Entornos de desarrollo

3

Pruebas del software

IES Nervión
Miguel A. Casado Alías

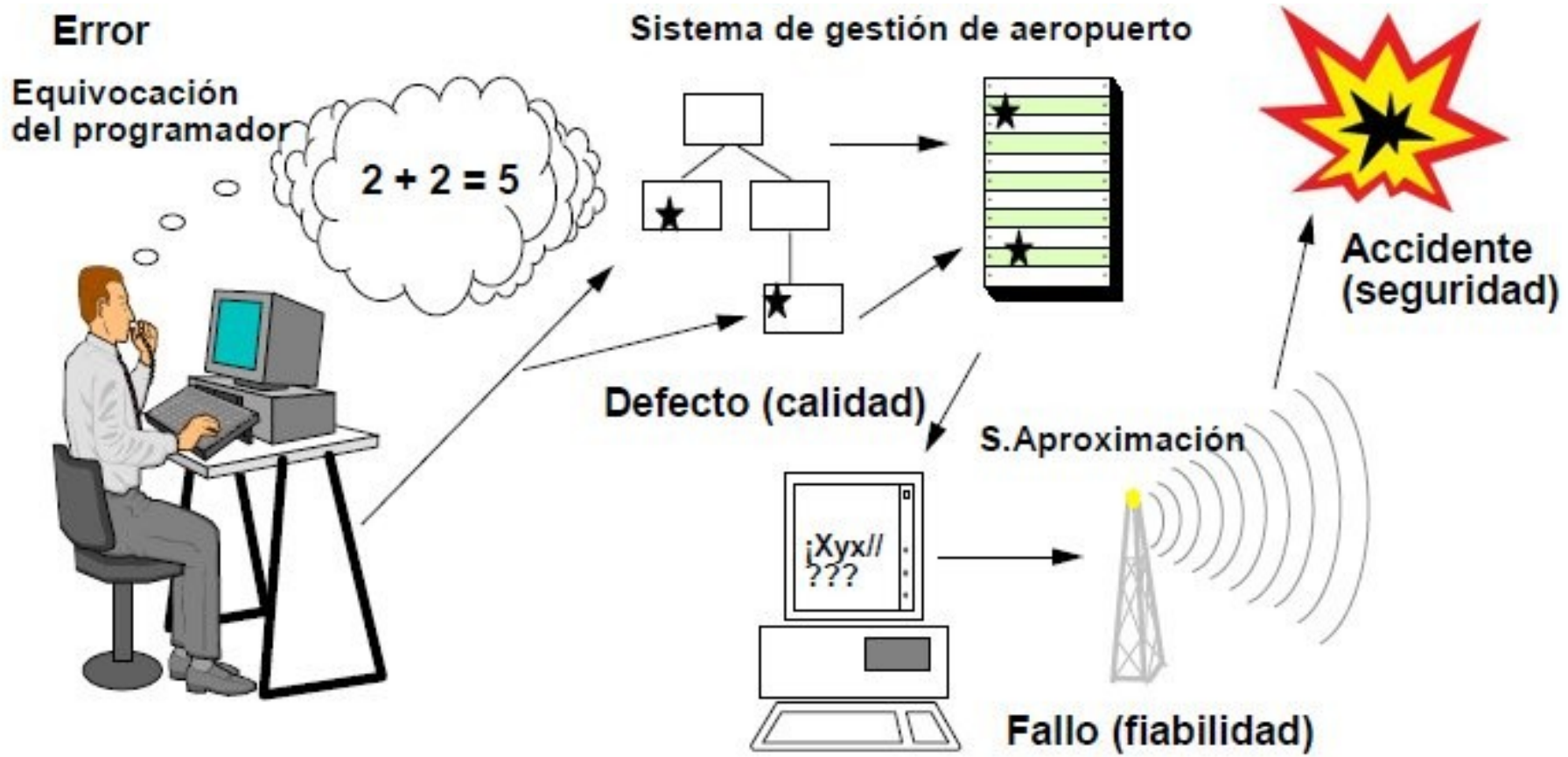
¿Qué son las pruebas del software?

