

PPSS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2020-21

Sesión S01: Pruebas: proceso de diseño: caja blanca



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 "filas"

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"

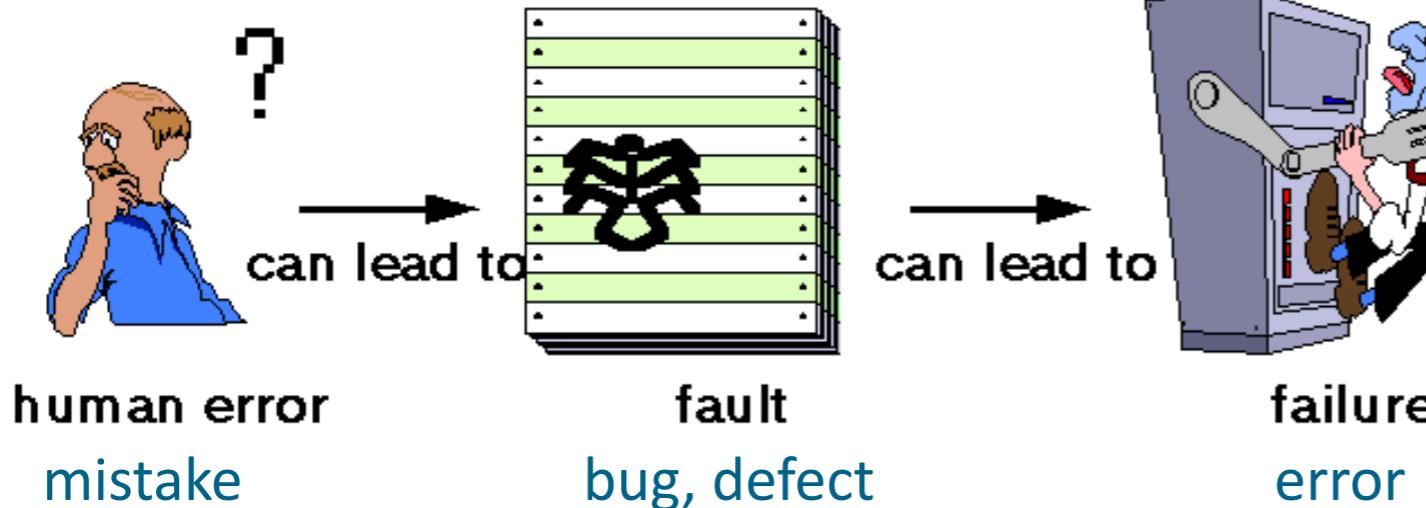
Glenford J. Myers (1979)

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)

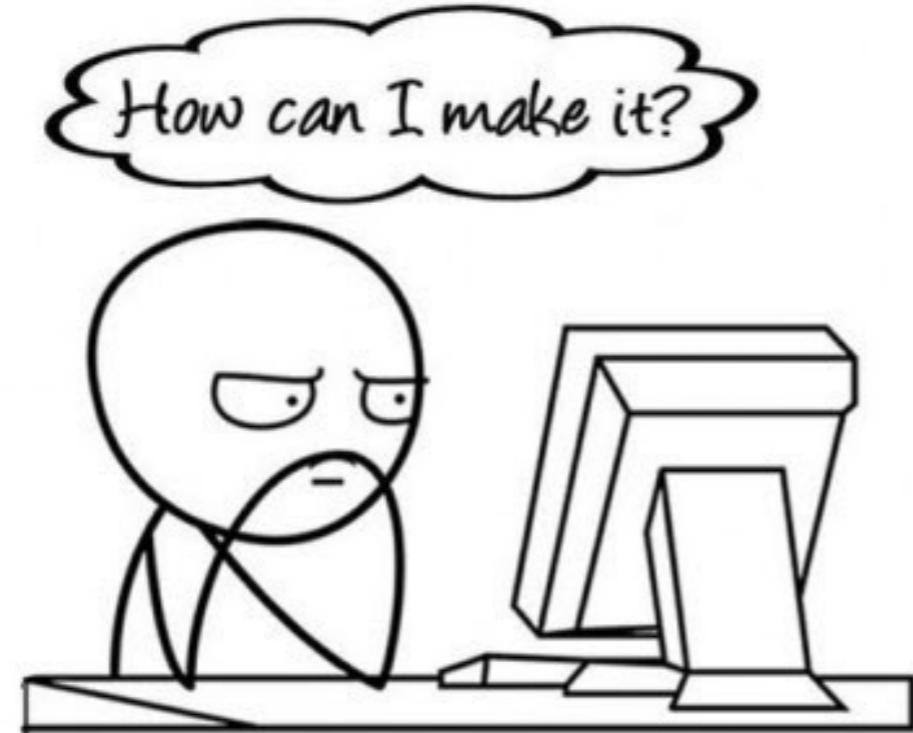


DEFINICIÓN DEL PROCESO DE PRUEBAS

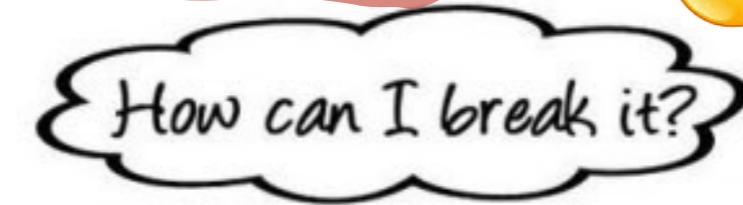
P

P

DEVELOPER



TESTER



A LOS USUARIOS NO LES GUSTAN LOS ERRORES!!



Uno de los objetivos de éxito del proyecto 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)



P ACTIVIDADES DEL PROCESO DE PRUEBAS

S 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

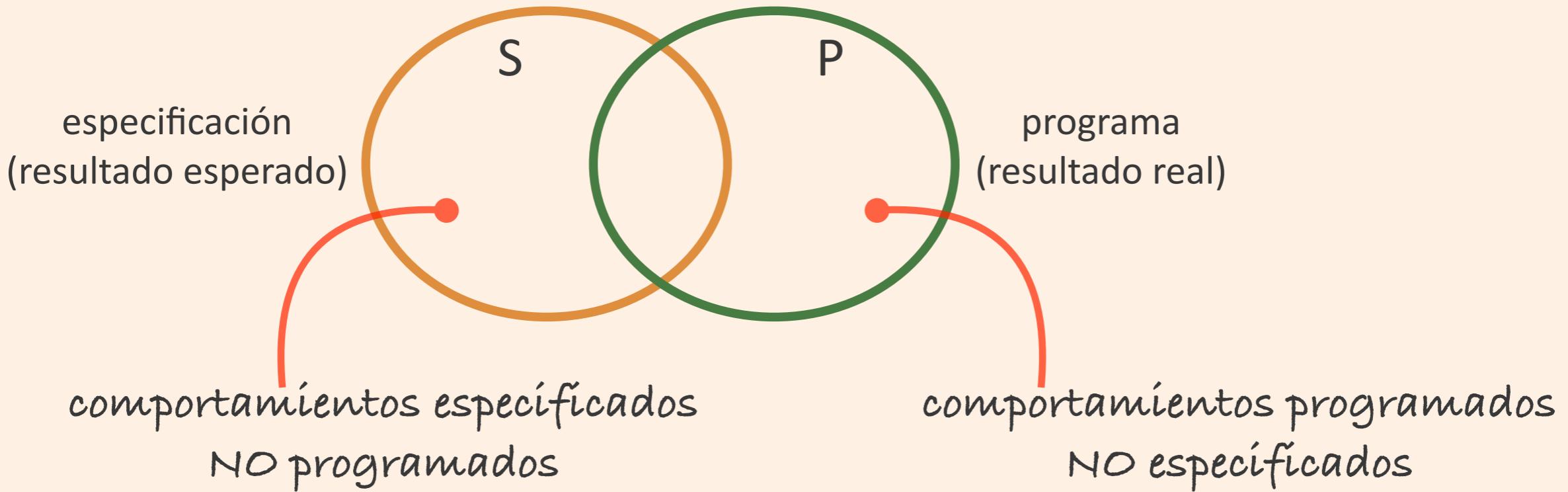
4 EVALUACIÓN del proceso de pruebas y emitir un informe

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!!!

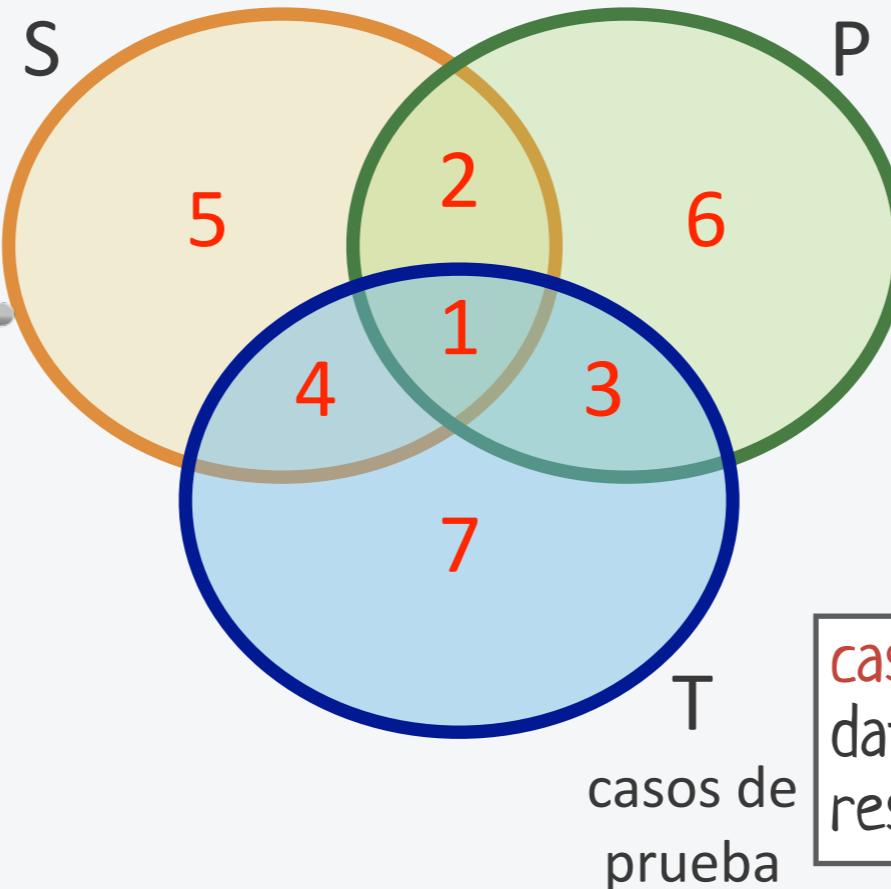
COMPORTAMIENTOS ESPECIFICADOS, PROGRAMADOS, Y PROBADOS

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

- O ¿Qué puede hacer un tester para conseguir que la región 1 sea lo más grande posible?

→ Identificar un conjunto de **casos de prueba** utilizando algún método de DISEÑO de pruebas (el objetivo es encontrar el máximo número posible de defectos con el mínimo número de casos de prueba)

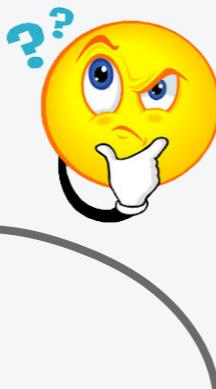
DISEÑO DE CASOS DE PRUEBA

S Selección de un subconjunto **mínimo** de entradas 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

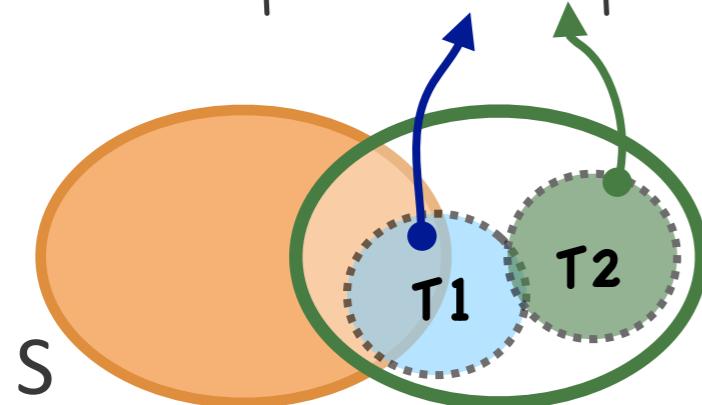
ID	d1	d2	...	Expected Output
C1	?	?	?	?
C2				
C3				
...				
CN?				



Cada FILA de la tabla es un CASO DE PRUEBA = datos entrada concretos + resultado esperado

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

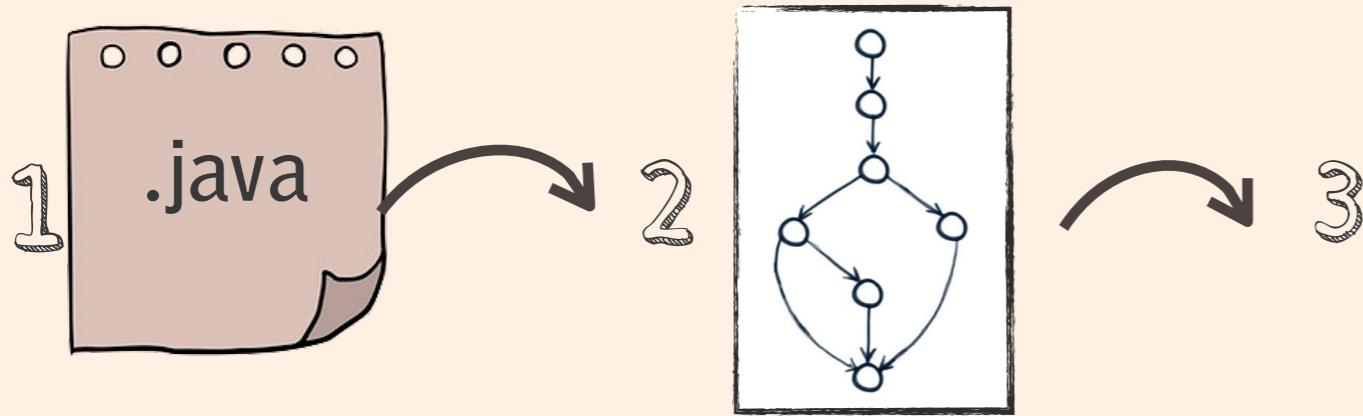


Tabla de casos de prueba

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

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

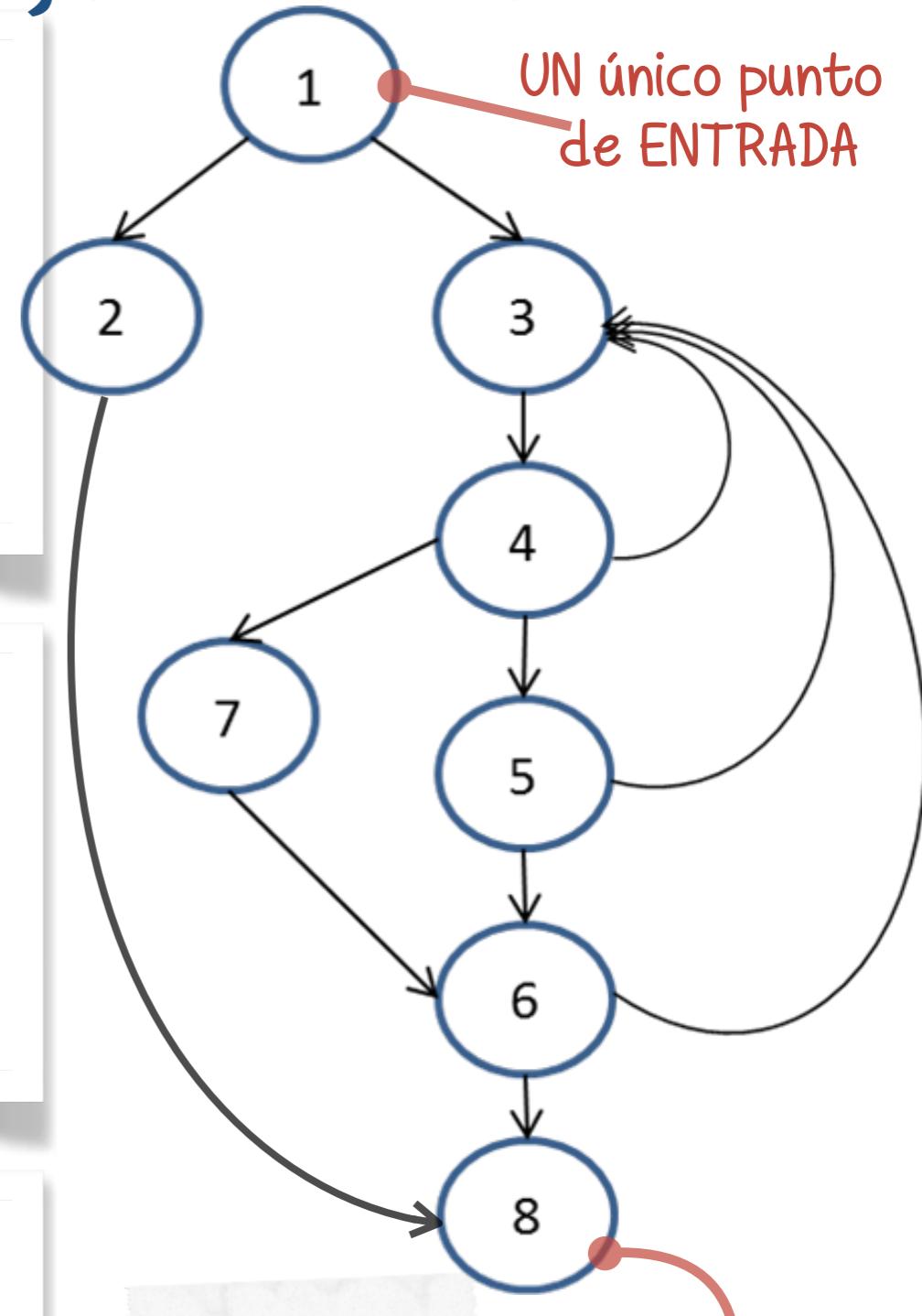
- Una llamada a una "función" cede el control a dicha función
- Dentro de la unidad a probar, la invocación de una "función" es una sentencia secuencial

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

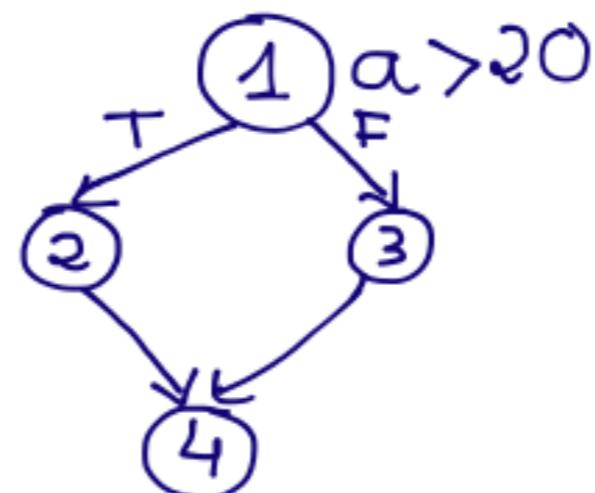
UN único punto de ENTRADA
UN único punto de SALIDA

GRAFO DE FLUJO DE CONTROL (CFG)

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

- Un CFG es una **representación gráfica** de una **unidad** de programa. Usaremos un **GRAFO DIRIGIDO**, en donde:
 - 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)

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

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

```
1. if ((a > 1) && (a < 200)) {  
2. ...  
3. }
```

```
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. }
```

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

```
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;
```

```
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.        }
```



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 las 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 ₁₁	d ₁₂	...	d _{1n}	r ₁
...					
C2	d ₂₁	d ₂₂	...	d _{2n}	r ₂

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

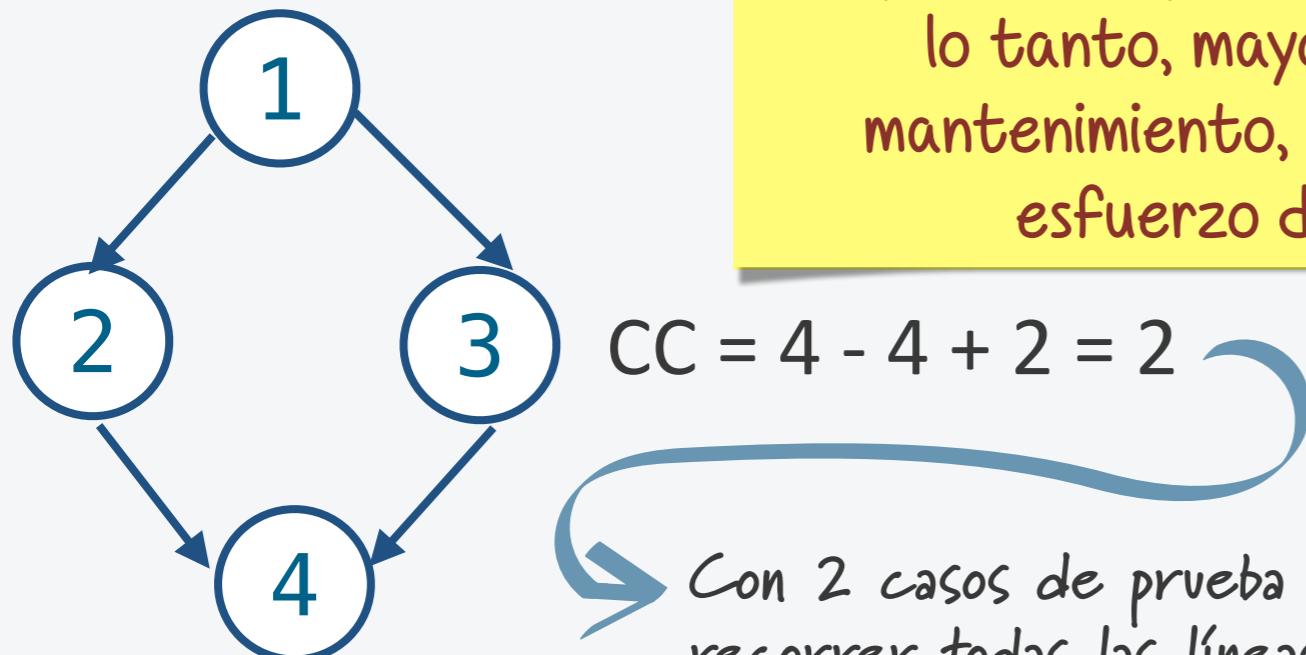
- Se calcula a partir del grafo de flujo:

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

- 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

P

P

S

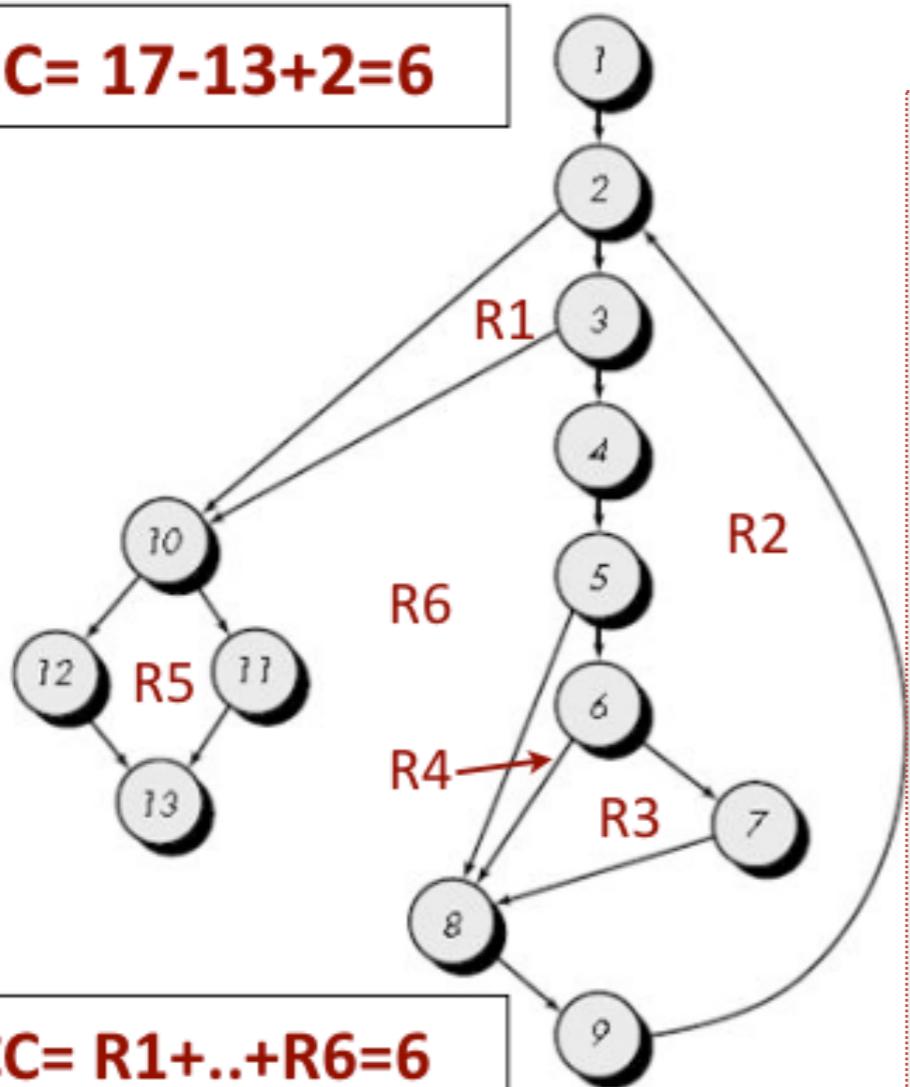
CC = número de arcos – número de nodos + 2

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;

```

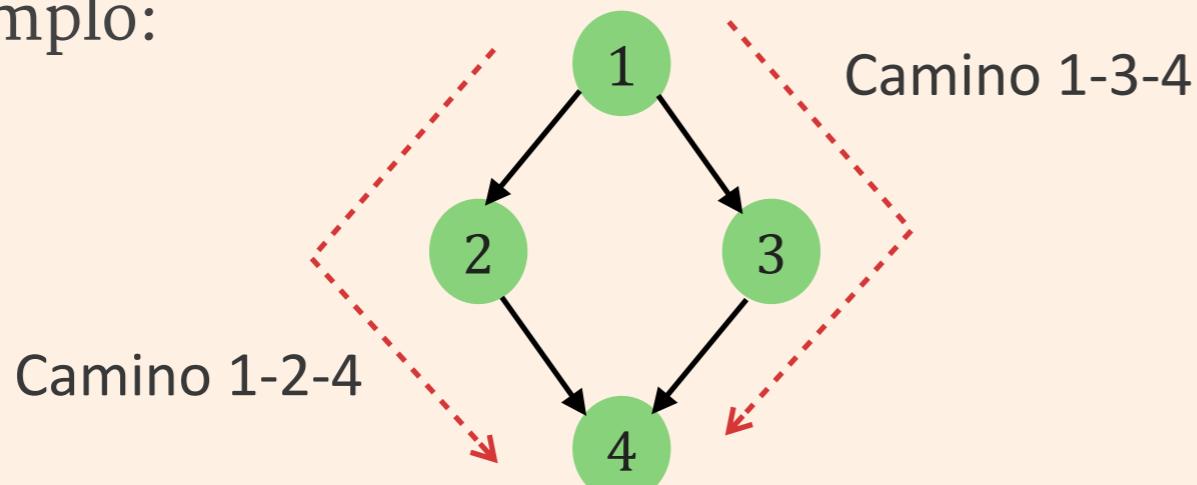
$$CC = 5 + 1 = 6$$

CAMINOS INDEPENDIENTES

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

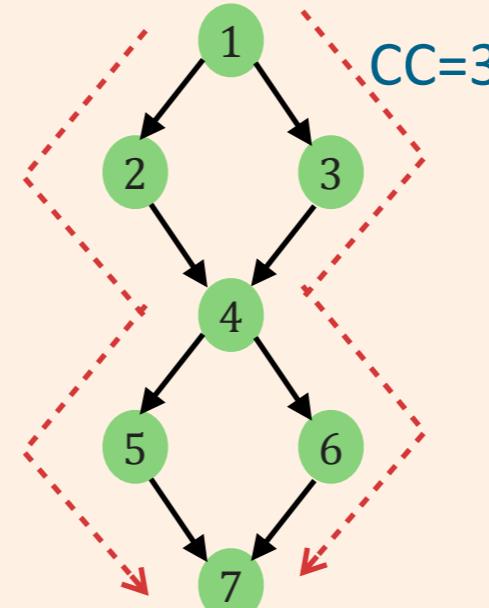
- Buscamos (como máximo) tantos caminos independientes como valor obtenido de CC
 - Cada camino independiente contiene un nodo, o bien una arista, que no aparece en los caminos independientes anteriores
 - Con ellos recorreremos TODOS los nodos y TODAS las aristas del grafo

- 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;  
}
```



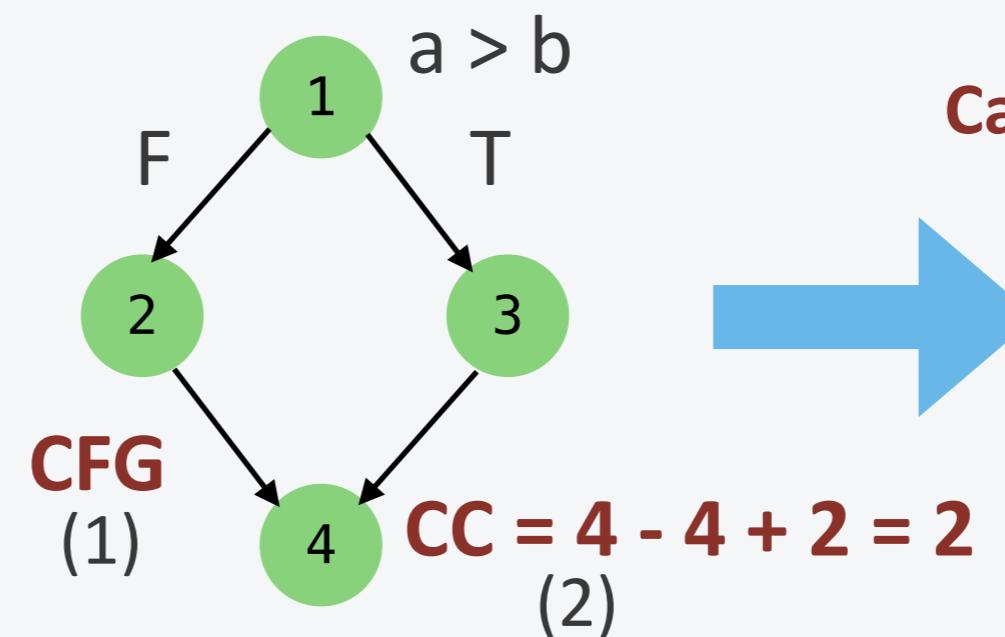
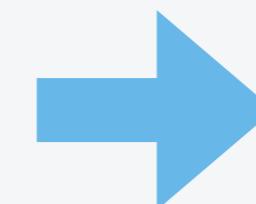
Ambas opciones son válidas

EJEMPLO DE APLICACIÓN DEL MÉTODO

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

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

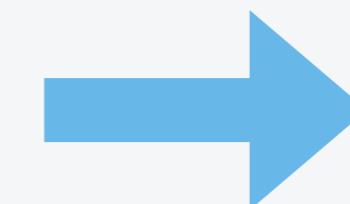
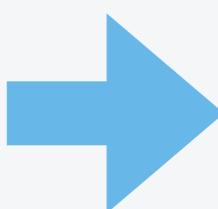


