



Sesión 2: Diseño de pruebas de caja blanca

Diseño de casos de pruebas estructurales

Método del camino básico

Diseño de casos de prueba

■ Ya conocemos lo que es una Tabla de casos de prueba:

* En el laboratorio hemos confeccionado una tabla de este tipo, con casos de prueba que os hemos proporcionado, e incluso habéis añadido alguna fila ...

Tabla de casos de prueba

ID	d1	d2	...	Expected Output	Real Output	
c1	?	?	?	?	R1	<input checked="" type="checkbox"/> ok
c2	?	?	?	?	R2	<input checked="" type="checkbox"/> ok
c3					R3	<input type="checkbox"/>
...				
<u>CN?</u>						

* ¿COMO RELLENAMOS LAS FILAS DE LA TABLA?

* ¿CUÁNTAS FILAS ES NECESARIO AÑADIR?

Utilizando un MÉTODO de
DISEÑO de casos de prueba!!!!



Formas de identificar los casos de prueba

DISEÑAR

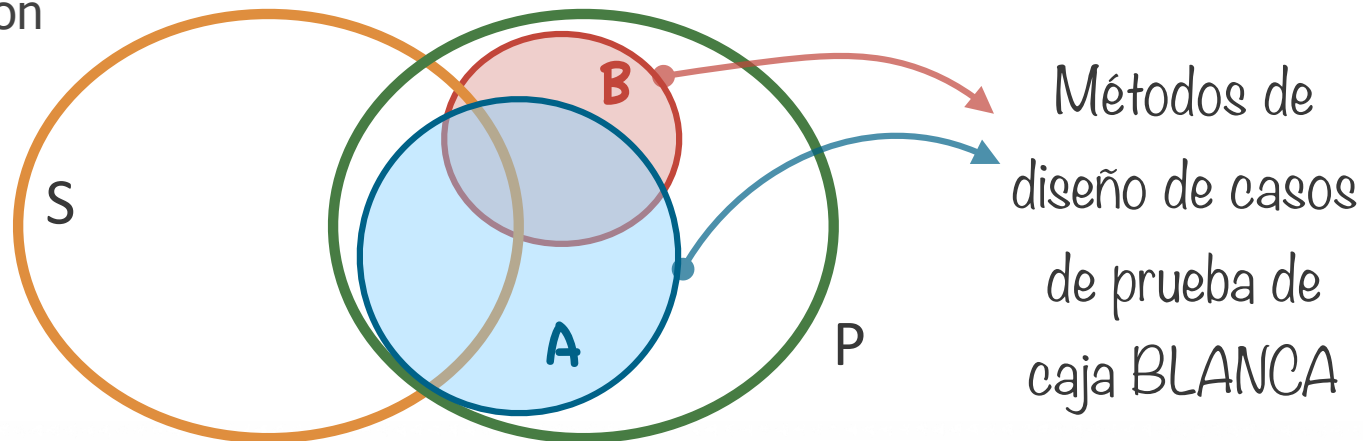
■ Fundamentalmente hay dos formas de proceder:

* **Functional testing:** aproximación basada en la ESPECIFICACIÓN

- Cualquier programa puede considerarse como una función que “mapea” valores desde un dominio de entrada a valores en un dominio de salida. El elemento a probar se considera como una “caja negra”
- Los casos de prueba obtenidos son independientes de la implementación
- El diseño de los casos de prueba puede realizarse en paralelo o antes de la implementación

* **Structural testing:** aproximación basada en la IMPLEMENTACIÓN

- Utilizamos **EL CÓDIGO** para determinar el conjunto de casos de prueba. Esta aproximación también se conoce con el nombre de “**caja blanca**”
- Es esencial conocer conceptos de teoría de grafos para entender bien esta aproximación





Métodos de diseño de caja blanca

■ Existen múltiples métodos de diseño de pruebas de caja blanca:

* **Control-flow testing** (flujo de control entre instrucciones)

- La idea principal es seleccionar convenientemente un conjunto de caminos en un programa y observar si la ejecución de dichos caminos seleccionados producen el resultado esperado
- Para seleccionar los caminos necesitamos:
 - generar un CFG (Control Flow Graph)
 - establecer un criterio para seleccionar dichos caminos (p.ej. seleccionar los caminos de forma que cada sentencia se ejecute al menos una vez)
- Ejemplo de método de diseño: McCabe's basis path method

* **Data flow testing** (propagación de valores entre variables)

- A partir de una representación del flujo de valores de datos de las variables, podemos:
 - detectar de forma ESTÁTICA potenciales defectos del programa (anomalías de datos), identificando situaciones "anormales" que deben revisarse. Por ejemplo: variables definidas dos veces consecutivas
 - detectar de forma DINÁMICA defectos en el programa identificando caminos en los que se manipulan las variables. Por ejemplo: ejecutar todos los caminos en los que una variable es definida y usada



Métodos estructurales. Observaciones

■ Los métodos estructurales:

- * Analizan el código y obtienen una representación en forma de grafo
- * A continuación seleccionan un conjunto de caminos según algún criterio
- * Obtienen un conjunto de casos de prueba que ejercitan dichos caminos

■ Dependiendo del método utilizado, obtendremos conjuntos DIFERENTES de casos de prueba:

- * Pero el conjunto obtenido será EFECTIVO y EFICIENTE!!!!

