Introducción a la Ingeniería del Software

TEMA 7: VERIFICACIÓN Y PRUEBAS

Grado en Ingeniería Informática
Grado en Ingeniería del Software
Grado en Ingeniería de Computadores

UNIVERSIDAD DE MÁLAGA
E.T.S. INGENIERÍA INFORMÁTICA

Índice de contenidos

Reflexiones

- *"If you look at how most programmers spend their time, you'll find that writing code is actually a small fraction. Some time is spent in figuring out what ought to be going on, some time is spent designing, but most time is spend debugging … Fixing a bug is usually pretty quick, but finding it is a nightmare. And then when you do fix a bug, there's always a chance that another one will appear and that you might not even notice it until much latter. Then you spend ages finding that bug".*
- Martin Fowler. Refactoring (Chapter 4)

Depuración de errores

- Tradicionalmente
  - La fase de pruebas de un proyecto tenía lugar después de la codificación y antes de la entrega del producto al cliente.
  - Inconvenientes: errores en etapas tempranas (requisitos, modelado) son muy costosos de reparar.
- Alternativa
  - Realizar pruebas en paralelo a la fase de desarrollo.
- Alternativa extrema
  - *Test Driven Development*
  - Las pruebas pasan a ser el centro del proceso de desarrollo.

Verificación

- Un producto software es un producto de calidad si
  - Cumple con los requisitos de usuario y de sistema.
    - Requisitos funcionales y no funcionales
- La finalidad de las pruebas es:
  - **Verificar** que el producto desarrollado cumple con todos los requisitos
    - ¿se ha construido el sistema correctamente?
  - **Validar** el producto
    - ¿se ha construido el sistema correcto?

Conceptos básicos

- Las pruebas de software (*software testing*) se definen como:
  - El proceso que permite identificar la corrección completitud, seguridad y calidad del producto software desarrollado.
  - NO SON
    - Un método para demostrar que NO hay errores,
    - Un método para demostrar que el software funciona correctamente.
- Una prueba en sí es:
  - El proceso de ejecutar ciertos componentes software con el fin de encontrar errores.

Conceptos básicos

- Un caso de prueba es:
  - Conjunto de condiciones bajo las cuales se puede determinar si los requisitos del software se cumplen parcial o totalmente, o bien no se cumplen.
  - Un buen caso de prueba es aquel que tiene una alta probabilidad de encontrar un error no descubierto hasta entonces.
- Definición de error:
  - Discrepancia entre un valor o condición calculado, observado o medido y el valor o condición específica teóricamente correctos.

Principios a seguir

- Definir los casos de prueba en la fase de diseño
- Un buen caso de prueba tiene alta probabilidad de mostrar un error.
- Saber el resultado esperado del programa es fundamental en un caso de prueba.
- Un programador debe evitar probar su programa.
- Cuanto más se modifique un programa, más hay que probarlo.
- Es fundamental documentar bien los casos de prueba.

¿Quién prueba el software?

Desarrollador

Comprende el sistema pero probará “amablemente”. Está condicionado por la entrega

Probadores independientes

Debe comprender el sistema, pero intentará “romperlo”. Está dirigido por la calidad

