre el código la condición de cada decisión.
  - IF, SWITCH, DO, WHILE
- Agrupar el resto de las sentencias situadas entre cada dos condiciones según los siguientes esquemas
- 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.

T.7.- Prueba del software

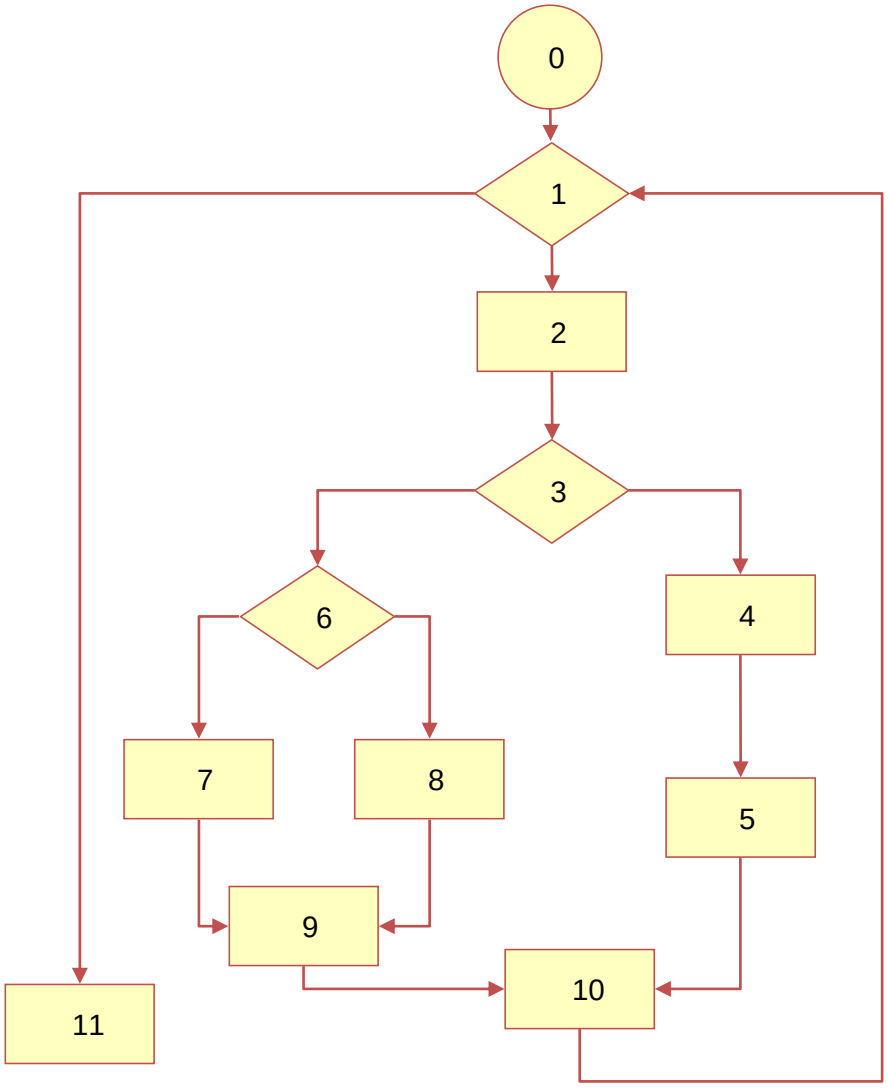
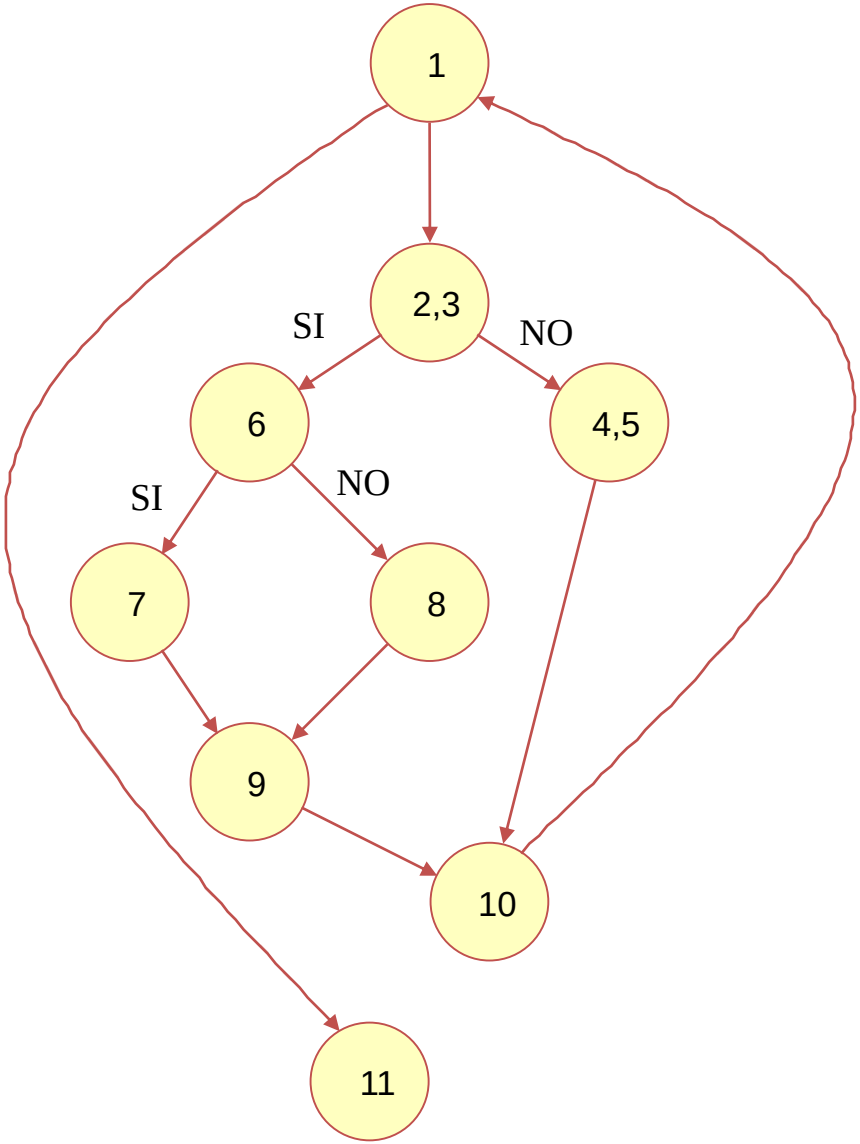


Diagrama de flujo



Grafo de flujo

Criterios de cobertura lógica

- **Cobertura de Sentencias:** Se trata de generar CP 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 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 múltiple 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.

