

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"

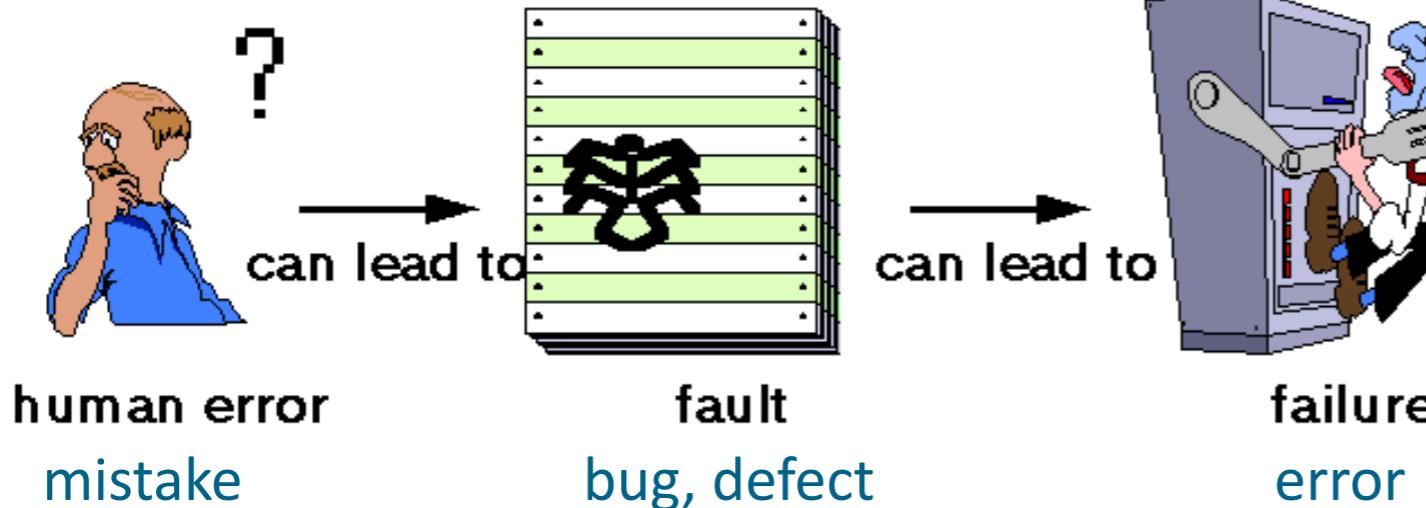
Glenford J. Myers (1979)

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)

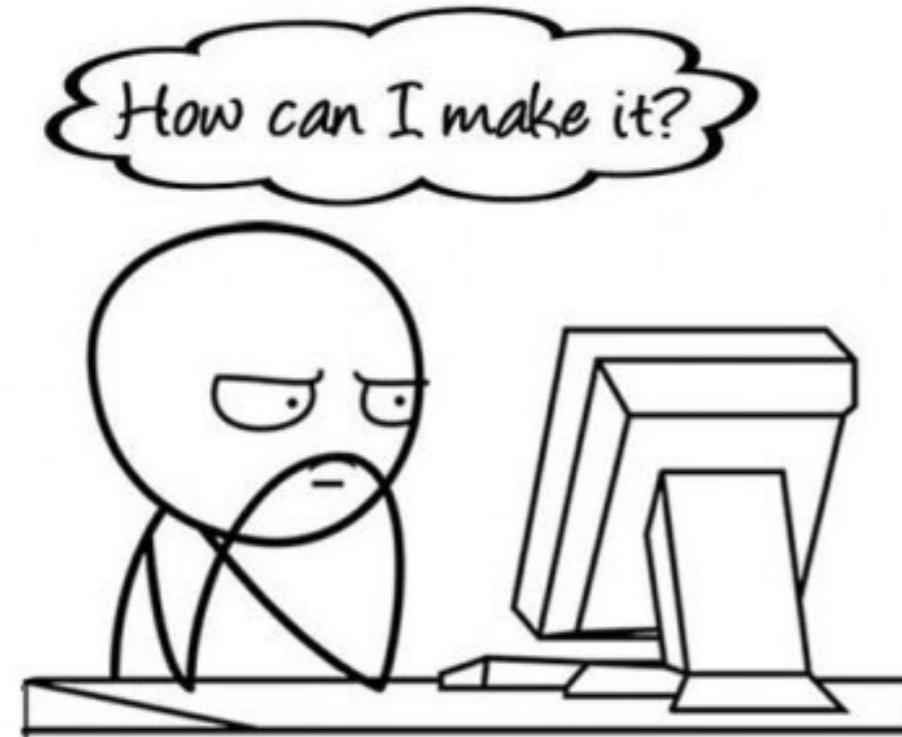


DEFINICIÓN DEL PROCESO DE PRUEBAS

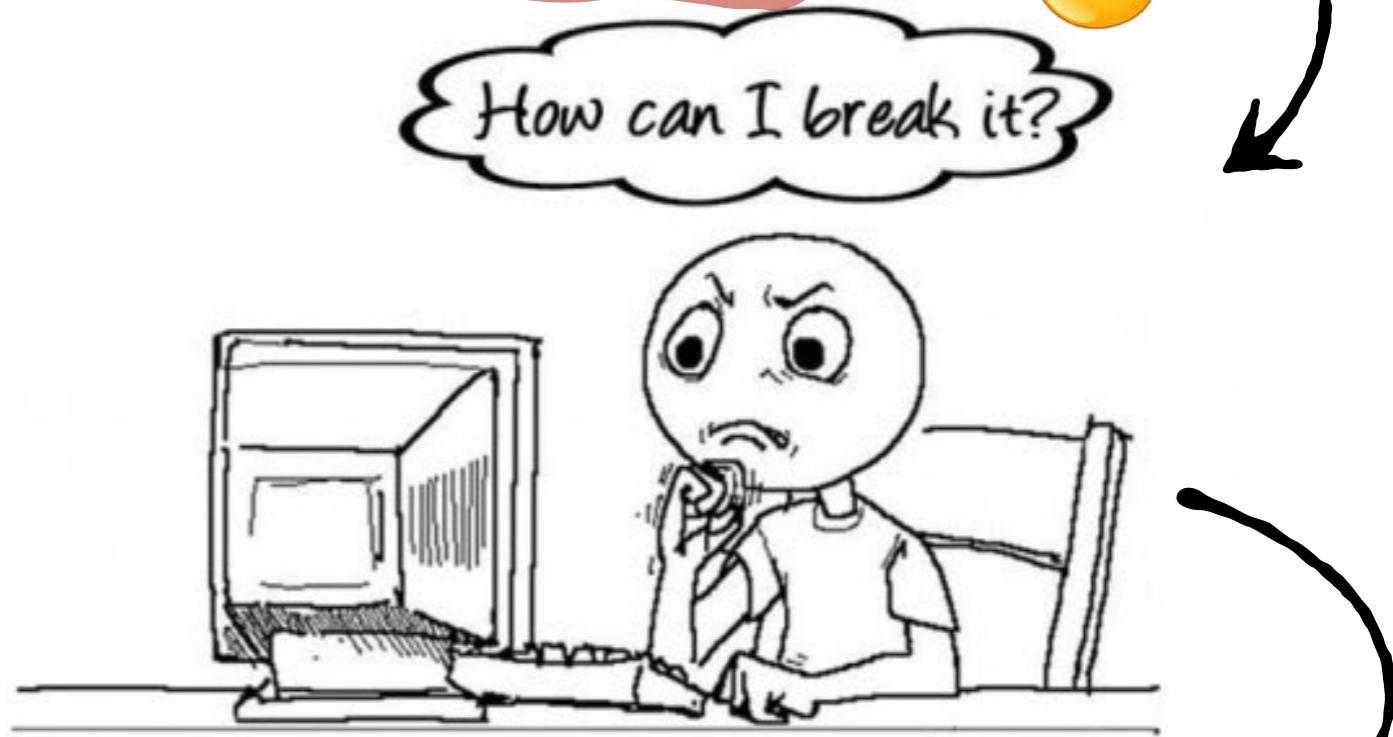
P

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

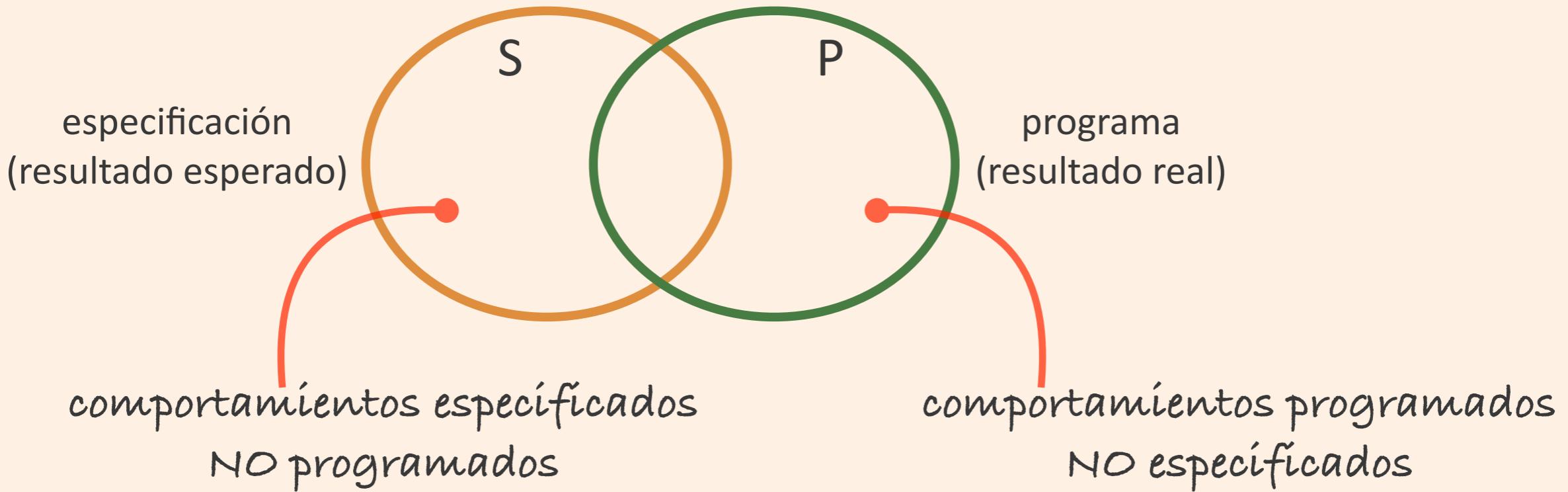
4 EVALUACIÓN del proceso de pruebas y emitir un informe

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!!!

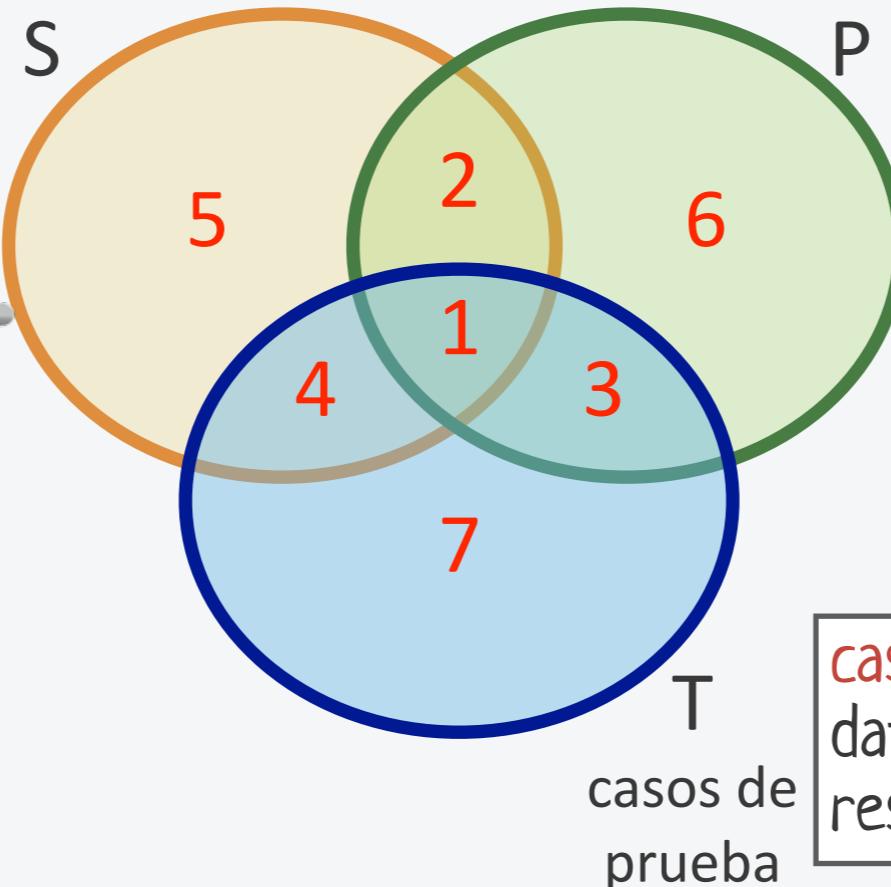
COMPORTAMIENTOS ESPECIFICADOS, PROGRAMADOS, Y PROBADOS

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

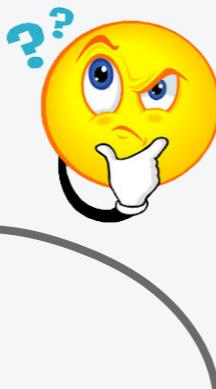
DISEÑO DE CASOS DE PRUEBA

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

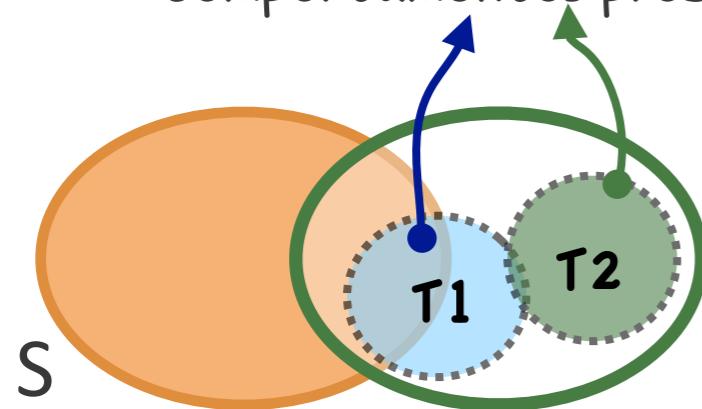
ID	d1	d2	...	Expected Output
C1	?	?	?	?
C2				
C3				
...				
CN?				



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

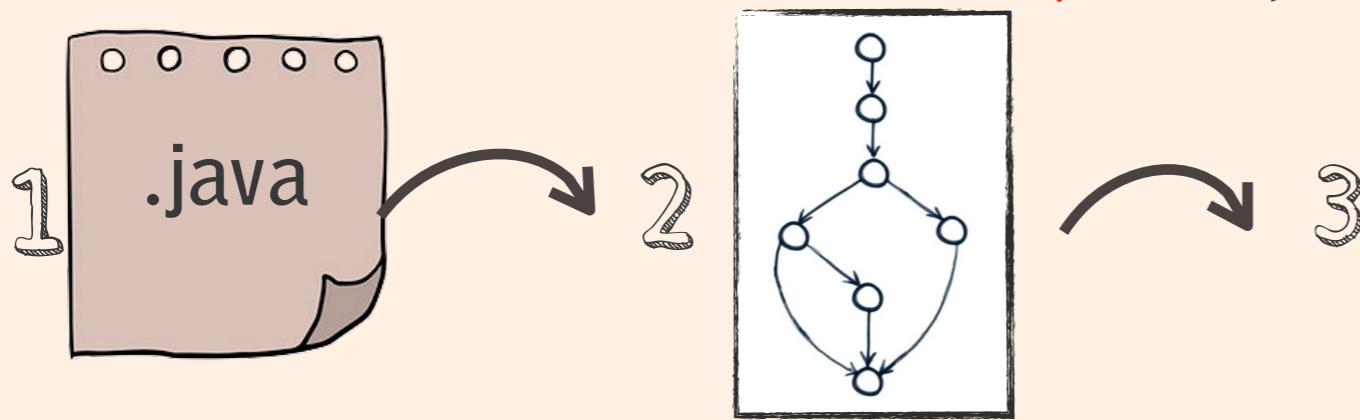


Tabla de casos de prueba

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

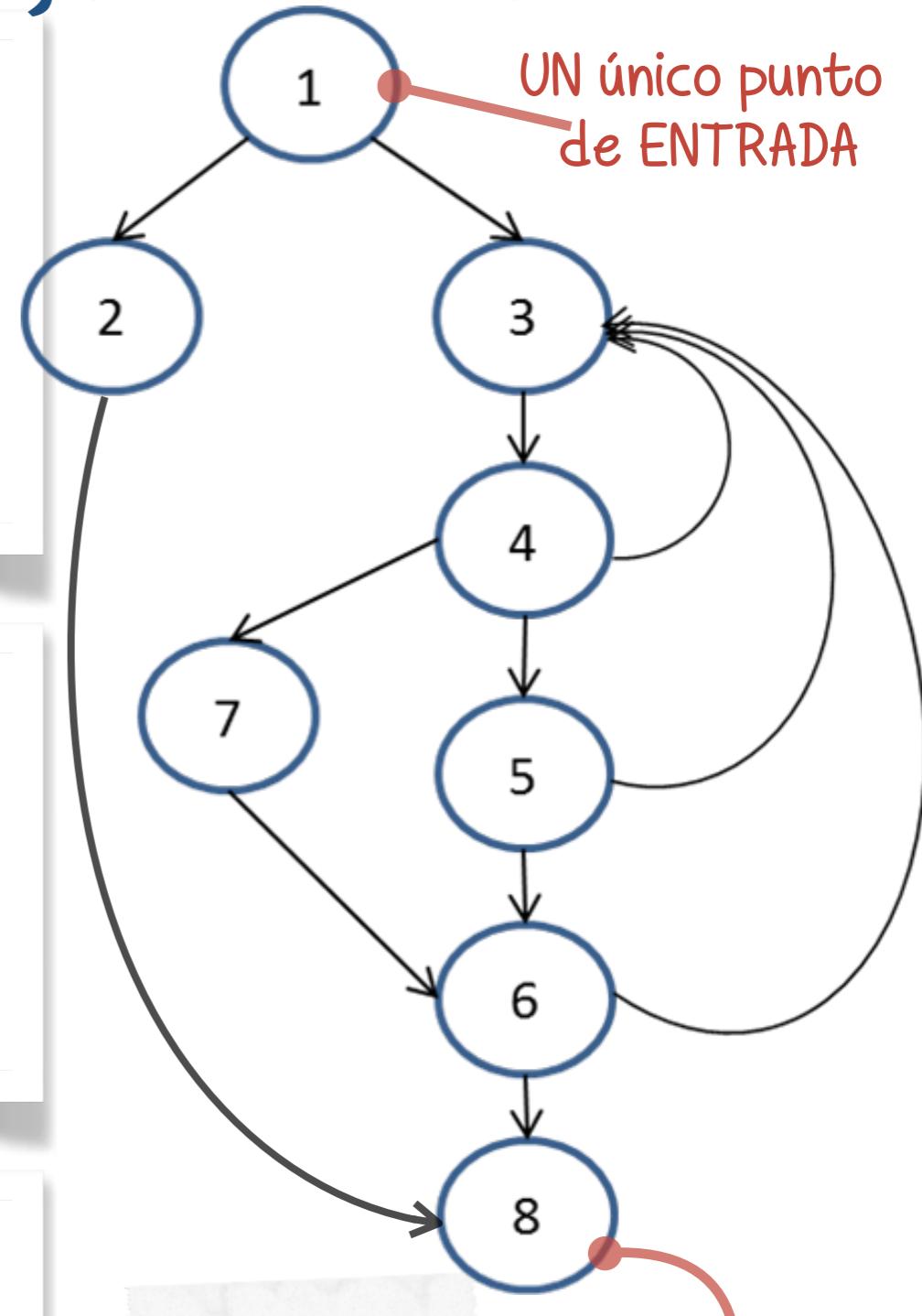
- Una llamada a una "función" cede el control a dicha función
- Dentro de la unidad a probar, la invocación de una "función" es una sentencia secuencial

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



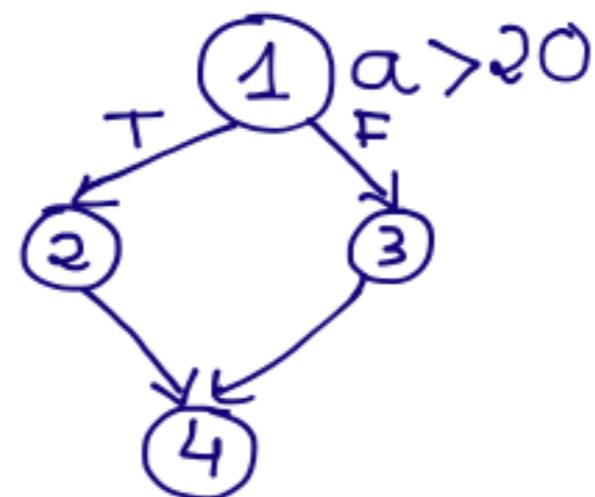
UNIDAD de programa = método Java

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 ejercent 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

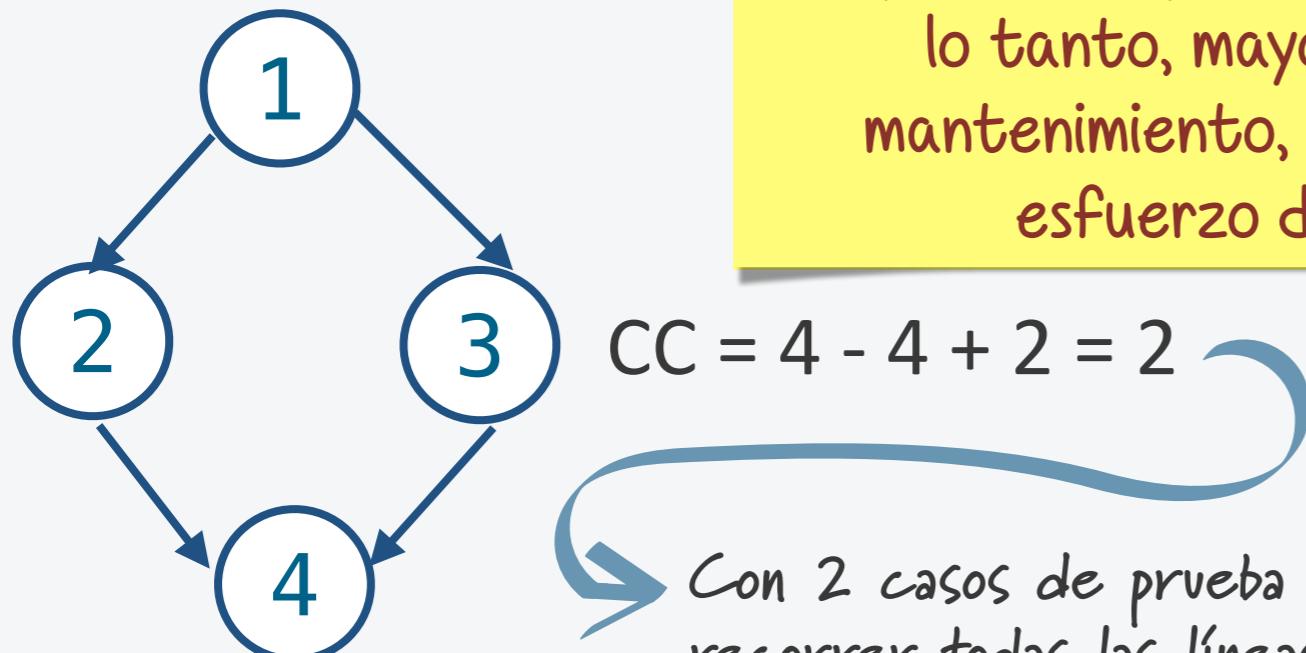
- Se calcula a partir del grafo de flujo:

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

- 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

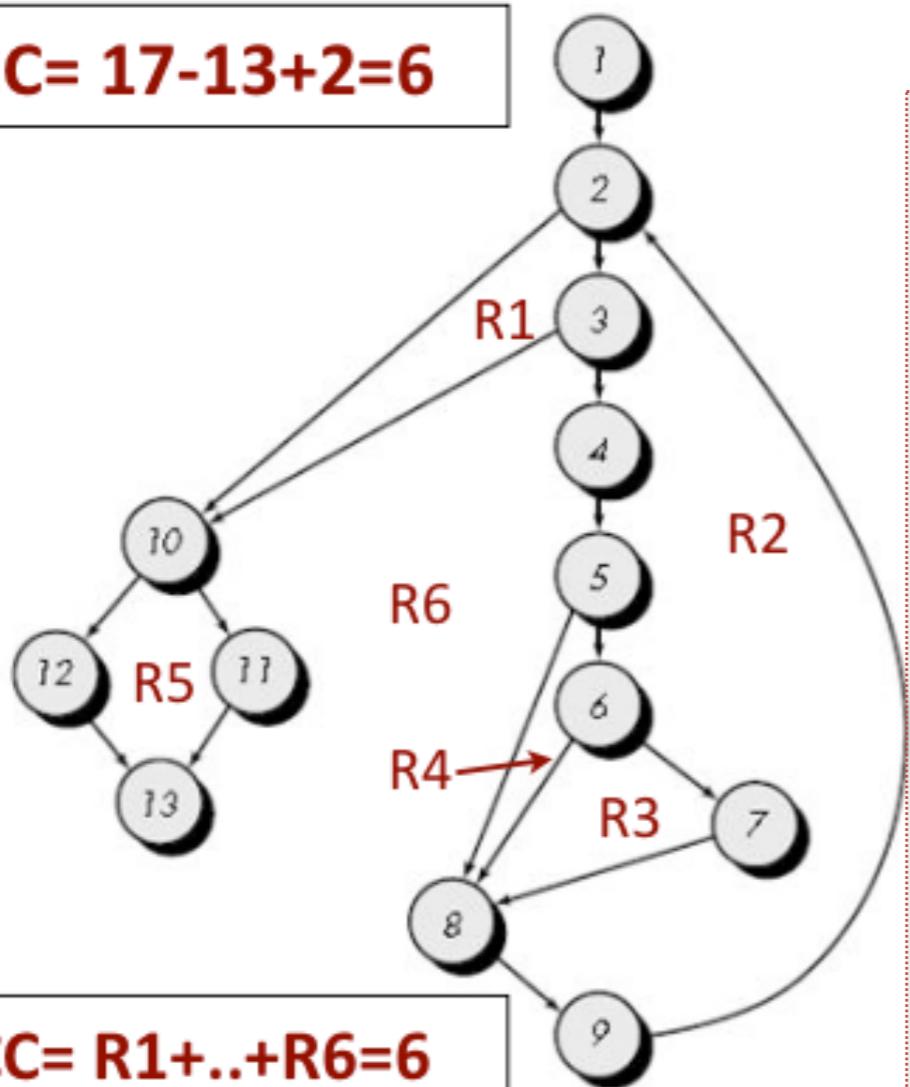
CC = número de arcos – número de nodos + 2

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;

```

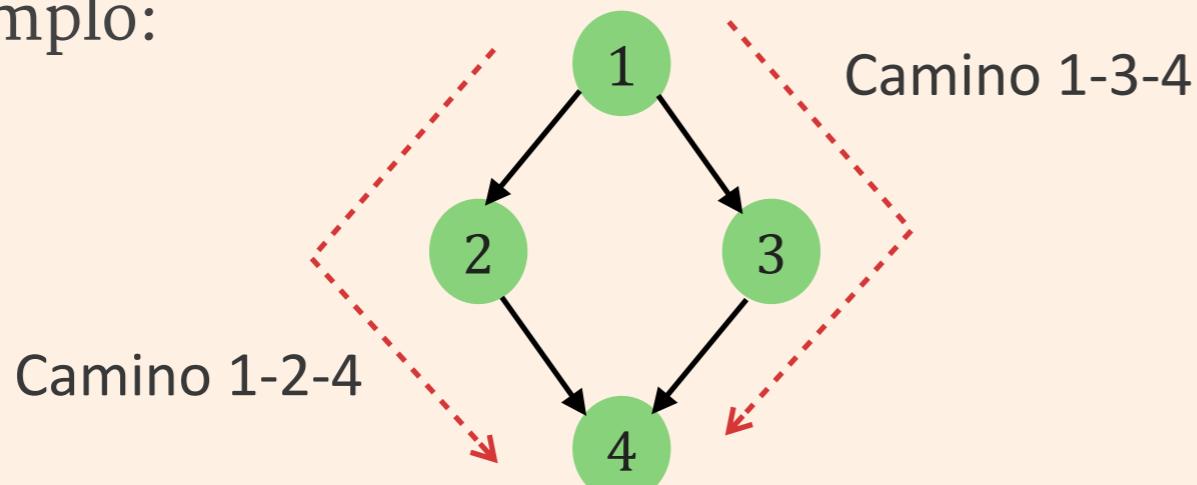
$$CC = 5 + 1 = 6$$

CAMINOS INDEPENDIENTES

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

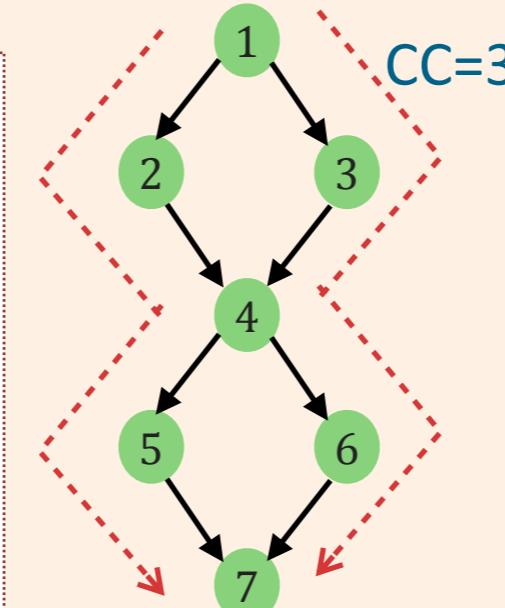
- Buscamos (como máximo) tantos caminos independientes como valor obtenido de CC
 - Cada camino independiente contiene un nodo, o bien una arista, que no aparece en los caminos independientes anteriores
 - Con ellos recorreremos TODOS los nodos y TODAS las aristas del grafo

- 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;  
}
```



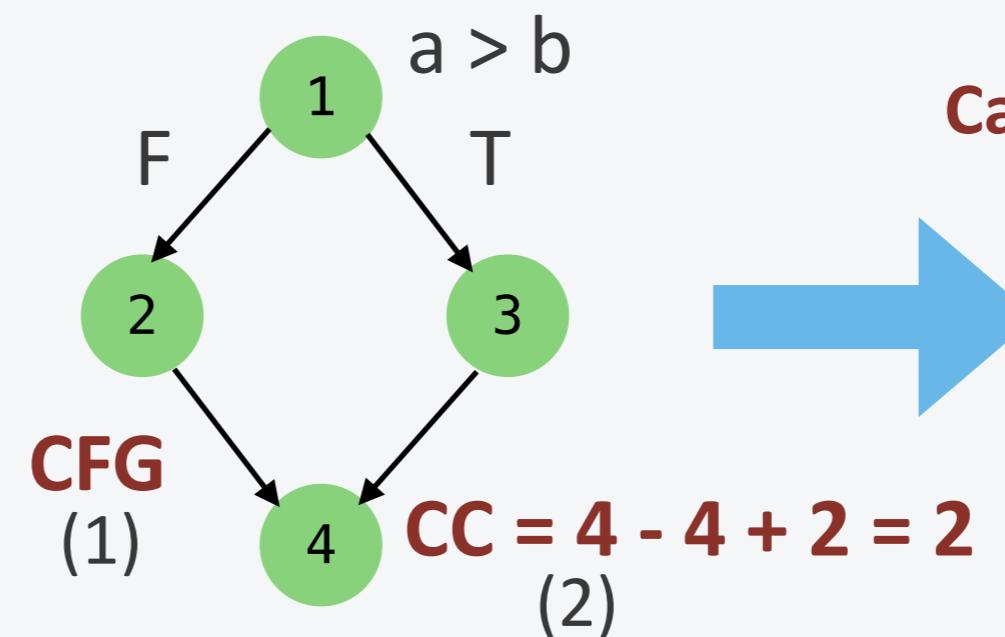
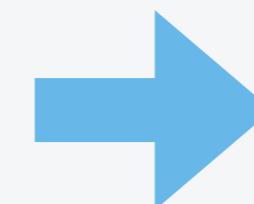
Ambas opciones son válidas

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

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

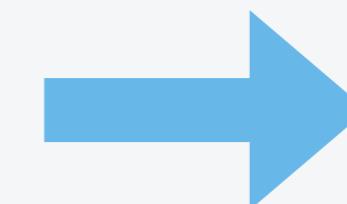
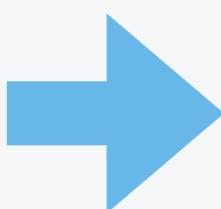


Tabla resultante del diseño de casos de prueba



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
```

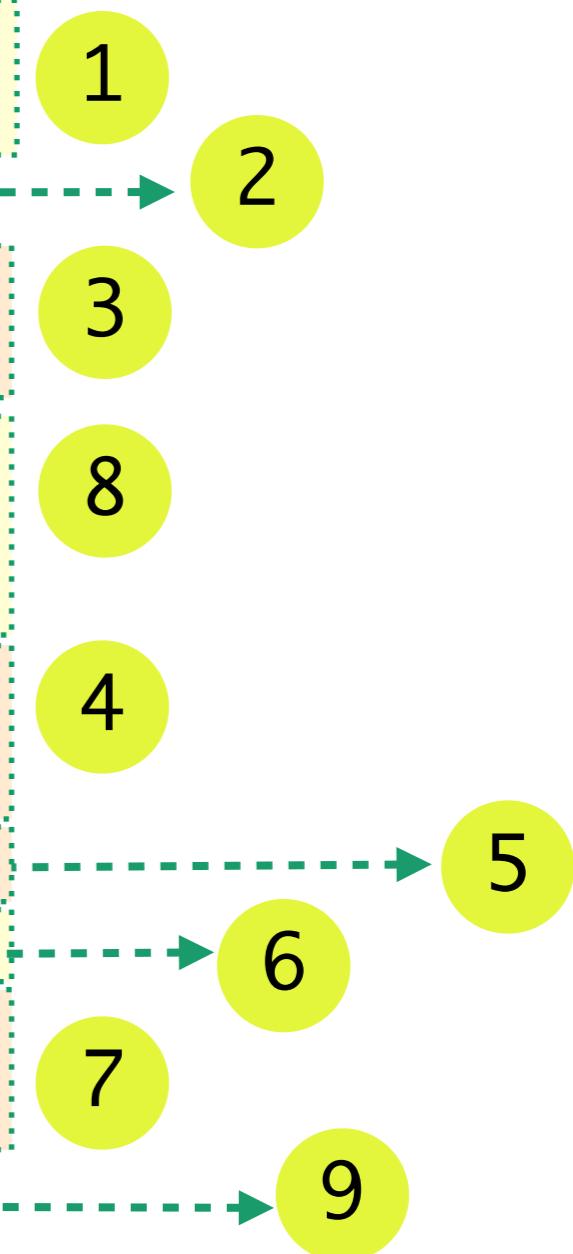
Especificación del método search():
Dado un vector de enteros ordenados
ascendentemente, y dado un entero
(key) como entrada, el método
search() busca la posición de key en
el vector y devuelve el valor
found=true si lo encuentra, así como
su posición en el vector (dada por
índex). Si el valor de key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

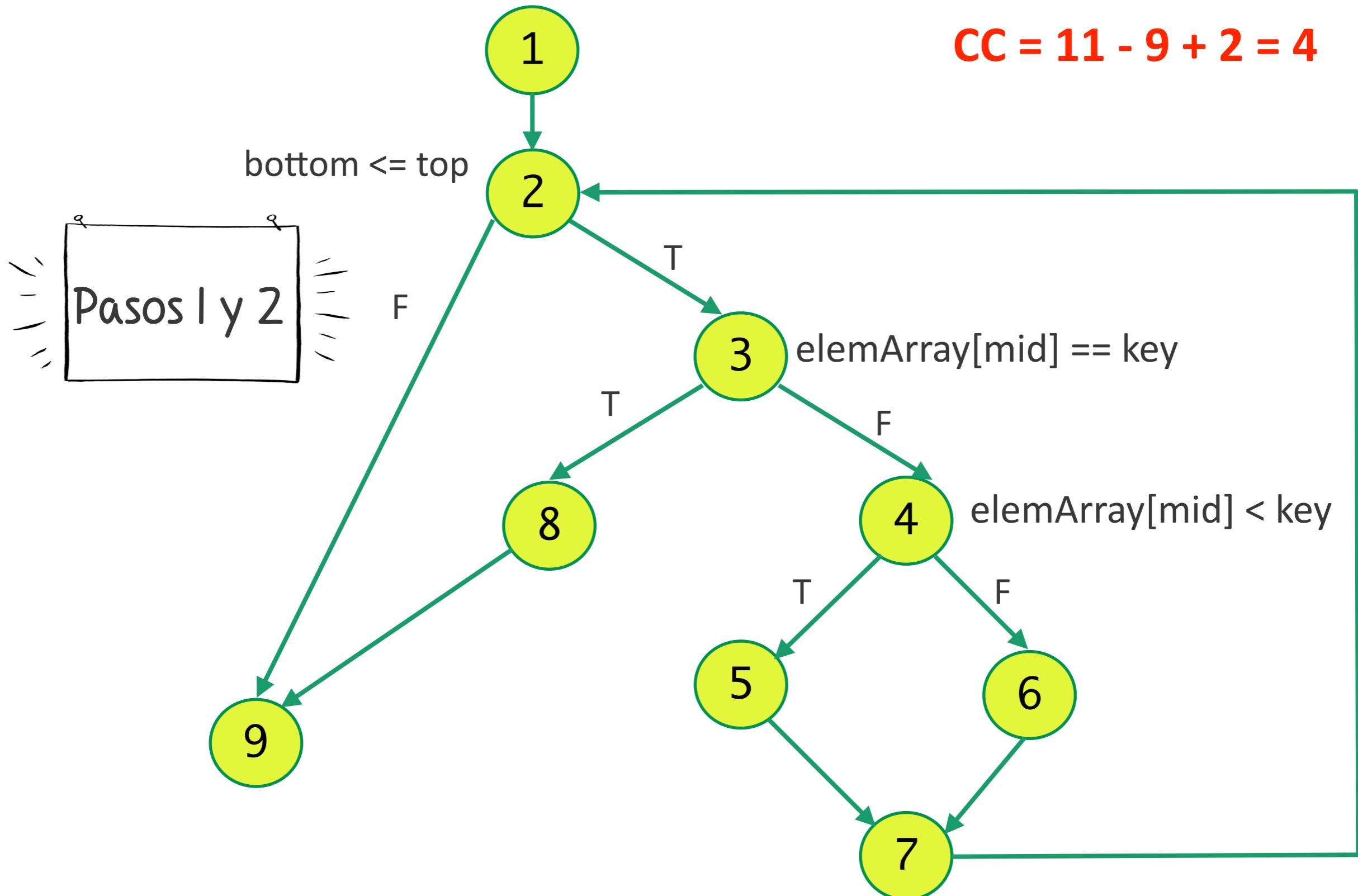
class BinSearch

```
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

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



CAMINOS INDEPENDIENTES

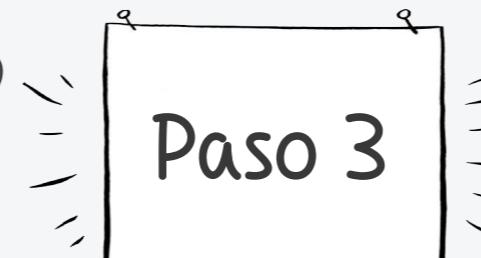
P

S

- Posible conjunto de caminos independientes

P

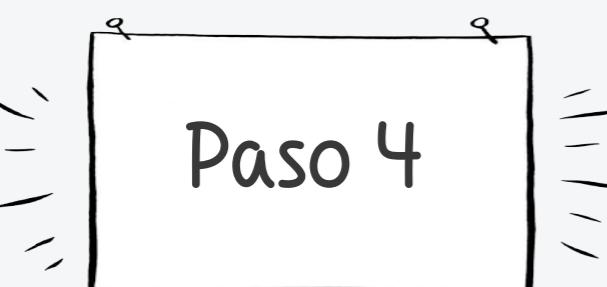
- C1: 1, 2, 3, 4, 6, 7, 2, 9
- C2: 1, 2, 3, 4, 5, 7, 2, 9
- C3: 1, 2, 3, 8, 9



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!!!

