Calidad y pruebas de Software

Sesión 8- Pruebas de Software

Ing. Fany Sobero Rodriguez

Temario

Pruebas de Software

1. Introducción
2. Definición de pruebas de software
3. Principios de las pruebas de software
4. Proceso de las pruebas de software
5. Técnicas de prueba
6. Diseño de casos de prueba

Introducción

- Total de recursos empleados en pruebas:
  - 30% a 90% [Beizer 1990]
  - 50% a 75% [Hailpern & Santhanam, 2002]
  - 30% a 50% [Hartman, 2002]
  - Mercado de herramientas: $2,6 billion
- Coste por infraestructura inadecuada:
  - Transporte y manufactura: $1,840 billion
  - Servicios financiero: $3,342 billion

Introducción

- Costes de reparación en función del instante en el ciclo de vida [Baziuk 1995]
  - Requisitos: x 1
  - Pruebas de sistema: x 90
  - Pruebas de instalación: x 90-440
  - Pruebas de aceptación: x 440
  - Operación y mantenimiento: x 880

- Fuente: The economic impact of inadequate infrastructure for software testing. NIST Report – May 2002

Introducción

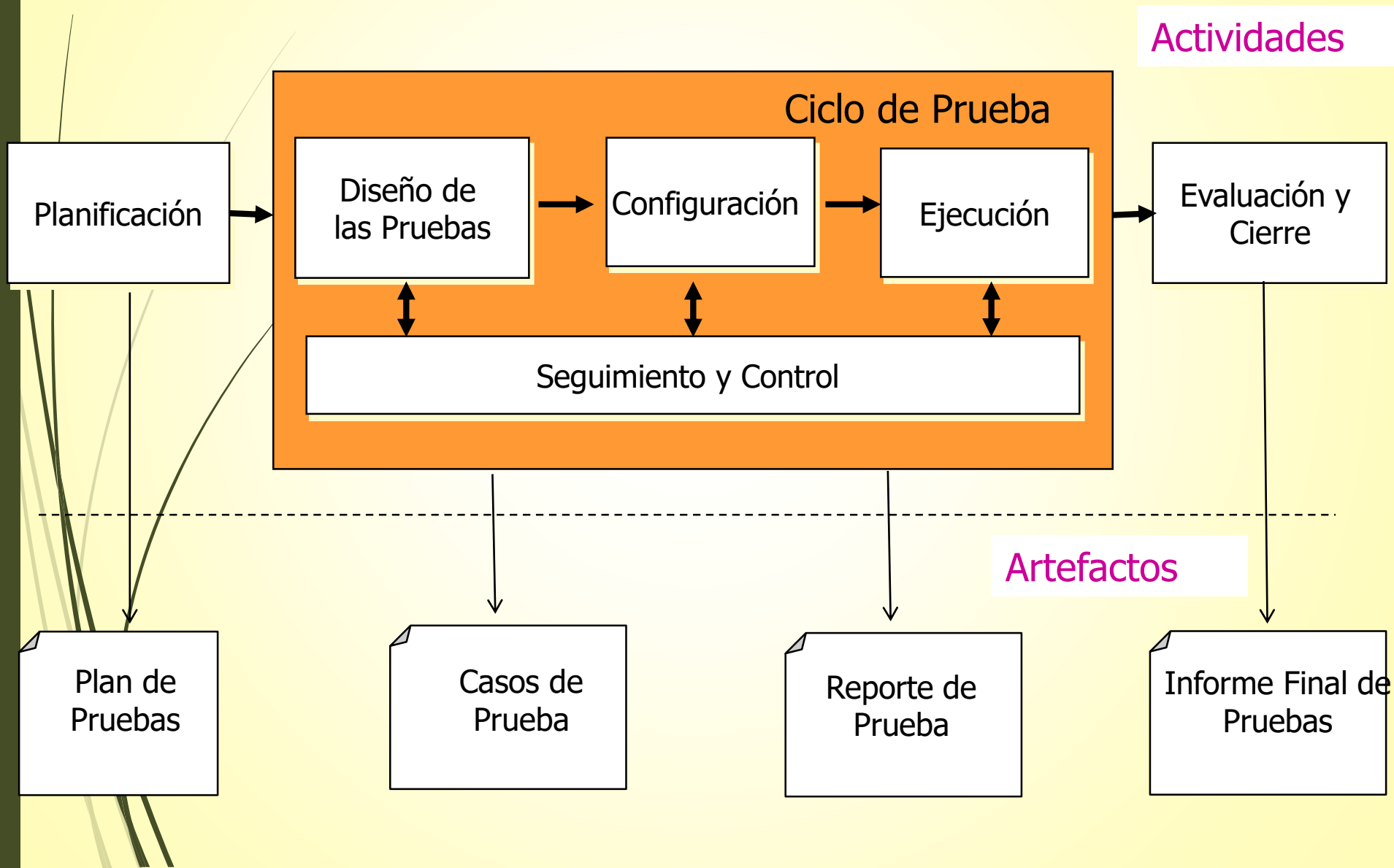
- No es una actividad secundaria:
  - 30-40% del esfuerzo de desarrollo
  - En aplicaciones críticas (p.ej. control de vuelo, reactores nucleares),