Myers [Myers 79] – Piattini 429

Descomposición de condiciones compuestas

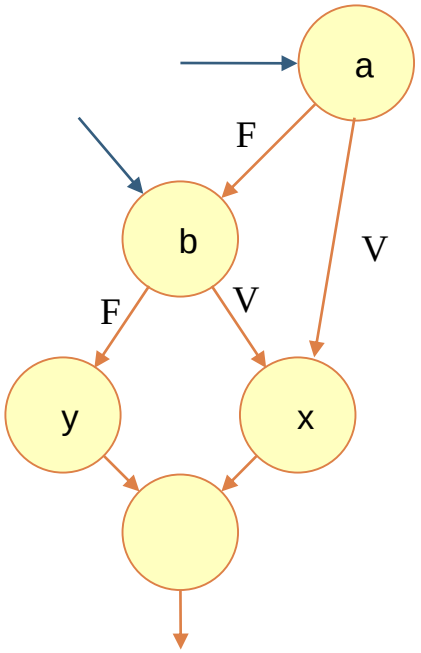
Condiciones Compuestas

IF a OR b

Then Procedimiento X

Else Procedimiento Y

Endif



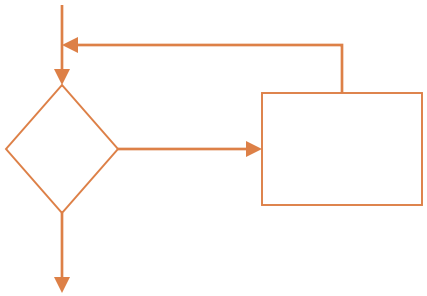
- Nodo Predicado: Contiene una condición y se caracteriza porque dos o más aristas parten de él.

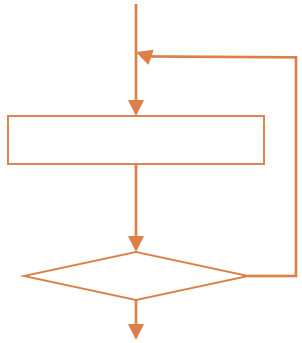
Criterios de cobertura lógica

- **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. [HUANG]

Tratamiento de Bucles

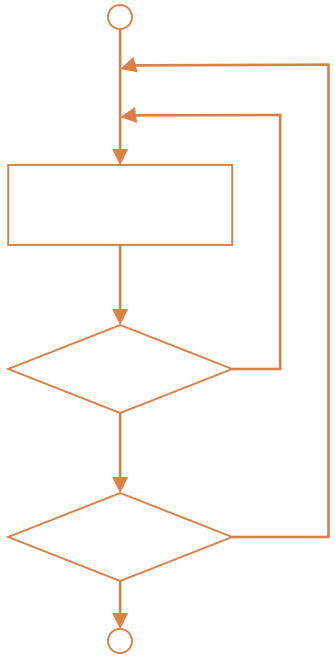
- **Bucles simples:** n es el número máximo de pasos permitidos para el bucle
  1. Pasar totalmente por alto el bucle
  2. Pasar una sola vez por el bucle
  3. Pasar dos veces por el bucle
  4. Hacer m pasos con m<n
  5. Hacer n-1, n y n+1 pasos por el bucle.





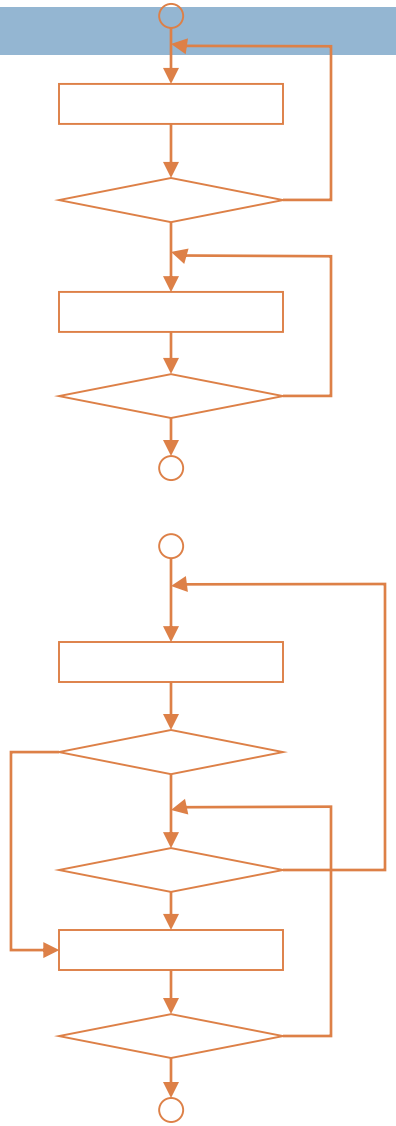
Tratamiento de Bucles

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



Tratamiento de Bucles

- **Bucles Concatenados:**
  - Como los bucles simples si son independientes
  - Como los bucles anidados si los bucles no son independientes, por ejemplo, los valores del bucle 1 se usan como iniciales para el bucle 2.
- **Bucles no estructurados**
  - Siempre que sea posible deben reconstruirse como estructurados.



Complejidad Ciclomática de McCabe

- Dado un grafo cuantos 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 permitirá 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

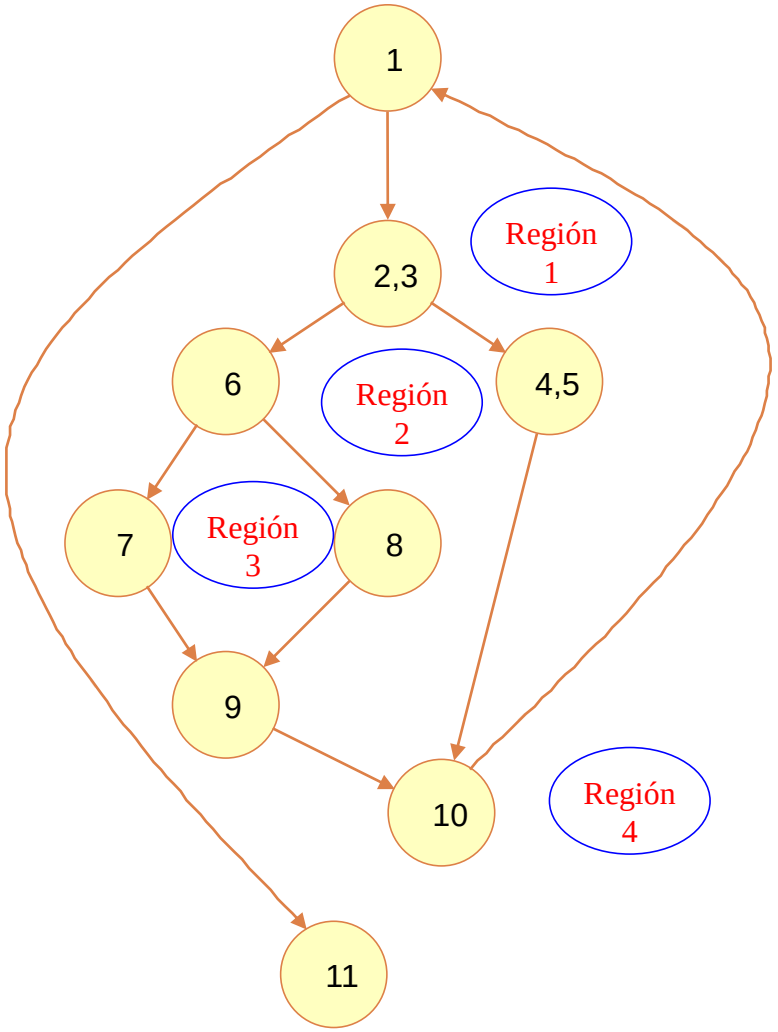
- Cálculo de la complejidad ciclomática
  - V(G) = a – n + 2, siendo a el número de arcos o aristas del grafo y n el número de nodos
  - V(G) = r, siendo r el número de regiones cerradas del grafo
  - V(G) = c + 1, siendo c el número de nodos de condición. (una condición de n arcos de salida se contabiliza como n-1)