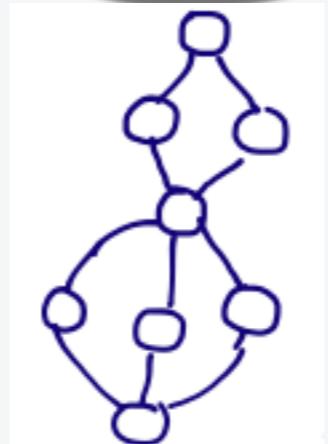
```
package ppss;

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	d11	d12	...	d1q	r11	...	r1k	
...								
CN	dn1	dn2	...	dnq	r n1	...	r nk	

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

REFERENCIAS BIBLIOGRÁFICAS



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

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

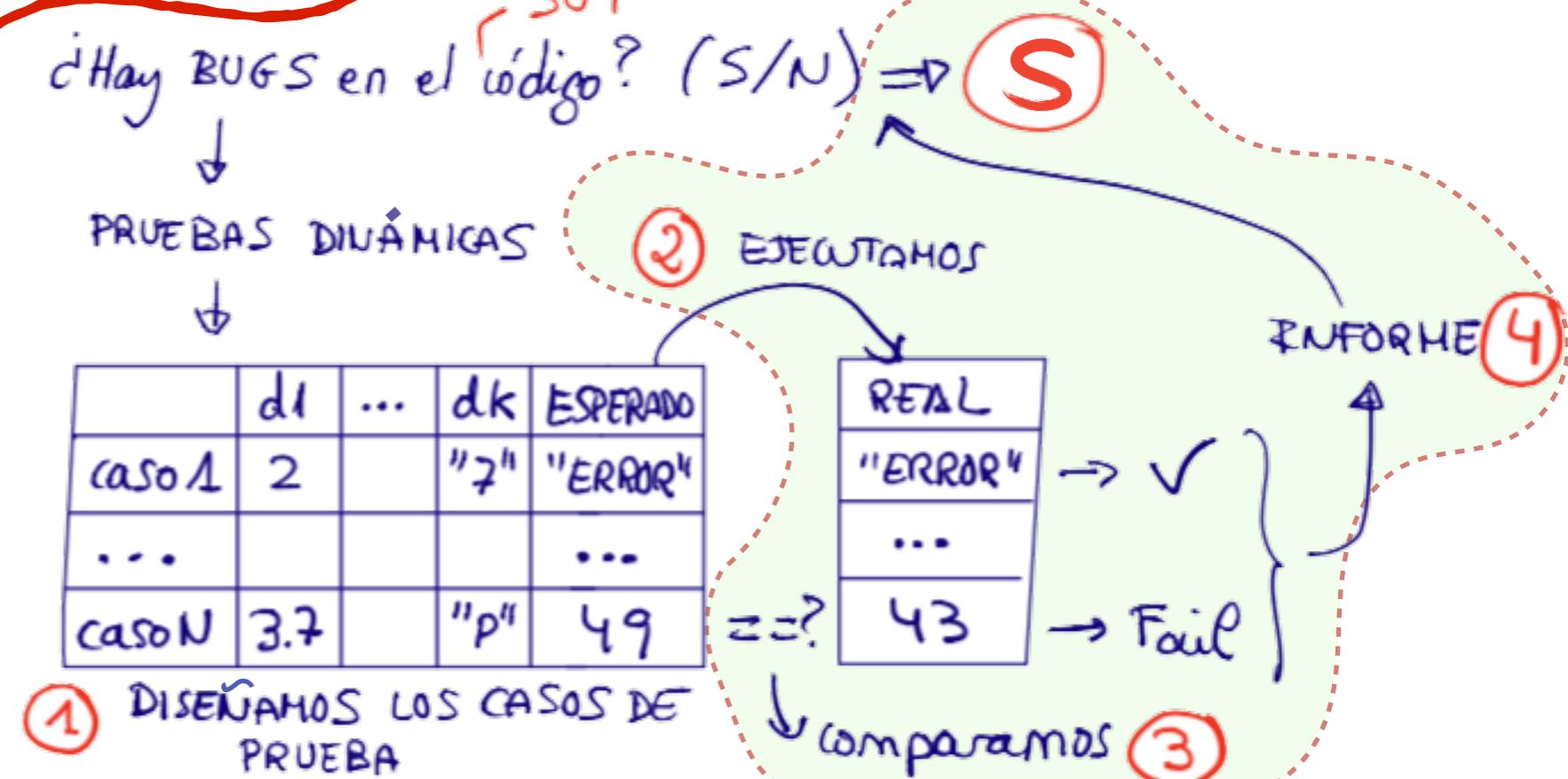
ACTIVIDADES
DEL PROCESO DE PRUEBAS

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)



El OBJETIVO es poder realizar los procesos 2, 3 y 4 "pulsando un botón"

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

P

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

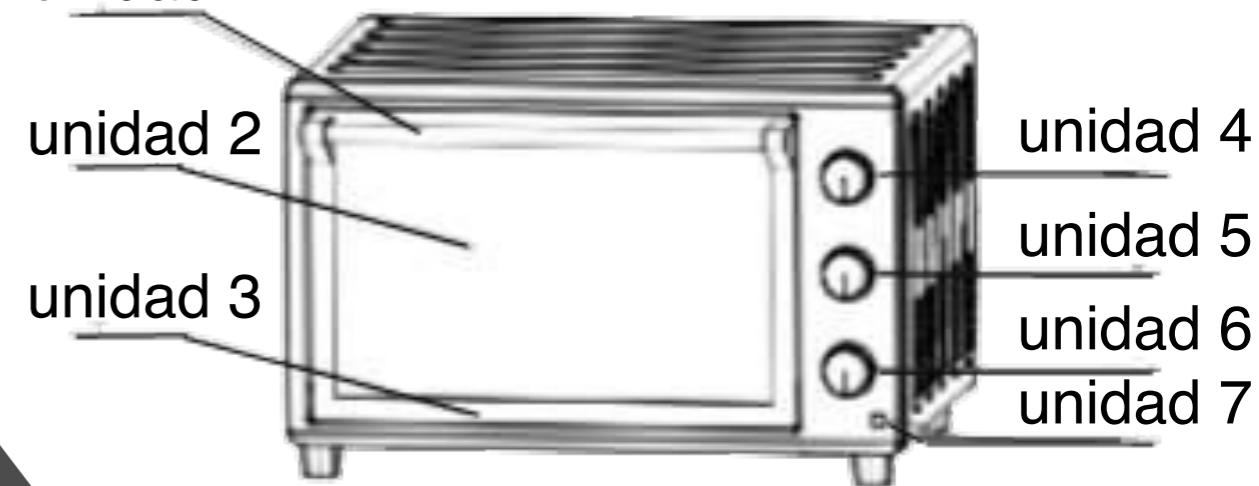
unidad 3

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

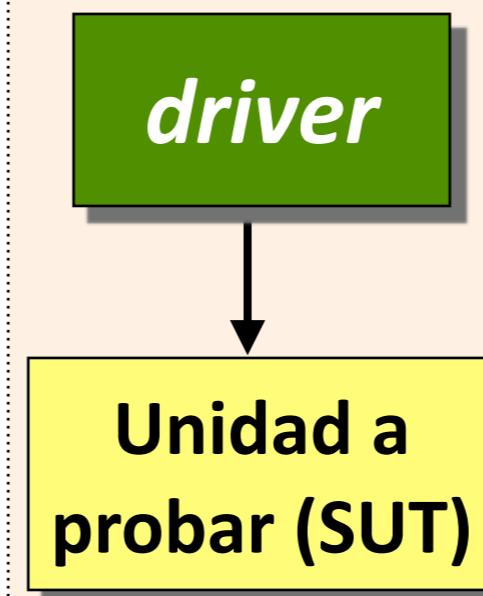
P Requieren ejecutar una unidad de forma **AISLADA** para poder detectar defectos en el código de dicha unidad

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.

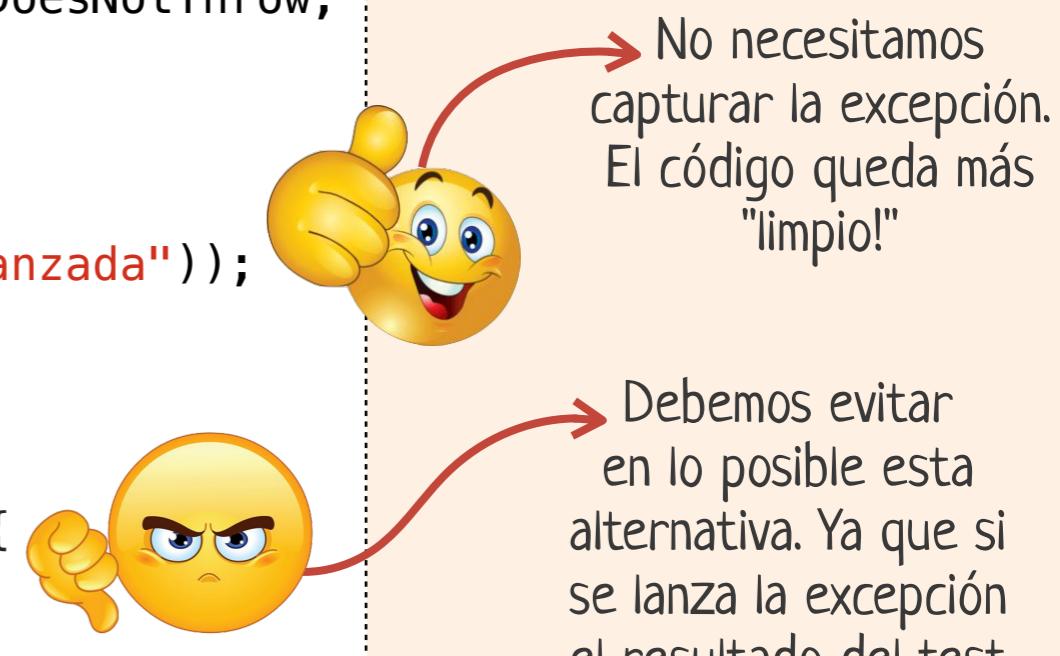
Usage Note

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.

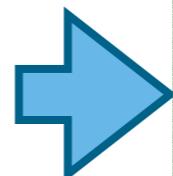
El test parametrizado se invocará tantas veces como elementos de tipo `Arguments` tengamos. En el ejemplo serán 7 veces.

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;
...
```

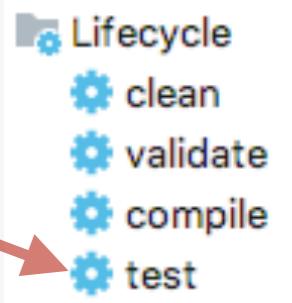
- El valor "test" de la etiqueta <scope> indica que la librería sólo se utiliza para compilar y ejecutar los tests. Por lo tanto NO podremos importar ninguna de sus clases desde /src/main/java
- Si usamos tests **parametrizados**, necesitaremos incluir también la libreria "junit-params" para poder usar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

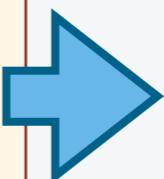


mvn test

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

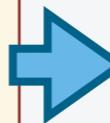
- Debes tener en cuenta de que si configuramos el plugin en el pom y también desde línea de comandos, la configuración del pom PREVALECE sobre la de línea de comandos. Si por ejemplo, quisiéramos ejecutar siempre un cierto subconjunto de tests y ocasionalmente otro subconjunto, podríamos hacer esto:

```
<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>
```



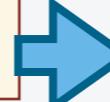
mvn test

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"



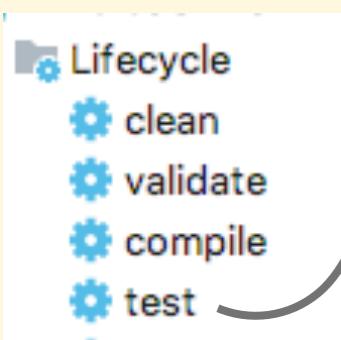
mvn test -Dfiltrar.por=""

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

Run 'CalculadoraTest'

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

Desde el menú
contextual de
CalculadoraTest.java

Sesión 2: Drivers

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

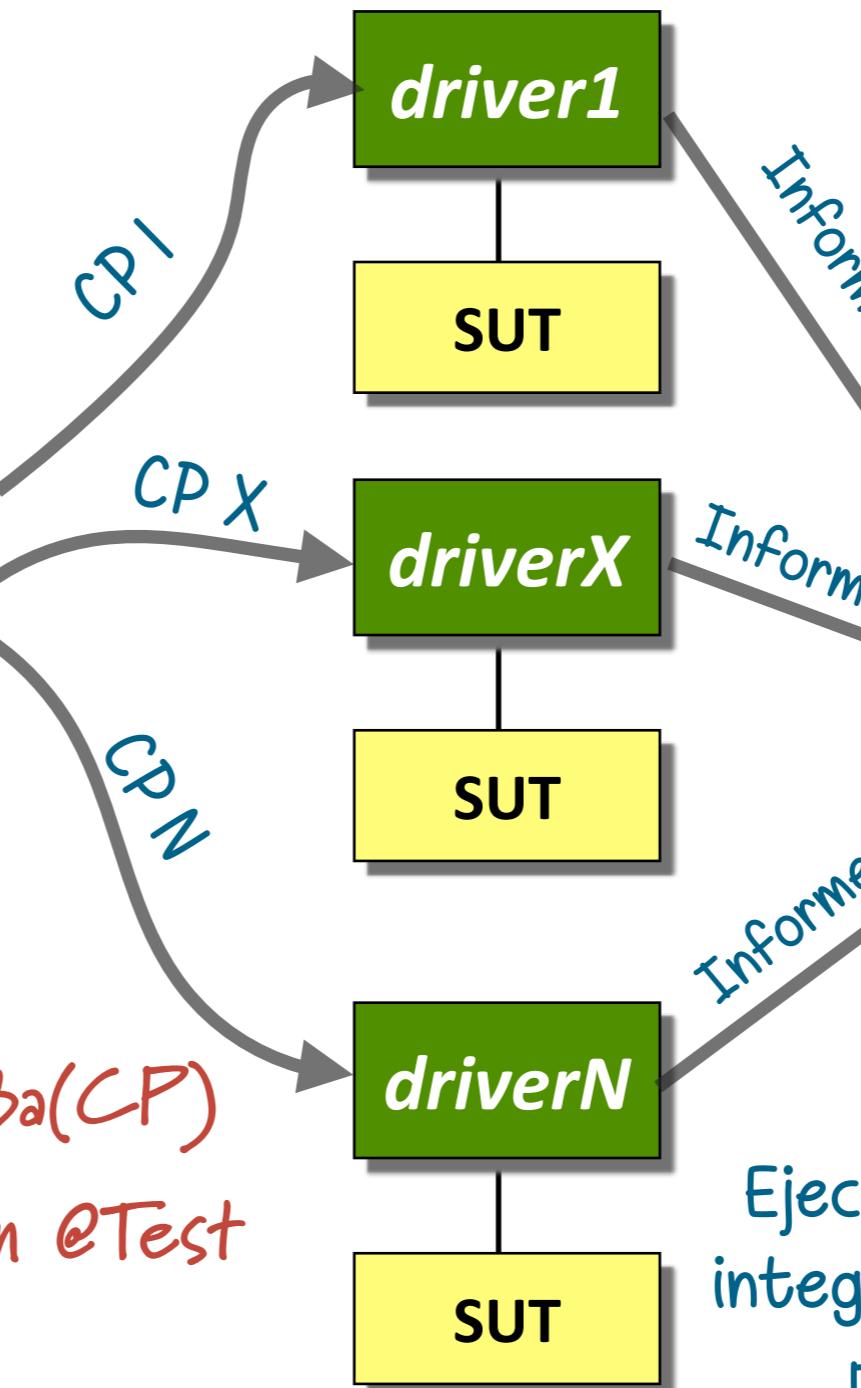
tabla de casos de prueba

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

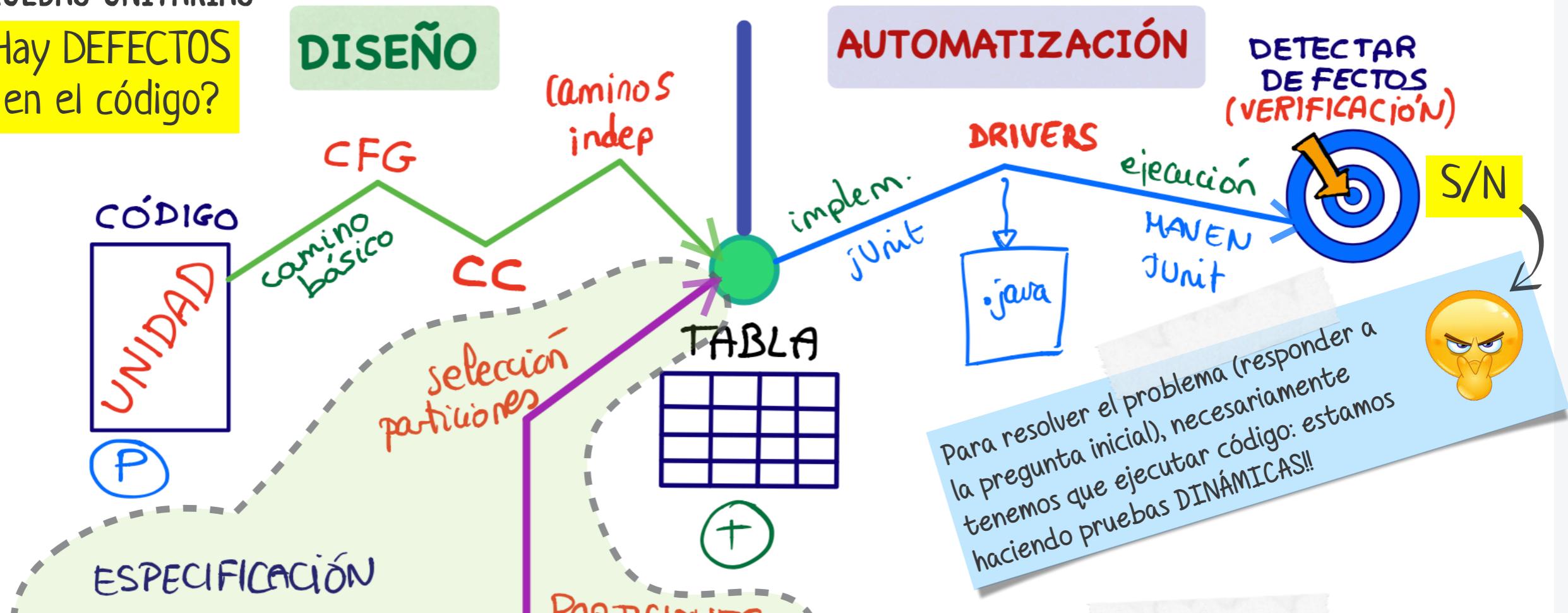
Vamos al laboratorio...

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)

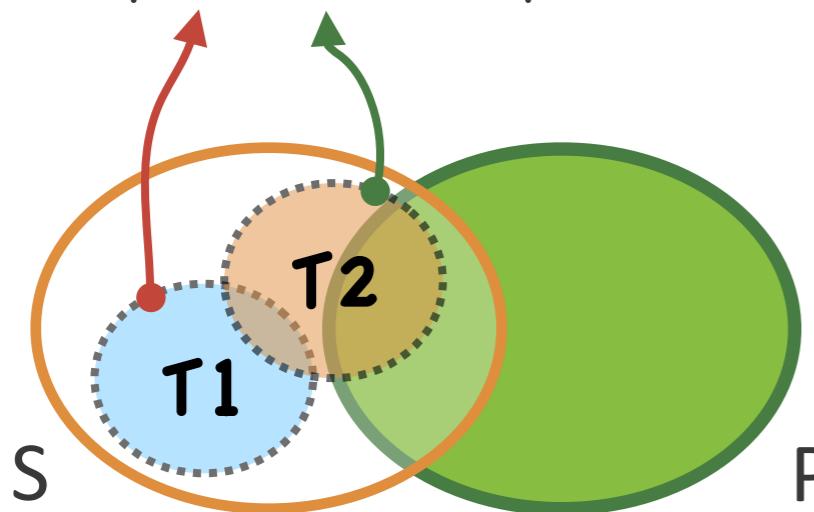
Ahora vamos a explicar cómo hacerlo partiendo de la **ESPECIFICACIÓN**.

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

Pruebas de sistema

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

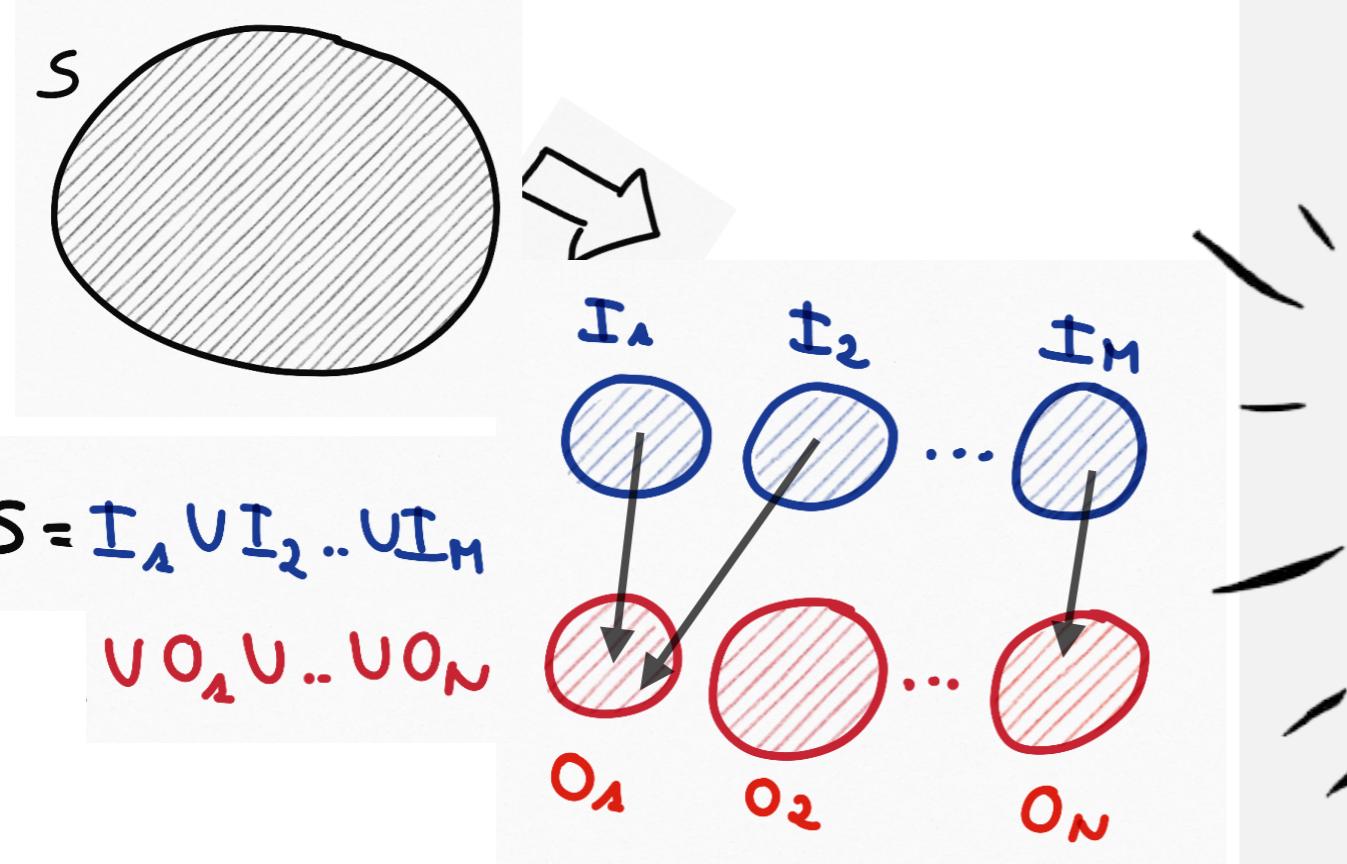
- Se trata de un proceso **SISTEMÁTICO** que identifica, a partir de la **ESPECIFICACIÓN** disponible, un conjunto de **CLASES** de equivalencia para **cada** una de las **entradas** y **salidas** del "elemento" (unidad, componente, sistema) a probar
- Cada clase de equivalencia (o partición) de entrada 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



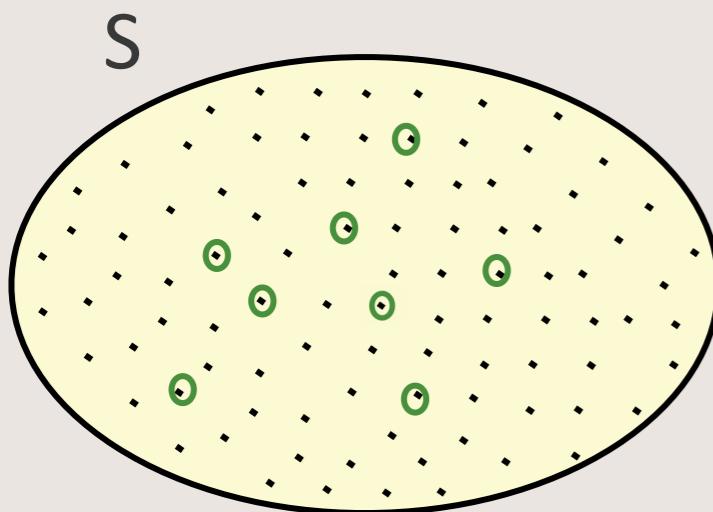
- Cada caso de prueba usará un **subconjunto** de particiones
- **NO** se trata de probar **TODAS** las combinaciones posibles, sino de garantizar que **TODAS** las particiones de entrada (y de salida) se prueban **AL MENOS UNA VEZ**

SISTEMATICIDAD Y PARTICIONAMIENTOS

detectar el máximo nº posible de errores

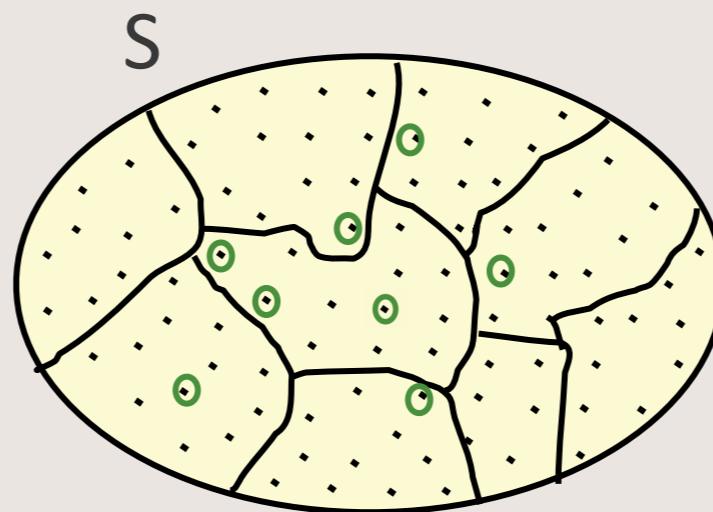
... con el MENOR nº posible de casos de prueba

- 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

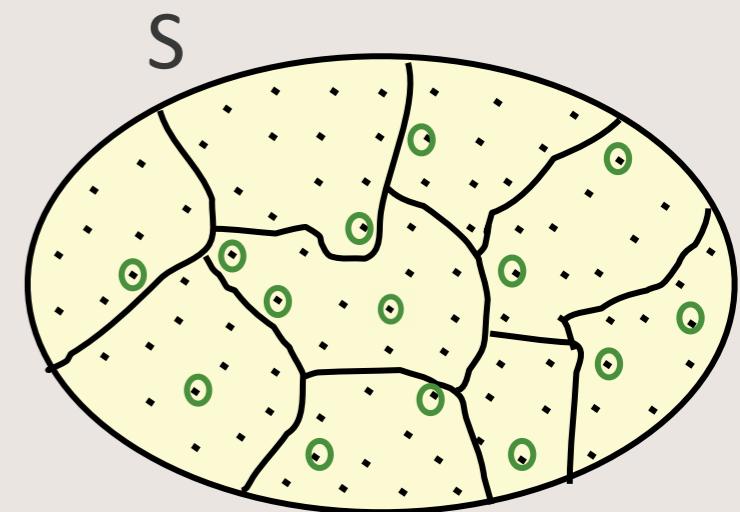


No particiones. Datos de prueba (círculos verdes) elegidos aleatoriamente

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



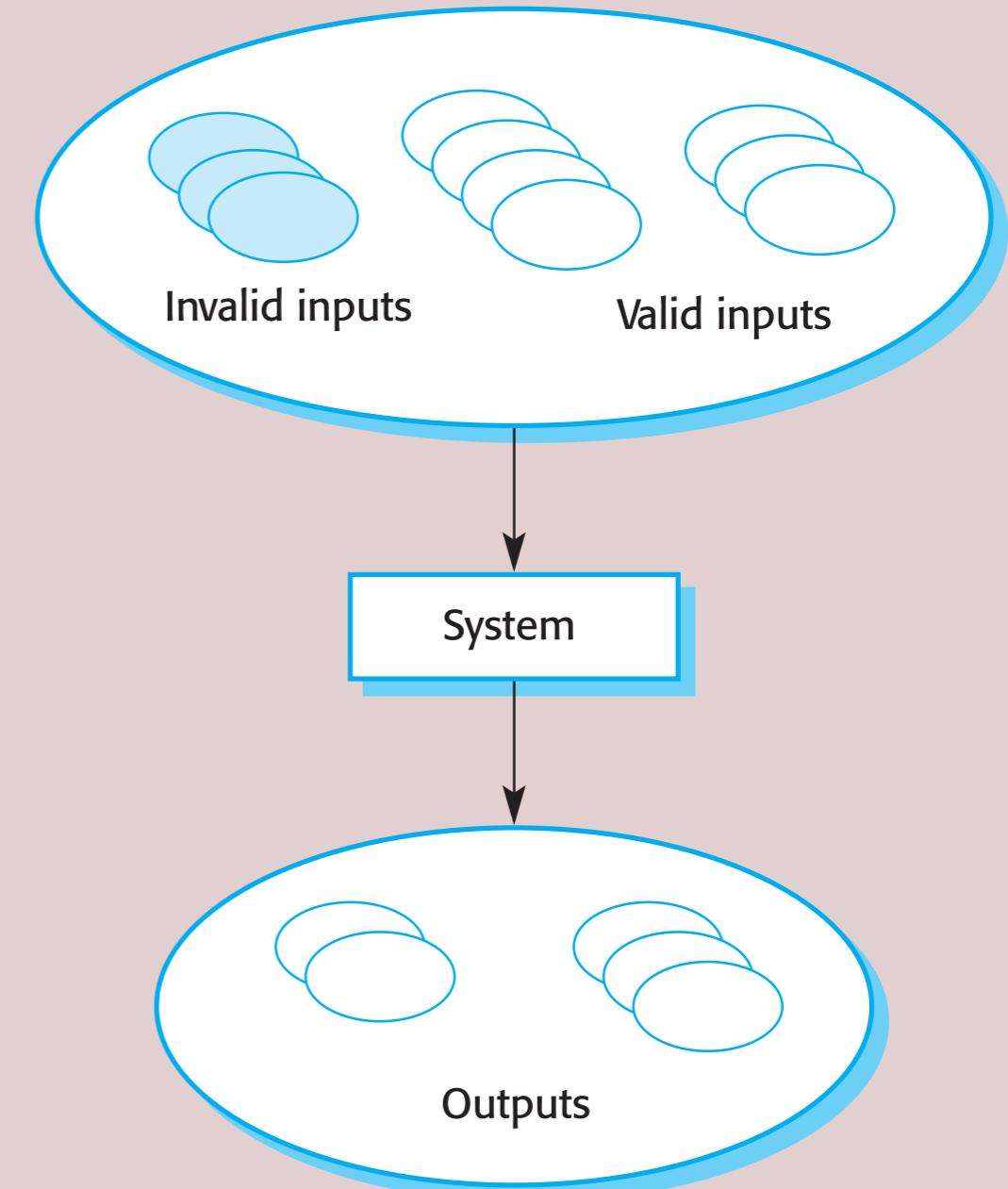
Lógicamente, para poder identificar las condiciones sobre las entradas, primero hay que tener claro cuántas y qué ENTRADAS tiene el elemento que queremos probar!!!!

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!!

P

P

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 (fuera 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: $m < 1 \wedge d = \{1..31\}$	



Aplicamos la regla #6



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

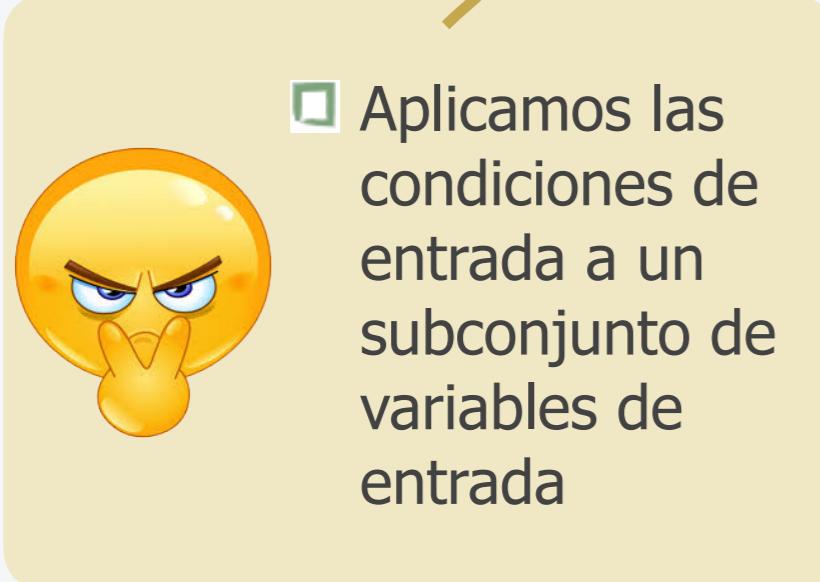
3 Entradas + 1 Salida

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



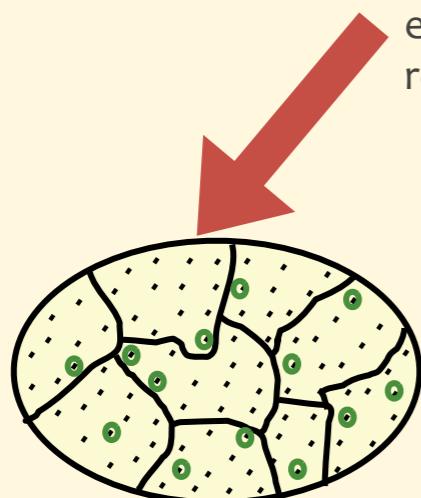
- Etiqueta las particiones de una misma E/S con la misma letra.
 - Por ejemplo, si las entradas son A y B, las particiones podrían etiquetarse como A₁, A₂,... N_{A1}, N_{A2},..., B₁, B₂,... N_{B1}, N_{B2},... para las clases válidas e inválidas de las entradas A y B respectivamente
- No olvides tener 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= ...

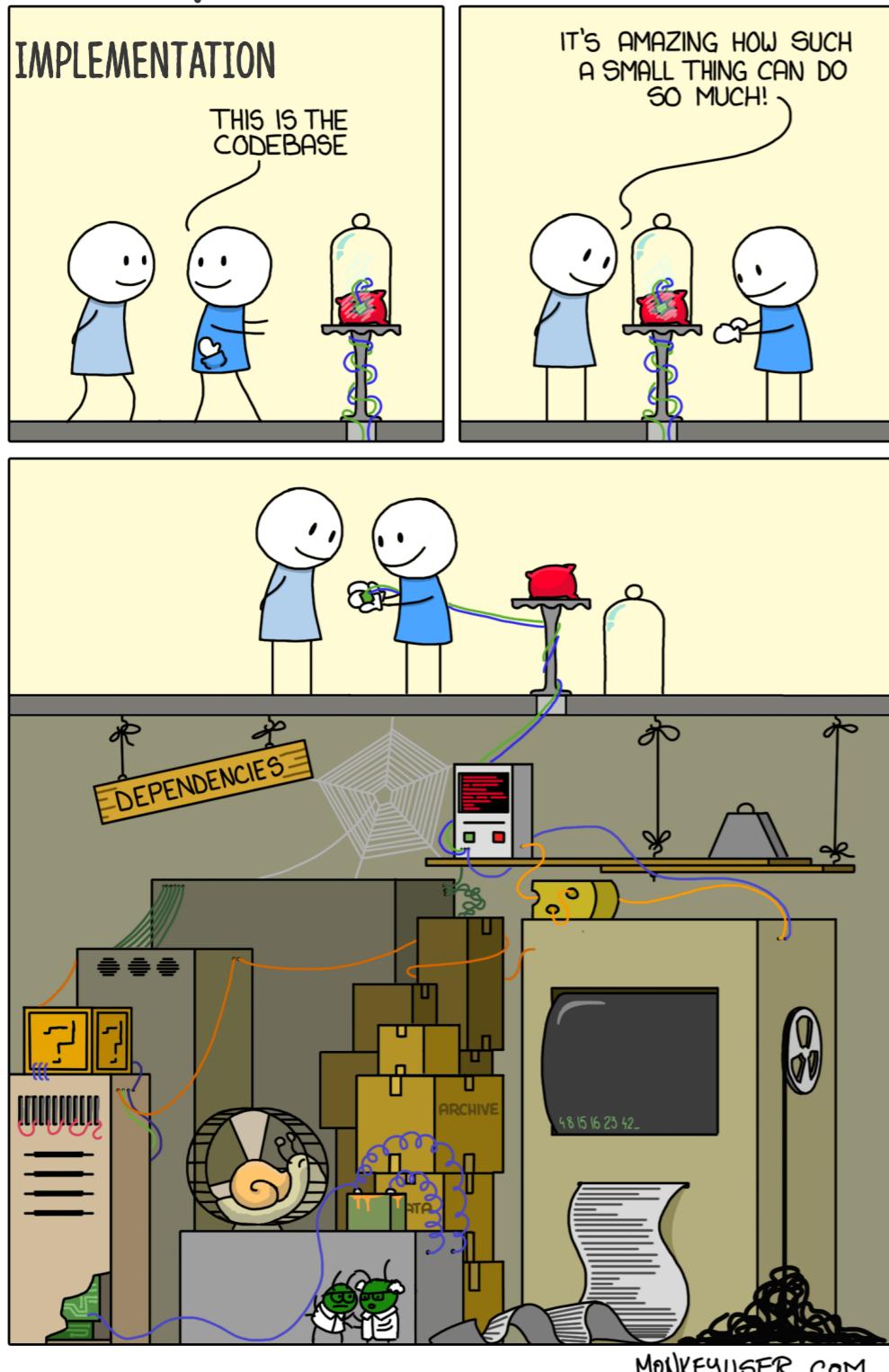
La utilizaremos en la siguiente práctica!!!...

