

# PPSS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2022-23

## Sesión S01: Pruebas: proceso de diseño: caja blanca



Maria Isabel Alfonso Galipienso  
Universidad de Alicante [eli@ua.es](mailto:eli@ua.es)

Pruebas y proceso de pruebas.

Diseño de casos de prueba: structural testing

- Objetivo: obtener una tabla de casos de prueba a partir del conjunto de **comportamientos programados**
- El conjunto obtenido debe detectar el máximo número posible de defectos en el código, con el mínimo número posible de casos de prueba

**Control flow testing:** Método del camino básico

- Paso 1: Análisis de código: Construcción del CFG
- Paso 2: Selección de caminos: Cálculo de CC y caminos independientes
- Paso 3. Obtención del conjunto de casos de prueba

Ejemplos y ejercicios

Vamos al laboratorio...

# DEFINICIÓN DEL PROCESO DE PRUEBAS

"Testing is the process of executing a program with the intent of **finding errors**. If our goal is to show the absence of errors, we will discover fewer of them. If our goal is to show the presence of errors, we will discover a large number of them"

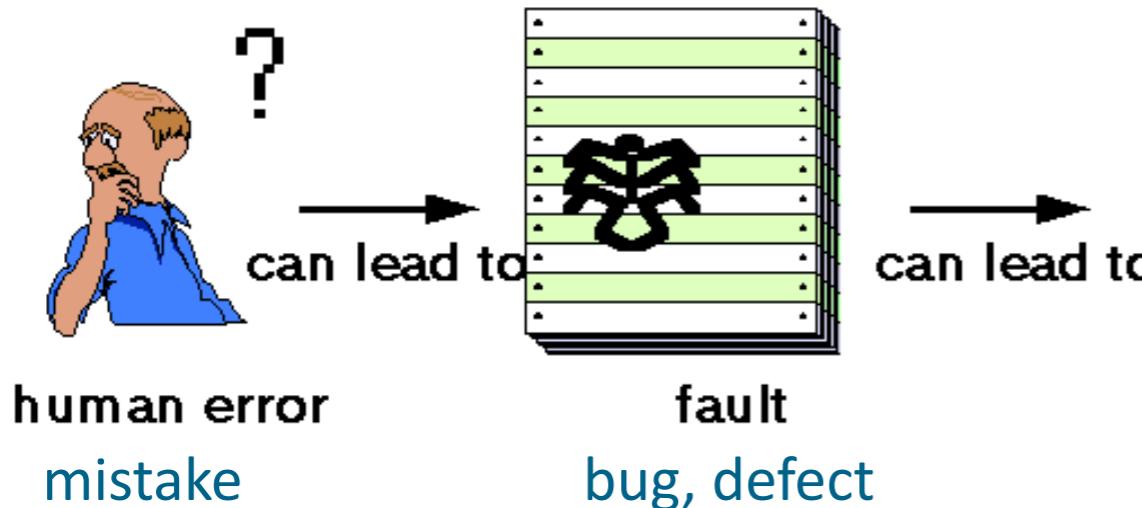
Glenford J. Myers (1979)

Las pruebas son un conjunto de actividades conducentes a conseguir alguno de estos objetivos:

- \* **Encontrar defectos**
- \* **Evaluar el nivel de calidad del software**
- \* **Obtener información para la toma de decisiones**
- \* **Prevenir defectos**

(ISQTB Foundation Level Syllabus -2011)

## HAY MUCHOS TIPOS DE "ERRORES"!!



Las pruebas muestran la **PRESENCIA de defectos** (no pueden demostrar la ausencia de los mismos. Si no se encuentra un defecto, no significa que no los haya)