Complejidad Ciclomática de McCabe

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

V(G)=R=4

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



En el número de regiones hay que tener presente que las fórmulas de McCabe sólo pueden aplicarse a grafos fuertemente conexos para los cuales siempre existe un camino entre cualesquiera dos nodos que elijamos. Esto no se verifica en los programas que tienen un nodo de inicio y otro de final por lo que para calcular las regiones debemos unir estos nodos o como alternativa contabilizar la región externa.

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 muy alta si no se debe a sentencias case-of

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 este camino de forma que cada variación se distinga en al menos 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 estas 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.

Ejemplo

CAMINOS INDEPENDIENTES

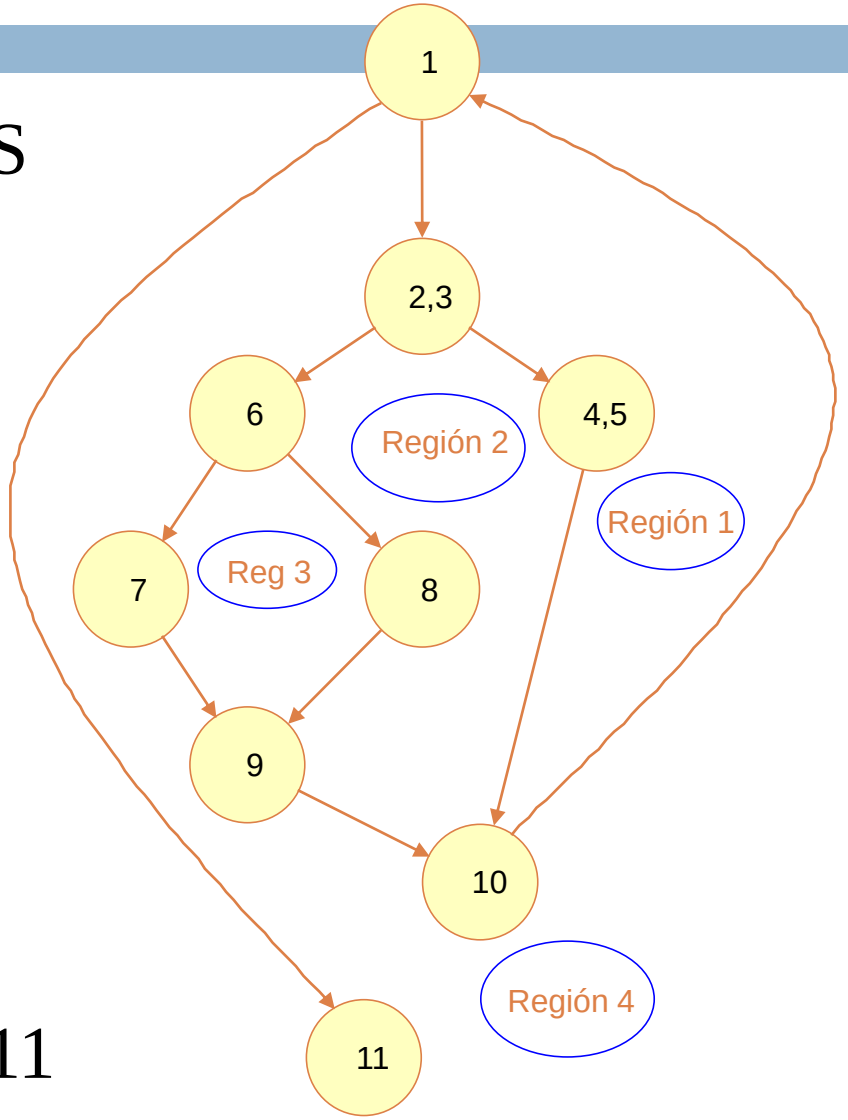
Camino 1: 1-11

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



Ejemplo completo

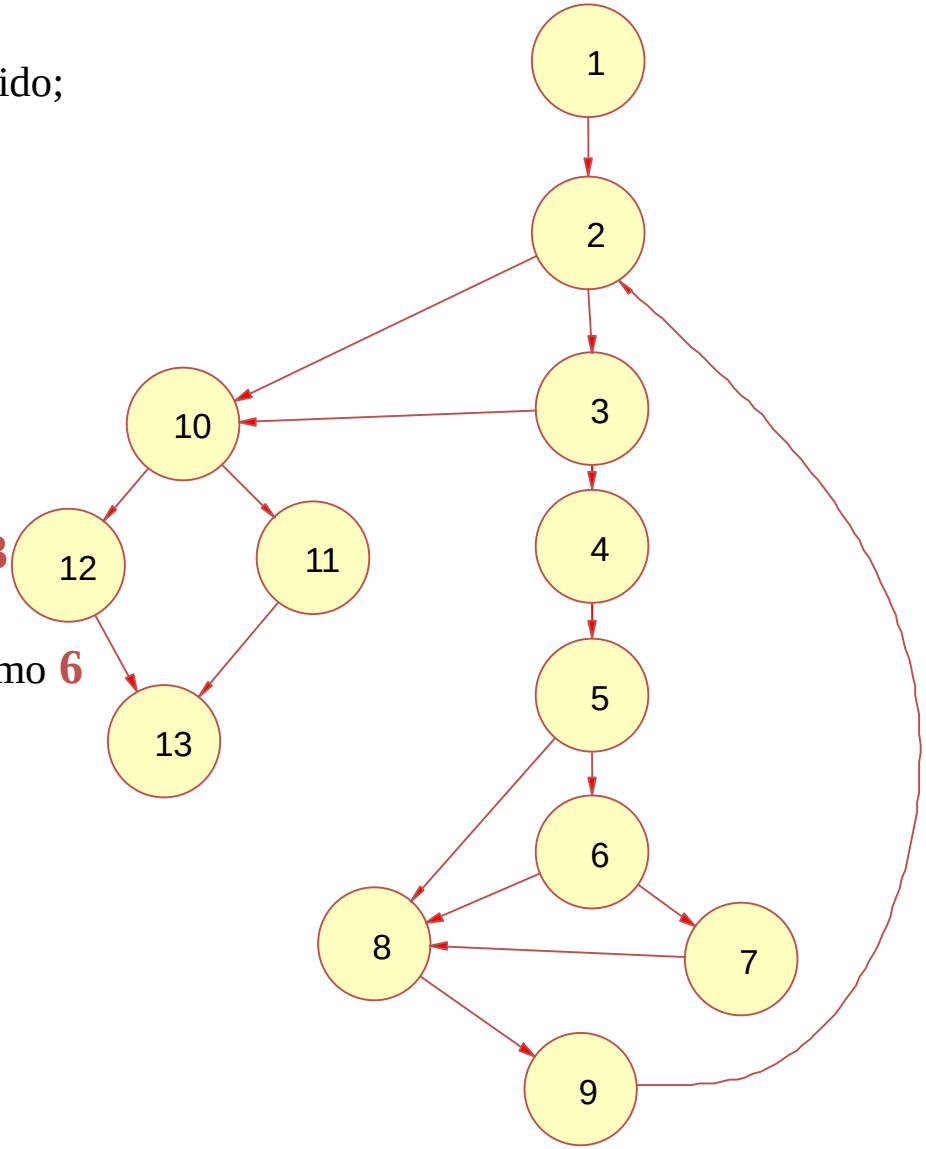
```
PROCEDURE media;
INTEFACE DEVUELVE media, total.entrada, total.válido;
INTERFACE ACEPTA  valor, mínimo, máximo
TYPE valor[1:100] ES ARRAY FLOAT;
TYPE media, total.entrada, total.válido ES FLOAT
TYPE mínimo, máximo, suma ES FLOAT
TYPE i ES ENTERO
i=1;
total.entrada=total.válido=0;
suma=0;
DO WHILE valor[i]<>-999   AND total.entrada<100
        total.entrada+=1;
        IF valor[i]>=mínimo and valor[i]<= maximo
          THEN total.válido+=1;
                  suma+=valor[i];
          ELSE ignorar
        ENDIF
        i+=1;
END DO
IF total.válido >0
        THEN media=suma/total.válido;
        ELSE media=-999;
ENDIF
```