■ Las técnicas o métodos estructurales suelen aplicarse sólo a nivel de UNIDADES de programa

■ Los métodos estructurales no pueden detectar todos los defectos en el programa (faults, bugs)

- * Aunque seleccionemos todas las posibles entradas, no podremos detectar todos los defectos si "faltan caminos" en el programa. De forma intuitiva, diremos que falta un camino en el programa si no existe el código para manejar una determinada condición de entrada.

- P.ej. si la implementación no prevé que un divisor pueda tener un valor cero, entonces no se incluirá el código necesario para manejar esta situación



Control flow testing

- Los dos tipos de sentencias básicas en un programa son:
 - * Sentencias de asignación
 - Por defecto se ejecutan de forma secuencial
 - * Sentencias condicionales
 - Alteran el flujo de control secuencial en un programa
- Las llamadas a "funciones" son un mecanismo para proporcionar abstracción en el diseño de un programa
 - * Una llamada a una "función" cede el control a dicha función
- Podemos considerar que una unidad de programa tiene un punto de entrada y de salida bien definidos
 - * La ejecución de una secuencia de instrucciones desde el punto de entrada al de salida de una unidad de programa se denomina CAMINO (path)
 - * Puede haber un número potencialmente grande de caminos, incluso infinito, en una unidad de programa
 - * Un valor específico de entrada provoca que se ejecute un camino específico en el programa

Grafo de flujo de control (CFG)

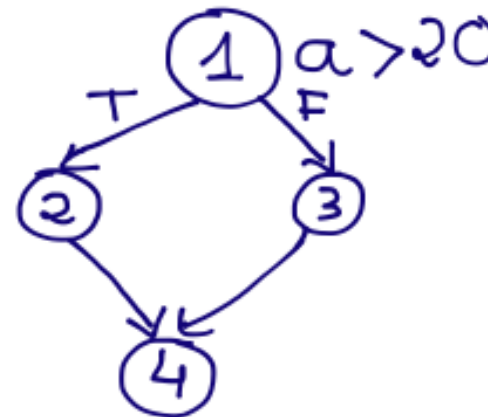
■ Un CFG es una representación gráfica de una unidad de programa

* Podemos utilizar diferentes notaciones para representar el CFG:

□ Utilizaremos una representación en forma de GRAFO DIRIGIDO:

- Cada **nodo** representa una o más sentencias secuenciales y/o una ÚNICA CONDICIÓN (así como los puntos de entrada y de salida de la unidad de programa)
 - Cada nodo estará etiquetado con un entero cuyo valor será único
 - Si un nodo contiene una condición anotaremos a su derecha dicha condición
- Las **aristas** representan el flujo de ejecución entre dos conjuntos de sentencias (representadas en los nodos)
 - Si uno nodo contiene una condición etiquetaremos las aristas que salen del nodo con "T" o "F" dependiendo de si el valor de la condición que representa es cierto o falso.

```
if (a > 20) {  
    k = "valor correcto"  
} else {  
    k = "repita entrada"  
}
```





Construcción de un CFG (I)

- Representa los grafos de flujo asociados a los siguientes códigos java:

```
if ((a > 1) && (a < 200)) {  
    ...  
}
```



Inténtalo!

```
1. if ((a != b) && (a != c) && (b != c)) {  
2.     ...  
3. } else {  
4.     if (a == b) {  
5.         if (a == c) {  
6.             ...  
7.         }  
8.     } else {  
9.         ...  
10.    }  
11. }
```




Construcción de un CFG (II)

- Fíjate que para poder crear correctamente el CFG necesitamos conocer BIEN el funcionamiento del subconjunto de sentencias de CONTROL del lenguaje de programación utilizado
- Por ejemplo, en Java se utilizan sentencias try..catch, para capturar y tratar las excepciones:

```
1. try {  
2.     s1; //puede lanzar Exception1;  
3.     s2; //no lanza ninguna excepción  
4.     ...  
5. } catch (Exception1 e) {  
6.     ...  
7. } finally {  
8.     ...  
9. }
```



Inténtalo!