error: resultado esperado diferente del real

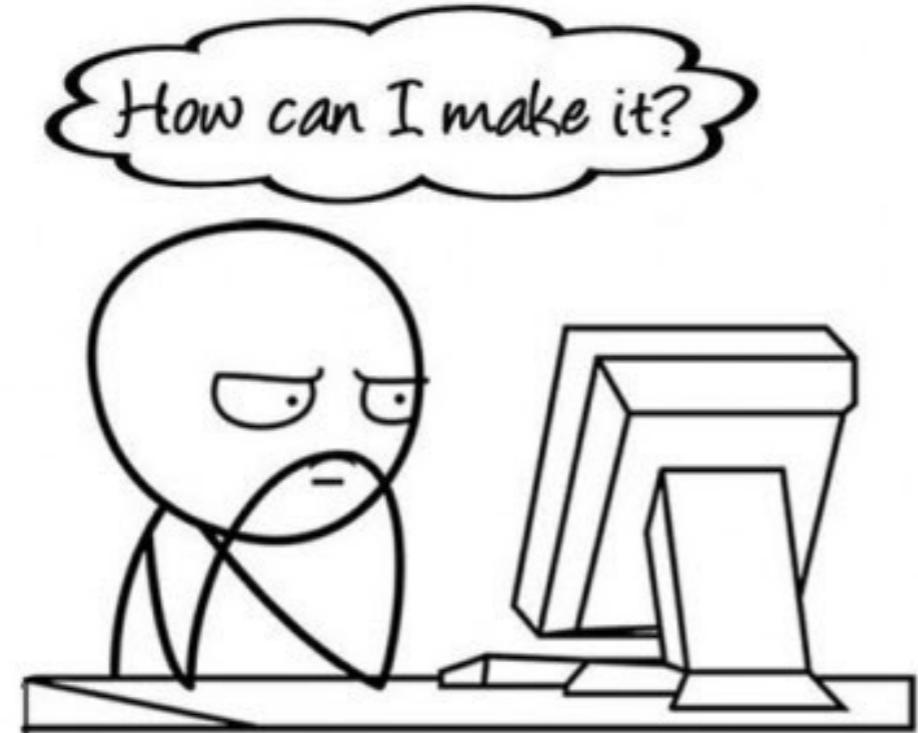


# DEFINICIÓN DEL PROCESO DE PRUEBAS

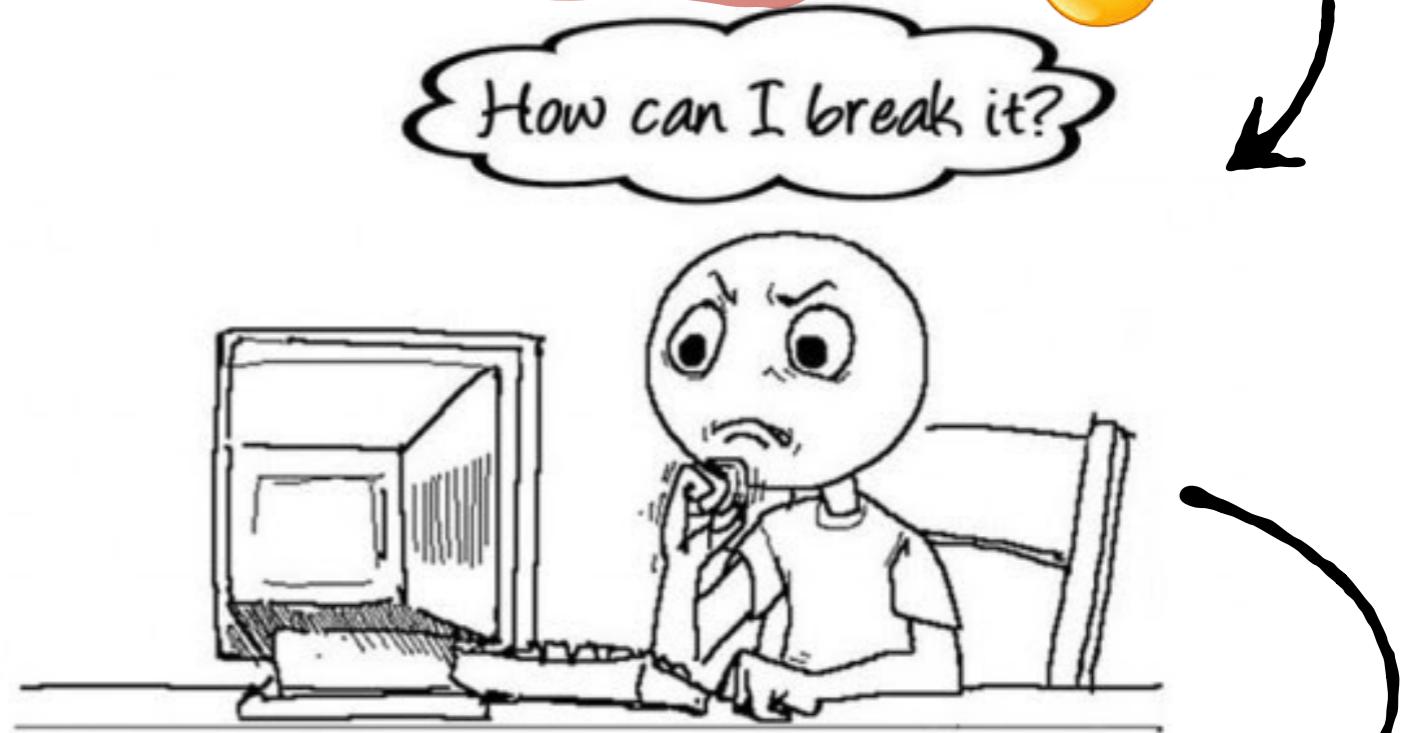
P

P

DEVELOPER



TESTER



## A LOS USUARIOS NO LES GUSTAN LOS ERRORES!!



Uno de los objetivos a cumplir para que un proyecto tenga éxito es que el software **satisfaga las expectativas del cliente**

Si las expectativas del cliente no se satisfacen, éste se sentirá justificadamente agraviado

Una prueba tiene EXITO cuando revela la **PRESENCIA de defectos**.

Además, no es suficiente el encontrar defectos, el programa debe satisfacer las necesidades y expectativas del cliente)



# ACTIVIDADES DEL PROCESO DE PRUEBAS



SEGÚN ISQTB FOUNDATION LEVEL SYLLABUS (2011)

## 1 PLANIFICACIÓN y control de las pruebas

*Definimos los objetivos de las pruebas, y en todo momento tenemos que asegurarnos de que cumplimos con esos objetivos (p.ej. queremos realizar pruebas sobre el 95% de código)*

## 2 DISEÑO de las pruebas

*Es el proceso más importante, si queremos efectivamente cumplir los objetivos marcados. Básicamente consiste en decidir con qué datos de entrada concretos vamos a probar el código, de forma que seamos capaces de detectar el máximo número de errores posibles, en el menor tiempo posible*

## 3 IMPLEMENTACIÓN y ejecución de las pruebas

*Creamos código, para probar nuestro código!!*



*...No, no nos hemos vuelto locos. La idea es que podemos ejecutar las pruebas pulsando un botón, en lugar de hacerlo de forma "manual". Lógicamente, hay que prestarle mucha atención al código de pruebas para que efectivamente nos ayude a conseguir nuestro objetivo*

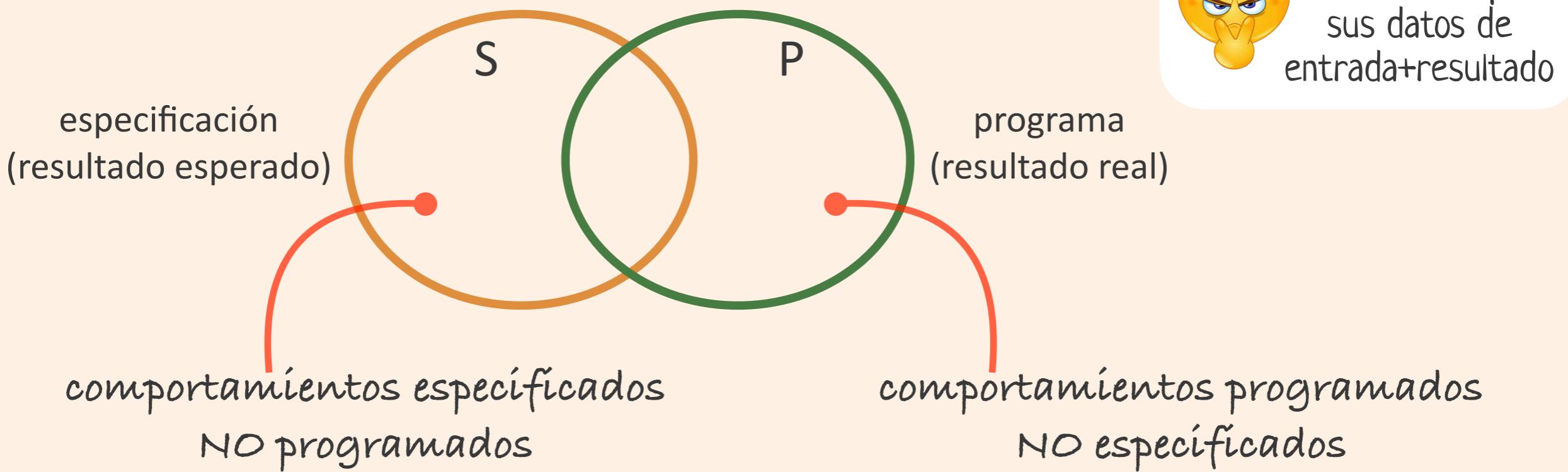
## 4 EVALUACIÓN del proceso de pruebas y emitir un informe

*Aplicamos las métricas adecuadas para comprobar si hemos alcanzado los objetivos de pruebas planificados*

# PRUEBAS Y COMPORTAMIENTO

S y P deberían ser idénticos!!!

- Las pruebas conciernen fundamentalmente al **comportamiento** del elemento a probar: ¿qué debería hacer aquello que estoy probando?
- Supongamos un universo de comportamientos de programa. Dado un programa y su especificación, consideraremos:
  - el conjunto S de comportamientos especificados para dicho programa
  - el conjunto P de comportamientos programados



Estos son los problemas con los que se enfrenta un tester!!!