Ejemplo completo

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

Ejemplo completo

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

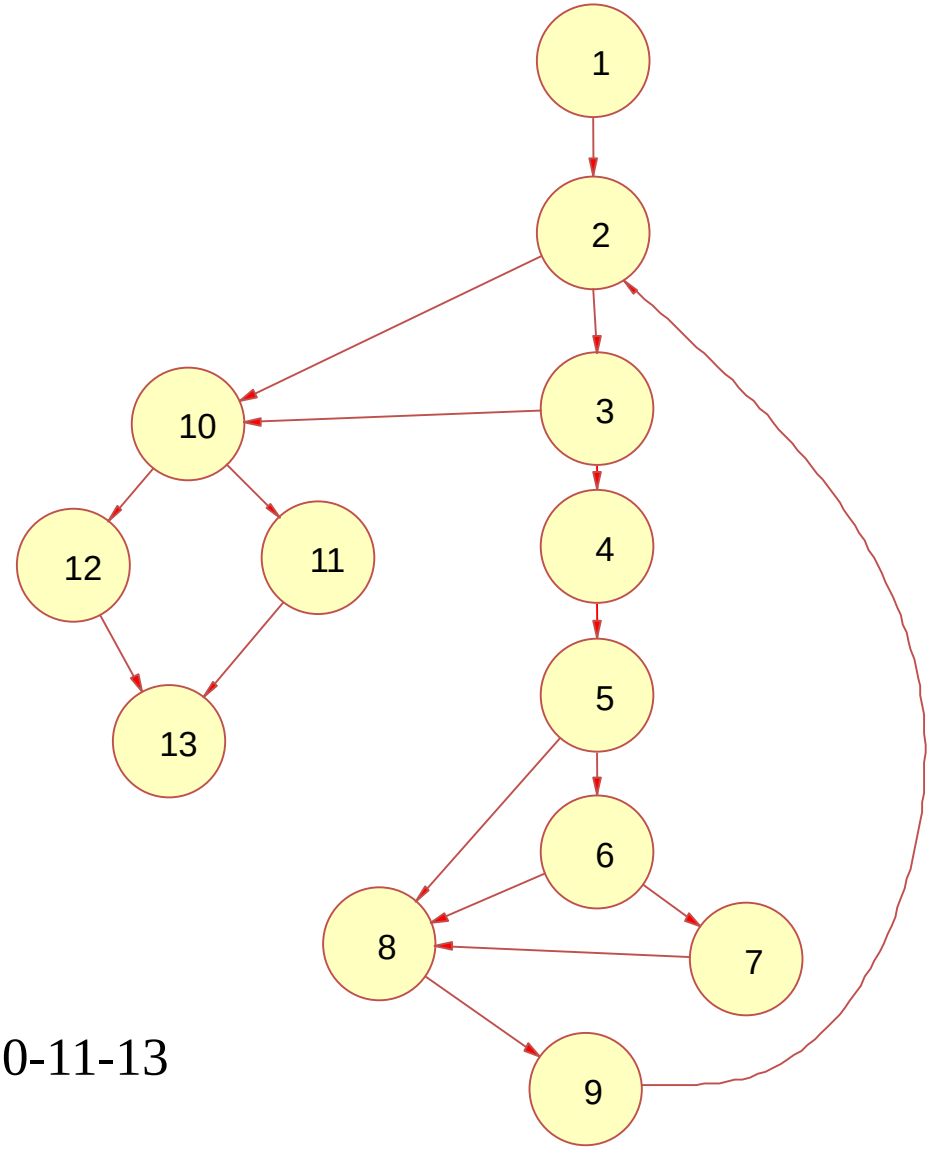


Ejemplo completo

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



Ejemplo completo

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

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

Valor(1) = dato válido

Valor(i) = -999, 2<=i<=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<=i<=100

Salida:

Media = 10

Ejemplo completo

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

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

Valor(1) = dato válido

Valor(i) = -999, 2<=i<=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<=i<=100

Salida:

Media = -999

Ejemplo completo

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

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

Valor(1) = dato válido

Valor(i) = -999, 2<=i<=100

Media = Valor(1)

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

Valor(1) = -999

Media = -999

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

Intentamos procesar 100 valores, con los 100 primeros <> -999

Media = Valor medio de los 99

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

Ejemplo completo

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

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

Valor(i)= dato válido, i<100

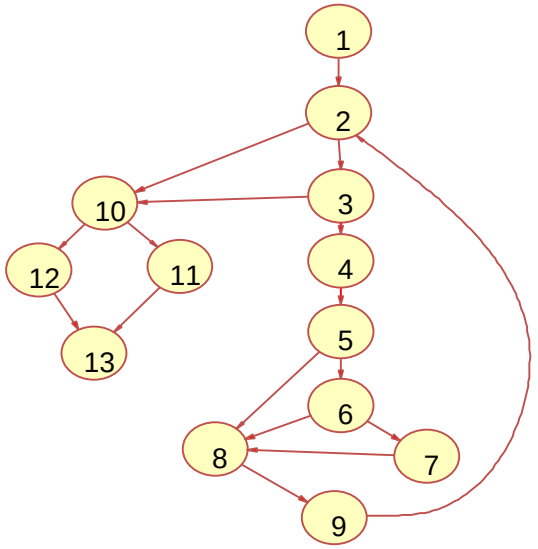
Valor(k) >máximo, k<i

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



Tema 6. Pruebas del software

Prueba Funcional. Caja Negra.