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 S04: 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

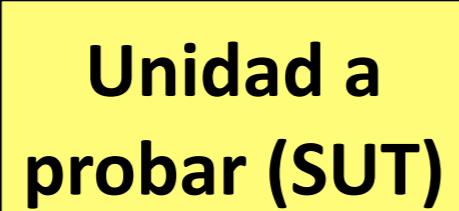
Verificación

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



casos de prueba

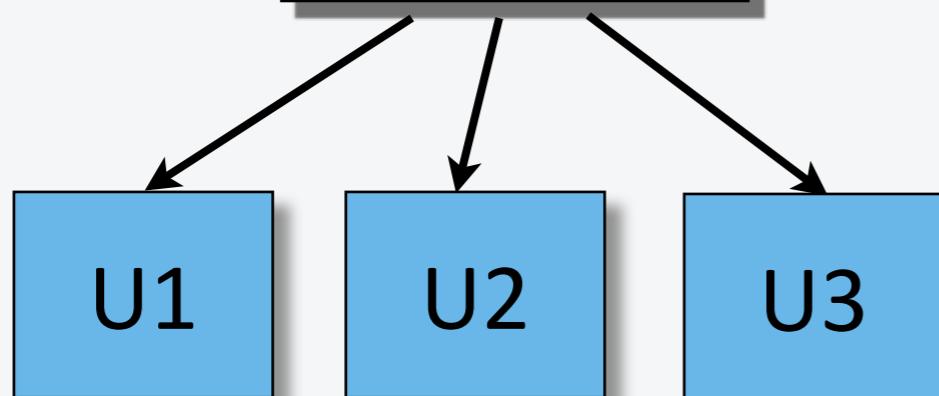
(datos de prueba + resultado esperado)



informes de pruebas

SUT : Código que queremos probar.

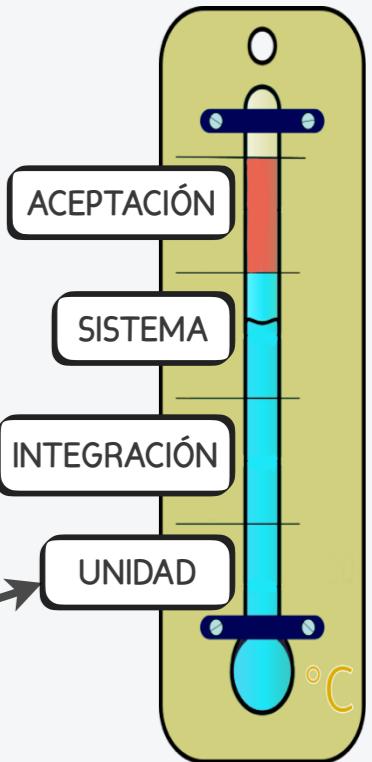
SUT= método Java



¿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)

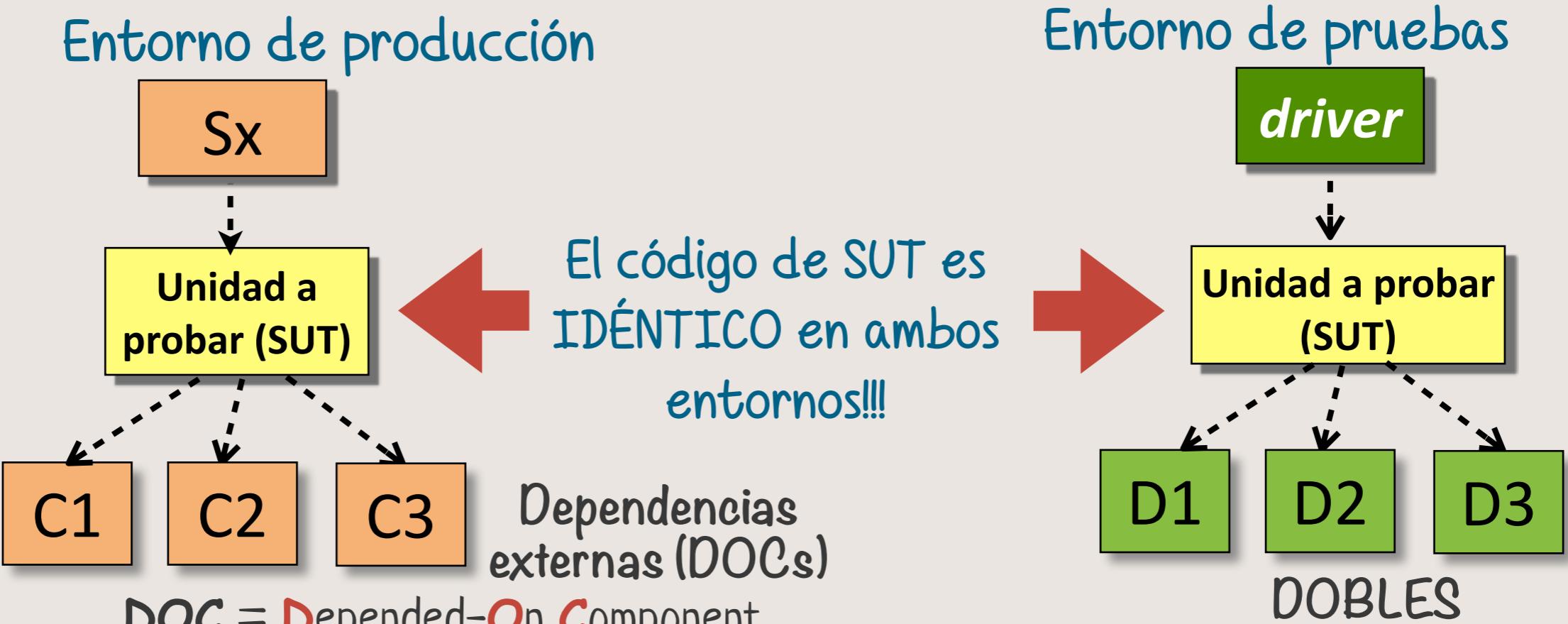
```
public class GestorPedidos {  
    private Factura factura;  
  
    1. public Facturas generarFacturas() {  
    2.     //... código  
    3.     //boolean ok = facturas.pendientes();  
    4.     ok = true;  
    5.     if (ok) {  
    6.         //sentencias then  
    7.     } else {  
    8.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

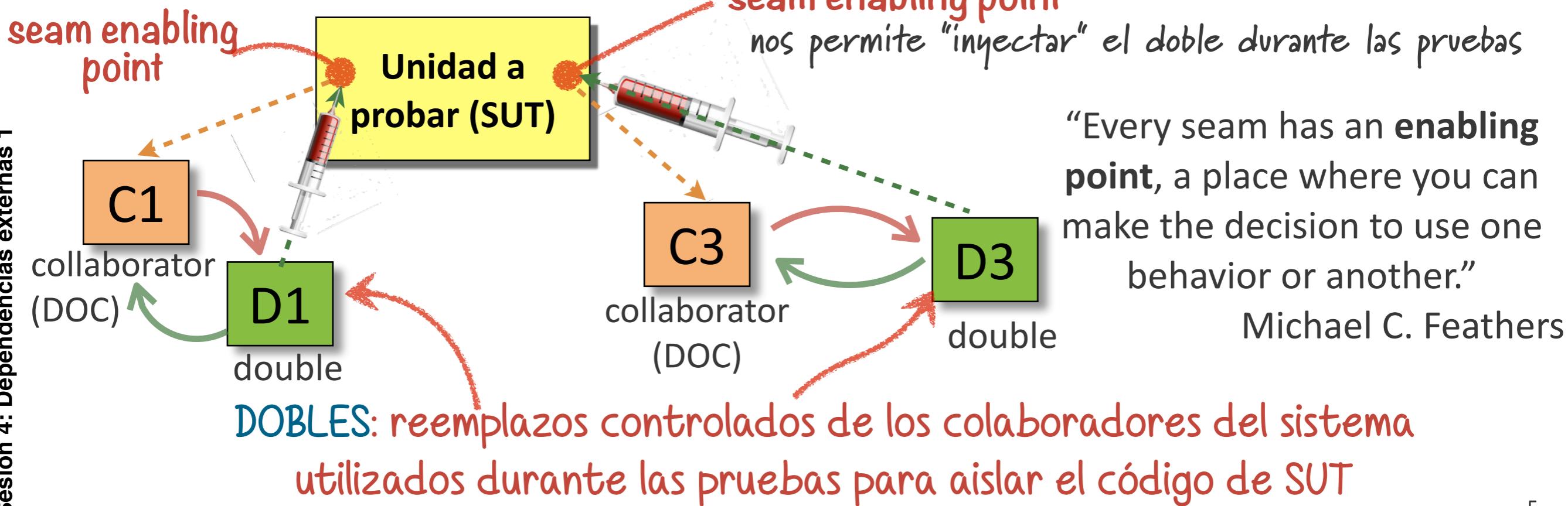
<http://www.informit.com/articles/article.aspx?p=359417&seqNum=3>

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 la 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

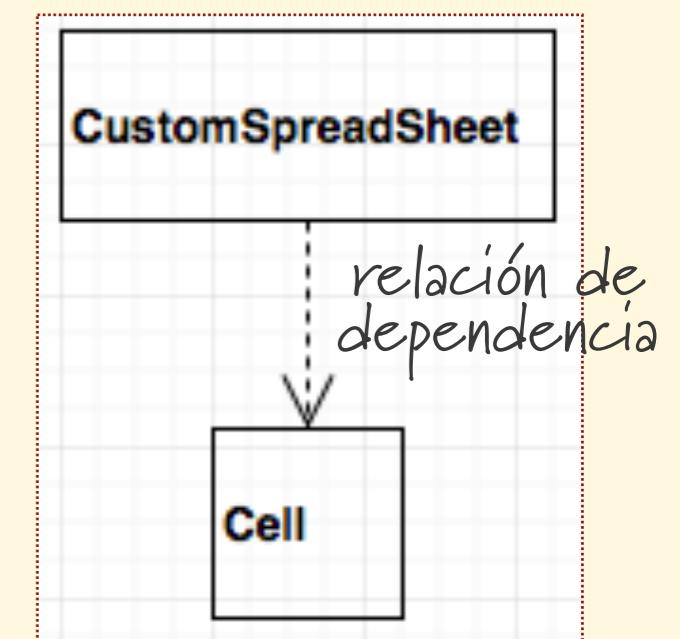
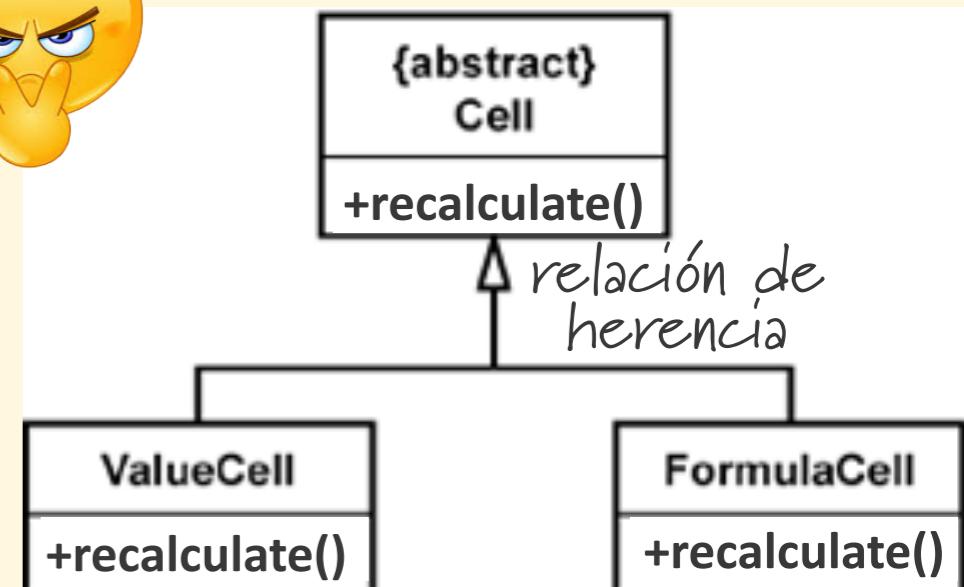
Sesión 4: Dependencias externas 1

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:

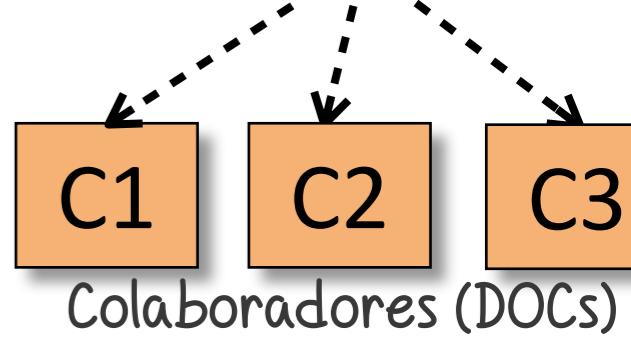
nos interesa, además, verificar que las interacciones entre nuestro SUT y las dependencias externas se realizan correctamente

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!!!

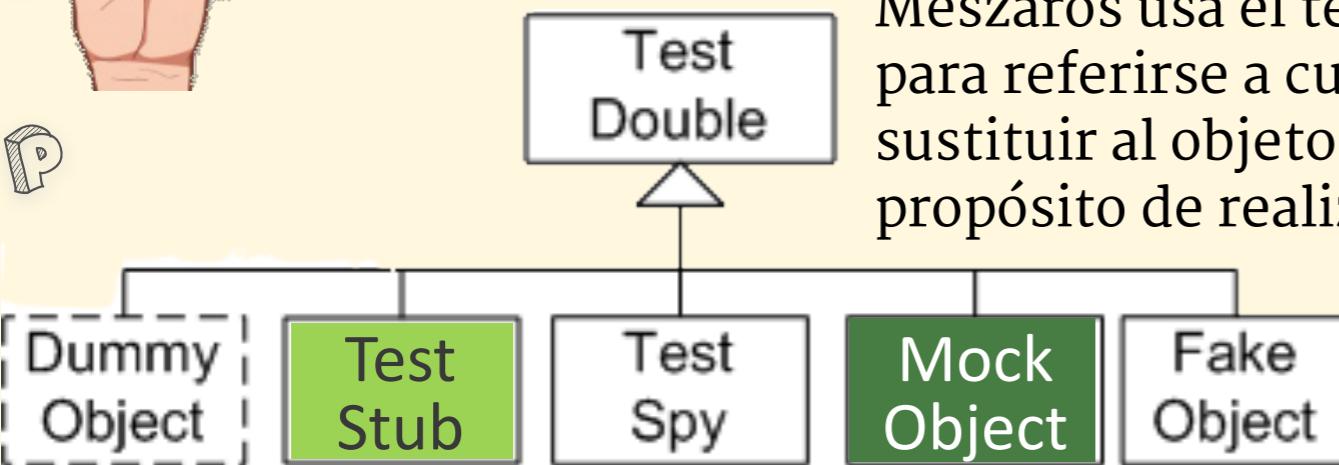


IMPLEMENTAMOS EL DOBLE



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

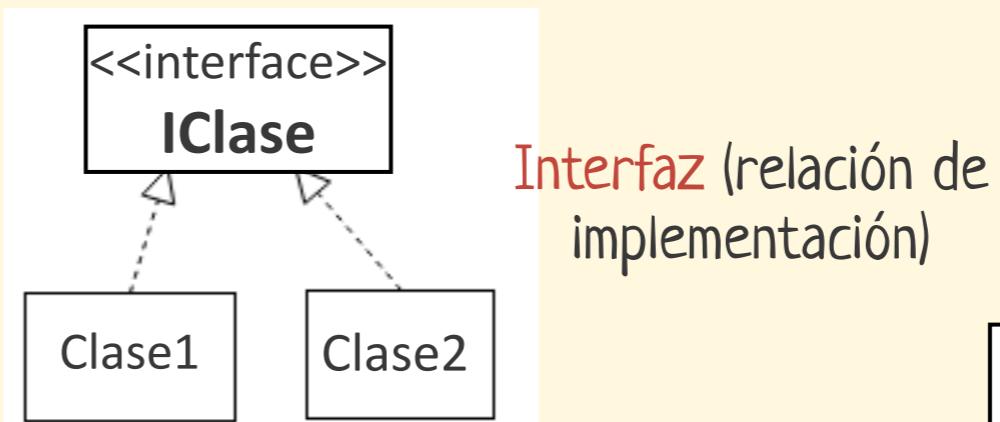
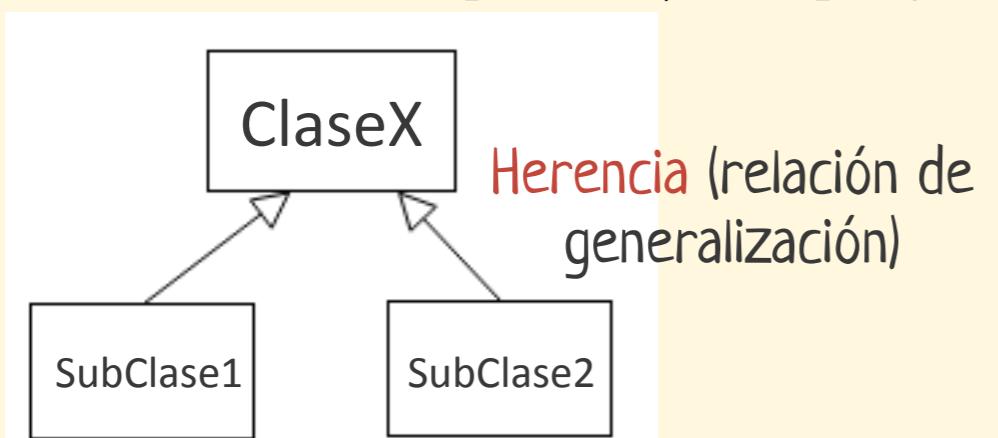
S nuestro doble pertenecerá a una clase que "extienda" o "implemente" la clase que contiene nuestro SUT

**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



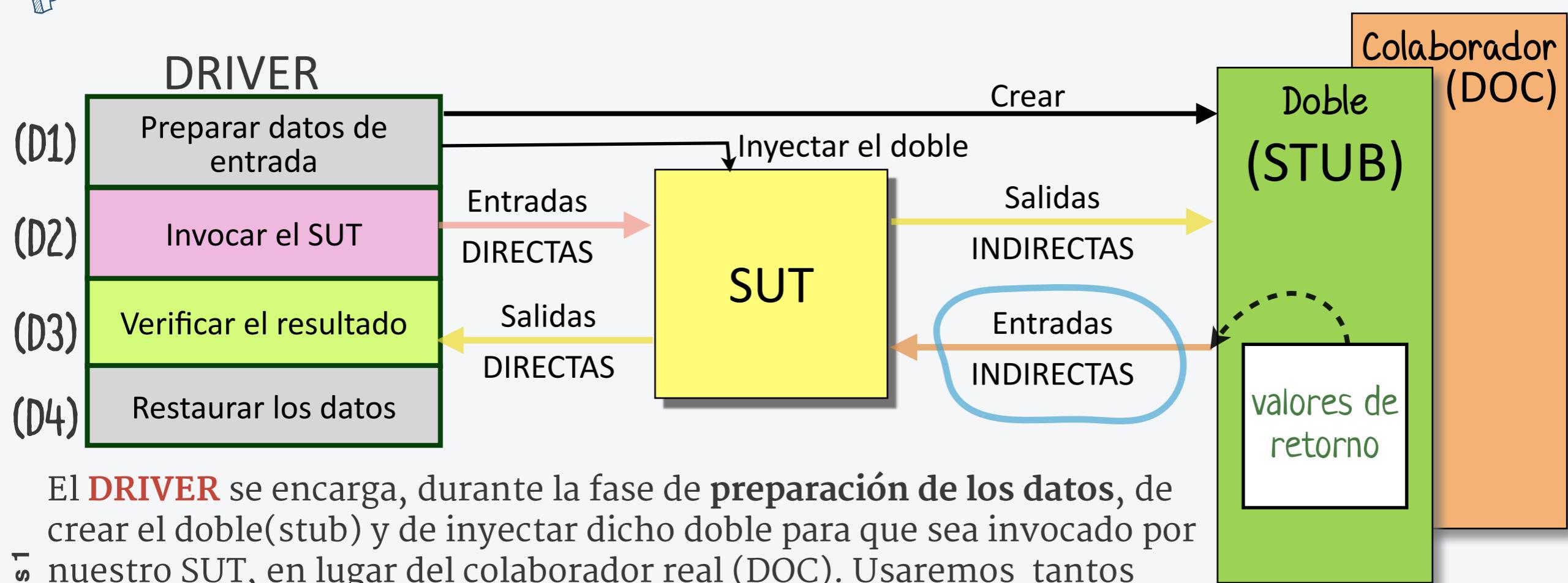
La implementación del STUB
debe ser GENÉRICA. Debe servir para todos los casos de prueba de nuestro SUT (sólo un doble por tabla) !!!

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();
        BuscadorStub 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!

Sesión 4: Dependencias externas

```
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();  
        BuscadorStub 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.         float discountedPrice = order.getProduct().getPrice()  
12.             * (1 - (discountPercentage / 100));  
13.         order.setBalance(discountedPrice);  
14.     }  
15. }  
16.  
17.
```

SUT TESTABLE!!!

SUT

DOC

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));  
    }  
}
```

en src/test

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...

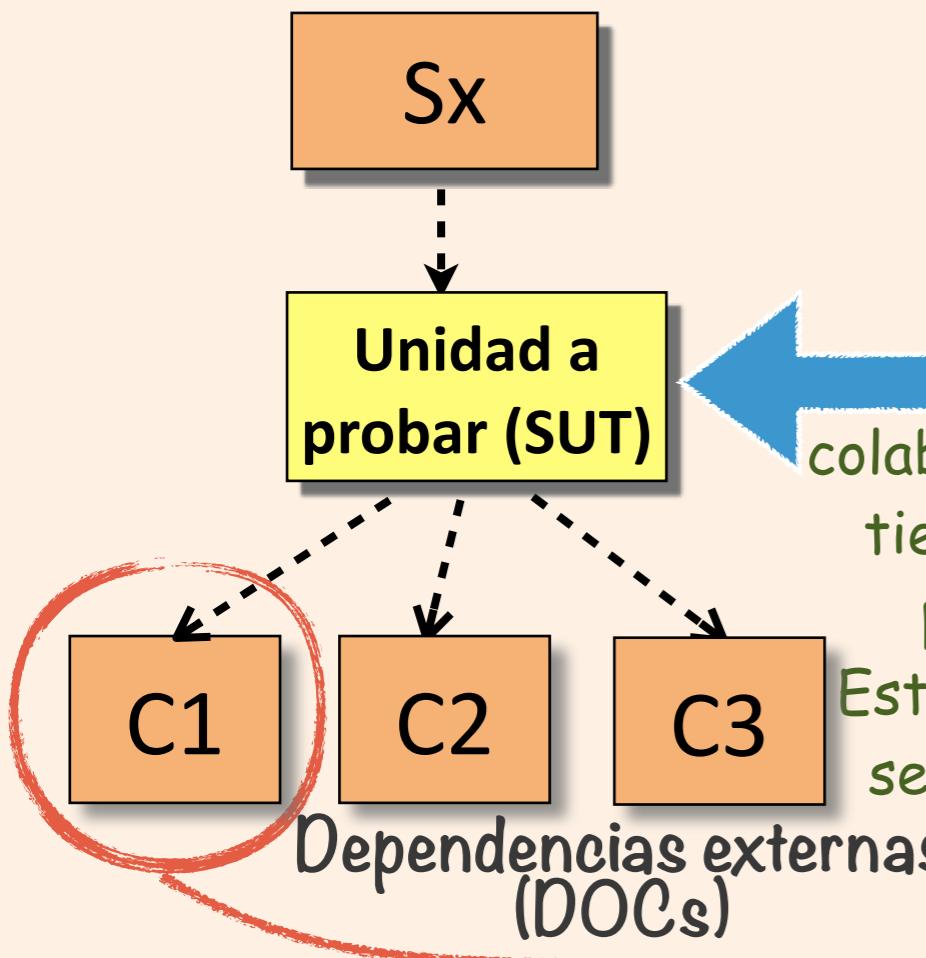
P

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

P

src/main/java

Entorno de producción



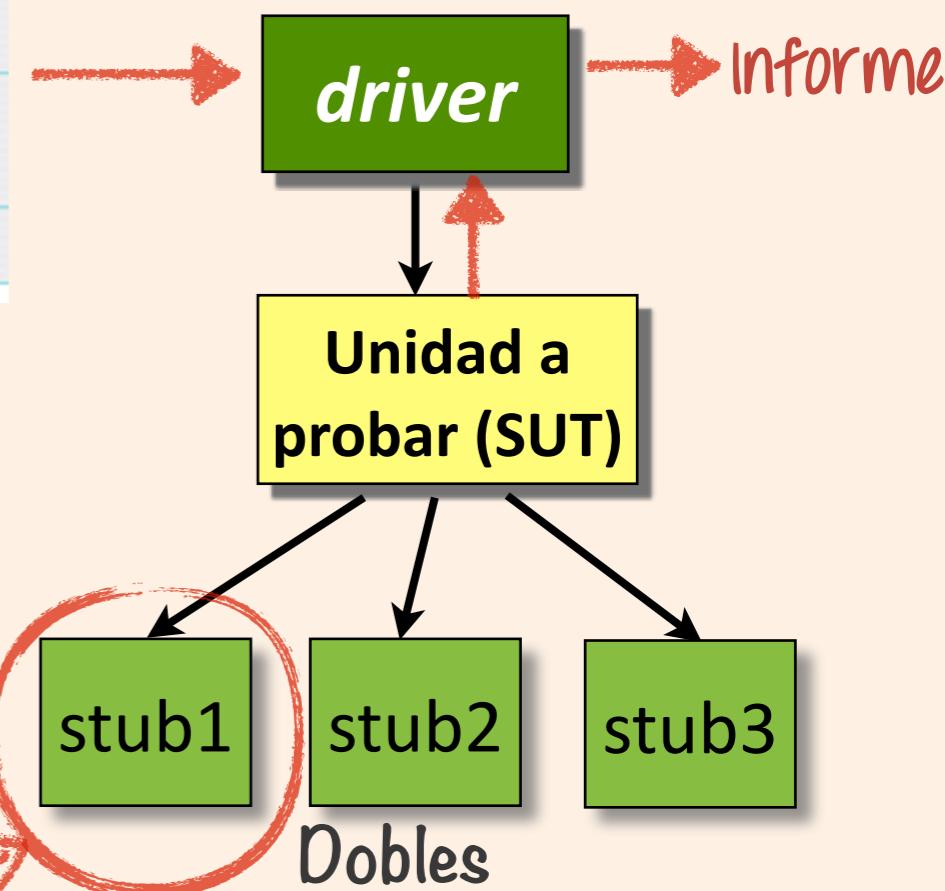
Casos de prueba

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

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

src/test/java

Tests Unitarios



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

PPSS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2020-21

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:

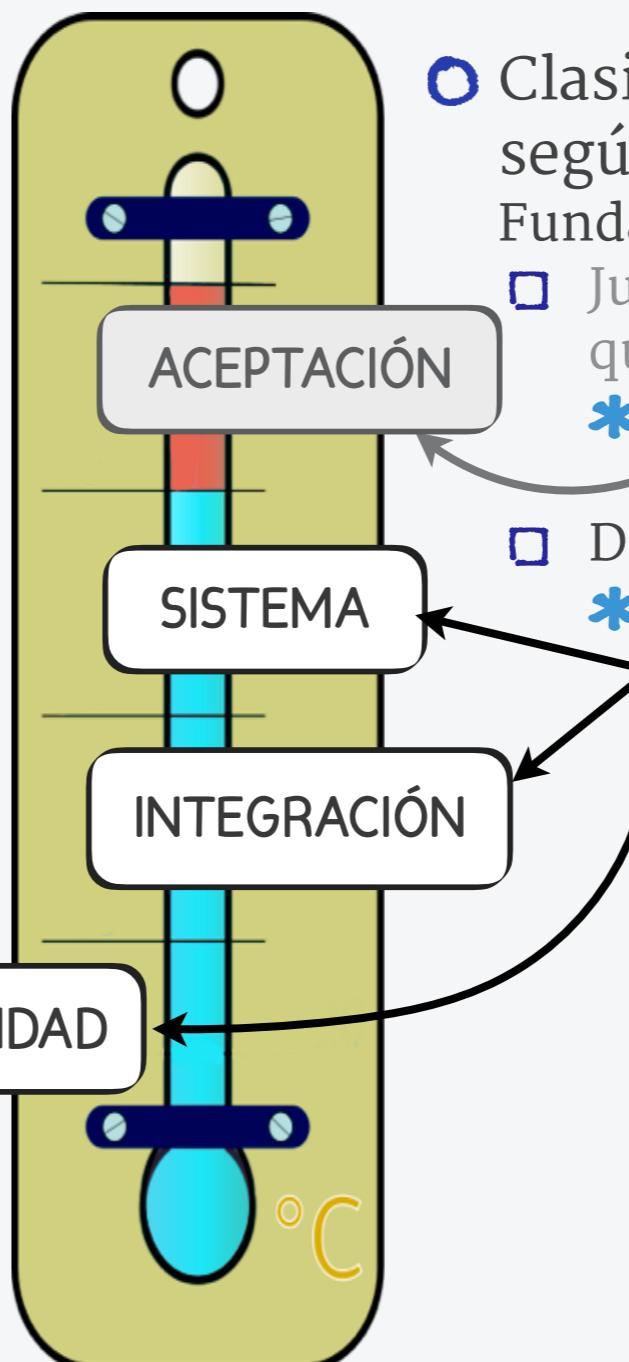
- Creación de los mocks
- Programación de expectativas
- Comunicar al framework que los mocks están listos para ser invocados
- Verificar las expectativas de los mocks

Vamos al laboratorio...

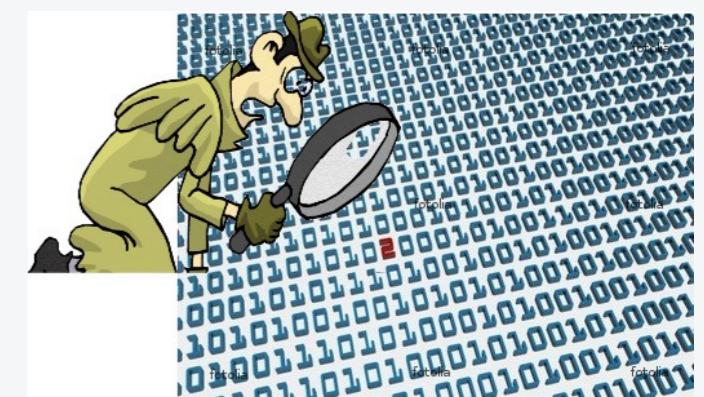
VERIFICACIÓN Y PRUEBAS UNITARIAS

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!!!

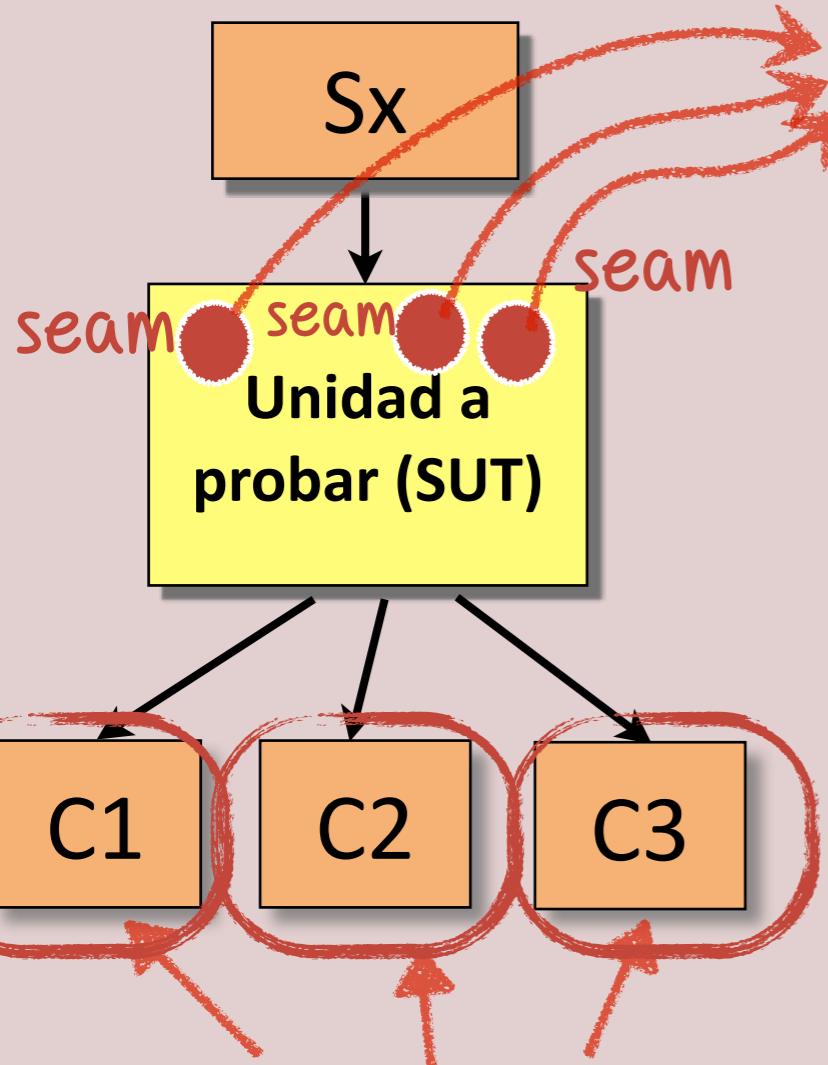
YA HEMOS VISTO CUÁL ES EL PROCESO PARA AUTOMATIZAR PRUEBAS UNITARIAS

P

P

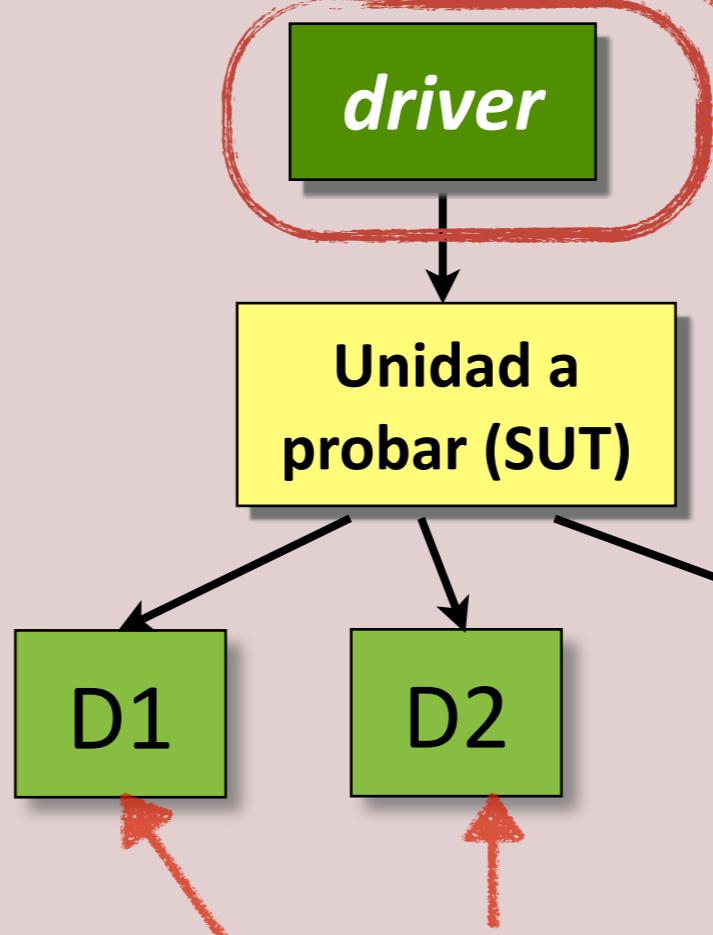
P

(1) Identificar las dependencias externas



(2) Conseguir que nuestro código sea testable

(4) Implementar los drivers



Verificación basada en el estado (stub)

Verificación basada en el comportamiento (mock)

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

La implementación de los dobles la hemos hecho "manualmente"

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 EASYSMOK

P

P

SUT

}

```
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;
        }
    }
}
```



SUT TESTABLE!!!

prototipo del método getPortion()

public boolean getPortion(CoffeeCupType coffeeCupType) throws NotEnoughException

```
public class CoffeeMachineEasyMockTest {
    CoffeeMachine coffeeMachine;
    Container coffeeContainerStub;
    Container waterContainerStub;
```

DRIVER

```
@BeforeEach
public void setup() {
    coffeeContainerStub = EasyMock.niceMock(Container.class);
    waterContainerStub = EasyMock.niceMock(Container.class);
    coffeeMachine = new CoffeeMachine(coffeeContainerStub, waterContainerStub);
}
```

STUBS

inyectamos los dobles

```
@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)));
}
```

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**

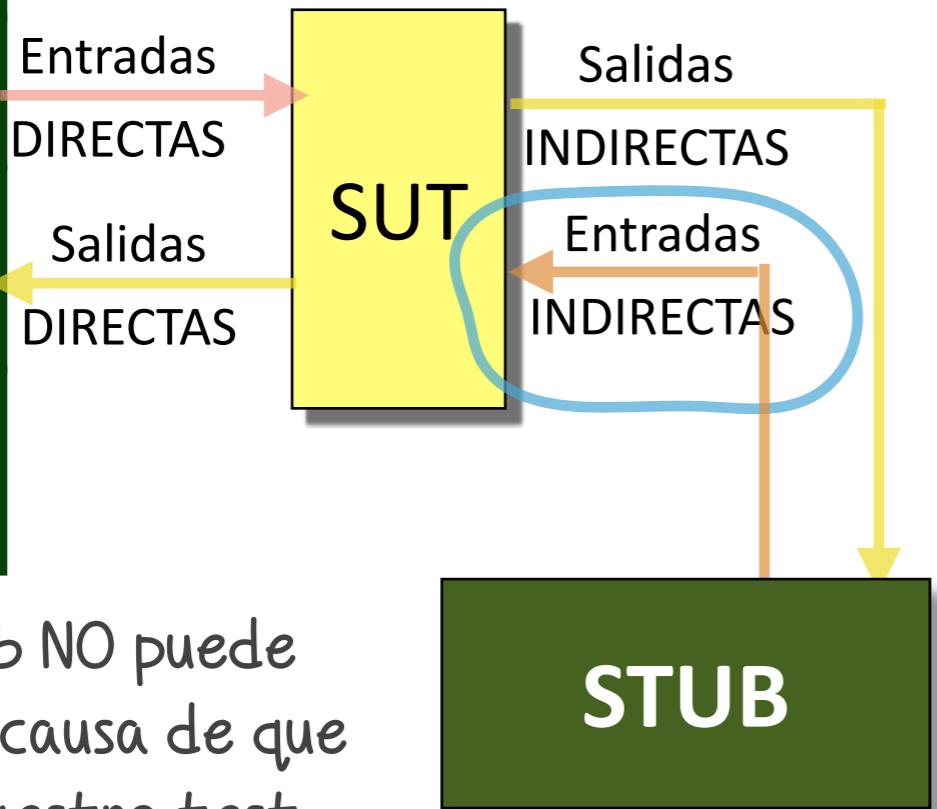
DRIVER

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

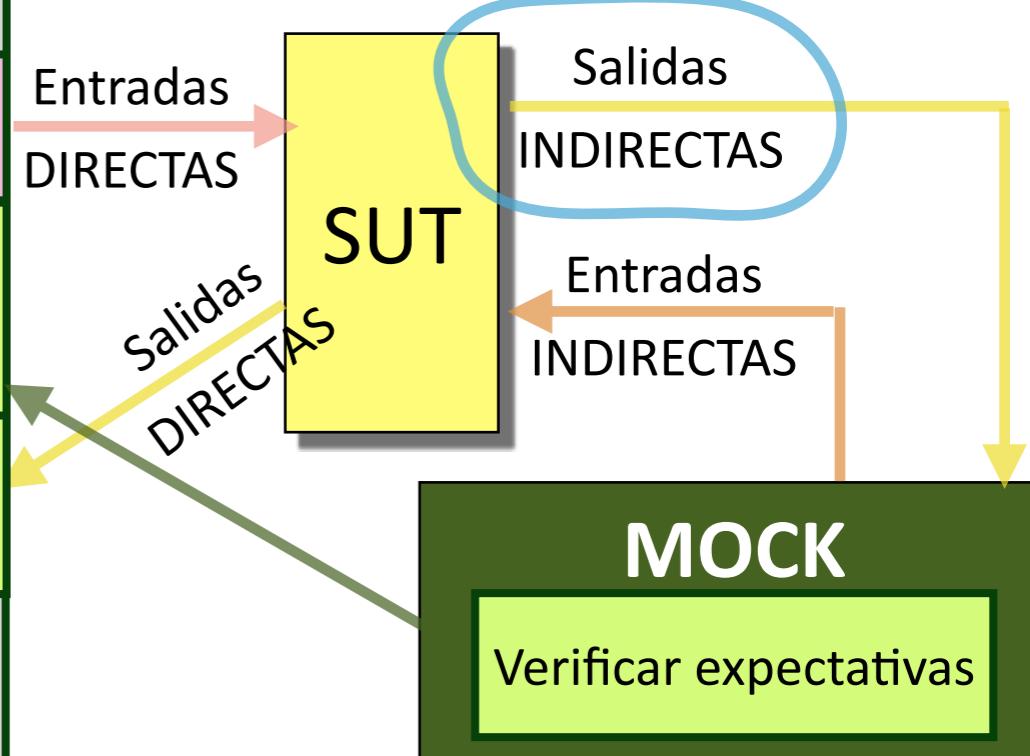
Invoker el SUT
Entradas DIRECTAS

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

EASYSOCK 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)

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)

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 implementaos 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**

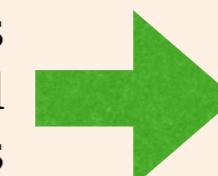
Ejemplo con varios objetos StrictMock.