Tabla resultante del diseño de casos de prueba



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
```

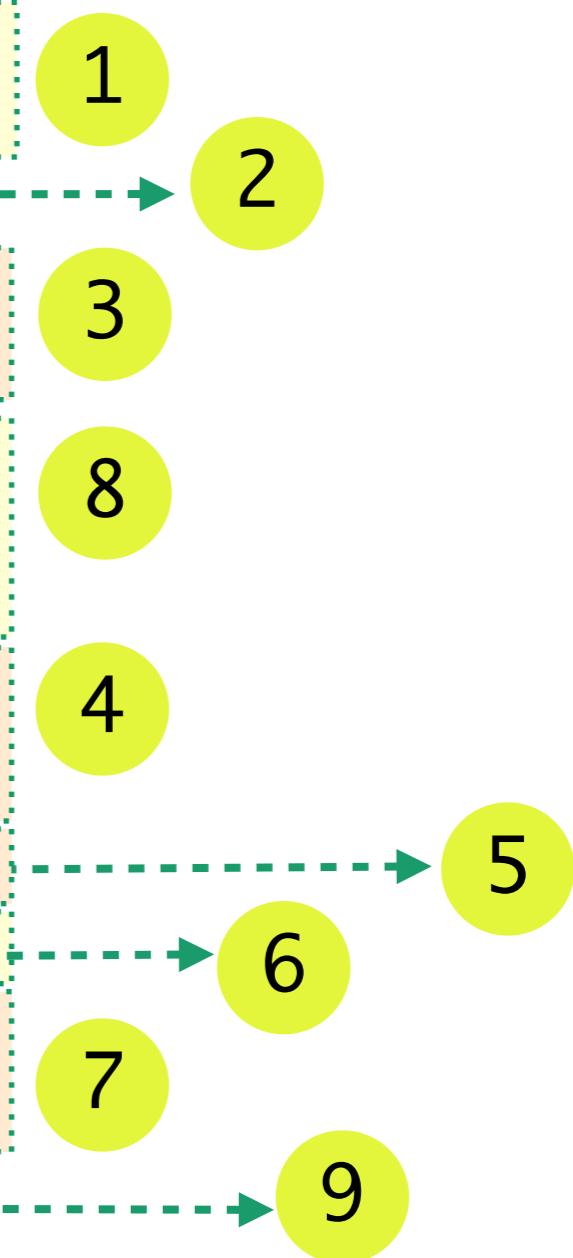
Especificación del método search():
Dado un vector de enteros ordenados
ascendentemente, y dado un entero
(key) como entrada, el método
search() busca la posición de key en
el vector y devuelve el valor
found=true si lo encuentra, así como
su posición en el vector (dada por
índex). Si el valor de key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



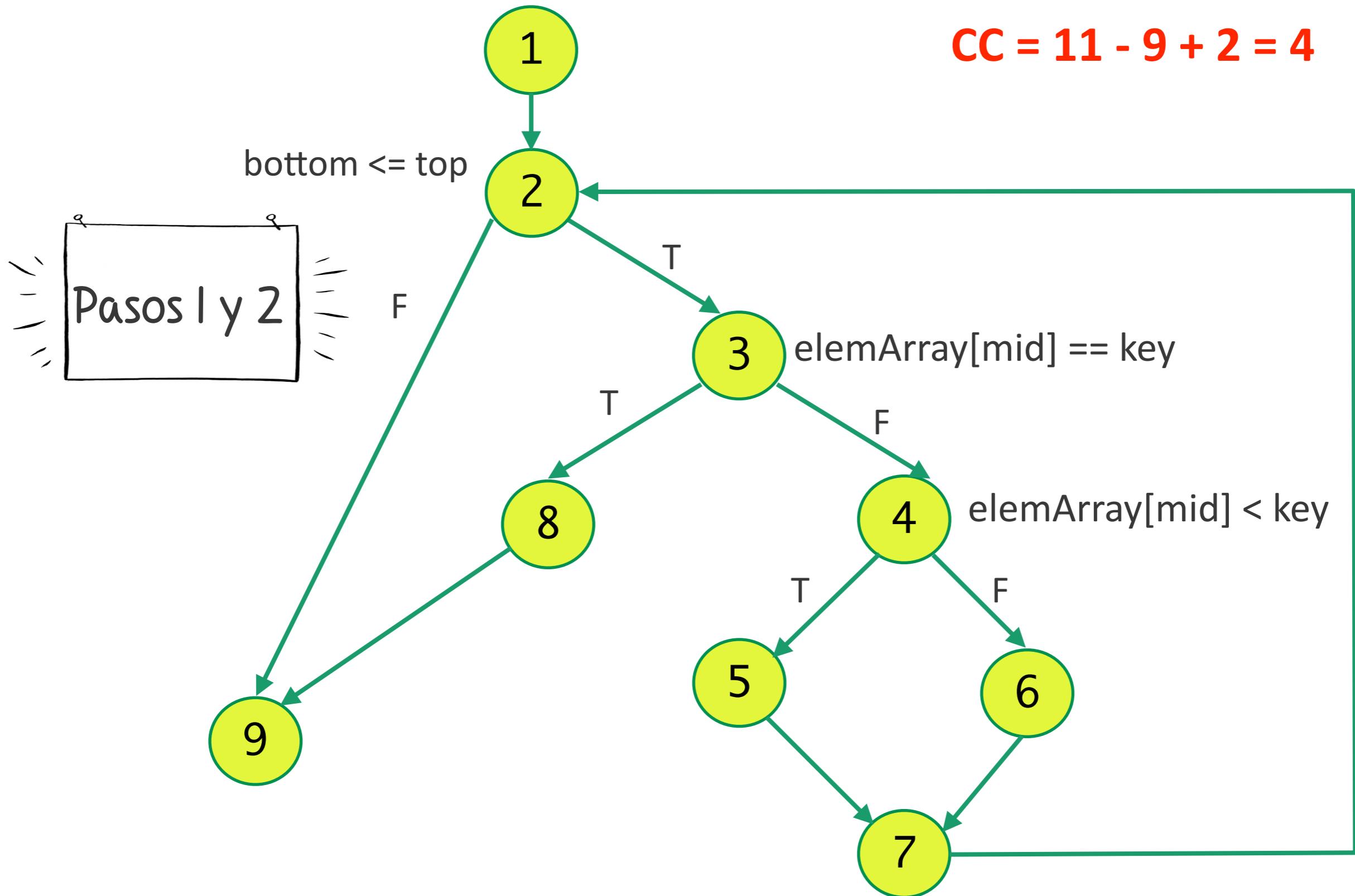
GRAFO ASOCIADO Y VALOR DE CC

3

P

P

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



CAMINOS INDEPENDIENTES

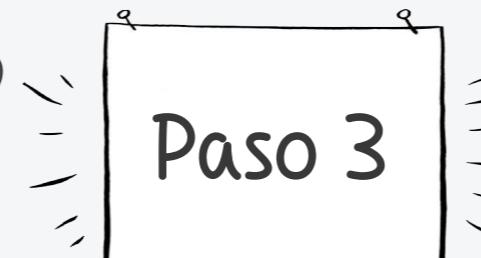
P

S

- Posible conjunto de caminos independientes

P

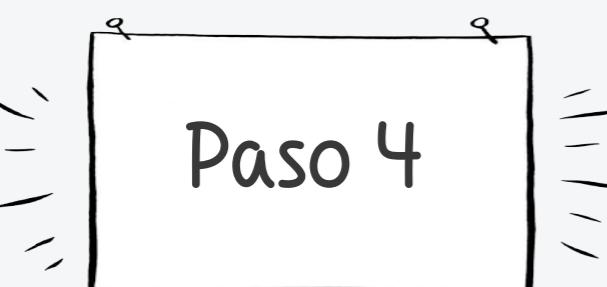
- C1: 1, 2, 3, 4, 6, 7, 2, 9
- C2: 1, 2, 3, 4, 5, 7, 2, 9
- C3: 1, 2, 3, 8, 9



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)



O 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.             codigo_resuesta = obtener_pin(Pin pinNumber);  
11.             if (!codigo_resuesta.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



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 obtendrá después de introducirlo a través de un teclado (asumimos que en el teclado solamente hay teclas numéricas (0..9), y una tecla para cancelar). 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!!!

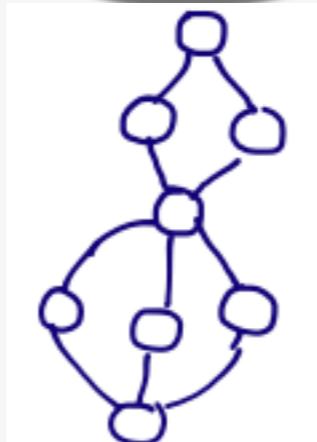
```
package ppss;

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	d ₁₁	d ₁₂	...	d _{1q}	r ₁₁	...	r _{1k}	
...								
CN	d _{n1}	d _{n2}	...	d _{nq}	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

PPSS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2020-21

Sesión S02: Drivers

Sarge, it's awful, Sarge ...

They eventually started finding all of us... They are fixing us everywhere... They... are even adding ... a new feature ...



I'll show'em ... I want two brave bugs to attack the new feature!!



Copyright 2005 Kazem A. Andekani

Maria Isabel Alfonso Galipienso
Universidad de Alicante eli@ua.es

Automatización de las pruebas

- Objetivo: ejecutar de forma automática todos los casos de prueba previamente diseñados
- Como resultado obtendremos un informe (que revelará la presencia de defectos en el código)

Pruebas unitarias

- El código a probar se denomina SUT.
- En nuestro caso SUT representa una unidad.
- Hemos definido una unidad como un método java

Pruebas de unidad dinámicas: drivers

- Un driver es un código que permite ejecutar un caso de prueba de forma automática.
- Necesitamos ejecutar código para detectar defectos (pruebas dinámicas)

Implementación de drivers: JUnit 5

- Librería para implementar y ejecutar las pruebas

Ejecución de los tests unitarios (drivers) con Maven

Vamos al laboratorio...

P AUTOMATIZACIÓN DE LAS PRUEBAS

(se trata de implementar código (drivers) para ejecutar los tests de forma automática)

P

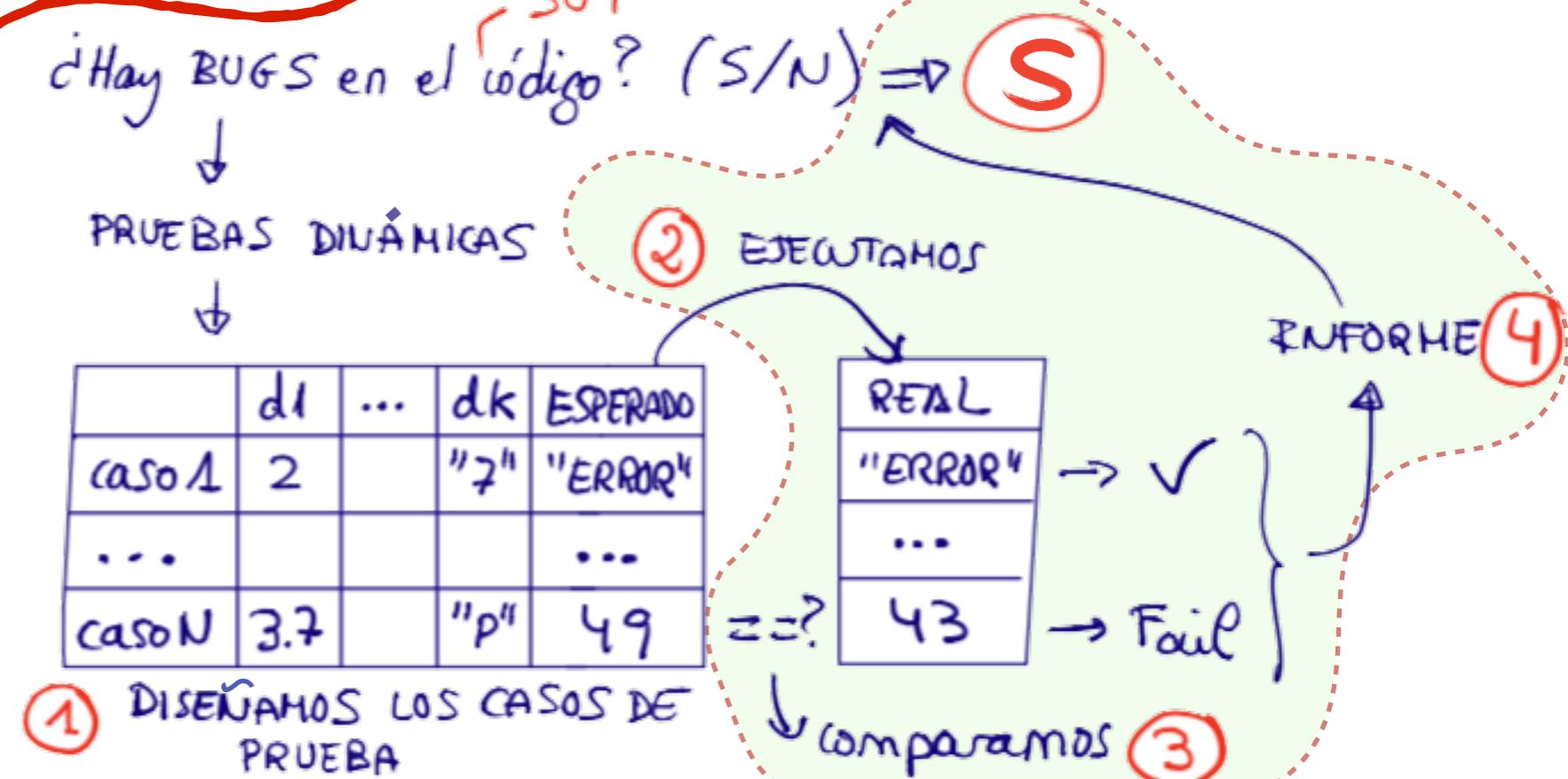
ACTIVIDADES
DEL PROCESO DE PRUEBAS

PLANIFICACIÓN Y CONTROL

DISEÑO

AUTOMATIZACIÓN

EVALUACIÓN



Para eso nuestro código de pruebas tendrá que:

- Una vez establecidas las precondiciones sobre los valores de entrada:
- Proporcionar los datos de entrada + resultado esperado (1) al código a probar (SUT)
- Obtener el resultado real (2)
- Comparar el resultado esperado con el real (3)
- Emitir un informe que contestará a nuestra pregunta inicial (4)



A dicho código de pruebas lo llamaremos **DRIVER**.

Implementaremos tantos drivers como casos de prueba.

Cada driver invocará a nuestra SUT y proporcionará un informe

SUT = System Under Test

PRUEBAS UNITARIAS

P

Sintácticamente, una unidad de programa es una "pieza" de código, que puede ser invocada desde fuera de la unidad y puede invocar a otras unidades de programa

Una unidad de programa implementa una función bien definida, y proporciona un nivel de abstracción para la implementación de funcionalidades de mayor nivel

Las pruebas unitarias son realizadas por los propios programadores. Éstos necesitan

VERIFICAR que el código funciona correctamente (tal y como se esperaba)

Hasta que el programador no implemente la unidad y esté completamente probada, el código fuente de una unidad no se pone a disposición del resto de miembros del grupo (normalmente a través de un sistema de control de versiones)

Pueden realizarse pruebas unitarias de forma estática y/o dinámica

→ para detectar errores
necesitamos ejecutar código!!!

S

S

Objetivo: encontrar DEFECTOS en el código de las UNIDADES probadas

Nuestra SUT será una Unidad

unidad 1

unidad 2

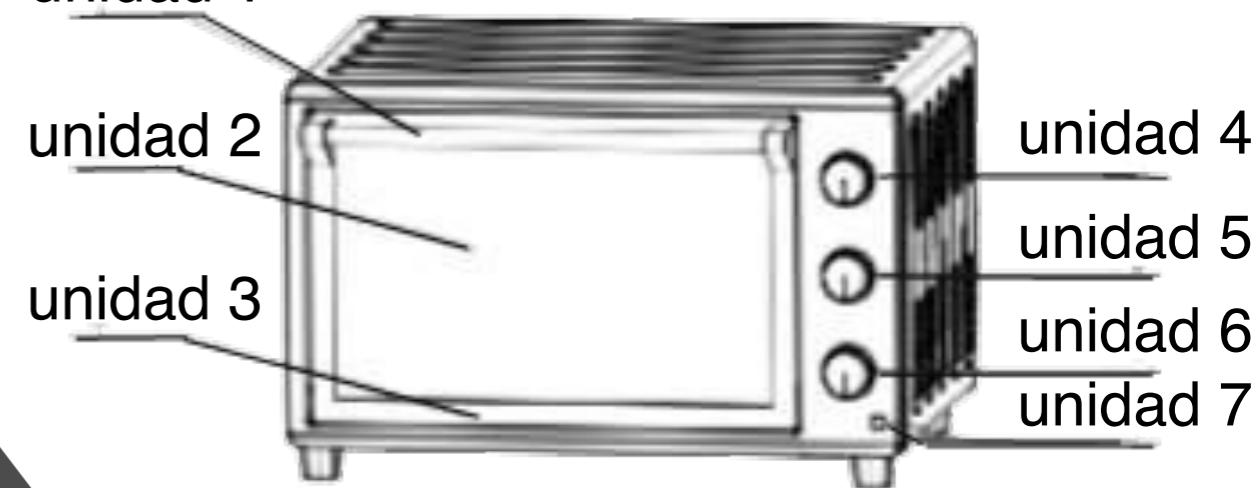
unidad 3

unidad 4

unidad 5

unidad 6

unidad 7



Queremos probar cada UNIDAD por SEPARADO!



La CUESTIÓN fundamental será cómo AISLAR el código de cada unidad a probar

PRUEBAS DE UNIDAD DINÁMICAS: DRIVERS

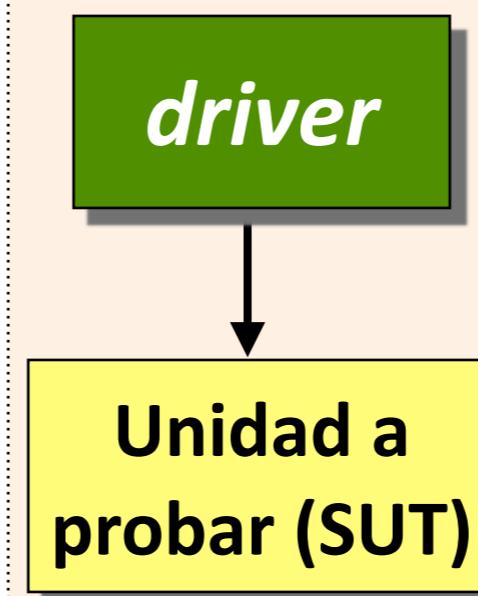
P Requieren ejecutar una unidad de forma **AISLADA** para poder detectar defectos en el código de dicha unidad

P El "tamaño" de las unidades dependerá de lo que consideremos como unidad.

En nuestro caso concreto, vamos a definir una UNIDAD como un MÉTODO JAVA

Cada unidad será invocada desde un driver durante las pruebas, con los datos de entrada diseñados previamente

```
1. //algoritmo de un driver
2. informe driver() {
3.   d= prepara_datos_entrada();
4.   esperado= resultado Esperado;
5.   //invocamos al SUT
6.   real= SUT(d);
7.   //comparamos el resultado
8.   //real con el esperado
9.   c= (esperado == real);
10.  informe= prepara_informe(c);
11.  return informe;
12. }
```



driver : conductor de la prueba.
Contiene el código necesario para EJECUTAR el caso de prueba sobre SUT

SUT : es el código que queremos probar. En este caso, representa a una unidad

En esta sesión vamos a ver cómo implementar drivers con JUnit para ejecutar pruebas unitarias dinámicas !!!!

JUNIT 5

P



<https://junit.org/junit5/docs/current/user-guide/>

P

JUnit es un API java que permite **implementar** los drivers y **ejecutar** los casos de prueba sobre componentes (SUT) de forma automática

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

Módulo que proporciona APIs relacionados con la "detección" y ejecución de los tests desde frameworks cliente (p.e. maven o IntelliJ)

Módulo que proporciona APIs para implementar y ejecutar los tests JUnit5

Módulo que proporciona APIs que para ejecutar tests JUnit3 o JUnit4

- JUnit se puede utilizar para implementar drivers de pruebas **unitarias** y también de **integración**
 - En una prueba **unitaria** estamos interesados en probar una única unidad (nuestra SUT será un método Java)
 - En una prueba de **integración** estamos probando varias unidades (la SUT en este caso representará a un subconjunto de unidades)



No es suficiente con conocer el API JUnit, hay que usarlo correctamente, siguiendo unas normas !!!

¿CÓMO IDENTIFICAMOS UN DRIVER CON JUNIT 5?

driver = test JUnit = método anotado con @Test

- Un **driver** automatiza la ejecución de un caso de prueba.
 - JUnit denomina test (uno por cada caso de prueba) a un MÉTODO sin parámetros, que devuelve void, y está anotado con @Test (@org.junit.jupiter.api.Test)

```
package mismo.paquete.claseAprobar;  
  
import org.junit.jupiter.api.Test;  
  
class TrianguloTest {  
  
    @Test  
    void C01testQueNoHaceNada() {  
    }  
}
```

Los tests deben agruparse lógicamente con el SUT correspondiente (deben pertenecer al mismo PAQUETE)

La CLASE de pruebas tendrá el mismo nombre que la clase que contiene el SUT precedida (o seguida) por "Test"

El DRIVER será un método sin parámetros que devuelve "void" y está anotado con @Test

Cada método anotado con @Test implementará un driver para un ÚNICO caso de prueba !!!!

No es necesario que la clase de pruebas ni los tests sean "public" !!

IMPLEMENTACIÓN DE UN DRIVER(I)

Cualquier driver sigue el mismo algoritmo:

```
//algoritmo de un driver
informe driver() {
    d= prepara_datos_entrada();
    esperado= resultado Esperado; >>>
    real= SUT(d); //llamamos a SUT
    c= (esperado == real); //comparamos el resultado real con el esperado
    informe= prepara_informe(c);
    return informe;
}
```

P IMPLEMENTACIÓN DE UN DRIVER(II)

El código de pruebas está físicamente separado del código fuente del SUT

P

SUT

```
package ppss;

public class Triangulo {
    public String tipo_triangulo
        (int a, int b, int c) {
    ...
}
```

/src/main/java → fuentes
/target/classes → ejecutables

ubicación física de SUT y
DRIVER si usamos Maven

DRIVER

```
package ppss;
import ...

class TrianguloTest {
    int a,b,c;
    String real, esperado;
    Triangulo tri= new Triangulo();

    @Test
    void testTipo_trianguloC1() {
        a = 1;
        b = 1;
        c = 1;
        resultadoEsperado = "Equilatero";
        resultadoReal = tri.tipo_triangulo(a,b,c);
        assertEquals(esperado, real);
    }
}
```

/src/test/java → fuentes
/target/test-classes → ejecutables
/target/surefire-reports → informes

SENTENCIAS ASSERT (org.junit.jupiter.api.Assertions)

- Junit proporciona sentencias (aserciones) para determinar el **resultado** de las pruebas y poder emitir el **informe** correspondiente
- Son **métodos estáticos**, cuyas principales características son:
 - Se utilizan para comparar el resultado esperado con el resultado real
 - El **orden de los parámetros** para los métodos assert... es:
 - resultado ESPERADO, resultado REAL [, mensaje opcional]

```
@Test
void standardAssertions() {
    /*todas las aserciones presentan
     estas tres variantes:      */
    assertEquals(2, 2);
    assertEquals(4, 4, "Mensaje opcional");
    assertEquals(8, 8, () -> "Mensaje creado"
                + "en tiempo de ejecución");
}
```

Podemos usar los métodos directamente:

```
import org.junit.jupiter.api.Assertions;
...
Assertions.assertEquals(...);
```

O referenciarlos mediante un import estático:

```
import static
org.junit.jupiter.api.Assertions.*;
...
assertEquals(...);
```

Todos los métodos "assert" generan una excepción de tipo **AssertionFailedError** si la aserción no se cumple!!!

Un test puede requerir varias aserciones

P AGRUPACIÓN DE ASERCIÓNES

Qué ocurre cuando un test contiene varias aserciones??

Dado que un test termina en cuanto se lanza la primera excepción (no capturada), en el caso de que nuestro test contenga varias aserciones, usaremos el método **assertAll**, para agruparlas. Este caso, se ejecutan todas, y si alguna falla se lanza la excepción **MultipleFailuresError**

assertAll

```
public static void assertAll(String heading,  
                           Executable... executables)  
throws MultipleFailuresError
```



Asserts that *all* supplied executables do not throw exceptions.

```
// Añadimos un elemento a una colección llena  
@Test  
public void testC3Add() {  
    int[] arrayEsperado = {1,2,3,4,5,6,7,8,9,10};  
    int numElemEsperado = 10;  
  
    int[] arrayEstadoInicial = Arrays.copyOf(arrayEsperado, arrayEsperado.length);  
    DataArray colección = new DataArray(arrayEstadoInicial, 10);  
    colección.add(11);  
    //Agrupamos las aserciones. SE EJECUTAN TODAS siempre  
    assertAll("GrupoTestC3",  
             ()-> assertArrayEquals(arrayEsperado, colección.getColección()),  
             ()-> assertEquals(numElemEsperado, colección.size())  
    ); //Se muestran todos los "fallos" producidos  
}
```



Esta opción nos proporciona más información!!

```
// Añadimos un elemento a una colección llena  
@Test  
public void testC3Add() {  
    ...  
    //No agrupamos las excepciones. Si falla la primera, la segunda NO se ejecuta  
    assertArrayEquals(arrayEsperado, instance.getColección());  
    assertEquals(numElemEsperado, instance.size());
```



Sólo se ejecuta si el assert anterior no falla

PRUEBAS DE EXCEPCIONES

assertThrows()

assertThrows

```
public static <T extends Throwable> T assertThrows(Class<T> expectedType,  
Executable executable, String message)
```

Assert that execution of the supplied executable throws an exception of the expectedType and return the exception.

If no exception is thrown, or if an exception of a different type is thrown, this method will fail.

If you do not want to perform additional checks on the exception instance, simply ignore the return value.

Fails with the supplied failure message.

- Si el resultado esperado es que nuestro SUT lance una excepción, usaremos la aserción assertThrows()

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.assertThrows;  
  
@Test  
void exceptionTesting() {  
    //si sut() lanza la excepción de tipo ExpectedException  
    //asignamos la excepción a la variable "exception"  
    ExpectedException exception = assertThrows(ExpectedException.class,  
        () -> sut(e1,e2));  
    //mostramos el mensaje asociado a la excepción  
    assertEquals("a message", exception.getMessage());  
}
```

Si al ejecutar sut()
NO se lanza la
excepción de tipo
ExpectedException,
la ejecución del
test fallará.

Si solamente queremos comprobar que se lanza la
excepción, esta sentencia NO es necesaria

PRUEBAS DE EXCEPCIONES

assertDoesNotThrow()

assertDoesNotThrow

```
@API(status=STABLE, since="5.2") public static void assertDoesNotThrow(Executable executable,  
String message)
```

Assert that execution of the supplied executable does *not* throw any kind of exception.

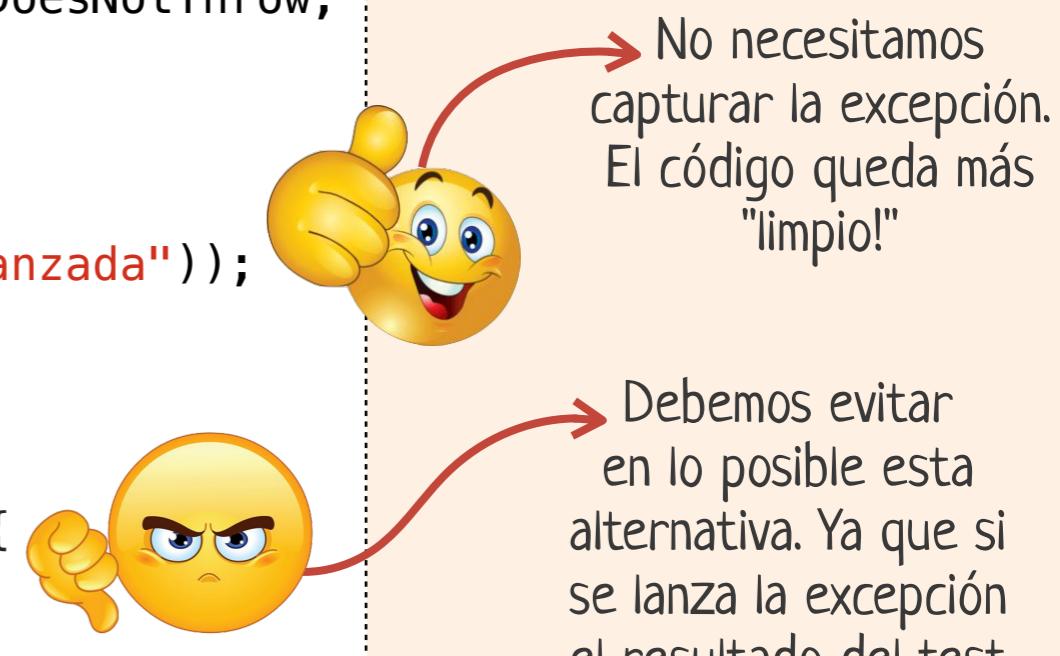
Usage Note

Although any exception thrown from a test method will cause the test to *fail*, there are certain use cases where it can be beneficial to explicitly assert that an exception is not thrown for a given code block within a test method.

Fails with the supplied failure message.

- Si el resultado esperado es que nuestro SUT no lance ninguna excepción, usaremos la aserción assertDoesNotThrow()

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.assertDoesNotThrow;  
  
@Test  
void NotExceptionTestingV1() {  
    //si sut() lanza una excepción el test fallará  
    assertDoesNotThrow(() -> sut(e1,e2), "Excepción lanzada"));  
}  
  
@Test  
void NotExceptionTestingV2() throws ExpectedException{  
    sut(e1,e2);  
}
```



```
void sut(Type1 param1, Type2 param2) throws ExpectedException
```

PANOTACIONES @BeforeEach, @AfterEach, @BeforeAll, @AfterAll

- En el caso de que **TODOS** los tests requieran las **MISMAS** acciones para preparar los datos de entrada (antes de ejecutar el elemento a probar), implementaremos dichas acciones comunes en un método anotado con **@BeforeEach**.
 - De esta forma reduciremos la duplicación de código y nos aseguraremos de que todos los tests parten del mismo estado inicial
- De igual forma, usaremos la anotación **@AfterEach** en un método que contenga todas las acciones comunes a realizar **después de la ejecución de CADA test** (por ejemplo, podríamos necesitar asegurarnos de que una conexión por socket esté cerrada después de ejecutar cada test)

@BeforeEach y **@AfterEach** se usan con métodos void SIN parámetros!!!

- Si es necesario realizar acciones previas a la ejecución de **TODOS** los tests una **ÚNICA VEZ**, (o después de ejecutar todos los tests), implementaremos dichas acciones en un método anotado con **@BeforeAll** o **@AfterAll**, respectivamente.

@BeforeAll y **@AfterAll** se usan con métodos estáticos void SIN parámetros!!!

- Estas anotaciones permiten inicializar y restaurar el estado del entorno en el que se ejecuta cada test o cada conjunto de tests, de forma que si algún test (o conjunto de tests) "alteran" dicho estado, el siguiente test/conjunto de tests pueda ejecutarse normalmente con independencia del resultado de la ejecución de tests anteriores.

NO debemos implementar tests cuya ejecución dependa del resultado de ejecutar ningún otro test!!!



EJEMPLO @BeforeEach, @AfterEach, @BeforeClass, @AfterClass

```
import org.junit.jupiter.api.AfterAll;  
import org.junit.jupiter.api.AfterEach;  
import org.junit.jupiter.api.BeforeAll;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
class OutputTest {  
    static File output;  
  
    @BeforeEach void createOutputFile() {  
        output = new File(...);  
    }  
    @AfterEach void deleteOutputFile() {  
        output.delete();  
    }  
    @BeforeAll static void initialState() {  
        //initial code  
    }  
    @AfterAll static void finalState() {  
        //final code  
    }  
    @Test void test1WithFile() {  
        // code for test case objective  
    }  
    @Test void test2WithFile() {  
        // code for test case objective  
    }  
}
```

ORDEN de ejecución asumiendo que
test1Withfile() se ejecuta ANTES de
test2WithFile()

1. initialState()
2. createOutputFile()
3. test1WithFile()
4. deleteOutputFile()
5. createOutputFile()
6. test2WithFile()
7. deleteOutputFile()
8. finalState()

No debemos asumir
ningún orden de
ejecución de nuestros
tests!!!



P

ETIQUETADO DE LOS TESTS: @Tag

S Permiten SELECCIONAR la ejecución de un subconjunto de tests

- Tanto las clases como los tests pueden anotarse con @Tag. Esta anotación permite "etiquetar" nuestros tests para FILTRAR su "descubrimiento" y "ejecución"
 - Si anotamos la clase, automáticamente estaremos etiquetando todos sus tests
 - Podemos usar varias "etiquetas" para la clase y/o los tests

```
import org.junit.jupiter.api.Tag;  
import org.junit.jupiter.api.Test;  
  
{@Tag("fast")  
{@Tag("model")  
class TaggingDemo {  
  
    @Test  
    @Tag("taxes")  
    void testingTaxCalculation() {  
        ...  
    }  
}}
```

Este test está etiquetado como "fast", "model" y "taxes"

Las anotaciones @Tag nos permitirán DISCRIMINAR la ejecución de los tests según sus etiquetas.

Ejemplos de etiquetas:

- "Firefox", "Explorer", "Safari", ...
- "Windows", "OSX", "Ubuntu", ...
- "Unitarios", "Integracion", "Sistema", ...