- Son ejecuciones o ensayos de funcionamiento posteriores a la terminación del código del software.
- Constituyen un método para poder:
  - **Verificar** el software (comprobar que los productos software son correctos)
  - **Validar** el software (comprobar si una aplicación satisface los requisitos marcados por el usuario)

Definiciones

- **Prueba**: Ejecución del programa en circunstancias controladas para detectar errores.
- **Caso de prueba**: Conjunto de datos de entrada, condiciones de ejecución y resultados esperados.
- **Defecto (*bug*)**: Parte del software mal construida.
- **Fallo**: Incapacidad de realizar las funciones requeridas cumpliendo los requisitos de rendimiento.
- **Error**:
  - Diferencia entre el resultado esperado y el producido
  - Acción humana que provoca un resultado incorrecto

Ejemplo relación “error - defecto - fallo”



Error (del programador) —Se plasma→ Defecto (en el software) —Da lugar a→ Fallo (el sistema no se comporta como debería) —Provoca→ Efectos negativos (dependiendo de la criticidad del sistema)

Filosofía de las pruebas

- Las características del software hacen difícil su prueba puesto que:
  - Es imposible probar todas las posibilidades de su funcionamiento.
  - Prejuicios: creencia de que los defectos son producto de una negligencia

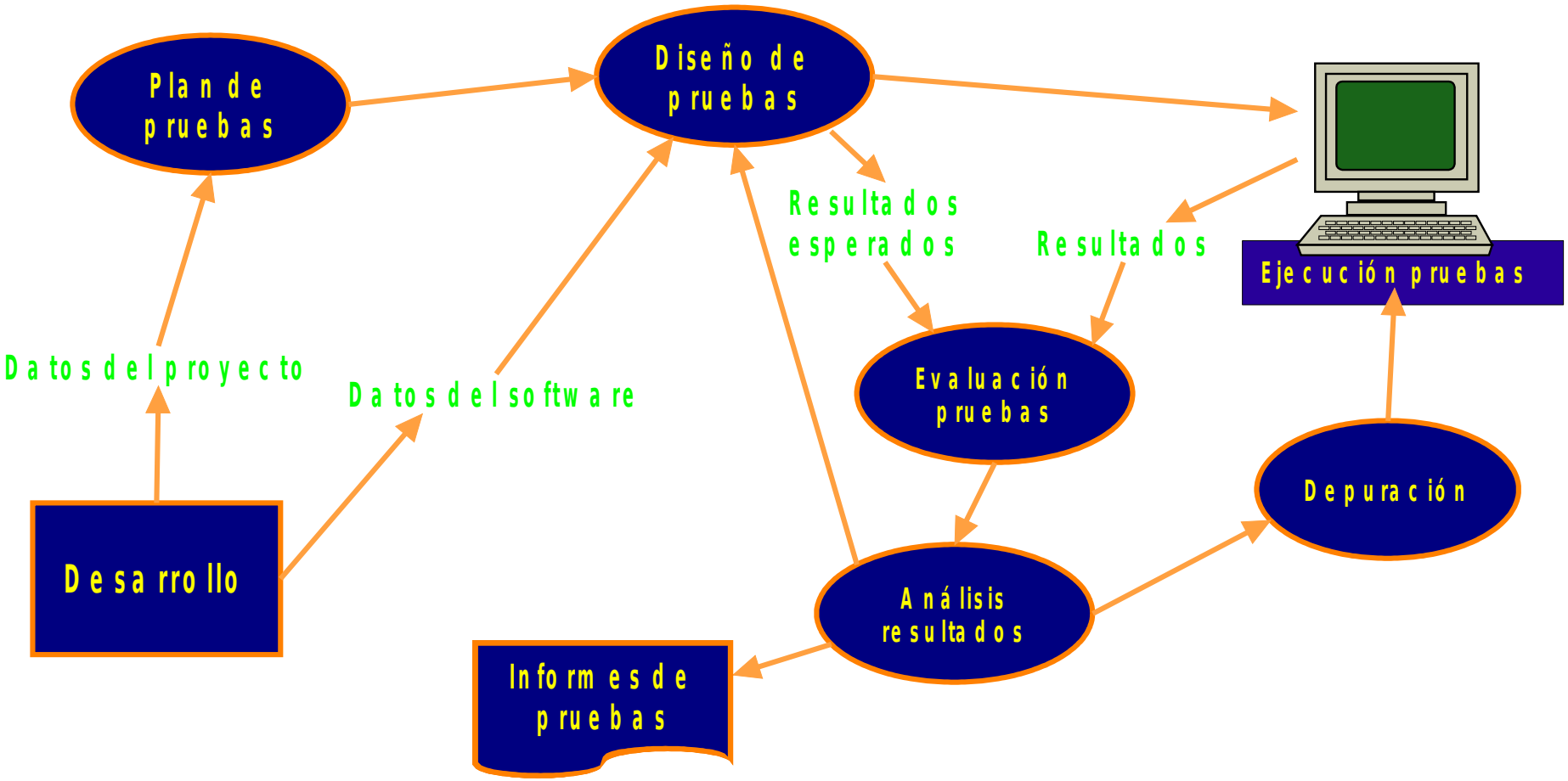
¡¡ENCONTRAR UN DEFECTO ES UN ÉXITO!!