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

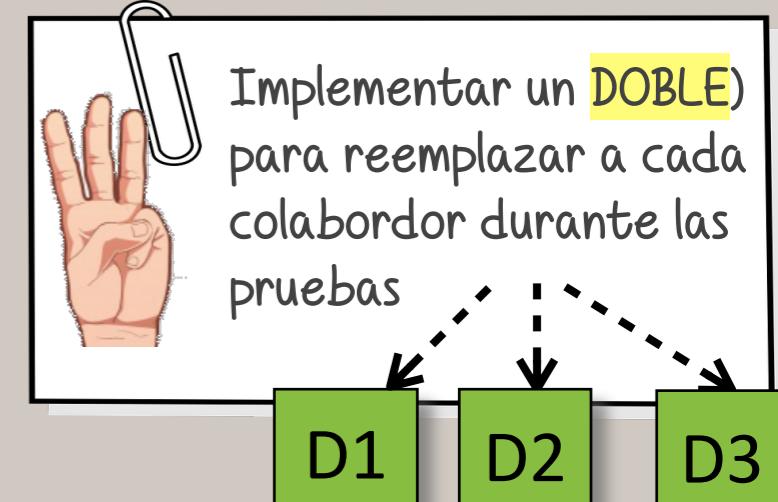
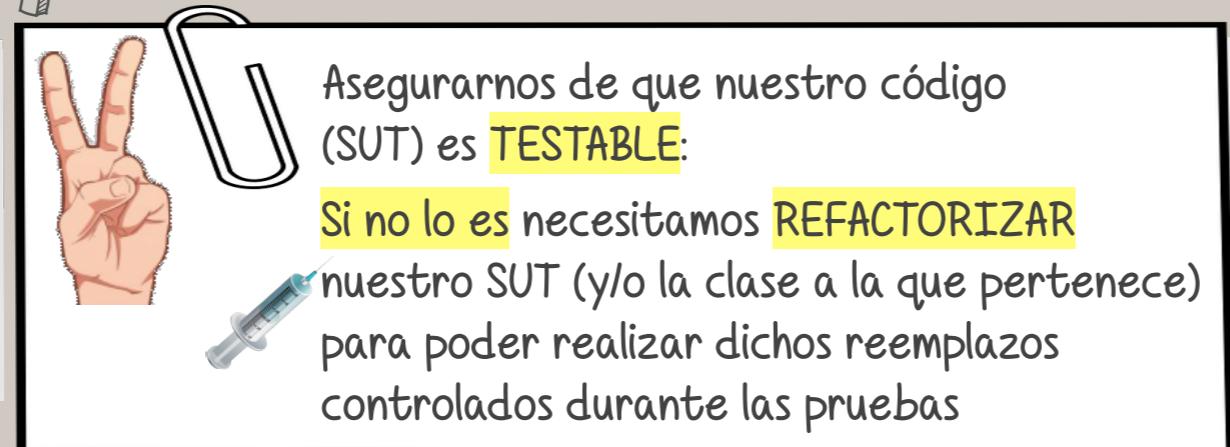
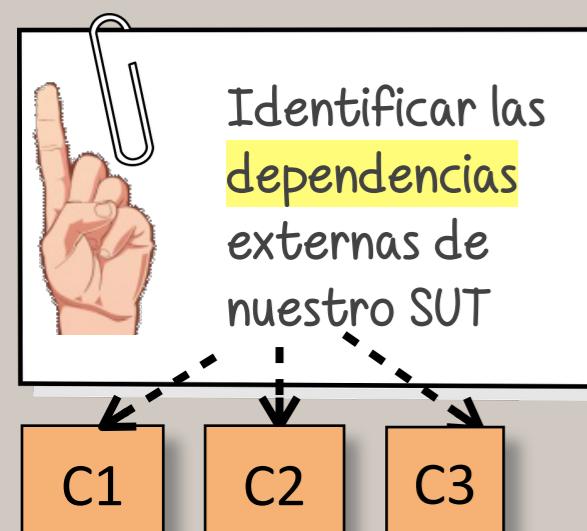
Verificación basada en el comportamiento

- 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



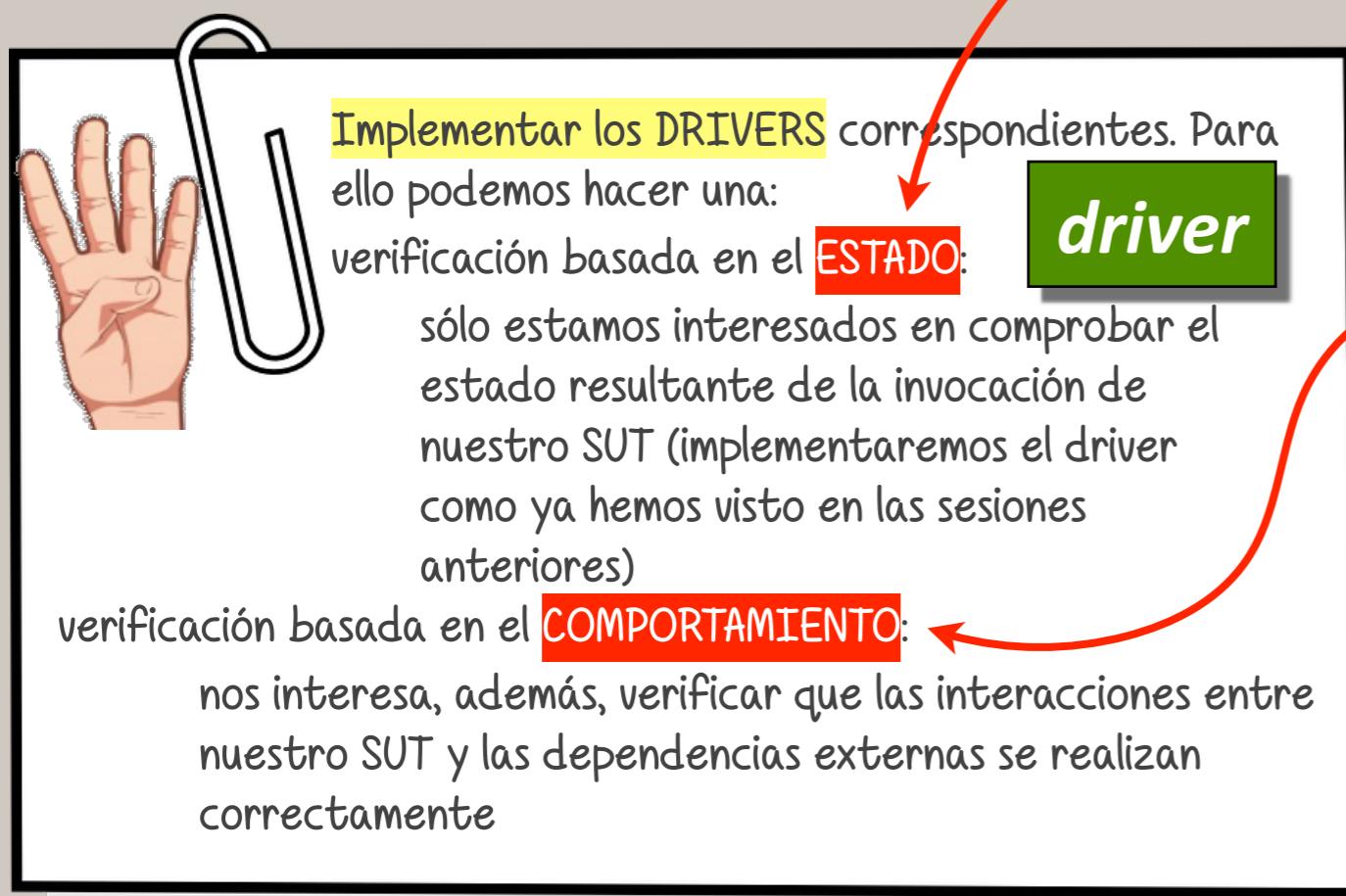
- Si nuestras dependencias externas no proporcionan entradas indirectas al SUT que debamos controlar, nuestro doble no será un stub, usaremos un mock y por tanto tendremos que utilizar verificación basada en el comportamiento si queremos comprobar que el doble ha sido invocado desde el SUT
- Si los colaboradores 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);  
    }  
}
```



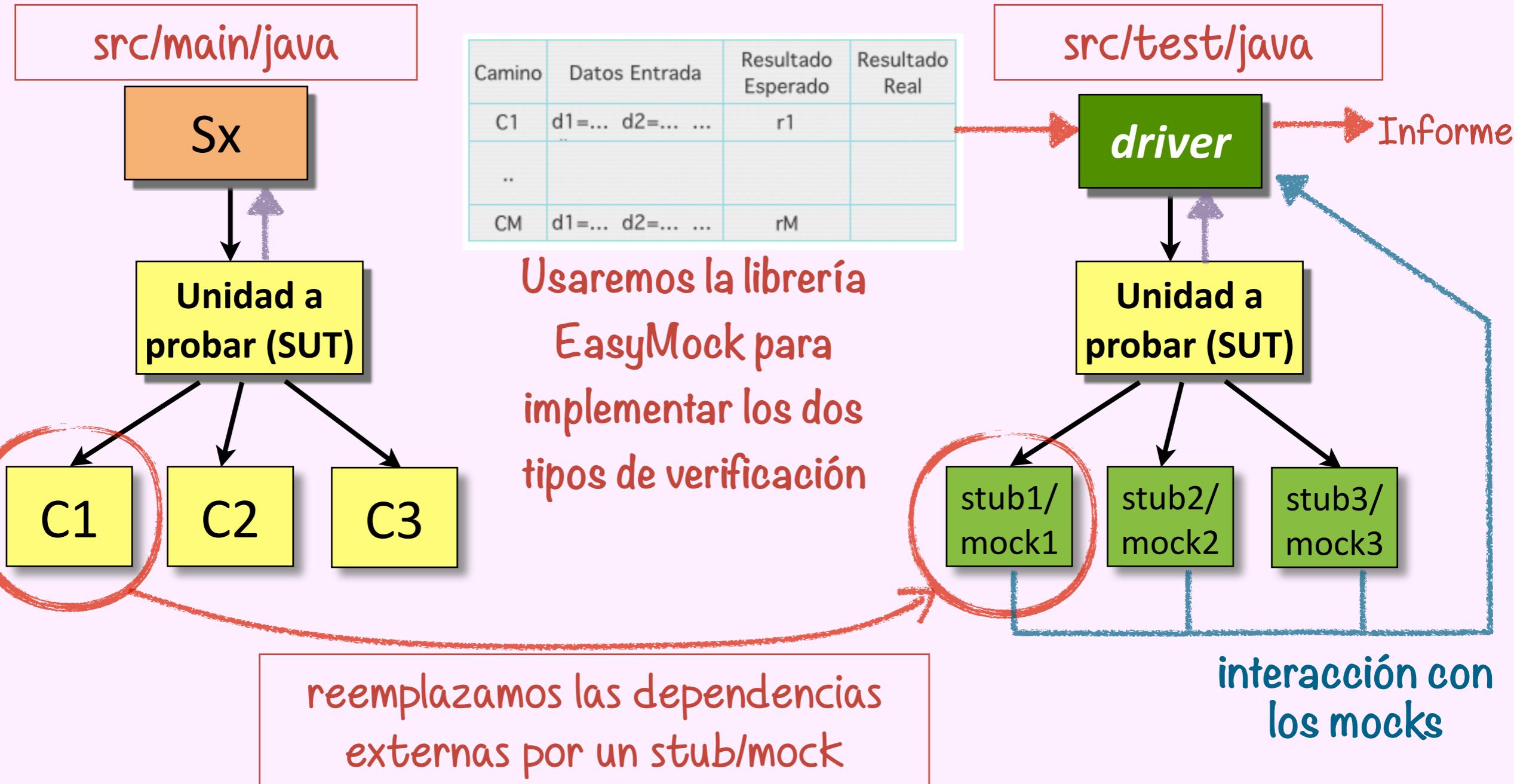
Cuando termines el ejercicio debes tener claro:

- 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

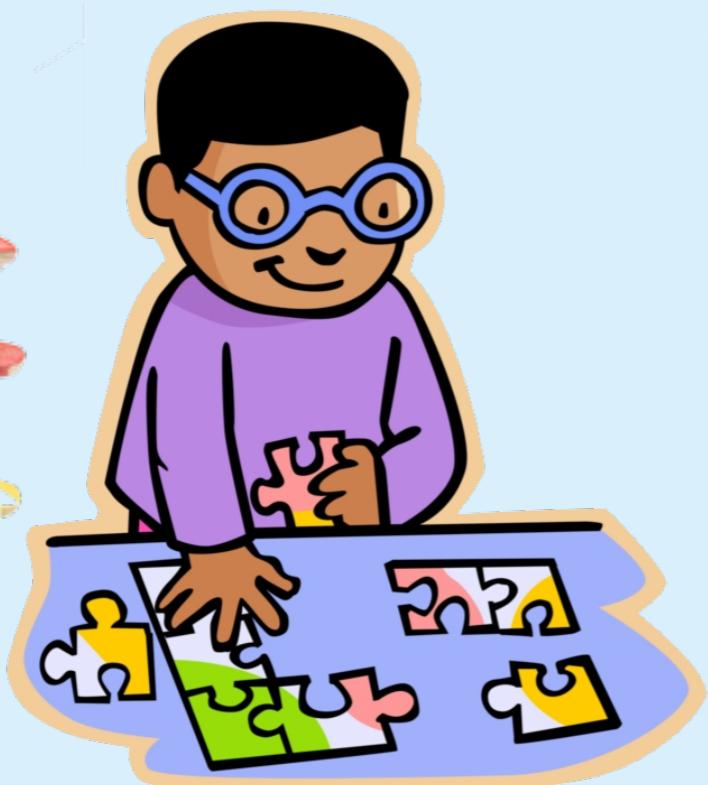
- The art of unit testing: with examples in C#. 2nd edition Roy Osherove. Manning, 2014.
 - Capítulo 4. Interaction testing using mock objects
- Easier testing with EasyMock. Elliotte Rusty Harold
 - <http://www.ibm.com/developerworks/java/library/j-easymock/index.html>
- Mocks aren't stubs. Martin Fowler. 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>

Sesión S06:

Pruebas de integración



unidades



Objetivos de las pruebas de integración

- Encontrar defectos en las INTERFACES de las unidades
- SUT = conjunto de unidades

Diseño de pruebas de integración

Estrategias de integración

- Determinan el ORDEN en el que se van a integrar las unidades
- Se trata de un proceso INCREMENTAL en el que las pruebas de REGRESIÓN son fundamentales

Integración con una base de datos

Automatización de las pruebas con DbUnit

Maven y pruebas de integración

Ejemplo de integración

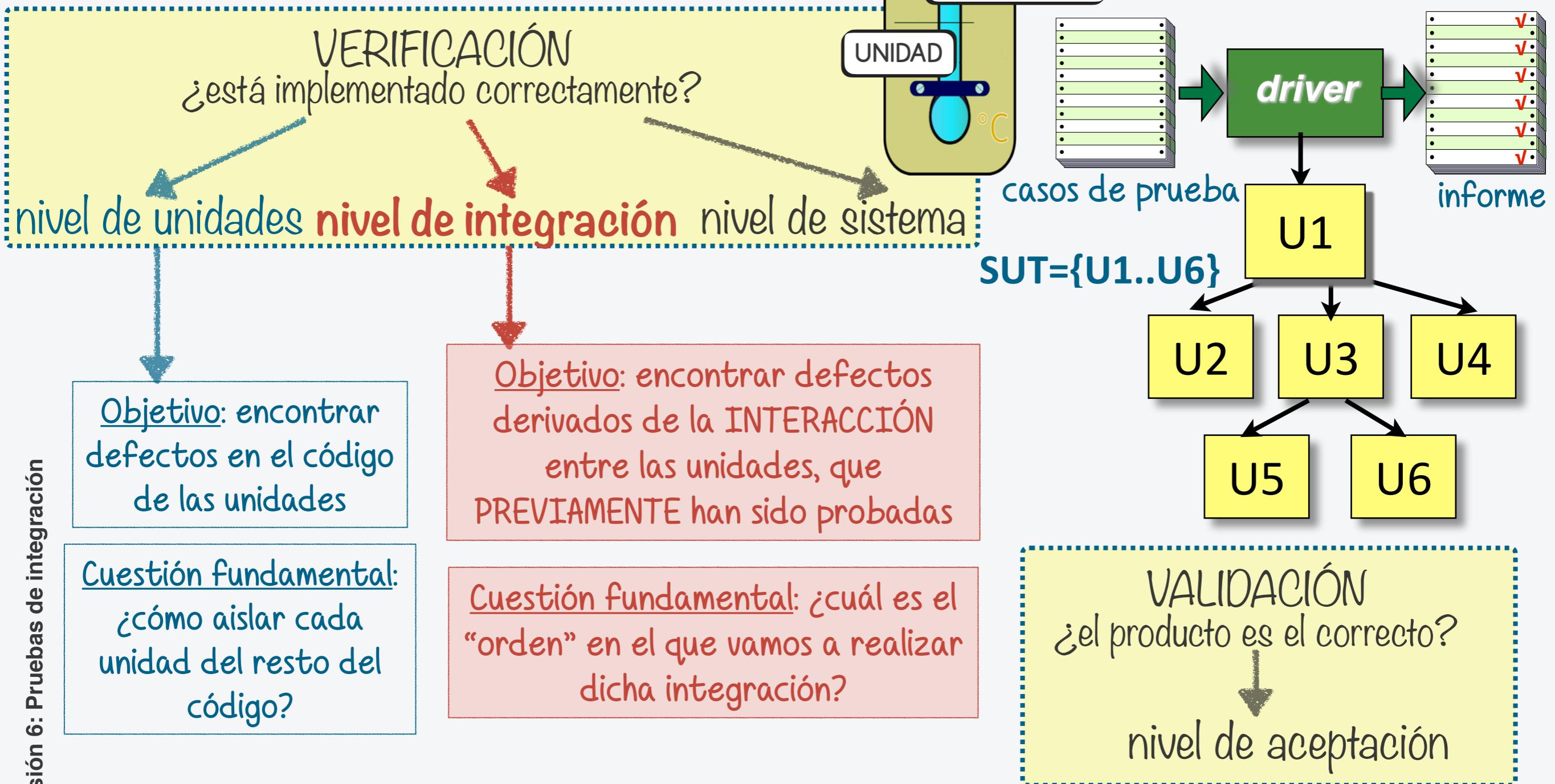
Vamos al laboratorio...

NIVELES DE PRUEBAS

P

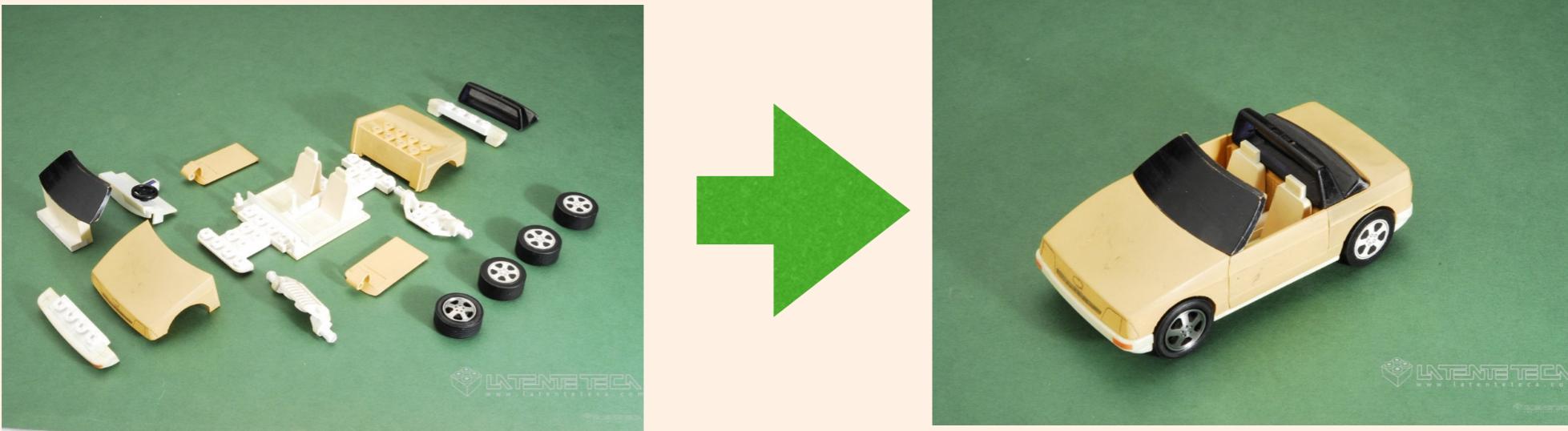
- Las pruebas se realizan a diferentes niveles, durante el proceso de desarrollo

P



P IMPORTANCIA DE LAS PRUEBAS DE INTEGRACIÓN

- A nivel de pruebas unitarias, el sistema "existe" en forma de "piezas" bajo el control de los programadores
- La siguiente tarea importante es "reunir" todas las "piezas" para construir el sistema completo
 - Un sistema es una colección de "componentes" interconectados de una determinada forma para cumplir un determinado objetivo



- Construir el sistema completo a partir de sus "piezas" no es una tarea fácil debido a los numerosos errores sobre las INTERFACES
 - A pesar de esforzarnos en realizar un buen diseño y documentación, las malinterpretaciones, errores, y descuidos son una realidad
 - Los errores de interfaz entre los diferentes componentes son provocados fundamentalmente por los programadores

EL PROCESO DE INTEGRACIÓN

P

P

- El objetivo de la integración del sistema es construir una versión "operativa" del sistema mediante:
 - la agregación, de forma **incremental**, de nuevos componentes,
 - asegurándonos de que la adición de nuevos componentes no "perturba" el funcionamiento de los componentes que ya existen (**pruebas de regresión**)
- Las pruebas de integración del sistema constituyen un proceso sistemático para "ensamblar" un sistema software, durante el cual se ejecutan pruebas conducentes a descubrir errores asociados con las **interfaces** de dichos componentes
 - Se trata de garantizar que los componentes, sobre los que previamente se han realizado pruebas unitarias, funcionan correctamente cuando son "combinados" de acuerdo con lo indicado por el **diseño**



LINENTETCA
LINEA INTEGRATIVA CA

TIPOS DE INTERFACES Y ERRORES COMUNES

ver Bibliografía: Ian Sommerville, 9th ed. Cap. 8.1.3

- La interfaz entre componentes (unidades, módulos), puede ser de varios tipos:
 - **Interfaz a través de parámetros**: los datos se pasan de un componente a otro en forma de parámetros. Los métodos de un objeto tienen esta interfaz
 - **Memoria compartida**: se comparte un bloque de memoria entre los componentes. Los componentes escriben datos en la memoria compartida, que son leídas por otros
 - **Interfaz procedural**: un componente encapsula un conjunto de procedimientos que pueden llamarse desde otros componentes. Por ejemplo, los objetos tienen esta interfaz
 - **Paso de mensajes**: un componente A prepara un mensaje y lo envía al componente B. El mensaje de respuesta del componente B incluye los resultados de la ejecución del servicio. Por ejemplo, los servicios web tienen esta interfaz
- Errores más comunes derivados de la interacción de los componentes a través de sus interfaces:
 - Mal uso de la interfaz
 - Malentendido sobre la interfaz
 - Errores temporales
- Las pruebas para detectar defectos en las interfaces son difíciles, ya que algunos de ellos pueden sólo manifestarse bajo condiciones inusuales!!!

GUÍAS GENERALES PARA DISEÑAR LAS PRUEBAS

- Examinar el código a probar y listar de forma explícita cada llamada a un componente externo. Diseñar un conjunto de pruebas con los valores de los parámetros a componentes externos en los **extremos de los rangos**. Estos valores pueden revelar inconsistencias de la interfaz con una mayor probabilidad
- Si se pasan punteros a través de la interfaz, siempre probar con punteros nulos
- Cuando se invoca a un componente con una **interfaz procedural**, diseñar tests que provoquen, de forma deliberada, que el componente falle. Diferentes asunciones sobre los fallos son uno de los malentendidos sobre la especificación más comunes
- Utilizar pruebas de estrés en sistemas con **paso de mensajes**. Esto implica que deberíamos diseñar los tests de forma que se generen más mensajes de los que probablemente ocurran en la práctica. Esta es una forma efectiva de revelar problemas temporales
- Cuando varios componentes interaccionan con **memoria compartida**, diseñaremos los tests en el orden en los que estos componentes son activados. De esta forma revelaremos asunciones implícitas hechas por el programador sobre el orden en el que los datos son producidos y consumidos

Usaremos algún método de diseño de caja negra (p.ej. particiones equivalentes)

ESTRATEGIAS DE INTEGRACIÓN

P

 YouTube ES [WHAT IS INTEGRATION TESTING](#)

S

Establecen el **ORDEN** en el que se van a integrar las unidades probadas
componentes = unidades

P

- **Big Bang**: una vez realizadas las pruebas unitarias, se integran TODAS las unidades (a la vez)
 - Sólo si el tamaño de nuestra aplicación es muy "pequeño" o tiene pocas unidades
- **Top-down**: integramos primero los componentes con mayor nivel de abstracción, y vamos añadiendo componentes cada vez con menor nivel de abstracción
 - Necesitamos implementar dobles
 - Adecuado para sistemas con una interfaz de usuario "compleja"
- **Bottom-up**: integramos primero los componentes de infraestructura que proporcionan servicios comunes, como p.ej. acceso a base de datos, acceso a red,... y posteriormente añadimos los componentes funcionales, cada vez con mayor nivel de abstracción
 - Necesitamos implementar muchos menos dobles
 - Adecuado para sistemas con una infraestructura y/o lógica de negocio "compleja"
- **Sandwich**: es una mezcla de las dos estrategias anteriores
- **Dirigida por los riesgos**: se eligen primero aquellos componentes que tengan un mayor riesgo (p.ej. aquellos con más probabilidad de errores por su complejidad)
- **Dirigida por las funcionalidades** (casos de uso, historias de usuario...)/**hilos de ejecución**: se ordenan las funcionalidades con algún criterio y se integra de acuerdo con este orden

El ORDEN de integración es muy importante!!
Elegiremos uno u otro dependiendo del TIPO de aplicación

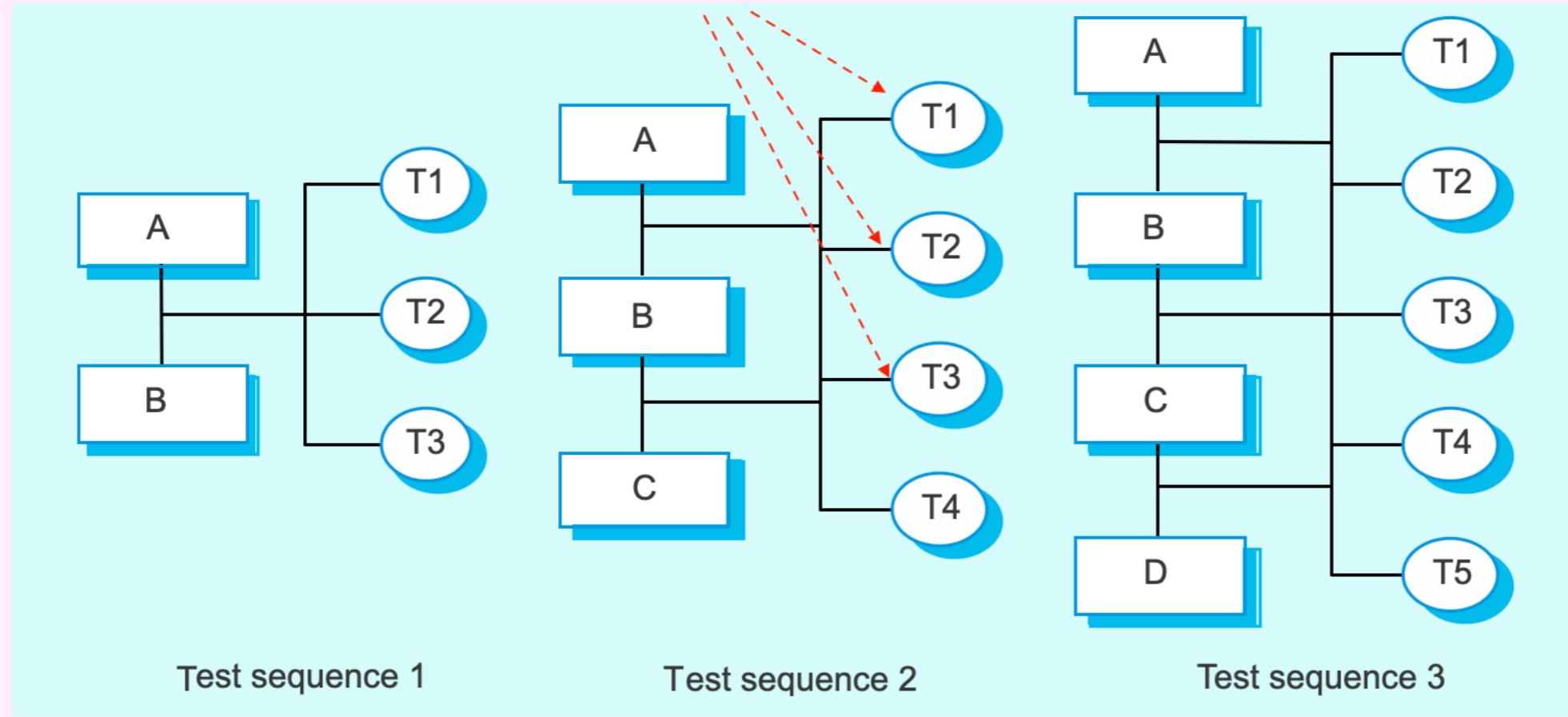


P INTEGRACIÓN Y PRUEBAS DE REGRESIÓN S

Los últimos componentes (unidades) integrados son siempre los MENOS PROBADOS!!!

- Las pruebas de **REGRESIÓN** consisten en repetir las pruebas realizadas cuando integramos un nuevo componente

PRUEBAS DE REGRESIÓN



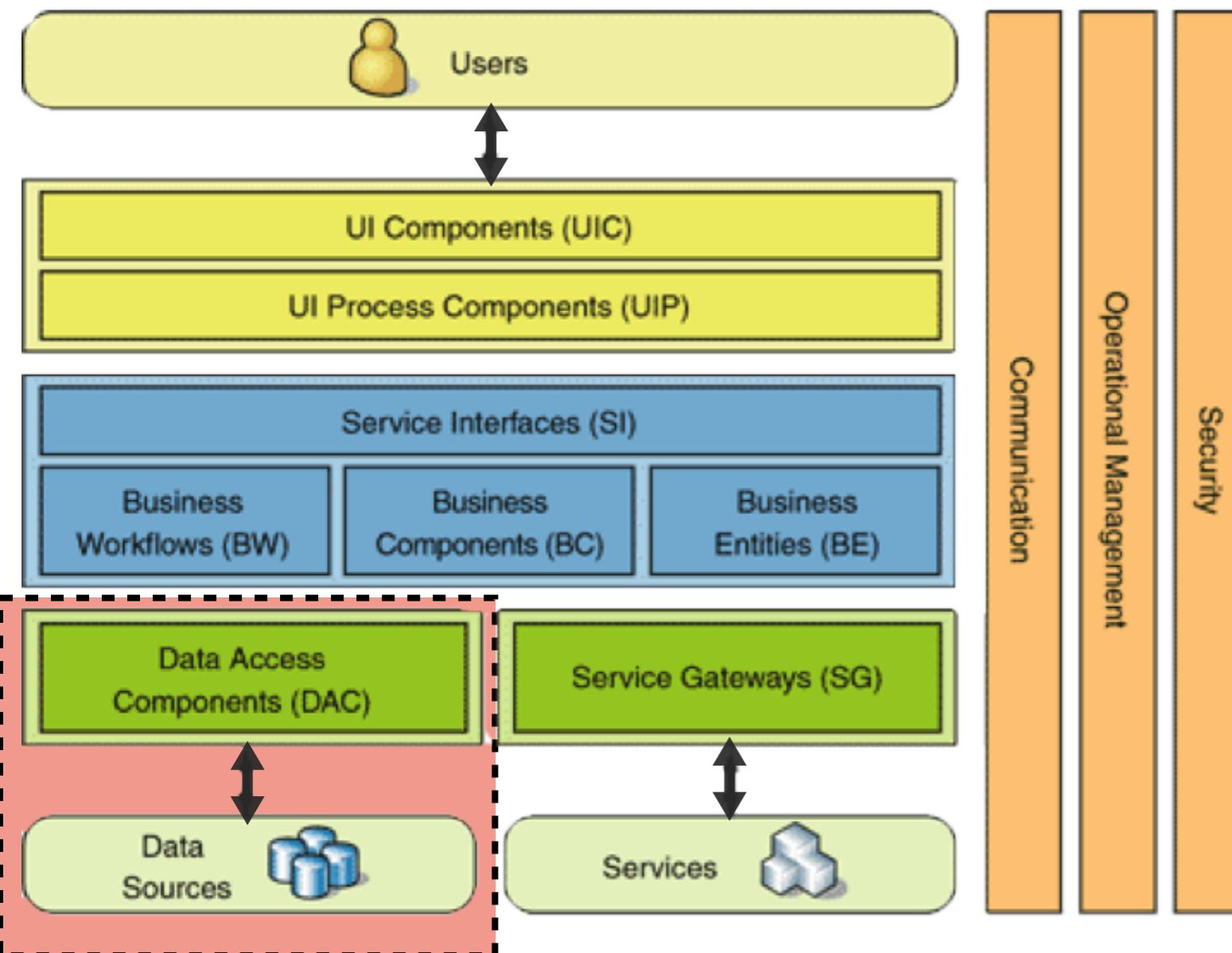
A,B,..,D= Componentes a probar; T1..Tn= Tests

Las pruebas de regresión no son necesarias únicamente cuando estamos haciendo pruebas de integración, son indispensables cuando modificamos el software durante la fase de mantenimiento, o añadimos una nueva funcionalidad. También son necesarias cuando depuramos algún defecto. Las pruebas de regresión nos permiten comprobar que "todo lo que funcionaba correctamente" antes de hacer los cambios sigue funcionando, de forma que no hemos "roto" nada al introducir dichos cambios en el código.

INTEGRACIÓN CON UNA BASE DE DATOS

- La mayor parte de las aplicaciones de empresa utilizan una Base de Datos (BD) como mecanismo de persistencia

- Una arquitectura software típica de aplicaciones de empresa es una arquitectura por capas, en donde la BD se sitúa entre las capas inferiores



- Las pruebas de integración para dichas aplicaciones requieren que existan datos en la BD para funcionar correctamente
 - ¿Cómo podemos realizar pruebas de integración sobre las clases que dependen directamente de dicha BD?
 - ¿Cómo podemos asegurarnos de que nuestro código está realmente leyendo y/o escribiendo los datos correctos de dicha BD?
- * La respuesta es... Utilizando **DbUnit**

¿QUÉ ES DBUNIT?

Componentes principales del API:

- `IDatabaseTester` → `JdbcDatabaseTester`
- `IDatabaseConnection`
- `DatabaseOperation`
- `IDataSet` → `FlatXmlDataSet`
- `ITable`
- `Assertion`



DBUnit **NO** sustituye a la Base de Datos, sólo nos permite **CONTROLAR** su estado previo y verificar su estado después de la invocación de nuestro SUT

- DbUnit es un **framework** de código abierto creado por Manuel Laflamme basado en JUnit (es, de hecho, una extensión de JUnit)
- DbUnit proporciona una solución “elegante” para controlar la dependencia de las aplicaciones con una base de datos
 - Permite **gestionar el estado** de una base de datos durante las pruebas
 - Permite ser utilizado juntamente con JUnit
- El escenario típico de ejecución de pruebas con bases de datos utilizando DbUnit es el siguiente:
 1. Eliminar cualquier estado previo de la BD resultante de pruebas anteriores (siempre ANTES de ejecutar cada test)
NO restauramos el estado de la BD después de cada test
 2. Cargar los datos necesarios para las pruebas en la BD
Sólo cargaremos los datos NECESARIOS para cada test
 3. Ejecutar las pruebas utilizando métodos de la librería DbUnit para las aserciones



La ejecución de cada uno de los tests debe ser **INDEPENDIENTE** del resto!!!

P INTERFAZ ITABLE

ITable (org.dbunit.dataset)

- Representa una colección de datos tabulares (de una tabla de la BD)
- Se puede usar para preparar los datos iniciales de la BD
- También se utiliza en aserciones, para comparar tablas de bases de datos reales con esperadas
- Implementaciones que se pueden utilizar:
 - * **DefaultTable** - ordenación por clave primaria
 - * **SortedTable** - proporciona una vista ordenada de la tabla
- **ColumnFilterTable** - permite filtrar columnas de la tabla original

table

id	login	password
1	John	John
2	Karl	Karl

S INTERFAZ IDATASET

IDataSet (org.dbunit.dataset)

- Representa una colección de tablas
- Se utiliza para situar la BD en un estado determinado y para comparar el estado actual de la BD con el estado esperado
- Implementaciones que se pueden utilizar:
 - * **FlatXmlDataSet** - lee/escribe datos en formato xml
 - * **QueryDataSet** - guarda colecciones de datos resultantes de una query
- Métodos que se pueden utilizar:
 - * **getTable(tabla)** - devuelve los datos de la tabla especificada

dataset

id	nombre	apellido
1	Ana	Alvarez
2	Carlos	López
3	Pepe	García

id	login	password
1	John	John
2	Karl	Karl

id	firstname	street
1	John	1 Main Street

CLASE FLATXMLDATASET



○ FlatXmlDataSet (org.dbunit.dataset.xml)

- Permite crear "datasets" a partir de documentos XML que contienen datos de varias tablas de la BD con el siguiente formato:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
    <customer id="1"
        firstname="John"
        street="1 Main Street" />
    <user id="1"
        login="John"
        password="John" />
    <user id="2"
        login="Karl"
        password="Karl" />
</dataset>
```