    ¡de 3 a 5 veces más que el resto de pasos juntos de la ingeniería del software!
  - El coste aproximado de los errores del software (*bugs*) para la economía americana es el equivalente al 0,6% de su PIB, unos 60.000 millones de dólares

Definición de Pruebas de Software

- **Definición 1:**
  - La prueba (testing) es el proceso de ejecutar un programa con la intención de encontrar fallos [Glenford J. Myers]
  - Un buen caso de prueba es el que tiene alta probabilidad de detectar un nuevo error
  - Un caso de prueba con éxito es el que detecta un error nuevo
- **Definición 2 :**
  - Una investigación técnica del producto bajo prueba …para proporcionar a los interesados (stakeholders) …información relacionada con la calidad [Cem Kaner]

Proceso de Pruebas de Software



Principios de la pruebas de Software

- Empezar las pruebas en módulos individuales y avanzar hasta probar el sistema entero.
- No son posibles las pruebas exhaustivas.
- Deben realizarse por un equipo independiente al equipo de desarrollo.

Principios de la pruebas de Software

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

Pruebas exhaustivas

- Para probar completamente un sistema se deben ejercitar todos los caminos posibles del programa a probar.
- Myers mostró en 1979 un programa que contenía un loop y unos pocos if, este programa tenia 100 trillones de caminos, un tester podía probarlos todos en un billón de años.

Psicología de las pruebas

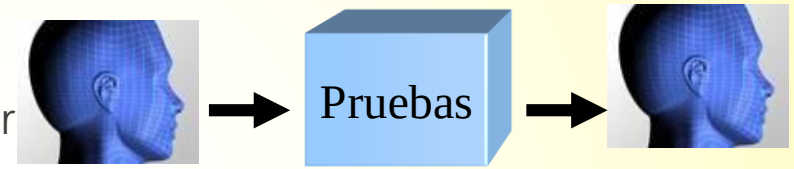
- El autor de un programa tiende a cometer los mismos errores al probarlo.
- Debido a que es “SU” programa inconscientemente tiende a hacer casos de prueba que no hagan fallar al mismo.
- Puede llegar a comparar mal el resultado esperado con el resultado obtenido debido al deseo de que el programa pase las pruebas.

Niveles de Independencia

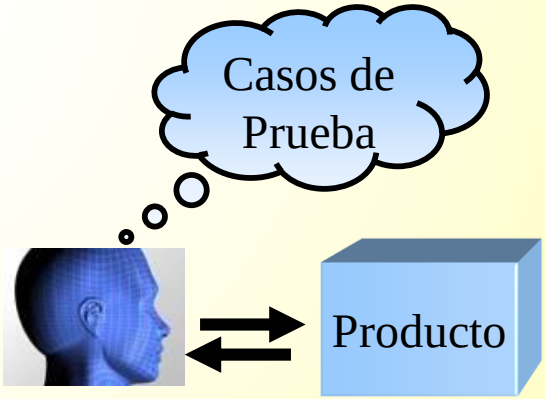
- Pruebas diseñadas por las personas que escribieron el software.
- Pruebas diseñadas por personas distintas pero del equipo de desarrollo.
- Pruebas diseñadas por personas de otro grupo de la organización (área de testing)
- Pruebas diseñadas por personas de otra organización (outsourcing)