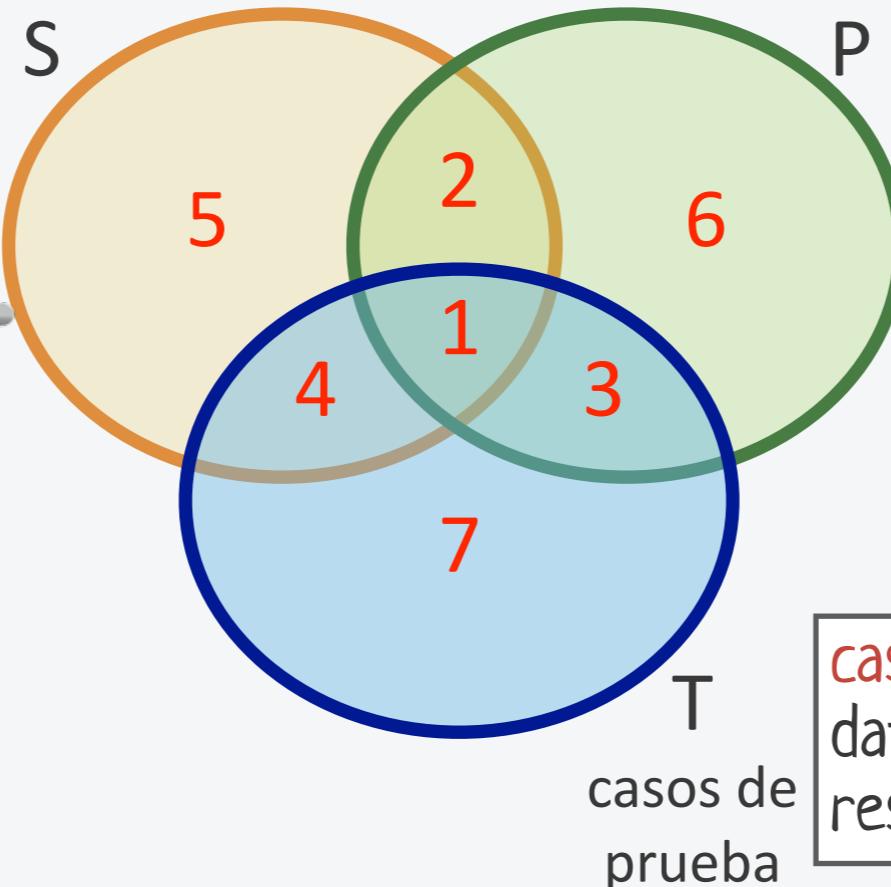
# COMPORTAMIENTOS ESPECIFICADOS, PROGRAMADOS, Y PROBADOS

P

- O Incluyamos el conjunto T de comportamientos probados a la figura anterior:

P

especificación  
(resultado esperado)



programa  
(resultado real)

Trabajaremos con  
técnicas de pruebas  
**DINÁMICAS**.

Una prueba **DINÁMICA**  
es aquella que requiere  
**EJECUTAR** el código para  
detectar la presencia de  
defectos

Cada una de estas  
regiones es importante

**caso de prueba** =  
datos concretos de entrada +  
resultado esperado



Indica qué representan las regiones:

1 2+5

2 1+4

3 3+7

4 2+6

5 1+3

6 4+7

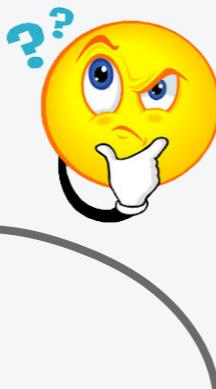
# DISEÑO DE CASOS DE PRUEBA

Selección de un subconjunto **mínimo** de casos de prueba para evidenciar el **máximo** número de errores posibles

P El resultado del proceso de diseño es una **Tabla de casos de prueba**. Cada fila contiene los datos de entrada y el resultado esperado de un comportamiento del programa

Tábla de casos de prueba

ID	d1	d2	...	Expected Output
C1	?	?	?	?
C2				
C3				
...				
CN?				



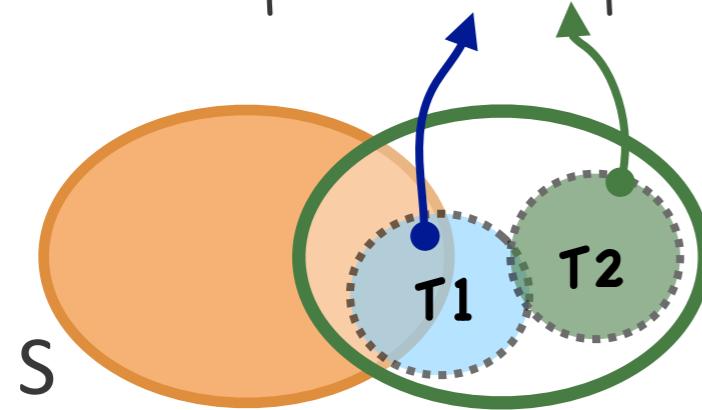
- ¿COMO RELLENAMOS LAS FILAS DE LA TABLA?
- ¿CUÁNTAS FILAS ES NECESARIO AÑADIR?

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

Cada FILA de la tabla es un CASO DE PRUEBA = datos entrada concretos + resultado esperado  
Cada caso de prueba representa un comportamiento a probar

## STRUCTURAL TESTING = DISEÑO DE PRUEBAS DE CAJA BLANCA

Comportamientos probados



Comportamientos implementados

Podemos detectar comportamientos no especificados  
Nunca podremos detectar comportamientos no implementados

- Consiste en determinar los **valores de entrada** de los casos de prueba a partir de la **IMPLEMENTACIÓN**.
- El **resultado esperado** se obtiene **SIEMPRE** de la especificación
- Los comportamientos probados podrán estar o no especificados
- Es esencial conocer conceptos de teoría de grafos para entender bien esta aproximación



# CUALQUIER MÉTODO BASADO EN EL CÓDIGO SIGUE ESTOS PRINCIPIOS!!!

P



Los métodos de diseño de casos de prueba basados en el CÓDIGO:

1. Analizan el código y obtienen una representación en forma de GRAFO
2. Seleccionan un conjunto de **caminos** en el grafo según algún criterio
3. Obtienen un conjunto de **casos de prueba** que ejercitan dichos caminos