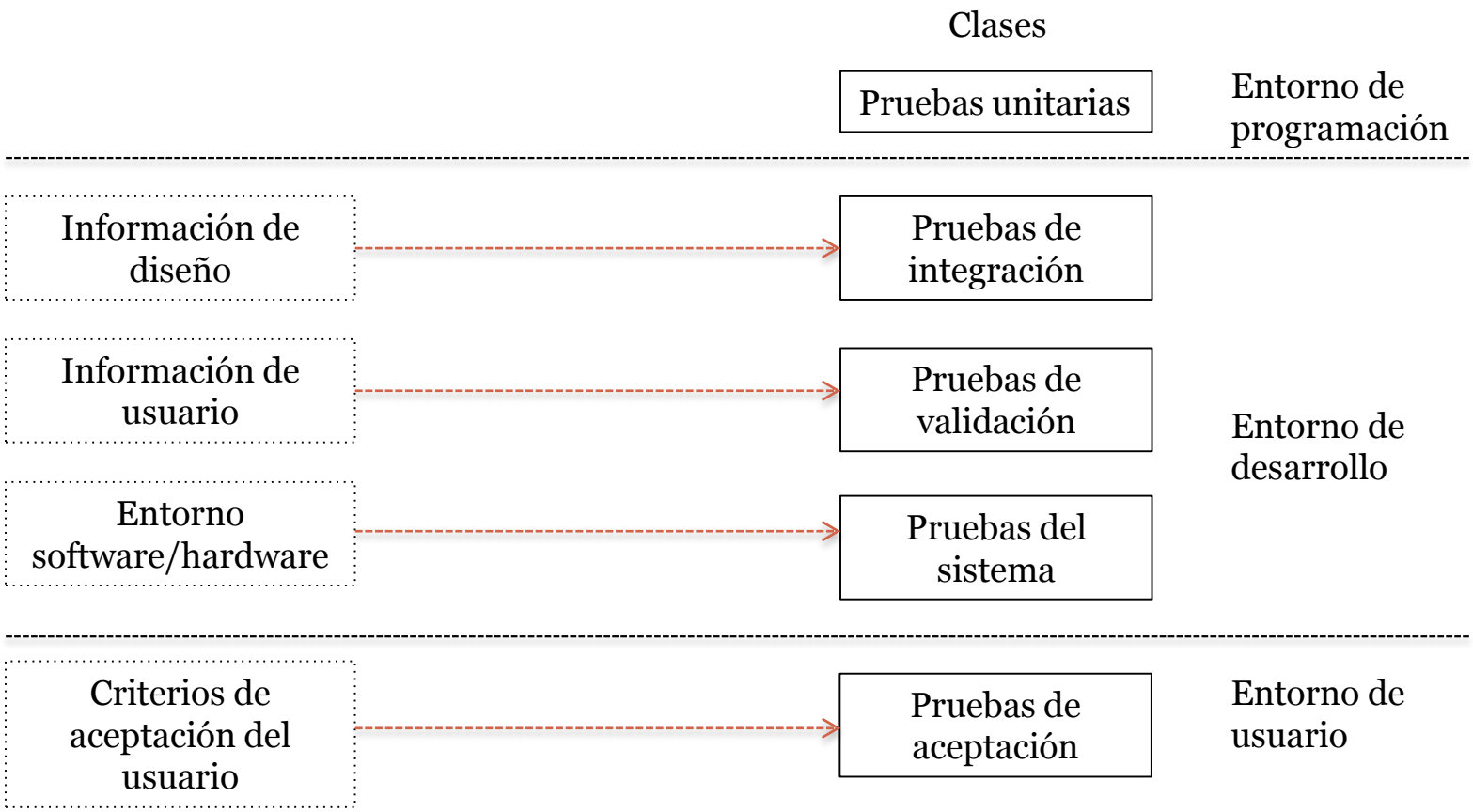
Pruebas de caja blanca y de caja negra

- Son dos tipos de pruebas relacionadas con la ejecución de código
- Pruebas de caja blanca:
  - Comprueban el funcionamiento interno y la lógica del código
- Pruebas de caja negra:
  - Son pruebas guiadas por los datos de entrada y de salida, sin tener en cuenta los detalles de implementación.
  - No se dispone de acceso al código fuente.

Clasificación de tipos de pruebas

- Pruebas unitarias (de componentes aislados).
- Pruebas de integración (varios componentes a la vez).
- Pruebas de sistema (sistema completo).
- Pruebas de validación (se cumplen los requisitos?).
- Pruebas de aceptación (validación por el usuario).
- Pruebas de regresión (después de un cambio, verificar que no hay errores).
- Pruebas de estrés (prueba bajo alta carga).
- Pruebas de carga (prueba bajo carga normal).

Relación entre los tipos de pruebas



Herramientas

- Multitud de herramientas existentes:
  - jUnit: http://www.junit.org/
  - nUnit: http://www.nunit.org/
  - jMock: http://www.jmock.org/
  - PEX: http://research.microsoft.com/en-us/projects/pex/
  - Mockito: http://code.google.com/p/mockito/
  - ...

Pruebas de caja blanca

- Idea básica:
  - Asegurar que todas las sentencias y condiciones han sido ejecutadas al menos una vez.
- Es decir, hay que diseñar casos de prueba que garanticen que:
  - se ejecuten por lo menos una vez todos los caminos independientes de cada módulo.
  - Se comprueben todas las decisiones lógicas (V y F).
  - Se comprueben los bucles en sus límites.
  - Se analicen estructuras internas de datos.
- Cobertura de código:
  - Porcentaje del código sometido a pruebas.

Criterios de Cobertura

- Criterio de Cobertura de Sentencias.
  - Es el más básico de todos y su función es garantizar que cada sentencia sea ejecutada al menos una vez.
- Criterio Cobertura de Condición:
  - Este criterio requiere que cada condición de cada decisión sean evaluados cuando es verdadera o cuando es falsa.
- Cobertura de Múltiples Condiciones:
  - Este criterio requiere que todas las condiciones tomen valor V y F, de manera que se recorra toda la tabla de verdad.
- Criterio de Cobertura de Decisión:
  - Este criterio requiere que todas las decisiones sean evaluadas cuando es verdadera y cuando es falsa.
- Criterio de Condición/Decisión:
  - Este criterio requiere que cada condición de cada decisión se evalúe cuando es V y F al menos una vez, y que cada decisión se evalúe cuando es F y V al menos una vez.
- Criterio de Cobertura de Camino.

Cobertura de Sentencia

- Es el nivel más bajo de testeo.
- Cada sentencia debe ser ejecutada al menos una vez.
- Pero muchos caminos pueden no ser testeados.
- Ejemplo: If (a>0) {x++;} if (b==3) {y=0;}
  - La cobertura de este código podría a través de un caso de prueba cuando a>0 es true y b==3 es true.

Condición/Decisión

- Una condición es una variable booleana o un par de expresiones relacionadas por un operador relacional (<,>,=,≥,≤,≠):

Expresión 1 <operador relacional> Expresión 2

- Una decisión es una lista de condiciones conectadas por operadores lógicos (and, or):