Tabla customer

id	firstname	street
1	John	1 Main Street

Tabla user

id	login	password
1	John	John
2	Karl	Karl

dataset

- También puede "escribir" en un fichero xml en contenido de un "dataset"

P S INTERFAZ IDATABASETESTER

- Es la interfaz que permite el acceso a la BD, devuelve conexiones con una BD de tipo **IDatabaseConnection**
- Implementaciones:
 - **JdbcDatabaseTester** - usa un DriverManager para obtener conexiones con la BD
- Métodos que se pueden utilizar:
 - **setDataSet(IDataSet dataSet),
getDataSet()**
 - * **setDataSet** inyecta los datos de prueba (de tipo **IDataSet**) para inicializar la BD para las pruebas. También podemos usar **getDataSet()** para recuperar dichos datos.
 - **onSetUp(),
setsetUpOperation(DatabaseOperation operacion)**
 - * Por defecto, **onSetUp()** realiza una operación **CLEAN_INSERT** en la BD, insertando en la BD el valor del "dataset" inyectado con el método **setDataSet**. Podemos cambiar la operación con **setUpOperation()**
 - **getConnection()**
 - * Devuelve la conexión (de tipo **IDatabaseConnection**) con la BD

(org.dbunit)

EJEMPLO: uso de **IDatabaseTester**

```

public class TestDBUnit {
    // databaseTester nos permitirá acceder a la BD
    private IDatabaseTester databaseTester;

    @BeforeEach
    public void setUp() throws Exception {
        // Configuramos databaseTester:
        // - Clase que implementa el driver
        // - Cadena de conexión con la base de datos
        // - Login y password para acceder a la BD
        String cadena_conexionDB =
            "jdbc:mysql://localhost:3306/DBUNIT?useSSL=false";
        databaseTester = new JdbcDatabaseTester(
            cadena_conexionDB, "root", "ppss");
        ...
        // dataSet contiene los datos a insertar en la BD
        databaseTester.setDataSet(dataSet);
        // onSetup() realiza CLEAN_INSERT sobre la BD
        // Dicha operación inserta en la BD el contenido
        // de la variable dataSet
        databaseTester.onSetup();
        ...
    }

    @Test
    public void testInsert() throws Exception {
        ...
        // obtenemos una conexión con la BD
        IDatabaseConnection connection =
            databaseTester.getConnection();
        ...
    }
}

```

P CLASE DATABASEOPERATION

- Clase abstracta que define el contrato de la interfaz para **OPERACIONES** realizadas sobre la BD
- Utilizaremos un dataset como entrada para una **DatabaseOperation**
 - **DatabaseOperation.DELETE_ALL**
 - * Elimina todas las filas de las tablas especificadas en el dataset. Si en la BD existe alguna tabla no referenciada en el dataset, ésta no se ve afectada
 - **DatabaseOperation.CLEAN_INSERT**
 - * Realiza una operación **DELETE_ALL**, seguida de un **INSERT** (inserta los contenidos del dataset en la BD. Asume que dichos datos no existen en la BD). Se utiliza en el método **onSetUp()** para inicializar los datos de la BD. Es la forma más segura de garantizar que la BD se encuentre en un estado conocido.
 - **DatabaseOperation.REFRESH**
 - * Realiza actualizaciones e inserciones basadas en el dataset. Los datos existentes no incluidos en el dataset no se ven afectados.
 - **DatabaseOperation.NONE**
 - * No hace nada

S INTERFAZ IDATABASECONNECTION

- Representa una **CONEXIÓN** con una BD (básicamente es un wrapper o adaptador de una conexión JDBC)
- Métodos que se pueden utilizar:
 - **createDataSet()** - crea un dataset (conjunto de datos) con TODOS los datos existentes actualmente en la Base de datos
 - **createDataSet(lista de tablas)** - crea un dataset contenido en las tablas de la BD de la lista de parámetros
 - **createTable(tabla)** - crea un objeto ITable con el resultado de la query "select * from tabla" sobre la BD
 - **createQueryTable(tabla, sql)** - crea un objeto ITable con el resultado de la query sql sobre la BD
 - **getConfig()** - devuelve un objeto de tipo DatabaseConfig, que contiene parejas de (propiedad, valor) con la configuración de la conexión
 - **getRowCount(tabla)** - devuelve el número de filas de una tabla

CLASE ASSERTION

P

S

S

Assertion (org.dbunit)

- Clase que define métodos estáticos para realizar aserciones
- Métodos que se pueden utilizar:
 - **assertEquals**(IDataSet, IDDataSet)
 - **assertEquals**(ITable, ITable)

El API JUnit5 usa el método:
org.junit.jupiter.api.Assertions.assertEquals

El API DBunit usa el método:
org.dbunit.Assertion.assertEquals



EJEMPLO: uso de

- **IDatabaseConnection**,
- **ITable**, **IDataset**,
- **Assertion**

```
public class TestDBUnit {  
    @Test  
    public void testInsert() throws Exception {  
        ...  
  
        // obtenemos la conexión con la BD  
        IDatabaseConnection connection =  
            databaseTester.getConnection();  
        //recuperamos TODOS los datos de la BD  
        //y los guardamos en un IDataset  
        IDataset databaseDataSet= connection.createDataSet();  
        //recuperamos los datos de la tabla "customer"  
        ITable actualTable =  
            databaseDataSet.getTable("customer");  
        //establecemos los valores esperados desde el fichero  
        // customer-expected.xml  
        DataFileLoader loader = new FlatXmlDataFileLoader();  
        IDataset expectedDataSet =  
            loader.load("/customer-expected.xml");  
        ITable expectedTable =  
            expectedDataSet.getTable("customer");  
  
        //comparamos la tabla esperada con la real  
        Assertion.assertEquals(expectedTable, actualTable);  
    }  
}
```

DBUNIT Y PRUEBAS DE INTEGRACIÓN CON MAVEN

Para utilizar DbUnit en nuestros drivers en un proyecto Maven tendremos que incluir en el fichero de configuración (pom.xml) las siguientes dependencias (además de junit5)

```
<dependency>
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>2.7.0</version>
    <scope>test</scope>
</dependency>
```

Librería DbUnit

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.23</version>
    <scope>test</scope>
</dependency>
```

Librería para acceder a una BD MySql

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>2.22.2</version>
            <executions>
                <execution>
                    <goals>
                        <goal>integration-test</goal>
                        <goal>verify</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

La goal "failsafe: integration-test" está asociada por defecto a la fase "integration-test"

En esta fase se ejecutan los métodos anotados con @Test de las clases **/IT*.java, **/*IT.java, o **/*ITCase.java

Los informes se generan en formato *.xml en /target/failsafe-reports

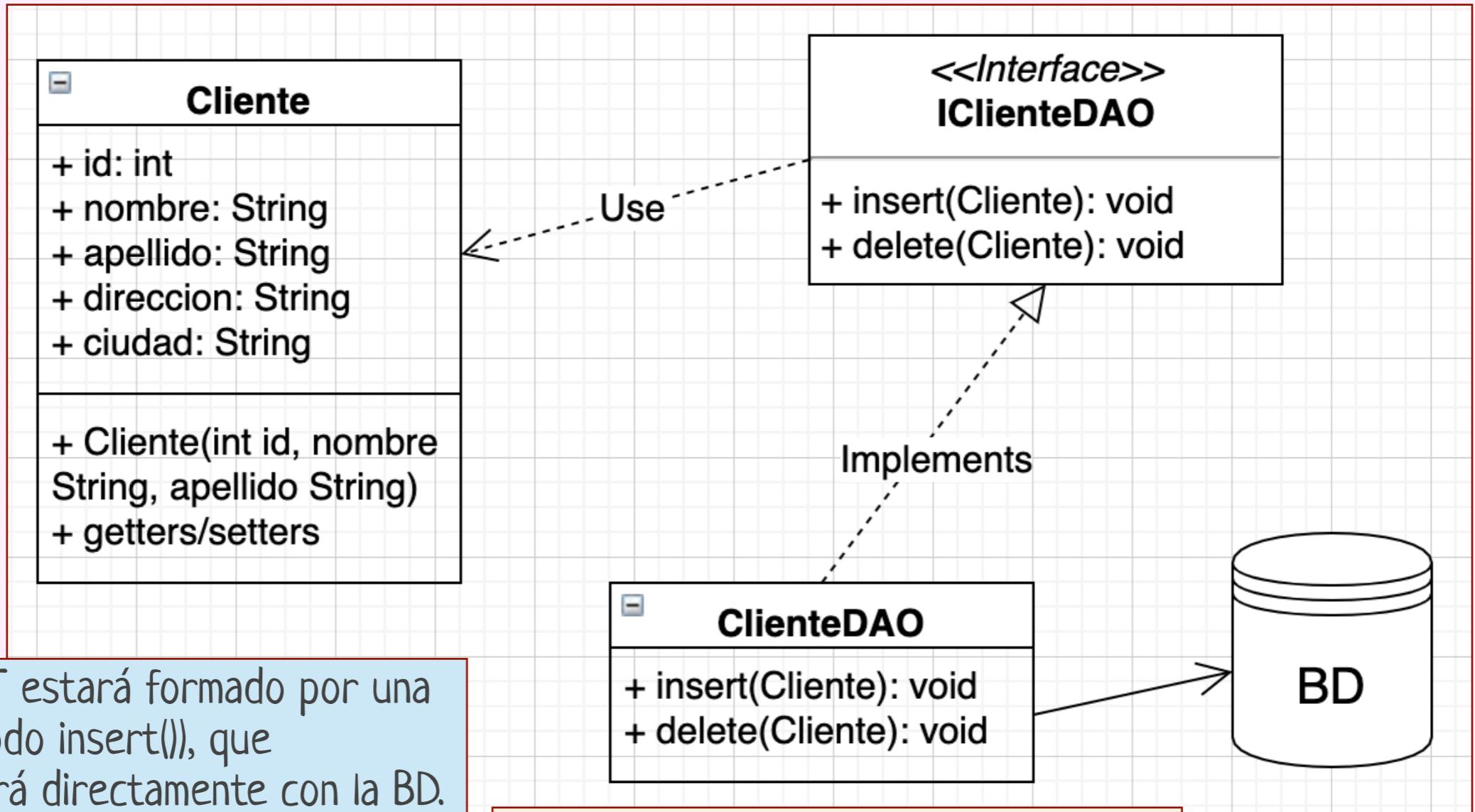
La goal "failsafe: verify" está asociada por defecto a la fase "verify"

Detiene la ejecución si algún test de integración "falla"

EJEMPLO DE TEST DE INTEGRACIÓN

Nuestro test no usará un doble para la BD, sino que accede a la base de datos REAL

- La clase ClienteDAO depende de la base de datos
 - concretamente los métodos insert() y delete()



Nuestro SUT estará formado por una unidad (método insert()), que interaccionará directamente con la BD.

Usaremos DbUnit para controlar el estado de la base de datos antes de las pruebas y comprobar el resultado del acceso a la BD.

Los métodos insert y delete, insertan y borran (respectivamente) un cliente en una base de datos MySql

IMPLEMENTACIÓN DE LOS DRIVERS

- Vamos a ver cómo implementar un driver para realizar pruebas de integración del método ClienteDAO.insert()
- ANTES DE CADA TEST, eliminamos cualquier estado previo de la BD utilizando el método IDatabaseTester.onSetup()

```
public class ClienteDAO_IT {  
  
    private ClienteDAO clienteDAO; //contiene nuestro SUT  
    private IDatabaseTester databaseTester;  
    private IDatabaseConnection connection;  
  
    @BeforeEach  
    public void setUp() throws Exception {  
        String cadenaConexionDB = "jdbc:mysql://localhost:3306/DBUNIT?useSSL=false";  
        databaseTester = new JdbcDatabaseTester(cadenaConexionDB, "root", "ppss");  
  
        //obtenemos la conexión con la BD  
        connection = databaseTester.getConnection();  
  
        // inicializamos el dataset para inicializar la BD  
        IDataSet dataSet = new FlatXmlDataFileLoader().load("/cliente-init.xml");  
        // inyectamos el dataset  
        databaseTester.setDataSet(dataSet);  
  
        // inicializamos la BD con el dataset inicial  
        databaseTester.onSetup();  
  
        clienteDAO = new ClienteDAO();  
    }  
...  
}
```

Necesitamos una instancia de IDatabaseTester para acceder a la BD

Datos para la conexión con la BD

Obtenemos la conexión con la BD

Dataset inicial: tabla de clientes VACÍA

Borra los datos de las tablas del dataset inicial en la BD e inserta en la BD el contenido del dataset inicial

Instancia que contiene nuestro SUT

PI IMPLEMENTACIÓN DEL CASO DE PRUEBA

- Probaremos la inserción de un cliente en una base de datos vacía

```
@Test  
public void testInsert() throws Exception {  
    Cliente cliente = new Cliente(1,"John", "Smith");  
    cliente.setDireccion("1 Main Street");  
    cliente.setCiudad("Anycity");  
  
    //invocamos a nuestro SUT  
    clienteDAO.insert(cliente);  
  
    //recuperamos los datos de la BD después de invocar al SUT  
    IDDataSet databaseDataSet = connection.createDataSet();  
    //Recuperamos los datos de la tabla cliente  
    ITable actualTable = databaseDataSet.getTable("cliente");  
  
    //creamos el dataset con el resultado esperado  
    IDDataSet expectedDataSet = new FlatXmlDataFileLoader().load("/cliente-esperado.xml");  
    ITable expectedTable = expectedDataSet.getTable("cliente");  
  
    //comparamos la tabla esperada con la real  
    Assertion.assertEquals(expectedTable, actualTable);  
}  
//fin testInsert()
```

Datos de entrada del caso de prueba

Ejercitamos nuestro SUT

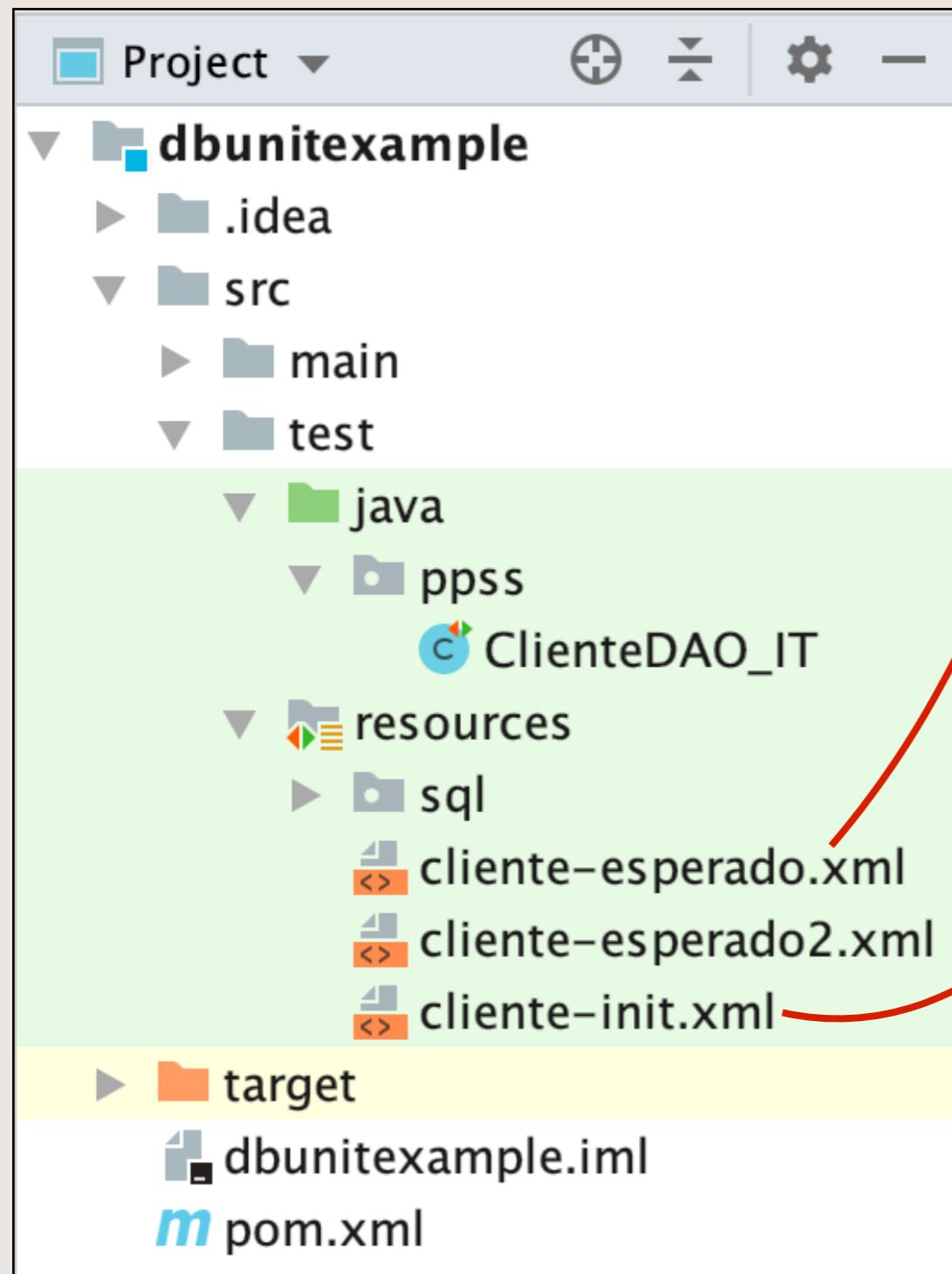
Resultado REAL

Resultado ESPERADO

Comparamos el resultado ESPERADO con el REAL

DATOS DE ENTRADA Y RESULTADO ESPERADO

Los datos de entrada y el resultado esperado los almacenamos en ficheros de recursos xml que convertiremos en un IDataSet



```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <cliente id="1">
    <nombre>John</nombre>
    <apellido>Smith</apellido>
    <direccion>1 Main Street</direccion>
    <ciudad>AnyCity</ciudad>
  </cliente>
</dataset>
```

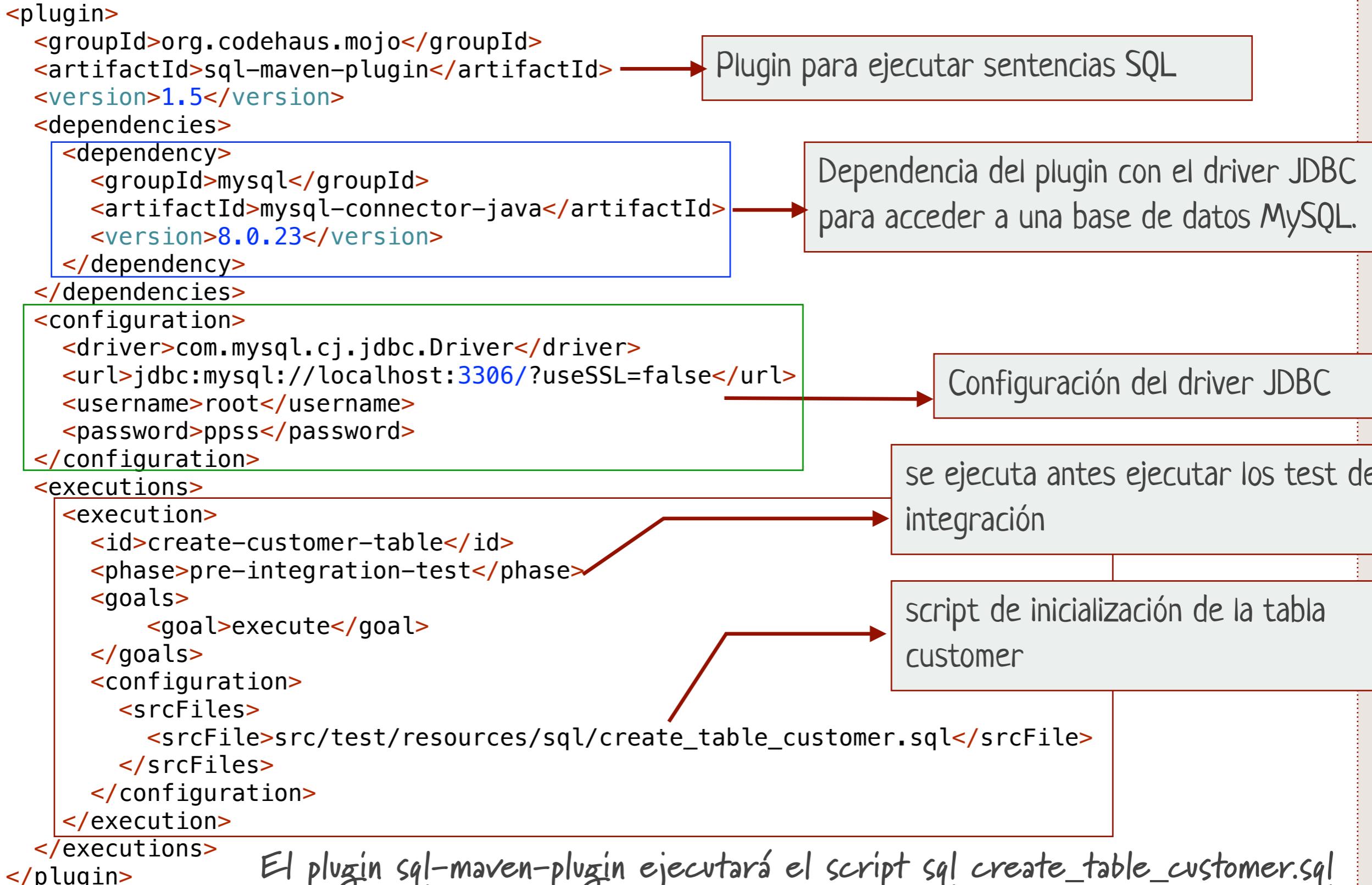
Resultado esperado: la tabla cliente contiene únicamente el cliente insertado

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <cliente />
</dataset>
```

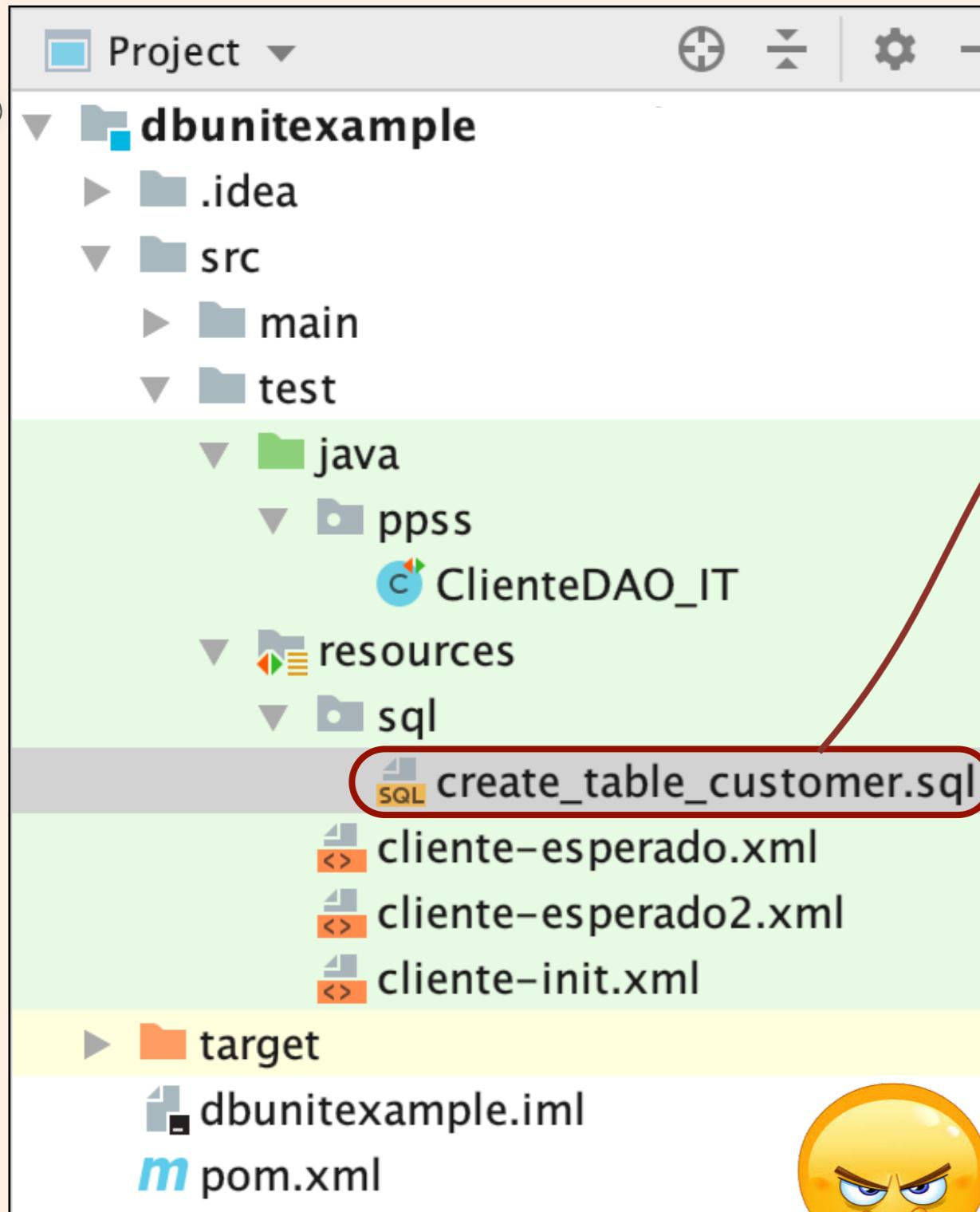
Inicialmente la tabla cliente no contiene ningún dato

P S PLUG-IN SQL-MAVEN-PLUG-IN: SCRIPT INICIALIZACIÓN BD

Configuramos el pom para inicializar las tablas de nuestra BD ANTES de ejecutar los tests:



INICIALIZACIÓN DE LA TABLA CLIENTE



```
SQL create_table_customer.sql ×  
1  DROP DATABASE IF EXISTS DBUNIT;  
2  CREATE DATABASE IF NOT EXISTS DBUNIT;  
3  USE DBUNIT;  
4  
5  DROP TABLE IF EXISTS cliente;  
6  
7  CREATE TABLE cliente (  
8      id int(11) NOT NULL,  
9      nombre varchar(45) DEFAULT NULL,  
10     apellido varchar(45) DEFAULT NULL,  
11     direccion varchar(45) DEFAULT NULL,  
12     ciudad varchar(45) DEFAULT NULL,  
13     PRIMARY KEY (id)  
14 );
```

La carpeta `src/test/resources` almacenará cualquier fichero adicional (fichero no java) que necesites utilizar para ejecutar tu código de pruebas (desde `src/test/java`)

Dentro de la carpeta de recursos puedes crear todos los subdirectorios que consideres oportunos.

Por ejemplo, hemos creado el subdirectorio `sql` con el script `sql` para inicializar la BD

Maven copia los ficheros de `src/test/resources` al directorio `target`, durante la fase `process-test-resources`



P PROCESO COMPLETO PARA AUTOMATIZAR LAS PRUEBAS DE P INTEGRACIÓN CON MAVEN

pre-integration-test: se ejecutan acciones previas a la ejecución de los tests

integration-test: se ejecutan los tests de integración. Deben tener el prefijo o sufijo IT. Si algún test falla NO se detiene la construcción.

post-integration-test: en esta fase “detendremos” todos los servicios o realizaremos las acciones que sean necesarias para volver a restaurar el entorno de pruebas

verify: en esta fase se comprueba que todo está listo (no hay ningún error) para poder copiar el artefacto generado en nuestro repositorio local

Si algún test ha fallado, se detiene la construcción



Las fases pre-integration-test ... verify, por defecto NO tienen asociada ninguna goal cuando el empaquetado del proyecto es jar, por lo que tendremos que incluir y configurar los plugins necesarios en el pom del proyecto

GOALS por defecto:

compiler:compile

resources:testResources

compiler:testCompile

surefire:test

jar:jar

sql:execute

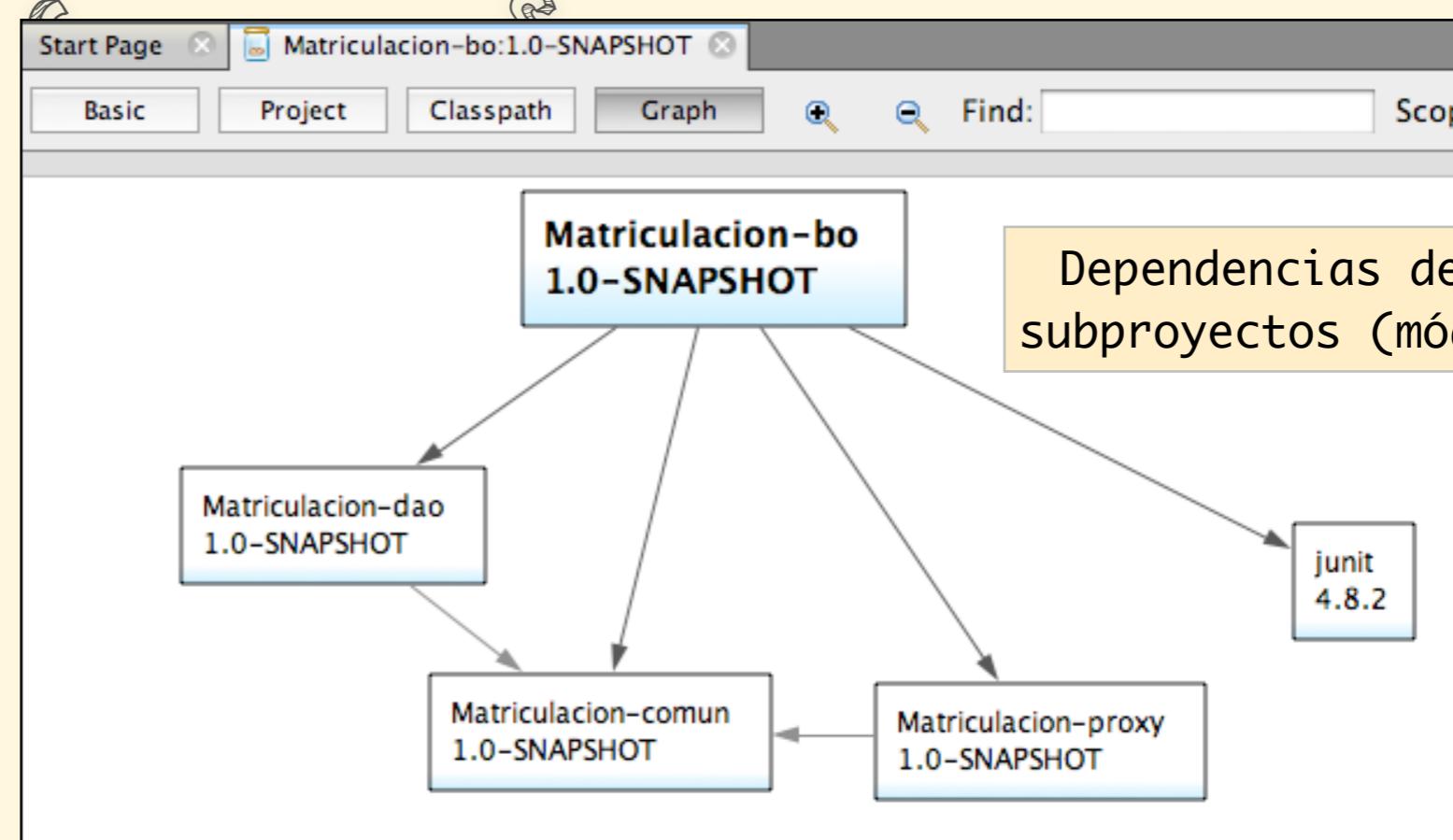
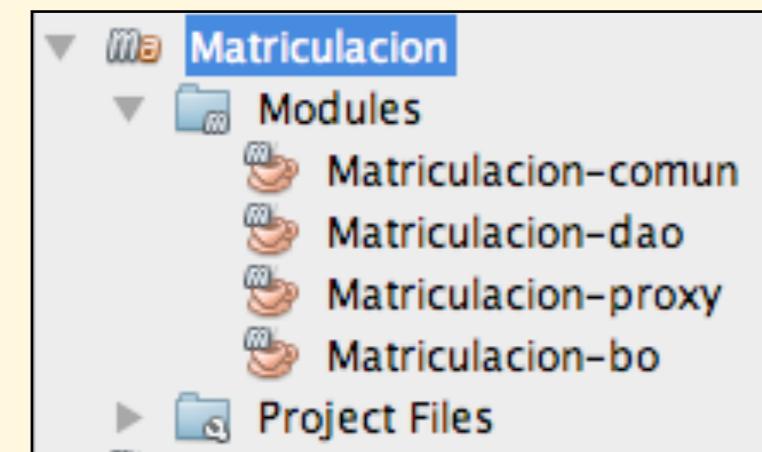
failsafe:integration-test

failsafe:verify

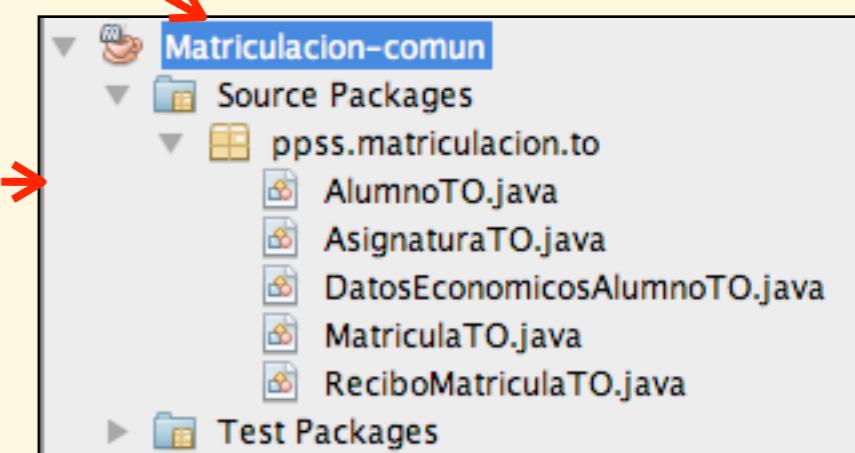
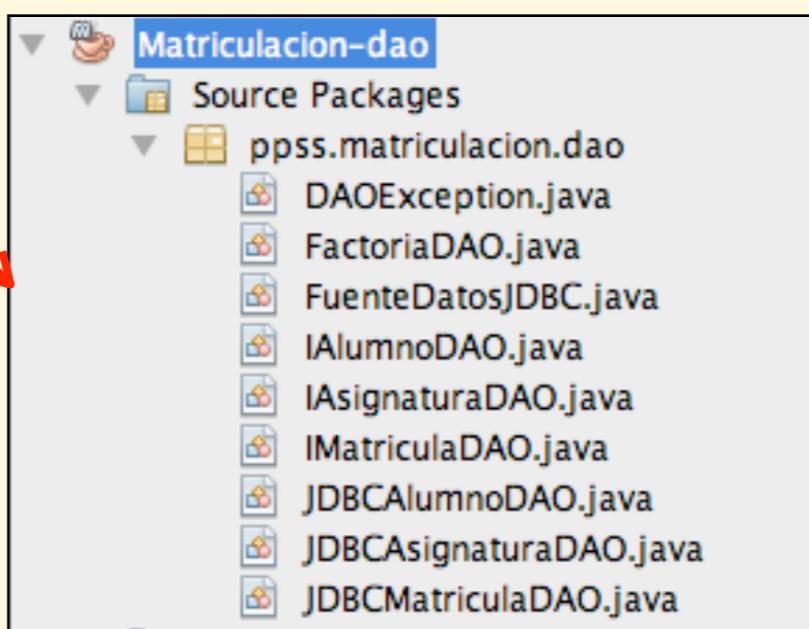
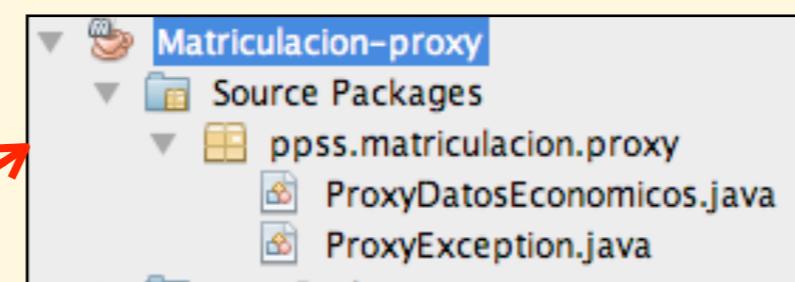
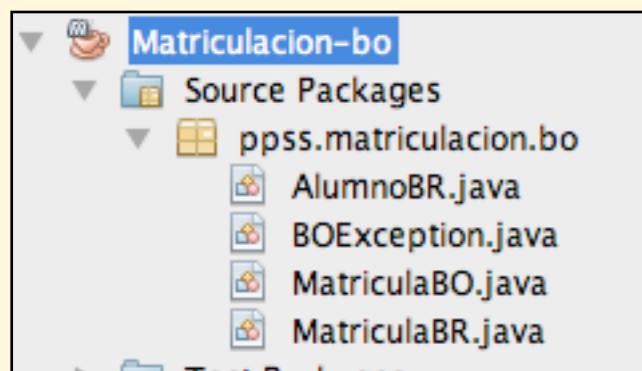
validate
initialize
generate-sources
process-sources
generate-resources
process-resources
compile
process-classes
generate-test-sources
process-test-sources
generate-test-resources
process-test-resources
test-compile
process-test-classes
test
prepare-package
package
pre-integration-test
integration-test
post-integration-test
verify
install
deploy