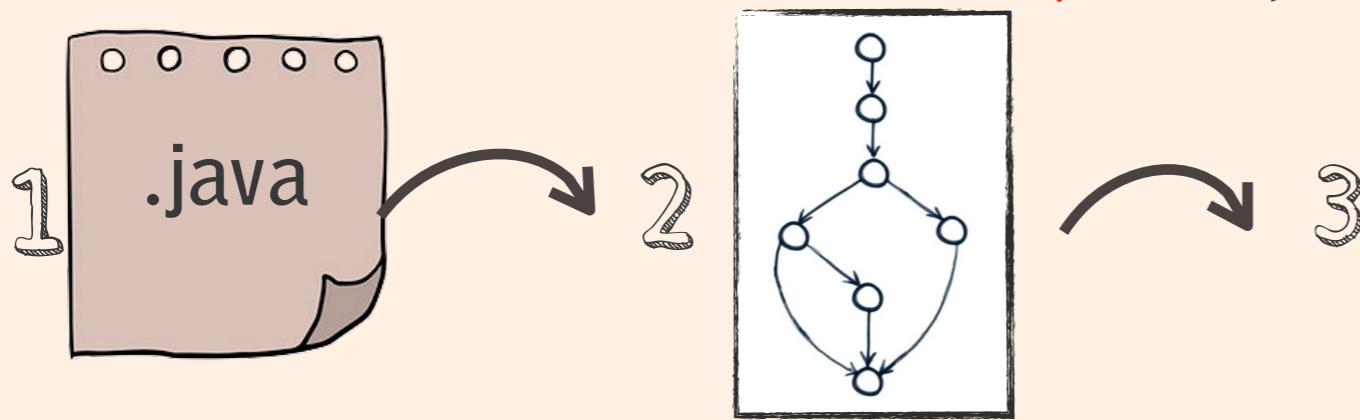


Tabla de casos de prueba

Test case	a	b	c	outcome

## OBSERVACIONES SOBRE LOS MÉTODOS ESTRUCTURALES

- Dependiendo del método concreto utilizado, obtendremos conjuntos DIFERENTES de casos de prueba. Pero el conjunto obtenido será **EFFECTIVO** y **EFICIENTE!!!!**
- Las técnicas o métodos estructurales son costosos de aplicar, por lo que suelen usarse sólo a nivel de **UNIDADES** de programa
- Los métodos estructurales **NO** pueden DETECTAR **TODOS** los defectos en el programa (defecto = fault = bug). 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.  
**Por ejemplo:** si la implementación no prevé que un divisor pueda tener un valor cero, entonces no se incluirá el código necesario para manejar esta situación

# GRAFO DE FLUJO DE CONTROL (CFG)

P  
P  
S

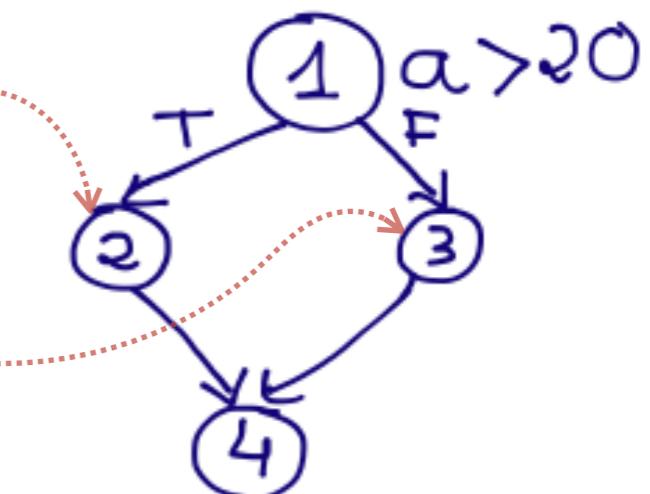
1 nodo representa cero o más sentencias secuenciales + cero o UNA!! condición

- Un CFG es una **representación gráfica** de los comportamientos de una **unidad** de programa. Usaremos un **GRAFO DIRIGIDO**, en donde:
  - Cada **nodo** representa cero 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 secuenciales (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.



TODAS las sentencias de la unidad a probar deben estar representadas en el grafo!!!

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



# P ANÁLISIS DE CÓDIGO BASADO EN EL FLUJO DE CONTROL

- 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

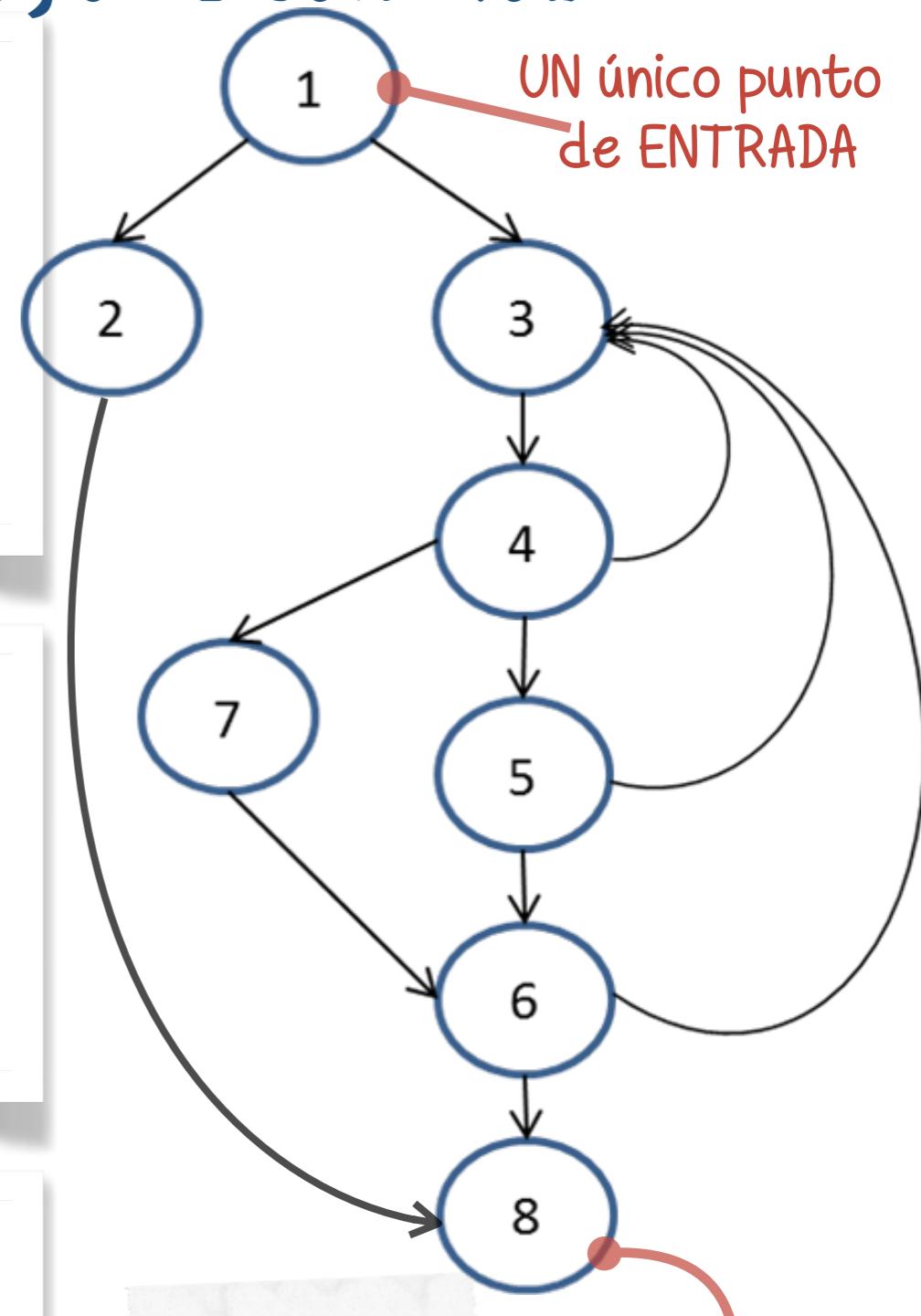
- Una llamada a una "función" cede el control a dicha función
- Dentro de la unidad a probar, la invocación de una "función" es una sentencia secuencial

Cada camino en el grafo se corresponde con un **COMPORTAMIENTO!!!**

La **EJECUCIÓN** de una secuencia de instrucciones desde el punto de entrada al de salida de una **unidad** de programa se denomina **CAMINO** (path)

En una unidad de programa puede haber un **número potencialmente grande** de caminos, incluso **infinito**.