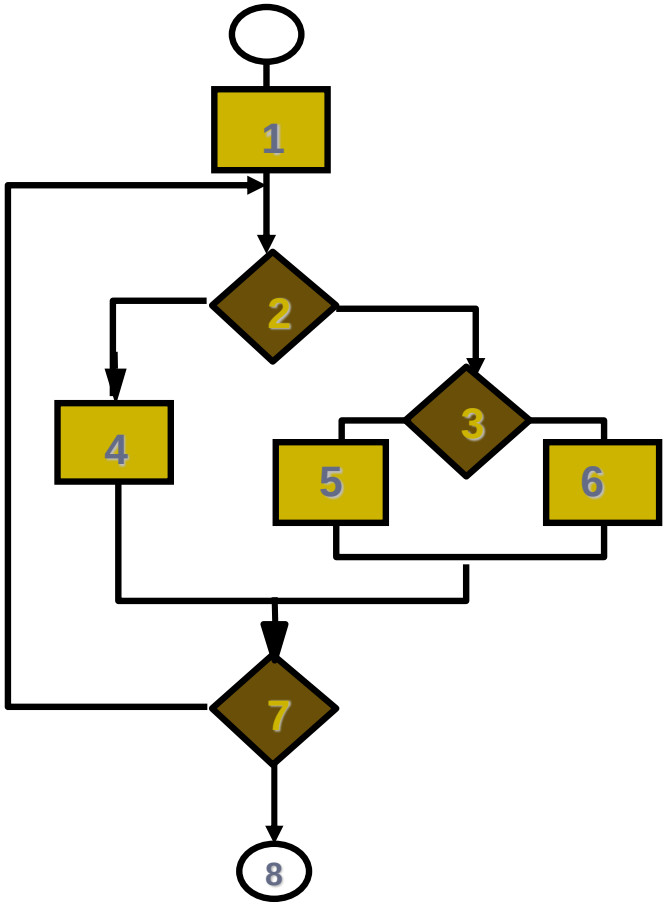
Condición 1 <operador lógico> Condición 2 …<operador lógico> Condición n

Cobertura de Condición de Decisión

- En Inglés: Decision Condition Coverage.
- La cobertura de condición de decisión es el porcentaje de todos los resultados de condición y resultados de decisión que han sido practicados por un juego de pruebas.
- Un 100% de cobertura de condición-decisión implica tanto un 100% de cobertura de condición como un 100% de cobertura de decisión.

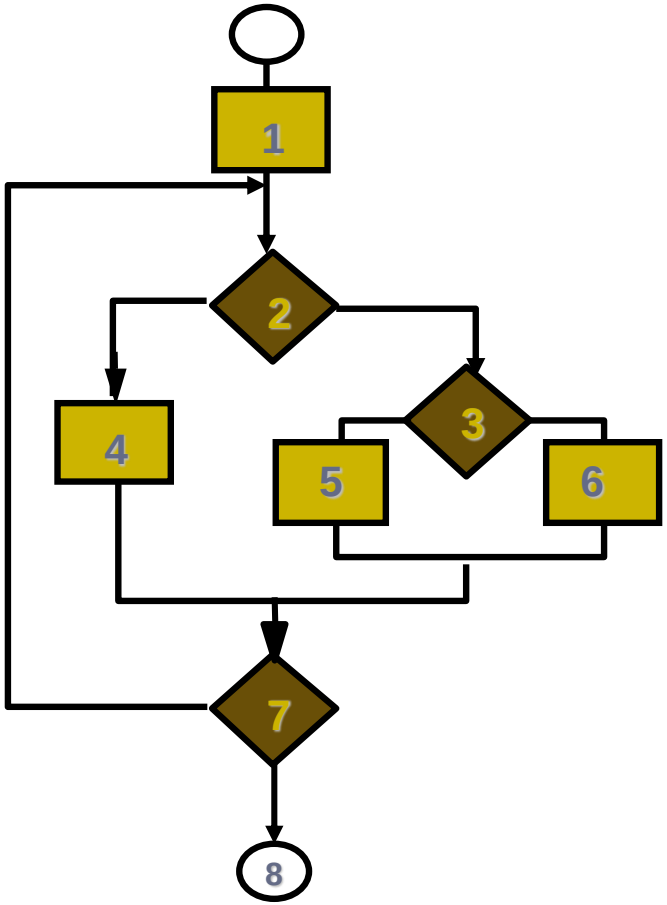
Cobertura de Camino (Path Coverage)

- Utilizamos las pruebas de camino base. Este es el nivel más alto de cobertura 100%.



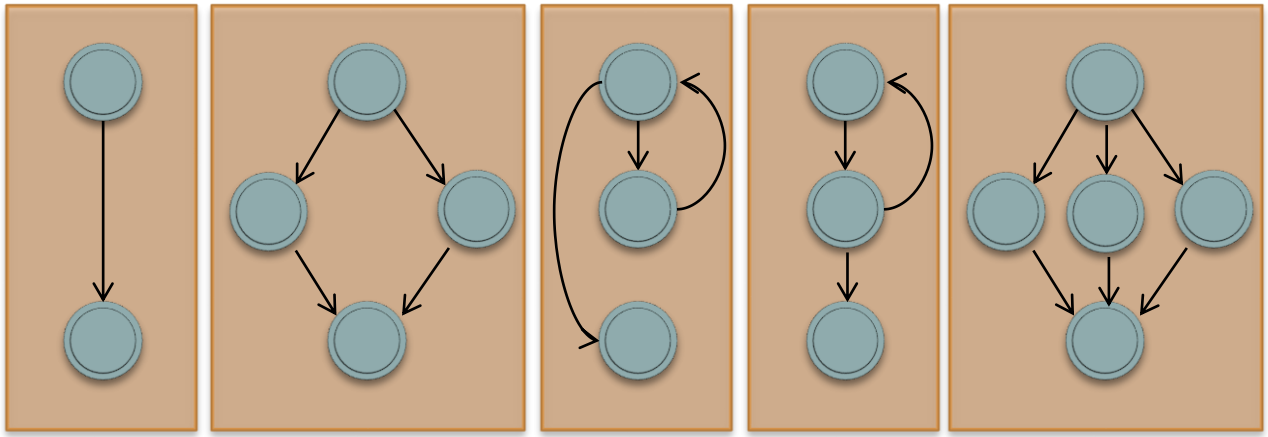
Prueba de los caminos base

- Permite obtener una medida de la **complejidad lógica** de un diseño procedimental
- Y usar esa medida como guía para la definición de un conjunto de caminos base de ejecución
- *Notación de grafo de flujo*



Prueba de los caminos base

- Complejidad ciclomática
  - Es una métrica que proporciona una medición cuantitativa de la complejidad lógica de un programa
  - Indica el número de caminos base de un grafo de ejecución de un programa



Caminos Base

Prueba de los caminos base

- Complejidad Ciclomática
  - Proporciona un límite superior para el número de pruebas que se deben realizar para asegurar que se ejecutan todas las sentencias al menos una vez.
- A mayor valor de complejidad ciclomática , mayor probabilidad de errores.