TESTS PARAMETRIZADOS @ParameterizedTest, @ValueSource

- Si el código de varios tests es idéntico a excepción de los valores concretos del caso de prueba que cada uno implementa, podemos sustituirlos por un **test parametrizado**.
- Se trata de implementar un único test, que anotaremos con **@ParameterizedTest**, y que tendrá como parámetros los valores concretos en los que se diferencian los tests a los que sustituye.
- Si el test parametrizado solamente necesita un parámetro, de tipo primitivo o String, usaremos la anotación **@ValueSource** para indicar los valores para ese parámetro

EJEMPLO de Test parametrizado usando @ValueSource

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

@ParameterizedTest
@ValueSource(strings = {"racecar", "radar", "able was I ere I saw elba"})
void palindromes(String candidate) {
    assertTrue(c.isPalindrome(candidate));
}
```

En este caso, los valores de la colección de parámetros son de tipo String

- El método **palindromes** es un tests parametrizado, con un parámetro de tipo String.
- El test se ejecuta 3 veces (con cada uno de los 3 parámetros indicados en **@ValueSource**).
- Otras alternativas posibles son **@ValueSource(doubles = {...})**, **@ValueSource(ints = {...})**, o **@ValueSource(longs = {...})**, dependiendo del tipo de dato del parámetro del test anotado con **@ParameterizedTest**

TESTS PARAMETRIZADOS @ParameterizedTest, @MethodSource

EJEMPLO de Test parametrizado usando @MethodSource

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

@ParameterizedTest(name = "User {1}, when Alert level is {2} should
                    have access to transporters of {0}")
@MethodSource("casosDePrueba")
void testParametrizado(boolean expected, Person user, Alert alertStatus) {
    transp.setAlertStatus(alertStatus);
    assertEquals(expected, transp.canAccessTransporter(user), invocación SUT
        () -> generateFailureMessage("transporter",
            expected, user, alertStatus));
}

//los valores devueltos por este método son los argumentos
//del método anotado con @ParameterizedTest
private static Stream<Arguments> casosDePrueba() {
    return Stream.of(
        Arguments.of(true, picard, Alert.NONE),
        Arguments.of(true, barclay, Alert.NONE),
        Arguments.of(false, lwaxana, Alert.NONE),
        Arguments.of(false, lwaxana, Alert.YELLOW),
        Arguments.of(false, q, Alert.YELLOW),
        Arguments.of(true, picard, Alert.RED),
        Arguments.of(false, q, Alert.RED)
    );
}

//método que construye y devuelve un mensaje de error en caso
//de que el resultado esperado no coincida con el real
private String generateFailureMessage(String system, boolean expected,
                                      Person user, Alert alertStatus) {
    String message = user.getFirstName() + " should";
    if (!expected) {
        message += " not";
    }
    message += " be able to access the " + system +
               " when alert status is " + alertStatus;
    return message;
}
```

El método `casosDePrueba()` devuelve un `Stream` con los `Argumentos` que se pasarán como parámetros al test parametrizado. Cada objeto de tipo `Arguments` es un caso de prueba

Si el método anotado con `@ParameterizedTest` requiere más de un parámetro, usaremos la anotación `@MethodSource` indicando un nombre de método.

El método `casosDePrueba` devuelve una colección de "tuplas" de valores (cada tupla representa una fila de la tabla de casos de prueba). El número de elementos de la tupla se corresponderá con el número de parámetros del test parametrizado.

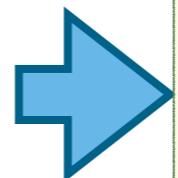
El test parametrizado se invocará tantas veces como elementos de tipo `Arguments` tengamos. En el ejemplo serán 7 veces.

JUNIT 5 Y MAVEN

Para poder implementar y compilar de los tests

- Para poder **implementar** los tests con JUnit5 (y compilarlos) necesitamos incluir la librería "junit-jupiter-engine". De esta forma tendremos acceso las clases del paquete org.junit.jupiter.api, importándolas desde nuestro código de pruebas.

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.7.0</version>
  <scope>test</scope>
</dependency>
```



```
//código fuente de los tests en /src/
test/java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Tag;
...
```

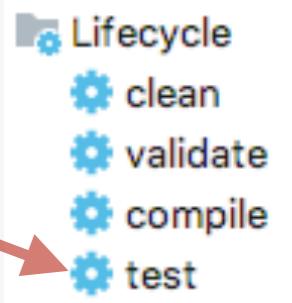
- El valor "test" de la etiqueta <scope> indica que la librería sólo se utiliza para compilar y ejecutar los tests. Por lo tanto NO podremos importar ninguna de sus clases desde /src/main/java
- Si usamos tests **parametrizados**, necesitaremos incluir también la libreria "junit-params" para poder usar las anotaciones correspondientes en nuestro código de pruebas.

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>5.7.0</version>
  <scope>test</scope>
</dependency>
```

```
//código fuente de los tests en /src/test/java
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;
import org.junit.jupiter.params.provider.ValueSource;
...
```



Modifica el pom SIEMPRE
manualmente!! No permitas que el
IDE añada/modifique el pom por tí!!



JUNIT 5 Y MAVEN

Para poder EJECUTAR los tests

<https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html>

- Ejecutaremos los tests mediante la goal **surefire:test**. El plugin surefire será, por tanto, el encargado usar las librerías JUnit5 que correspondan para ejecutar las pruebas
- Necesitamos incluir el plugin surefire en nuestro pom:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
</plugin>
```

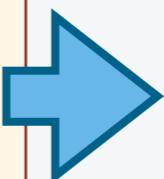


mvn test

la goal **surefire:test** está asociada por defecto a la fase **test** de maven

- La goal **surefire:test** ejecuta todos los métodos anotados con `@Test` (o `@ParameterizedTest`) dentro de las clases cuyo nombre se corresponda con alguno de estos patrones: `**/Test*.java`, `**/*Test.java`, `**/*Tests.java`, `**/*TestCase.java`
- Para "filtrar" la ejecución de los tests en función de sus anotaciones `@Tag`, tendremos que configurar el plugin usando las propiedades "**groups**" y "**excludedGroups**" del plugin

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
  <configuration>
    <groups>etiqueta1,etiqueta2</groups>
    <excludedGroups>excluidos</excludedGroups>
  </configuration>
</plugin>
```



mvn test

En este ejemplo se ejecutarán los tests etiquetados como "etiqueta1" y también los etiquetados con "etiqueta2". También podemos excluir una (o varias etiquetas) del filtro

Sí algún test falla, el proceso de construcción se detiene y se obtiene un BUILD FAILURE!!

CONFIGURACIÓN DEL PLUGIN SUREFIRE

Se puede configurar cualquier plugin

<https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html>

- Podemos alterar el valor de ciertas propiedades de los plugins usando la etiqueta **<configuration>**. Cada plugin tiene su conjunto de propiedades. La única forma de saber cómo usar correctamente un plugin es acceder a su documentación.
- Como ya hemos visto, podemos filtrar la ejecución de los tests a través de sus etiquetas, usando las propiedades **groups** y/o **excludedGroups** del plugin surefire, en el pom de nuestro proyecto Maven
- De forma alternativa, podemos configurar cualquier plugin desde línea de comandos:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
</plugin>
```



mvn test

Usamos la configuración por defecto en el pom, y la cambiamos desde línea de comandos

-Dgroups=etiqueta1,etiqueta2 -DexcludedGroups=excluidos

- Debes tener en cuenta de que si configuramos el plugin en el pom y también desde línea de comandos, la configuración del pom PREVALECE sobre la de línea de comandos. Si por ejemplo, quisiéramos ejecutar siempre un cierto subconjunto de tests y ocasionalmente otro subconjunto, podríamos hacer esto:

```
<properties>
  <filtrar.por>importantes, fase1</filtrar.por>
</properties>
...
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
  <configuration>
    <groups>${filtrar.por}</groups>
  </configuration>
</plugin>
```



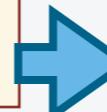
mvn test

Se ejecutan los tests etiquetados como "importantes" más todos los tests etiquetados como "fase1"



mvn test -Dfiltrar.por=rapidos

Se ejecutan los tests etiquetados como "rapidos"



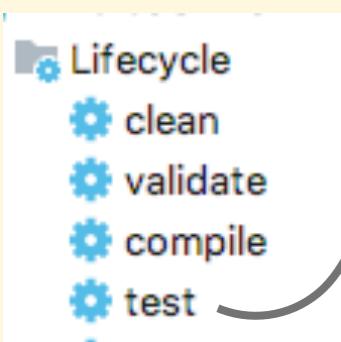
mvn test -Dfiltrar.por=""

Se ejecutan todos los tests

INFORMES JUNIT

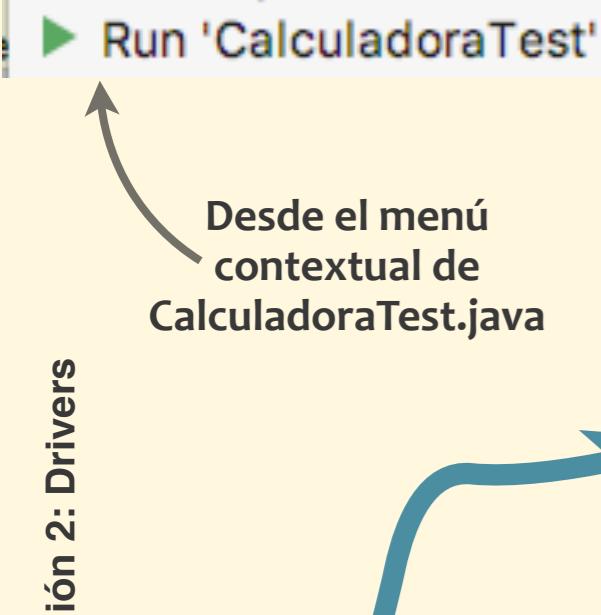
Ejecutaremos los tests desde la ventana Maven!!!

- P O Cuando ejecutamos cada test, podemos obtener 3 resultados posibles:
 - **Pass**: cuando el resultado esperado coincide con el real
 - **Failure**: el método Assert lanza una excepción de tipo **AssertionFailedError**
 - **Error**: se genera cualquier otra excepción durante la ejecución del test
- P O El **informe** de la ejecución de todos los tests se guarda en **target/surefire-reports**, en formatos **txt** y **xml** (un informe por clase)
 - Dependiendo de la herramienta que "lea" dichos informes, éstos se mostrarán de forma diferente