- Trataremos de encontrar errores en las siguientes categorías:
  1. Funciones incorrectas o ausentes
  2. Errores de interfaz
  3. Errores en estructuras de datos o acceso a BD
  4. Errores de rendimiento
  5. Errores de inicialización y terminación

Prueba funcional. Métodos de Caja Negra.

- La prueba funcional o de caja negra se centra en el estudio de las especificaciones.
  - Conocidas las entradas que salidas esperamos.

- No es posible la prueba exhaustiva.

Prueba funcional. Métodos de Caja Negra.

- Selección de casos de prueba
  - Enfoque sistemático
    - Búsqueda de buenos casos de prueba
      - Un buen caso de prueba es aquel que tienen una alta probabilidad de detectar un nuevo error
        - El caso ejecute el máximo 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

Prueba funcional. Métodos de 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 de Myers:
  - 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 consiste:
  - 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 5<N<7
    - Una clase válida y dos no válidas
  - R2.- Lista de valores de tamaño variable. Ej. Titulares de una Cuenta Bancaria (es posible:1,2,3 ó 4)
    - Una clase válida y dos no válidas
  - R3.- Situación del tipo debe ser o booleana. Edad es número
    - Una clase válida y una no válida
  - R4.- Valores admitidos con comportamientos distintos. Pago con tarjeta
    - Una clase válida por cada uno de ellos, y otra no válida
  - R5.- Si es necesario se subdividen las clases

Partición o clases de equivalencia

- Método de diseño de casos:
  - Identificar las clases de equivalencia
  - Crear los casos de prueba correspondientes.
    - Pasos a seguir:
      1. Numeramos las clases de equivalencia
      2. Hasta que todas las C.E.Válidas hallan sido cubiertas, especificar casos de prueba que cubran el mayor número posible de clases válidas no cubiertas
      3. Hasta que todas las C.E. No Válidas hallan sido cubiertas, especificar casos de prueba únicos por cada CENV sin cubrir.

Partición o clases de equivalencia

- *Pasos a seguir:*

  3.- Hasta que todas las C.E. No Válidas hallan sido cubiertas, especificar casos de prueba únicos por cada CENV sin cubrir.

  Contraejemplo: Introducir un valor entre 5 y 25

  200A

  El primer error enmascararía el segundo, por lo que un solo caso de prueba no llegaría

Partición o clases de equivalencia

- Ejemplo
  - Leemos de fichero dos tipos de registros:
    - Registro_Participantes:
      - Tipo_Registro: Definido en un dominio numérico sin signo, sólo toma el valor 0
      - Ganador: Definido en el dominio lógico, toma el valor Verdadero si el participante es ganador y Falso en caso contrario
      - Nombre: Cadena alfanumérica
    - Registro_Preguntas:
      - Tipo_Registro: Definido en un dominio numérico sin signo, sólo toma el valor 1
      - Pregunta: Definido en el dominio numérico, toma valores entre 1 y 99
      - Respuesta: Respuesta a la pregunta, dominio alfanumérico

Partición o clases de equivalencia

- Reglas de identificación de clases
  - R1.- Si se especifica un **rango** de valores para los datos de entrada, se creará una clase válida y dos no válidas
  - R2.- Si se especifica un **número de valores** (alternativas) posibles (por ejemplo el número de titulares de una cuenta bancaria ha de ser mayor que cero y menor que seis), se establece una clase válida y dos no válidas.
  - R3.- Si se especifica una situación del tipo debe ser o **booleana**, se identifica una clase válida y una no válida
  - R4.- Si se especifica un **conjunto de valores** admitidos, y el software trata de forma diferente cada uno de ellos, se establece una clase válida por cada uno de ellos, y otra no válida
  - R5.- En cualquier caso, si es necesario se **subdivide** cada clase válida en otras menores

Partición o clases de equivalencia

- Ejemplo

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

- Ejemplo

Casos de Prueba Válidos				
	0	TRUE	Pepe	1, 5, 8
	0	FALSE	Pepe	1, 7, 8
	1	50	Mi casa	2, 10,13
Casos de Prueba No Válidos				
	2	FALSE	Pepe	3, 5, 8
	a	FALSE	Pepe	4, 5, 8
	0		Pepe	1, 6, 8
	0	FALSE		1, 5, 9
	1	-1	Mi casa	2, 11, 13
	1	222	Mi casa	2, 12, 13
	1	50		2, 10, 14

Análisis de valores Límite (AVL)

- Técnica que complementa la de Particiones en Clases de Equivalencia con dos diferencias:
  - Seleccionamos elementos de la frontera de cada clase
  - Examinamos también el dominio de salida

Análisis de valores Límite (AVL)

- Reglas de construcción
  - R1.- Si una condición de entrada especifica un intervalo cerrado de valores [-1.0,1.0], examinaremos exactamente los valores extremos del intervalo -1.0 y 1.0, y los casos no válidos justo fuera del intervalo, esto es –1.1 y 1.1 (si sólo se admite un decimal)
  - R2.- Si la condición especifica un número de valores de entrada (i.e. El fichero tendrá de 1 a 255 registros), diseñaremos casos con los valores máximo y mínimo y uno más que el máximo y uno menos que el mínimo (i.e. 0, 1, 255, 256)

Análisis de valores Límite (AVL)

- Reglas de Construcción
  - R3.- Usar la regla 1 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 2 para cada condición de salida. Por ejemplo: El programa puede generar de 1 a 4 informes; trataríamos 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, etc.), nos centraríamos en el primer y último elemento.

Análisis de valores Límite (AVL)

- Consideraciones sobre las reglas de salida R3 y R4:
  - 1.- Los valores límite de entrada no siempre nos dan valores límites de salida.

    y=seno(x) x € [0,2PI] y € [-1,1]
  - 2.- No siempre se pueden obtener resultados fuera del rango de salida, pero es interesante considerar esta posibilidad

Conjetura de errores

- Se trata de hacer una lista de equivocaciones que pueda cometer un programador y diseñar un caso de prueba para cada elemento de la lista.
  - Valor 0 (entrada/salida)
  - En listas de valores concentrarse en el caso de que no haya valores, que haya 1 o que todos sean iguales
  - Intentar imaginar que se puede malinterpretar en las especificaciones
  - Prever 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 al sistema. Podemos apoyarnos en métodos estadísticos para conseguir la misma distribución de entrada o en reglas de comportamiento que deban 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

- Identificación de nodos y sus atributos
  - Es muy útil identificar los nodos de entrada y de salida.
- Establecer enlaces y sus pesos (p ej. Tiempo para la generación de un informe)
- Utilidad. Facilitan la identificación de casos de prueba para:
  - Identificación de bucles a probar. Prueba de Bucle.
  - Cobertura de nodo. Que no se han omitido nodos
  - Cobertura de enlace. Todas las relaciones se prueban separadamente basándose en sus propiedades.
    - Reflexiva.
    - Simetría.
    - Transitividad.