Recomendaciones

- Los casos de prueba indicarán los resultados esperados
- Las pruebas debe hacerlas el peor enemigo del programador
- Estudiar despacio los resultados de las pruebas
- Incluir datos de entrada no válidos y no esperados
- Objetivos:
  - Probar si hace lo que debe
  - Probar si no hace lo que no debe
- Documentar los casos
- Siempre hay defectos que localizar, por ello las pruebas exigen dedicación (40% del esfuerzo de desarrollo => PRUEBAS)
- Los defectos suelen agruparse (donde hay un defecto hay otros)
- Las pruebas exigen creatividad “si cree que la construcción del programa ha sido difícil, aún no ha visto nada”

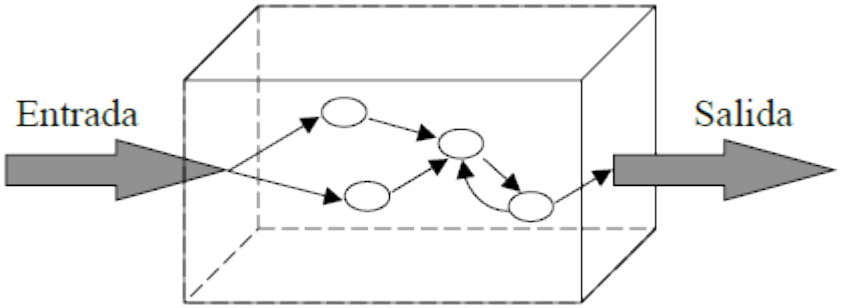
Por lo tanto la filosofía más adecuada para las pruebas consiste en planificarlas y diseñarlas de forma sistemática, para detectar el máximo número y variedad de defectos.

Proceso de prueba

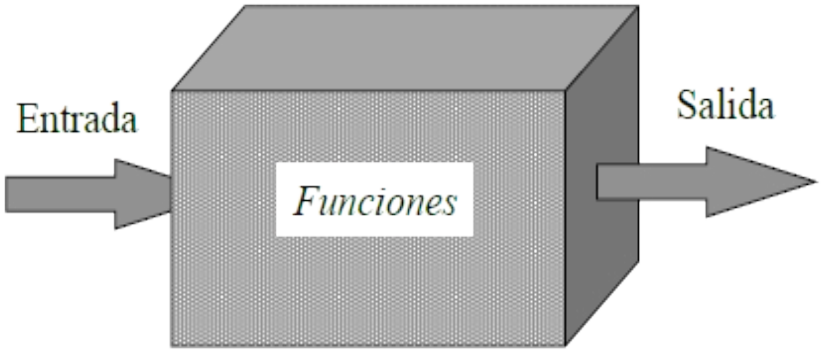


Diseño de casos de prueba

- Enfoque estructural (caja blanca)
  - Se centra en la estructura interna del programa
- Enfoque funcional (caja negra)
  - Se centra en las entradas y salidas del programa
- Enfoque aleatorio
  - Basado en modelos (utilizar modelos que representen las posibles entradas al programa para crear a partir de ellos casos de prueba)



Caja Blanca



Caja Negra

Pruebas estructurales

- Es de gran ayuda dibujar el diagrama de flujo del programa.