In graph theory, there exists a proof that states that the cyclomatic complexity of a strongly connected graph is the number of linearly independent circuits in a graph

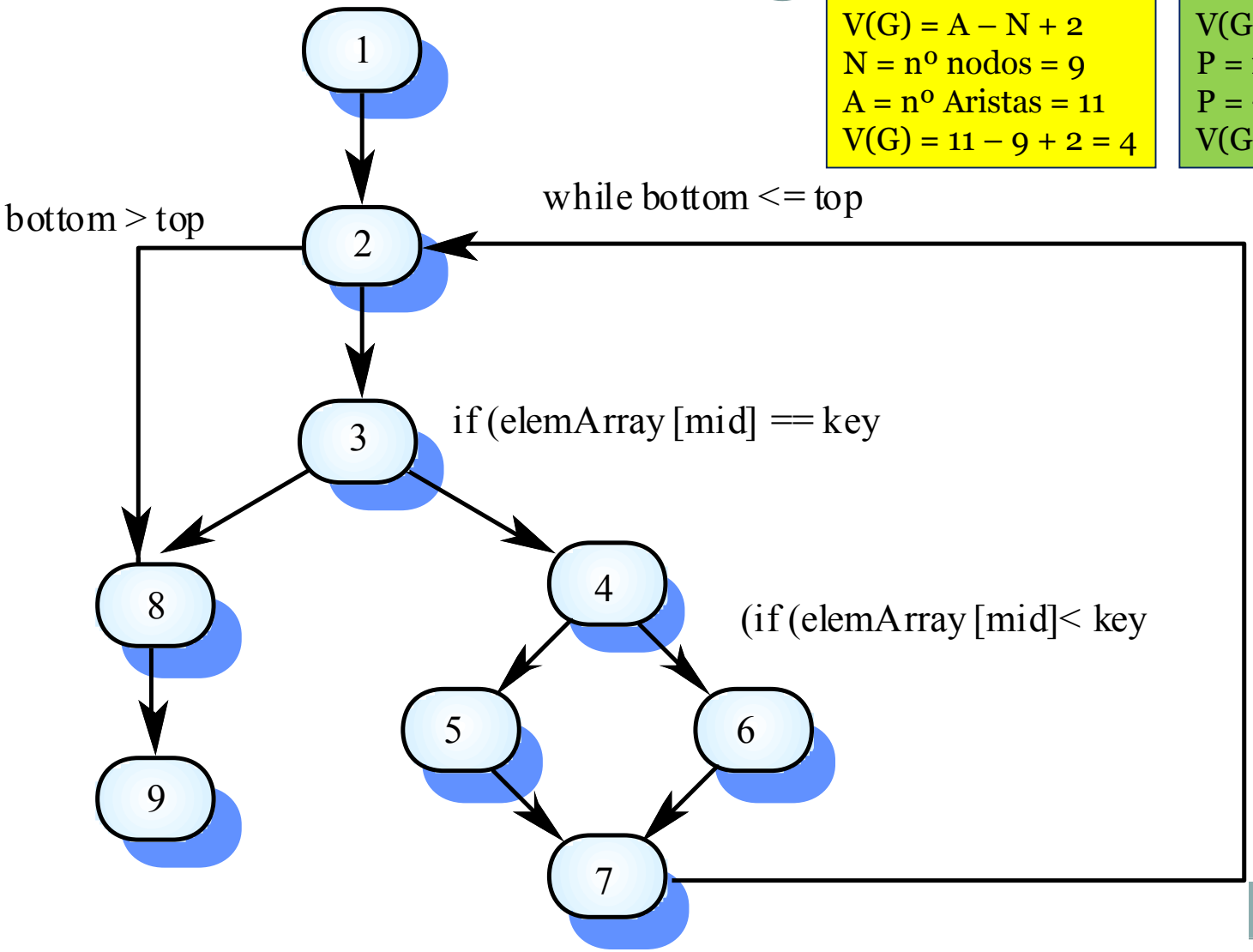
Prueba de los caminos base

- Complejidad Ciclomática
  - Se basa en la teoría de grafos
  - Se puede calcular de varias formas:
    - V(G) = Aristas – Nodos + 2
    - V(G) = P +1, donde P es el número de nodos predicados (nodos que contienen una condición, por lo que dos o más aristas parten de ellos)
  - El número de caminos base coincide con la complejidad ciclomática