Pressman 5º Ed. pp. 295-296

Métodos de Caja Negra basados en grafos

- Son posibles distintos modelos
  - Modelo de Estados Finitos
  - Modelo de Transacción
  - Modelo de Flujo de Datos.
- Todos son modelos de Caja Negra
  - No modelan como está programado el software
    - No son los Estados del software, ni sus transacciones programadas ni sus flujos de datos.
  - Modelan su comportamiento visible, pantallas por las que pasa, o el contexto del problema, cómo se hace una reserva (con software o sin el).

Pressman 5º Ed. pp. 295-296

Métodos de Caja Negra basados en grafos

- Modelado de estado finito:
  - Los nodos son estados del software observables por el usuario y los enlaces representan las transiciones que ocurren para moverse de un estado 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 PANTALLAS
      - Las transiciones reflejan ACCIONES que cambian la pantalla

Pressman 5º Ed. pp. 295-296

Métodos de Caja Negra basados en grafos

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

Pressman 5º Ed. 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.

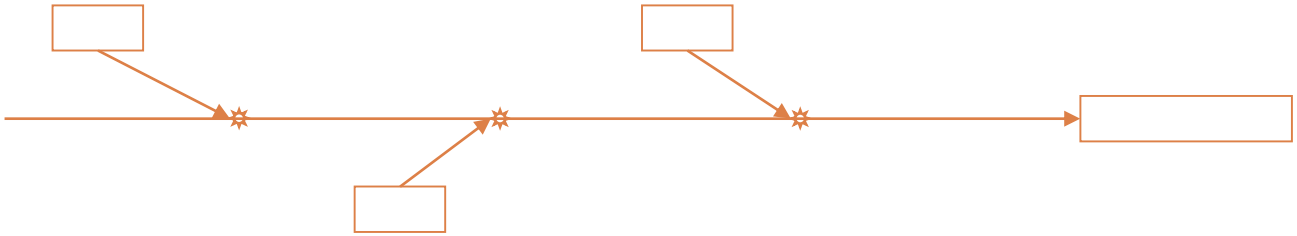
    - Ejemplo: Cálculo de hipoteca. A partir del valor de hipoteca y plazo de amortización resultan unos gastos, mensualidades e intereses que pueden llevar a cambiar el valor o el plazo

Pressman 5º Ed. pp. 295-296

Tema 6. Pruebas del software

Enfoque recomendado

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

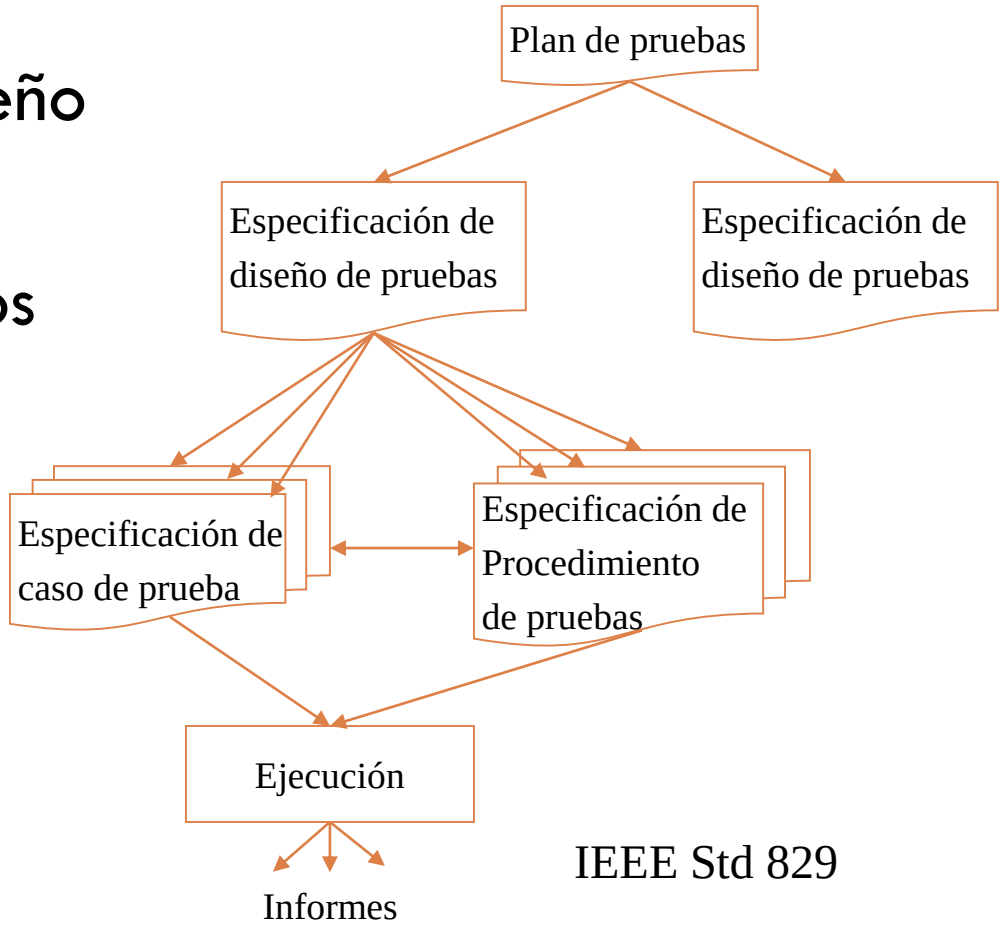


2. Identificar clases de equivalencia válidas y no validas para la entrada y la salida
3. En todos los casos usar AVL para añadir casos de prueba
4. Utilizar conjetura de errores para añadir nuevos casos
5. Ejecutar los casos generados hasta el momento y analizar la cobertura obtenida
6. Elegir casos precisos de caja blanca si en 5 no se ha cumplido el criterio de cobertura lógica elegido.

Tema 6. Pruebas del software

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
  - Señalar el enfoque, los recursos y el esquema de las actividades de prueba
  - Los elementos y características a probar y a no probar
  - Las actividades de prueba,
  - El personal responsable
  - Los riesgos asociados
- Especificación del diseño de prueba
- Especificación de casos de prueba
- Especificación de procedimiento de pruebas
- Ejecución.

Documentación de diseño de pruebas

- Plan de pruebas
- Especificación del diseño de prueba: Detalla el anterior plan de pruebas
  - Características de los elementos a probar
  - Detalles como Técnicas y métodos de análisis de resultados.
  - Identificación de pruebas (un Id para cada prueba)
    - Describe los casos de prueba a utilizar (un Id para cada CP)
    - Procedimientos a seguir
  - Criterios de paso/fallo de la prueba
- Especificación de casos de prueba
- Especificación de procedimiento de pruebas
- Ejecución.