Diagrama de flujo

- Estructura alternativa SI

```
SI (A < 40)
        Proceso 1
FIN_SI
```

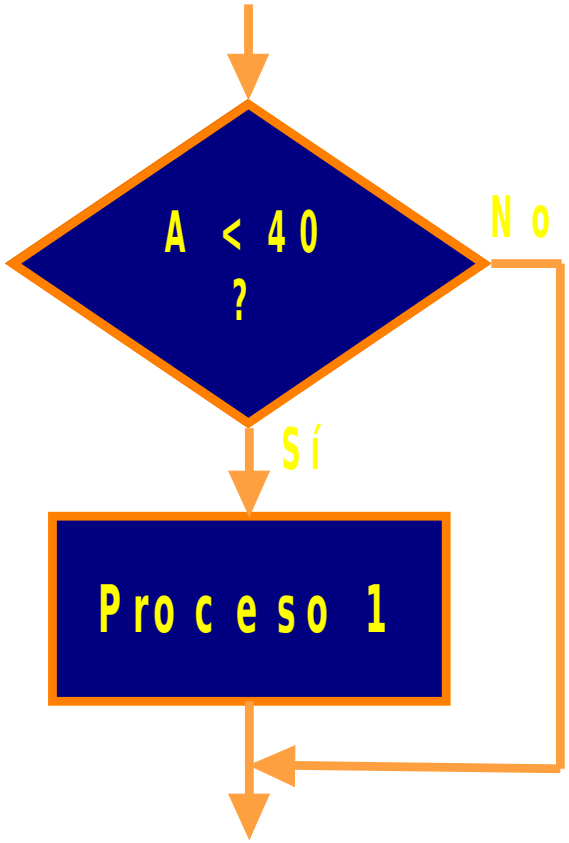


Diagrama de flujo

- Estructura alternativa

```
SI EN_OTRO_CASO
        SI (A < 40)
              Proceso 1
        EN_OTRO_CASO
              Proceso 2
        FIN_SI
```