Prueba de los caminos base

V(G) = A – N + 2
N = nº nodos = 9
A = nº Aristas = 11
V(G) = 11 – 9 + 2 = 4

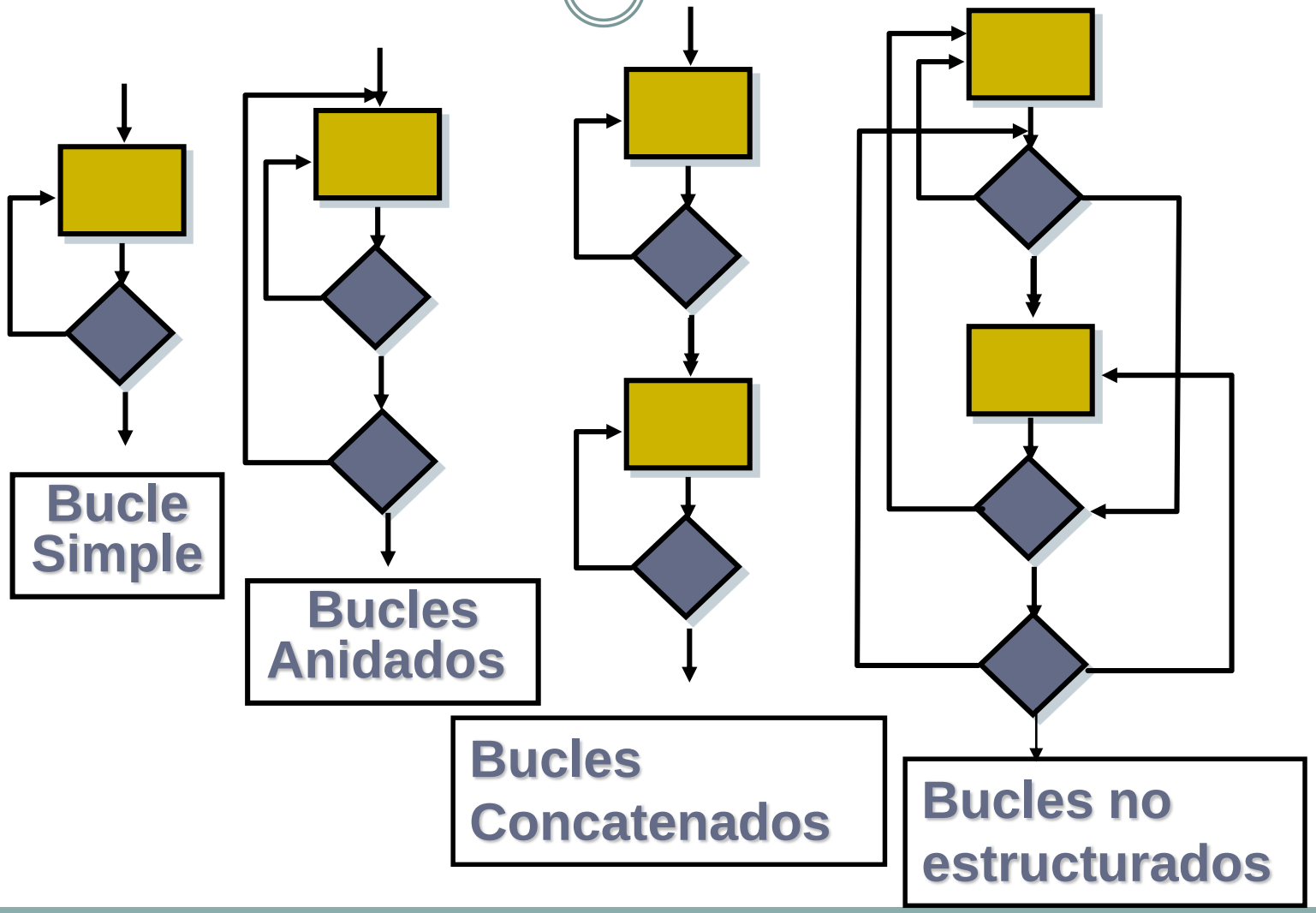
V(G) =|P|+1
P = nodos predicados
P = {2, 3, 4}
V(G) = 3 + 1 = 4



Prueba de los caminos base

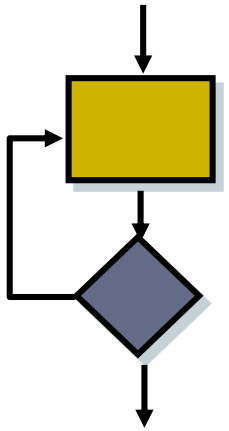
- Caminos independientes:
  - 1, 2, 8, 9
  - 1, 2, 3, 8, 9
  - 1, 2, 3, 4, 5, 7, 2, 8, 9
  - 1, 2, 3, 4, 6, 7, 2, 8, 9
- Los casos de prueba han de cubrir todos estos caminos.
- Un analizador dinámico podría comprobar que estos caminos han sido ejecutados.

Pruebas de bucles



Pruebas de bucles

- Bucles simples
  - Siendo n el número máximo de pasos
    - Saltar el bucle
    - Pasar una sola vez
    - Pasar dos veces
    - Hacer m pasos, siendo m < n
    - Hacer n -1, n y n+1 pasos