Estrategias de prueba funcional

- **Testing planificado**
  - Diseño de casos de prueba
  - Ejecución de pruebas, incluso por otro tester

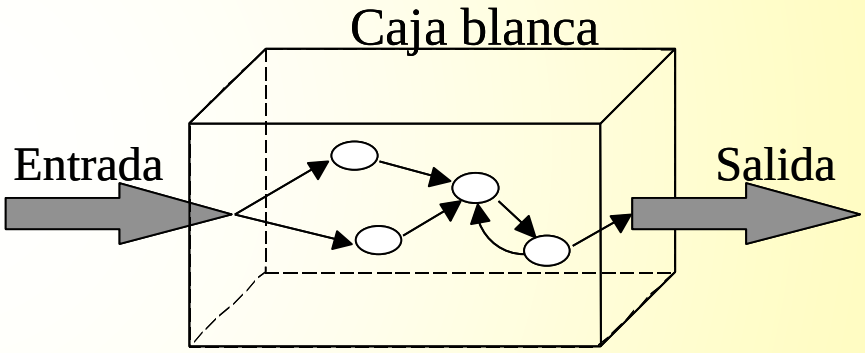


- **Testing exploratorio**
  - Diseño y ejecución de pruebas simultáneos



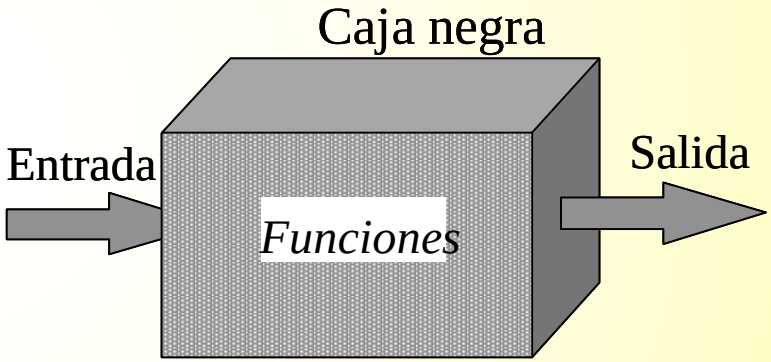
Técnicas de prueba

- Caja Blanca:
  - Cuando se diseña la prueba a partir del conocimiento de la estructura interna del código
  - Para ello se utiliza algún criterio de cobertura
  - Problemas: No se prueba la especificación y no se detectan ausencias



Técnicas de prueba

- Caja negra:
  - Cuando se diseña la prueba a partir del conocimiento de la especificación, ignorando la estructura interna.
  - Problemas: Infinitas posibilidades para las entradas



Técnica de caja blanca

- *¿Porqué usar pruebas de caja blanca?*
  - Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.
  - A veces creemos que un camino lógico tiene pocas posibilidades de ejecutarse cuando puede hacerlo de forma normal.

"Los errores se esconden en los rincones y se aglomeran en los límites" (Beizer 90)

Técnica de caja blanca

- Aseguran que la operación interna del programa se ajusta a las especificaciones y que todos los componentes internos se han probado adecuadamente.
- Usa la estructura de control para obtener los casos de prueba.
- Intentan garantizar que todos los caminos de ejecución del programa quedan probados.

Pruebas Caja blanca

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

Técnica de Caja negra

- Se centra en los requisitos funcionales del software.
- Permite obtener un conjunto de condiciones de entrada que ejerciten completamente los requisitos funcionales del programa.
- Complementan a las pruebas de caja blanca.