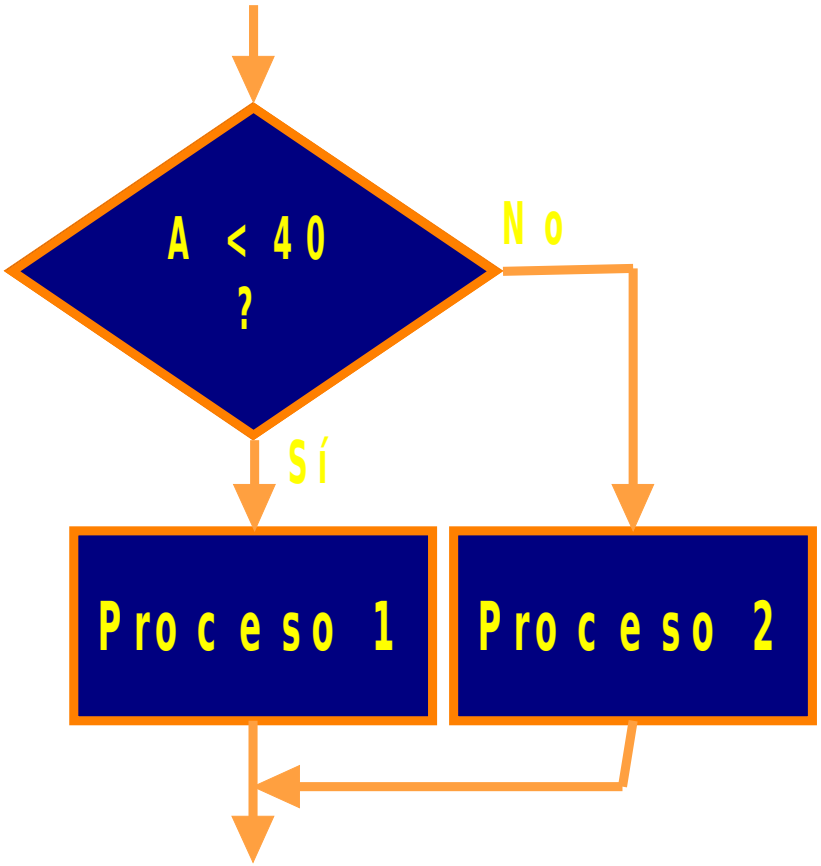


Diagrama de flujo

- Estructura repetitiva con condición al principio

```
MIENTRAS (Opcion <> ‘N’)
    Proceso 1
FIN_MIENTRAS
Proceso 2
```

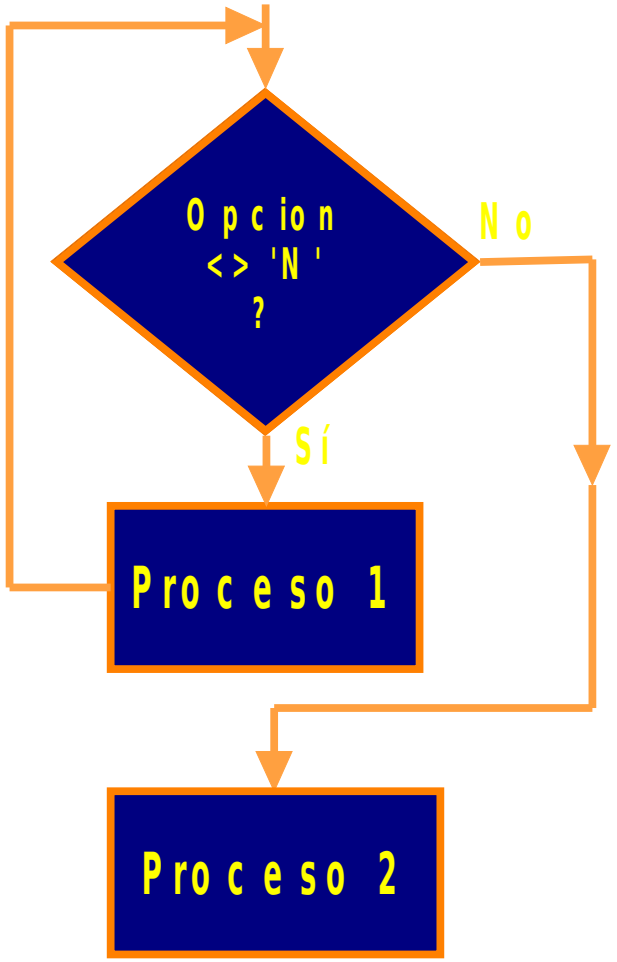


Diagrama de flujo

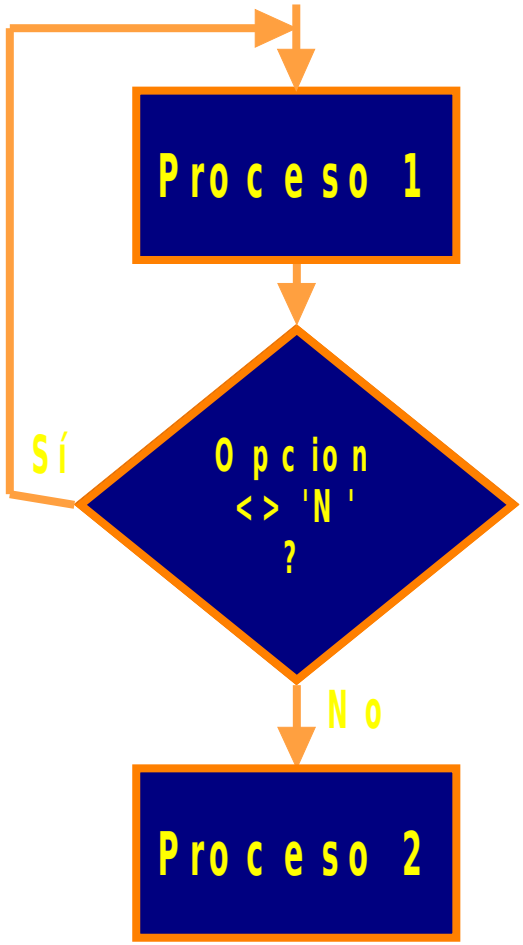
- Estructura repetitiva con condición al final