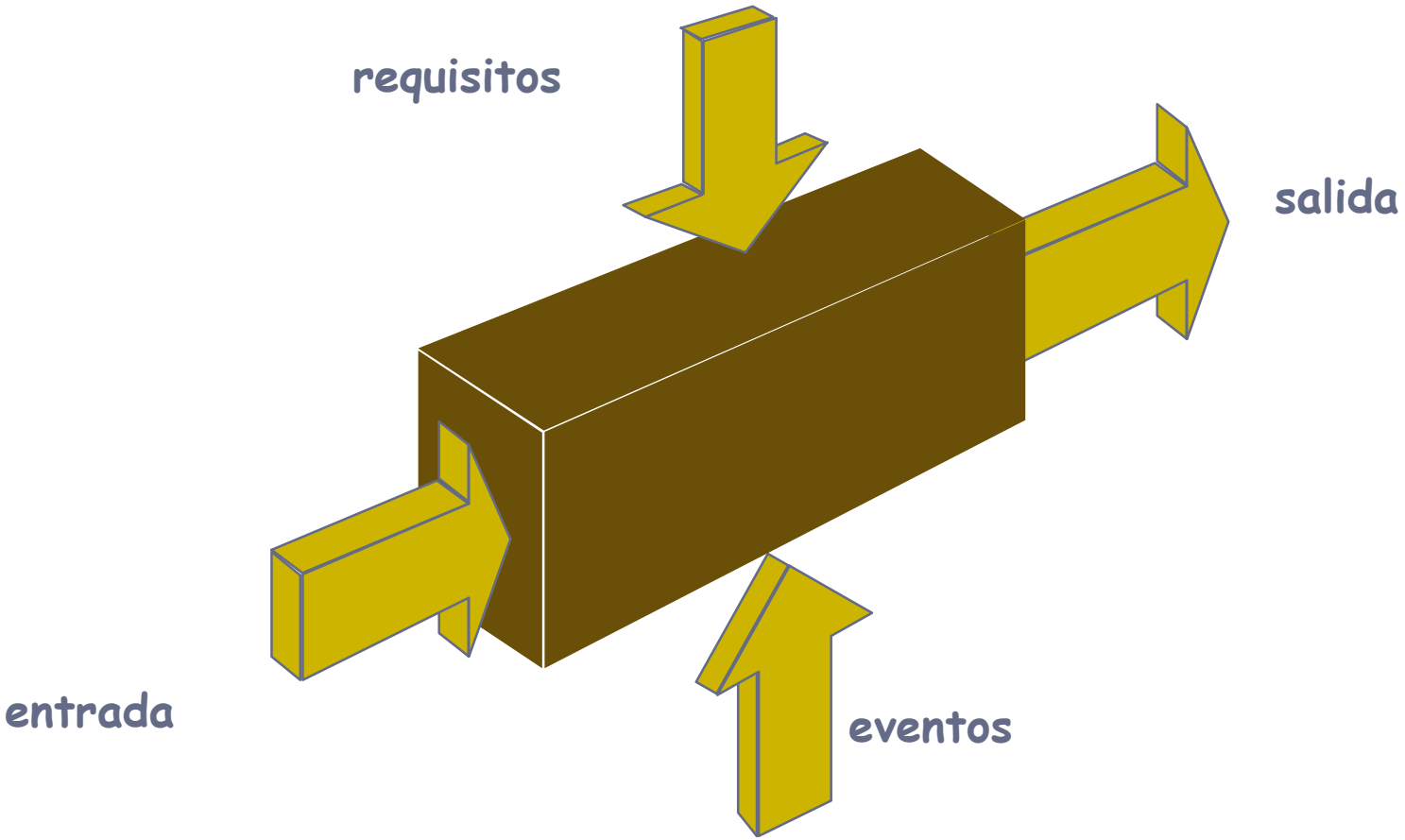
Pruebas de bucles

- Bucles anidados
  - Aumentaríamos geométricamente el número de pruebas
    - Comenzar por el bucle más interior y los demás al mínimo
    - Pruebas de bucles simples para el bucle más interior, resto a mínimos.
    - Progresar hacia fuera y para cada bucle realizar pruebas de bucles simples manteniendo los bucles externos en mínimos y los bucles internos en sus valores medios o típicos.

Pruebas de bucles

- Bucles concatenados
  - Si son independientes, se prueba como bucles simples.
  - Si no, se prueban como bucles anidados.
- Bucles no estructurados
  - Son bucles mal diseñados: rediseñar de forma correcta.

Pruebas de caja negra



Pruebas de caja negra

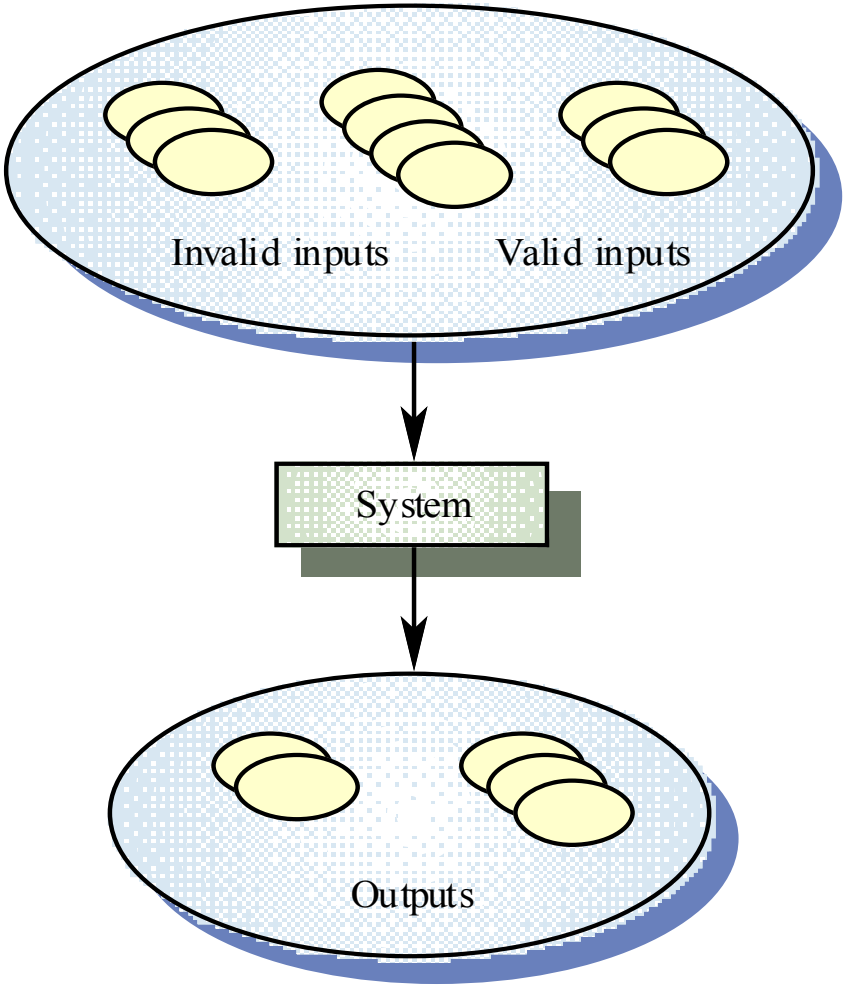
- Se centran en los requisitos funcionales del software
  - Obtener conjuntos de condiciones de entrada que prueben todos los requisitos funcionales del programa
  - No es una alternativa a las pruebas de caja blanca
    - Es un enfoque complementario