EJEMPLO

Proyecto Matriculacion

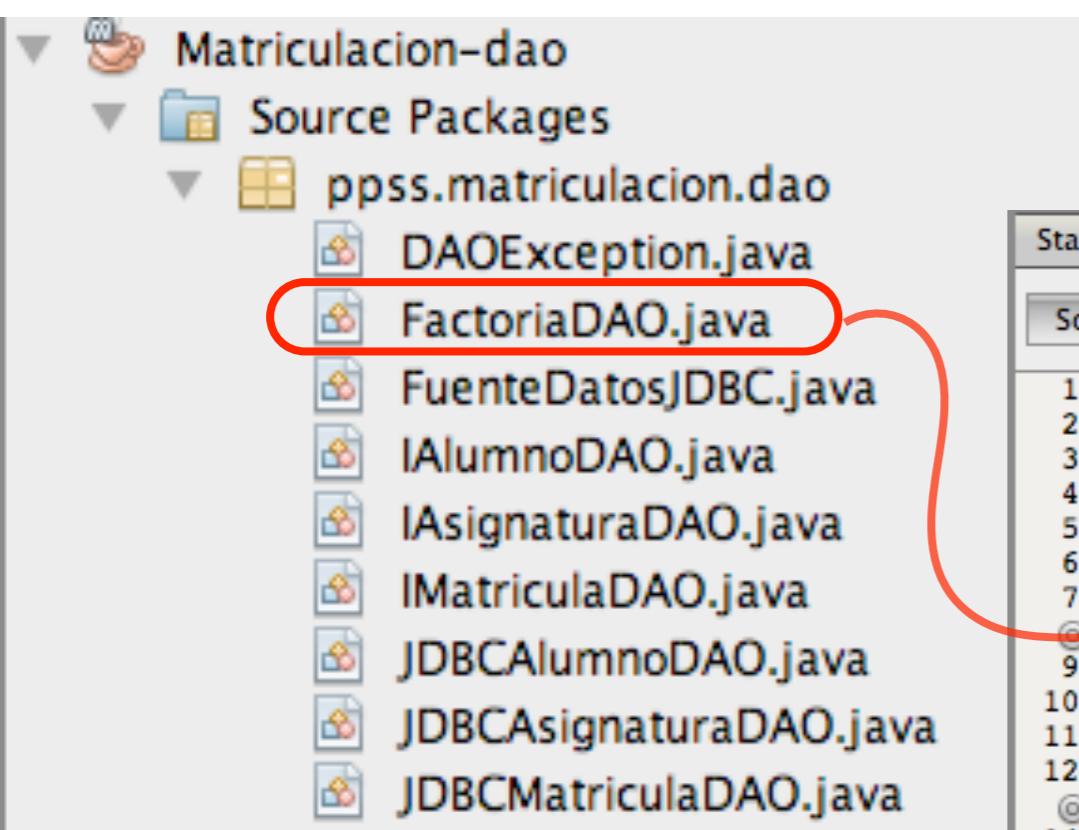


Dependencias de los subproyectos (módulos)



CAPA DE ACCESO A DATOS: FACTORIA DE DAOs Subproyecto Matriculacion-dao

Usamos una clase factoría que proporciona los DAOs utilizados por los objetos de la capa de negocio (*Business Objects*)



FactoriaDAO proporciona objetos DAOs JCBC de los siguientes tipos:

- JDBCAlumnoDAO
 - JDBCAsignaturaDAO
 - JDBCMatriculaDAO

The screenshot shows a Java code editor with the file `FactoriaDAO.java` open. The code defines a class `FactoriaDAO` with three methods: `getAlumnoDAO()`, `getAsignaturaDAO()`, and `getMatriculaDAO()`, each returning a specific DAO implementation (`JDBCAlumnoDAO`, `JDBCAsignaturaDAO`, and `JDBCMatriculaDAO` respectively). The code is annotated with Javadoc comments describing its purpose. A red arrow points to the first method definition.

```
1 package ppss.matriculacion.dao;
2
3 /**
4 * La clase <code>GestorDAO</code> se utilizará como factoría de los ob-
5 *jetos de datos de la aplicación.
6 *
7 */
8 public class FactoriaDAO {
9     /**
10      * Devuelve al DAO para acceder a los datos de los alumnos.
11      * @return DAO que da acceso a los alumnos.
12      */
13     public IAlumnoDAO getAlumnoDAO() {
14         return new JDBCAlumnoDAO();
15     }
16
17     /**
18      * Devuelve al DAO para acceder a los datos de las asignaturas.
19      * @return DAO que da acceso a las asignaturas.
20      */
21     public IAsignaturaDAO getAsignaturaDAO() {
22         return new JDBCAsignaturaDAO();
23     }
24
25     /**
26      * Devuelve al DAO para acceder a los datos de matrículación.
27      * @return DAO que da acceso a las matrículaciones.
28      */
29     public IMatriculaDAO getMatriculaDAO() {
30         return new JDBCMatriculaDAO();
31     }
32 }
```

CAPA DE ACCESO A DATOS: IMPLEMENTACIÓN DE DAOs

P

Cada DAO implementa una interfaz (operaciones CRUD sobre la BD)

P

The screenshot shows an IDE interface with several tabs at the top: Start Page, FactoriaDAO.java, JDBCAlumnoDAO.java (which is the active tab), and FuenteDatosJDBC.java. Below the tabs is a toolbar with various icons. The main area displays the source code for JDBCAlumnoDAO.java:

```
1 package ppss.matriculacion.dao;
2
3 import java.sql.Connection;
4 import java.sql.PreparedStatement;
5 import java.sql.ResultSet;
6 import java.sql.SQLException;
7 import java.util.ArrayList;
8 import java.util.List;
9
10 import ppss.matriculacion.to.AlumnoTO;
11
12
13 /**
14 * Implementaci&on del acceso a los datos de los alumnos almacenados en una base de
15 * datos.
16 *
17 */
18 public class JDBCAlumnoDAO implements IAlumnoDAO {
19
20     public void addAlumno(AlumnoTO a) throws DAOException {
21         Connection con = null;
22         PreparedStatement ps = null;
23
24         if(a==null) {
25             throw new DAOException("El alumno a insertar no puede ser nulo");
26         }
27
28         try {
29             con = FuenteDatosJDBC.getInstance().getConnection();
30             ps = con.prepareStatement("INSERT INTO alumnos(nif, nombre, dir");
31             ps.setString(1, a.getNif());
32             ps.setString(2, a.getNombre());
33             ps.setString(3, a.getDireccion());
34             ps.setString(4, a.getEmail());
35             ps.setDate(5, new java.sql.Date(a.getFechaNacimiento().getTime()));
36
37             ps.executeUpdate();
38
39         } catch (SQLException e) {
40             throw new DAOException("Error al insertar el alumno: " + e.getMessage());
41         } finally {
42             try {
43                 if(ps != null)
44                     ps.close();
45                 if(con != null)
46                     con.close();
47             } catch (SQLException e) {
48                 throw new DAOException("Error al cerrar la conexión: " + e.getMessage());
49             }
50         }
51     }
52 }
```

A red arrow points from the package structure on the left to the implements clause in the code. Another red circle highlights the JDBCAlumnoDAO.java file in the package structure.

JDBCAlumnoDAO implementa la interfaz IAlumnoDAO

JDBCAlumnoDAO implementa operaciones CRUD de los objetos T0, p.ej. addAlumno

CAPA DE ACCESO A DATOS: INTERFACES DE LOS DAOs

Depende de la BD

P

P

La interfaz IAlumnoDAO permite que los cambios en la implementación de la fuente de datos subyacente NO afecten a los componentes de negocio.



Los objetos de la capa de negocio son CLIENTES de la capa de acceso a datos

The screenshot shows an IDE interface with several tabs at the top: Start Page, FactoriaDAO.java, JDBCAlumnoDAO.java, FuenteDatosJDBC.java, and IAlumnoDAO.java. Below the tabs is a toolbar with various icons. The main area displays the code for the IAlumnoDAO interface:

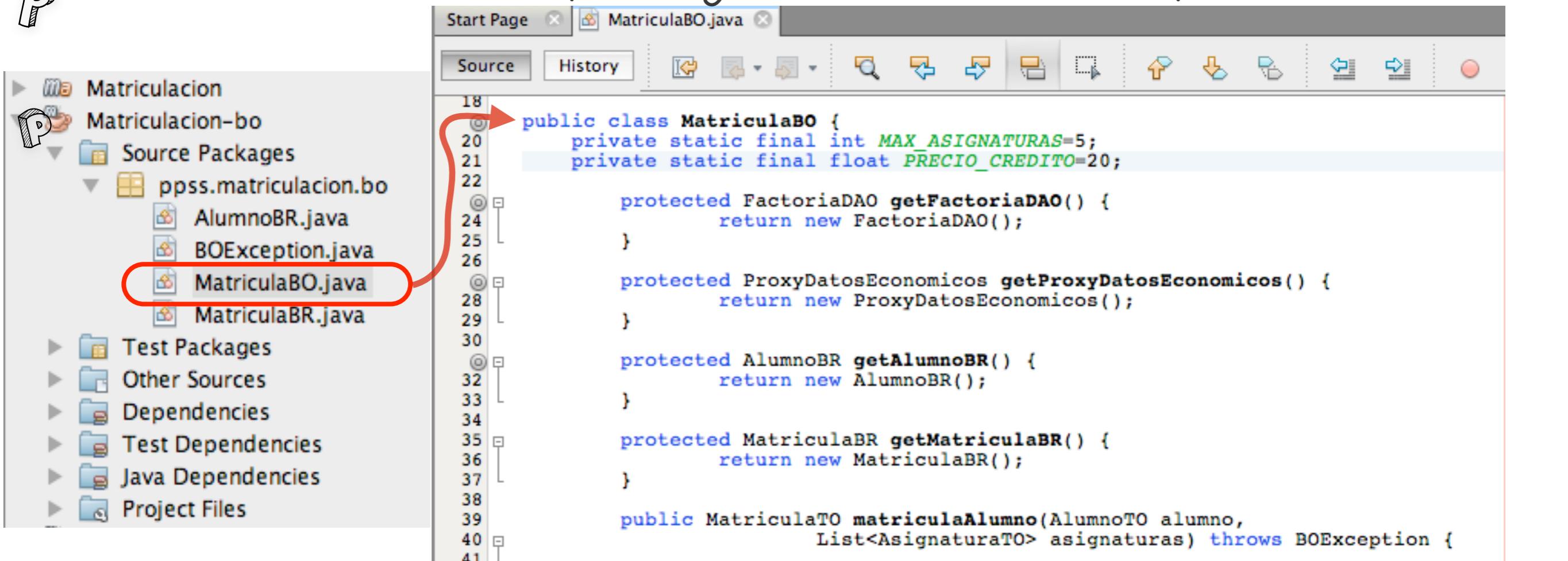
```
1 package ppss.matriculacion.dao;
2
3 import java.util.List;
4 import ppss.matriculacion.to.AlumnoTO;
5
6 /**
7  * Interfaz común de los objetos que dan acceso a los datos de los alumnos.
8  *
9  */
10 public interface IAlumnoDAO {
11
12     /**
13      * De alta una alumno.
14      * @param a Alumno que se añadirá. Se producirá un error si el alumno
15      * ya existe, o si el parámetro o su campo <code>nif</code> es <code>nu
16      * @throws DAOException Si ocurre un error al añadir al alumno
17      */
18     public abstract void addAlumno(AlumnoTO a) throws DAOException;
19
20     /**
21      * De baja un alumno.
22      * @param nif Nif del alumno a eliminar. Se producirá un error si el al
23      * existe, o si el parámetro es <code>null</code>.
24      * @throws DAOException Si ocurre un error al eliminar al alumno
25      */
26     public abstract void delAlumno(String nif) throws DAOException;
27
28     /**
29      * Obtiene los datos de un alumno.
30      * @param nif Nif del alumno del cual queremos obtener los datos
31      * @return Datos del alumno, o <code>null</code> si el alumno no existe
32      * @throws DAOException Si ocurre un error al recuperar los datos
33      */
34     public abstract AlumnoTO getAlumno(String nif) throws DAOException;
35 }
```

A red arrow points from the text "Los objetos de la capa de negocio son CLIENTES de la capa de acceso a datos" to the code line "public abstract void addAlumno(AlumnoTO a) throws DAOException;". Another red circle highlights the file "IAlumnoDAO.java" in the project tree on the left. The project tree also lists other files like DAOException.java, FactoriaDAO.java, FuenteDatosJDBC.java, IAsignaturaDAO.java, IMatriculaDAO.java, JDBCAlumnoDAO.java, JDBCAsignaturaDAO.java, and JDBCMatriculaDAO.java.

IAlumnoDAO especifica las operaciones que se pueden realizar sobre la BD. Operaciones CRUD (Create, Retrieve, Update, Delete)

CAPA DE NEGOCIO

La capa de negocio es CLIENTE de la capa de acceso a datos



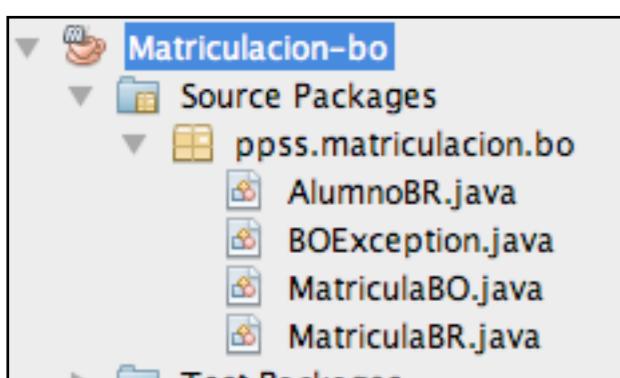
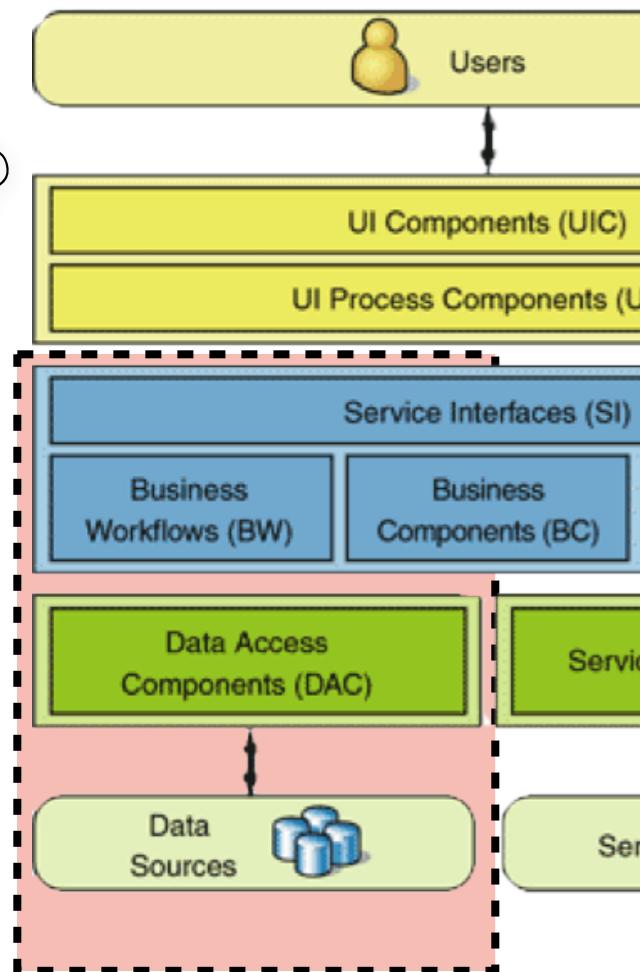
MatriculaB0 utiliza objetos DAO para acceder a los objetos T0 persistentes.

Si cambiamos el mecanismo de persistencia NO necesitamos modificar el código de MatriculaBO (lógica de negocio)

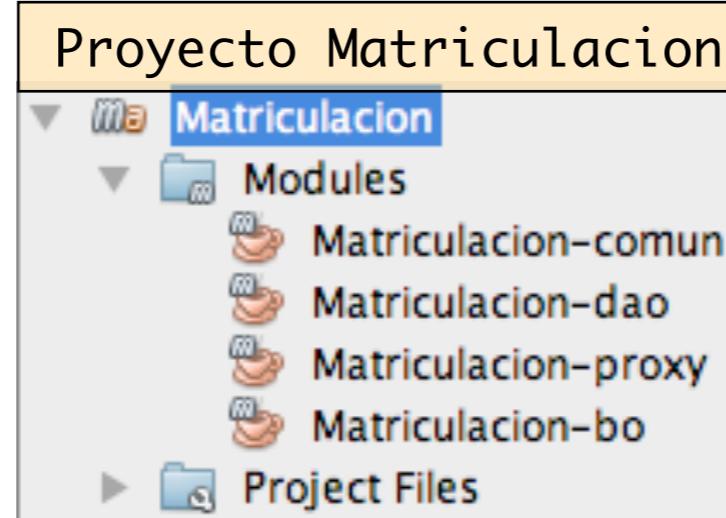
EJEMPLO DE ESTRATEGIA DE INTEGRACIÓN

P

P



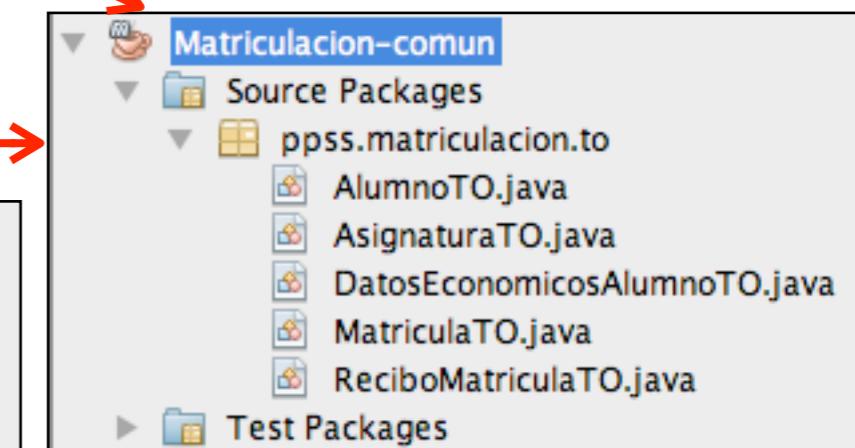
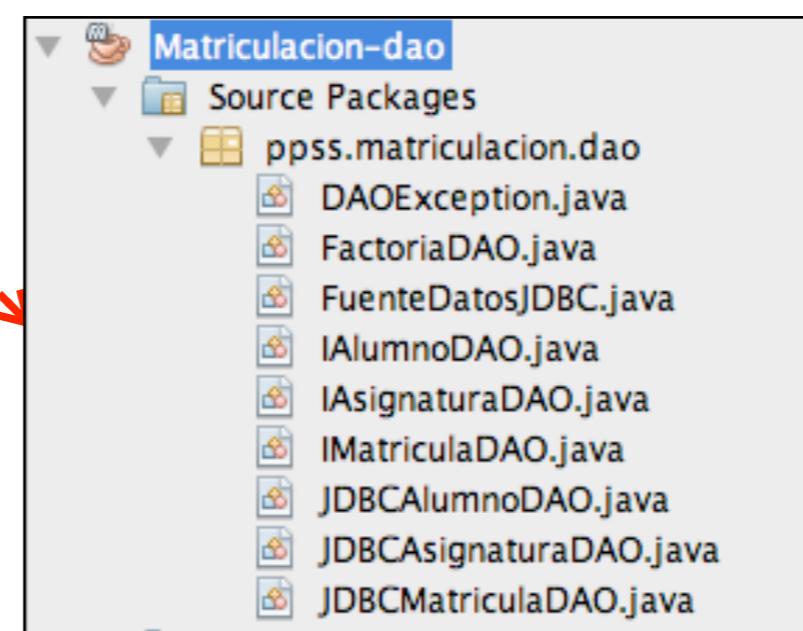
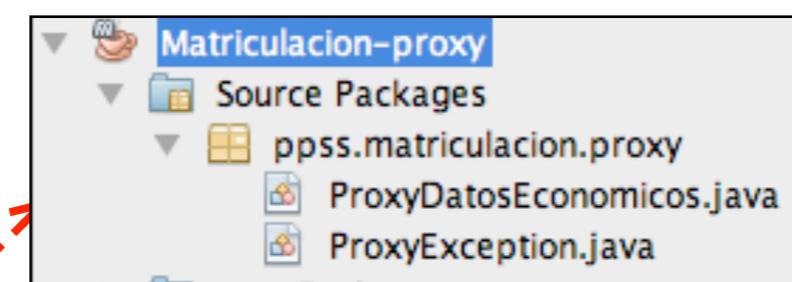
Dependencias de los subproyectos



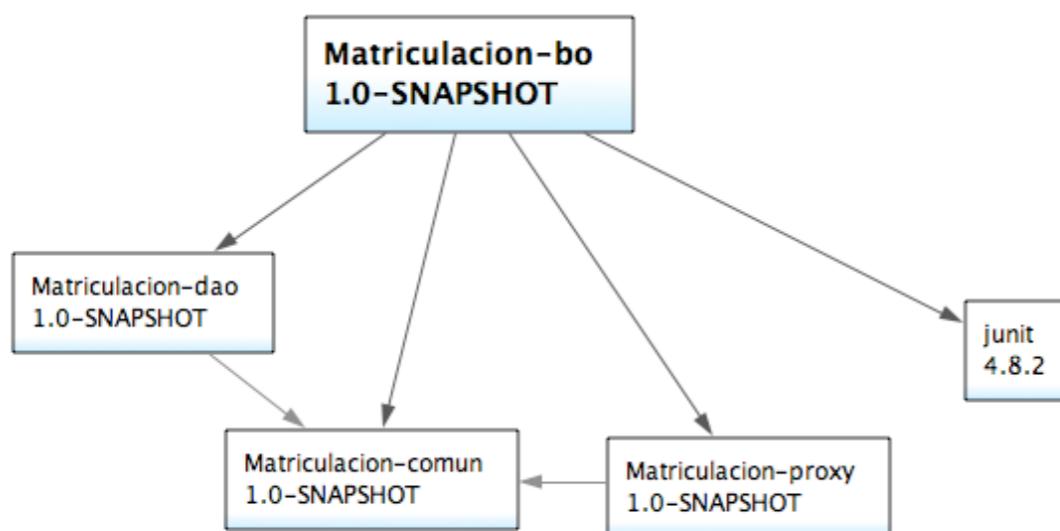
Possible estrategia de integración

Bottom up

- 1 **Matriculacion-dao**
- 2 **Matriculacion-bo + Matriculacion-dao**
- 3 **Matriculacion-bo + Matriculacion-dao + Matriculacion-proxy**



EJECUCIÓN DE LA ESTRATEGIA DE INTEGRACIÓN



El orden de ejecución de los tests debería ser: 1, 2, 3
Para ello podríamos usar etiquetas para los tests de Matriculación-bo

mvn verify -Dgroups="Integracion-fase1"
mvn verify -Dgroups="Integracion-fase2"
mvn verify -Dgroups="Integracion-fase3"

Para automatizar las pruebas necesitaremos implementar en cada una de las fases de la integración:

1

Matriculacion-dao

Utilizamos DbUnit para implementar Tests de integración con la BD:

/src/test/java/AlumnoDA0IT.java, /src/test/java/AsignaturaDA0IT.java,
/src/test/java/MatriculaDA0IT.java

implementados en el proyecto Matriculacion-dao
@Tag Integracion-fase1

mvn verify -Dgroups="Integracion-fase1"

2

Matriculacion-bo + Matriculacion-dao

implementados en el proyecto Matriculacion-bo
@Tag Integracion-fase2

Implementamos:

- stubs para Matriculación-proxy
 - tests DBUnit:
 - /src/test/java/MatriculaB0-daoIT.java
- mvn verify -Dgroups="Integracion-fase2"

3

Matriculacion-bo + Matriculacion-dao +
Matriculacion-proxy

implementados en el proyecto Matriculacion-bo
@Tag Integracion-fase3

Implementamos:

- tests DBUnit:
 - /src/test/java/MatriculaB0-dao-proxyIT.java

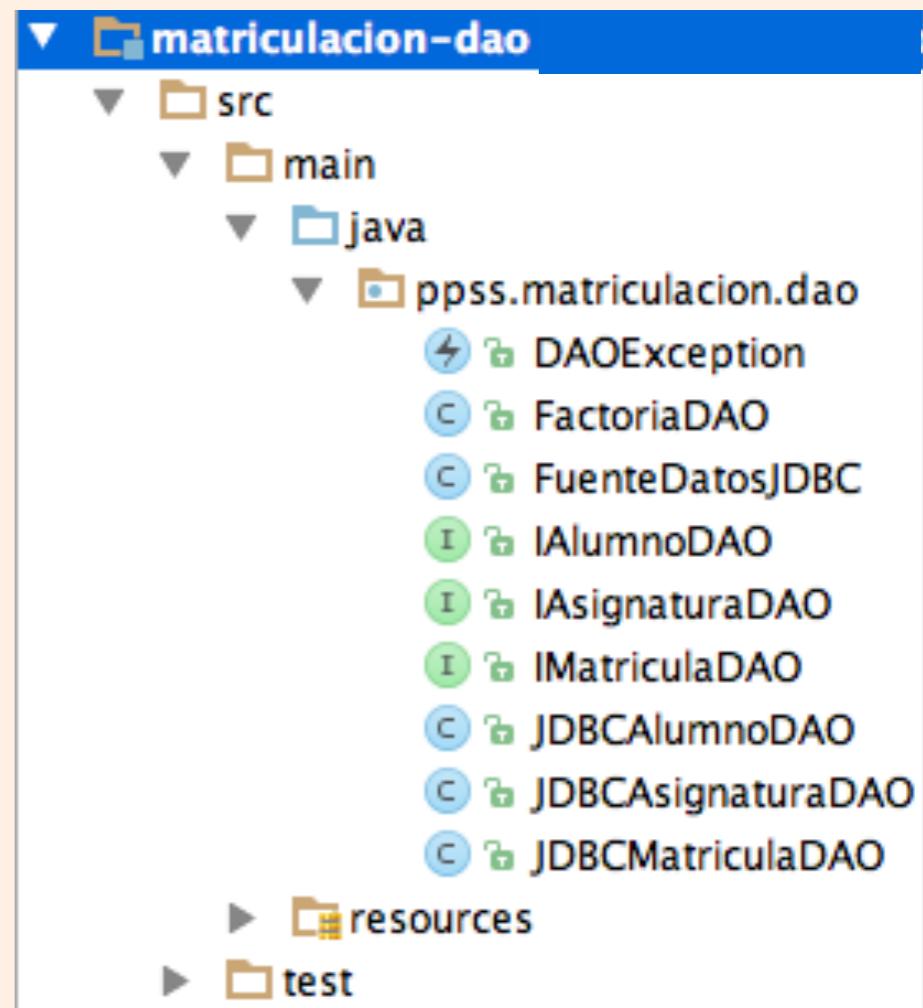
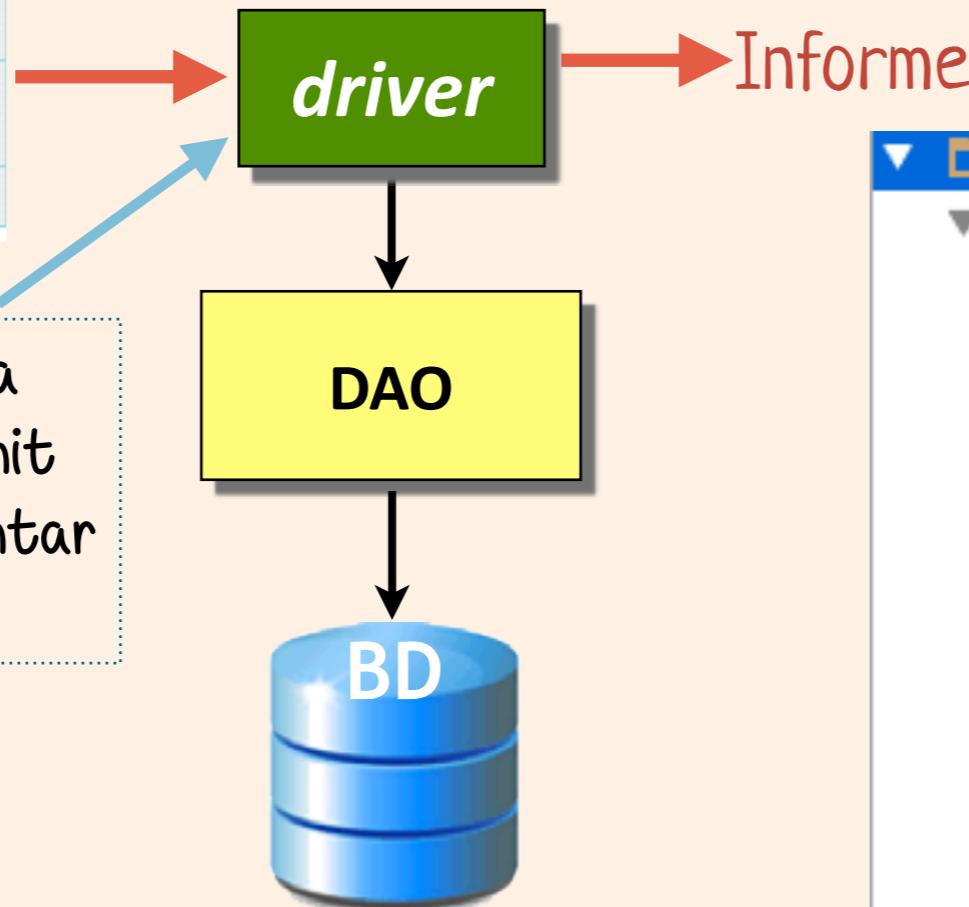
mvn verify -Dgroups="Integracion-fase3"

Y AHORA VAMOS AL LABORATORIO...

Vamos a implementar tests de integración con una BD MySql utilizando DbUnit

Camino	Datos Entrada	Resultado Esperado
C1	d1=... d2=... ...	r1
..
CM	d1=... d2=... ...	rM

Tests integración



REFERENCIAS BIBLIOGRÁFICAS

○ Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008

- Capítulo 7: System Integration Testing

○ Software Engineering. 9th edition. Ian Sommerville. 2011

- Capítulo 8.1.3: Component testing

○ DbUnit (sitio oficial)

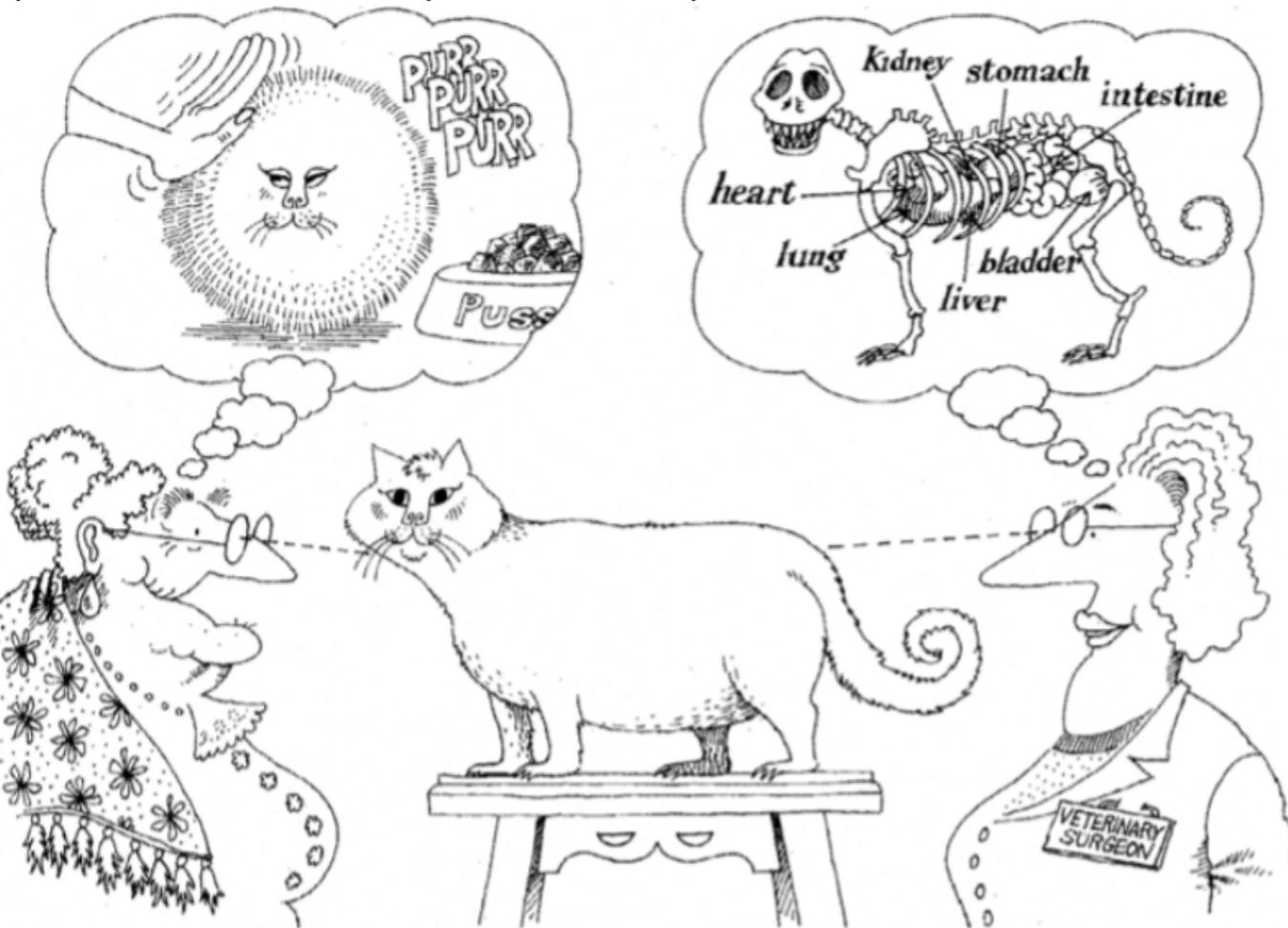
- <http://dbunit.sourceforge.net/>

○ Core J2EE Patterns – Data Access Object

- <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>

Sesión S07: Pruebas del sistema y aceptación

pruebas de aceptación pruebas del sistema



Pruebas: del sistema

- Objetivo: encontrar defectos en las funcionalidades (desde el punto de vista del desarrollador)
- Diseño:
 - Método de transición de estados
 - Método basado en casos de uso

Pruebas de aceptación

- Objetivo: validar las expectativas del cliente
- Propiedades emergentes funcionales (siempre desde el punto de vista del usuario) y criterios de aceptación
- Diseño:
 - Método basado en requerimientos
 - Método basado en escenarios
- Automatización:
 - Selenium IDE (para aplicaciones con interfaz web)
 - Webdriver (para aplicaciones con interfaz web)
 - Se pueden usar también para pruebas del sistema

Vamos al laboratorio...

P

VALIDACIÓN

¿el producto es el correcto?

El objetivo es DEMOSTRAR que se satisfacen las EXPECTATIVAS DEL CLIENTE

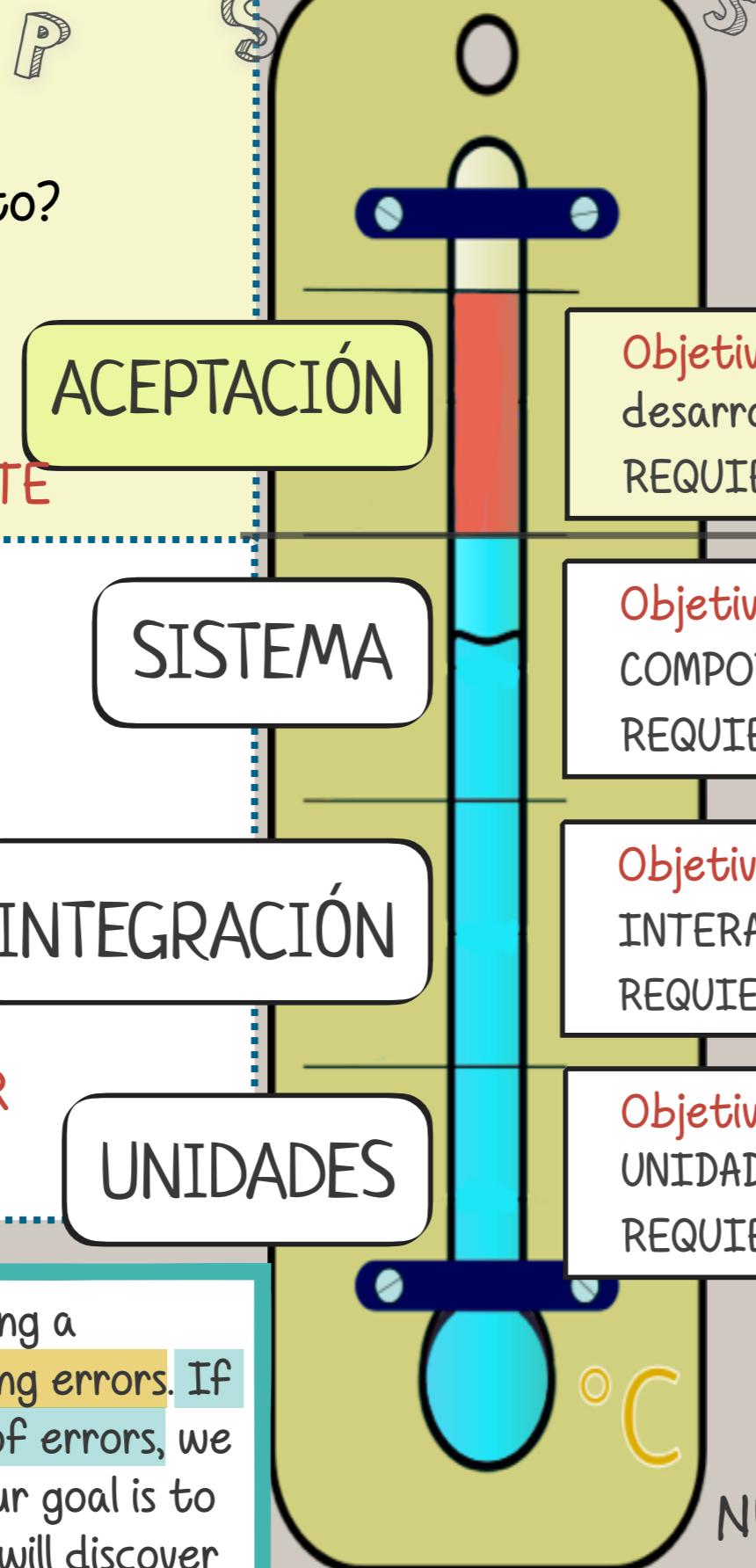
VERIFICACIÓN

¿el producto está implementado correctamente?