Técnica de Caja negra

- Con este tipo de pruebas se intenta encontrar:

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

Caso de prueba

- **Objetivo**: conseguir confianza aceptable en que se encontrarán todos los defectos existentes, sin consumir una cantidad excesiva de recursos.

- *"Diseñar las pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible."*

Caso de prueba

- Un buen caso de prueba es aquel que tiene una probabilidad muy alta de descubrir un nuevo error.
- Un caso de prueba no debe ser redundante.
- Debe ser el mejor de un conjunto de pruebas de propósito similar.
- No debe ser ni muy sencillo ni excesivamente complejo, es mejor realizar cada prueba de forma separada si se quiere probar diferentes casos.

Diseño de casos de prueba

- Definir los casos de prueba que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo.
- **Pruebas de caja blanca**
  - Encontrar casos de prueba "viendo" el código interno.
- **Pruebas de caja negra**
  - Encontrar casos de prueba "viendo" los requisitos funcionales.

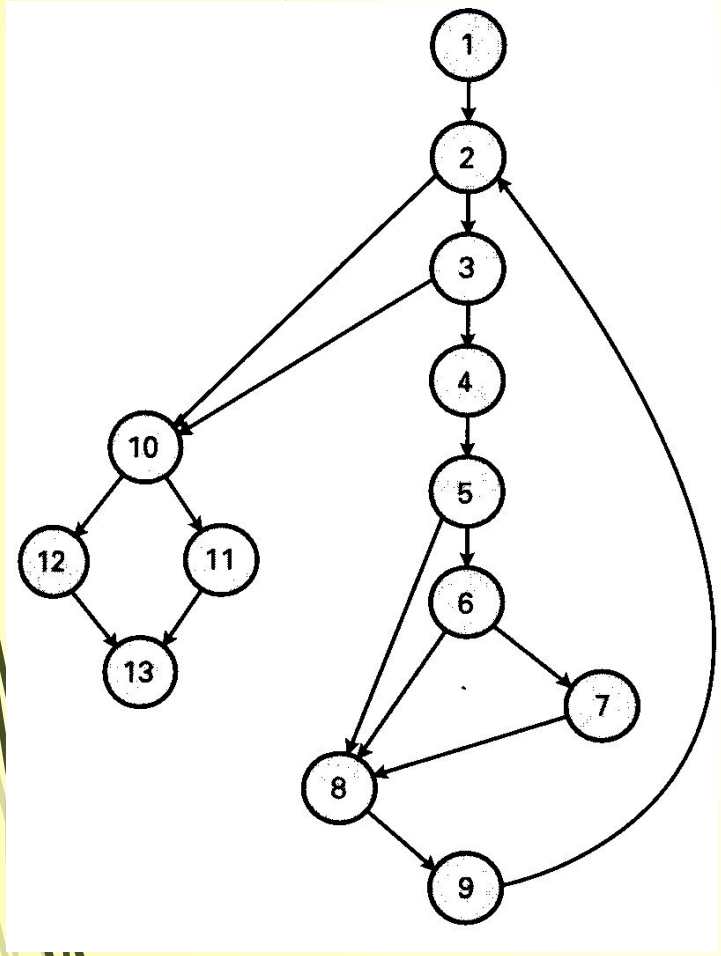
Complejidad Ciclomática

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

Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una nueva condición

Definición: El número de caminos independientes del conjunto básico de un programa y nos da un límite superior para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez

Complejidad Ciclomática

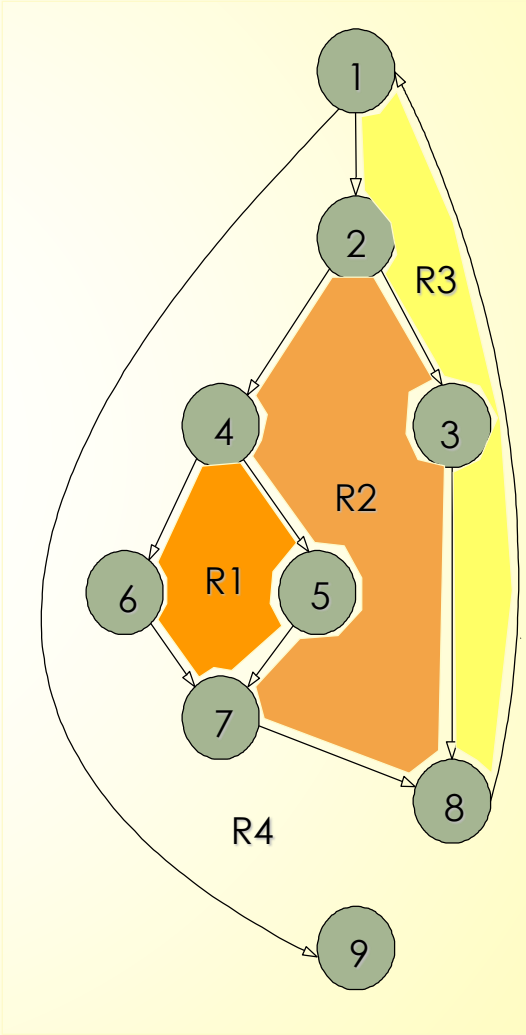


Se determina aplicando uno de los algoritmos descritos en la Sección
Se debe tener en cuenta que se puede determinar *V(G)* sin desarrollar el grafo de flujo, contando todas las sentencias condicionales del LDP

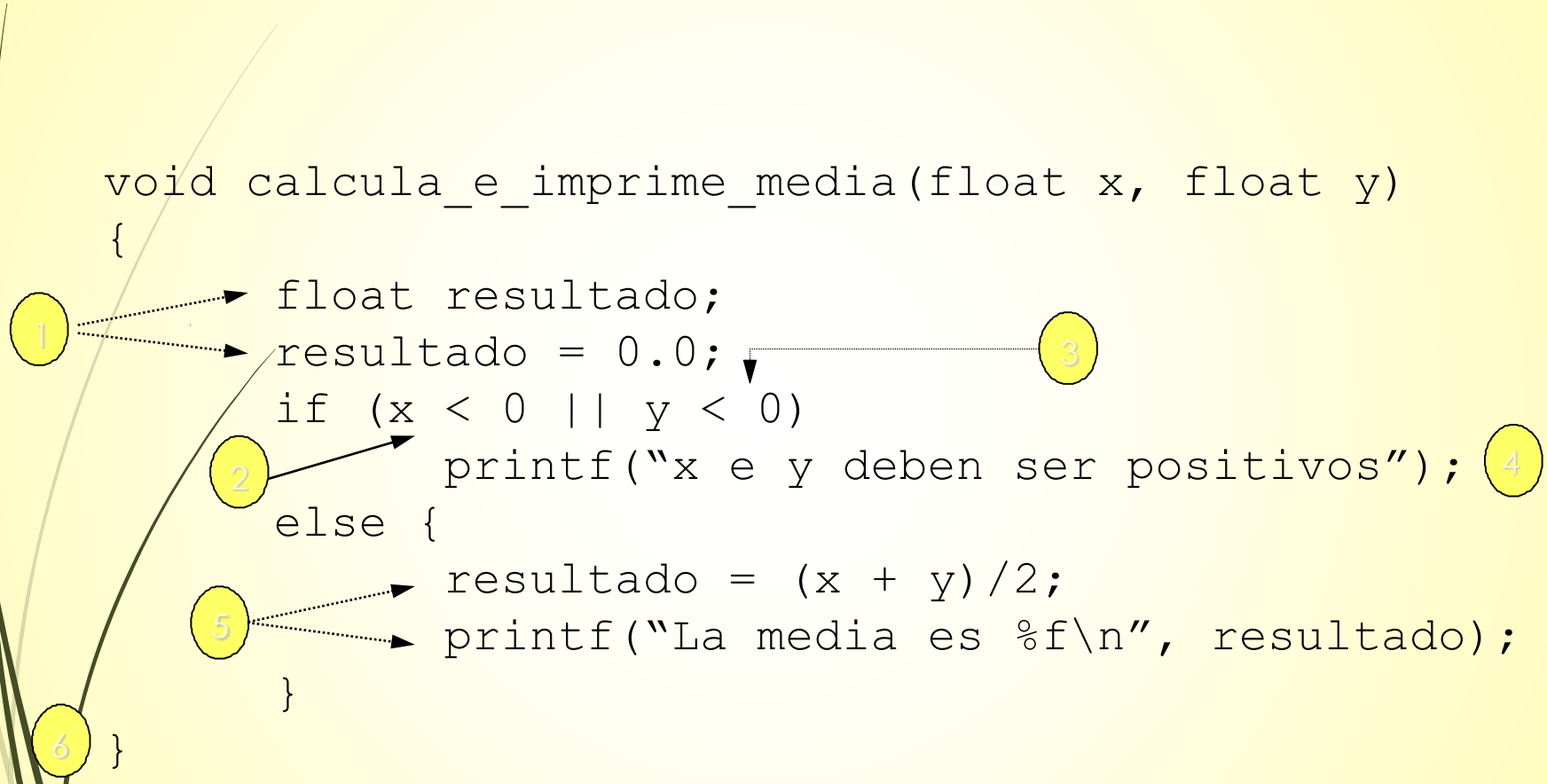
V(G) = 6 regiones
V(G) = 17 aristas - 13 nodos +2 = 6
V(G) = 5 nodos predicado + 1 = 6