Un conjunto de **valores específicos** de **entrada** provoca que se **ejecute** un camino específico en el programa



UNIDAD de programa = método Java

UN único punto de ENTRADA  
UN único punto de SALIDA

# CONSTRUCCIÓN DE UN CFG

Hay que tener claro cómo funcionan las sentencias de control del lenguaje!!!

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

P  
1

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

```
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. }
```

P  
2

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

P  
4

```
1. while ((a!=0) || (b < VALOR)){  
2.     switch (c) {  
3.         case 5:  
4.         case 10:  
5.         case 20:  
6.         case 50: cantidad += c; break;  
7.         case 0: if (cantidad < VALOR2)  
8.                     sentencia1;  
9.                     break;  
10.                default: sobrante += moneda;  
11.            }  
12.            if (cantidad != 0) {  
13.                sentencia2;  
14.            }  
15.        }
```

P  
5

```
1. do {  
2.     metodo1(var1, var2);  
3.     metodo2(var3);  
4.     ...  
5.     var2 = metodo3();  
6. } while (var2 !=VALOR);
```

P  
3



Inténtalo!

# CRITERIOS DE SELECCIÓN DE CAMINOS

Las pruebas EXHAUSTIVAS son  
IMPOSIBLES!!!

- **Estructuralmente** un camino es una secuencia de instrucciones en una unidad de programa (desde el punto de entrada, hasta el punto de salida)
- **Semánticamente** un camino es una instancia de una ejecución de una unidad de programa (comportamiento programado)
- Es necesario **seleccionar** un conjunto de caminos con algún **criterio de selección**. Algunos ejemplos son:
  - Elegimos todos los caminos del grafo
  - Elegimos el conjunto mínimo de caminos para conseguir ejecutar TODAS las SENTENCIAS al menos una vez
  - Elegimos el conjunto mínimo de caminos para conseguir ejecutar TODAS las CONDICIONES al menos una vez
- No generaremos entradas para los tests en los 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

Cada método de diseño usa un criterio de selección diferente!!!!

Ese criterio depende de un **OBJETIVO**. Pero cualquier método de diseño proporciona un conjunto de casos de prueba efectivos y eficientes para ese objetivo!!!



# MCCABE'S BASIS PATH METHOD



(Método del camino básico)

- Es un método de DISEÑO de pruebas de caja blanca que permite 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 procedural 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)

*El objetivo del método es éste!!!*



*El número de caminos independientes determinará el número de filas de la tabla. Cada fila detectará defectos en un determinado subconjunto de sentencias del programa (ejercitando un determinado comportamiento)*

# DESCRIPCIÓN DEL MÉTODO

S Recuerda que debes ser sistemático a la hora de aplicar los pasos y tener claro para qué estamos haciendo cada uno de ellos!!!

- P
- P
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 concretos de entrada (y salida esperada) de la unidad a probar, de forma que se ejerciten todos los caminos independientes. El resultado esperado siempre se obtendrá en función de la ESPECIFICACIÓN de la unidad a probar

## Tabla resultante del diseño de las pruebas:

Camino	Entrada 1	Entrada 2	...	Entrada n	Resultado Esperado
C1	d <sub>11</sub>	d <sub>12</sub>	...	d <sub>1n</sub>	r <sub>1</sub>
...					
C2	d <sub>21</sub>	d <sub>22</sub>	...	d <sub>2n</sub>	r <sub>2</sub>

Recuerda que los valores de entrada y salida deben ser CONCRETOS!!!

Una columna para CADA dato de entrada

Una columna para CADA dato de salida

# P COMPLEJIDAD CICLOMÁTICA

Determina el número de FILAS de la tabla!!

P

- Es una MÉTRICA que proporciona una medida de la **complejidad lógica** de un componente software

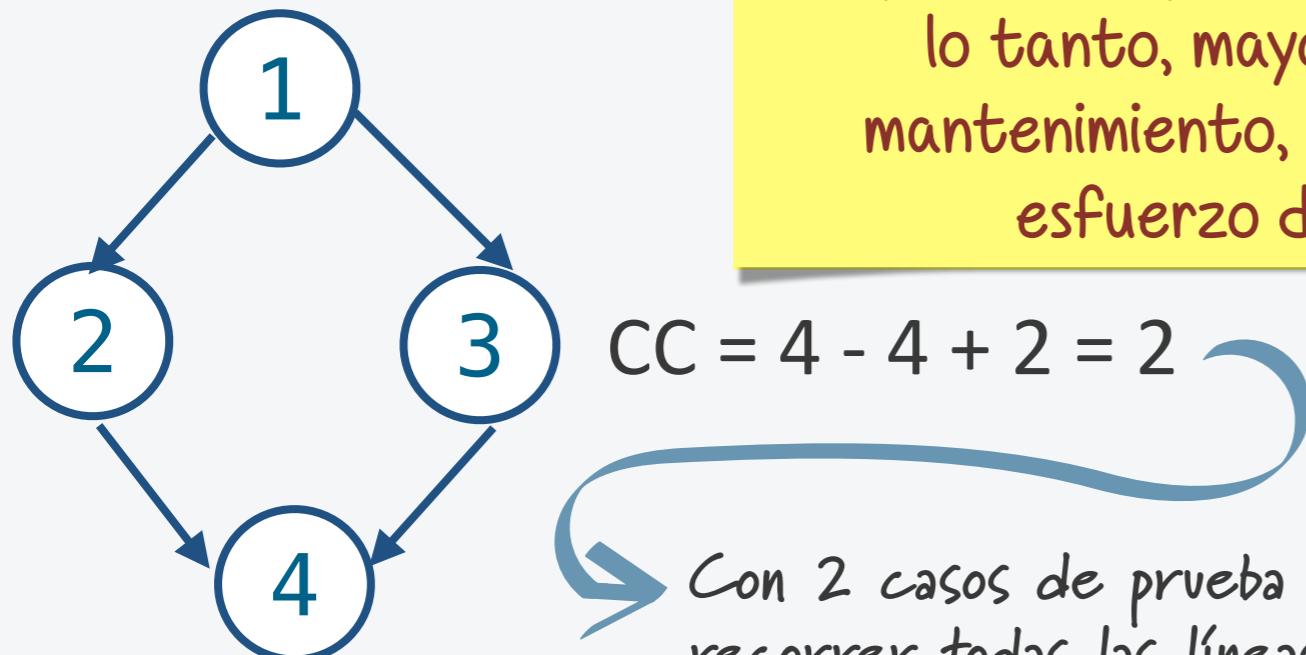
- Se calcula a partir del grafo de flujo:

$$CC = \text{número de arcos} - \text{número de nodos} + 2$$

- El valor de CC indica el MÁXIMO número de **caminos independientes** en el grafo