REPETIR

    Proceso1

MIENTRAS (Opcion <> ‘N’)

Proceso 2



Criterios de cobertura

- Ante la imposibilidad de probar todos los casos posibles, debemos procurar escoger un criterio para cubrir el máximo número de ellos que nuestros recursos permitan.
- Criterios ordenados de menor a mayor coste, y de menor a mayor seguridad de detección de defectos:
  - Cobertura de sentencias (generar casos de prueba para que cada sentencia se ejecute al menos una vez)
  - Cobertura de decisiones (que haya un resultado V y otro F)
  - Cobertura de condiciones (&& y || => Al menos un V y otro F)
  - Criterio de decisión/condición (hacer que la cobertura de condiciones incluya la cobertura de decisiones)
  - Criterio de condición múltiple (todas las combinaciones posibles de V y F)

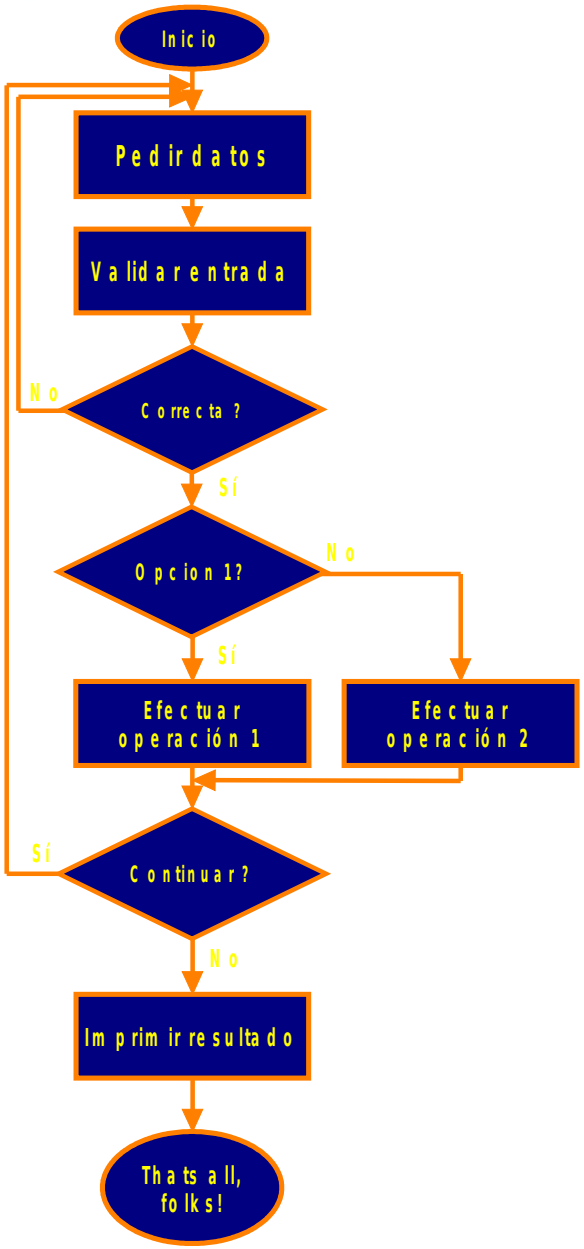
Cobertura de caminos

- El criterio más elevado es la cobertura de caminos
  - Todos los “caminos” que llevan del principio al fin del módulo deben recorrerse una vez (impracticable)
  - Los caminos de prueba evitan las iteraciones (por seguridad tres)
  - Se usan grafos (algorítmica)

Complejidad ciclomática de McCabe

- Indica el número de caminos independientes de un grafo
  - V(G) = a – n + 2 (a: aristas, n: nodos)
  - V(G) = r (r: regiones)
  - V(G) = c + 1 (c: nodos de condición)
- Una vez conocido el número, se identifican y se diseñan casos de prueba que los recorran