El objetivo general (intención) es ENCONTRAR DEFECTOS

"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)



NIVELES DE PRUEBAS

Cada nivel tiene una granularidad y objetivos concretos diferentes. Hay un ORDEN temporal entre ellos.

Objetivo: valorar en qué grado el software desarrollado satisface las expectativas del cliente
REQUIERE los CRITERIOS DE ACEPTACIÓN

Objetivo: encontrar DEFECTOS derivados del COMPORTAMIENTO del sw como un todo
REQUIERE los REQUISITOS (funcionalidades) del sistema

Objetivo: encontrar DEFECTOS derivados de la INTERACCIÓN de las unidades probadas
REQUIERE establecer un ORDEN de las unidades a integrar

Objetivo: encontrar DEFECTOS en el código de las UNIDADES probadas
REQUIERE AISLAR el código de cada unidad a probar

NUESTRA intención es MUY IMPORTANTE!!!

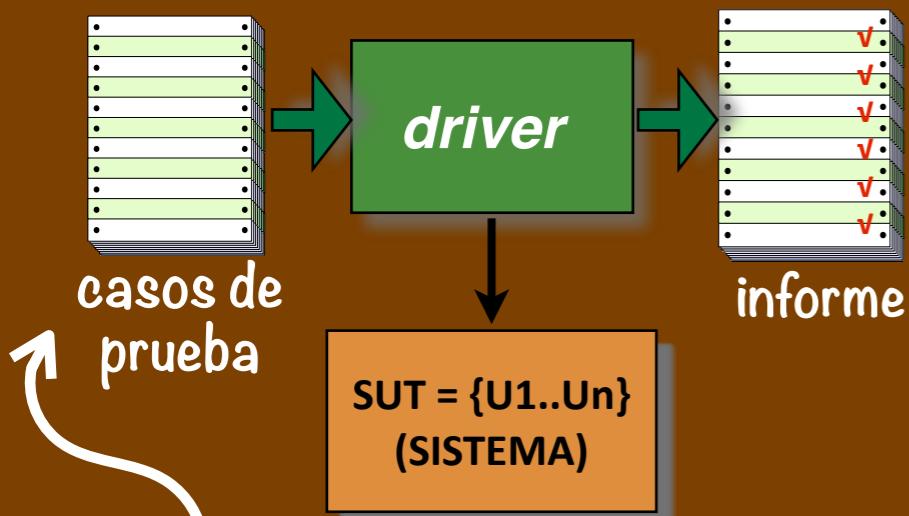


PS PR. SISTEMA VS PR. ACEPTACIÓN SP

VALIDACIÓN

VERIFICACIÓN

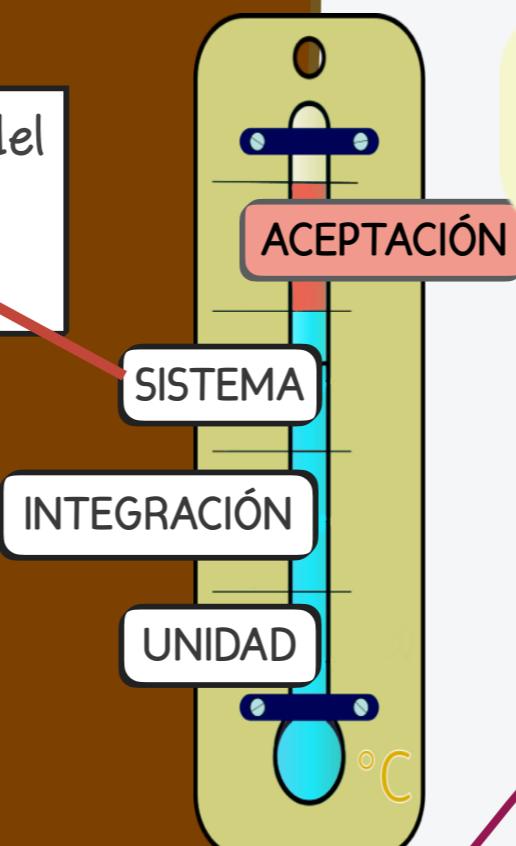
Objetivo: encontrar **DEFECTOS** derivados del **COMPORTAMIENTO** del sw como un todo
REQUIERE los **REQUISITOS** del sistema



Los casos de prueba se diseñan desde el **punto de vista del DESARROLLADOR**

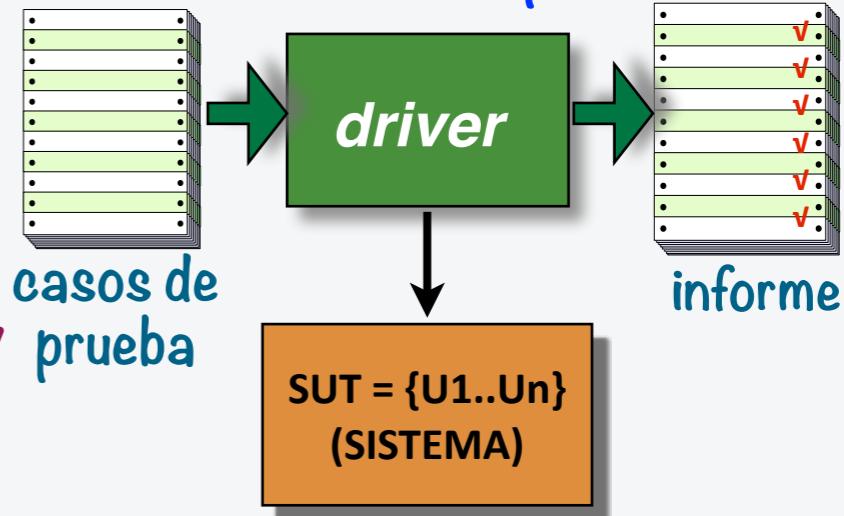


Los casos de prueba se diseñan desde el **punto de vista del USUARIO (CLIENTE)**

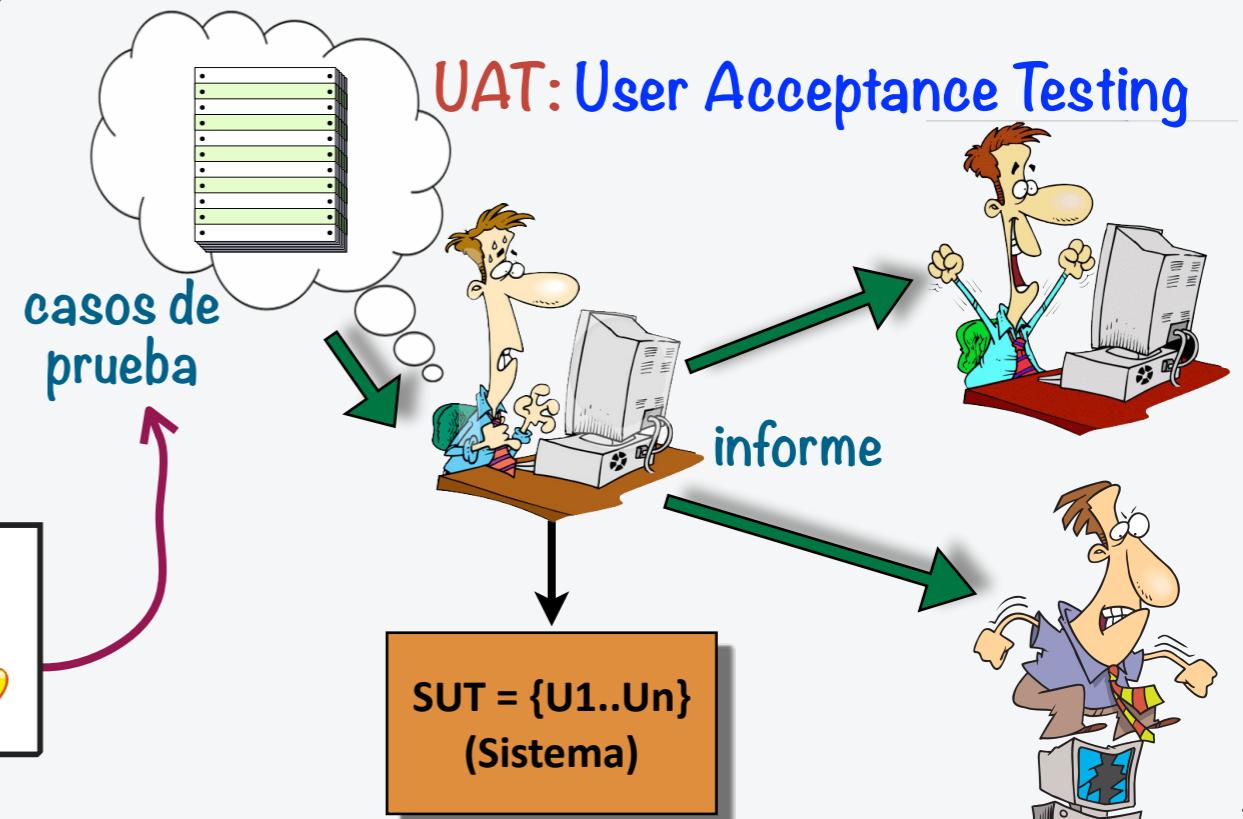


Objetivo: valorar en qué grado el software desarrollado satisface las expectativas del cliente
REQUIERE los **CRITERIOS DE ACEPTACIÓN**

BAT: Business Acceptance Testing



UAT: User Acceptance Testing



PRUEBAS DEL SISTEMA

P ver Cap. 8 Sommerville. 10ed "Software testing"

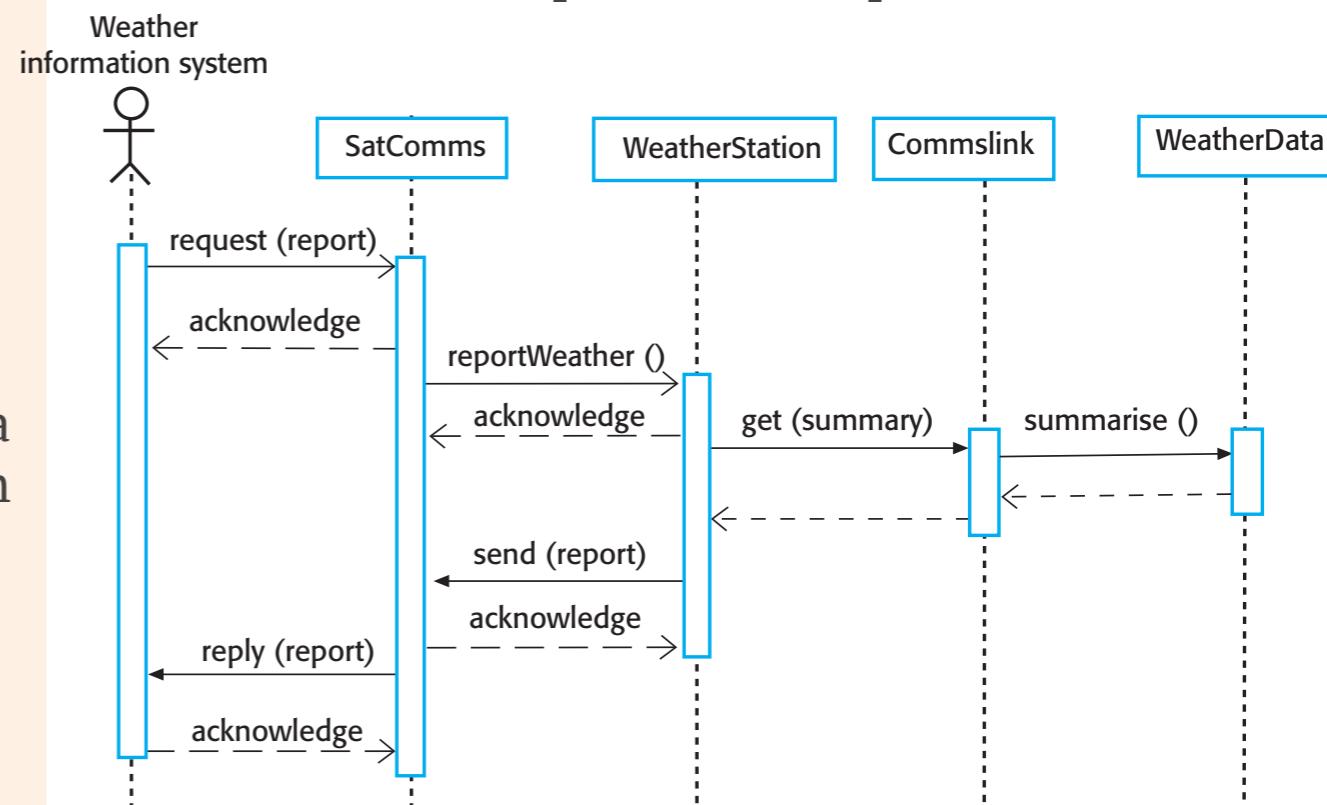
- Son pruebas dinámicas y obtienen los casos de prueba a partir de las especificaciones del sistema (caja negra)
- Se realizan durante el proceso de desarrollo, y usan todos los componentes del sistema, una vez que éstos han sido integrados. Se prueban:
 - Si los componentes son compatibles, si interactúan correctamente y transfieren el dato adecuado en el instante de tiempo adecuado a través de sus interfaces
- Las diferencias con las pruebas de integración son:
 - Se incluyen componentes reutilizables desarrollados por "terceros"
 - Los comportamientos a probar son los especificados para el sistema en su conjunto. P.ej. en un sistema de compra on-line, se prueba el proceso "completo" de compra por parte de un usuario
- Ejemplo de métodos de diseño del sistema:
 - Método de diseño basado en **casos de uso**
 - Método de diseño de **transición de estados**

componente = UNIDAD o conjunto de UNIDADES



Recuerda que "vemos" el sistema desde el punto de vista del DESARROLLADOR!!

- Diseño basado en **CASOS DE USO**:
 - Permiten probar las interacciones entre componentes
 - Cada caso de uso se implementa utilizando varios componentes del sistema. Dichas interacciones se muestran en los diagramas de secuencia asociados al caso de uso
 - La prueba de un caso de uso "fuerza" el que se lleven a cabo las diferentes interacciones entre los componentes implicados



DISEÑO DE PRUEBAS DE SISTEMA

P

Sobre el diseño de basado en **casos de uso**:

- Los diagramas de secuencia nos ayudan a diseñar los casos de prueba adecuados, ya que muestran qué entradas se requieren y qué salidas deben producirse entre los componentes
- Puesto que no podemos realizar pruebas exhaustivas, ¿Cuántos casos de prueba necesitamos diseñar para dar eficiencia a nuestro proceso de pruebas? En este caso, se deben fijar políticas de pruebas para elegir de forma adecuada un sub-conjunto de pruebas efectivo, como por ejemplo:
 - Todas las funciones del sistema que sean accedidas desde menús deben ser probadas
 - Combinaciones de funciones que sean accedidas a través del mismo menú, deben ser probadas
 - Siempre que se proporcionan entradas de usuario, se deben probar entradas correctas e incorrectas
 - Las funciones más utilizadas en un uso normal, deben ser las más probadas

S

S

Método de diseño de **transición de estados**:

- Se usa en sistemas con ESTADO. Un estado viene dado por un conjunto de valores de variables del sistema.
 - Por ejemplo, un sistema de gestión de pedidos (en donde un pedido puede estar "iniciado", "cancelado", "procesado", "pendiente", ...)
- En este caso, a partir de la especificación se obtiene una representación en forma de grafo (denominado diagrama de transición de estados). Dicho grafo modela los estados de una entidad del sistema, representados en sus nodos. Las aristas representan las transiciones entre estados.
- A partir del grafo, se selecciona un conjunto de caminos mínimo para conseguir un objetivo concreto. Por ejemplo: recorrer todas las transiciones del grafo.
- Finalmente se determinan los datos de entrada y salida esperados para los comportamientos que ejercitan dicho conjunto de caminos

ACCEPTANCE TESTING

El sistema se prueba desde el "punto de vista" del usuario

- Un producto está listo para ser entregado (delivered) al cliente después de que se hayan realizado las pruebas del sistema
 - A continuación, los clientes ejecutan los tests de **aceptación** basándose en sus expectativas sobre el producto. Finalmente, si los tests de aceptación son "aceptados" por el cliente, el sistema será puesto en PRODUCCIÓN.
- Las pruebas de aceptación son pruebas formales orientadas a determinar si el sistema satisface los criterios de aceptación (**acceptance criteria**)
 - Los **criterios de aceptación** de un sistema deben satisfacerse para ser aceptados por el cliente (éste generalmente se reserva el derecho de rechazar la entrega del producto si los tests de aceptación no "pasan")
- Hay dos categorías de pruebas de aceptación:
 - **User acceptance testing (UAT)**
 - * Son **dirigidas por el cliente** para asegurar que **el sistema** satisface los criterios de aceptación contractuales (pruebas α y pruebas β)
 - **Business acceptance testing (BAT)**
 - * Son **dirigidas por la organización** que desarrolla el producto para asegurar que el sistema eventualmente "pasará" las UAT. Son un ensayo de las UAT en el lugar de desarrollo

en el lugar de
desarrollo y para
usuarios "conocidos"

para usuarios
"anónimos"

Los criterios de aceptación se definen en etapas tempranas del desarrollo, pero los probamos al final del desarrollo, y después de haber verificado!!!

ACCEPTANCE CRITERIA

Los criterios de aceptación se refieren a **TODO** el sistema, no a una parte del mismo

- Los criterios de aceptación deben ser DEFINIDOS y ACORDADOS entre el proveedor (organización a cargo del desarrollo) y el cliente
 - Constituyen el "núcleo" de cualquier acuerdo contractual entre el proveedor y el cliente
- Una cuestión clave es: ¿Qué criterios debe satisfacer el sistema para ser aceptado por el cliente?
 - El principio básico para diseñar los criterios de aceptación es asegurar que la calidad del sistema es aceptable
 - Los criterios de aceptación deben ser medibles, y por lo tanto, cuantificables
- Algunos **atributos de calidad** que pueden formar parte de los criterios de aceptación son:
 - **Corrección funcional y Completitud**
 - * ¿El sistema hace lo que se quiere que haga? ¿Todas las características especificadas están presentes?
 - **Exactitud**, integridad de datos, **rendimiento**, **fiabilidad** y disponibilidad, mantenibilidad, robustez, confidencialidad, escalabilidad,...

P

PROPIEDADES EMERGENTES

S

Sólo son visibles después de haber integrado TODO el sistema

P

- Cualquier atributo incluido en los criterios de aceptación es una **propiedad emergente**
- Las propiedades "emergentes" son aquellas que no pueden atribuirse a una parte específica del sistema, sino que "emergen" solamente cuando los componentes del sistema han sido integrados, ya que son el resultado de las complejas interacciones entre sus componentes. Por lo tanto, no pueden "calcularse" directamente a partir de las propiedades de sus componentes individuales
- Hay dos tipos de propiedades emergentes:
 - **Funcionales**: ponen de manifiesto el propósito del sistema después de integrar sus componentes
 - **No funcionales**: relacionadas con el comportamiento del sistema en su entorno habitual de producción (p.ej. fiabilidad, seguridad...)
- En esta sesión nos vamos a centrar en las pruebas de propiedades emergentes FUNCIONALES

Nos centraremos en las propiedades FUNCIONALES, pero recuerda que diseñaremos las pruebas sin tener en cuenta consideraciones técnicas o detalles internos de la aplicación. Consideraremos siempre el punto de vista del usuario.

DISEÑO DE PRUEBAS DE ACEPTACIÓN

Los casos de prueba requieren (como siempre) datos de entrada y resultados esperados CONCRETOS



- Deberían ser responsabilidad de un grupo separado de pruebas que no esté implicado en el proceso de desarrollo. Se trata de determinar que el sistema es lo suficientemente "bueno" como para ser usado (entregado al cliente, o ser lanzado como producto): cumple los criterios de aceptación
- Normalmente se trata de un proceso black-box (functional testing) basado en la especificación del sistema. También reciben el nombre de "pruebas funcionales", para indicar que se centran en la "funcionalidad" y no en la implementación
- Ejemplos de **métodos de diseño** de pruebas emergentes funcionales:
 - Diseño de pruebas basado en **requerimientos**
 - * son pruebas de validación (se trata de demostrar que el sistema ha implementado de forma adecuada los requerimientos y que está preparado para ser usado por el usuario). Cada requerimiento debe ser "testable"
 - Diseño de pruebas de **escenarios**
 - * los escenarios describen la forma en la que el sistema debería usarse

Los DATOS de entrada pueden incluir "secuencias" de acciones llevadas a cabo por el usuario. Nuestros drivers tendrán muchas más líneas de código!!

DISEÑO DE PRUEBAS BASADO EN REQUERIMIENTOS (I)

Capítulo 8.3.1 "Software Engineering" 9th. Ian Sommerville

- Un principio general de una buena especificación de un requerimiento es que debe escribirse de forma que se pueda diseñar una prueba a partir de él
 - [610-12-1990-IEEE Standard Glossary of Software Engineering Terminology]. A requirement is:
 1. A condition or capability needed by a user to solve a problem or achieve an objective
 2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
 3. A documented representation of a condition or capability as in 1 or 2.
- Ejemplo de requerimiento “testable”

If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.

If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

- Ejemplo de casos de prueba que podemos derivar del requerimiento anterior:
 1. Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system

DISEÑO DE PRUEBAS BASADO EN REQUERIMIENTOS (II)

- Ejemplo de casos de prueba que podemos derivar del requerimiento anterior (continuación):

2. Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system.
 3. Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
 4. Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
 5. Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.
- Fijaos que para un ÚNICO requerimiento necesitaremos VARIOS tests para asegurar que "cubrimos" todo el requerimiento

DISEÑO DE PRUEBAS BASADO EN ESCENARIOS (I)

Capítulo 8.3.2 "Software Engineering" 9th. Ian Sommerville

- Un escenario es una descripción de un ejemplo de interacción del usuario/s con el sistema. Un escenario debería incluir:

- Una descripción de las asunciones iniciales, una descripción del flujo normal de eventos, y de situaciones excepcionales; y una descripción del estado final del sistema cuando el escenario termine

- Ejemplo de escenario:

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, Kate logs into the MHC-PMS and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. Kate looks up Jim's record and is prompted for her key phrase to decrypt the record. She checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so she notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. He agrees so Kate enters a prompt to call him when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim's record.

After, finishing her consultations, Kate returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for Kate of those patients who she has to contact for follow-up information and make clinic appointments.

DISEÑO DE PRUEBAS BASADO EN ESCENARIOS (II)



- El escenario anterior describe las siguientes características (requisitos o requerimientos) de nuestro sistema:
 - Authentication by logging on to the system.
 - Downloading and uploading of specified patient records to a laptop
 - Home visit scheduling
 - Encryption and decryption of patient records on a mobile device
 - Record retrieval and modification
 - Links with the drugs database that maintains side-effect information
 - The system for call prompting
- Las pruebas basadas en escenarios normalmente prueban varios requerimientos en un mismo escenario. Por ello, además de probar los requerimientos individuales, también estamos probando la combinación de varios de ellos
 - El diseño de pruebas resultante se obtiene agregando los casos de prueba que tengan en cuenta los requerimientos anteriores: el tester, cuando ejecuta este escenario, adopta el rol de Kate, y debe contemplar situaciones como introducir credenciales erróneas, o información incorrecta sobre el paciente

PRUEBAS DE PROP. EMERGENTES FUNCIONALES: SELENIUM

<https://www.selenium.dev>



■ Las pruebas de propiedades emergentes funcionales tienen como objetivo el comprobar que el sistema ofrece la **FUNCIONALIDAD** esperada por el cliente

- ◆ Se diseñan con técnicas de caja negra, y prueban la funcionalidad del sistema a través de la interfaz de usuario
 - ▶ Diseño de pruebas basado en requerimientos
 - ▶ Diseño de pruebas basado en escenarios

■ **Selenium** es un conjunto de herramientas de pruebas open-source para automatizar pruebas de propiedades emergentes funcionales sobre **aplicaciones Web** (o cualquier aplicación cuyo cliente sea el navegador)

- ◆ **Selenium WebDriver** (API): Permiten crear tests robustos, que pueden escalarse y distribuirse en diferentes entornos 
- ◆ **Selenium IDE** (extensión de un navegador: Firefox, Chrome). Permite crear scripts de pruebas utilizando la aplicación web tal y como un usuario haría normalmente: a través del navegador. 
- ◆ **Selenium Grid**: es un servidor proxy que permite ejecutar tests en paralelo usando múltiples máquinas y diferentes navegadores. 

P SELENIUM IDE

P

Nombre del proyecto

Un proyecto Selenium está formado por un conjunto de Tests Cases.

Podemos agrupar los Tests Cases (en una Test Suite). Nuestro proyecto puede estar formado por Tests Cases y/o Tests Suites

Podemos añadir/borrar/modificar los Test Cases y/o Tests Suites

Ventana de ayuda

Podemos mostrar:

- la "traza" de ejecución de los tests (Log),
- la documentación de referencia de los comandos Selenese (Reference)

The screenshot shows the Selenium IDE interface with several annotations:

- Ejecución de los tests**: A red box highlights the top navigation bar with the text "Project: Prueba*".
- Botón de grabación**: A red arrow points to the "REC" button in the toolbar.
- Editor de scripts**: A red arrow points to the bottom section where commands like "open", "set window size", and "click" are listed.
- Detalle de comandos**: A red arrow points to the expanded view of the third click command, showing its target and value.
- Log**: A red arrow points to the "Log" tab in the bottom-left corner, which displays the log output: "Running 'Prueba'".
- Reference**: A red arrow points to the "Reference" tab in the bottom-left corner, which displays the log output: "Running 'Prueba'".

The main table shows the following test steps:

	Command	Target	Value
1	open	/	
2	set window size	1012x693	
3	click	linkText=ESTUDIOS	
4	click	css=#wrappe rContenidoM	

The bottom status bar shows a warning: "Warning Element found with secondary locator css=#enlaceMenuHeader3 > a. To use it by default, update the test step to use it as the primary locator."

EJEMPLO

Abrimos SeleniumIDE, fijamos <https://web.ua.es/dccia/>, como url base, y pulsamos el botón grabar

1+2. Accederemos directamente a <https://web.ua.es/dccia/>

3+4. Seleccionamos el texto "Horario" en la página, y con botón derecho seleccionamos el comando "verifyText ... "

5+6. Desplegamos el elemento "COMPONENTES" y Pinchamos sobre "PROFESORADO"

7. Pinchamos sobre "Alfonso Galipienso, María Isabel"

8+9. Seleccionamos el texto "2516" y con botón derecho "verifyText ... "

Selenium IDE - Prueba*

Project: Prueba*

Tests + https://web.ua.es/dccia/

Test 1	Command	Target	Value
Test 2*	1 open	https://web.ua.es/dccia/	
	2 set window size	1221x810	
	3 click	css=.columna_1 > .subapartado	
	4 verify text	css=.columna_1 > .subapartado	Horario de secretaría:
	5 click	css=#menu .desplegable:nth-child(3) > .sprite	
	6 click	linkText=Profesorado	
	7 click	linkText=Alfonso Galipienso	
	8 click	css=.bloque_datos:nth-child(4) li:nth-child(2)	
	9 verify text	css=.bloque_datos:nth-child(4) li:nth-child(2)	teléfono +34 965903400 x 2516

Log Reference

Running 'Test 2'

1. open on https://web.ua.es/dccia/ OK 17:56:48

2. setWindowSize on 1221x810 OK 17:56:48

...

6. click on linkText=Profesorado OK 17:56:52

7. click on linkText=Alfonso Galipienso OK 17:56:53

8. click on css=.bloque_datos:nth-child(4) li:nth-child(2) OK 17:56:54

9. verifyText on css=.bloque_datos:nth-child(4) li:nth-child(2) with value teléfono +34 965903400 x 2516 OK 17:56:55

'Test 2' completed successfully 17:56:56

Traza de ejecución y resultado del test

P CÓDIGO DE LOS DRIVERS (TESTS CASES) S

Se guardan en formato JSON

	Command	Target	Value
1	open	https://web.ua.es/dccia/	
2	set window size	1221x810	
3	click	css=.columna_1 > .subapartado	
4	verify text	css=.columna_1 > .subapartado	Horario de secretaría:
5	click	css=#menu .desplegable:nth-child(3) > .sprite	
6	click	linkText=Profesorado	
7	click	linkText=Alfonso Galipienso	

```
...
{
  "id": "1d6c87e5-e123-4e90-bd32-75acef0f486b",
  "name": "Test 2",
  "commands": [
    {
      "id": "1e33d944-1a66-4c6b-977f-5e883524e1c8",
      "comment": "",
      "command": "open",
      "target": "https://web.ua.es/dccia/",
      "targets": [],
      "value": ""
    },
    ...
  ],
  ...
}
{
  "id": "5b2db81d-6950-43f1-b37e-e2ed99b29254",
  "comment": "",
  "command": "click",
  "target": "css=.columna_1 > .subapartado",
  "targets": [
    ["css=.columna_1 > .subapartado", "css:finder"],
    ["xpath=/div[@id='contenido']/div/div[2]/h5", "xpath:idRelative"],
    ["xpath=/div[2]/h5", "xpath:position"],
    ["xpath=/h5[contains(.,'Horario de secretaría:')]", "xpath:innerText"]
  ],
  "value": ""
} ...
```

El driver se implementa como un script de comandos selenese.

Los comandos se generan automáticamente durante la grabación, pero también podemos generarlos de forma manual

El proyecto se puede guardar en un fichero con extensión .side en formato Json.

La herramienta permite importar proyectos para su edición/modificación/ ejecución

COMANDOS SELENESE

S COMMAND + [TARGET] + [VALUE]

- Un comando Selenese puede tener uno o dos parámetros:
 - **target**: hace referencia a un elemento HTML de la página a la que se accede. (p.ej "link" indica un hipernlace)
 - **value**: contiene un texto, patrón, o variable a introducir en un campo de texto o para seleccionar una opción de una lista de opciones
- Una secuencia de comandos Selenese forman un "test script" (caso de prueba). Una secuencia de tests scripts forman una test suite

actions

interactúan directamente con los elementos de la página.

P.ej. "click", "type"

Muchas actions pueden llevar el sufijo AndWait

control flow

alteran el flujo secuencial de ejecución de los comandos.

Pueden crear ramas **condicionales**.

P.ej: "if", "else"

O pueden crear **bucles**. P.ej: "times", "while"

HAY 4 TIPOS DE COMANDOS:

accessors

permiten almacenar valores en variables
P.ej. "storeTitle"

assertions

verifican que se cumple una determinada condición

Hay 3 tipos:

- **Assert**: detienen la ejecución si no se cumple
- **Verify**: se registra el fallo y continúa la ejecución
- **WaitFor**: espera a que se cumpla una condición antes de fallar (el tiempo de espera se indica en milisegundos)

EJEMPLOS DE COMANDOS SELENESE

open: abre una página usando una URL y espera a que se cargue

click: pulsa con el botón izquierdo del ratón sobre un elemento de la página

verify title/assert title: verifica el valor del título de una página

verify text: verifica que un texto esté presente en algún elemento de la página

verify element present: verifica que un elemento, definido por su etiqueta html esté presente en algún sitio de la página

store: almacena un string en una variable

store text: obtiene el texto de un elemento de la página y lo almacena en una variable

type: escribe un texto en un campo html

SINTAXIS:

open url

click locator

verify title text, assert title text

verify text locator text

verify element present locator

store text variableName

store text locator variableName

type locator value

Ver <https://www.seleniumhq.org/selenium-ide/docs/en/api/commands/>

Ver <https://www.seleniumhq.org/selenium-ide/docs/en/introduction/control-flow/>

Ver <https://www.guru99.com/first-selenium-test-script.html>

LOCATORS

Ver <https://www.guru99.com/locators-in-selenium-ide.html>

- Se utilizan en el campo **target** e identifican un elemento en el código de una página web (un botón, un cuadro de texto...). La sintaxis es **locatorType=location**
- Podemos utilizar los siguientes tipos de "locators":
 - **id**: hace referencia al atributo **id** del elemento html
 - **name**: atributo **name** del elemento html. Adicionalmente podemos añadir un "filtro" consistente en añadir otro atributo adicional junto con su valor
 - **xpath**: es el lenguaje utilizado para localizar nodos en un documento HTML
 - * Podemos utilizar rutas absolutas: `xpath=/html/body/form[1]`
 - * o rutas relativas: `xpath=/form[1]`

Command	Target	Value
open	http://demo.guru99.com/test/newtours/	
verifyText	//tr[4]/td/table/tbody/tr[2]/td/font	User*
verifyElementPresent	name=password	
open	http://demo.guru99.com/test/facebook.html	
type	id=email	hola

EJEMPLOS DE LOCATORS (I)

- Suponemos que el código HTML de nuestra página web es:

```

1 <html>
2   <body>
3     <form id="loginForm">
4       <input name="username" type="text" />
5       <input name="password" type="password" />
6       <input name="continue" type="submit" value="Login" />
7       <input name="cancel" type="button" value="Clear" />
8     </form>
9   </body>
10  <html>
```

Usamos un filtro para
refinar la búsqueda

- id=loginForm (línea 3)
- name=username (4)
- name=continue type=button (7)
- xpath=/html/body/form[1] (3) es una ruta absoluta
- //form[1] (3) es una ruta relativa (encuentra el primer "form" en el HTML)
- linkText=Continue (4)
- linkText=Cancel (5)

```

1 <html>
2   <body>
3     <p>Are you sure you want to do this?</p>
4     <a href="continue.html">Continue</a>
5     <a href="cancel.html">Cancel</a>
6   </body>
7   <html>
```

LOCALIZACIÓN DE ELEMENTOS DESDE CHROME

- Podemos obtener información de los elementos de una página web mediante la utilidad "inspeccionar" desde dicha página web con el menú contextual (desde dicho elemento)

1. Situamos el ratón dentro del cuadro de texto y seleccionamos "Inspecionar" desde el menú contextual (botón derecho)

2. En la ventana del inspector se resalta el código del elemento

3. Si nos posicionamos sobre el resultado en azul de la ventana de inspección (2), también se resaltará en elemento en la página

4. En cualquier momento podemos "buscar" cualquier otro elemento activando el botón de inspección

A No es seguro | demo.guru99.com/test/newtours/

cool summer ARUBA

SIGN-ON REGISTER SUPPORT CONTACT

Jul 6, 2017

Find A Flight

Registered users can sign-in here to find the participating air

User Name:

Password:

Submit

Destinations

```
<td align="right">...</td>
<td width="112" style="font-family:Arial, Helvetica; font-size:13px; color:#000; padding:5px;">
<input type="text" name="userNam" size="10"> == $0
</td>
</tr>
<tr>...</tr>
<tr>...</tr>
<tr>...</tr>
</tbody>
</table>
</td>
</tr>
<tr>...</tr>
```

LOCALIZACIÓN DE ELEMENTOS DESDE SELENIUM IDE

P

Project: Prueba*

Tests + DΞ D ⓘ

Search tests... http://demo.guru99.com/test/newtours/

DemoTours*

Test 1

Test 2

1. elegimos el comando

Command → verify element present

4. Se "rellena" al seleccionar el elemento en la página

Target → name=userName

Value

Description

2. Activamos la selección del target en la página



5. También podemos buscar en la página el valor de target. Se resaltará momentáneamente el elemento en la página

3. Al pasar el ratón por la página se resaltan los elementos. Seleccionamos el elemento que nos interese



Log • Reference

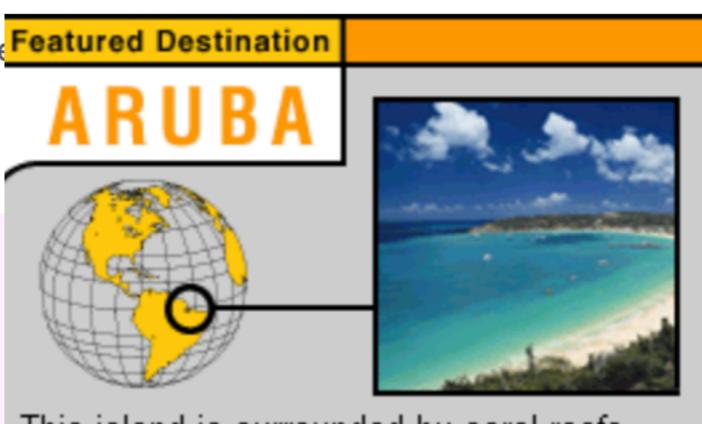
verify element present locator

Soft assert that the specified element is somewhere on the page. The test will continue.

arguments:

locator - An element locator.

Aquí podemos consultar la información sobre el comando



Jul 6, 2017

Find A Flight

Registered users can sign-in here to find the lowest fare on participating airlines.

User Name:

Password:

CONTROL FLOW

P

P

Command	Target	Value
execute script	return "a"	myVar
if	<code> \${myVar} === "a"</code>	
execute script	return "a"	output
else if	<code> \${myVar} === "b"</code>	
execute script	return "b"	output
else		
execute script	return "c"	output
end		
assert	output	a

COMANDOS que controlan el flujo de ejecución:

- if, else if, else, end
- times, end
- do, repeat if
- while, end



<https://www.seleniumhq.org/selenium-ide/docs/en/introduction/control-flow/>

<http://www.cheat-sheets.org/saved-copy/jsquick.pdf>

COMANDO execute script

Parámetros: (sentencia javascript, variable)
Almacena en una variable el resultado de la sentencia o expresión javascript

EJEMPLOS de sentencias Javascript

`return "a"` → devuelve el string "a"

`${myVar} === "a"` → compara el valor de la variable myVar con el string "a"

COMANDO assert

Parámetros: (actual value, expected value)

Convierte el valor de las variables en un string y compara sus valores. Si no coinciden el tests falla

Command	Target	Value
execute script	return 1	check
do		
execute script	return \${check}	check
repeat if	<code> \${check} < 3</code>	
assert	check	3

Y AHORA VAMOS AL LABORATORIO...