Criterios de selección de caminos

- Estructuralmente un camino es una secuencia de instrucciones en una unidad de programa
- Semánticamente un camino es una instancia de una ejecución de una unidad de programa
- Es necesario escoger un conjunto de caminos con algún criterio de selección, de forma que, por ejemplo:
 - * Todas las construcciones del programa se ejerciten al menos una vez
 - * No generaremos entradas para los tests de forma que se ejecute el mismo camino varias veces. Aunque, si cada ejecución del camino actualiza el estado del sistema, entonces múltiples ejecuciones del mismo camino pueden no ser idénticas
- Algunos criterios de selección utilizados son:
 - * Elegimos todos los caminos,
 - * Elegimos el conjunto mínimo de caminos para conseguir ejecutar todas las sentencias
 - * Elegimos el conjunto mínimo de caminos para conseguir ejecutar todas las condiciones....



McCabe's Basis path method

- Es un método de diseño de pruebas de caja blanca cuyo OBJETIVO es ejercitar (ejecutar) cada **camino independiente** en el programa
 - * Fue propuesto inicialmente por Tom McCabe en 1976. Este método permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esta medida como guía para la definición de un conjunto básico de caminos de ejecución
 - * El método también se conoce como "Método del camino básico"
- Si ejecutamos TODOS los caminos independientes, estaremos ejecutando TODAS las **sentencias** del programa, al menos una vez
 - * Además estaremos garantizando que TODAS las **condiciones** se ejecutan en sus vertientes verdadero/falso
- ¿Qué es un camino independiente?
 - * Es un camino en un grafo de flujo (CFG) que difiere de otros caminos en al menos un nuevo conjunto de sentencias y/o una nueva condición (Pressman, 2001)



Descripción del método

1. Construir el grafo de flujo del programa (CFG) a partir del código a probar
2. Calcular la complejidad ciclomática (CC) del grafo de flujo
3. Obtener los caminos independientes del grafo
4. Determinar los datos de prueba de entrada de la unidad a probar de forma que se ejerciten todos los caminos independientes
5. Determinar el resultado esperado para cada camino, en función de la especificación de la unidad a probar

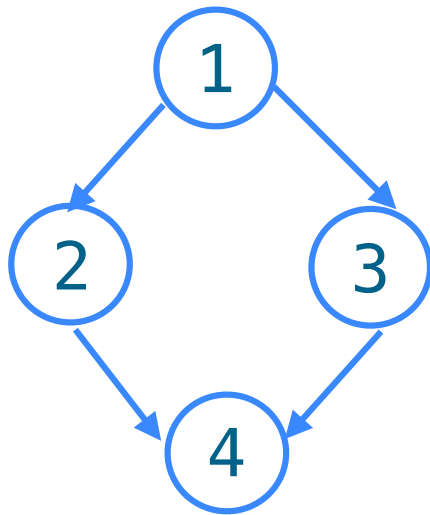
Tabla resultante del diseño de las pruebas:

Camino	Datos Entrada	Resultado Esperado	Resultado Real
C1	d1, d2, ...dn	r1	
C2	d1, d2, ...dn	r2	



Complejidad ciclomática (I)

- Es una métrica que proporciona una medida de la complejidad lógica de un componente software
- Se calcula a partir del grafo de flujo:
 - * $CC = \text{número de arcos} - \text{número de nodos} + 2$
- El valor de CC indica el MÁXIMO número de caminos independientes en el grafo
- Ejemplo:



$$CC = 4 - 4 + 2 = 2$$

A mayor CC, mayor complejidad lógica, por lo tanto, mayor esfuerzo de mantenimiento, y también mayor esfuerzo de pruebas!!!

El valor máximo de CC comúnmente aceptado como "tolerable" es 10

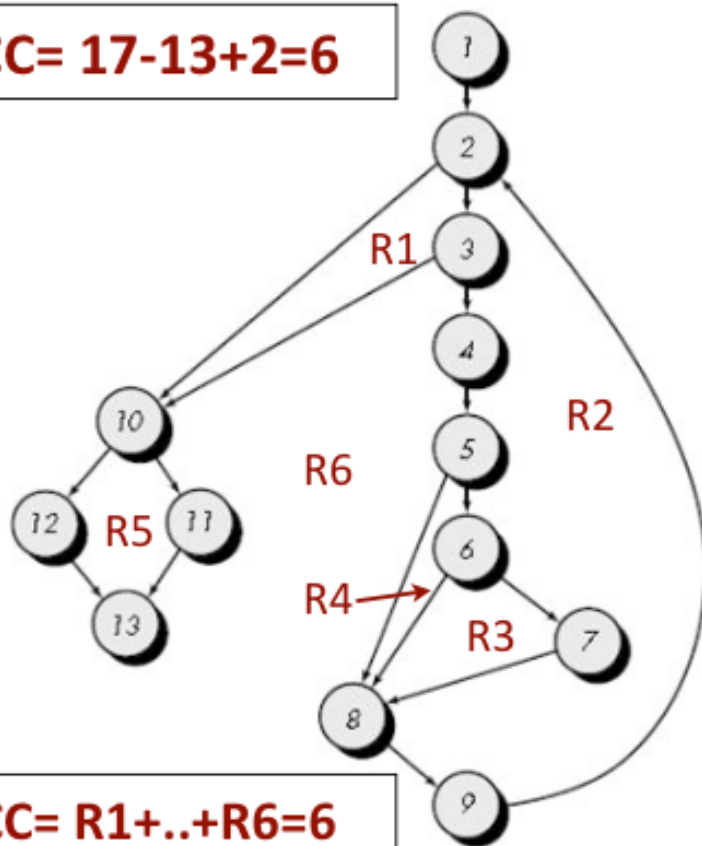
Complejidad ciclomática (II)