```
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   CalculadoraTest.suma1:22 expected: <1> but was: <0>
[ERROR]   CalculadoraTest.suma2:28 expected: <7> but was: <4>
[ERROR]   CalculadoraTest.suma4:40 expected: <13> but was: <5>
[ERROR] Errors:
[ERROR]   CalculadoraTest.suma3:34 » Arithmetic / by zero
[INFO]
[ERROR] Tests run: 5, Failures: 3, Errors: 1, Skipped: 0
```

Informe de pruebas
Maven



Desde el menú
contextual de
CalculadoraTest.java

Test Results		49 ms
!	CalculadoraTest	49 ms
×	sumá1()	38 ms
×	sumá2()	2 ms
!	sumá3()	2 ms
×	sumá4()	1 ms
✓	sumá5()	6 ms

Informe de pruebas IntelliJ

Sólo muestra los
informes de la última
ejecución de la fase
test de Maven

!	ppss.CalculadoraTest
×	sumá1
×	sumá2
!	sumá3
×	sumá4
✓	sumá5

Informe obtenido por
el plugin "Maven Test
Results" de IntelliJ, a
partir del informe de
Maven

MAVEN Y PRUEBAS UNITARIAS

P

P

Fase compile:

Se compilan los fuentes del proyecto (/src/main/java)

GOAL por defecto: **compiler:compile**

artefactos generados en /target/classes

Si hay errores de compilación se detiene la construcción.

Por defecto



Fase test-compile:

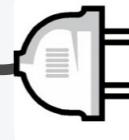
Se compilan los tests unitarios (/src/test/java)

GOAL por defecto: **compiler:testCompile**

artefactos generados en /target/test-classes

Si hay errores de compilación se detiene la construcción.

Por defecto



Fase test:

Se ejecutan los tests unitarios (/target/test-classes)

Se ejecutan los métodos anotados con @Test de las clases

**/Test*.java, **/*Test.java, o **/*TestCase.java

GOAL por defecto: **surefire:test**

artefactos generados en /target/surefire-reports.

Por defecto

Se detiene el proceso de construcción si algún tests falla.

Default lifecycle

validate

initialize

generate-sources

process-sources

generate-resources

process-resources

compile

process-classes

generate-test-sources

process-test-sources

generate-test-resource

process-test-resources

test-compile

process-test-classes

test

prepare-package

package

pre-integration-test

integration-test

post-integration-test

verify

install

deploy

Y AHORA VAMOS AL LABORATORIO...

P

Vamos a automatizar el diseño de casos de prueba que hemos obtenido en prácticas anteriores

P

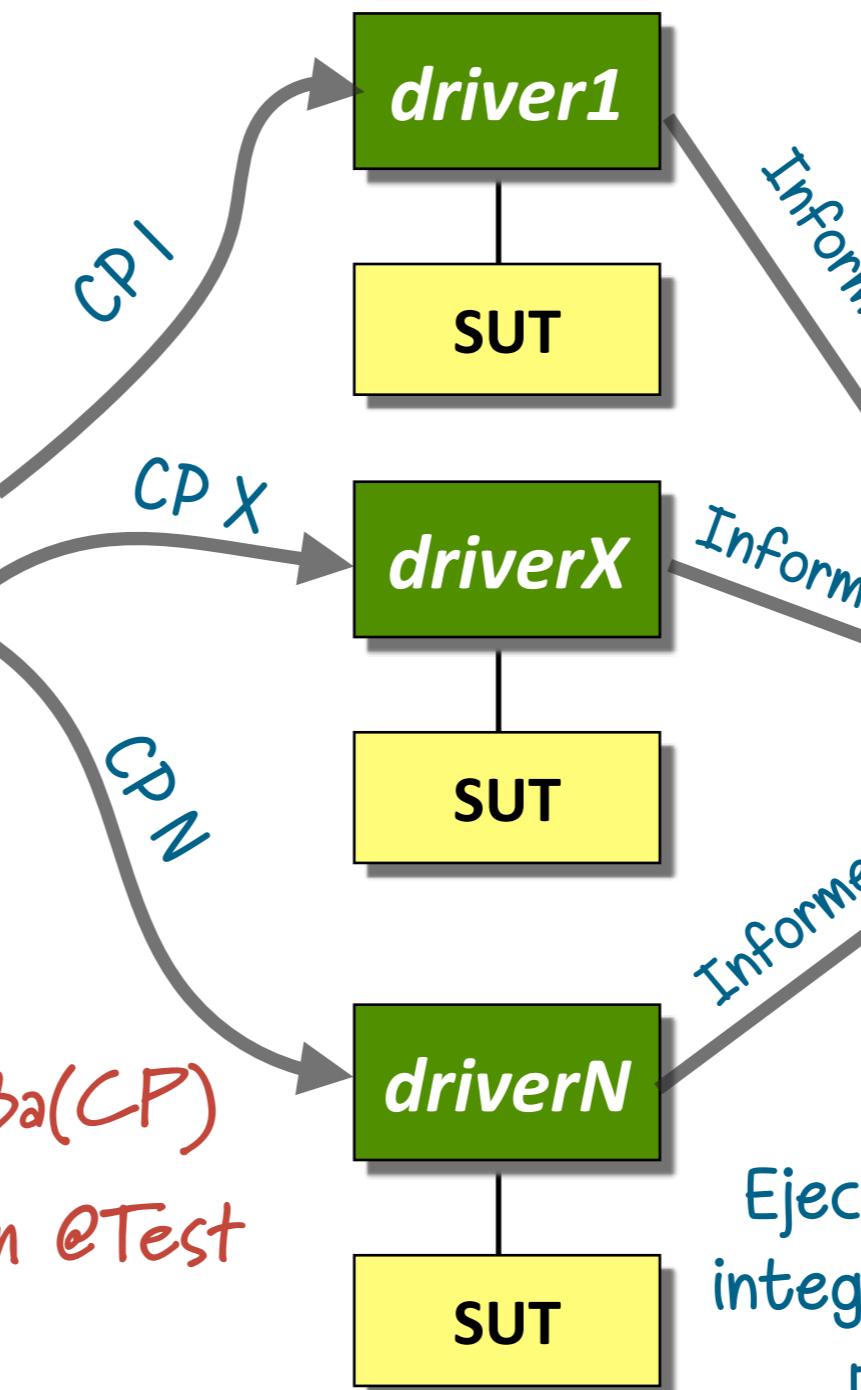
tabla de casos de prueba

Dato Entrada 1	Dato Entrada 2	Dato Entrada k	Resultado Esperado
d11=...	d21=...	dk1=...	r1
d1n=...	d2n=...	dkn=	r n

Implementaremos los drivers
utilizando JUnit 5

1 driver x cada caso de prueba(CP)

1 driver = método anotado con @Test



Ejecutaremos los tests JUnit
integrándolos en el ciclo de vida
por defecto de Maven

PREFERENCIAS BIBLIOGRÁFICAS

- P
 - Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
 - Capítulo 3: Unit Testing
 - JUnit 5 (<https://junit.org/junit5/docs/current/user-guide/>)
 - Annotations
 - Assertions
 - Tagging and Filtering
 - Parameterized Tests
 - Running tests
 - What's new in JUnit 5? (Blog Scott logic, 10 octubre 2017)
 - Java Lambda Expressions (tutorials.jenkov.com, 18 enero 2019)
 - Java Lambda Expressions Basics (Dzone, java zone, 2013)

Sesión S03: Diseño de pruebas: **caja negra**



You are a lucky bug. I'm seeing that you'll
be shipped with the next three releases

copyright 2005 Kazem A. Andekani

Diseño de casos de prueba: functional testing

- Objetivo: obtener una tabla de casos de prueba a partir del conjunto de **comportamientos especificados**
- El conjunto de casos de prueba obtenido debe detectar, dado un objetivo, el máximo número posible de defectos en el código, con el mínimo número posible de "filas" (efectividad y eficiencia)

Método de **Particiones equivalentes**

- Paso 1: Análisis de la especificación: Particionamos cada cada entrada (y salida) en conjuntos "equivalentes"
- Paso 2: Selección de comportamientos usando las particiones obtenidas:
- Paso 3. Obtención del conjunto de casos de prueba

Ejemplos y ejercicios

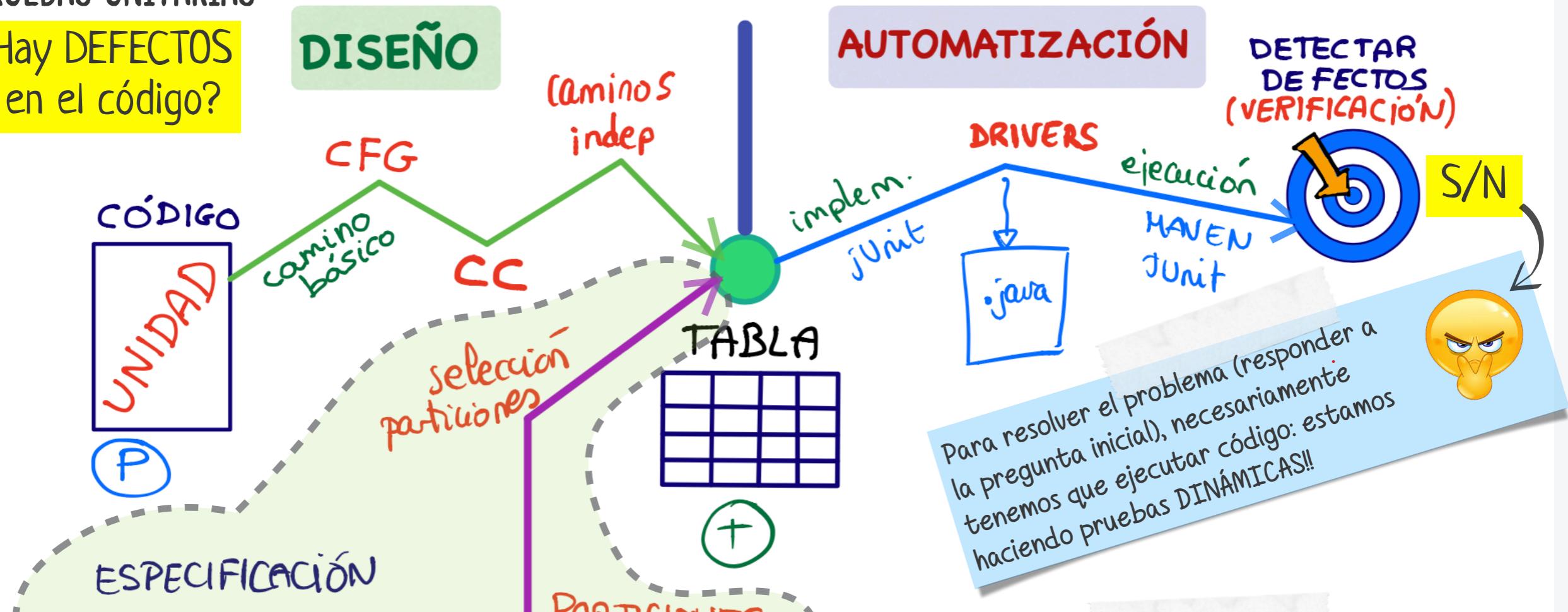
Vamos al laboratorio...

DISEÑO DE CASOS DE PRUEBA

Seleccionamos de forma sistemática un conjunto de casos de prueba efectivo y eficiente!!

PRUEBAS UNITARIAS

Hay DEFECTOS en el código?



Independientemente del método de diseño elegido,
SIEMPRE obtendremos un conjunto EFICIENTE y
EFECTIVO

Ya hemos visto una forma de seleccionar los comportamientos a probar a partir del código implementado (métodos estructurales)

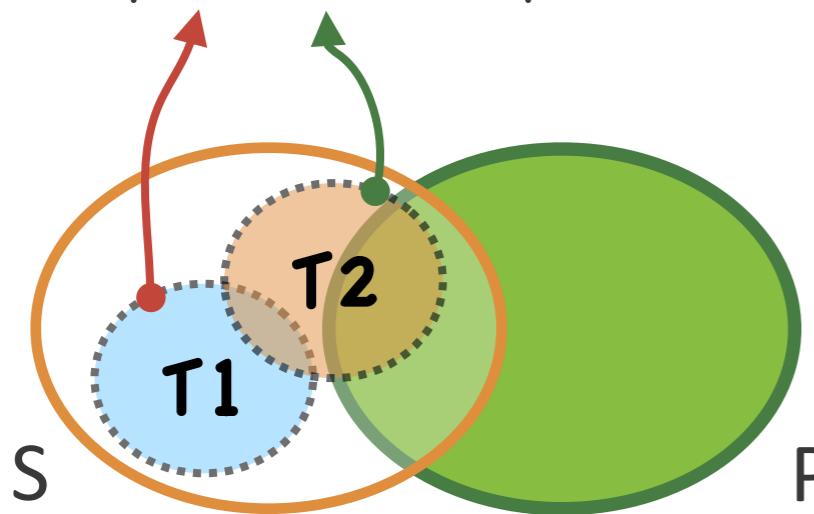
Ahora vamos a explicar cómo hacerlo partiendo de la **ESPECIFICACIÓN**.

AMBAS técnicas son necesarias y complementarias

FORMAS DE IDENTIFICAR LOS CASOS DE PRUEBA

FUNCTIONAL TESTING

Comportamientos probados



Podemos detectar comportamientos NO IMPLEMENTADOS

Nunca podremos detectar comportamientos implementados, pero no especificados

- 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



Los métodos de diseño basados en la ESPECIFICACIÓN:

1. **Analizan** la especificación y **PARTICIONAN** el conjunto S (dependiendo del método se puede usar una representación en forma de grafo)
2. **Seleccionan** un conjunto de **comportamientos** según algún criterio
3. **Obtienen** un conjunto de **casos de prueba** que ejercitan dichos comportamientos

MÉTODOS DE DISEÑO DE CAJA NEGRA

- P
- Existen MUCHOS métodos de diseño de pruebas de caja negra:

- Método de particiones equivalentes
- Método de análisis de valores límite
- Método de tablas de decisión

Pruebas unitarias

- Método de grafos causa-efecto
- Método de diagramas de transición de estados
- Método de pruebas basado en casos de uso

Pruebas de sistema

- Método de pruebas basado en requerimientos
- Método de pruebas basado en escenarios

Pruebas de aceptación

En todos ellos, la identificación de DOMINIOS de entradas y salidas contribuye a PARTICIONAR los comportamientos en clases (particiones)

A diferencia de los métodos de caja blanca, se pueden aplicar en CUALQUIER nivel de pruebas

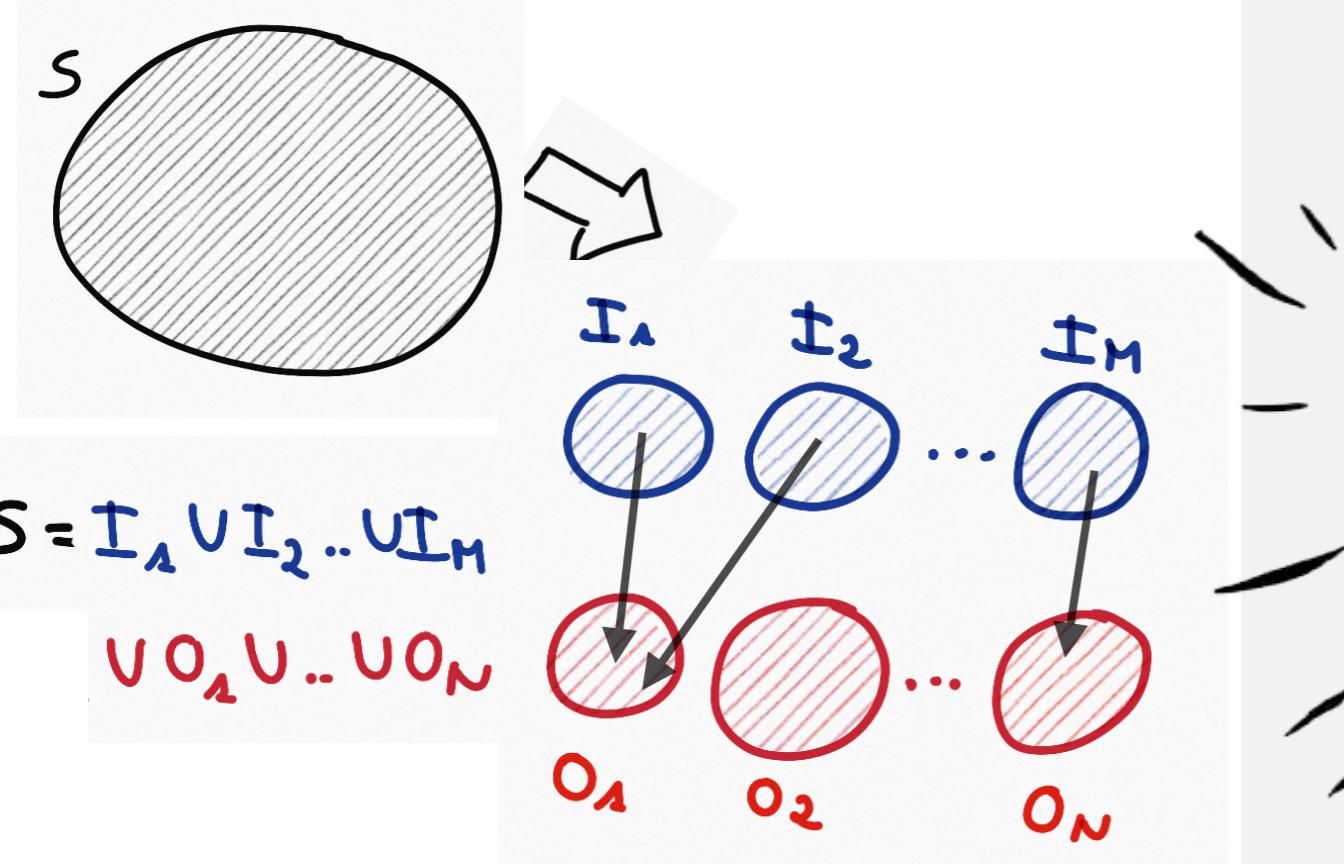
- Se trata de un proceso **SISTEMÁTICO** que identifica, a partir de la **ESPECIFICACIÓN** disponible, un conjunto de **CLASES** de equivalencia para **cada** una de las **entradas** y **salidas** del "elemento" (unidad, componente, sistema) a probar
- Cada clase de equivalencia (o partición) de entrada representa un subconjunto del total de datos posibles de entrada que tienen un mismo comportamiento (Los elementos de una partición de entrada se caracterizan por tener su "imagen" en la misma partición de salida)

P

El **OBJETIVO** es **MINIMIZAR** el número de casos de prueba requeridos para cubrir **TODAS** las particiones al menos una vez, teniendo en cuenta que las particiones de entrada inválidas se prueban de una en una.

MÉTODO DE DISEÑO: PARTICIONES EQUIVALENTES

Sesión 3: Diseño de caja negra



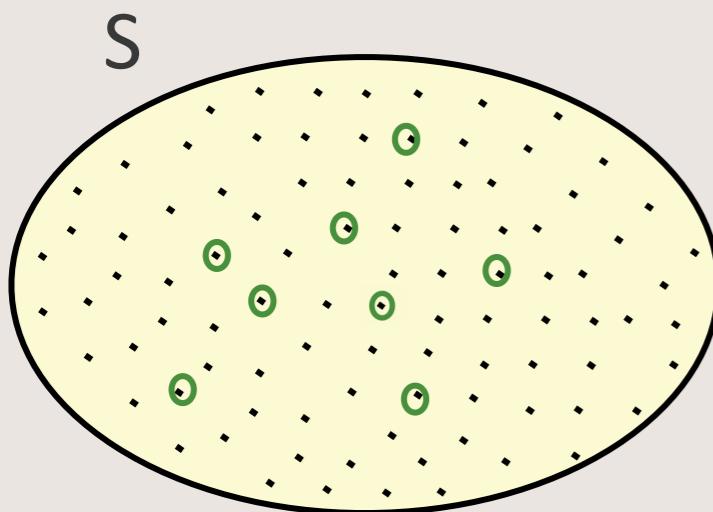
- Cada caso de prueba usará un **subconjunto** de particiones
- **NO** se trata de probar **TODAS** las combinaciones posibles, sino de garantizar que **TODAS** las particiones de entrada (y de salida) se prueban **AL MENOS UNA VEZ**

SISTEMATICIDAD Y PARTICIONAMIENTOS

detectar el máximo nº posible de errores

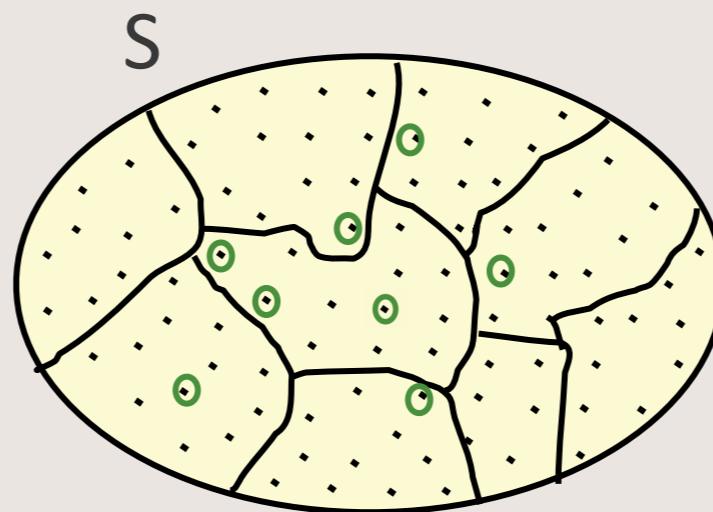
... con el MENOR nº posible de casos de prueba

- P Para conseguir un conjunto de pruebas EFECTIVO y EFICIENTE, tenemos que ser SISTEMÁTICOS a la hora de determinar las particiones de entrada/ salida
 - P Las particiones representan conjuntos de posibles comportamientos del sistema
 - P Se deben elegir muestras significativas de CADA partición
 - P Tenemos que asegurarnos de que cubrimos TODAS las particiones

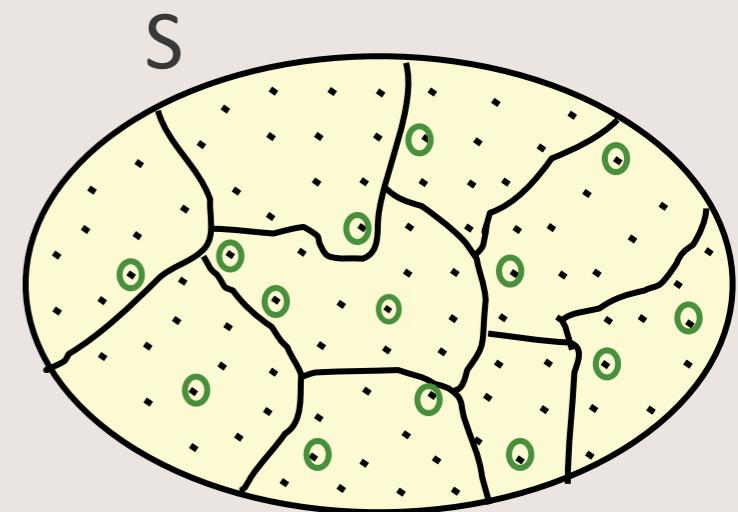


No particiones. Datos de prueba (círculos verdes) elegidos aleatoriamente

Las pruebas no son efectivas (hay tipos de comportamientos sin probar) ni eficientes (hay datos de prueba redundantes)



Particiones. Se eligen muestras de cada partición



Aquí aseguramos la efectividad del diseño (probamos TODOS los tipos de comportamientos diferentes). Mantenemos algunos datos de prueba redundantes.

¿CÓMO IDENTIFICAMOS UNA PARTICIÓN?

Particionamos CADA ENTRADA

- P Las particiones (o clases de equivalencia) se identifican en base a CONDICIONES de entrada/salida de la unidad a probar (de hecho en la literatura se utilizan indistintamente los términos partición de entrada, clase de equivalencia de entrada o condición de entrada)
- P Una condición de entrada/salida, puede aplicarse a una única variable de entrada/salida en una especificación o con un subconjunto de ellas
 - P.ej. Dados tres enteros: a, b, c, que representan los lados de un triángulo con valores positivos menores o iguales a 20 ...

* particiones de entrada:

- (1) $a, b, c > 0$ y $a, b, c \leq 20$
- (2) $a > 20$
- (3) $b > 20$
- (4) $c > 20$
- ...

la condición de entrada se aplica a las variables a, b y c

la condición de entrada sólo se aplica a una variable



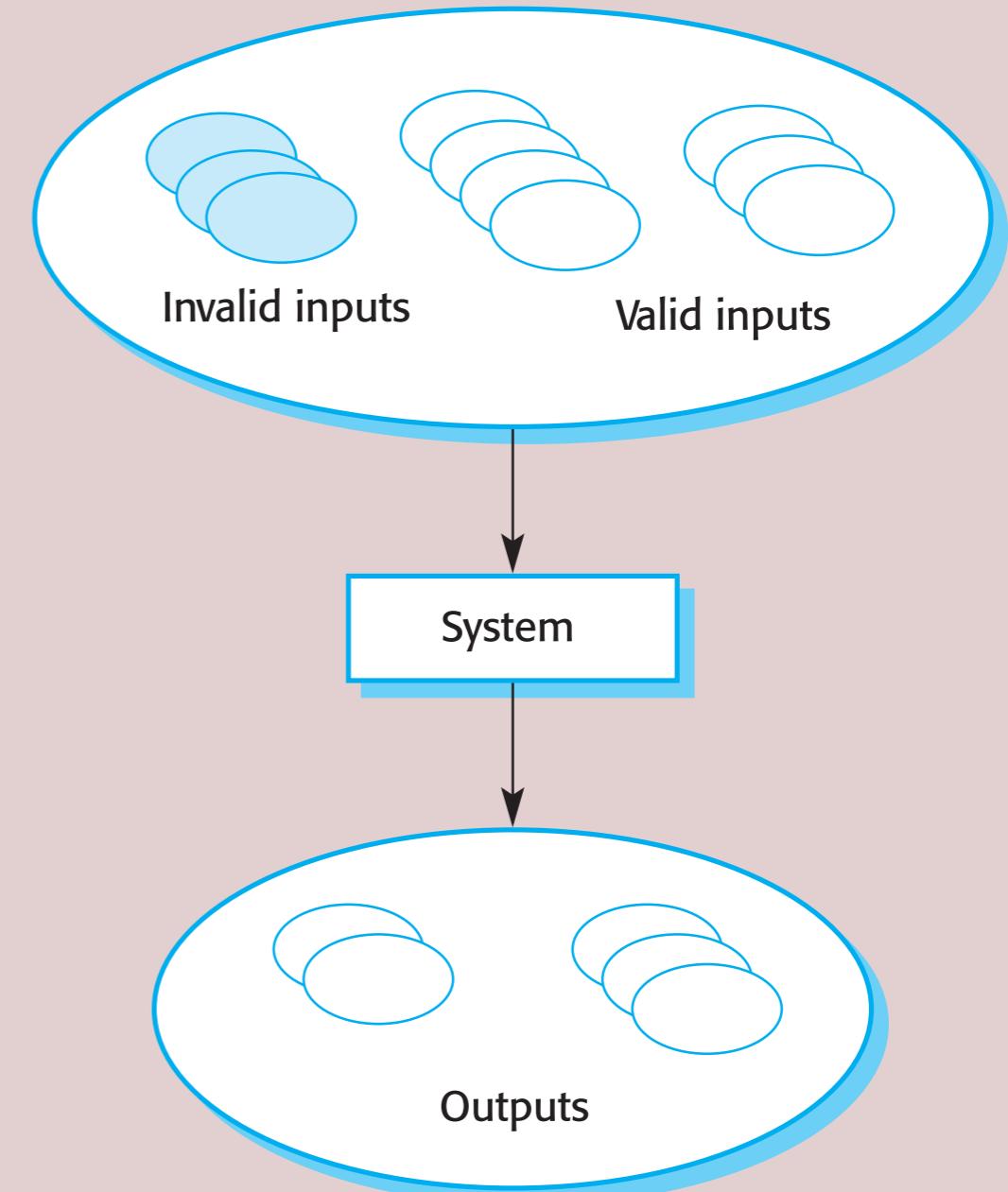
Lógicamente, para poder identificar las condiciones sobre las entradas, primero hay que tener claro cuántas y qué ENTRADAS tiene el elemento que queremos probar!!!!

MÁS SOBRE PARTICIONES DE ENTRADA/SALIDA

- Las "variables" de entrada/salida **no necesariamente** se corresponden con "parámetros" de entrada/salida de la unidad a probar
 - P.ej. El método `validar_pin()` comprueba si un pin (obtenido invocando a otra unidad) es válido o no. Si es válido, el método devuelve también el pin obtenido...
 - * Supongamos que el método a probar es:
 - **boolean validar_pin(Pin pinValido)**
 - * Las "variables" de entrada/salida que debemos considerar son:
 - Entradas: (1) tupla con un máximo de 3 "intentos", en donde cada intento = valor del pin obtenido por la unidad externa + código de respuesta devuelto por la unidad externa, (2) valor booleano que indica si el pin es válido
 - Salida: booleano + objeto pin válido
- Las particiones deben ser DISJUNTAS (las particiones No comparten elementos).
 - P.ej. Dada una entrada "a" que representa un entero, las particiones: (1) $a \geq 10$, y (2) $a \leq 10$ **NO** son disjuntas puesto que el valor $a=10$ pertenece a dos particiones diferentes
- Recordad además que todos los miembros de una partición de entrada deben tener su "imagen" en la misma partición de salida (si dos elementos de la misma partición de entrada se corresponden con dos elementos de particiones de salida diferentes, entonces la partición de entrada NO está bien definida)

PARTICIONES VÁLIDAS E INVÁLIDAS

- P
 - Las clases de equivalencia (condiciones, particiones) de entrada, pueden clasificarse como **VÁLIDAS** o **INVÁLIDAS**.
 - Ej: variable "mes" de tipo entero que representa un mes del año.
 - * Clase válida: Los valores 1..12 son valores válidos.
 - * Clases inválidas: Un valor superior a 12, o inferior a 1 podemos considerarlos inválidos.
 - Las particiones de entrada inválidas normalmente tienen asociadas clases de salida inválidas.



Sólo puede haber una partición INVÁLIDA de entrada en un caso de prueba

IDENTIFICACIÓN DE LAS CLASES DE EQUIVALENCIA



Debes usarlas
SIEMPRE!!

Paso 1. Identificar las clases de equivalencia (particiones) para **CADA** entrada/salida (E/S), siguiendo las siguientes HEURÍSTICAS:

- #1 Si la E/S especifica un **RANGO** de valores válidos, definiremos **una clase válida** (**dentro del rango**) **y dos inválidas** (**fueras de cada uno de los extremos del rango**). Ej. x puede tomar valores entre 1..12. Clase válida: $x = 1..12$; Clases inválidas: $x > 12$ y $x < 1$
- #2 Si la E/S especifica un **NÚMERO N** de valores válidos, definiremos **una clase válida** (**número de valores entre 1 y N**) **y dos inválidas** (**ningún valor, más de N valores**). Ej. x puede tomar entre 1 y 3 valores. Clase válida: x toma entre 1 y 3 valores; Clases inválidas: x no tiene ningún valor y x tiene más de 3 valores
- #3 Si la E/S especifica un **CONJUNTO** de valores válidos, definiremos **una clase válida** (**valores pertenecientes al conjunto**) **y una inválida** (**valores que no pertenecen al conjunto**). Ej. x puede ser uno de estos tres valores {valorA, valorB, valorC}. Clase válida: x toma uno de los valores \in al conjunto; Clase inválida: x toma cualquier valor que no \in al conjunto
- #4 Si por alguna razón, se piensa que cada uno de los **valores** de entrada se van a **tratar de forma diferente por el programa**, entonces **definir una clase válida para cada valor de entrada**
- #5 Si la E/S especifica una **situación DEBE SER**, definiremos **una clase válida y una inválida**. Ej. x comenzar por un número. Clase válida: x empieza con un dígito; Clase inválida: x NO empieza por un dígito
- #6 Si por alguna razón, se piensa que los **elementos de una partición** van a ser tratados de **forma distinta**, **subdividir la partición** en particiones más pequeñas

IDENTIFICACIÓN DE LOS CASOS DE PRUEBA

O **Paso 2.** Identificar los casos de prueba de la siguiente forma:



El orden
de los
pasos
importa!!

Debemos asignar un **IDENTIFICADOR ÚNICO** para cada partición

2.1 Hasta que todas las clases válidas estén cubiertas (probadas), escribir un nuevo caso de prueba que cubra el máximo número de clases válidas todavía no cubiertas

2.2 Hasta que todas las clases inválidas estén cubiertas (probadas), escribir un nuevo caso de prueba que cubra una y sólo una clase inválida (de entrada) todavía no cubierta

Si un caso de prueba contiene más de una clase de entrada NO válida, puede que alguna de ellas no se ejecute nunca, ya que alguna de las clases no válidas puede "enmascarar" a alguna otra, o incluso terminar con la ejecución del caso de prueba

2.3 Elegir un valor concreto para cada partición

El resultado de este proceso será una TABLA con tantas FILAS como CASOS de PRUEBA hayamos obtenido

- * Cada caso de prueba "cubrirá" un subconjunto de las particiones
- * El conjunto obtenido debe contemplarlas todas como mínimo una vez
- * Sólo puede haber una partición inválida de entrada en un caso de prueba

EJEMPLO 1: IMPRESIÓN DE CARACTERES

3 Entradas + 1 Salida

ESPECIFICACIÓN: Método en el que, dados como entradas: un carácter X introducido por el usuario, un número N entre 5 y 10, y el valor “rojo” o “azul”, devuelve (salida) una cadena de N caracteres X de color rojo o (N-1) caracteres de color azul, o bien el mensaje “ERROR: repite entrada” si el usuario proporciona un valor de N < 5 ó N >10.

- **Entrada 1 (E1) (carácter X):** puede ser cualquier carácter
 - Clase válida: V1
- **Entrada 2 (E2)(número N):** un valor comprendido entre 5 y 10
 - Clase válida: V2: valores entre 5 y 10 ($5 \leq N \leq 10$)
 - Clases inválidas: N1: valores menores que 5 ($N < 5$), y N2: valores mayores que 10 ($N > 10$)
- **Entrada 3 (E3).** uno de los valores: “rojo”, “azul”
 - Clases válidas: V3: “rojo”, V4: “azul”
- **Salida (cadena de N caracteres):**
 - Clase válida: S1: Cadena de N caracteres de color rojo
 - Clase válida: S2: Cadena de (N-1) caracteres de color azul
 - Clase inválida: NS1: “ERROR: repite entrada”

Opcionalmente, podríamos haber añadido
 N3: entrada con más de 1 carácter.
 NS2: ?? (salida no especificada)
 Pero E1 es de tipo "char", no son necesarias

Nota: nos indican que los valores “rojo” o “azul” se elegirán de una lista desplegable

Clases	Datos Entrada			Resultado Esperado
	E1	E2	E3	
V1-V2-V3-S1	‘c’	7	"rojo"	“ccccccc”
V1-V2-V4-S2	‘x’	6	"azul"	“xxxxx”
V1-N1-V4-NS1	‘c’	3	"azul"	“ERROR: repite entrada”
V1-N2-V4-NS1	‘j’	13	"azul"	“ERROR: repite entrada”

EJEMPLO 2: VALIDAR FECHA

2 Entradas + 1 Salida

P

S

ESPECIFICACIÓN: El método `valida_fecha()` tiene como parámetros de entrada las variables de tipo entero: día y mes, de forma que dados ambos valores, devuelve cierto o falso, en función de que sea una fecha válida. Supongamos que el año es 2021

P

○ En este caso:

- para realizar las particiones aplicamos las condiciones de entrada al subconjunto formado por día y mes, ya que:
 - * hay valores de entrada de una variable que pueden considerarse válidos o inválidos, dependiendo del valor de la otra variable. Por ejemplo el día 31, y los meses febrero y marzo
- por lo tanto consideraremos una única entrada:
 - * (dia, mes) agrupamos las 2 entradas en una para realizar las particiones
- y como salida:
 - * valor booleano indicando si la fecha es válida o no
- además, aplicaremos la regla #6, y subdividiremos tanto el día como el mes en particiones más pequeñas

Sesión 3: Diseño de caja negra



Regla #6: Si por alguna razón, se piensa que los elementos de una partición van a ser tratados de forma distinta, subdividir la partición en particiones más pequeñas

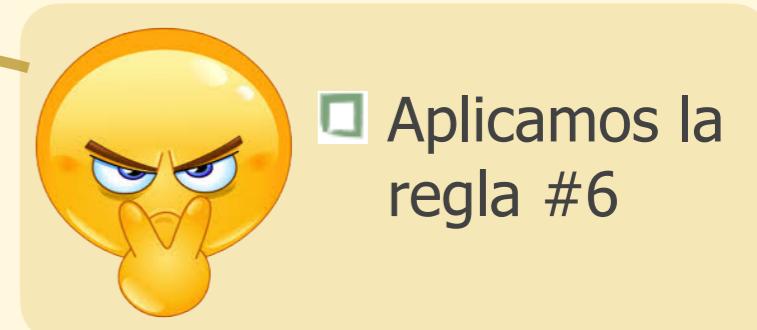
Primero hay que identificar las E/S y luego las particionamos (SIEMPRE!!!)

EJEMPLO 2: VALIDAR FECHA (PARTICIONES)

2 Entradas + 1 Salida
agrupamos las 2 en una única entrada
para realizar las particiones

- Aplicamos las condiciones de entrada a las variables de entrada y salida, de forma que obtenemos las siguientes particiones:
(Paso 1)

Particiones	
Entrada: dia(D) + mes(M)	Salida: S
DM1: d = {1..28} \wedge m = {1..12}	S1: true
DM2: d = {29,30} \wedge m = {1,3,..,12}	NS1: false
DM3: d = {31} \wedge m = {1,3,5,7,8,10,12}	
NDM1: d > 31 \wedge m = {1..12}	
NDM2: d < 1 \wedge m = {1..12}	
NDM3: d = {29,30} \wedge m = {2}	
NDM4: d = 31 \wedge m = {2,4,6,9,11}	
NDM5: m > 12 \wedge d = {1..31}	
NDM6: m < 1 \wedge d = {1..31}	



Si agrupas entradas,
NUNCA debes
considerar más de una
partición inválida en
dicha agrupación.

ENTRADA: 3 particiones válidas +
6 particiones inválidas

SALIDA: 1 partición válida +
1 partición inválida

EJEMPLO 2: TABLA RESULTANTE DE VALIDA_FECHA() (PASO 2)

- Una posible elección de casos de prueba podría ser ésta:

(Paso 2) ↗ entrada 1 ↗ entrada 2 ↗ salida 1

Particiones	dia	mes	salida
DM1-S1	14	5	true
DM2-S1	30	6	true
DM3-S1	31	7	true
NDM1-NS1	43	10	false
NDM2-NS1	-3	6	false
NDM3-NS1	30	2	false
NDM4-NS1	31	4	false
NDM5-NS1	29	16	false
NDM6-NS1	29	-3	false

siempre valores CONCRETOS!!!!



EJEMPLO 3: EL PROBLEMA DEL TRIÁNGULO

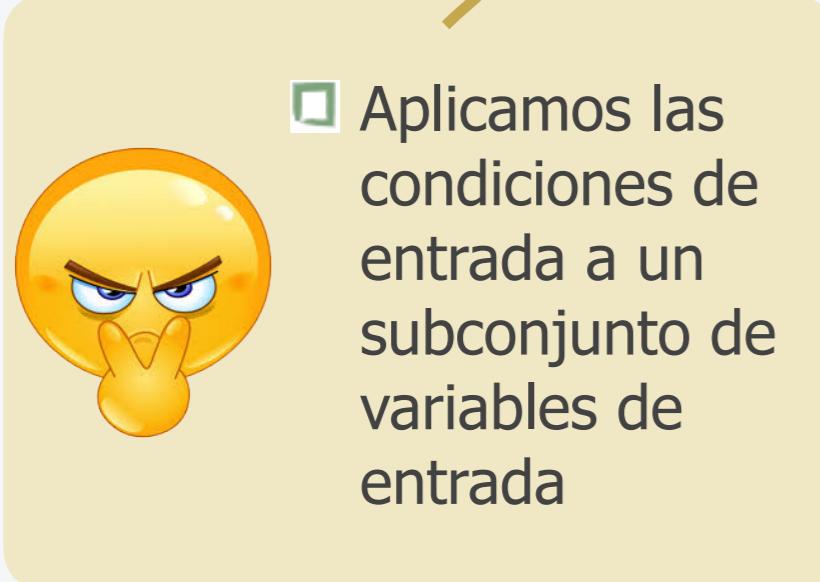
3 Entradas + 1 Salida

agrupamos las 3 en una única entrada
para realizar las particiones

ESPECIFICACIÓN: dados tres enteros: a , b , y c , que representan la longitud de los lados de un triángulo: cada uno de ellos debe tener un valor positivo menor o igual a 20. La unidad a probar, a partir de las entradas a , b y c devuelve el tipo de triángulo:

- * "Equilátero", si $a = b = c$
- * "Isósceles", si dos cualesquiera de sus lados son iguales y el tercero desigual
- * "Escaleno", si los tres lados son desiguales
- * "No es un triángulo", si $a \geq b+c$, ó $b \geq a+c$, ó $c \geq a+b$

○ Paso 1. Inicialmente podemos definir las siguientes particiones de entrada:



Aplicamos las condiciones de entrada a un subconjunto de variables de entrada

Entrada: (a,b,c)

$C_1: (a,b,c > 0) \wedge (a,b,c \leq 20)$	$NC_1 = a > 20$
	$NC_2 = b > 20$
	$NC_3 = c > 20$
	$NC_4 = a \leq 0$
	$NC_5 = b \leq 0$
	$NC_6 = c \leq 0$

EJEMPLO 3: PARTICIONES

S
ENTRADA: 4 particiones válidas +
6 particiones inválidas

- Utilizando la **heurística #6** del Paso 1, vamos a dividir C1 en subclases, puesto que diferentes combinaciones de valores de a,b, y c se van a tratar de forma diferente (darán lugar a diferentes salidas):
 - es-triángulo: $a < b+c \wedge b < a+c \wedge c < a+b$
- También particionamos las salidas

S
SALIDA: 4 particiones válidas +
1 partición inválida

Entrada: a,b,c	Salida	
C ₁₁ : a=b=c	NC ₁ = a > 20	S ₁ : "Equilátero"
C ₁₂ : (a=b \wedge a <> c) \vee (a=c \wedge a <> b) \vee (b=c \wedge b <> a)	NC ₂ = b > 20	S ₂ : "Isósceles"
C ₁₃ : (a <> b) \wedge (a <> c) \wedge (b <> c)	NC ₃ = c > 20	S ₃ : "Escaleno"
C ₁₄ : (a ≥ b+c) \vee (b ≥ a+c) \vee (c ≥ a+b)	NC ₄ = a ≤ 0 NC ₅ = b ≤ 0 NC ₆ = c ≤ 0	S ₄ : "No es triángulo" NS ₁ : ???

C11: valores de entrada correspondientes a un triángulo equilátero

C12: valores de entrada correspondientes a un triángulo isósceles

C13: valores de entrada correspondientes a un triángulo escaleno

C14: valores de entrada que se corresponden con valores válidos pero que no forman un triángulo

Las clases **C11**, **C12**, y **C13** incluyen, además, la condición es-triángulo

Qué ocurre si la entrada es NC_i?

EJEMPLO 3: TABLA DE CASOS DE PRUEBA

- La tabla resultante de casos de prueba puede ser ésta:

Clases	Datos Entrada	Resultado Esperado
C11-S1	a=11, b=11, c=11	"Equilátero"
C12-S2	a=7, b=7, c=6	"Isósceles"
C13-S3	a=10, b=3, c=9	"Escaleno"
C14-S4	a=8, b=2, c=4	"No es triángulo"
NC1-S5	a=30, b=15, c=6	???
NC2-S5	a=10, b=100, c=10	???
NC3-S5	a=7, b=14, c=21	???
NC4-S5	a=-5, b=10, c=11	???
NC5-S5	a=12, b=-10 c=10	???
NC6-S5	a=8, b=5, c=-1	???

1 Salida

3 Entradas



No debes asumir un resultado esperado que no esté indicado en la especificación

ALGUNOS CONSEJOS...

P



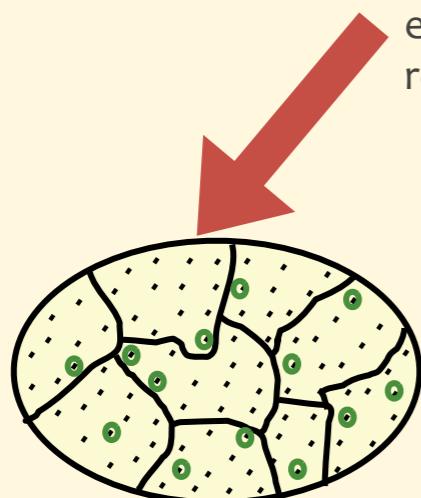
- Etiqueta las particiones de una misma E/S con la misma letra.
 - Por ejemplo, si las entradas son A y B, las particiones podrían etiquetarse como A₁, A₂,... N_{A1}, N_{A2},..., B₁, B₂,... N_{B1}, N_{B2},... para las clases válidas e inválidas de las entradas A y B respectivamente
- No olvides tener en cuenta las precondiciones de entrada/salida al realizar las particiones
 - P.ej. los valores de entrada se eligen de una lista desplegable (por lo tanto no debemos considerar la partición formada por los elementos que no pertenecen al conjunto)
- Si las E/S son objetos, tendrás que considerar cada atributo del objeto como un parámetro diferente.
 - P.ej. supón que una entrada es el objeto coordenadas (c), el cual tiene como atributos, valores de "x" e "y". Tendrás que hacer las particiones tanto para "c.x" como para "c.y"
- Los objetos en java siempre son referenciados por las variables. Por lo tanto tendremos que considerar el valor NULL como partición no válida al usar objetos
- Las E/S NO son ÚNICAMENTE parámetros de la unidad a probar.
 - P.ej. en el método realizaReserva() que hemos visto en prácticas, el resultado de invocar a la unidad reserva(socio.isbn) es una entrada para el método realizarReserva() que debemos incluir en la tabla de casos de prueba.

Y AHORA VAMOS AL LABORATORIO...

Usaremos el método de particiones equivalentes

ESPECIFICACIÓN

(unidad)



Método de
particiones
equivalentes

A nivel de unidad

Supongamos que queremos realizar pruebas sobre un **método** que calcula el nuevo importe de la renovación anual de una póliza de seguros. Si el asegurado es mayor de 25 años, y no tiene ningún parte de reclamación registrado en el último año, entonces se le incrementa en 25 euros el valor de la póliza, si tiene una reclamación, entonces se le incrementa en 50 euros, si tiene entre 2 y 4 reclamaciones, la actualización será de 200 euros y se le envía una carta al asegurado. Si el asegurado tiene 25 años o menos y ninguna reclamación cursada, la póliza se actualiza en 50 euros más. Si tiene una reclamación, se incrementa en 100 euros y se le envía una carta. Entre 2 y 4 reclamaciones, el incremento será de 400 euros y también se le envía una carta. Independientemente de la edad, si un asegurado tiene más de cinco reclamaciones se le cancelará la póliza

Identificaremos casos de prueba
utilizando caja negra

Entrada 1 (A): Particiones válidas: A1, A2, A3
Particiones inválidas : NA1, NA2

...
Entrada k: (K) Particiones válidas: K1, K2, K3
Particiones inválidas: NK1

Salida 1 (S): Particiones válidas: S1, S2, S3
Particiones inválidas: NS1, NS2

...
Salida P: Particiones válidas: P1, P2
Particiones inválidas: NP1, NP2

Tabla de casos de prueba

Particiones	Identificador	Datos Entrada	Resultado Esperado
A1- B1- ... - K1	C1	d1=... d2=... ...	r1= ... r2= ...
...	...		
AX- BY- ... - NKZ	CM	d1=... d2=... ...	r1= .. r2= ...

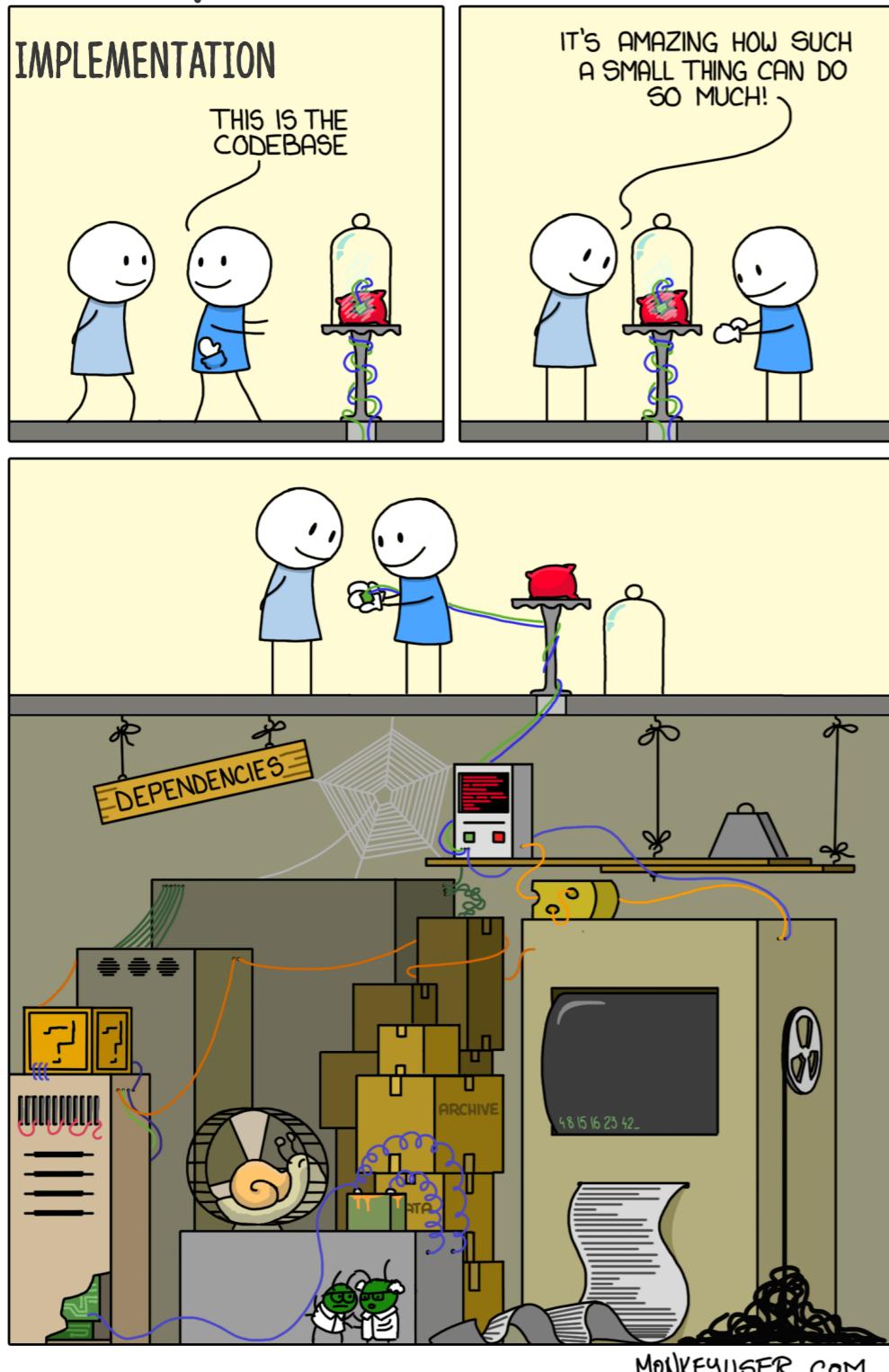
La utilizaremos en la siguiente práctica!!!...

REFERENCIAS



- A practitioner's guide to software test design. Lee Copeland.
Artech House Publishers. 2007
 - Capítulo 3: Equivalence Class Testing
- Pragmatic software testing. Rex Black. Wiley. 2007
 - Capítulo 11: Equivalence Classes Exercise

Sesión S05: Dependencias externas



Pruebas **unitarias**: implementación de drivers utilizando verificación basada en el **estado**

Conceptos de código testable y "seam"

Proceso para **aislar** la unidad de sus dependencias externas (control de entradas indirectas):

- Paso 1: Identificación de dependencias externas
- Paso 2: Refactorización de la unidad (sólo si es necesario) para conseguir injectar los dobles de las dependencias externas
- Paso 3: Control de las dependencias externas: implementamos un doble (stub) para controlar las entradas indirectas al SUT
- Paso 4: Implementación del driver utilizando verificación basada en el estado

Vamos al laboratorio...

PRUEBAS UNITARIAS Y DEPENDENCIAS EXTERNAS

P

Las pruebas de unidad dinámicas requieren **ejecutar cada unidad** (SUT: System Under Test) de forma **AISLADA** para poder detectar defectos (bugs) en dicha unidad

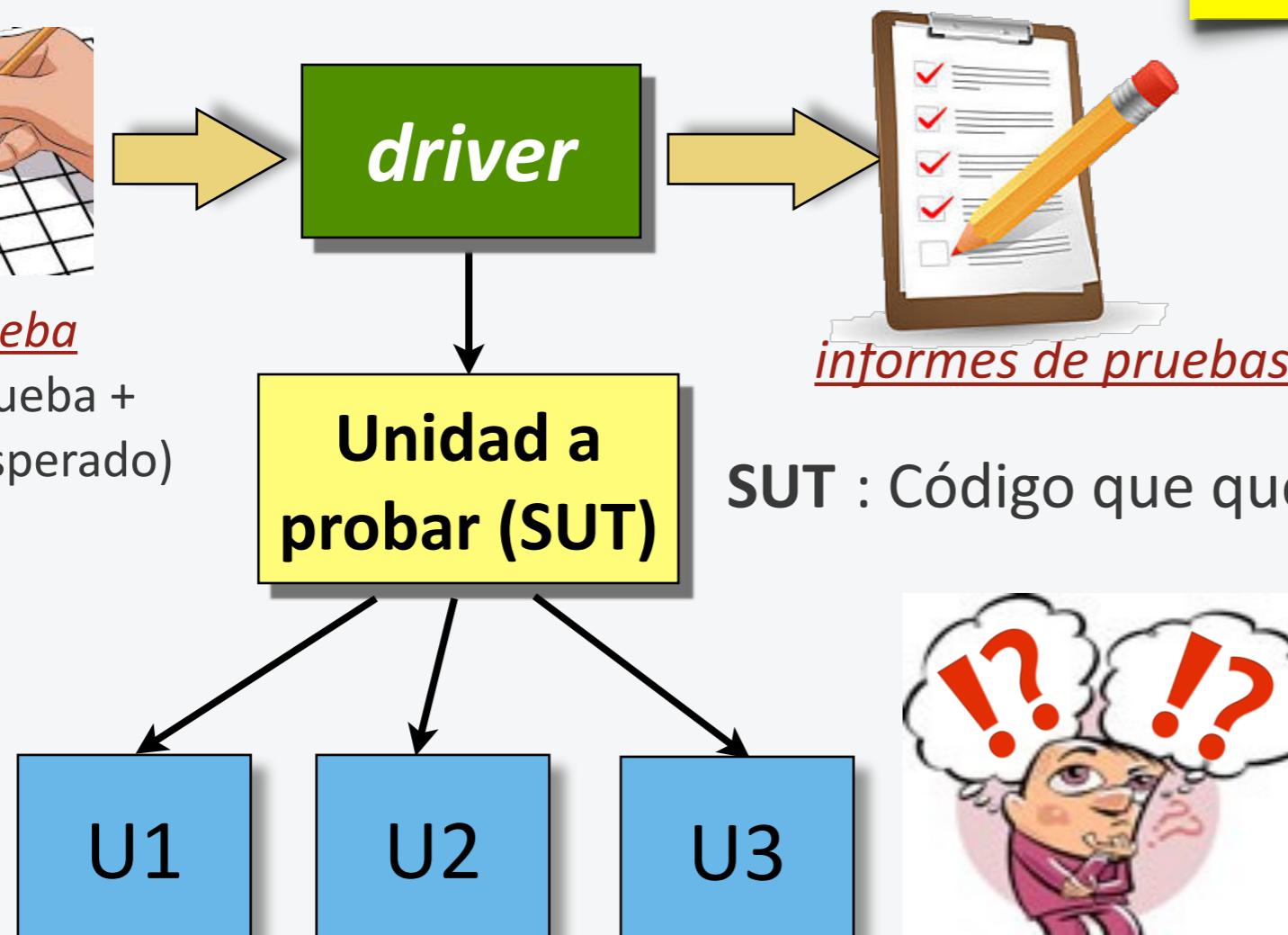
P

Verificación
Objetivo: encontrar **DEFECTOS** en el código de las **UNIDADES** probadas



casos de prueba

(datos de prueba + resultado esperado)



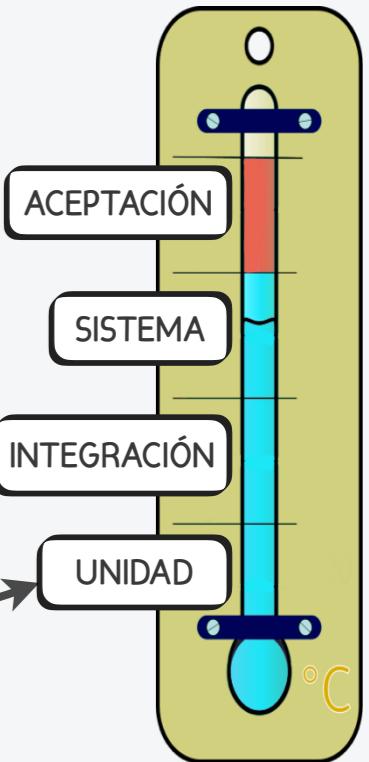
SUT : Código que queremos probar.

SUT= método Java

¿Qué ocurre si desde SUT se invoca a otras unidades ???? ...



... que necesitaremos CONTROLAR la ejecución de dichas dependencias externas si queremos AISLAR el código a probar!!!!!!



LA REGLA "DE ORO" PARA REALIZAR LAS PRUEBAS

El código de la unidad a probar (SUT) tiene que ser **exactamente el mismo código** que se utilizará en producción.



Es decir, no está permitido "alterar circunstancialmente/temporalmente" el código de SUT de ninguna forma con el propósito de realizar las pruebas.

Por ejemplo: supongamos que queremos realizar una prueba unitaria sobre el método GestorPedidos.generarFacturas()