Documentación de diseño de pruebas

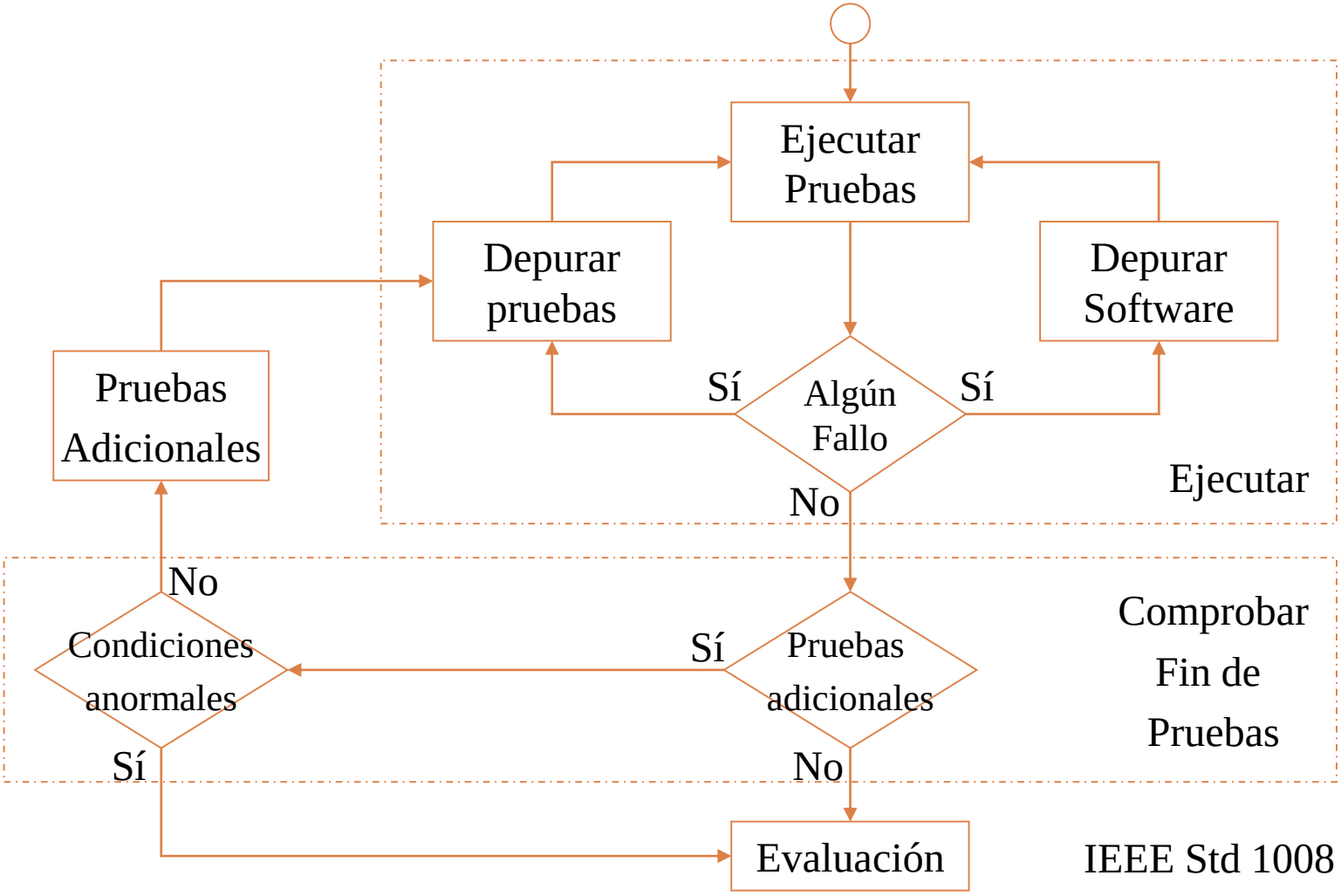
- Plan de pruebas
- Especificación del diseño de prueba
- Especificación de casos de prueba: Define con detalle los casos de prueba mencionados en el punto anterior.
  - Elementos Software y características a probar
  - Especificación de entradas requeridas, y su sincronización)
  - Especificación de las salidas (y sus características: tiempo de respuesta)
  - Necesidades del entorno (Hardware, software, personal...)
  - Requisitos especiales
  - Dependencias entre casos.
- Especificación de procedimiento de pruebas
- Ejecución.

Documentación de diseño de pruebas

- Plan de pruebas
- Especificación del diseño de prueba
- Especificación de casos de prueba
- Especificación de procedimiento de pruebas: Indican como proceder en 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.