Complejidad Ciclomática

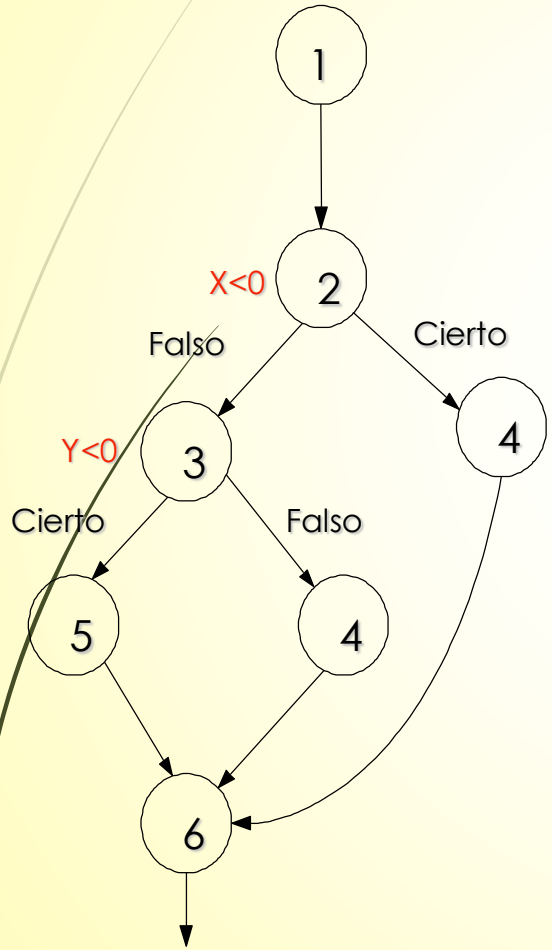
- V(G) = 4
  - El grafo de la figura delimita cuatro regiones.
  - 11 aristas - 9 nodos + 2 = 4
  - 3 nodos predicado + 1 = 4



Complejidad Ciclomática con código



Complejidad Ciclomática con código



V(G) = 3 regiones. Por lo tanto, hay que determinar tres caminos independientes.

- Camino 1: 1-2-3-5-6
- Camino 2: 1-2-4-6
- Camino 3: 1-2-3-4-6

Casos de prueba para cada camino:

Camino 1: x=3, y= 5, rdo=4

Camino 2: x=-1, y=3, rdo=0, error

Camino 3: x=4, y=-3, rdo=0, error

Esquema prueba de la partición equivalente

1

Identificar las condiciones de entrada y su tipo

2

Determinar las clases equivalencia

Rango: una válida y dos no válidas.
Valor específico: una válida y dos no válidas.
Miembro de un conjunto:
- Si cada elemento se trata igual, una válida y otra no válida.
- Si cada elemento se trata diferente, una válida por cada elemento y otra no válida

Lógica: una válida y otra no válida.