```
public class GestorPedidos {  
    private Facturas facturas;  
  
    1. public Facturas generarFacturas() {  
    2.     //... código  
    3.     boolean ok = facturas.pendientes();  
    4.     if (ok) {  
    5.         //sentencias then  
    6.     } else {  
    7.         //sentencias else  
    8.     }  
    9.     return facturas;  
10. }
```

SUT

NO estamos interesados en ejecutar el código del que depende nuestro SUT

La variable "ok" es una entrada INDIRECTA de la unidad (SUT)

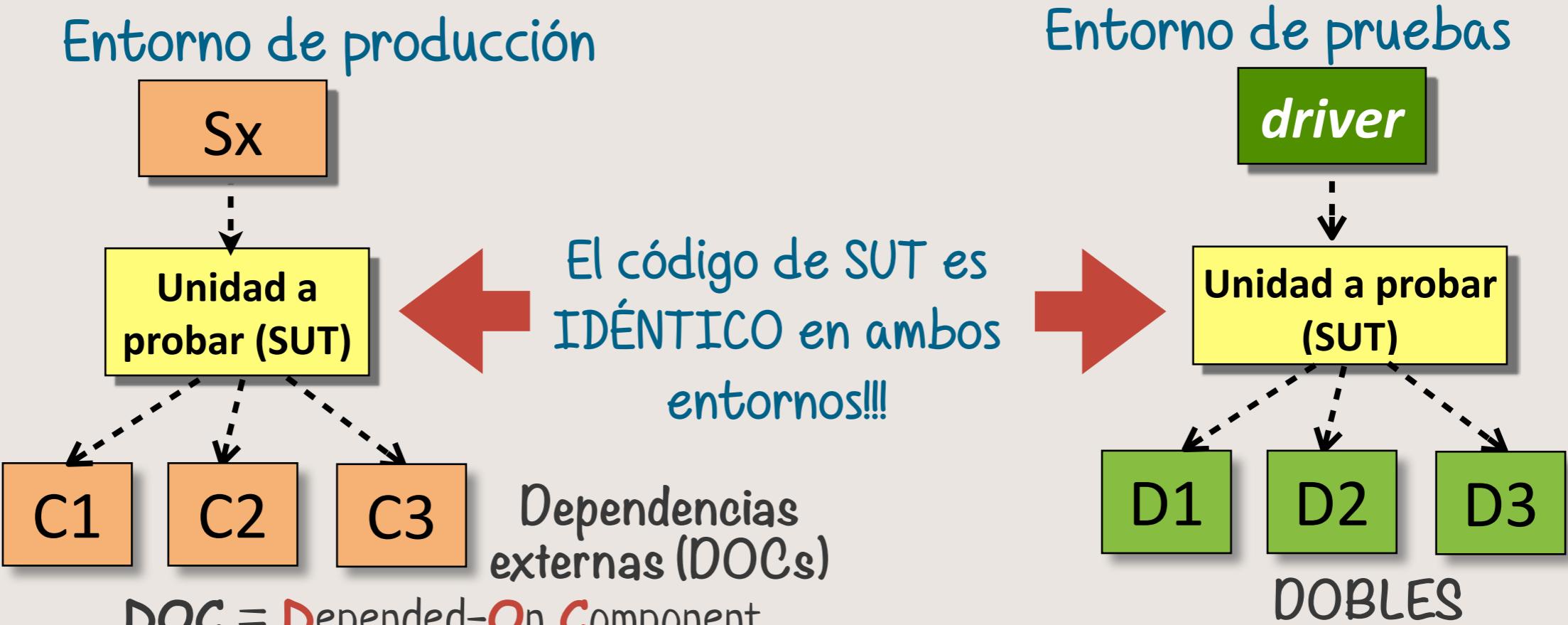
```
public class GestorPedidos {  
    private Factura factura;  
  
    1. public Facturas generarFacturas() {  
    2.     //... código  
    3.     //boolean ok = facturas.pendientes();  
    4.     ok = true;  
    5.     if (ok) {  
    6.         //sentencias then  
    7.     } else {  
    8.         //sentencias else  
    9.     }  
10.     return facturas;  
11. }
```

La opción: "Comentamos temporalmente la línea 3 y añadimos la línea 4 sólo para poder hacer las pruebas. Después de hacer las pruebas volveremos a dejar nuestro SUT como estaba", **ESTÁ TOTALMENTE PROHIBIDA!!!**

CÓDIGO TESTABLE Y CONTROL DE DEPENDENCIAS

<http://www.loosecouplings.com/2011/01/testability-working-definition.html>

- Podemos definir un **código testable** como aquél que permite que un componente sea fácilmente probado de forma AISLADA
 - Para poder probar un componente de forma aislada debemos ser capaces de **CONTROLAR** sus **DEPENDENCIAS** externas, también denominadas **COLABORADORES**, o **DOCs**
 - Una **dependencia externa** es un objeto con quién interactúa nuestro código a probar (más concretamente, con uno de sus métodos) y sobre el que no tenemos ningún control



Para poder realizar este REEMPLAZO controlado necesitamos que SUT contenga uno (o varios) **SEAMS!!!!**

CONCEPTO DE SEAM

"A seam is a place where you can alter behavior in your program without editing in that place"

Michael Feathers

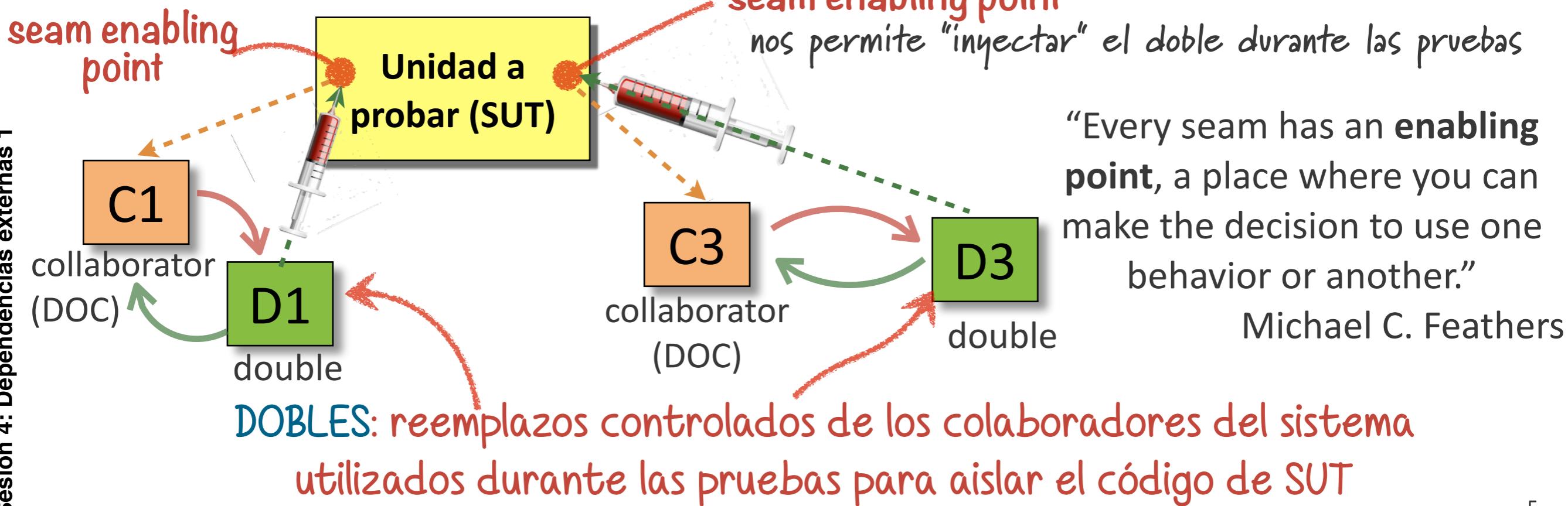
<http://www.informit.com/articles/article.aspx?p=359417&seqNum=3>

Para poder conseguir un seam en nuestro código PUEDE que necesitemos REFACTORIZAR nuestro SUT

"Refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior... It is a disciplined way to clean up code that minimizes the chances of introducing bugs"

Martin Fowler and Kent Beck

<http://agile.dzone.com/articles/what-refactoring-and-what-it-0>



CÓMO IDENTIFICAR UN SEAM

P

- Dadas las siguientes clases, ¿cuál de los tres métodos ejecutaremos si tenemos la siguiente sentencia?

P

- myCell.recalculate();**
- Si no conocemos el tipo del objeto "myCell", no podemos saber a qué método se invocará desde esta línea de código
- Si **podemos cambiar el método** que se invocará desde esta línea SIN alterar el código que la unidad que la contiene, entonces esta línea de código **es un SEAM**

- En un lenguaje orientado a objetos, no todas las llamadas a métodos son seams:

```
public class CustomSpreadsheet extends Spreadsheet {
    public Spreadsheet buildMartSheet() {
        ...
        Cell myCell = new FormulaCell(this, "A1", "=A2+A3");
        ...
        myCell.recalculate()
    }
    ...
}
```

CÓDIGO NO TESTABLE!!

NO es un seam, ya que no podemos cambiar el método al que se invocará sin modificar el código

SUT

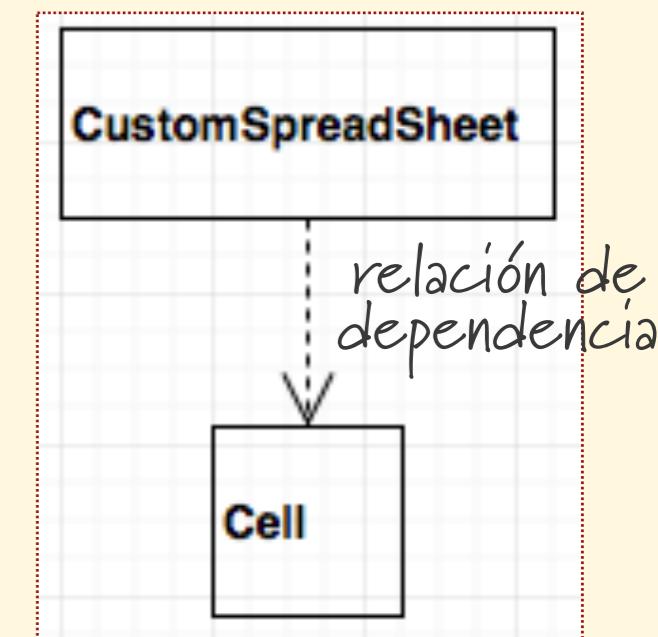
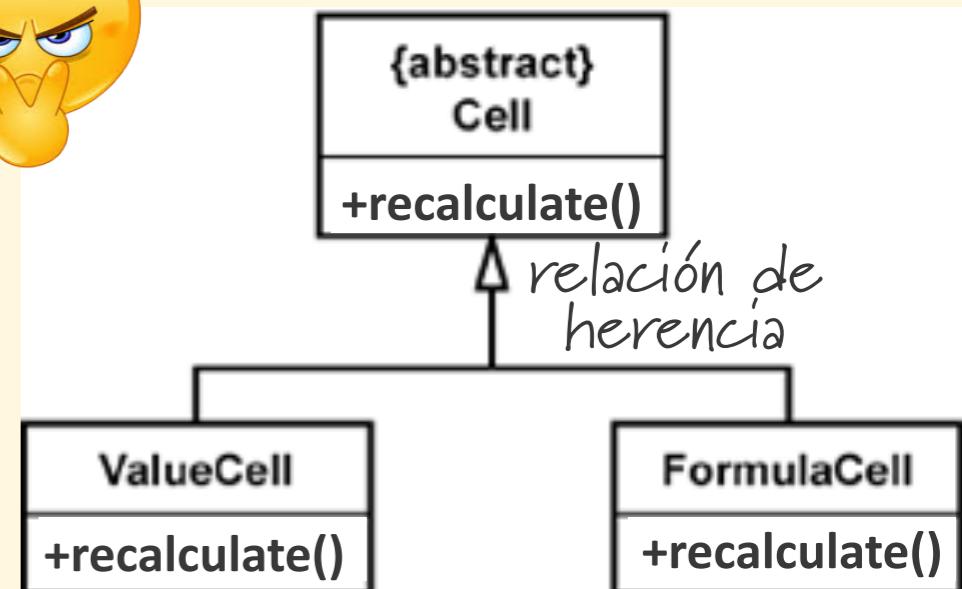
Sesión 4: Dependencias externas 1

SUT

```
public class CustomSpreadsheet extends Spreadsheet {
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        cell.recalculate()
    }
    ...
}
```

CÓDIGO TESTABLE!!

SÍ es un seam, ya que podemos cambiar el método al que se invocará sin modificar el código



seam enabling point

Ejecutaremos **ValueCell.recalculate()** o **FormulaCell.recalculate()** dependiendo del tipo de objeto que inyectemos. Podemos inyectar cualquier **subtipo** de Cell

SEAM: MÁS EJEMPLOS

P

- Cada SEAM debe tener un "punto de inyección", que nos permitirá, durante las pruebas, reemplazar cada dependencia externa por su doble (SIN alterar el código de SUT).
- El código de nuestro SUT durante las pruebas debe ser idéntico al de producción!!!

```
public class CustomSpreadsheet extends Spreadsheet {
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        recalculate(cell);
        ...
    }
    protected void recalculate(Cell cell) {
        ...
    }
}
```

CÓDIGO TESTABLE!!

SÍ es un seam

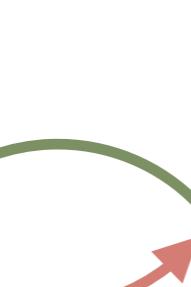
Usaremos esta clase durante las pruebas e invocaremos al doble el lugar de al DOC

```
public class TestableCustomSpreadsheet
    extends CustomSpreadsheet {
    @Override
    protected void recalculate(Cell cell) {
        ...
    }
}
```

```
public class CustomSpreadsheet extends Spreadsheet {
    public Cell getCell() {
        return new Cell();
    }
    public Spreadsheet buildMartSheet() {
        ...
        Cell myCell = getCell();
        ...
        myCell.recalculate();
        ...
    }
}
```

CÓDIGO TESTABLE!!

Inyectaremos el doble durante las pruebas



SÍ es un seam

```
public class TestableCustomSpreadsheet
    extends CustomSpreadsheet {
    @Override
    public Cell getCell() {
        ...
    }
}
```

```
public class DobuleCell extends Cell {
    @Override
    public void recalculate() {
        ...
    }
}
```

PASOS A SEGUIR PARA AUTOMATIZAR LAS PRUEBAS

Identificar las dependencias externas de nuestro SUT



Colaboradores (DOCs)

Asegurarnos de que nuestro código (SUT) es TESTABLE: puede ser probado de forma aislada.

Para ello tendrá que poderse realizar un reemplazo controlado de cada dependencia externa por su doble SIN modificar su código

Puede que necesitemos REFACTORIZAR nuestro SUT (y/o la clase a la que pertenece) para poder realizar dichos reemplazos controlados durante las pruebas

Proporcionar una implementación ficticia (DOBLE), que reemplazará al código real de cada dependencia externa durante las pruebas



Dobles

Implementar los DRIVERS correspondientes. Para ello podemos hacer una:

verificación basada en el ESTADO:

sólo estamos interesados en comprobar el estado resultante de la invocación de nuestro SUT (implementaremos el driver como ya hemos visto en las sesiones anteriores)

driver

verificación basada en el COMPORTAMIENTO:

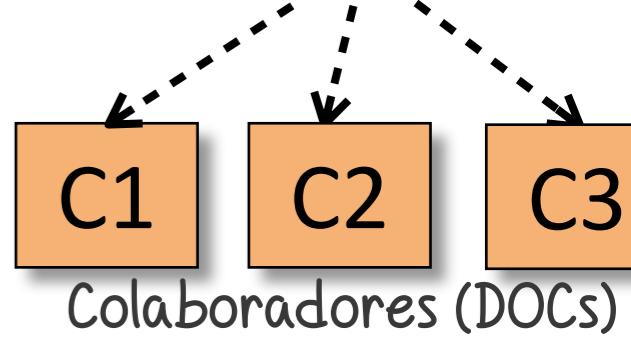
nos interesa, además, verificar que las interacciones entre nuestro SUT y las dependencias externas se realizan correctamente

P



DEPENDENCIAS EXTERNAS

Cuántas y qué dependencias externas tiene nuestra SUT??



P

- Una **dependencia externa** es otra **UNIDAD** en nuestro sistema con la cual interactúa nuestro código a probar (SUT), y sobre la que no tenemos ningún control
 - Nuestro test (driver) no puede controlar lo que dicha dependencia devuelve a nuestro código a probar ni cómo se comporta
 - Utilizaremos dobles para controlar el resultado de nuestra dependencia externa y así probar nuestra unidad de forma AISLADA

- Ejemplo:

```

public class GestorPedidos {

    public Factura generarFactura(Cliente cli) throws FacturaException {
        Factura factura;
        Buscador buscarDatos = new Buscador();

        int numElems = buscarDatos.elemPendientes(cli);
        if (numElems>0) {
            //código para generar la factura
            factura = ...;
        } else {
            throw new FacturaException("No hay nada pendiente de facturar");
        }
        return factura;
    }
}
  
```

SUT

1 Dependencias externas

generarFactura

depende de...

Buscador.elemPendientes

1 dependencia externa



Estamos interesados en aislar nuestro SUT. Por lo tanto NO queremos ejecutar los tests sobre la implementación real del método elemPendientes(), solamente nos interesa controlar el valor que devuelve este método



NUESTRO SUT DEBE SER TESTABLE



Para que sea testable, debe contener un SEAM

P

P

- Necesitamos poder cambiar la dependencia real por su doble (sin alterar el código de nuestro SUT). Esto no será posible si no tenemos un seam para CADA dependencia externa, que nos permita "inyectar" nuestro doble durante las pruebas.
- Dado que vamos a trabajar con Java:
 - Nuestro **DOBLE** debe **IMPLEMENTAR** la misma **INTERFAZ** que el colaborador (DOC), o
 - debe **EXTENDER** la misma **CLASE** que el colaborador (DOC)
- Nuestra SUT será **TESTABLE** si podemos "inyectar" dicho doble en nuestra SUT durante las pruebas de alguna de las siguientes formas:
 - (1) como un parámetro de nuestra SUT
 - (2) a través del constructor de la clase que contiene nuestra SUT
 - (3) a través de un método setter de la clase que contiene nuestra SUT
 - (4) a través de un método de fábrica local de la clase que contiene nuestra SUT, o una clase fábrica
- Si nuestra SUT NO es testable, entonces tendremos que **REFACTORIZAR** el código de nuestra SUT para que podamos injectar el doble de alguna de las formas anteriores, teniendo en cuenta que:
 - (1) SI añadimos un parámetro a nuestra SUT, estamos OBLIGANDO a que cualquier código cliente de nuestra SUT tenga que CONOCER dicha dependencia ANTES de invocar a nuestra SUT
 - (2-3) SI añadimos un parámetro al constructor de nuestra SUT, (o un método setter) estamos OBLIGADOS a declarar la dependencia (DOC) como un atributo de la clase que contiene nuestro SUT.
 - (3) No podremos añadir un método setter si el constructor realizase alguna acción significativa sobre nuestra dependencia. Además, tenemos que asumir que no se ejecutarán de forma automática acciones "intermedias" entre la invocación al constructor y al setter.
 - (4) Si usamos un método de fábrica local, no se ven afectados, ni los clientes de nuestro SUT, ni la estructura de la clase que contiene nuestro SUT, aunque alteramos el comportamiento de la clase que contiene nuestro SUT al añadir un nuevo método. Una clase fábrica implica añadir código en src/main/java que puede ser innecesario en producción.



P



NUESTRO SUT DEBE SER TESTABLE

P

Refactorizaremos para poder injectar el doble durante las pruebas

Y si no lo es, lo REFACTORIZAREMOS
... pero sólo SI ES NECESARIO!!!

Nuestro SUT es testable???



```
public class GestorPedidos {
    public Factura generarFactura(Cliente cli) throws FacturaException {
        Factura factura;
        Buscador buscarDatos = new Buscador(); ←
        int numElems = buscarDatos.elemPendientes(cli);
        if (numElems>0) {
            //código para generar la factura
            factura = ....;
        } else {
            throw new FacturaException("No hay nada pendiente de facturar");
        }
        return factura;
    }
}
```

Versión original

SUT

NO, porque no podemos invocar a un doble
de elemPendientes() SIN cambiar el
código de nuestro SUT

Como NO es testable,
tendremos que
REFACTORIZARLO
para que sea testable

Si, por ejemplo,
decidimos usar la
opción (l), nuestra SUT
quedaría así:

Sustituimos la versión
original de nuestro
SUT por la versión
refactorizada!!!

```
...
public Factura generarFactura(Cliente cli, Buscador buscar)
    throws FacturaException {
    Factura factura;
    int numElems = buscar.elemPendientes(cli);
    if (numElems>0) {
        //código para generar la factura
        factura = ....;
    } else {
        throw new FacturaException("No hay nada pendiente de facturar");
    }
    return factura;
}
...
```

Versión
refactorizada (en
src/main/java)

SUT

SUT REFACTORIZADA. AHORA SÍ ES TESTABLE!!!

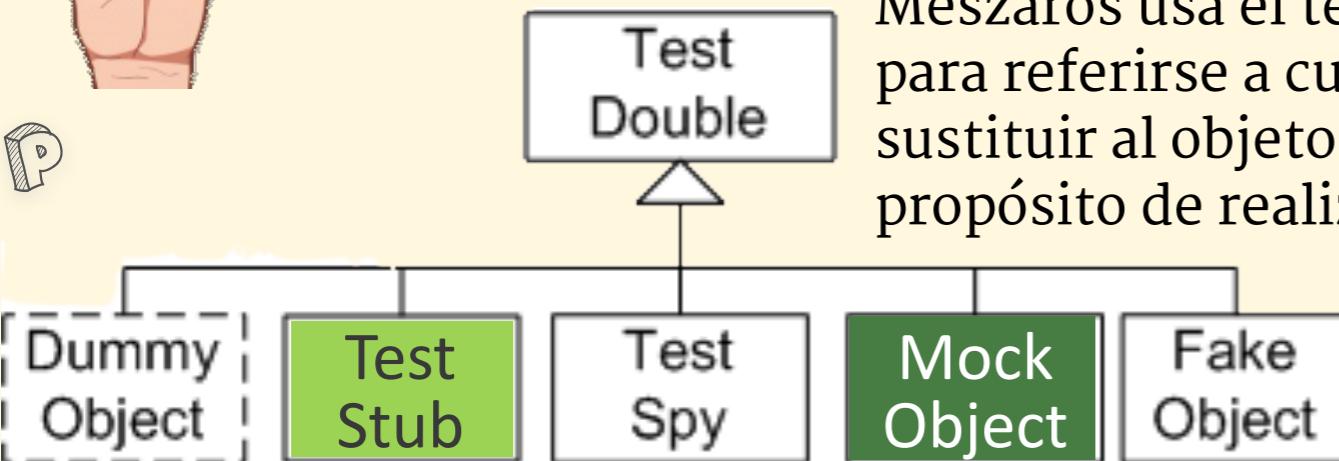


IMPLEMENTAMOS EL DOBLE



hay varios tipos de dobles

<http://xunitpatterns.com/Using%20Test%20Doubles.html>



Meszaros usa el término **Test Double** como un término genérico para referirse a cualquier objeto (o componente) que se utilice para sustituir al objeto o componente real (usado en producción), con el propósito de realizar pruebas

Test Stub

Es un objeto que actúa como un punto de CONTROL para entregar **ENTRADAS INDIRECTAS** al SUT, cuando se invoca a alguno de los métodos de dicho stub.

Un stub utiliza verificación basada en el estado

Hablaremos de este tipo de doble en la siguiente sesión

Mock Object

Es un objeto que actúa como un punto de observación para las **SALIDAS INDIRECTAS** del SUT.

Puede devolver información cuando se le invoca (igual que un stub), o no devolver nada.

Además registra las llamadas recibidas del SUT, y compara las llamadas reales con las llamadas previamente definidas como expectativas, de forma que hacen que el test falle si no se cumplen dichas expectativas.

Un mock utiliza verificación basada en el comportamiento

Usos de un Test Double

- Aislar el código a probar (pruebas unitarias)
- Acelerar la velocidad de la ejecución de los tests (un doble tiene mucho menos código que el objeto al que sustituye). (en pruebas de integración y/o sistema)
- Conseguir ejecuciones deterministas cuando el comportamiento depende de situaciones aleatorias o dependientes del tiempo
- Simular condiciones especiales. Por ejemplo una caída en la red
- Conseguir acceder a información oculta. Por ejemplo, comprobar si se ha invocado un determinado método dentro del SUT (test spy)
- Controlar entradas indirectas (stub) o salidas indirectas (mock) del SUT



IMPLEMENTAMOS EL DOBLE

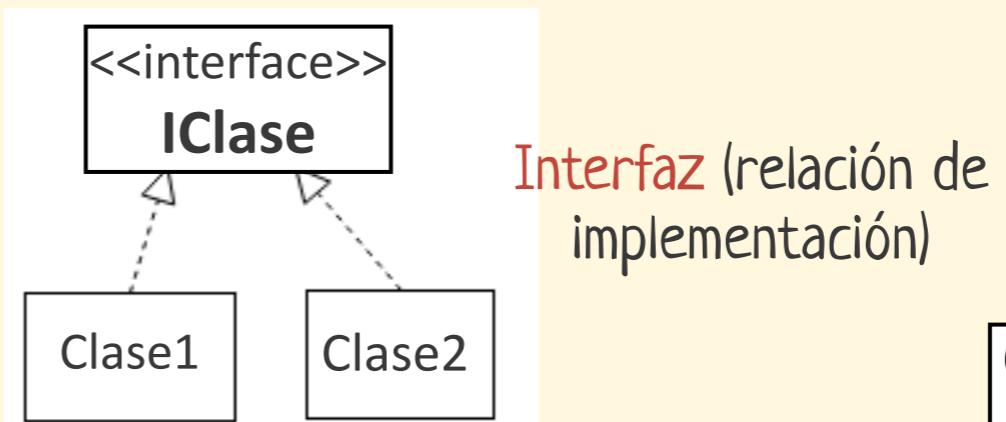
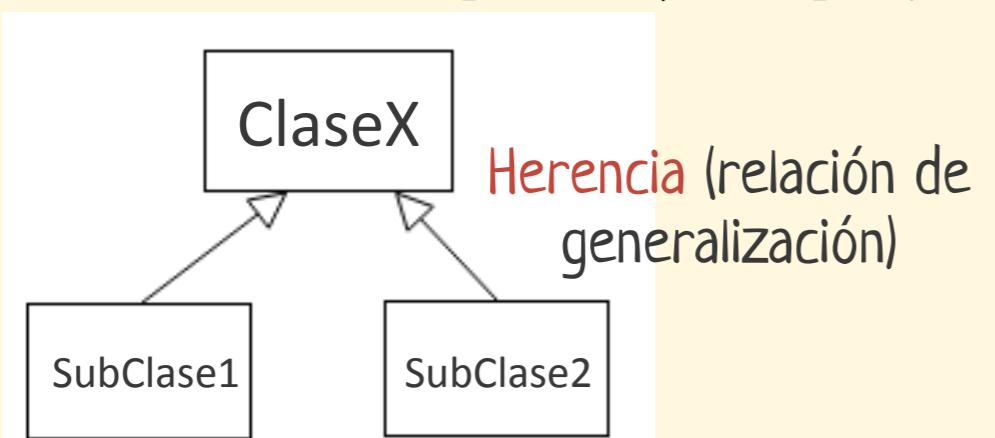
S nuestro doble pertenecerá a una clase que "extienda" o "implemente" la clase que contiene nuestro SUT

**Don't
FORGET!**



P Recuerda que nuestra dependencia externa siempre será un método Java. Por lo tanto nuestro doble debe consistir en una implementación ALTERNATIVA del MISMO método Java.

P En un lenguaje orientado a objetos, podemos usar los mecanismos de herencia y/o interfaces para implementar nuestros dobles, ya que podremos reemplazarlos por nuestro DOC cuando estemos ejecutando nuestras pruebas, siempre y cuando podamos inyectar dicho doble en nuestro SUT



```
public class SubClase1 extends ClaseX ...
public class SubClase2 extends ClaseX ...
...
ClaseX ejemplo1;
...
//estas tres asignaciones son VÁLIDAS
ejemplo1 = new ClaseX();
ejemplo1.metodoA(); //de ClaseX
ejemplo1 = new SubClase1();
ejemplo1.metodoA(); //de SubClase1
ejemplo1 = new SubClase2();
ejemplo1.metodoA(); //de SubClase2
```

```
public class Clase1 implements IClase;
public class Clase2 implements IClase;
...
IClase ejemplo2;
...
//estas dos asignaciones son VÁLIDAS
ejemplo2 = new Clase1();
ejemplo2.metodoB(); //de Clase1
ejemplo2 = new Clase2();
ejemplo2.metodoB(); //de Clase2
```

Como trabajamos con Java, nuestro DOBLE "extenderá" o "implementará" la clase que contiene nuestro SUT.
Si nuestro doble es un stub, simplemente tendrá que CONTROLAR las entradas indirectas al SUT

La implementación del doble debe ser lo más genérica posible (debemos implementar solamente un doble para cada tabla de casos de prueba), y lo más simple posible.
En la siguiente sesión usaremos una herramienta para generar el doble de forma automática



IMPLEMENTAMOS EL DOBLE



Ejemplo

Vamos a mostrar una posible implementación de un STUB, con el que podremos controlar lo que devuelve nuestro DOC (entrada indirecta de nuestro SUT)

en src/test/java

```
public class Buscador {          en src/main/java
    //código REAL de nuestro DOC
    //en src/main/java
    public int elemPendientes(Cliente cli) {
        ...
    }
}
```

IMPLEMENTACIÓN REAL

DURANTE las pruebas, nuestro SUT invocará al doble (en src/test/java):

BuscadorStub.elemPendientes()
en lugar de invocar al código real de nuestra dependencia externa (en src/main/java):

Buscador.elemPendientes(),
pero el código de nuestro SUT será idéntico en ambos casos!!!