■ Formas alternativas de obtener la complejidad ciclomática:

- * $CC = \text{número de arcos} - \text{número de nodos} + 2$
- * $CC = \text{número de regiones}$
- * $CC = \text{número de condiciones} + 1$

¡CUIDADO!: Las dos últimas formas de cálculo son aplicables SOLO si el código es totalmente estructurado (no saltos incondicionales)

$$CC = 17 - 13 + 2 = 6$$



$$CC = R1 + \dots + R6 = 6$$

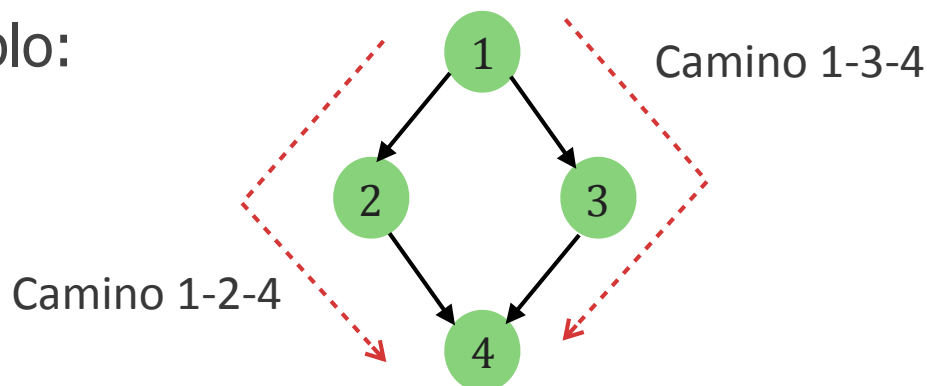
```
...
i=1;
total.input=total.valid=0;
sum=0;
do while ((value[i] <> -999) && (total.input<100))
{
    total.input+=1;
    if ((value[i]>= minimum) && (value[i]<= maximum))
    {
        total.valid+=1;
        sum= sum + value[i];
    }
    i+=1;
}
if (total.valid >0) {
    average= sum/total.valid;
} else average = -999;
return average;
```

$$CC = 5 + 1 = 6$$

Caminos independientes

- Buscamos (como máximo) tantos caminos independientes como valor obtenido de CC
 - * Cada camino independiente contiene un nodo, o bien una arista, que no aparece en el resto de caminos independientes
 - * Con ellos recorreremos TODOS los nodos y TODAS las aristas del grafo

■ Ejemplo:



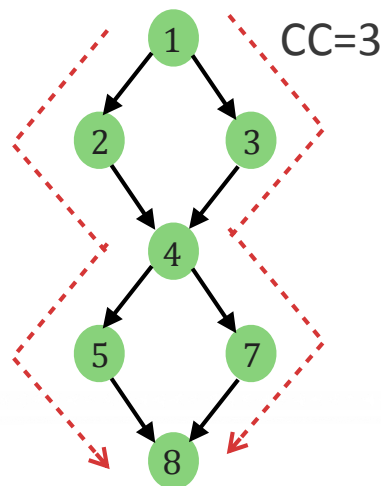
- Es posible que con un número inferior a CC recorramos todos los nodos y todas las aristas

* Ejemplo:

```

if (a >= 20) {
    result = 0;
} else {
    result = 10;
}
if (b >= 20) {
    result = 0;
} else {
    result = 10;
}

```



opción 1:

C1 = 1-3-4-7-8
C2 = 1-3-4-5-8
C3 = 1-2-4-5-8

opción 2:

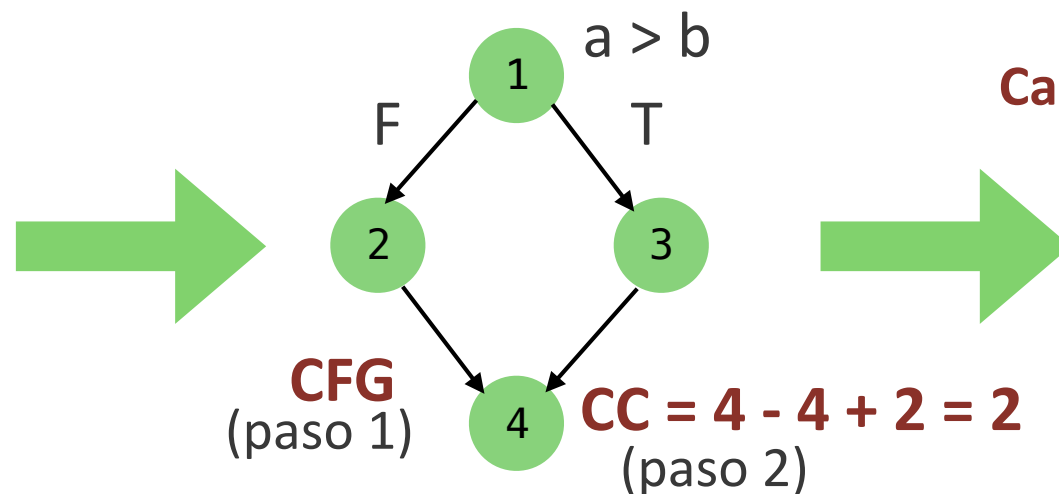
C1 = 1-3-4-7-8
C2 = 1-2-4-5-8

Ambas opciones son válidas

Ejemplo de aplicación del método

- Método que compara dos enteros a y b, y devuelve 20 en caso de que el valor a sea mayor que b, y cero en caso contrario:

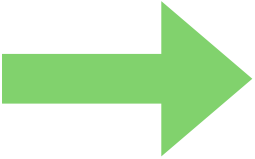
```
if (a > b) {
    result = 20
} else {
    result = 0
}
```



(paso 3)
Caminos independientes

C1 = 1-3-4
C2 = 1-2-4

Tabla resultante del diseño de casos de prueba

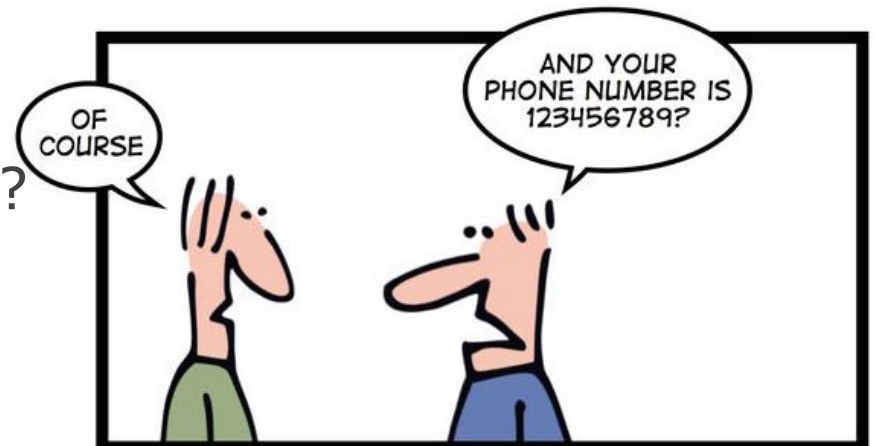
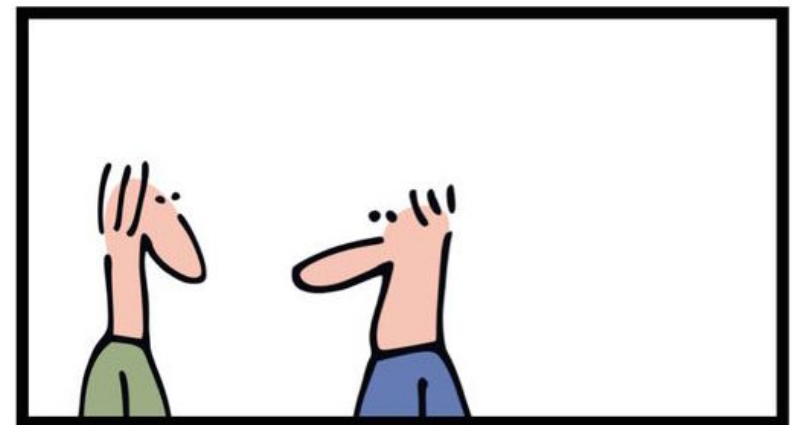
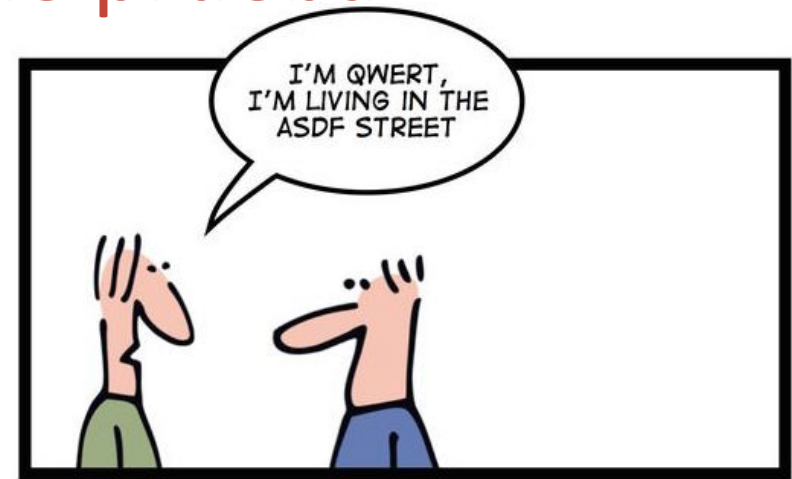