Un camino independiente aporta un nodo o una arista nuevas al conjunto de caminos

- Ejemplo:



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

Con 2 casos de prueba podemos recorrer todas las líneas y todas las condiciones en sus vertientes verdadera y falsa

Se puede reducir la complejidad lógica refactorizando el código para incrementar el nivel de abstracción (modularizar)

# OTRAS FORMAS DE CALCULAR CC

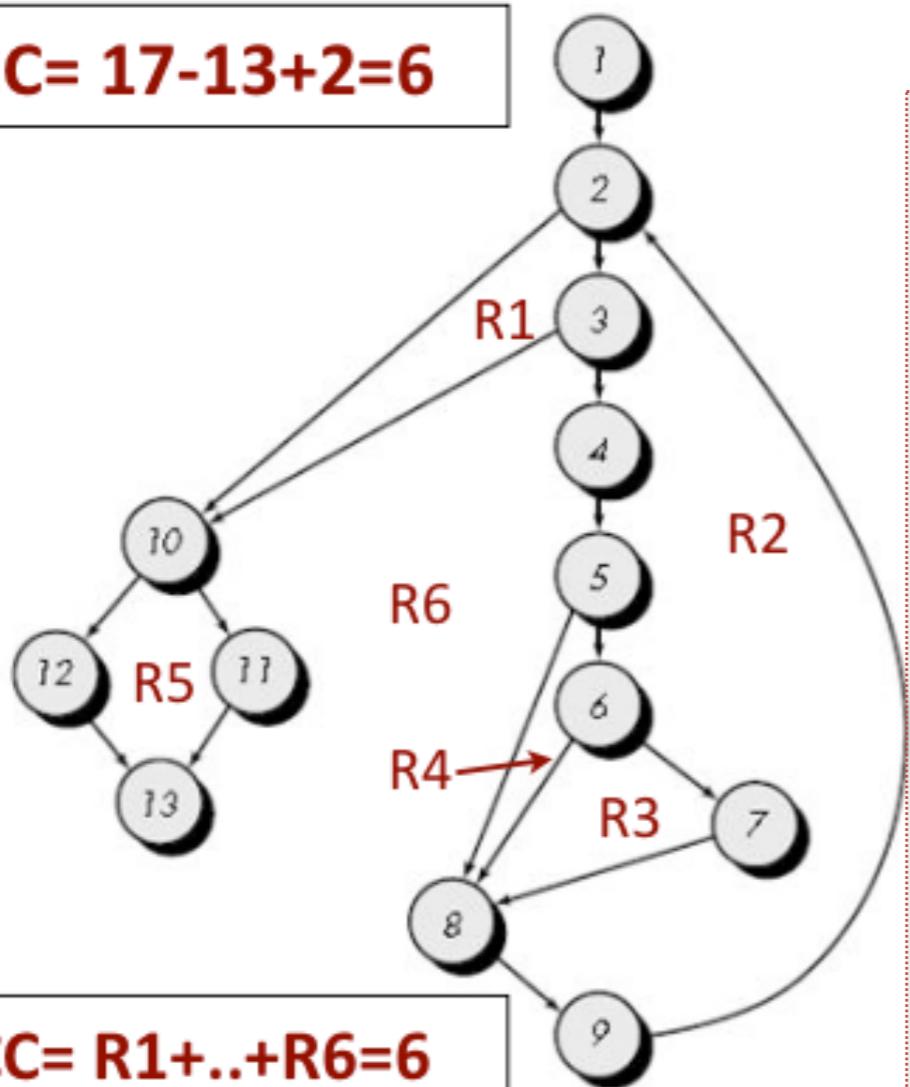
**CC** = número de arcos – número de nodos + 2

**CC** = número de regiones

**CC** = 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;
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;

```

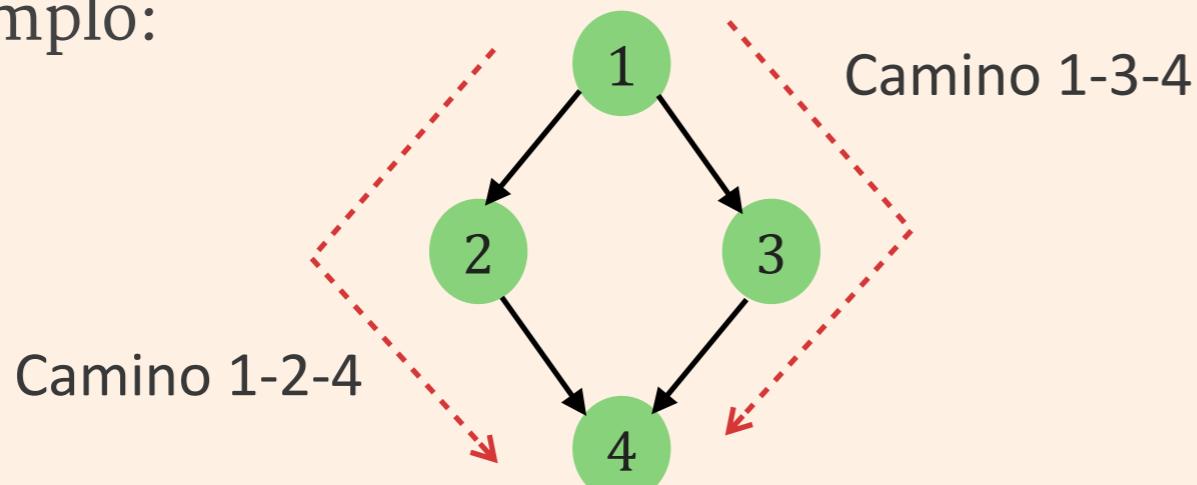
$$CC = 5 + 1 = 6$$

# CAMINOS INDEPENDIENTES

Cada camino independiente recorre un nodo o una arista (cómo mínimo) que no se había recorrido antes!!!

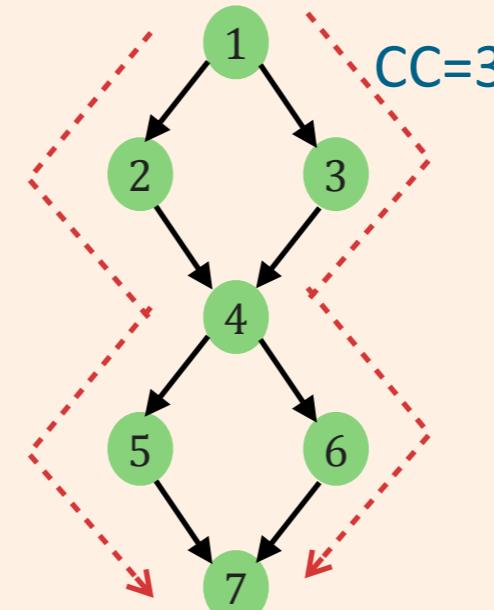
- Buscamos (como máximo) tantos caminos independientes como valor obtenido de CC
  - Cada camino independiente contiene un nodo, o bien una arista, que no aparece en los caminos independientes anteriores
  - Con ellos recorreremos TODOS los nodos y TODAS las aristas del grafo

- 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 = 1-3-4-6-7  
C2 = 1-3-4-5-7  
C3 = 1-2-4-6-7

opción 2:

C1 = 1-3-4-6-7  
C2 = 1-2-4-5-7

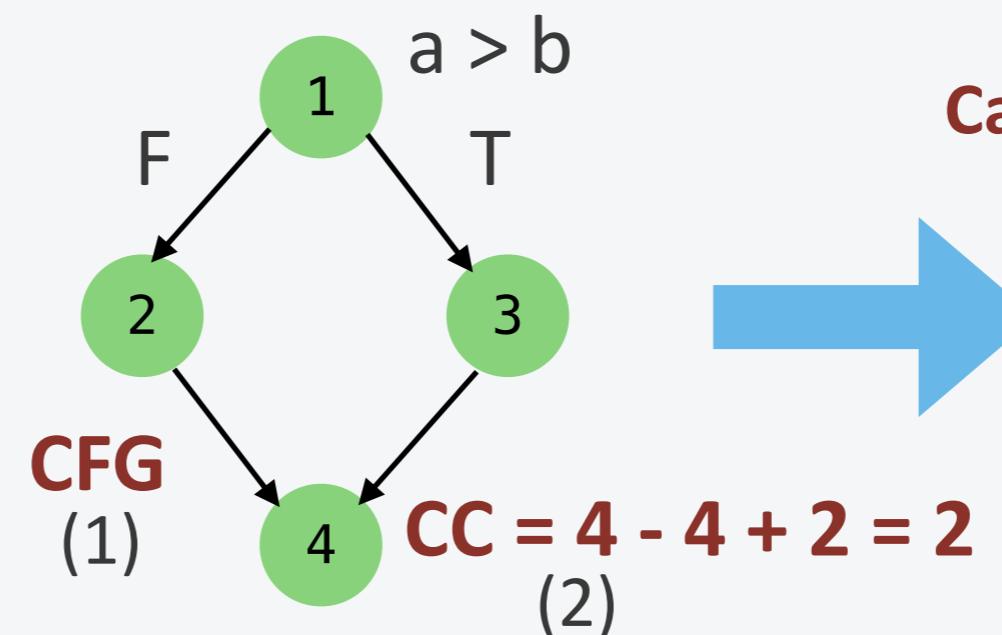
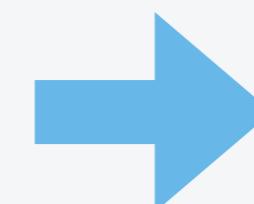
Ambas opciones son válidas

# EJEMPLO DE APLICACIÓN DEL MÉTODO S

Es muy importante ser sistemático pero sabiendo lo que haces en cada paso!!!

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;  
}
```