```
public class BuscadorStub extends Buscador {
    int result;
    public void setResult(int salida) {
        this.result = salida;
    }
    //código de nuestro doble
    //en src/test/java
    @Override
    public int elemPendientes(Cliente cli) {
        return result;
    }
}
```

DOBLE (STUB)

Injectaremos el doble (stub)
durante las pruebas

```
public Factura generarFactura(Cliente cli, Buscador buscar)
    throws FacturaException {
    Factura factura;
    int numElems = buscar.elemPendientes(cli);
    if (numElems>0) {
        //código para generar la factura
        factura = ...;
    } else {
        throw new FacturaException("No hay ...");
    }
    return factura;
}
```

SUT



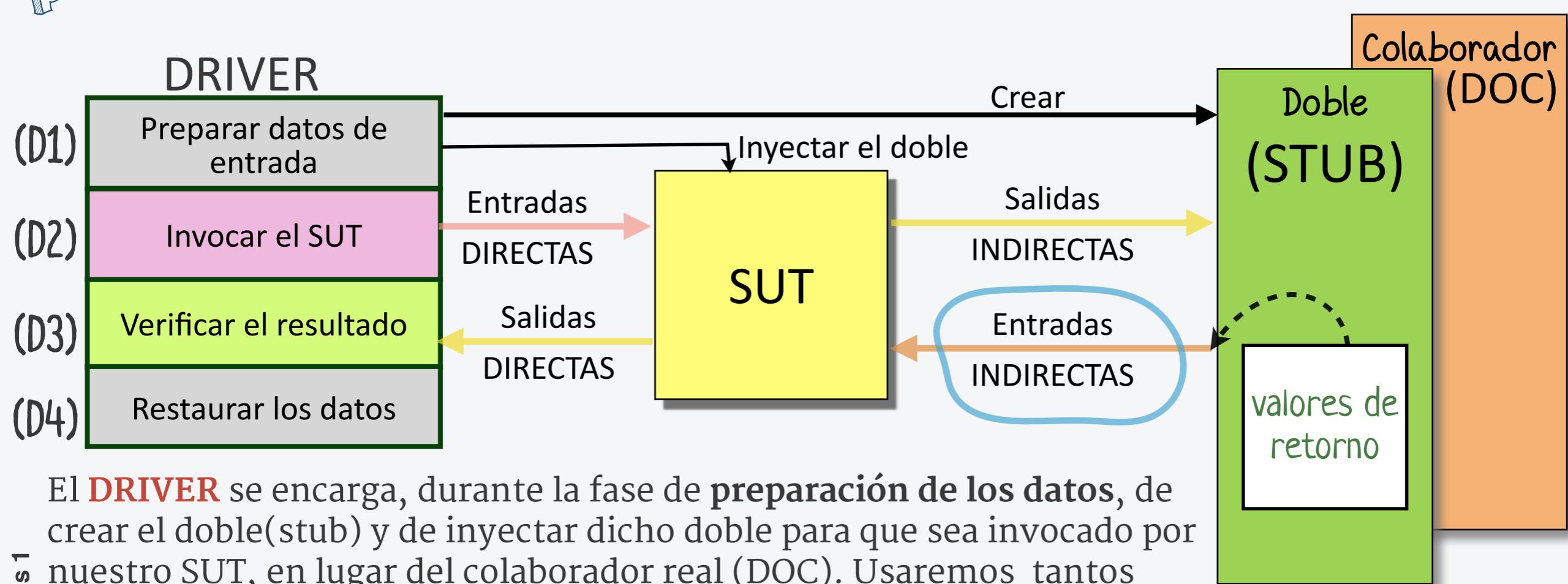
La implementación del STUB
debe ser GENÉRICA. Debe servir para todos los casos de prueba de nuestro SUT (sólo un doble por tabla) !!!

IMPLEMENTACIÓN DEL DRIVER



Usamos el algoritmo que ya conocemos

Recordemos que un STUB es un objeto que reemplaza al componente real (DOC) del cual depende el código del SUT, para que éste pueda controlar las entradas indirectas provenientes de dicha dependencia (valores de retorno, actualización de parámetros o excepciones lanzadas)



El **DRIVER** se encarga, durante la fase de **preparación de los datos**, de crear el doble(stub) y de inyectar dicho doble para que sea invocado por nuestro SUT, en lugar del colaborador real (DOC). Usaremos tantos stubs como dependencias externas necesitemos controlar

Cuando utilizamos un stub, la verificación del resultado de las pruebas: (pass, fail, o error) se realiza sobre la clase a probar (SUT). Verificamos que el estado resultante de nuestro SUT es el esperado (**Verificación basada en el estado**)



IMPLEMENTACIÓN DEL DRIVER



Ejemplo

- Nuestro driver se encargará de crear el STUB e inyectarlo en nuestro SUT antes de invocarlo con los datos de entrada diseñados.
- Un driver para pruebas de integración tendrá una implementación diferente.

Test unitario: aislamos la unidad a probar

```
public class GestorPedidosTest {
    @Test
    public void testGenerarFactura()
        throws Exception {
        Cliente cli = new Cliente(...);
        GestorPedidos sut = new GestorPedidos();
        Buscador buscarStub = new BuscadorStub();
        buscarStub.setResult = 10;
        Factura expectedResult = new Factura(...);
        Factura realResult =
            sut.generarFactura(cli,buscarStub);
        assertEquals(expectedResult, realResult);
    }
}
```

Aquí controlamos las entradas indirectas!

Test de integración: incluye varias unidades

```
public class GestorPedidosIT {
    @Test
    public void testGenerarFactura()
        throws Exception {
        Cliente cli = new Cliente(...);
        GestorPedidos sut = new GestorPedidos();
        Buscador buscar = new Buscador();
        Factura expectedResult = new Factura(...);
        Factura realResult =
            sut.generarFactura(cli,buscar);
        assertEquals(expectedResult, realResult);
    }
}
```

NO tenemos control sobre las entradas indirectas!

```
public Factura generarFactura(Cliente cli, Buscador buscar)
    throws FacturaException {
    Factura factura;
    int numElems = buscar.elemPendientes(cli);
    if (numElems>0) {
        //código para generar la factura
        factura = ...;
    } else {
        throw new FacturaException("No hay ...");
    }
    return factura;
}
```

Los dos tests ejecutan el MISMO CÓDIGO!!!

SUT



Los drivers de pruebas UNITARIAS son DIFERENTES de los drivers de pruebas de integración (y del resto de niveles) !!!



EJEMPLO 2

Sí nuestra SUT es testable NO es necesario refactorizar!!!

- Si nuestra dependencia externa implementa una interfaz, nuestro doble también lo hará. El código de nuestro driver dependerá de si refactorizamos o no y del tipo de refactorización que hagamos

```
public class GestorPedidos { /src/main/java

    public Factura generarFactura(Cliente cli)
        throws FacturaException {
        Factura factura = new Factura();
        IService buscarDatos = new Buscador();
        int numElems = buscarDatos.elemPendientes(cli); DOC
        if (numElems>0) {
            //código para generar la factura
            factura = ...;
        } else {
            throw new FacturaException("No hay nada
                pendiente de facturar");
        }
        return factura;
    }
}
```

SUT NO TESTABLE!!!

```
public class Buscador implements IService {
    @Override
    public int elemPendientes(Cliente cli)
        {...}
    ...
}
```

```
public interface IService { /src/main/java

    public int elemPendientes(Cliente cli);
}
```

- Tenemos que refactorizar nuestra unidad para poder injectar nuestro doble a la hora de hacer las pruebas sin cambiar el código de nuestra SUT. Podemos usar cualquiera de las opciones indicadas. En este caso, elegiremos la opción (4) usando un método de **factoría LOCAL**
 - El método de factoría local será una nueva dependencia externa, pero que pertenece a la misma clase que nuestro SUT. En este caso, necesitaremos implementar una **clase adicional** en **/src/test/java** para poder injectar nuestro doble
 - Los **dobles** y los **drivers** siempre se implementarán en **/src/test/java**



EJEMPLO 2

Refactorizamos nuestra SUT con la opción (4) usando una factoría local

```

public class GestorPedidos {
    public IService getBuscador() {
        IService buscar = new Buscador();
        return buscar;
    }

    public Factura generarFactura(Cliente cli)
            throws FacturaException {
        Factura factura = new Factura();
        IService buscarDatos = getBuscador();

        int numElems = buscarDatos.elemPendientes(cli);
        if (numElems>0) { //código para generar la factura
            factura = ...;
        } else {
            throw new FacturaException("No hay ...");
        }
        return factura;
    }
}

```

/src/main/java **SUT REFACTORIZADA!!!**

Implementamos el DRIVER

```

public class GestorPedidosTest {
    @Test
    public void testGenerarFactura() throws ... {
        Cliente cli = new Cliente(...);
        BuscadorSTUB stub = new BuscadorSTUB(10);
        GestorPedidosTestable sut = new
                GestorPedidosTestable();
        sut.setBuscador(stub);
        Factura expectedResult = new Factura(...);
        Factura realResult = sut.generarFactura(cli);
        assertEquals(expectedResult, realResult);
    }
}

```

/src/test/java **DRIVER**

Implementamos el DOBLE (STUB)

```

public class BuscadorSTUB implements IService {
    int resultado;

    public BuscadorSTUB(int salida) {
        this.resultado = salida;
    }

    @Override
    public int elemPendientes(Cliente cli) {
        return resultado;
    }
}

```

/src/test/java **DOBLE (STUB)**

Necesitamos la clase **GestorPedidosTestable** para injectar el stub durante las pruebas

```

public class GestorPedidosTestable extends
        GestorPedidos {
    IService busca;

    @Override
    public IService getBuscador() {
        return busca;
    }

    public void setBuscador(IService b) {
        this.busca = b;
    }
}

```

/src/test/java **SUT TESTABLE**

P

EJEMPLO 3

Refactorizamos nuestra SUT con la opción (4) usando una clase factoría

```
public class GestorPedidos { /src/main/java

    public Factura generarFactura(Cliente cli)
        throws FacturaException {
        Factura factura = new Factura();
        Buscador buscarDatos = new Buscador();

        int numElems = buscarDatos.elemPendientes(cli); DOC
        if (numElems>0) {
            //código para generar la factura
            factura = ...;
        } else {
            throw new FacturaException("No hay nada
                pendiente de facturar");
        }
        return factura;
    }
}
```

SUT NO TESTABLE!!!

```
public class GestorPedidos { /src/main/java

    public Factura generarFactura(Cliente cli)
        throws FacturaException {
        Factura factura = new Factura();
        Buscador buscarDatos = Factoria.create();

        int numElems =
            buscarDatos.elemPendientes(cli);
        if (numElems>0) {
            //código para generar la factura
            factura = ...;
        } else {
            throw new FacturaException("No hay ...");
        }
        return factura;
    }
}
```

SUT TESTABLE!!!

Implementamos el doble...

```
public class BuscadorStub extends Buscador {
    int result;

    public void setResult(int salida) {
        this.result = salida;
    }
    //código de nuestro doble
    //en src/test/java
    @Override
    public int elemPendientes(Cliente cli) {
        return result;
    }
}
```

/src/test/java STUB

Implementamos la clase factoría...

```
public class ServiceFactory {
    private static Buscador servicio= null;
    public static Buscador create() {
        if (servicio != null) {
            return servicio;
        } else {
            return new Buscador();
        }
    }
    static void setServicio (Buscador srv){
        servicio = srv;
    }
}
```

/src/main/java

EJEMPLO 3

Implementamos el driver (los drivers unitarios y de integración son DIFERENTES)

```
public class GestorPedidosTest {  
    @Test  
    public void testGenerarFactura() throws  
Exception {  
    Cliente cli = new Cliente(...);  
    GestorPedidos sut = new GestorPedidos();  
    Buscador buscadorStub = new BuscadorStub();  
    buscadorStub.setResult() = 10;  
ServiceFactory.setServicio(buscadorStub);  
    Factura expResult = new Factura(...);  
    Factura result = sut.generarFactura(cli);  
    assertEquals(expResult, result);  
}  
}
```

Test unitario

Ejecutamos nuestro SUT controlando su dependencia externa

Aquí inyectamos el stub.

El método assertEquals devuelve true si las dos variables referencian al mismo objeto. Si queremos comprobar si sus contenidos son iguales podríamos p.ej. redefinir su método equals()

Test de integración

```
public class GestorPedidosIT {  
    @Test  
    public void testGenerarFactura() throws  
Exception {  
    Cliente cli = new Cliente(...);  
    GestorPedidos sut = new GestorPedidos();  
    Factura expResult = new Factura(...);  
    Factura result = sut.generarFactura(cli);  
    assertEquals(expResult, result);  
}  
}
```



Por simplicidad hemos omitido los valores concretos de todos los atributos de Cliente y Factura, pero recuerda que en cualquier caso de prueba, TODOS los datos (de E/S) son CONCRETOS!!!

Ejecutamos nuestro SUT sin ningún control de su dependencia externa

EJERCICIO

Recuerda que primero tienes que identificar el SUT y sus dependencias externas!!!

- Supongamos que tenemos una clase, **OrderProcessor**, que procesa pedidos. Queremos implementar una prueba unitaria del método **process()**, que calcula y aplica un descuento sobre un pedido (instancia de la clase Order). El descuento se obtiene consultando el servicio **PricingService.getDiscountPercentage()**.

```
1. public class OrderProcessor {  
2.     private PricingService pricingService;  
  
4.     public void setPricingService(PricingService service) {  
5.         this.pricingService = service;  
6.     }  
  
8.     public void process(Order order) {  
9.         float discountPercentage = pricingService.getDiscountPercentage(order.getCustomer(),  
10.            order.getProduct());  
11.         DOC  
12.         float discountedPrice = order.getProduct().getPrice()  
13.             * (1 - (discountPercentage / 100));  
14.         order.setBalance(discountedPrice);  
15.     }  
16. }  
17.
```

SUT

P DEFINICIÓN DE CLASES UTILIZADAS POR NUESTRO SUT

- Mostramos la definición de las clases utilizadas por OrderProcessor: PricingService, Customer, Product y Order:

Las dependencias que consistan en getters/setters no las sustivimos por dobles

```
public class Order {  
    private Customer customer;  
    private Product product;  
    private float balance;  
    public Order(Customer c, Product p) {  
        customer = c; product = p;  
        balance = p.getPrice();  
    }  
    //getters y setters  
    ...  
}
```

```
public class PricingService {  
    public float getDiscountPercentage  
        (Customer c, Product p) {  
        //calcula el porcentaje de descuento  
        return ...;  
    }  
}
```

Sólo necesitamos controlar el resultado de PricingService!!

```
public class Customer {  
    private String name;  
  
    public Customer(String name) {  
        this.name = name;  
    }  
    //getters y setters  
    ...  
}
```

```
public class Product {  
    private float price;  
  
    private String name;  
  
    public Product(String name,  
                  float price) {  
        this.name = name;  
        this.price = price;  
    }  
    //getters y setters  
    ...  
}
```

NO IMPLEMENTAMOS DOBLES PARA ESTAS DEPENDENCIAS!!



¿CUÁL SERÍA LA IMPLEMENTACIÓN DEL STUB?



- Recuerda que tendremos que sustituir la implementación real de la dependencia externa por una implementación ficticia (stub) durante la ejecución de los tests unitarios, y que el código del SUT que se ejecutará durante las pruebas será IDÉNTICO al que se ejecutará en producción

```
public class PricingService {  
    public float getDiscountPercentage (Customer c, Product p) {  
        //calcula el porcentaje de descuento  
        return ...;  
    }  
}
```

código real, utilizado en el SUT

en src/main

```
public class PricingServiceStub extends PricingService {  
    private float discount;  
  
    public PricingServiceStub(float discount) {  
        this.discount = discount;  
    }  
}
```

código utilizado durante las pruebas

en src/test

```
@Override  
public float getDiscountPercentage(Customer c, Product p) {  
    return discount;  
}  
}
```

STUB

TEST UNITARIO

S S

Ejecutamos el código de nuestro DOBLE durante las pruebas!!!

- Implementación de un test unitario que realiza una verificación basada en el estado del siguiente caso de prueba:

DATOS DE ENTRADA				RESULTADO ESPERADO
Order	Customer	Product	% descuento	Order
o.customer = cus o.product = pro o.balance = 30.0	cus.name = "Pedro Gomez"	pro.name="TDD in Action" pro.price = 30.0	10 %	o.customer = cus o.product = pro o.balance = 27.0

NOTA:

"cus" es el objeto Customer, cuyos atributos son los especificados en la columna Customer

"pro" es el objeto Product, cuyos atributos son los especificados en la columna Product

```
public class OrderProcessorTest {  
  
    @Test  
    public void test_processOrder() {  
        float listPrice = 30.0f;  
        float discount = 10.0f;  
        float expectedBalance = 27.0f;  
        Customer customer = new Customer("Pedro Gomez");  
        Product product = new Product("TDD in Action", listPrice);  
        OrderProcessor processor = new OrderProcessor();  
        processor.setPricingService(new PricingServiceStub(discount));  
  
        Order order = new Order(customer, product);  
        processor.process(order);  
  
        assertAll -> ("Error en pedido",  
            () -> assertEquals(expectedBalance, order.getBalance(), 0.001f),  
            () -> assertEquals("Pedro Gomez", order.getCustomer().getName()),  
            () -> assertEquals("TDD in Action", order.getProduct().getName()),  
            () -> assertEquals(listPrice, order.getProduct().getPrice(), 0.001f));  
    }  
}
```

TEST DE INTEGRACIÓN

Ejecutamos el código REAL de nuestro DOC durante las pruebas!!!

- Implementación de un test de integración que realiza una verificación basada en el estado del test anterior:

```
public class OrderProcessorIT {  
  
    @Test  
    public void test_processOrder() {  
        float listPrice = 30.0f;  
        float expectedBalance = 27.0f;  
        Customer customer = new Customer("Pedro Gomez");  
        Product product = new Product("TDD in Action", listPrice);  
        OrderProcessor processor = new OrderProcessor();  
  
        Order order = new Order(customer, product);  
        processor.process(order); ← Ejecutamos nuestro SUT.  
        assertAll -> ("Error en pedido",  
                      () -> assertEquals(expectedBalance, order.getBalance(), 0.001f),  
                      () -> assertEquals("Pedro Gomez", order.getCustomer().getName()),  
                      () -> assertEquals("TDD in Action", order.getProduct().getName()),  
                      () -> assertEquals(listPrice, order.getProduct().getPrice(), 0.001f));  
    }  
}
```

Ejecutamos nuestro SUT.
No tenemos ningún control sobre su dependencia externa

Y AHORA VAMOS AL LABORATORIO...

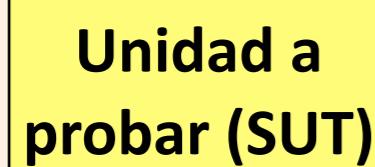
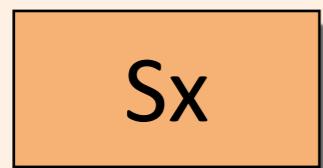
P

Vamos a implementar tests unitarios utilizando STUBS y verificación basada en el ESTADO

P

src/main/java

Entorno de producción



Dependencias externas
(DOCs)

el reemplazo de los
colaboradores por los dobles NO
tiene que afectar al código a
probar (código testable)
Esto es posible si SUT tiene un
seam para cada dependencia
externa

Casos de prueba

Datos Entrada	Resultado Esperado
d1=... d2=... ...	r1
..	
d1=... d2=... ...	rM

src/test/java

Tests Unitarios



driver

Informe

Unidad a probar (SUT)

stub1

stub2

stub3

Dobles

reemplazamos las dependencias externas durante las pruebas para
controlar su interacción con el SUT (entradas indirectas) y así poder
aislar la ejecución de cada unidad del resto del sistema

PREFERENCIAS BIBLIOGRÁFICAS

- P
 - The art of unit testing: with examples in .NET. Roy Osherove. Manning, 2009.
 - Capítulo 3. Using stubs to break dependencies.
 - * <http://www.manning.com/osherove/SampleChapter3.pdf>
 - xUnit Test Patterns. Refactoring test code. Gerard Meszaros
 - * Using test doubles: <http://xunitpatterns.com/Using%20Test%20Doubles.html>
 - Testing with JUnit. Frank Appel. Packt Publishing, 2015.
 - Capítulo 3. Developing independently testable units

Sesión S05: Dependencias externas (2)



(Pruebas unitarias)

Uso de **frameworks** para implementar dobles: EasyMock

EasyMock y **verificación basada en el estado**

Usaremos el API EasyMock para:

- Creación de los stubs
- Programación de expectativas
- Comunicar al framework que los stubs están listos para ser invocados EasyMock y **verificación basada en el comportamiento**

EasyMock y **verificación basada en el comportamiento**

Usaremos el API EasyMock para:

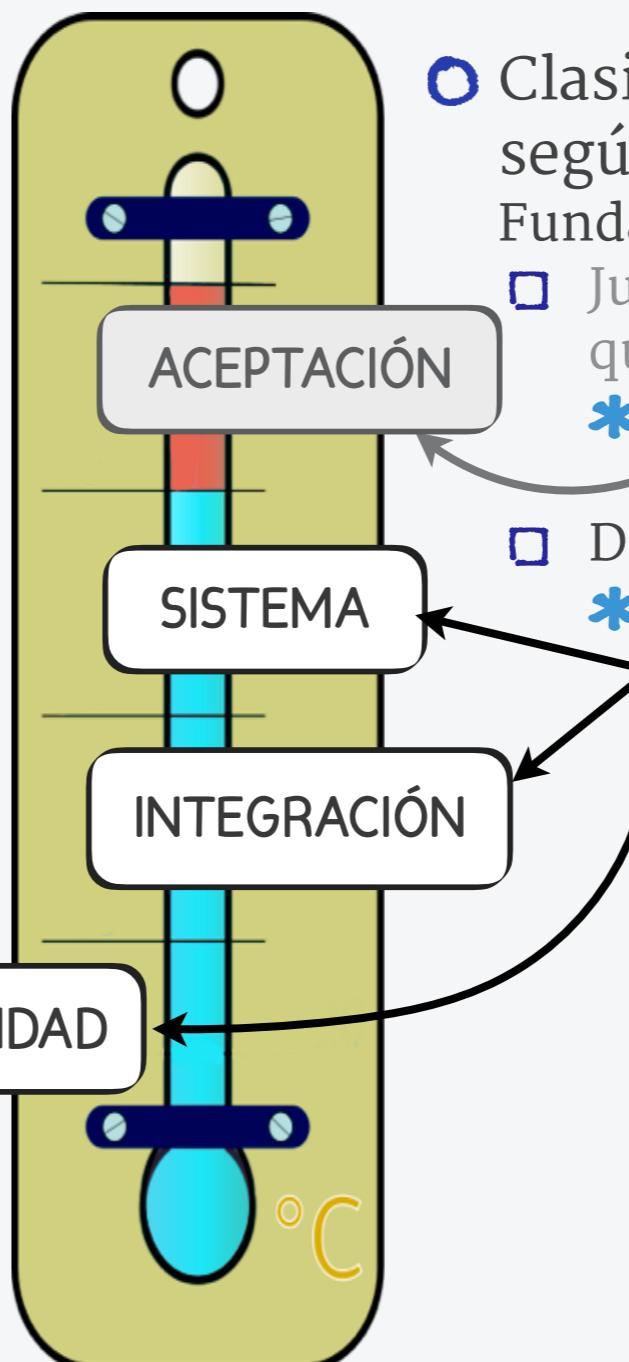
- Creación de los mocks
- Programación de expectativas
- Comunicar al framework que los mocks están listos para ser invocados
- Verificar las expectativas de los mocks

Vamos al laboratorio...

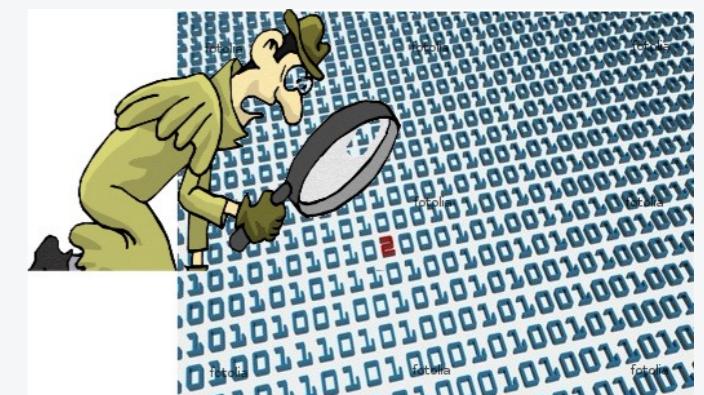
VERIFICACIÓN Y PRUEBAS UNITARIAS

P

P



- Clasificación de las pruebas según su OBJETIVO.
Fundamentalmente probamos PARA:
 - Juzgar la calidad del software o en qué grado es aceptable
* Este proceso se denomina VALIDACIÓN: ...
 - Detectar problemas
* Este proceso se denomina VERIFICACIÓN: se trata de buscar defectos en el programa que provocarán que éste no funcione correctamente (según lo esperado, de acuerdo con los requerimientos especificados previamente)



Nuestro objetivo es DEMOSTRAR la presencia de DEFECTOS en el código de las UNIDADES. Lo haremos de forma dinámica, y podemos implementar los drivers de varias formas para verificarlo!!!

YA HEMOS VISTO CUÁL ES EL PROCESO PARA AUTOMATIZAR PRUEBAS UNITARIAS

P

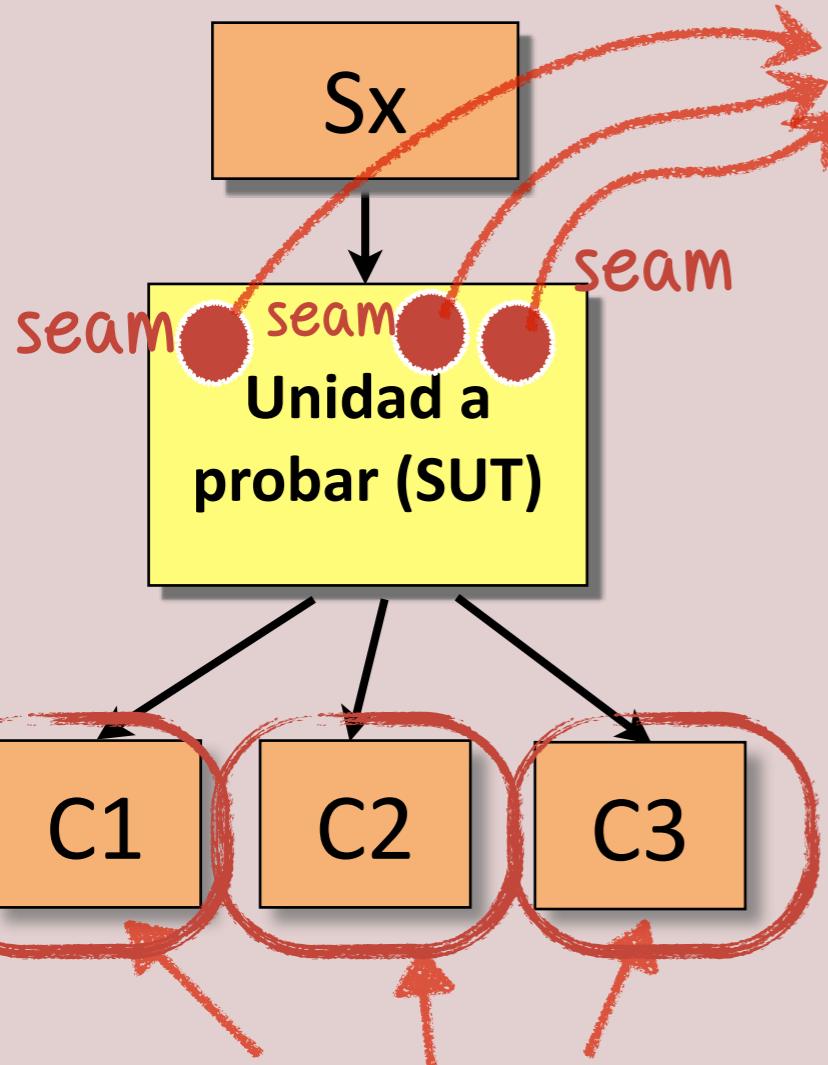
P

P

(1) Identificar las dependencias externas

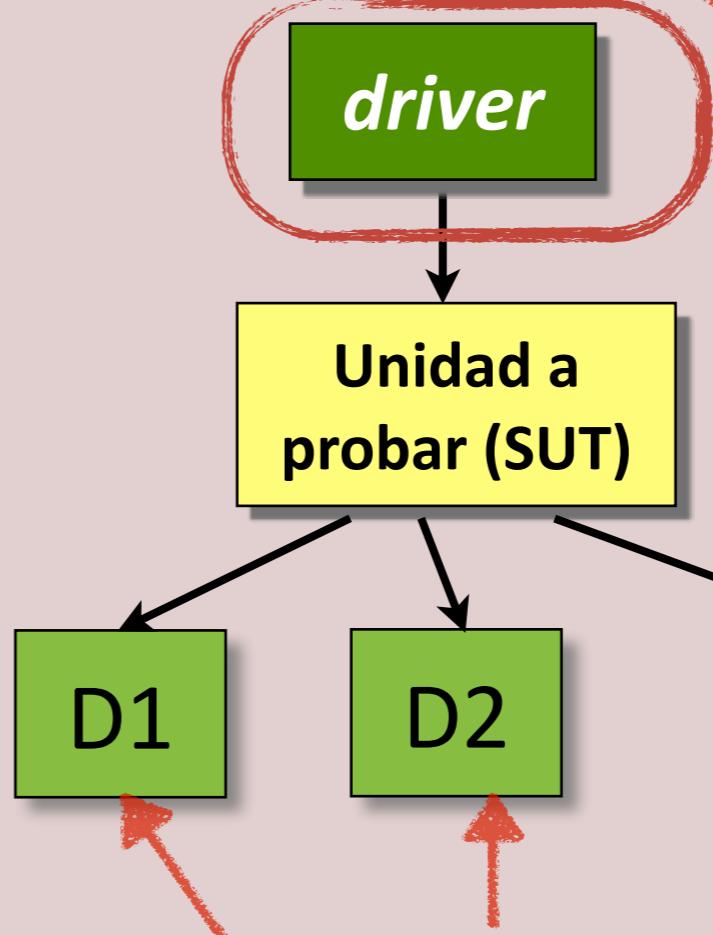


El proceso siempre es el MISMO, aunque del tipo de verificación que use el driver sea diferente



(2) Conseguir que nuestro código sea testable

(4) Implementar los drivers



Verificación basada en el estado (stub)

Verificación basada en el comportamiento (mock)

La implementación de los dobles la hemos hecho "manualmente"

(3) Proporcionar una implementación ficticia (doble: Di) para cada colaborador (Ci). Implementaremos mocks o stubs dependiendo del tipo de verificación que realice el driver

FRAMEWORKS PARA IMPLEMENTAR DOBLES



Implementar dobles!!
(paso 3)

Hemos visto como implementar dobles (stubs) de forma "manual"

P

- Podemos utilizar algún **framework** para evitarnos la implementación "manual" de los dobles de nuestras pruebas. **Recuerda que para automatizar las pruebas, escribimos código adicional que a su vez puede contener errores!**
 - En general, cualquier framework nos **generará** una implementación configurable de los dobles "on the fly", generando el bytecode necesario.
 - Primero **configuraremos** los dobles para controlar las entradas indirectas a nuestro SUT (stubs), o para registrar o verificar las salidas indirectas de nuestro SUT (spies y mocks).
 - **Inyectaremos** los dobles en la unidad a probar antes de invocarla.
- Los frameworks tienen defensores y detractores
 - Los frameworks suelen "tergiversar" la terminología que hemos visto sobre dobles (es importante tener esto en cuenta)
 - La verificación basada en el **comportamiento** (verificación de salidas indirectas) genera un riesgo de que los tests tengan un alto nivel de acoplamiento con los detalles de implementación. Un framework contribuye a crear tests **potencialmente frágiles y difíciles de mantener**
- Algunos ejemplos de frameworks:
 - **EasyMock**, Mockito, JMockit, PowerMock

EASYMOCK Y TIPOS DE DOBLES

EasyMock crea de forma automática la clase que contiene el doble de nuestra dependencia externa

Usaremos el siguiente método estático para crear STUBS

`EasyMock.niceMock(Clase.class)` → devuelve un doble para la clase o interfaz "clase.class"

El **orden** en el que se realizan las invocaciones a los métodos del doble NO se chequean

Por defecto se permiten las invocaciones a todos los métodos del objeto. Si no hemos programado las expectativas de algún método, éste devolverá valores "vacíos" adecuados (0, null o false)

Todas las llamadas esperadas a métodos realizadas por el doble deben realizarse con los argumentos especificados

Para usar el doble (STUB), primero tendremos que generar la clase que contendrá a dicho doble. EasyMock crea un doble para cada uno de los métodos de dicha clase.

Después "programaremos" el comportamiento de sus métodos. La clase creada por EasyMock verifica los argumentos con los que invocamos a nuestro doble, así como el número de invocaciones

Usamos stubs para realizar una verificación basada en el estado, por lo tanto, no estamos interesados en verificar el orden de las invocaciones de nuestro SUT al doble. Sólo queremos controlar las entradas indirectas a nuestro SUT

EasyMock implementa un doble para cada método de la clase. Por defecto cada método devolverá 0, null o false. Esto podemos cambiarlo programando las expectativas para ese método. Están permitidas, por tanto, invocaciones al objeto NiceMock no programadas (en cuyo caso se devolverá el valor por defecto correspondiente).

El objeto NiceMock verifica que el SUT invoca al doble usando los parámetros especificados. Si no queremos que se tenga en cuenta deberemos usar los métodos `anyObject()`, `anyInt()`, `any...` que corresponda



EasyMock llama doble a la clase generada automáticamente. Recuerda que hemos definido una unidad como un MÉTODO.

Por lo que nuestro doble siempre será un método de la clase generada por EasyMock

PEASYSMOCK: IMPLEMENTACIÓN DE STUBS

P <http://jblewitt.com/blog/?p=316>

1. Creamos el stub
2. Programamos las expectativas del stub (determinamos cuál será el valor de la entrada indirecta al SUT)
3. Indicamos al framework que el stub ya está listo para ser invocado por nuestro SUT



SETACIÓN DE STUBS

Usaremos verificación basada en el estado

- Podemos crear un stub a partir de una clase o de una interfaz