Pruebas de caja negra

- Trata de encontrar errores en:
  - 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 de terminación.
- Ignora intencionadamente la estructura de control.
- Se suele aplicar durante las fases finales de las pruebas.

Pruebas de caja negra

- Están conducidas por los datos de entrada y de salida.
- Hay que conocer de antemano las salidas correctas.
- Dos técnicas:
  - Partición de equivalencia
  - Análisis de valores límites



Partición de equivalencia

- Idea:
  - Hay que definir casos que descubran clases de errores.
  - Dividir el dominio de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba.
- Evaluar clases de equivalencia para una condición de entrada.
  - Una clase de equivalencia representa un conjunto de estados válidos o inválidos para condiciones de entrada.

Partición de equivalencia

- Se pueden definir de acuerdo con las siguientes directrices:
  - Un rango: define una clase de equivalencia válida y dos inválidas.
  - Un valor específico: una válida y dos no válidas.
  - Un miembro de un conjunto: una válida y una no válida.
  - Lógica: una válida y una no válida .

Partición de equivalencia

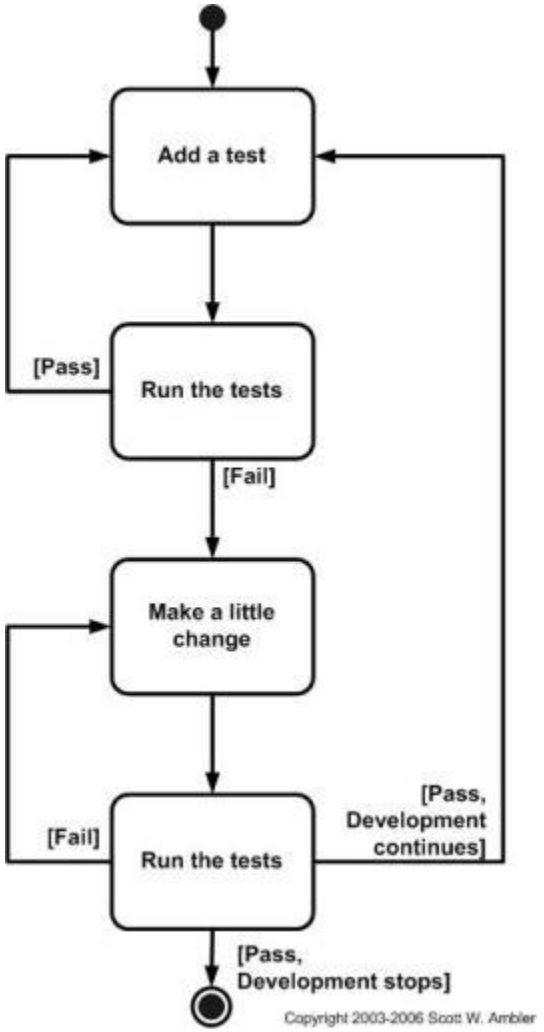
- Ejemplo:
  - Una entrada es un entero de 5 dígitos entre 10.000 y 99.999.
  - La particiones equivalentes son
    - <10.000
    - 10.000 – 99.999
    - > 100.000
  - Luego hay que elegir casos de prueba en los límites de estos conjuntos:
    - 00000, 09999, 10000, 99999, 100000, 100001

Análisis de valores límite

- Los errores tienden a darse más en los límites del campo de entrada que en el centro.
  - Consecuentemente, hay que plantear casos de pruebas que ejerciten los valores límite.
- Esta estrategia es complementaria a la partición equivalente.

Test Driven Development (TDD)

- Desarrollo dirigido por pruebas
  - Una de las características claves de Extreme Programming.
  - Proceso de desarrollo software basada en el siguiente ciclo:
    - Escritura de un caso de prueba para una funcionalidad a incluir en el software.
    - Implementar el código que pase el test.
    - Refactorizar el código.



Test Driven Development (TDD)

- Ventajas
  - Al centrarse en las pruebas, el código tiene menos errores y se aumenta la productividad,
  - Separación entre el comportamiento esperado y la implementación para conseguirlo,
  - Tests de regresión: al desarrollar de forma incremental los tests de regresión permiten encontrar errores,
  - Se reduce la necesidad de depurar código.
- Inconvenientes
  - Frecuentemente, el diseñador de los tests implementa el código,
  - Sensación de seguridad que puede ser irreal.