Ejemplo C. C. McCabe



- V(G) = a – n + 2

  V(G) = 12 -10 + 2 = 4
- V(G) = r

  V(G) = 4
- V(G) = c + 1

  V(G) = 3 + 1 = 4

Conclusiones C.C. McCabe

- V(G) nos indica el número mínimo de casos de prueba
- Si V(G) > 10 el módulo es muy complejo y tendrá más defectos

Pruebas funcionales

- Nos basamos en la funcionalidad del módulo, sus entradas y sus salidas
- Sabemos que es imposible probar todas las posibilidades, por tanto elegiremos valores representativos de categorías de datos de entrada

Pruebas funcionales: elección de casos de prueba

- Un caso de prueba está bien elegido si:
  - Reduce el número de otros casos necesarios para que la prueba sea razonable.
  - Cubre un conjunto extenso de otros casos posibles

Clases de equivalencia

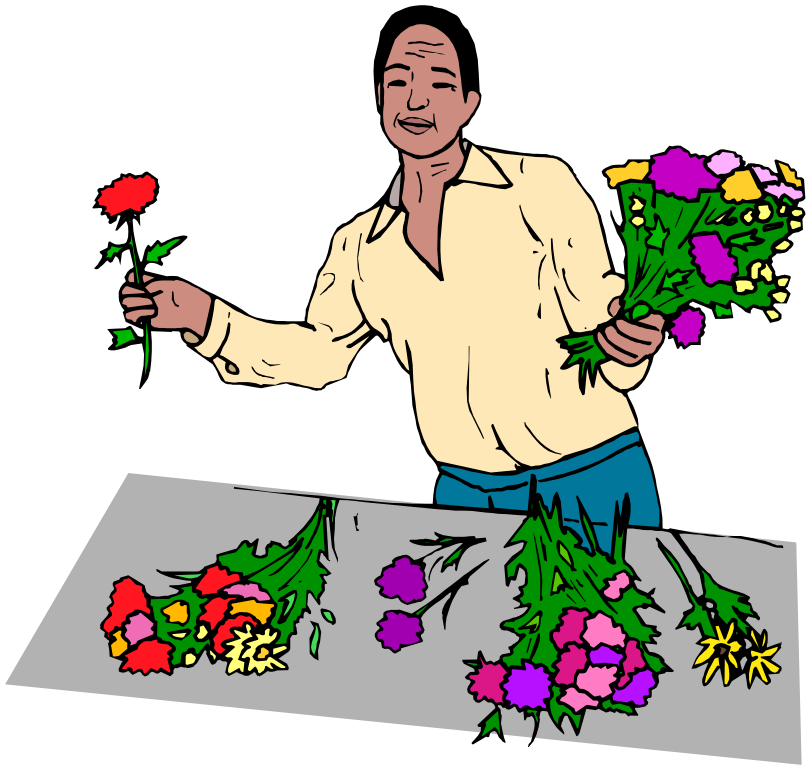
- Se dividen los posibles valores de entrada en clases de equivalencia
- Cada caso debe cubrir el máximo número de entradas
- La prueba de un valor representativo de una clase permite suponer que el resto de valores de dicha clase se comportarán igual

Clases de equivalencia: aplicación

- Identificación de clases de equivalencia
- Creación de los casos de prueba correspondientes

Identificación de clases

- Estudiar las condiciones de entrada (restricciones de formato, etc...)
- Considerar los datos válidos y los no válidos:
  - Rangos
  - Condiciones “debe ser”
  - Conjunto de valores admitidos
  - Elementos con tratamiento diferenciado



Obtención de casos de prueba

- Identificar cada clase con un número
- Escribir casos que incluyan tantas clases válidas como sea posible hasta que todas estén contempladas
- Escribir un caso por cada clase no válida

Ejemplo

- Cada autobús se identifica con un número positivo de tres cifras.
- La primera cifra no puede ser cero
- Las asignaciones de los autobuses pueden ser “S” (servicio), “L” (línea) o “N” (ninguna).
- El número de la línea es un entero positivo entre 1 y 100