```
import org.easymock.EasyMock; Clase que contiene la dependencia externa  
...  
Dependencia1 dep1 = EasyMock.niceMock(Dependencia1.class);  
//dep1 no chequea el orden de invocaciones  
//se permiten invocaciones no programadas, en ese caso se  
//devolverán los valores por defecto 0, null o falso
```

- Para programar las expectativas usaremos el método estático **EasyMock.expect()**

Queremos que nuestro stub pueda ser invocado desde nuestra SUT, pero no necesitamos CHEQUEAR cuántas veces se invoca al doble, cuándo se invoca o si es invocado o no

```
//metodo1() devolverá 9 si es invocado por nuestro SUT  
//independientemente de cuándo o cuántas veces sea invocado  
//y con qué valores de parámetros sea invocado  
EasyMock.expect(dep1.metodo1(anyString(),anyInt()))  
    .andStubReturn(9);  
//metodo2() devolverá una excepción cuando se invoque desde SUT  
EasyMock.expect(dep1.metodo2(anyObject()))  
    .andStubThrow(new MyException("message"));  
dep1.metodo3(anyInt());  
EasyMock.expectLastCall.asStub();
```

- Despues de programar las expectativas SIEMPRE tendremos que ACTIVAR el stub usando el método **replay()**

EasyMock.replay(dep1);



Si no cambiamos el estado del doble a "replay" las expectativas NO se tendrán en cuenta, por lo que si se invoca al doble desde nuestro SUT se devolverán los valores por defecto

EASYSOCK: PROGRAMACIÓN DE EXPECTATIVAS

2

Comportamiento de
STUBS

<https://easymock.org/user-guide.html#verification-expectations>

- Un stub puede devolver resultados diferentes dependiendo de los valores de los parámetros de las invocaciones
- O podemos "relajar" los valores de los parámetros, de forma que devolvamos un determinado resultado independientemente de los valores de entrada del stub
 - Usaremos métodos anyObject() anyString(), any... en esos casos

Ejemplo:

//Creamos el doble

Dependencia dep = EasyMock.niceMock(Dependencia.class);

El doble NO es la clase Dependencia!! Nuestro doble será un método de esta clase!!



//programamos las expectativas

//CADA vez que nuestro SUT invoque a servicio4 con un 12, devolverá 25

//independientemente del número de invocaciones

EasyMock.expect(dep.servicio4(12)).andStubReturn(25);

//si nuestro SUT invoca a servicio4 con cualquier otro valor, devolverá 30

//independientemente del número de veces que se invoque

EasyMock.expect(dep.servicio4(EasyMock.not(EasyMock.eq(12)))).andStubReturn(30);

//si nuestro SUT invoca servicio5(8) siempre -> el stub devolverá null todas las veces

//null es el valor por defecto para los Strings

//si nuestra SUT no invoca nunca servicio5(3), el test NO fallará

EasyMock.expect(dep.servicio5(3)).andStubReturn("pepe");

los métodos **not()** y **eq()** admiten como parámetro tanto un tipo primitivo como un objeto

//otros métodos que pueden usarse

and(X first, X second), or(X first, X second) //X puede ser de tipo primitivo o un objeto

lt(X value), leq(X value), geq(X value), gt(X value) //Para X = tipo primitivo

startsWith(String prefix), contains(String substring), endsWith(String suffix)

isNull(), notNull()

EJEMPLO 1 DE DRIVER CON STUBS Y EASYMOCK

P
P
SUT

```
public class GestorPedidos {
    public Factura generarFactura(Cliente cli, Buscador bus) throws FacturaException {
        Factura factura = new Factura();
        int numElems = bus.elemPendientes(cli); ← dependencia externa
        if (numElems>0) {
            //código para generar la factura
            factura = ...;
        } else {
            throw new FacturaException("No hay nada pendiente de facturar");
        }
        return factura;
    }
}
```

SUT TESTABLE!!!

Sesión 2: Dependencias externas

```
public class GestorPedidosTest {
    @Test
    public void testGenerarFactura() throws Exception {
        Cliente cli = new Cliente(...);
        GestorPedidos sut = new GestorPedidos();
```

DRIVER

```
        Buscador stub = EasyMock.niceMock(Buscador.class);
        EasyMock.expect(stub.elemPendientes(anyObject()))
            .andStubReturn(10);
        EasyMock.replay(stub);
```

La "implementación"
del doble la realizamos
"dentro" del test



```
        Factura expResult = new Factura(...);
        Factura result = sut.generarFactura(cli,stub);
        assertEquals(expResult, result);
    }
}
```

Inyección del doble

anyObject() → "cualquier objeto"
anyChar() → "cualquier char"
anyFloat() → "cualquier float"
anyInt() → "cualquier int"
anyString() → "cualquier objeto String" ...

EJEMPLO 2 DE DRIVER CON STUBS Y EASYMOCK

```
public class OrderProcessor {  
    private PricingService pricingService;  
  
    public void setPricingService(PricingService service) { this.pricingService = service; }  
  
    public void process(Order order) {  
        float discountPercentage =  
            pricingService.getDiscountPercentage(order.getCustomer(), order.getProduct());  
        float discountedPrice = order.getProduct().getPrice() * (1 - (discountPercentage / 100));  
        order.setBalance(discountedPrice);  
    }  
}
```



SUT TESTABLE!!!

```
public class OrderProcessorTest {  
    @Test  
    public void test_processOrderStub() {  
        float listPrice = 30.0f; float discount = 10.0f; float expectedBalance = 27.0f;  
        Customer customer = new Customer("Pedro Gomez");  
        Product product = new Product("TDD in Action", listPrice);  
        OrderProcessor sut = new OrderProcessor();  
  
        PricingService stub = EasyMock.niceMock(PricingService.class);  
        EasyMock.expect(stub.getDiscountPercentage(anyObject(), anyObject()))  
            .andStubReturn(discount);  
  
        sut.setPricingService(stub);  
        EasyMock.replay(stub);  
  
        Order order = new Order(customer, product);  
        sut.process(order);  
  
        assertEquals(expectedBalance, order.getBalance(), 0.001f);  
    }  
}
```

DRIVER

Implementación del doble dentro del test



Inyección del doble

EJEMPLO 3 DE DRIVER CON STUBS Y EASYMOCK

SUT TESTABLE!!!

```
public class CoffeeMachine {  
    private Container coffeeC, waterC;  
  
    public CoffeeMachine(Container coffee, Container water) {  
        coffeeC = coffee; waterC = water;  
    }  
    public boolean makeCoffee(CoffeeCupType type) throws NotEnoughException {  
        boolean isEnoughCoffee = coffeeC.getPortion(type);  
        boolean isEnoughWater = waterC.getPortion(type);  
  
        if (isEnoughCoffee && isEnoughWater) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

prototipo del método getPortion()

```
public boolean getPortion(CoffeeCupType coffeeCupType) throws NotEnoughException
```

```
public class CoffeeMachineEasyMockTest {  
    CoffeeMachine coffeeMachine;  
    Container coffeeContainerStub;  
    Container waterContainerStub;  
  
    @BeforeEach  
    public void setup() {  
        coffeeContainerStub = EasyMock.niceMock(Container.class);  
        waterContainerStub = EasyMock.niceMock(Container.class);  
        coffeeMachine = new CoffeeMachine(coffeeContainerStub, waterContainerStub);  
    }  
  
    @Test  
    public void makeCoffee_NotException() {  
        assertDoesNotThrow(() -> EasyMock.expect(coffeeContainerStub.getPortion(EasyMock.anyObject()))  
                           .andStubReturn(true));  
        EasyMock.replay(coffeeContainerStub);  
  
        assertDoesNotThrow(() -> EasyMock.expect(waterContainerStub.getPortion(EasyMock.anyObject()))  
                           .andStubReturn(true));  
        EasyMock.replay(waterContainerStub);  
  
        assertDoesNotThrow( ()->assertTrue(coffeeMachine.makeCoffee(CoffeeCupType.LARGE)));  
    }  
}
```

DRIVER

STUBS

inyectamos los dobles

programamos las expectativas de cada doble y los "activamos"

SUT

OBJETOS STUB VS. OBJETOS MOCK

Stub Object

Es un objeto que actúa como un punto de control para entregar **ENTRADAS INDIRECTAS** al SUT, cuando es invocado desde el SUT.

Un stub utiliza verificación basada en el **ESTADO**

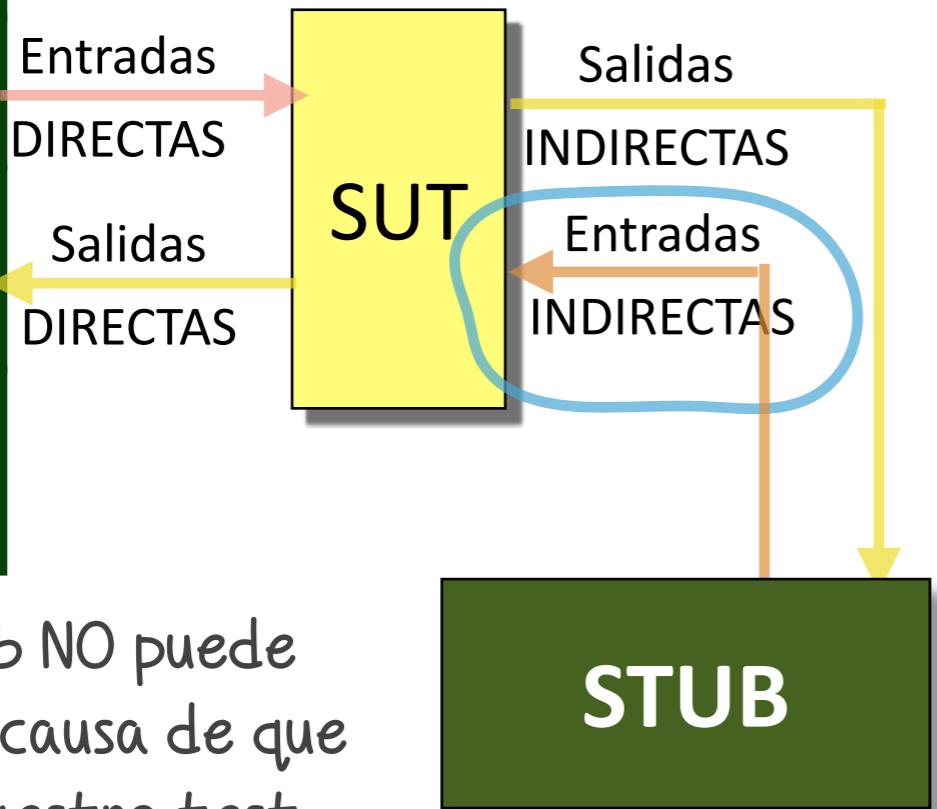
DRIVER

Preparar datos de entrada

Invoker el SUT
Entradas DIRECTAS

Verificar el resultado
Salidas DIRECTAS

Restaurar los datos



El stub NO puede ser la causa de que que nuestro test falle

Mock Object

Es un objeto que actúa como un punto de observación para las **SALIDAS INDIRECTAS** del SUT.

Puede devolver información cuando se le invoca (igual que un stub), o no devolver nada.

Además registra las llamadas recibidas del SUT, y compara las llamadas reales con las llamadas previamente definidas como expectativas, de forma que hacen que el test falle si no se cumplen dichas expectativas.

Un mock utiliza verificación basada en el comportamiento

DRIVER

Preparar datos de entrada

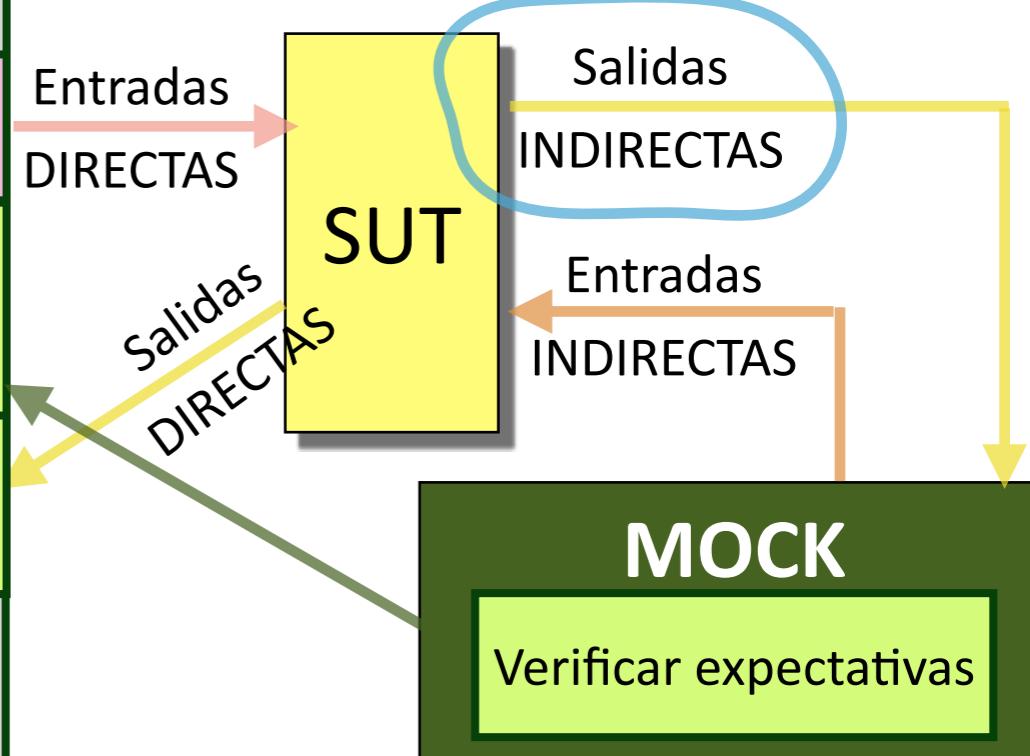
Invoker el SUT
Entradas DIRECTAS

Verificar que se invoca al colaborador
Salidas DIRECTAS

Verificar el resultado

Restaurar los datos

Un mock PUEDE decidir si nuestro test falla o no



VERIFICACIÓN BASADA EN EL ESTADO VS.

P

La implementación de los drivers es diferente si verificamos el ESTADO, o verificamos el COMPORTAMIENTO

Driver con Verificación basada en el estado

SETUP

Preparamos las entradas directas de nuestro SUT, creamos e injectamos los dobles (stubs) (**)

EXERCISE

Invocamos a la unidad a probar

VERIFY STATE

Utilizamos aserciones para comprobar el estado resultante de la invocación de nuestro SUT

TEARDOWN

Restauramos el estado si es necesario

VERIFICACIÓN BASADA EN EL COMPORTAMIENTO

Sesión 5: Dependencias externas 2

En este caso el test puede fallar si la interacción de nuestra SUT con la dependencia externa no es la correcta

Driver con Verificación basada en el comportamiento

SETUP DATA

(**) En este caso los dobles son mocks

SETUP EXPECTATIONS

Programamos las expectativas de cada mock invocado desde nuestro SUT

EXERCISE

Invocamos a la unidad a probar

VERIFY EXPECTATIONS

Verificamos que se han invocado a los métodos correctos, en el orden correcto, con los parámetros correctos

VERIFY STATE

Utilizamos aserciones para comprobar el estado resultante de la invocación de nuestro SUT

TEARDOWN

Restauramos el estado si es necesario

EASYMOCK Y TIPOS DE DOBLES

S EasyMock genera de forma automática diferentes clases para nuestros dobles

Creación de STUBS

P EasyMock.niceMock(Clase.class)

El **orden** en el que se realizan las invocaciones al doble NO se chequean

Por defecto se permiten las invocaciones a todos los métodos del objeto. Si no hemos programado las expectativas de algún método, éste devolverá valores "vacíos" adecuados (0, null o false)

Todas las llamadas esperadas a métodos realizadas por el doble deben realizarse con los argumentos especificados

Creación de MOCKS (si sólo hay 1 invocación del doble)

P EasyMock.mock(Clase.class)

El **orden** en el que se realizan las invocaciones al doble NO se chequean

El comportamiento por defecto para todos los métodos del objeto es lanzar un AssertionError para cualquier invocación no "esperada"

Todas las llamadas esperadas a métodos realizadas por el doble deben realizarse con los argumentos especificados

Creación de MOCKS (con cualquier nº de invocaciones)

P EasyMock.strictMock(Clase.class)

Se comprueba el **orden** en el que se realizan las invocaciones al doble.

Si se invoca a un método no "esperado" se lanza un AssertionError

Todas las llamadas esperadas a métodos realizadas por el doble deben realizarse con los argumentos especificados



A menos que indiquemos lo contrario, si usamos verificación basada en el comportamiento, estamos interesados en comprobar cuándo se invoca a nuestros mocks (orden), cómo (con qué parámetros), y el número de veces que invocan, así como si se invocan o no. Por lo que siempre usaremos el método strictMock() para crear el doble (o el método mock si sólo hay una invocación al mock desde nuestro SUT).

EASYSOCK: IMPLEMENTACIÓN DE MOCKS

Si implementamos mocks usaremos una verificación basada en el COMPORTAMIENTO!!!

El orden de invocaciones a dep1 NO importa!

1. Creamos el mock

2. Programamos las expectativas del mock: determinamos cuál será el valor de las salidas indirectas del SUT, y también las entradas indirectas al SUT

3. Indicamos al framework que el mock ya está listo para ser invocado por nuestro SUT

4. Verificamos las expectativas del mock

- Podemos crear un mock a partir de una clase o de una interfaz

```
import org.easymock.EasyMock;  
...  
//si sólo invocamos a dep 1 vez desde nuestra SUT  
Dependencia1 dep1 = EasyMock.mock(Dependencia1.class);  
//en cualquier otro caso  
Dependencia2 dep2 = EasyMock.strictMock(Dependencia2.class);
```

El orden de invocaciones a dep2 SI importa!

- Programamos las expectativas con el método **EasyMock.expect()**, para indicar cómo se invocará al doble. También podemos indicar el resultado esperado de dicha invocación (si procede)

```
//metodo1() será invocado desde nuestro SUT con los parámetros indicados  
// y devolverá 9  
EasyMock.expect(dep2.metodo1("xxx", 7)).andReturn(9);  
//metodo1() será invocado desde nuestro SUT y devolverá una excepción  
EasyMock.expect(dep2.metodo1("yy", 4)).andThrow(new MyException("message"));  
//metodo2() será invocado desde nuestra SUT.  
//metodo2() es un método que devuelve void  
dep1.metodo2(15);
```

- Después de programar las expectativas SIEMPRE tendremos que ACTIVAR el mock usando el método **replay()**

```
EasyMock.replay(dep1, dep2);
```

3



Si no "activamos" el mock, las expectativas NO tienen ningún efecto!!!

- Después de invocar a nuestra SUT, SIEMPRE debemos verificar que efectivamente nuestra SUT ha invocado a los mocks

```
EasyMock.verify(dep1, dep2);
```

4

EASYSOCK: PROGRAMACIÓN DE EXPECTATIVAS

②

Número de invocaciones

- P O Debemos indicar cómo interaccionará nuestro SUT con el objeto mock que hemos creado (cuántas veces lo invocará, a qué métodos, con qué parámetros, lo que devolverán, el orden en que se invocarán...)
- P Cuando ejecutemos nuestro SUT durante las pruebas, el mock registrará TODAS las interacciones desde el SUT,
- Si es un **StrictMock** y las invocaciones del SUT no coinciden con las expectativas programadas: (**nº de invocaciones, parámetros y orden**), entonces el doble provocará un fallo (AssertionError)
- Si es un **Mock** y las invocaciones del SUT no coinciden con las expectativas programadas: (**nº de invocaciones y parámetros**), entonces el doble provocará un fallo

- O Para especificar las expectativas podemos indicar:

- que se esperan un determinado número de invocaciones:

```
expect(mock.metodoX("parametro")) //invocamos al métodoX(), con el parámetro especificado  
    .andReturn(42).times(3) //devuelve 42 las tres primeras veces  
    .andThrow(new RuntimeException(), 4) //las siguientes 4 llamadas devuelven una excepción  
    .andReturn(-42); //la siguiente llamada devuelve -42 (una única vez)
```

- podemos también "relajar" las expectativas (número e invocaciones esperadas): 
- ```
times(int min, int max); //especifica un número de invocaciones entre min y max
atLeastOnce(); //se espera al menos una invocación
anyTimes(); //nos da igual el número de invocaciones
```

- Las expectativas pueden expresarse de forma "encadenada":

```
expect(mock.operation()).andReturn(true).times(1,5).
 andThrow(new RuntimeException("message"));
```

... en lugar de:

```
expect(mock.operation().andReturn(true).times(1,5));
expectLastCall().andThrow(new RuntimeException("message"));
```



A menos que se indique de forma explícita en el enunciado,  
NO "relajaremos" las expectativas del mock!!

# EASYSOCK: PROGRAMACIÓN DE EXPECTATIVAS

2

Valores de parámetros

- Debemos indicar cuáles serán los valores de los parámetros con los que nuestro SUT invocará al mock durante las pruebas:
  - Tanto si es un Mock como un StrictMock, los valores de los parámetros deben ser los programados
  - En un Mock, el orden de ejecución de las expectativas NO se chequea. En un StrictMock sí.

Con respecto a los valores de los parámetros (o valores de retorno), debemos tener en cuenta que:

- EasyMock, para comparar argumentos de tipo **Object**, utiliza por defecto el método **equals()** de dichos argumentos, por lo tanto si no redefinimos el método equals(), No estaremos comparando los valores de los atributos de dichos objetos.
- Si estamos interesados en que el parámetro de la expectativa sea exactamente la misma instancia, usaremos el método **same()**.

```
User user = new User();
expect(userService.addUser(same(user))).andReturn(true);
replay(userService);
```

- Los Arrays, desde la versión 3.5, son comparados por defecto con **Arrays.equals()**, por lo que estaremos comparando los valores de cada uno de los elementos del array.
- Podemos también "relajar" los valores de los parámetros:

```
anyObject(); //indica que el argumento puede ser cualquier objeto
anyBoolean(); //indica que el argumento puede ser cualquier booleano
anyInt(); //indica que el argumento puede ser cualquier entero
...
isNull(); //Comprueba que se trata de un valor nulo
notNull(); //Comprueba que se trata de un valor no nulo
```



No las usaremos si queremos hacer una verificación de comportamiento "estricta"!!

# EASYSOCK: PROGRAMACIÓN DE EXPECTATIVAS

② Orden de invocación de expectativas

- Con respecto al orden de ejecución de las expectativas de UN MOCK:

- Si usamos un Mock, el orden de las invocaciones a dicho objeto, NO se chequea.
- Si usamos un StrictMock, el orden de las invocaciones a dicho objeto, debe coincidir exactamente con el orden establecido en las expectativas

- Con respecto al orden de ejecución de las expectativas entre VARIOS MOCKS:

- Para poder establecer un orden de invocaciones entre objetos StrictMock, usaremos un **IMocksControl**

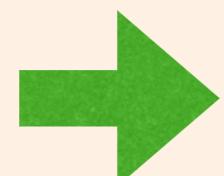
Ejemplo con varios objetos StrictMock.

Sólo se chequea el orden de invocaciones para cada objeto (no se chequea el orden de invocaciones ENTRE ellos)



```
Doc1 mock1 = EasyMock.strictMock(Doc1.class);
Doc2 mock2 = EasyMock.strictMock(Doc2.class);
/* si las expectativas determinan un cierto orden
entre las invocaciones a mock1 y mock2,
si nuestro SUT NO sigue ese orden de invocaciones
el test NO falla */
replay(mock1, mock2);
//invocamos a nuestro SUT
verify(mock1, mock2);
```

Mismo ejemplo, pero en el que se chequea el orden ENTRE los objetos (y también el orden de invocaciones para cada uno de ellos)



```
IMocksControl ctrl = EasyMock.createStrictControl();
Doc1 mock1 = ctrl.createMock(Doc1.class);
Doc2 mock2 = ctrl.createMock(Doc2.class);
//si las expectativas determinan un cierto orden
//entre las invocaciones a mock1 y mock2,
//si nuestro SUT no sigue ese orden de invocaciones
//el test fallará
ctrl.replay(); //no es necesario usar parámetros
//invocamos a nuestro SUT
ctrl.verify();
```

# EJEMPLO DE USO DE MOCKS CON EASYSOCK

Ejemplo extraído de: <http://www.ibm.com/developerworks/java/library/j-easymock/index.html>

- Para utilizar la librería easyMock en un proyecto Maven, tenemos que añadir la siguiente dependencia en el pom del proyecto

```
<dependency>
 <groupId>org.easymock</groupId>
 <artifactId>easymock</artifactId>
 <version>4.2</version>
 <scope>test</scope>
</dependency>
```

El código de nuestros tests DEPENDE de las clases de easymock-4.2.jar!!

Queremos implementar un driver unitario!!

- Queremos realizar una verificación basada en el comportamiento del método **Currency.toEuros()**, cuya implementación es la siguiente:

```
public class Currency {
 private String units;
 private long amount;
 private int cents;

 public Currency(double amount, String code) {
 this.units = code;
 setAmount(amount);
 }

 private void setAmount(double amount) {
 amount = new Double(amount).longValue();
 this.cents = (int) ((amount * 100.0) % 100);
 }

 @Override
 public String toString() {
 ...
 }
}
```

**SUT**

```
public Currency toEuros(ExchangeRate converter) {
 if ("EUR".equals(units)) {return this;}
 else {
 double input = amount + cents/100.0;
 double rate;
 try {
 rate = converter.getRate(units, "EUR");
 double output = input * rate;
 return new Currency(output, "EUR");
 } catch (IOException ex) {
 return null;
 }
 }
}

@Override
public boolean equals(Object o) {
 ...
}
} // class Currency
```

Necesitamos un mock que reemplace a *ExchangeRate.getRate()* durante las pruebas

# P IMPLEMENTACIÓN DEL DRIVER

S Implementamos el doble en el propio driver!!

- Usamos la librería EasyMock para implementar nuestro driver de pruebas unitarias usando verificación basada en el comportamiento. Nuestros dobles serán mocks

```
P public class CurrencyUnitTest {
```

```
@Test
```

```
public void testToEuros() throws IOException {
```

```
Currency testObject = new Currency(2.50, "USD");
```

```
Currency expected = new Currency(3.75, "EUR");
```

```
ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
```

```
EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);
```

```
EasyMock.replay(mock);
```

```
Currency actual = testObject.toEuros(mock);
```

```
EasyMock.verify(mock);
```

```
assertEquals(expected, actual));
```

```
}
```

Ejecutamos el método a probar utilizando el mock

Se verifica que realmente se invoca al mock desde nuestro SUT

Paso 4

Comparamos el resultado real con el esperado

Paso 1

Resultado esperado

Creamos el mock

Indicamos que el mock debe realizar una llamada a `getRate()`, y devolver el valor 1.5

Paso 2

Indicamos que ya estamos listos para ejecutar el mock (el estado del mock cambia de "record mode" a "replay mode"). Es necesario pasar a este estado antes de ejecutar el mock. Cuando se ejecute el mock se comprobará que los parámetros de la invocación sean los correctos y que se llama al método una sola vez

Paso 3

# PARTIAL MOCKING



P

- En ocasiones podemos necesitar proporcionar una implementación ficticia no de toda la clase, sino sólo de algunos métodos (partial mocking).
  - Esto ocurre normalmente cuando estamos probando un método que realiza llamadas a otros métodos de su misma clase
- La librería EasyMock nos permite realizar un mocking parcial de una clase, utilizando el método **partialMockBuilder()**, de la siguiente forma:

```
ToMock mock = partialMockBuilder(ToMock.class)
 .addMockedMethod("mockedMethod").createMock();
```

En este caso, el método añadido con "addMockedMethod()" será "mocked" (sustituidos por su doble), el resto se ejecutarán con su código "original". También se puede usar addMockedMethods(Method... methods)

```
public class Rectangle {
 private int x;
 private int y;

 int convertX() {...}
 int convertY() {...}

 public int getArea() {
 return convertX() *
 convertY();
 }
}
```

No se puede invocar a verify **ANTES** de invocar a nuestro SUT!!!



```
public class RectanglePartialMockingTest {
 private Rectangle rec;

 @Test
 public void testGetArea() {
 rec = partialMockBuilder(Rectangle.class)
 .addMockedMethods("convertX", "convertY")
 .createMock();
 expect(rec.convertX()).andReturn(4);
 expect(rec.convertY()).andReturn(5);
 replay(rec);
 Assertions.assertEquals(20, rec.getArea());
 EasyMock.verify(rec);
 }
}
```

# TEN EN CUENTA QUE ...

P

S

P

- Los objetos mock son diferentes de los stubs, ya que los objetos mock "registran" el comportamiento en lugar de simplemente devolver valores preestablecidos

## Verificación basada en el estado

- Si no usamos un framework, podemos tener que implementar manualmente un elevado número de objetos *stub*, para cada uno de los cuales tenemos que "predefinir" las respuestas que proporciona
- Sin un framework resulta más complejo el preparar todos los stubs, pero los tests son MÁS ROBUSTOS ante cambios en la implementación de nuestro SUT

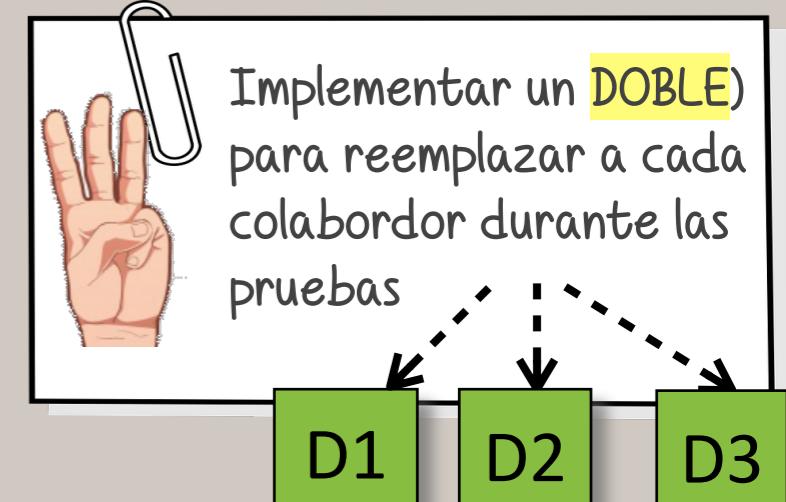
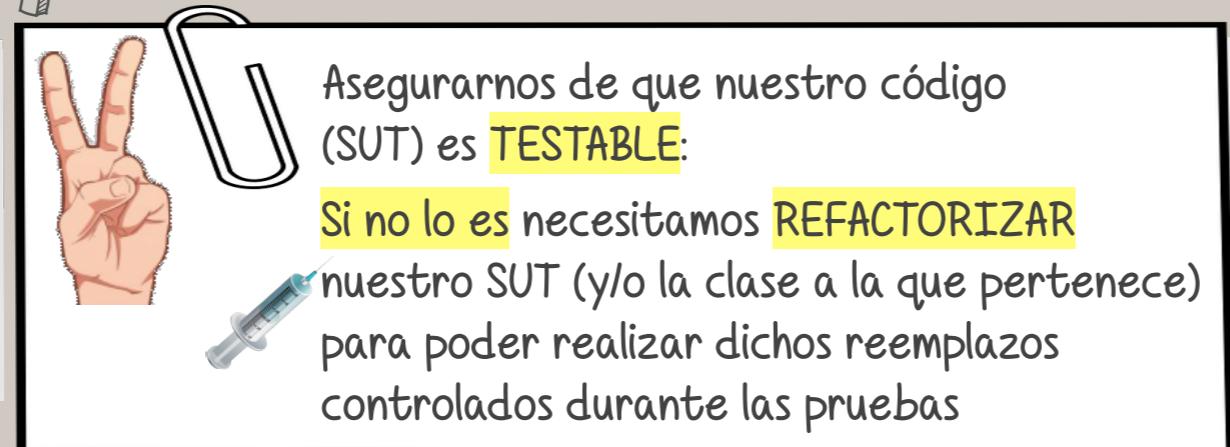
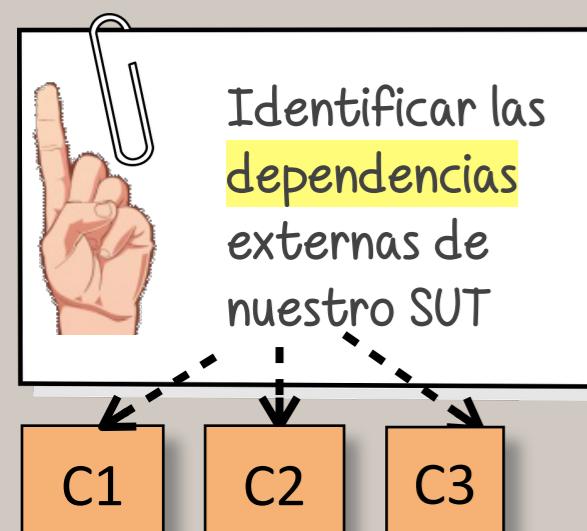
## Verificación basada en el comportamiento