EJERCICIOS

Ejercicios

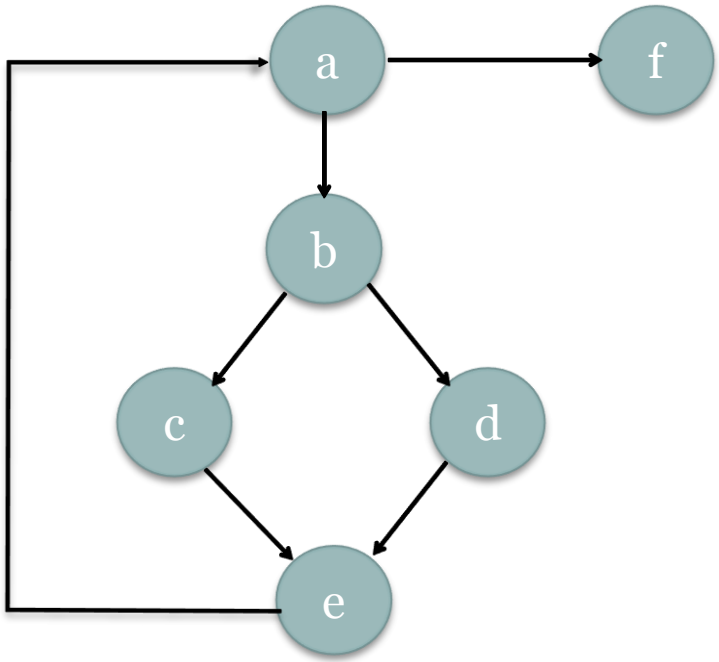
- Dada la siguiente función, que calcula el MCD de dos números, obtener el grafo correspondiente, la complejidad ciclomática y los caminos base:

```
public int MCD (int x, int y) {
  while (x != y) // a
  {
    if (x > y)    // b
      x = x - y   // c
    else
      y = y - x   // d
  }               // e
  return x ;      // f
```

Ejercicios

- Solución:

```
public int MCD (int x, int y) {
  while (x != y) // a
  {
    if (x > y)    // b
      x = x - y   // c
    else
      y = y - x   // d
  }               // e
  return x ;      // f
```



Ejercicios


```
/* Determina si un número es menor, mayor o igual a cero */
#include <iostream>
using namespace std;
int main()
{
   int num;
   cout << "Valor del número: ";
   cin >> num;
   if (num >= 0)
      if(num == 0)
        cout << "Número igual a cero \n";
      else
        cout << "Número mayor que cero \n";
   else
      cout << "Número menor que cero \n";
   return 0;
}
```

Pruebas de condición. Ejemplo


```
public void comprobarHora(int h, int m, int s) {
  if ((h >= 0) && (h <= 23)) {
    if ((m >= 0) && (m <= 59)) {
      if ((s >= 0) && (s <= 59)) {
        System.out.println ("Hora correcta") ;
        return ;
      }
    }
  }
  System.out.println ("Hora incorrecta") ;
}
```

Pruebas de condición. Ejemplo

- Tres decisiones:
  - D1: `((h >= 0) && (h <= 23))`
  - D2: `((m >= 0) && (m <= 59))`
  - D3: `((s >= 0) && (s <= 59))`
- Casos de prueba: ejemplos de valores

	Verdadero	Falso
D1	h=10	h=24
D2	m=25	m=65
D3	s=12	s=70

Pruebas de condición. Ejemplo

- Tres decisiones:
  - D1: `((h >= 0) && (h <= 23))`
  - D2: `((m >= 0) && (m <= 59))`
  - D3: `((s >= 0) && (s <= 59))`
- Casos de prueba para cubrir las decisiones:
  - Caso 1: D1 = verdadero, D2 = verdadero, D3 = verdadero
  - Caso 2: D1 = verdadero, D2 = verdadero, D3 = falso
  - Caso 3: D1 = verdadero, D2 = falso
  - Caso 4: D1 = falso

Pruebas de condición. Ejemplo

- Tres decisiones, dos condiciones por decisión:
  - D1: `((h >= 0) && (h <= 23))`
    - C1.1: `(h >= 0)`
    - C1.2: `(h <= 23)`
  - D2: `((m >= 0) && (m <= 59)`
    - C2.1: `(m >= 0)`
    - C2.2: `(m <= 59)`
  - D3: `((s >= 0) && (s <= 59))`
    - C3.1: `(s >= 0)`
    - C3.2: `(s <= 59)`

Pruebas de condición. Ejemplo

- Tres decisiones, dos condiciones por decisión
  - Hay que garantizar que cada condición tome al menos una vez el valor verdadero y otra el falso, garantizando la cobertura de la decisión

	Verdadero	Falso
C1.1	h=10	h = -1
C1.2	h=10	h=24
C2.1	m=30	m=-1
C2.2	m=30	m=60
C3.1	s=50	s=-1
C3.2	s=50	s=70

Pruebas de condición. Ejemplo

- Casos de prueba:
  - Caso 1: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = V, C3.1 = V, C3.2 = V
  - Caso 2: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = V, C3.1 = V, C3.2 = F
  - Caso 3 C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = V, C3.1 = F, C3.2 = V
  - Caso 4: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = F
  - Caso 5: C1.1 = V, C1.2 = V, C2.1 = F, C2.2 = V
  - Caso 6: C1.1 = V, C1.2 = F
  - Caso 7: C1.1 = F, C1.2 = V

Ejercicios


```
void  escribir_cuadrado( unsigned lado )
{ int i,j;
  for(i=1; i<=lado; i++)
    {
    for(j=1; j<=lado; j++)
        cout << "*";
    cout << endl;
    }
}
```