Camino	Datos Entrada	Resultado Esperado	Resultado Real
C1	a = 20; b = 10	result = 20	
C2	a = 10; b = 20	result = 0	

(paso 4) (paso 5)
Valores de entrada Resultado esperado

Concreción de los casos de prueba

- Recuerda que los datos de prueba y la salida esperada deben ser SIEMPRE valores CONCRETOS:
 - * Los casos de prueba tienen que poder REPETIRSE, y por lo tanto, dos ejecuciones del mismo test, tienen que tener valores de entrada IDÉNTICAS
 - * Por ejemplo:
 - D1= "cualquier dni válido" es INCORRECTO
 - D1= "12345678" es CORRECTO
- ¿Por qué repetir los tests?
 - * Piensa en razones por las que va a ser necesario repetir los tests
- ¿Qué ocurre si un test no puede repetirse?





Observaciones sobre los datos de prueba

- Normalmente los datos de entrada y/o salida podremos identificarlos en los parámetros del elemento a probar, (y/o valores de retorno). Pero esto NO siempre es así:
 - * Por ejemplo, supongamos que queremos probar el siguiente método:
 - nuevo_cliente() añade un nuevo cliente a lista de clientes de nuestra tienda virtual. Si el cliente que se pasa como entrada ya es cliente de la tienda, entonces el método no hará nada. Supongamos que el prototipo del método es:
 - public void nuevo_cliente(Client cli), ¿cuáles son las entradas y salidas?
 - En este caso, el estado de la lista de clientes "antes" de llamar al método lo tenemos que considerar como una entrada, y el estado de la lista de clientes "después" de llamar al método lo consideraremos una salida
- Si utilizamos un lenguaje orientado a objetos, como Java, consideraremos como datos de entrada los valores concretos de "cada atributo" del objeto
 - * Por ejemplo, supongamos que la clase Cliente está formada por los campos dni, nombre, dirección, y teléfonos. Un ejemplo de caso de prueba podría ser éste:
 - dni=12345678, nombre= pepe, dirección=calle del mar, teléfonos=(12345,99999)



Ejemplo: Búsqueda binaria

```
//Asumimos que la lista de elementos está ordenada de forma ascendente
class BinSearch
public static void search (int key, int [ ] elemArray, Result r) {
    int bottom = 0; int top = elemArray.length -1;
    int mid; r.found= false; r.index= -1;
    while (bottom <= top) {
        mid = (top+bottom)/2;
        if (elemArray [mid] == key) {
            r.index = mid;
            r.found = true;
            return;
        } else {
            if (elemArray [mid] < key)
                bottom = mid + 1;
            else top = mid -1;
        }
    } //while loop
} //search
} //class
```

Especificación del método search():
Dado un vector de enteros ordenados ascendentemente, y dado un entero (key) como entrada, el método search() busca la posición de key en el vector y devuelve el valor found=true si lo encuentra, así como su posición en el vector (dada por index). Si el valor de key no está en el vector, entonces devuelve el valor found=false



Vamos a identificar los nodos del grafo

//Asumimos que la lista de elementos está ordenada de forma ascendente

class BinSearch

public static void search (int key, int [] elemArray, Result r)

```
{  int bottom = 0;      int top = elemArray.length -1;
  int mid;              r.found= false; r.index= -1;
```

```
  while (bottom <= top) {
```

```
    mid = (top+bottom)/2;
```

```
    if (elemArray [mid] == key) {
```

```
      r.index = mid;
```

```
      r.found = true;
```

```
      return;
```

```
    } else {
```

```
      if (elemArray [mid] < key)
```

```
        bottom = mid + 1;
```

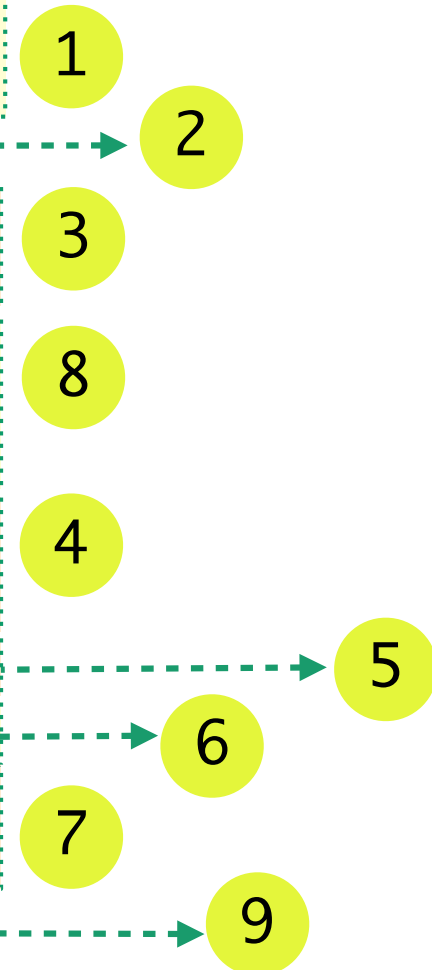
```
      else top = mid -1;
```

```
    }
```

```
  } //while loop
```