- Usaremos siempre un framework que nos permita implementar rápidamente los mocks, pero si la implementación cambia el orden en el que se llama a los métodos, o los parámetros utilizados, o incluso los métodos a los que se llama, tendremos que cambiar los tests
- Los tests son más "frágiles" y difíciles de mantener



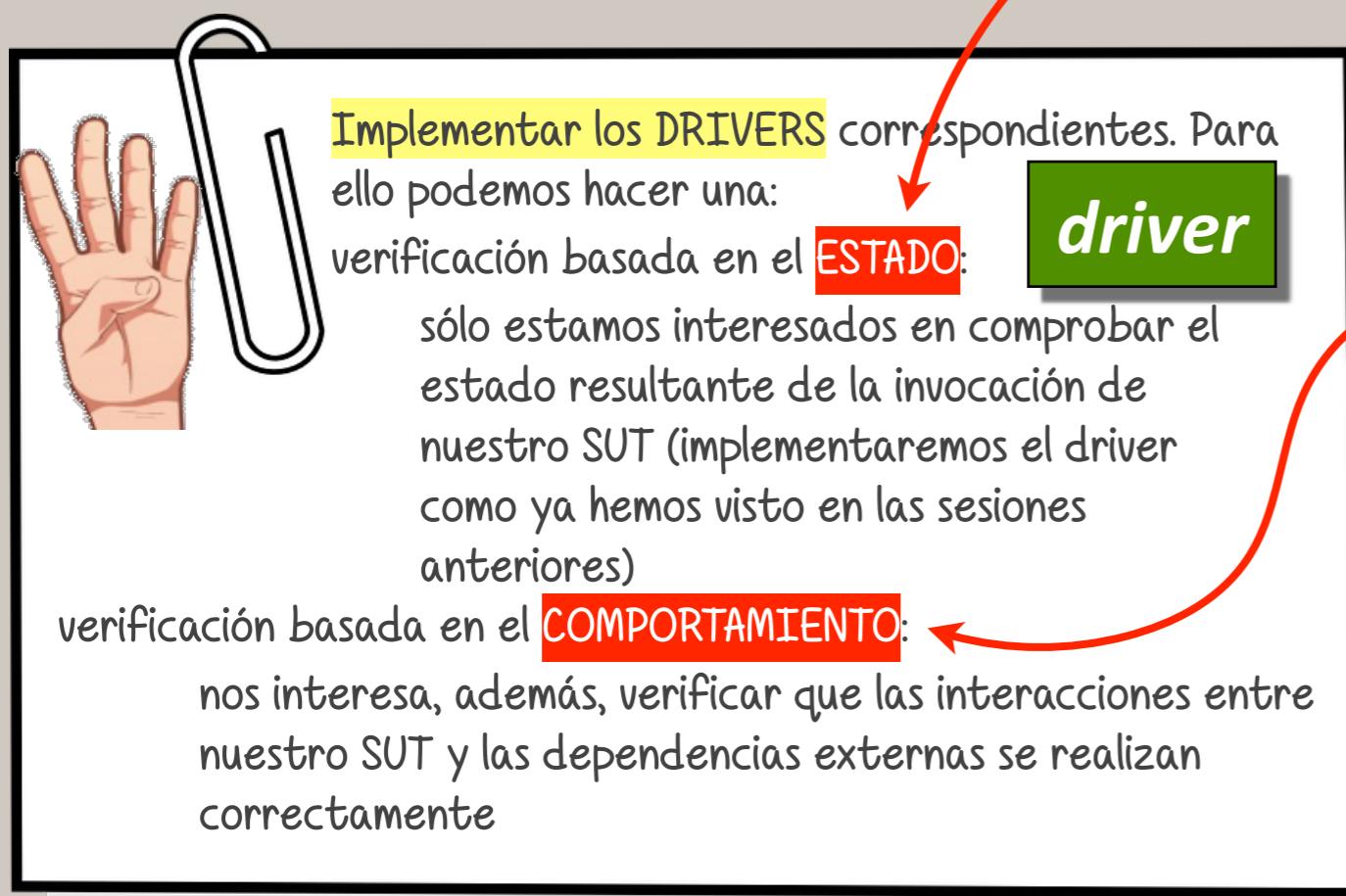
- Si nuestras dependencias externas no proporcionan entradas indirectas al SUT que debamos controlar, nuestro doble no será un stub, usaremos un mock y por tanto tendremos que utilizar verificación basada en el comportamiento si queremos comprobar que el doble ha sido invocado desde el SUT
- Si los colaboradores proporcionan entradas indirectas al SUT, debemos controlar dichas entradas con un stub para realizar pruebas unitarias. Podemos usar, o no, un framework para implementar los stubs
- Si queremos verificar el comportamiento de nuestro SUT, necesariamente usaremos mocks. Podemos no usar un framework, pero lo habitual es usar alguno

# PASOS A SEGUIR PARA AUTOMATIZAR LAS PRUEBAS UNITARIAS...



Colaboradores (DOCs)

Los dobles se pueden implementar manualmente o con EasyMock



**Stub**  
Es un objeto que actúa como un punto de CONTROL para entregar ENTRADAS INDIRECTAS al SUT, cuando se invoca a alguno de los métodos de dicho stub.  
Un stub utiliza verificación basada en el estado

**Mock**  
Es un objeto que actúa como un punto de observación para las SALIDAS INDIRECTAS del SUT.  
Puede devolver información cuando se le invoca (igual que un stub), o no devolver nada.  
Además registra las llamadas recibidas del SUT, y compara las llamadas reales con las llamadas previamente definidas como expectativas, de forma que hacen que el test falle si no se cumplen dichas expectativas.  
Un mock utiliza verificación basada en el comportamiento

# EJERCICIO RESUELTO

- P
- Usa la librería EasyMock para implementar el siguiente caso de prueba utilizando verificación basada en el comportamiento para el método  
GestorLlamadas.calculaConsumo()
- S

```
public interface ServicioHorario {
 public int getHoraActual();
}
```

**SUT**

```
public class GestorLlamadas {
 static double TARIFA_NOCTURNA=10.5;
 static double TARIFA_DIURNA=20.8;
 private ServicioHorario reloj;

 public void setReloj(ServicioHorario reloj) {
 this.reloj = reloj;
 }

 public double calculaConsumo(int minutos) {
 int hora = reloj.getHoraActual();
 if(hora < 8 || hora > 20) {
 return minutos * TARIFA_NOCTURNA;
 } else {
 return minutos * TARIFA_DIURNA;
 }
 }
}

SUT TESTABLE!!!
```



	minutos	hora	Resultado
C1	10	15	208

**DRIVER**

```
public class GestorLlamadasMockTest {
 private ServicioHorario mock;
 private GestorLlamadas gll;

 @Before
 public void incializacion() {
 mock = EasyMock
 .createMock(ServicioHorario.class);
 gll = new GestorLlamadas();
 gll.setReloj(mock);
 }

 @Test
 public void testC1() {
 EasyMock.expect(mock
 .getHoraActual()).andReturn(15);
 EasyMock.replay(mock);
 double result = gll.calculaConsumo(10);
 assertEquals(208, result, 0.001);
 EasyMock.verify(mock);
 }
}
```

# EJERCICIO PROPUESTO

P

- Se proporciona el código con una versión simplificada de la unidad GestorPedidos. generarFactura(). Dada la siguiente tabla de casos de prueba, implementa:
  - (a) Un driver que realice una verificación basada en el estado SIN utilizar Easymock
  - (b) Un driver que realice una verificación basada en el estado utilizando Easymock
  - (c) Un driver que realice una verificación basada en el comportamiento con Easymock

P

```

public class GestorPedidos {
 public Buscador getBuscador() {
 Buscador busca = new Buscador();
 return busca;
 }
 public Factura generarFactura(Cliente cli)
 throws FacturaException {
 Factura factura = new Factura();
 Buscador buscarDatos = getBuscador();

 int numElems = buscarDatos.elemPendientes(cli);
 if (numElems>0) {
 //código para generar la factura
 factura.setIdCliente(cli.getIdCliente());
 float total = cli.getPrecioCliente()*numElems;
 factura.setTotal_factura(total);
 } else {
 throw new FacturaException("No hay nada
 pendiente de facturar");
 }
 return factura;
 }
}

```

SUT

DATOS DE ENTRADA		RESULTADO ESPERADO
Cliente	nº elem	Factura
c.id= "cliente1" o.precio= 20.0€	10	f.idCliente = "cliente1" f.total_factura = 200.0
c.id= "cliente1" o.precio= 20.0€	0	FacturaException con mensaje1 (*)

mensaje1 = "No hay nada pendiente de facturar"

```

public class Cliente {
 private String idCliente;
 private float precioCliente;

 public Cliente(String idCliente, float precioC) {
 this.idCliente = idCliente;
 this.precioCliente = precioC;
 }

 //getters y setters
}

```

SUT TESTABLE!!!

# EJERCICIO PROPUESTO

P

- Mostramos parte de la implementación de las clases utilizadas en producción:

```
public class Factura {
 private String idCliente;
 private float total_factura;

 public Factura() {}

 public Factura(String idCliente) {
 this.idCliente = idCliente;
 this.total_factura = 0.0f;
 }

 //getters y setters
 ...

 @Override
 public boolean equals(Object o) {
 if (this == o) return true;
 if (o == null || getClass() != o.getClass()) return false;

 Factura that = (Factura) o;
 if (idCliente != that.getIdCliente()) { return false; }
 if (total_factura != that.getTotal_factura()) { return false; }
 return true;
 }

 @Override
 public int hashCode() {
 return idCliente != null ? idCliente.hashCode() : 0;
 }
}
```

```
public class FacturaException extends Exception {
 public FacturaException(String mensaje) {
 super(mensaje);
 }
}
```



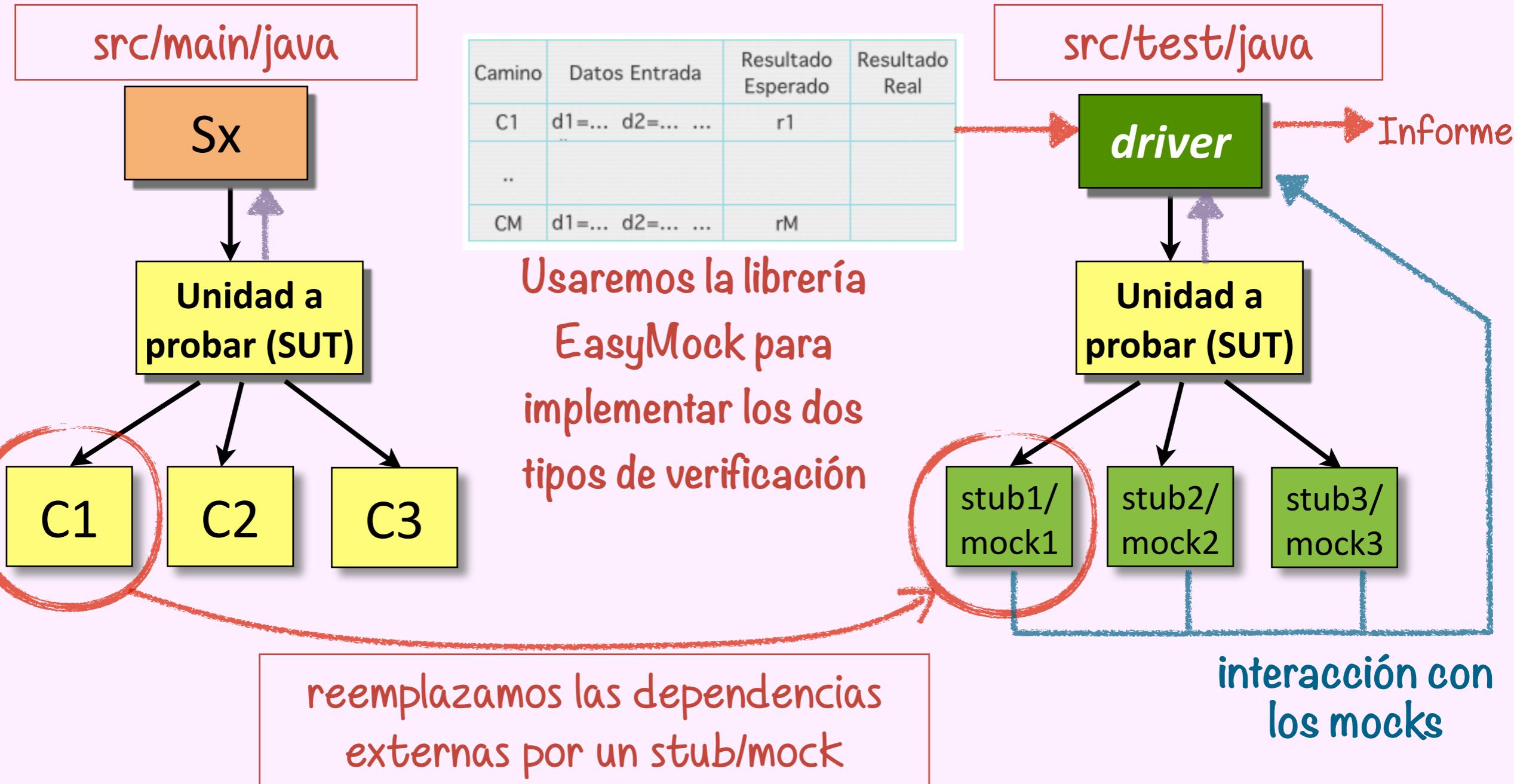
Cuando termines el ejercicio debes tener claro:

- dónde se ubican física y lógicamente cada uno de los ficheros.
- cómo debes configurar el pom del proyecto
- cómo lanzar la ejecución de los tests con maven
- las diferencias entre un stub y un mock
- las diferencias de los drivers cuando verificamos basándonos en el estado y en el comportamiento

Propuesta adicional: realiza las acciones necesarias para poder ejecutar a voluntad sólo los tests de cada uno de los tres apartados del ejercicio

# Y AHORA VAMOS AL LABORATORIO...

Vamos a implementar tests unitarios utilizando MOCKS y verificación basada en el COMPORTAMIENTO y/o STUBS con verificación basada en el estado



P

# REFERENCIAS BIBLIOGRÁFICAS

S

S

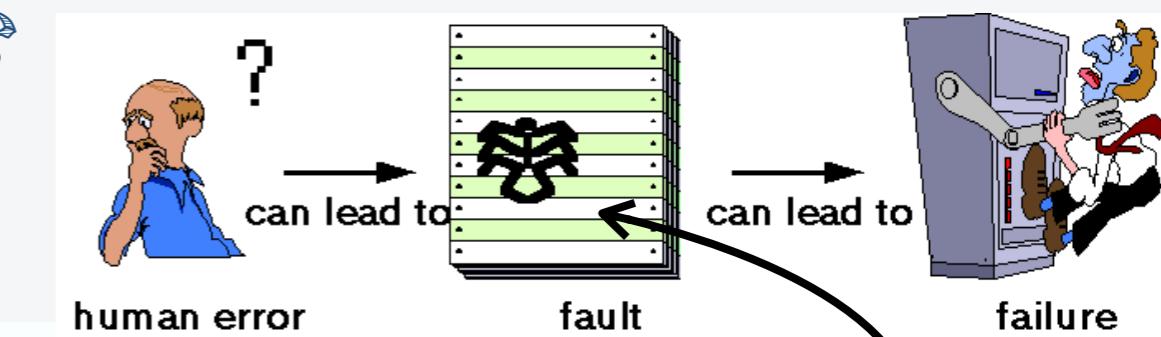
P

- The art of unit testing: with examples in C#. 2nd edition Roy Osherove. Manning, 2014.
  - Capítulo 4. Interaction testing using mock objects
- Easier testing with EasyMock. Elliotte Rusty Harold
  - <http://www.ibm.com/developerworks/java/library/j-easymock/index.html>
- Mocks aren't stubs. Martin Fowler. 2007
  - <http://martinfowler.com/articles/mocksArentStubs.html>
- Effective Unit Testing. Lasse Koskela. Manning, 2013.
  - Capítulo 3. Test doubles
- XUnit test patterns. Gerard Meszaros, 2007.
  - Capítulo 11. Using Test doubles
  - <http://xunitpatterns.com/Using%20Test%20Doubles.html>

# REPASO SESIONES S01..S05

## SESIÓN S01

P



### PROBAMOS PORQUE ...

- Cometemos errores

Las pruebas consumen mucho tiempo del desarrollo!!!

### TÉCNICAS

### CÓMO PROBAMOS??



### PROBAMOS PARA ...

- Encontrar defectos (VERIFICACIÓN)
- Juzgar si la calidad del sw es aceptable (VALIDACION)
- Prevenir defectos (Pruebas estáticas)
- Toma efectiva de decisiones (Métricas)

### Pruebas DINÁMICAS (es necesario ejecutar el código):

- Pruebas UNITARIAS: encuentran DEFECTOS en las unidades. Necesitamos AISLAR nuestro SUT (controlando sus dependencias externas con DOBLES)

#### Diseño:

- ▶ métodos de caja BLANCA (camino básico) (SESIÓN S01)
- ▶ métodos de caja NEGRA (particiones equivalentes) (SESIÓN S03)

#### Automatización

- (S02)
- ▶ Drivers (verif. basada en el estado, Usan STUBS para controlar las entradas indirectas de nuestro SUT) (S05)
  - ▶ Drivers (verif. basada en el comportamiento. Usan MOCKS para observar las entradas indirectas y registrar las interacciones de nuestro SUT con sus dependencias externas)

### HERRAMIENTAS

- Lenguaje JAVA
- Herram. construcción de proyectos: MAVEN
- Frameworks: JUnit, EasyMock

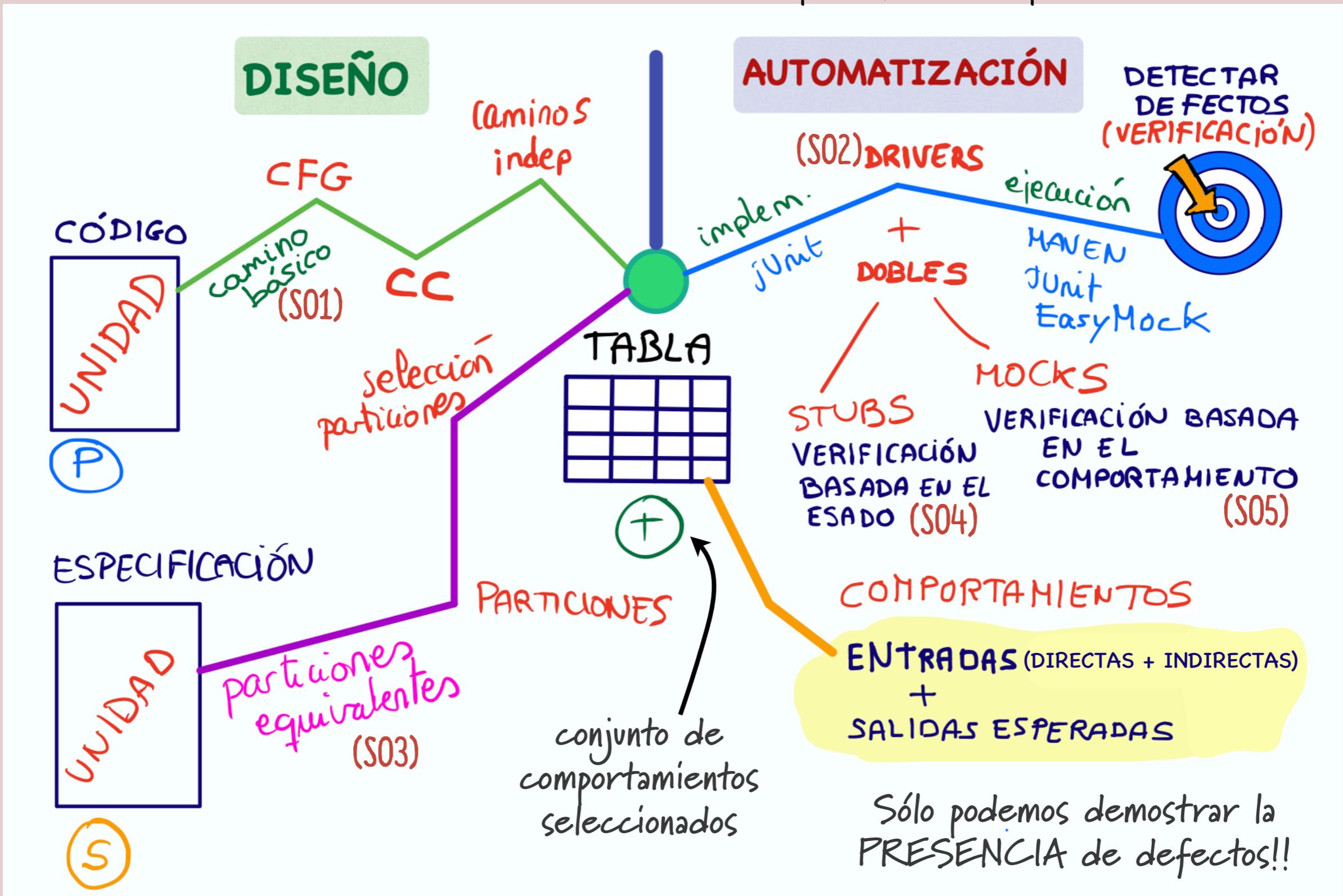
# REPASO SESIONES S01..S05

P  
P  
S

El proceso de **DISEÑO** consiste en seleccionar, de forma sistemática, un conjunto de comportamientos a probar. El conjunto obtenido será efectivo y eficiente

No podemos hacer pruebas exhaustivas!!!

El proceso de **AUTOMATIZACIÓN** consiste en la ejecución de los comportamientos seleccionados, realización de las verificaciones correspondientes y obtención del informe de prueba, todo ello "pulsando un botón"



# REPASO SESIONES P01..P05

Las sesiones prácticas nos permitirán comprender mejor los conceptos teóricos

## SESIÓN P01A

Durante el curso vamos realizar pruebas DINÁMICAS. Para ello necesitaremos AUTOMATIZAR la ejecución de casos de prueba. Necesitaremos una herramienta de CONSTRUCCIÓN DE PROYECTOS. Usaremos MAVEN. Maven proporciona 3 build scripts denominados ciclos de vida, y que podemos configurar usando el fichero pom.xml, en el que tendremos que reconocer 4 "zonas": coordenadas, propiedades, dependencias y build. Todos los proyectos maven tienen una estructura de directorios predefinida y fija. El proceso de construcción termina con BUILD SUCCESS o BUILD FAILURE, y genera el directorio target. Los ficheros usados y/o generados por Maven se denominan artefactos y se identifican por sus coordenadas.

## SESIÓN P01B

Comportamiento = datos de entrada+resultado esperado

Podemos diseñar los casos de prueba a partir del código de nuestro SUT usando el método del camino básico.

DISEÑO

Seleccionaremos un conjunto de comportamientos programados que garantizan la ejecución de todas las líneas de código (al menos una vez) de nuestro SUT y que se ejercitan todas las condiciones del SUT en su vertiente verdadera y falsa.

PRUEBAS UNITARIAS

El conjunto obtenido es efectivo y eficiente

## SESIÓN P02

AUTOMATIZACIÓN

Usaremos JUnit 5 para implementar drivers.. Un driver ejecuta un comportamiento seleccionado previamente (diseñado) Podremos parametrizarlos, reduciendo así la duplicación de código.

Disponemos de diferentes sentencias assert para verificar el resultado de los tests.

También podemos ejecutarlos de forma selectiva usando etiquetas.

PRUEBAS UNITARIAS

JUnit genera un informe con 3 posibles resultados. Compilaremos y ejecutaremos los tests a través de Maven incluyendo la librería JUnit en el pom.

## SESIÓN P03

### DISEÑO

Podemos diseñar los casos de prueba a partir de la especificación de nuestro SUT (método de particiones equivalentes).

Seleccionaremos un conjunto de comportamientos especificados que garantizan la ejecución de todas las particiones de entrada/salida (al menos una vez), y que las particiones inválidas se prueban de una en una. El conjunto obtenido es efectivo y eficiente

## SESIÓN P04

### AUTOMATIZACIÓN

Implementamos drivers usando VERIFICACIÓN basada en el ESTADO.

Usaremos dobles, que sustituirán a las dependencias externas de nuestro SUT durante las pruebas.

Los dobles (**STUBS**) nos permiten controlar las entradas indirectas a nuestro SUT, para así poder AISLAR la unidad a probar (método java).

Para poder injectar los dobles durante las pruebas puede que necesitemos refactorizar nuestro SUT.

La implementación del doble está separada del código del driver

Maven se encargará de ejecutar las pruebas de forma automática!!!

### PRUEBAS UNITARIAS

## SESIÓN P05

### AUTOMATIZACIÓN

Implementamos drivers usando VERIFICACIÓN basada en el COMPORTAMIENTO.

Usaremos dobles, que sustituirán a las dependencias externas de nuestro SUT durante las pruebas.

Los dobles (**MOCKS**) nos permiten verificar la interacción de nuestro SUT con las dependencias externas, AISLANDO el código de la unidad a probar (método java).

Para poder injectar los dobles durante las pruebas puede que necesitemos refactorizar nuestro SUT.

Usaremos la librería EasyMock para implementar los dobles "dentro" del driver.

La librería EasyMock permite implementar los dos tipos de verificaciones.

# EJERCICIO 1

```
P
P
S S
REPARTO S01..S05; P01..P05

public class Currency {
 private String units;
 private double amount;
 private int cents;
 ExchangeRate converter=null;

 public Currency(double amount, String code) {
 this.units = code;
 this.amount = Double.valueOf(amount).intValue();
 this.cents = (int) ((amount * 100.0) % 100);
 }

 public void setConverter(ExchangeRate converter) {
 this.converter = converter;
 }

 //comprobamos si el objeto es válido
 public boolean checkConverter() {
 throw new UnsupportedOperationException("Not supported yet");
 }

 public Currency toEuros() {
 if (checkConverter()) {
 if ("EUR".equals(units)) {
 return this;
 } else {
 double input = amount + cents / 100.0;
 double rate;
 try {
 rate = converter.getRate(units, "EUR");
 double output = input * rate;
 return new Currency(output, "EUR");
 } catch (IOException ex) {
 return null;
 }
 }
 } else return null;
 }
}
```

- Implementa un driver usando verificación basada en el comportamiento para la unidad Currency.toEuros(), teniendo en cuenta que queremos pasar a euros la cantidad de 2.50 dólares ("USD"), que usamos un objeto ExchangeConverter válido, que el resultado de invocar a ExchangeConverter.getRate() es 1.5 y que el resultado esperado es 3.75 "EUR"



Usaremos el método IMockBuilder.withConstructor(), cuando creamos el doble

# DRIVER

Versión sin chequear el orden de invocaciones entre mocks

P

- Usamos EasyMock para generar los dobles de forma automática

P

```
@Test
public void testToEuros() {
 //resultado esperado
 Currency expected = new Currency(3.75, "EUR");
 //Creamos los dobles
 ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
 Currency testable = EasyMock.partialMockBuilder(Currency.class)
 .withConstructor(2.50, "USD")
 .addMockedMethod("checkConverter")
 .createMock();

 //Programamos las expectativas
 Assertions.assertDoesNotThrow(()->
 EasyMock.expect(mock.getRate("USD", "EUR"))
 .andReturn(1.5)
);
 EasyMock.expect(testable.checkConverter()).andReturn(true);
 testable.setConverter(mock);
 EasyMock.replay(mock, testable);
 Currency actual = testable.toEuros();
 assertEquals(expected, actual);
 EasyMock.verify(mock, testable);
}
```

En lugar de invocar al constructor por defecto, podemos invocar a cualquier otro constructor de la clase

.withConstructor(double.class, String.class)  
.withArgsConstructor(2.50, "USD")

Si hay más de un constructor con parámetros compatibles tenemos que indicar primero el tipo de los parámetros

- Si, por ejemplo la clase ExchangeRate tuviese un constructor con un parámetro de tipo X, deberíamos usar:

```
object = new X();
ExchangeRate mock = EasyMock.createMockBuilder(ExchangeRate.class)
 .withConstructor(object)
 .createMock();
```

# DRIVER

Versión que comprueba orden de invocaciones entre mocks

```
1. @Test
2. //versión con verificación basada en el comportamiento ESTRITA!!!
3. public void testToEurosStrict() {
4. //resultado esperado
5. Currency expected = new Currency(3.75, "EUR");
6. //Creamos los dobles
7. IMocksControl ctrl = EasyMock.createStrictControl();
8. ExchangeRate mock = ctrl.createMock(ExchangeRate.class);
9. Currency testable = EasyMock.partialMockBuilder(Currency.class)
10. .withConstructor(2.50, "USD")
11. .addMockedMethod("checkConverter")
12. .createMock(ctrl);
13. //Programamos las expectativas
14. EasyMock.expect(testable.checkConverter()).andReturn(true);
15. Assertions.assertDoesNotThrow(()->
16. EasyMock.expect(mock.getRate("USD", "EUR"))
17. .andReturn(1.5)
18.);
19. //EasyMock.expect(testable.checkConverter()).andReturn(true);
20. testable.setConverter(mock);
21. ctrl.replay();
22. Currency actual = testable.toEuros();
23. assertEquals(expected, actual);
24. ctrl.verify();
25. }
```

Creamos los dos dobles en el contexto de un objeto IMocksControl. El objeto "ctrl" chequea el orden de invocaciones entre los dobles.

Si descomentamos esta línea y comentamos la línea 14, el test devolverá FAILURE

- Si, por ejemplo la clase ExchangeRate tuviese un constructor con un parámetro de tipo X, deberíamos usar:

```
object = new X();
ExchangeRate mock = EasyMock.createMockBuilder(ExchangeRate.class)
 .withConstructor(object)
 .createMock();
```

# EJERCICIO PROPUESTO

S

P

- O Dado el siguiente código, implementa un driver usando verificación basada en el comportamiento, suponiendo que en la BD tenemos los alumnos indicados en la siguiente tabla antes de ejecutar el método. El método en cuestión obtiene un listado de alumnos después de acceder a una base de datos (tabla alumnos), o bien devuelve una excepción de tipo SQLException

```
public class Listados {
 public String porApellidos(Connection con, String tableName) throws SQLException {
 Statement stm = con.createStatement();
 //realizamos la consulta y almacenamos el resultado en un ResultSet
 ResultSet rs =
 stm.executeQuery("SELECT apellido1, apellido2, nombre FROM " + tableName);
 String result = "";
 //recorremos el ResultSet
 while (rs.next()) {
 String ap1 = rs.getString("apellido1");
 String ap2 = rs.getString("apellido2");
 String nom= rs.getString("nombre");
 result += ap1 + " " + ap2 + ", " + nom + "\n";
 }
 return result;
 }
}
```

tabla alumnos

Apellido1	Apellido2	Nombre
Garcia	Planelles	Jorge
Pérez	Verdú	Carmen
González	Alamo	Eva
Martínez	López	Roque

**Nota:** Connection, Statement y ResultSet son Interfaces Java, cuyos métodos pueden devolver una excepción de tipo SQLException, al igual que el método a probar.

**Resultado esperado:**

"Garcia Planelles, Jorge\nPérez Verdú, Carmen\nGonzález Alamo, Eva\nMartínez López, Roque"