(3)

Caminos independientes

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

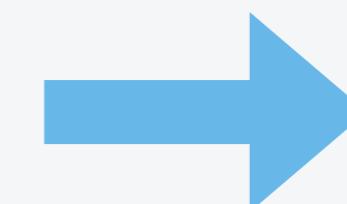
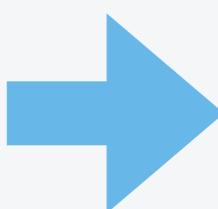


Tabla resultante del diseño de casos de prueba



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

(4) Valores de entrada

Resultado esperado

(5)

SIEMPRE son valores CONCRETOS!!!

# 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
```

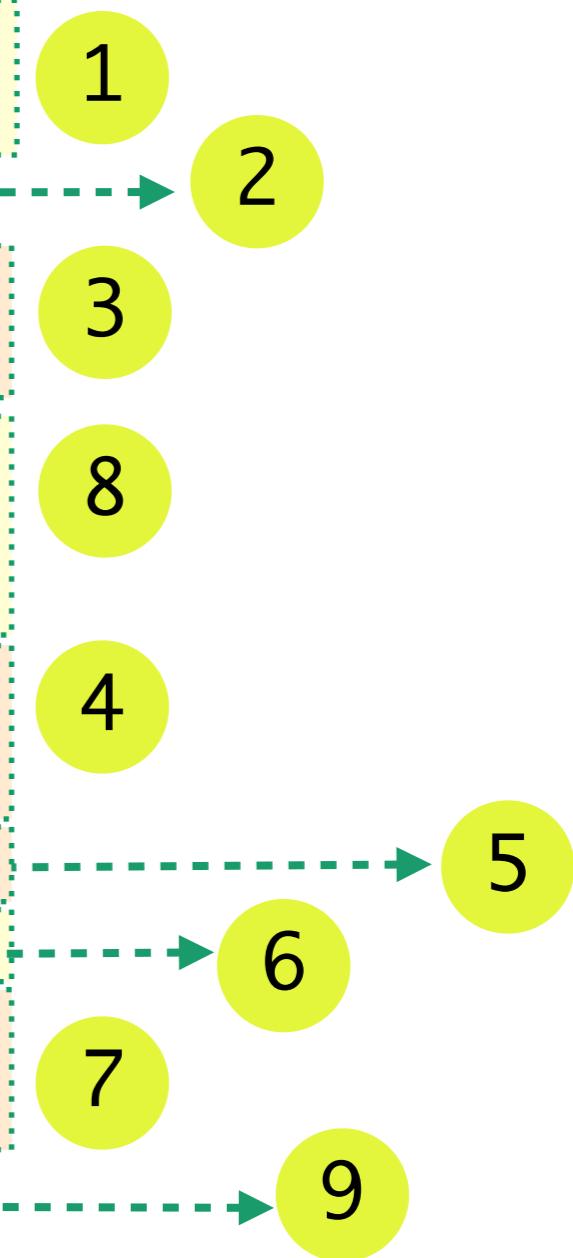
Especificación del método search():  
Dado un vector de enteros ordenados  
ascendentemente, y dado un entero  
(key) como entrada, el método  
search() busca la posición de key en  
el vector y devuelve el valor  
found=true si lo encuentra, así como  
su posición en el vector (dada por  
índex). Si 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

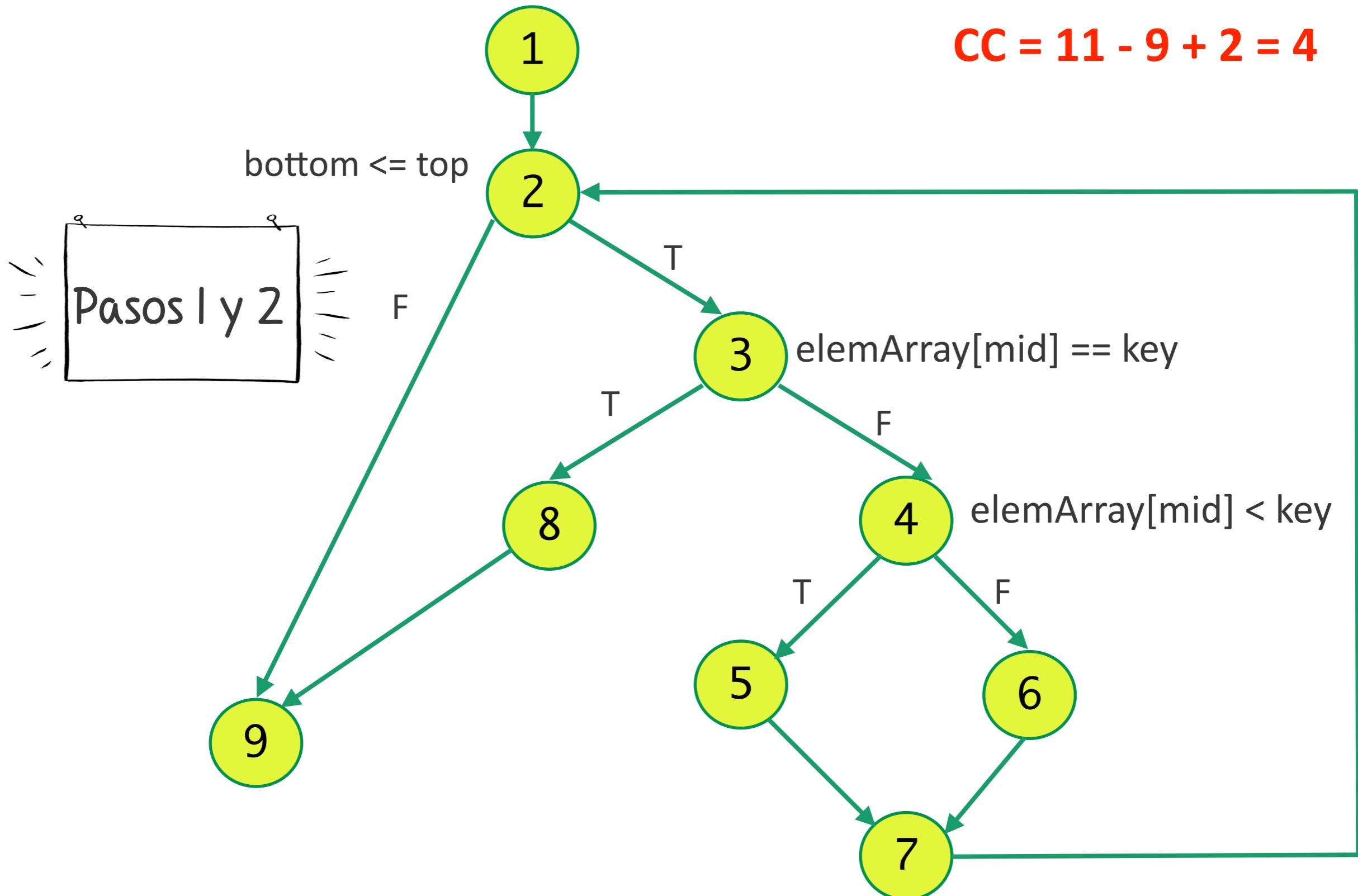
class BinSearch