Ejemplo

Condición	Nº	Clase	Tipo
Cód. autobús	1	99 < C_auto < 1000	V
	2	C_auto < 100	N
	3	C_auto >= 1000	N
Asignaciones	4	Asig = “S”	V
	5	Asig = “L”	V
	6	Asig = “N”	V
	7	Ninguna de las anteriores	N
Numero línea	8	0 < N_linea <= 100	V
	9	N_linea < 1	N
	10	N_linea > 100	N

Ejemplo

- Casos válidos:
  - 300 , S , 15 (Clases 1, 4, 8)
  - 257, L , 28 (Clases 1, 5, 8)
  - 450 , N , 40 (Clases 1, 6, 8)
- Casos no válidos:
  - 45 , S , 17 (Clase 2)
  - 1245 , S , 23 (Clase 3)
  - 343 , F , 40 (Clase 7)
  - 123 , L , 0 (Clase 9)
  - 200 , S , 500 (Clase 10)

Análisis de valores límite

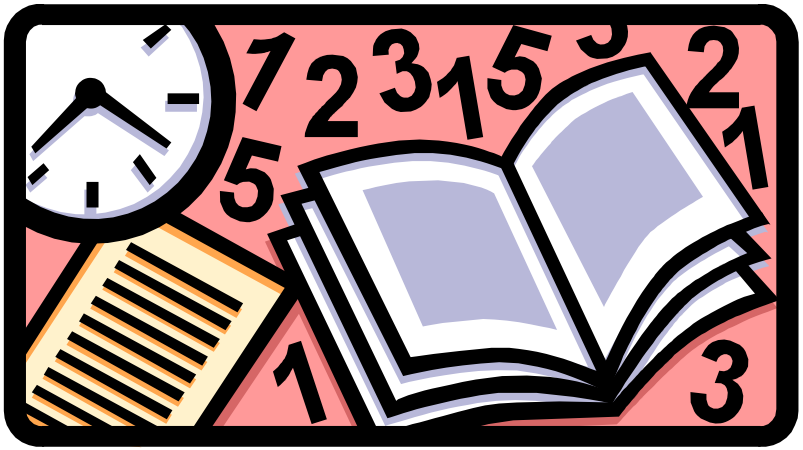
- La mayor parte de los errores se producen al tratar los datos límite
- Es una técnica complementaria a las clases de equivalencia
- En lugar de elegir cualquier elemento de la clase, escogeremos los marginales
- Se consideran también los límites de las salidas del módulo

Conjeturas de errores

- Se prueban los elementos que suelen ser problemáticos
  - El 0
  - Ningún dato, uno sólo, todos iguales
  - Especificaciones ambiguas
  - Usuario tonto
  - Usuario malintencionado

Pruebas aleatorias

- Se generan valores de forma aleatoria para probar el módulo
- Es más sencilla
- Es menos fiable
- Hay herramientas generadoras



Depuración

- Es el proceso de analizar y corregir los defectos que se sospecha que contiene el software. Consejos:
  - Analizar la información y pensar
  - Al llegar a un punto muerto, pasar a otra cosa. También ayuda el hecho de describir el problema a otra persona
  - Usar herramientas de depuración sólo como apoyo
  - No experimentar cambiando el programa sin pensar
  - Se deben atacar los errores individualmente
  - Se debe fijar la atención también en los datos
  - Donde hay un defecto, suele haber más
  - Cuidado con crear nuevos defectos

Estrategia de aplicación de las pruebas

- Se comienza en la prueba de cada módulo, que normalmente la realiza el propio personal de desarrollo en su entorno (**pruebas unitarias**)
- Los módulos probados se integran para comprobar sus interfaces en el trabajo conjunto (**prueba de integración**)
- El software totalmente ensamblado e integrado con el resto del sistema (p.ej: interfaces electrónicas, elementos mecánicos...) se prueba como un conjunto para comprobar si cumple o no tanto los requisitos funcionales como los requisitos de rendimiento, seguridad, etc. (**prueba del sistema**)
- Por último, el producto final se pasa a la **prueba de aceptación** para que el usuario compruebe en su propio entorno de explotación si lo acepta como está o no.