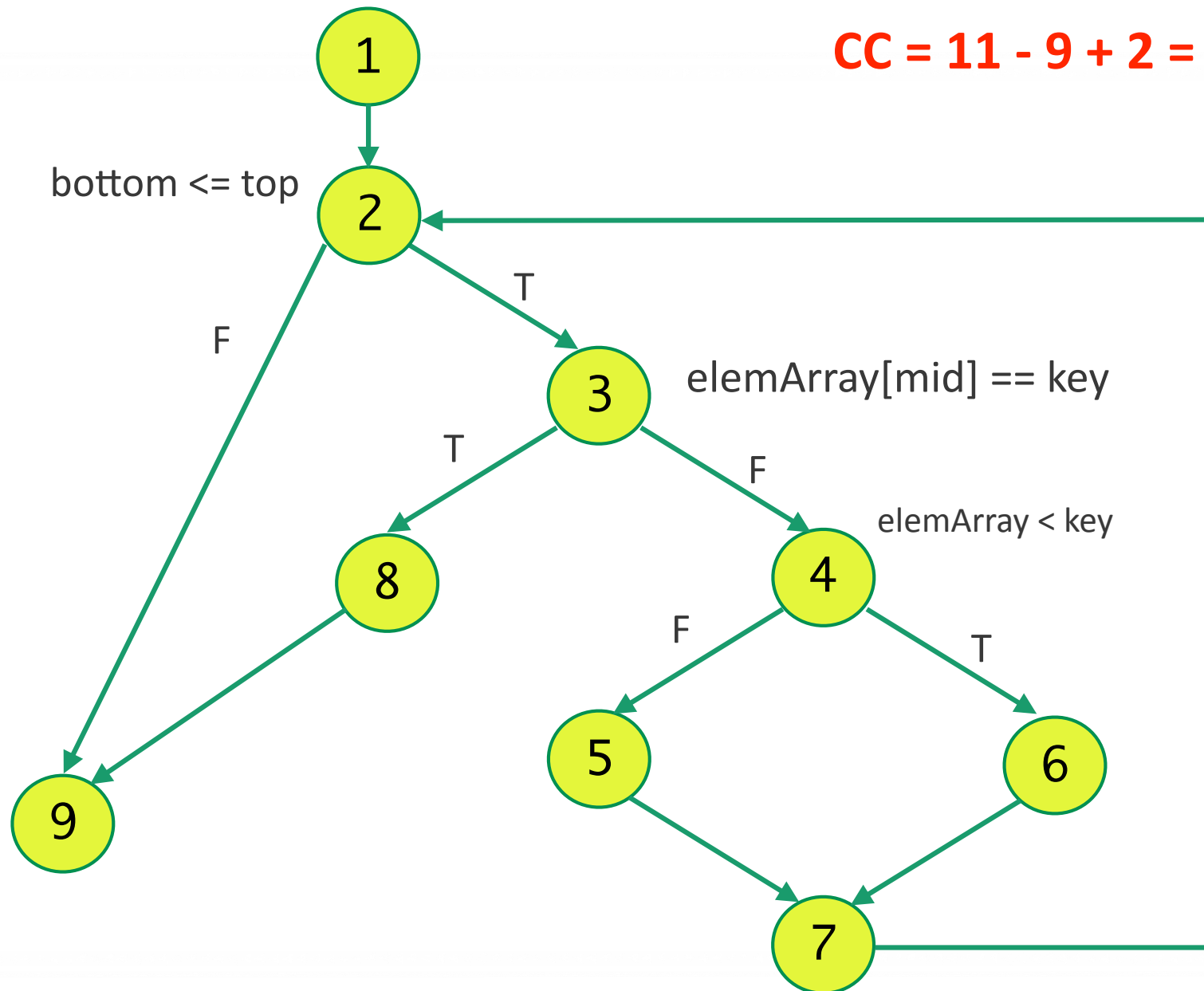
```
} //search
```

```
} //class
```



Grafo asociado y valor de CC

$$CC = 11 - 9 + 2 = 4$$





Caminos independientes

■ Posible conjunto de caminos independientes

* C1: 1, 2, 3, 4, 6, 7, 2, 9

* C2: 1, 2, 3, 4, 5, 7, 2, 9

* C3: 1, 2, 3, 8, 9

En este ejemplo, con tres caminos podemos recorrer todos los nodos y todas las aristas