Tema 6. Pruebas del software

Ejecución de las pruebas

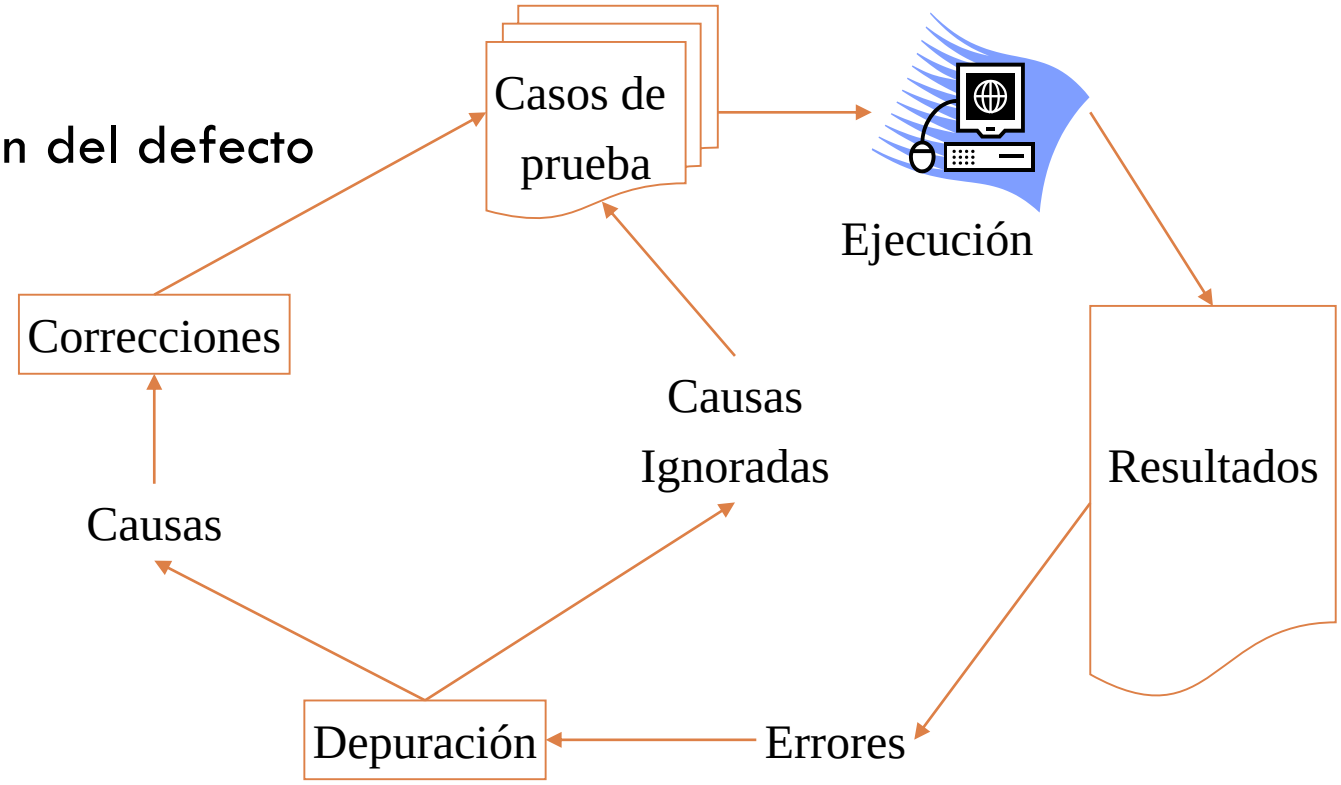


Documentación de la ejecución

- Histórico de pruebas
  - Registro cronológico de la ejecución.
  - Elementos probados y su entorno
    - Fecha
    - Referencia al informe de incidencia si lo hay
- Informe de Incidencia
  - Resumen del incidente y analiza su 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 o diseño.
  - Valoración de la cobertura lógica alcanzada.
  - Resumen de las actividades (Detalles de recursos).

Depuración

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



Consejos de depuración

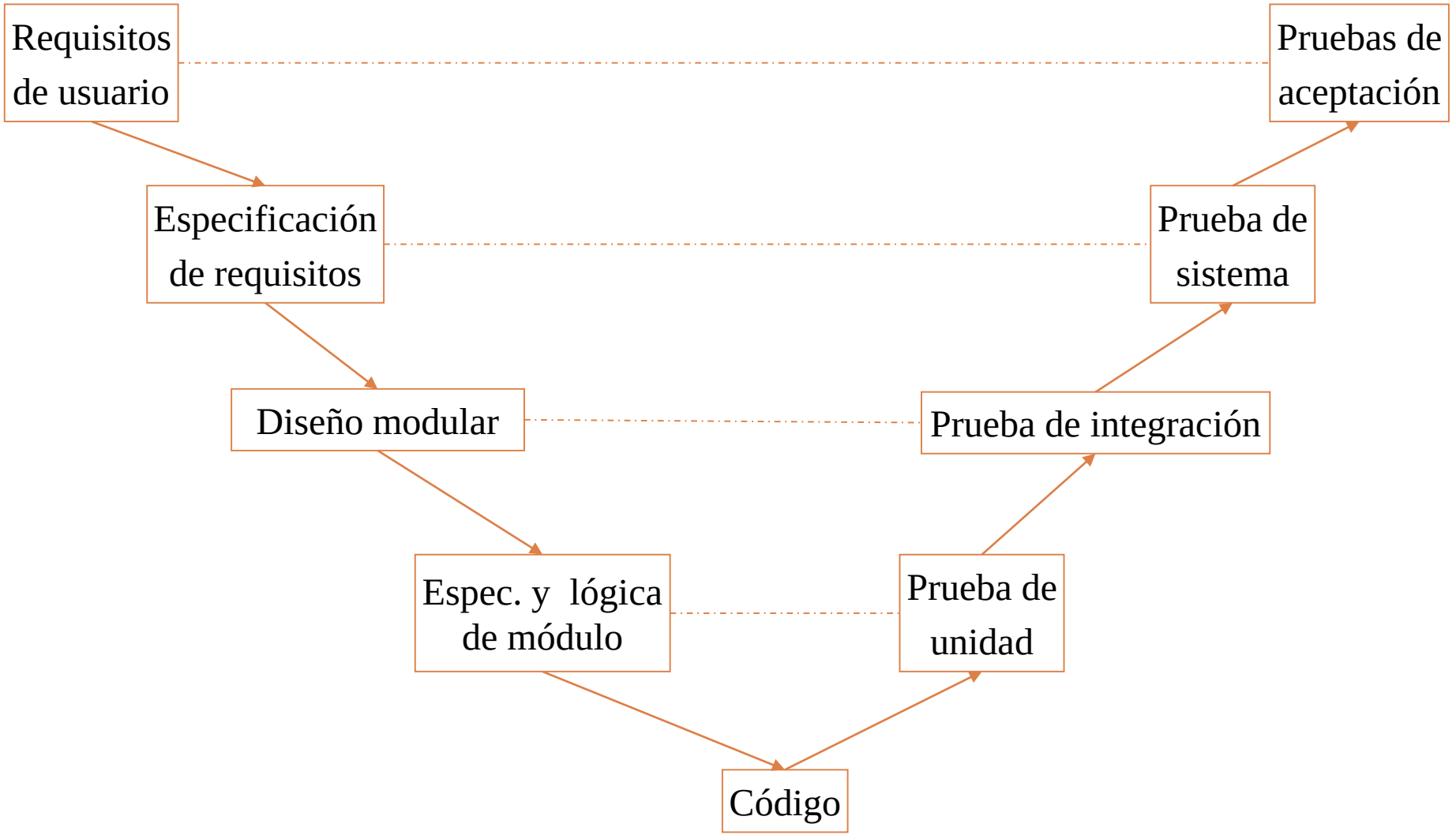
- Localización del error
  - Proceso mental de solución de un problema.
    - Analizar la información.
    - No explorar aleatoriamente.
    - No experimentar cambiando el programa
  - Usar herramientas de depuración sólo como recurso secundario
  - Al llegar a un punto muerto
    - pasar a otra cosa
    - Describir el problema a otra persona
  - Se deben atacar los errores individualmente
  - Se debe fijar la atención también en los datos y no sólo en la lógica del proceso.

Consejos de depuración

- 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 nuevos defectos
  - La corrección debe situarnos temporalmente en la fase de diseño

Tema 6. Pruebas del software

Estrategia de aplicación de pruebas



Tema 6. Pruebas del software

Pruebas en desarrollos OO

- Técnicas de caja negra: Totalmente válidos
  - Análisis de valores límite, tratamiento de combinaciones de entrada, conjetura de errores.
  - Diseñar los casos de prueba basándose
    - Datos y eventos de los escenarios de los casos de uso.
    - Flujos alternativos, tratamientos de error y excepciones.
- Técnicas de caja blanca.
  - Quedan confinadas a su aplicación en los métodos de las clases.

Pruebas en desarrollos OO

- Diseño de pruebas: Distinguir criterios según nivel
  - Pruebas de unidad:
    - Análisis por clase de equivalencia y AVL para entradas y salidas de métodos de clase
  - Pruebas de integración
    - Clases independientes o con un funcionamiento dependiente de pocas clases para ir añadiendo el resto
    - Probar hilos de clases que colaboran en una función
  - La filosofía de la programación orientado a objetos cambian:
    - Herencia: probar un método en la clase padre no garantiza su funcionamiento en clases hijas
    - Polimorfismo: un único método puede tener implementaciones diferentes en función de la clase en la que se use.