3

Condición Entrada	CLASES VÁLIDAS	CLASES NO VÁLIDAS
a	(1)	(2)
b	(3)	(4) (5)
. . .	. . .	. . .

TABLA de CLASES de EQUIVALENCIA ENUMERADAS

4

Casos de Prueba Válidos:
que cubran clases válidas

Casos de Prueba No Válidos:
que cubran una clase no válida y el resto válidas

Clases de equivalencia

- **Procedimiento** :
  - Identificar condiciones de entrada/salida que influyen en diferentes comportamientos del programa
  - Identificar en cada condición de entrada/salida las categorías que causan comportamientos homogéneos (clases de equivalencia)
  - Crear casos de prueba para cada una de las clases

Clases de equivalencia

- Cada caso de prueba debe cubrir el máximo nº de entradas.
- Debe tratarse el dominio de valores de entrada dividido en un nº finito de clases de equivalencia

  ⇒ la prueba de un valor representativo de la clase permite suponer "razonablemente" que el resultado obtenido será el mismo que probando cualquier otro valor de la clase
- Método de diseño:

  1. Identificación de clases de equivalencia.
  2. Creación de los casos de prueba correspondientes.

Condiciones de entrada

- Pueden ser del tipo:
  1. Rango
  2. valor
  3. Miembro de un conjunto,
  4. lógica

Ejemplo Clase de equivalencia

- Aplicación bancaria. Datos de entrada:
  - **Código de área**: número de 3 dígitos que no empieza por 0 ni por 1
  - **Nombre de identificación de operación**: 6 caracteres
  - **Órdenes posibles**: "Cheque", "Depósito", "Pago factura", "Retiro de fondos"

Ejemplo : Determinar las clases de equivalencia

Código de área:

Rango:

- 1 clase válida: 200 < código < 999
- 2 clases no válidas: código < 200; código > 999

Nombre de identificación:

Valor:

- 1 clase válida: 6 caracteres
- 2 clases no válidas: más de 6 caracteres; menos de 6 caracteres

Órdenes posibles:

Miembro de un conjunto:

- 1 clase válida: 4 órdenes válidas
- 1 clase no válida: orden no válida

Ejemplo Clase de equivalencia

		CLASES VÁLIDAS		CLASES NO VÁLIDAS	
Condición de Entrada	**Tipo**	**Entrada**	**Código**	**Entrada**	**Código**
Código de área	Rango	200<= código<=999	CEV<01>	código<200	CENV<01>
				código>999	CENV<02>
Identificación	Calor	Identificación= 6 caracteres	CEV<02>	Identificación< 6 caracteres	CENV<03>
				Identificación> 6 caracteres	CENV<04>
Orden	Miembro de un Conjunto	Cheque	CEV<03>		
		Deposito	CEV<04>		
		Pago Factura	CEV<05>		
		Retiro de fondos	CEV<06>		

Ejemplo caso de prueba (entradas válidas)

	Condiciones de entrada		
Clases de equivalencia	**Código de área**	**Identificación**	**Orden**
CEV<01>, CEV<02>, CEV<03>	350	ch1234	Cheque
CEV<01>, CEV<02>, CEV<04>	230	de1234	Deposito
CEV<01>, CEV<02>, CEV<05>	200	fa1254	Pago Factura
CEV<01>, CEV<02>, CEV<06>	999	ret5624	Retiro de fondos

Ejemplo caso de prueba (entradas no válidas)

	Condiciones de entrada		
Clases de equivalencia	**Código de área**	**Identificación**	**Orden**
CEV<01>, CEV<03>, CEV<05>	130	21222222222	
CEV<02>, CEV<04>, CEV<05>	1500	1	

Ejercicio


Simulador de
Depósitos de Ahorro
Monto:
0.00
Periodo:
días
Fec. Apertura:
15/07/2011
Tasa (T.E.A.):
Moneda:
Nacional
Aplicar
Cerrar