```
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

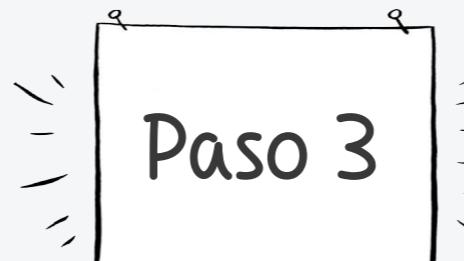
P

S

- Posible conjunto de caminos independientes

P

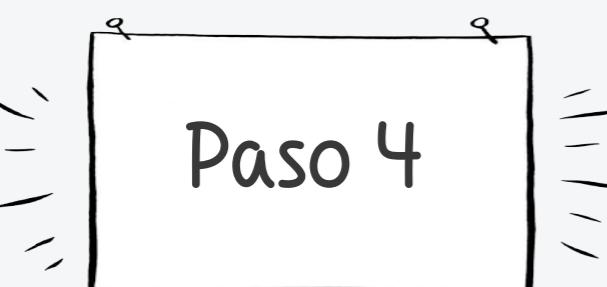
- C1: 1, 2, 3, 4, 6, 7, 2, 9
- C2: 1, 2, 3, 4, 5, 7, 2, 9
- C3: 1, 2, 3, 8, 9



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	
	key	elemArray	r.found	r.index
C1	2	[3,4]	false	?
C2	5	[1,4]	false	?
C3	2	[1,2,3]	true	1



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



# EJERCICIOS PROPUESTOS (I)

P

S

P

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

```
public int 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(int m,int n){  
    try {  
        int result = divide(m,n);  
        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)

S

P

Diseña los casos de prueba para el método validar\_PIN(), cuyo código es el siguiente:

P

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

la especificación la tenéis a continuación



# P EJERCICIOS PROPUESTOS (II) (CONTINUACIÓN)

Especificación del método validar\_PIN():

- P
- El método validar\_PIN() anterior valida un código numérico de cuatro cifras (objeto de la clase Pin). Dicho código se introducirá a través de un teclado (asumimos que en el teclado solamente hay teclas numéricas (0..9), y una tecla para cancelar, y que sólo puede introducir 4 caracteres). Si el usuario pulsa en algún momento la tecla de cancelar, entonces la validación se considerará "false". 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.

**Nota:** La introducción del código numérico se ha implementado en otra unidad (el método obtener\_pin()), que se encargará de “leer” 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 ‘\’).

Además usamos el método comprobar\_pin(), que 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.

Recuerda que el método del camino básico sólo lo aplicaremos a nivel de UNIDAD!!



Hemos definido una unidad como  
UNIDAD = MÉTODO JAVA

# Y AHORA VAMOS AL LABORATORIO...

P

P El proceso de diseño lo haremos "manualmente"

Diseñaremos casos de prueba utilizando el método del CAMINO BÁSICO

Hay que tener claros TODOS los pasos!!!

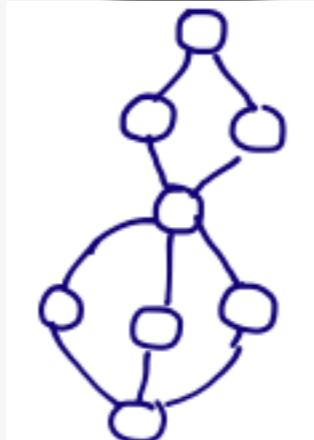
```
package ppss;

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;
    }
}
```

**unidad a probar**

CFG



CC

CC = ...

**tabla de casos de prueba**

**caminos independientes**

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

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

...

CN: 1-2-7-... -14

(N ≤ CC)

Camino	DATOS DE ENTRADA							
C1	d11	d12	...	d1q	r11	...	r1k	
...								
CN	dn1	dn2	...	dnq	r n1	...	r nk	

ESTA TABLA La utilizaremos en la siguiente práctica!!!...

# 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