■ Ejercicio: Calcula la tabla resultante:

Camino	Datos Entrada	Resultado Esperado	Resultado Real
C1	key= elemArray=	r.found= r.index=	
C2	key= elemArray=	r.found= r.index=	
C3	key= elemArray=	r.found= r.index=	

Para indicar el valor del resultado esperado necesitamos conocer la especificación del método



Ejercicios propuestos (I)

- Calcula la CC para cada uno de estos códigos Java:

```
public void divide(int numberToDivide, int numberToDivideBy) throws BadNumberException{
    if(numberToDivideBy == 0){
        throw new BadNumberException("Cannot divide by 0");
    }
    return numberToDivide / numberToDivideBy;
}
```

Código 1

```
public void callDivide(){
    try {
        int result = divide(2,1);
        System.out.println(result);
    } catch (BadNumberException e) {
        //do something clever with the exception
        System.out.println(e.getMessage());
    }
    System.out.println("Division attempt done");
}
```

Código 2

```
public void openFile(){
    try {
        // constructor may throw FileNotFoundException
        FileReader reader = new FileReader("someFile");
        int i=0;
        while(i != -1){
            //reader.read() may throw IOException
            i = reader.read();
            System.out.println((char) i );
        }
        reader.close();
        System.out.println("--- File End ---");
    } catch (FileNotFoundException e) {
        //do something clever with the exception
    } catch (IOException e) {
        //do something clever with the exception
    }
}
```

Código 3



Ejercicios propuestos (II)

- Diseñar los casos de prueba para el método validar_PIN(), cuyo código es el siguiente:

```
1.  public class Cajero {
2.      ...
3.  public boolean validar_PIN (Pin pinNumber) {
4.      boolean pin_valido= false;
5.      String codigo_respuesta="GOOD";
6.      int contador_pin= 0;
7.
8.      while ((!pin_valido) && (contador_pin <= 2) &&
9.              !codigo_respuesta.equals("CANCEL")) {
10.         codigo_respuesta = obtener_pin(Pin pinNumber);
11.         if (!codigo_respuesta.equals("CANCEL")) {
12.             pin_valido = comprobar_pin(pinNumber);
13.             if (!pin_valido) {
14.                 System.out.println("PIN inválido, repita");
15.                 contador_pin=contador_pin+1;
16.             }
17.         }
18.     }
19.     return pin_valido;
20. }
21. ...
22. }
```

la especificación la tenéis a continuación →



Ejercicio propuesto 2

■ Especificación del método validar_PIN():

- * El método validar_PIN() anterior valida un código numérico de cuatro cifras (objeto de la clase Pin) introducido a través de un teclado (asumimos que en el teclado solamente hay teclas numéricas (0..9), y una tecla para cancelar). El método obtener_pin() “lee” el código introducido por teclado creando una nueva instancia de un objeto Pin, y devuelve “GOOD” si no se pulsa la tecla para cancelar, o “CANCEL” si se ha pulsado la tecla para cancelar (carácter ‘\’). El método comprobar_pin() verifica que el código introducido tiene cuatro cifras y se corresponde con la contraseña almacenada en el sistema para dicho usuario, devolviendo cierto o falso, en función de ello. El usuario dispone de tres intentos para introducir un pin válido, en cuyo caso el método validar_PIN() devuelve cierto, así como el número de pin, y en caso contrario devuelve falso.



Y ahora vamos al laboratorio...

No vamos a usar ninguna herramienta software

Identificaremos casos de prueba utilizando el método del CAMINO BÁSICO

```
package ppss;  
  
public class Matricula {  
    public float calculaTasaMatricula(int edad,  
        boolean familiaNumerosa,  
        boolean repetidor) {  
        float tasa = 500.00f;  
  
        if ((edad <= 25) && (!familiaNumerosa) || (!repetidor)) {  
            tasa = tasa + 1500.00f;  
        } else {  
            if ((familiaNumerosa) || (edad > 65)) {  
                tasa = tasa / 2;  
            }  
            if ((edad >= 50) && (edad < 65)) {  
                tasa = tasa - 100.00f;  
            }  
        }  
        return tasa;  
    }  
}
```

CFG



CC

CC = ...

camínos independientes

C1: 1-2-4-... -14

C2: 1-3-6-... -14

...

CM: 1-2-7-... -14

($M \leq CC$)

tabla de casos de prueba

Camino	Datos Entrada	Resultado Esperado	Resultado Real
C1	d1=...	r1	
..			
CM	d1=...	rM	



Referencias bibliográficas

- A practitioner's guide to software test design. Lee Copeland. Artech House Publishers. 2004
 - * Capítulo 10: Control Flow Testing
- Pragmatic software testing. Rex Black. Wiley. 2007
 - * Capítulo 21: Control Flow Testing
- Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
 - * Capítulo 4: Control Flow Testing