Vamos a implementar tests de aceptación (para validar propiedades emergentes funcionales) para una aplicación web con selenium IDE

Camino	Datos Entrada	Resultado Esperado
C1	d1=... d2=... ...	r1
..		
CM	d1=... d2=... ...	rM

Tests aceptación

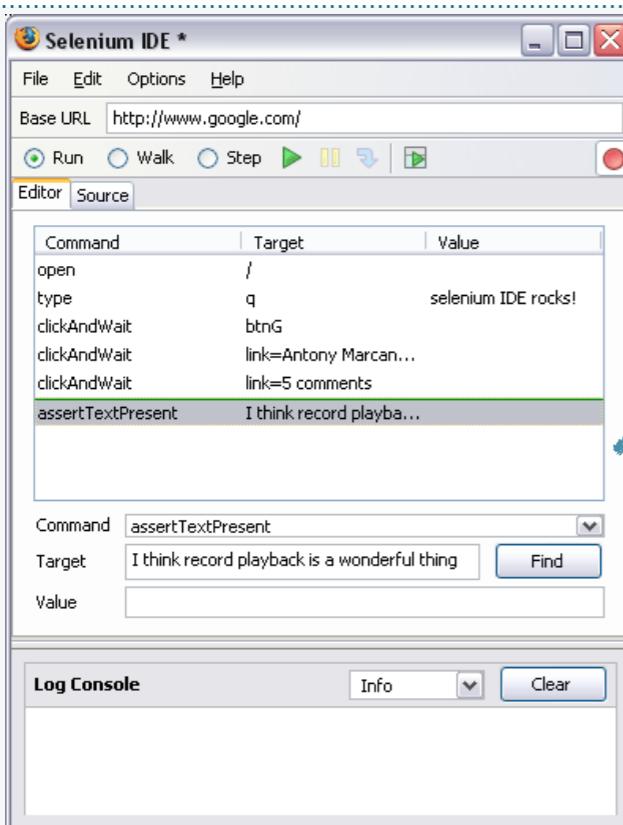


driver

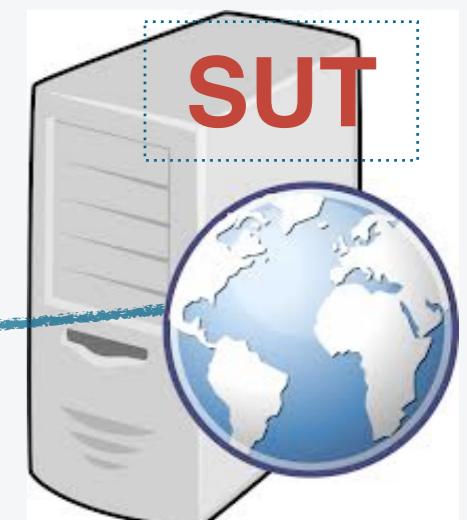
Informe

Navegador FireFox

usaremos scripts Selenese
utilizando Selenium IDE



Sevidor web



P REFERENCIAS BIBLIOGRÁFICAS

- P
 - Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
 - Capítulo 14: Acceptance testing
 - Software Engineering. 9th edition. Ian Sommerville. 2011
 - Capítulo 8.3: Release testing
 - Tutorial Selenium (<http://www.guru99.com/selenium-tutorial.html>)
 - Apartados 1..6

Sesión S07: Pruebas aceptación (2)



Pruebas: de aceptación

- Objetivo: validar las expectativas del cliente
- Propiedades emergentes funcionales
- Automatización:
 - Webdriver (para aplicaciones con interfaz web)
 - Se puede usar también para pruebas del sistema

Webdriver

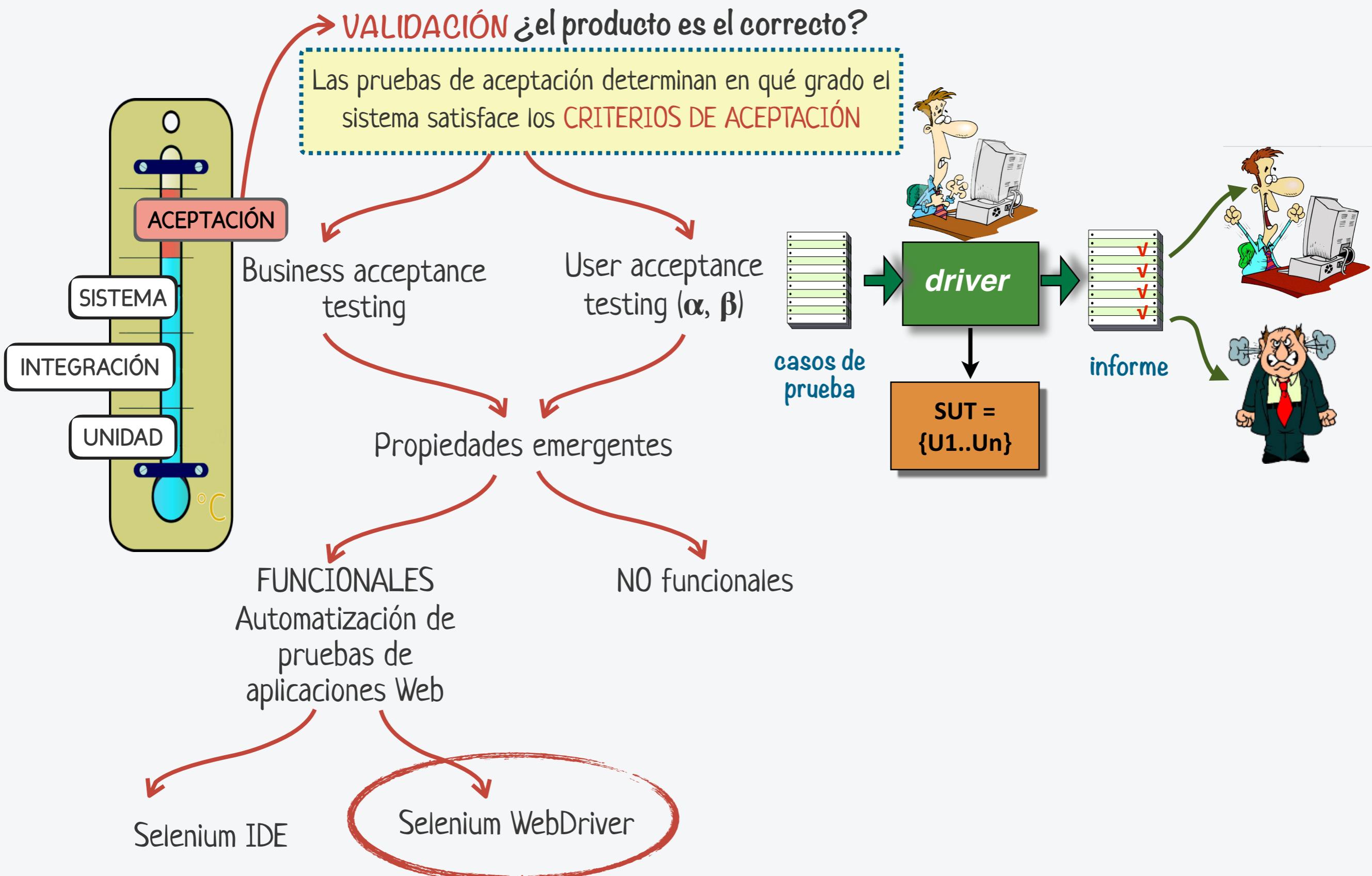
- Mecanismos de localización
- Acciones sobre los WebElements
- Tiempos de espera

Mantenibilidad de nuestros tests

- Patrón Page Object
- Uso de PageFactory y anotaciones @FindBy

Vamos al laboratorio...

PS PRUEBAS DE ACEPTACIÓN



SOBRE SELENIUM IDE



P

- Selenium IDE utiliza el patrón "Record and Playback Pattern"
 - La idea es permitir que el usuario "guarde" (record) las actividades de testing y las pueda ejecutar posteriormente (playback) utilizando la herramienta de pruebas
- Algunas **ventajas** de utilizar Selenium IDE son:
 - Podemos implementar tests más rápidamente, por lo que podemos crear grandes conjuntos de suites en horas en lugar de semanas
 - No se requiere ninguna experiencia previa con lenguajes de programación
 - La búsqueda de elementos en la página es muy fácil y rápida
- Algunos **inconvenientes** de utilizar Selenium IDE son:
 - Tests inflexibles debido a que la ejecución de los tests es idéntica a la grabación de los mismos (¿qué ocurre si necesitamos ejecutar cada test con un usuario diferente cada vez?)
 - Duplicación de código: aunque se ha incorporado un comando para reutilizar reutilizar el código de los tests, no se pueden parametrizar
 - No soporta la gestión de errores ni se pueden integrar los tests en el proceso de construcción del proyecto
- Las limitaciones indicadas se pueden superar utilizando algún lenguaje de programación. **WebDriver** nos permite utilizar varios lenguajes, entre ellos, java, para programar los tests de pruebas de propiedades emergentes **funcionales** sobre aplicaciones web en diferentes navegadores

CARACTERÍSTICAS DE SELENIUM WEBDRIVER



- Proporciona un buen control del navegador a través de implementaciones específicas para cada uno de ellos
- Permite realizar una programación más flexible de los tests

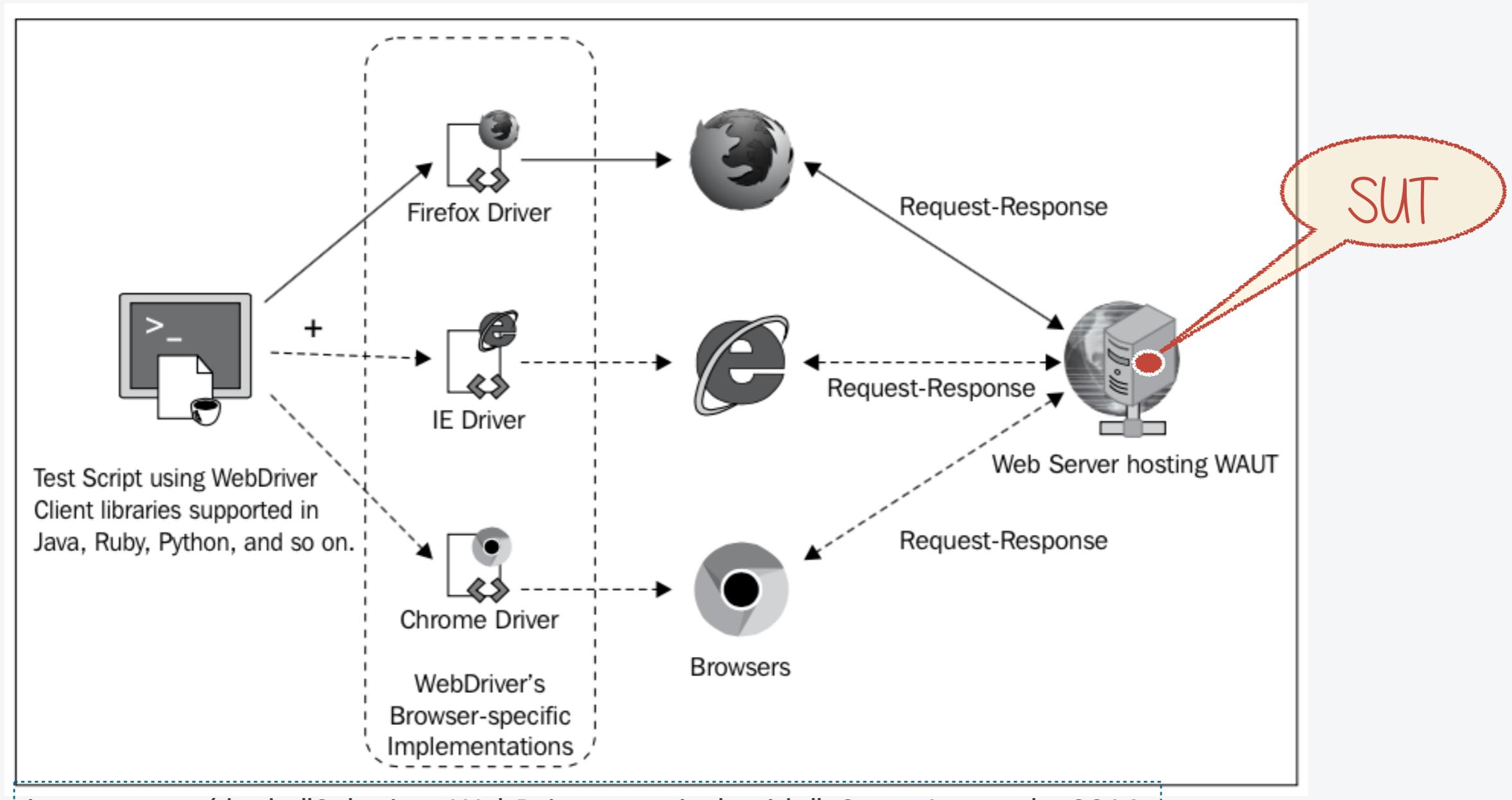
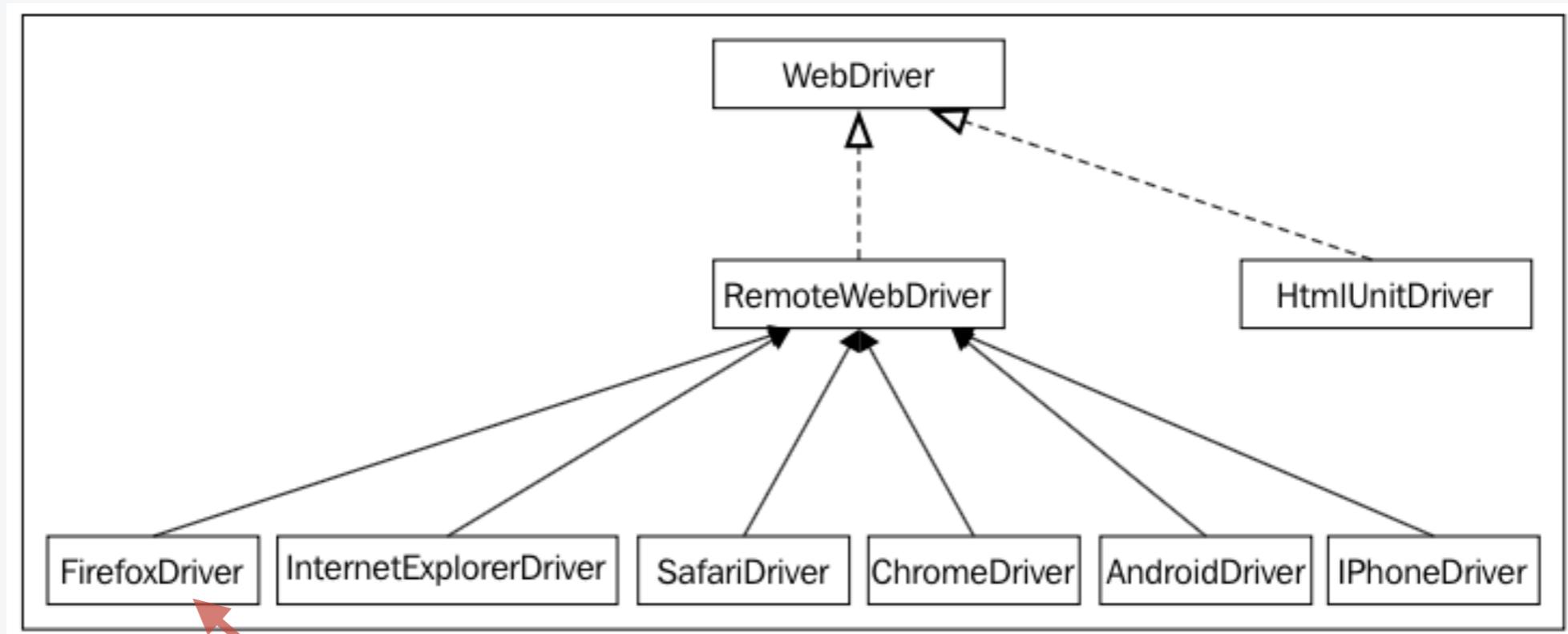


Imagen extraída de "Selenium WebDriver practical guide". Satya Avasarala. 2014.

WAUT: Web Application Under Test

WEBDRIVER INTERFACE Y WEBELEMENTS

- WebDriver es una interfaz cuya implementación concreta la realizan dos clases: RemoteWebDriver y HtmlUnitDriver



```
WebDriver driver = new ChromeDriver();           → "driver" representa el navegador
driver.get("http://www.google.com");             → abrimos una URL
WebElement searchBox = driver.findElement(By.name("q")); → localizamos un elemento en la página
searchBox.sendKeys("Packt Publishing");           → interaccionamos con los
searchBox.submit();                            → elementos de la página
```

- Una página web está formada por elementos HTML, que son objetos de tipo **WebElement** en el contexto de WebDriver
- Una vez localizados los WebElements, podremos realizar acciones sobre ellos

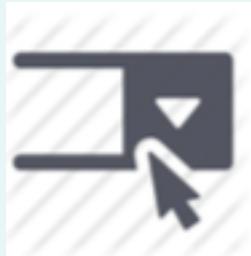
ELEMENTOS HTML MÁS USADOS

P

S

P

Los objetos WebElement representan elementos HTML



<select> <option ...



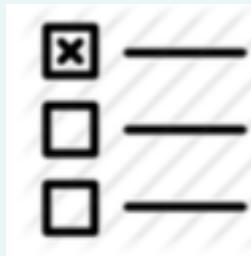
Drop Down



<input type="radio" ...



Radio Button



<input type="checkbox" ...



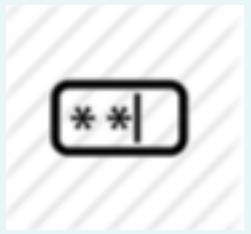
Check Box



<input type="text" ...



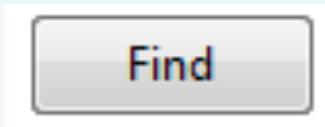
Text Box



<input type="password" ...



Password Box



<input type="button" ...



Button

MECANISMOS DE LOCALIZACIÓN

P

P

S

- Antes de realizar alguna acción (entradas del usuario) con WebDriver, debemos LOCALIZAR el elemento que nos interese. Para ello utilizamos los métodos:

```
WebElement findElement(By by) throws NoSuchElementException  
java.util.List<WebElement> findElements(By by)
```

- Como parámetro de entrada se requiere una instancia de "By", que nos permite localizar elementos en una página web
 - El inspector de elementos de Chrome puede ayudarnos a localizar los elementos HTML de las páginas web cargadas por el navegador. Es tarea del desarrollador el elegir el "locator" adecuado para utilizarlo en findElement()
- Hay 8 formas de **localizar** un WebElement en una página web:
 - By.name(), By.id(), By.tagName(), By.className(), By.linkText(), By.partialLinkText(), By.xpath(), By.cssSelector()
 - Locator **css**: los "css selectors" son patrones de caracteres utilizados para identificar un elemento HTML basado en una combinación de etiquetas HTML, id, class, y otros atributos. Los formatos más comunes para los selectores css son:
 - * css=tag#id → valor de etiqueta seguida de "#" y del valor del atributo id
 - * css=tag.class → valor de etiqueta seguida ":" y del valor del atributo class
 - * css=tag.class[attribute=value]
 - * css=tag[attribute=value]
 - * css=tag:contains("inner text")

EJEMPLOS DE LOCATORS CSS

- Suponemos que el código HTML de nuestra página web es:

```
1 <html>
2   <body>
3     <form id="loginForm">
4       <input class="required" name="username" type="text" />
5       <input class="required passfield" name="password" type="password" />
6       <input name="continue" type="submit" value="Login" />
7       <input name="continue" type="button" value="Clear" />
8     </form>
9   </body>
10  <html>
```

- `css=form#loginForm` (línea 3)
- `css=input.required[type="text"]` (4)
- `css=input[name="username"]` (4)
- `css=input.passfield` (5)

```
<td align="right">
  <font size="2" face="Arial, Helvetica, sans-serif">Password:</font>
</td>
```

- `css=font:contains("Password:")`

PACCIONES SOBRE LOS WEBELEMENTS

- P** **O** Una vez que hemos localizado el elemento que nos interesa, podemos ejecutar **ACCIONES** sobre ellos.

 - Cada tipo de elemento tiene asociado un conjunto diferente de posibles acciones. P.ej. sobre un elemento textbox, podemos introducir un texto o borrarlo
- O** Ejemplos de acciones:

 - sendKeys**(secuencia de caracteres)
 - *** Se utiliza para introducir texto en elementos textbox o textarea
 - clear()** se utiliza para borrar texto en elementos textbox o textarea
 - submit()**
 - *** Puede aplicarse sobre un elemento form, o sobre un elemento que esté dentro de un form. Envía el formulario de la página web al servidor en el que reside la aplicación web
- O** Ejemplos de acciones que pueden ejecutarse sobre **cualquier** WebElement:

 - getAttribute()**, **getLocation()**, **getText()**, **isDisplayed()**, **isEnabled()**, **isSelected()**

EJEMPLOS DE ACCIONES

Imágenes extraídas de:

<http://www.guru99.com/accessing-forms-in-webdriver.html>

ver también: <http://www.guru99.com/locators-in-selenium-ide.html>

Text field

User Name: tutorial

Password: ••••••••

Password field

Introducir texto en un text box y password box

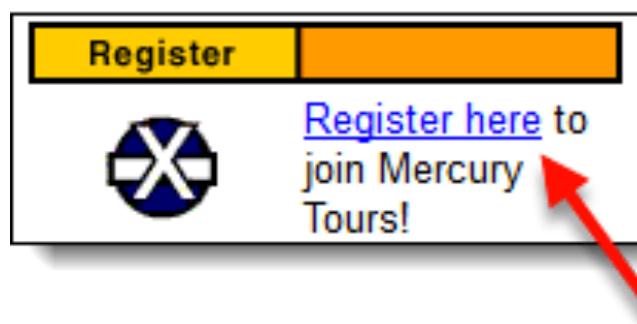
```
driver.findElement(By.name("username")).sendKeys("tutorial");
```

Service Class:

- Economy class
- Business class
- First class

seleccionar un radio box

```
driver.findElement(By.cssSelector("input[value='Business']")).click();
```



pulsar sobre un enlace de texto

```
driver.findElement(By.linkText("Register here")).click();
```

```
driver.findElement(By.partialLinkText("here")).click();
```

Mercury Tours Registration Page

Country: UNITED STATES

```
<td>
  <select size="1" name="country">
  </td>
```

seleccionar elementos en un drop box

```
import org.openqa.selenium.support.ui.Select;
Select drpCountry =
  new Select(driver.findElement(By.name("country")));
drpCountry.selectByVisibleText("ANTARTICA");
```

```
P @Test  
S public void accesoUAC1() {  
P   WebDriver driver = new ChromeDriver();  
S   WebDriverWait wait = new WebDriverWait(driver, 10);  
  
P   driver.get("https://www.ua.es");  
S   //En la página hay dos elementos con el mismo enlace  
P   List<WebElement> enlacesEstudios=driver.findElements(By.linkText("Estudios"));  
S   //queremos acceder al segundo de ellos  
P   enlacesEstudios.get(1).click();  
  
S   WebElement enlaceGrados= driver.findElement(By.linkText("Grados Oficiales"));  
P   enlaceGrados.click();  
  
S   WebElement buscadorAsignaturas= driver.findElement(By.linkText("BUSCADOR DE ASIGNATURAS"));  
P   JavascriptExecutor jse = (JavascriptExecutor) driver;  
S   //Below code will scroll the page till the element is found  
P   jse.executeScript("arguments[0].scrollIntoView()", buscadorAsignaturas);  
S   //ahora ya tenemos el elemento visible y podemos hacer click sobre el  
P   buscadorAsignaturas.click();  
  
S   WebElement campoCodigo = wait.until(presenceOfElementLocated(By.id("TextCodAsi")));  
P   campoCodigo.sendKeys("34027");  
S   WebElement boton=driver.findElement(By.id("ButBuscar"));  
P   //hacemos scroll hasta ver el botón  
S   jse.executeScript("arguments[0].scrollIntoView()", boton);  
P   //ahora ya tenemos el elemento visible y podemos hacer click sobre el  
S   boton.click();  
  
S   WebElement enlacePpss= driver.findElement(By.partialLinkText("PLANIFICACIÓN Y PRUEBAS"));  
P   Assertions.assertEquals("PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE", enlacePpss.getText());  
S }  
ACCESO A LA WEB DE LA UA
```

EJEMPLO DE DRIVER:

ACCESO A LA WEB DE LA UA

TIEMPOS DE ESPERA

- Podemos establecer tiempos de espera en nuestros tests para evitar errores debidos a que no se localiza un elemento en la página porque todavía se esté cargando (excepción NoSuchElementException)
- Tiempo de espera **implícito**: es común a todos los WebElements y tiene asociado un timeout global para todas las operaciones del driver
- Tiempo de espera **explícito**: se establece de forma individual para cada WebElement

```
...
WebDriver driver = new ChromeDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("www.google.com");
...
```

timeout implícito

```
...
//Create Wait using WebDriverWait.
//This will wait for 10 seconds for timeout before
//title is updated with search term
WebDriverWait wait = new WebDriverWait(driver, 10);
wait.until(ExpectedConditions.titleContains("selenium"));
...
```

timeout explícito

Si el elemento se carga antes del límite especificado se cancela el timeout

MÚLTIPLES ACCIONES (GRUPOS DE ACCIONES)

Las acciones agrupadas se ejecutan de forma secuencial

- P
- Podemos indicar a WebDriver que realice múltiples acciones agrupándolas en una acción compuesta, siguiendo estos tres pasos:

- Invocar la clase **Actions** para agrupar las acciones (1)
 - * La clase Actions se utiliza para emular eventos complejos de usuario
- Construir la acción (**Action**) compuesta por el conjunto de acciones anteriores (2)
- Realizar (ejecutar) la acción compuesta (3)

```
WebDriver driver = new ChromeDriver();
driver.get("http://www.example.com");
WebElement one = driver.findElement(By.name("one"));
WebElement three = driver.findElement(By.name("three"));
WebElement five = driver.findElement(By.name("five"));
// Add all the actions into the Actions builder
Actions builder = new Actions(driver); (1)
// Generate the composite action
Action compositeAction = builder.keyDown(Keys.CONTROL)
    .click(one)
    .click(three)
    .click(five)
    .keyUp(Keys.CONTROL)
    .build();
// Perform the composite action.
compositeAction.perform(); (3)
```

En este ejemplo el usuario selecciona los tres elementos (manteniendo pulsada la tecla Ctrl mientras realiza la selección)

EJEMPLOS DE ACCIONES BASADAS EN EL RATÓN



- El método **click()** se utiliza para simular que pulsamos el botón izquierdo del ratón:
 - **public Actions click()**
 - * Pulsación del botón izquierdo del ratón, independientemente o no de que estemos sobre algún elemento de la página
 - * Este método suele usarse combinado con otros, para crear una acción compuesta. P.ej.

```
WebElement one = driver.findElement(By.name("one")); Actions builder = new Actions(driver);
//Click on One
builder.moveToElement(one).click().build().perform();
```

- **public Actions click(WebElement onElement)**
 - * Pulsación del botón izquierdo del ratón sobre un WebElement

```
WebElement one = driver.findElement(By.name("one")); Actions builder = new Actions(driver);
//Click on One
builder.click(one); builder.build().perform();
```

- Método **public Actions moveToElement(WebElement toElement)**

```
WebElement one = driver.findElement(By.name("one")); Actions builder = new Actions(driver);
//Click on One
builder.moveToElement(one).click().build().perform();
```

- Otros métodos que podemos utilizar son:
 - * **public Actions doubleClick();** (doble click con botón izquierdo)
 - * **public Actions contextClick();** (botón derecho del ratón)

EJEMPLOS DE ACCIONES BASADAS EN EL TECLADO

- métodos `keyDown()` y `keyUp()`

- `public Actions keyDown(Keys theKey) throws IllegalArgumentException`
* Se genera una excepción si el argumento no es una de las teclas Shift, Ctrl, Alt
 - `public Actions keyUp(Keys theKey)`

- método `sendKeys()`

- `public Actions sendKeys(CharSequence keysToSend)`
* Se utiliza para teclear caracteres en elementos de la página como text boxes, ...
 - también se puede utilizar el método `WebElement.sendKeys(CharSequence k)`

- Ejemplo:

```
driver = new ChromeDriver();
driver.get(baseUrl);
WebElement linkText = driver.findElement(By.linkText("Element"));

Actions builder = new Actions(driver);
builder.contextClick(linkText) //activamos el menú contextual de linkText
    .sendKeys(Keys.ARROW_DOWN)
    .sendKeys(Keys.ENTER) //seleccionamos la primera de las opciones del menú
    .build()
    .perform();
```

OPERACIONES DE NAVEGACIÓN

- Navegar a la página anterior: `driver.navigate().back()`
- Navegar a la página siguiente: `driver.navigate().forward()`
- Métodos de refresco: `driver.navigate().refresh()`
- Manejo de frames: `driver.switchTo.frame(index)`
- Manejo de ventanas: `driver.switchTo.window(window)`

□ Ejemplo:

- * Si tenemos varias ventanas:

```
driver.get(baseUrl);
String window1 = driver.getWindowHandle();
System.out.println("First Window Handle is: "+window1);

link = driver.findElement(By.linkText("Google Search"));
link.click();
String window2 = driver.getWindowHandle();
System.out.println("Second Window Handle is: "+window2);
System.out.println("Number of Window Handles so far: "
    +driver getWindowHandles().size());
driver.switchTo().window(window1);
```

- * Si tenemos varias pestañas en una única ventana:

```
//Open a new tab using Ctrl + t
driver.findElement(By.cssSelector("body")).sendKeys(Keys.CONTROL +"t");
//Switch between tabs using Ctrl + \t
driver.findElement(By.cssSelector("body")).sendKeys(Keys.CONTROL +"\t");
```

WEBDRIVER Y MAVEN



P

P

- Necesitamos incluir la dependencia con la librería de WebDriver en nuestro proyecto Maven:

```
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
  </dependency>
</dependencies>
```

- ¿Dónde implementaremos nuestros tests de aceptación?
 - Opción 1: en src/test/java, junto con el resto de drivers del proyecto. Serán tests ejecutados por failsafe. En src/main/java tendremos el código fuente de nuestro proyecto
 - Opción 2: en src/test/java de un proyecto maven independiente (el proyecto únicamente contiene los tests de integración). Serán tests ejecutados por surefire. En src/main/java tendremos código que usarán nuestros tests (clases del patrón de diseño page object)

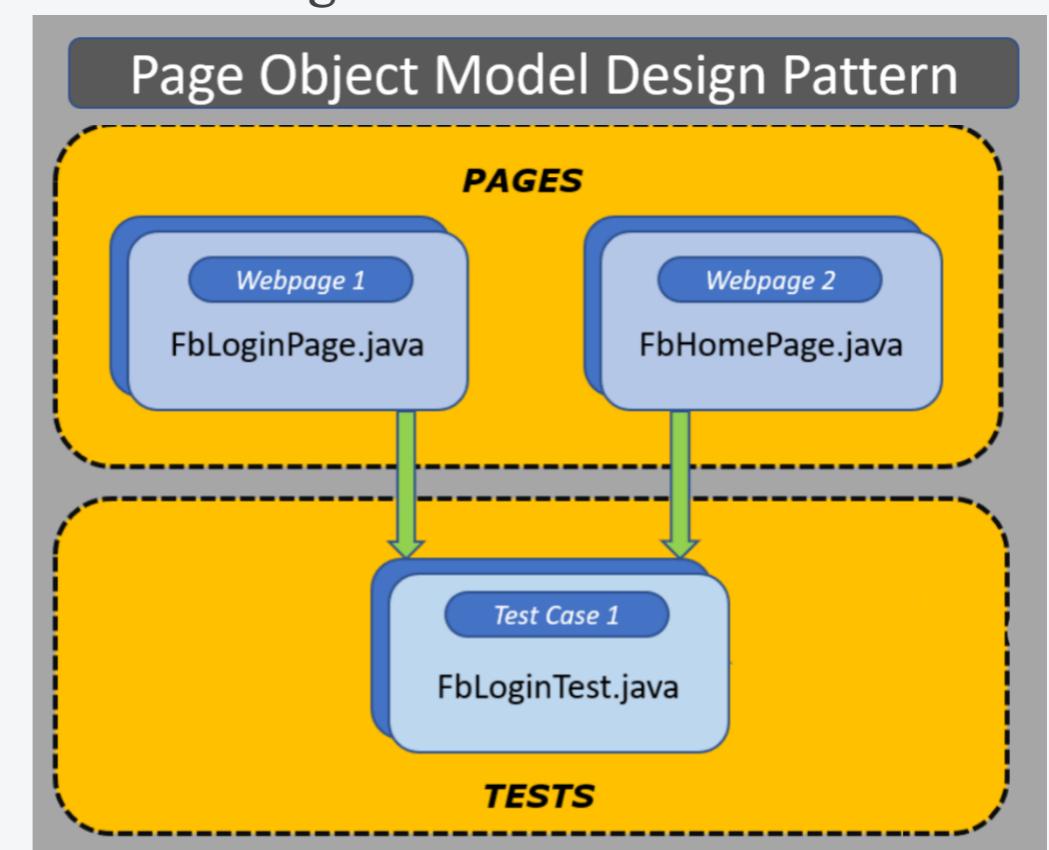
MANTENIBILIDAD DE NUESTROS TESTS

P

- Los tests implementados para nuestra aplicación web, funcionarán siempre y cuando no se produzcan cambios en la vista web (código de las páginas web)
 - Si una o más páginas de nuestra aplicación web sufren cambios, tendremos que cambiar el código de nuestros tests (probablemente en muchos de ellos). P.ej. supongamos que un elemento de la página cambia su ID. Si dicho elemento es accedido desde N tests, tendremos que refactorizar todos ellos
- Para facilitar la mantenibilidad, y reducir la duplicación de código de nuestros tests es útil el patrón de diseño "Page Object Model" (POM)



- La idea es independizar los tests de las páginas html
- Básicamente consiste en crear una clase para cada página web, en la que:
 - * sus miembros (atributos) serán los elementos de la página web correspondiente, y
 - * sus métodos serán todos los SERVICIOS que nos proporciona la página



- El API de Webdriver proporciona varios elementos para implementar este patrón:
 - Anotación **@FindBy** para injectar los objetos que representan los elementos html de una página web
 - Clase **PageFactory** para obtener los objetos que representan las páginas html

PAGE OBJECT PATTERN. EJEMPLO

ver <http://www.guru99.com/page-object-model-pom-page-factory-in-selenium-ultimate-guide.html>

- Supongamos que queremos implementar un test para una aplicación bancaria (<http://guru99.com/V4>)

Guru99 Bank

UserID
Password
LOGIN RESET

Servicio de login

Servicio de reset

- El test consistirá en:
- A. Accedemos a la url inicial de la aplicación
- B. Verificamos que estamos en la página correcta
- C. Nos logueamos con éxito
- D. Verificamos que accedemos a la página correcta

Steps To Generate Access

- Visit - [here](#)
- Enter your email id
- Login credentials is allocated to you and mailed at your id
- Login credentials are only valid for 20 days! So Hurry Up and quickly complete your tasks

Servicio de info de acceso a la aplicación



- Los servicios se implementan como métodos
- Los elementos html se implementan como atributos del objeto PageObject

LoginPage

userID

password

login

reset

info

elementos
html

login()

reset()

servicios

inforegister()

PÁGINA WEB CON LOS SERVICIOS DEL BANCO

Guru99 Bank

Manager

New Customer

Edit Customer

Delete Customer

New Account

Edit Account

Delete Account

Deposit

Withdrawal

Fund Transfer

Change Password

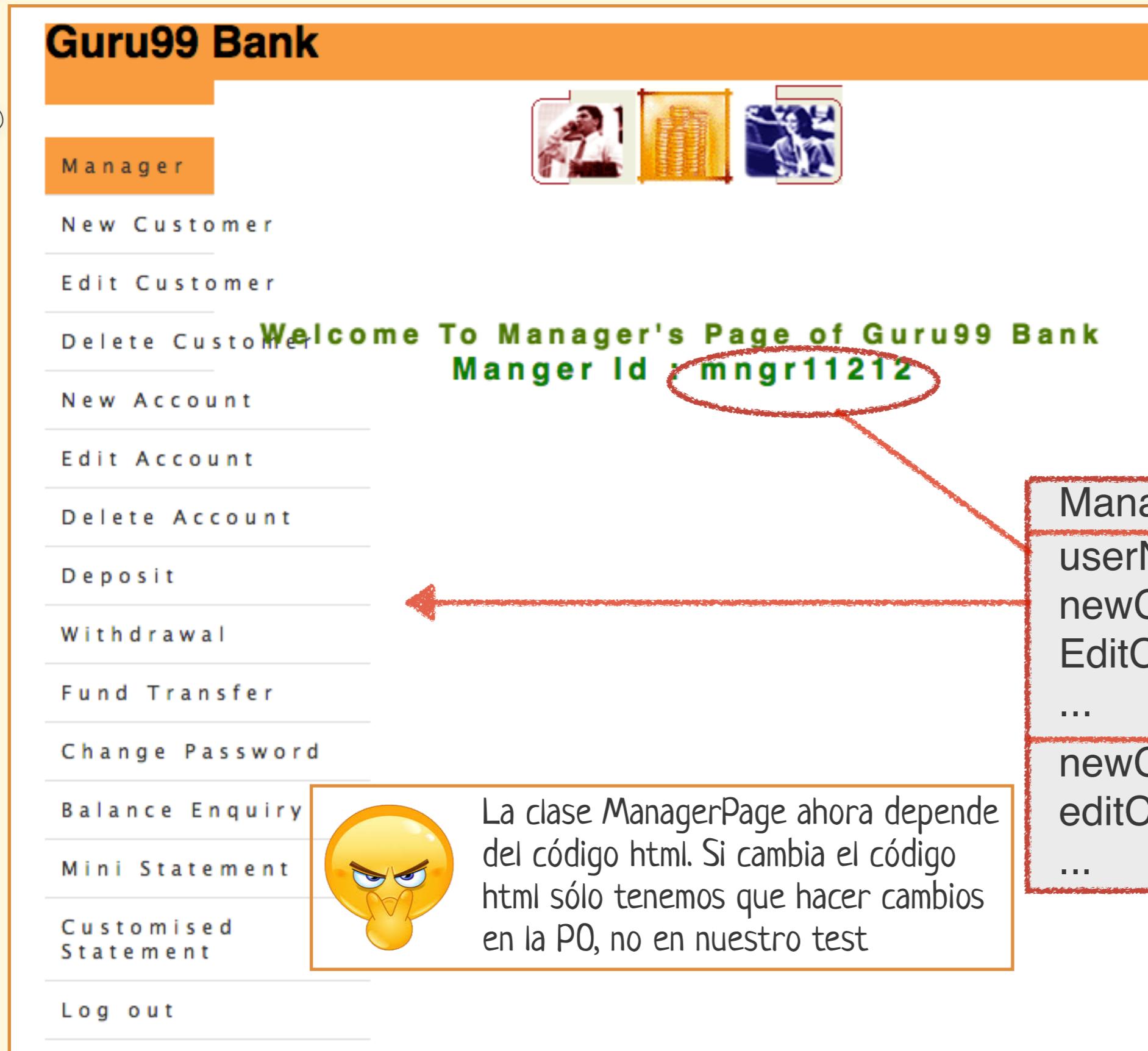
Balance Enquiry

Mini Statement

Customised Statement

Log out

Welcome To Manager's Page of Guru99 Bank
Manger Id : mngr11212



The screenshot shows a web application interface for a bank manager. On the left is a vertical menu with various banking services like Manager, New Customer, and Log out. The main content area displays a welcome message and the manager's ID ('mngr11212'). A red circle highlights the manager ID, and a red arrow points from this circle to a callout box containing explanatory text. The callout box also features an angry emoji.

La clase ManagerPage ahora depende del código html. Si cambia el código html sólo tenemos que hacer cambios en la PO, no en nuestro test



Llamaremos a las clases creadas clases PO (Page Object).

Una PO es una clase que representa una página web.

Creamos una clase PO para cada página html

ManagerPage

userName

newCustomer

EditCustomer

...

newCustomer()

editCustomer()

...

CLASES LOGINPAGE Y MANAGERPAGE

S

P

- Estas clases estarán implementadas en src/main/java

P

```
public class LoginPage {  
    WebDriver driver;  
    WebElement userID;  
    WebElement password;  
    WebElement login;  
    WebElement pTitle;  
  
    public LoginPage(WebDriver driver){  
        this.driver = driver;  
        this.driver.get("http://demo.guru99.com/V4");  
        userID = driver.findElement(By.name("uid"));  
        password =  
            driver.findElement(By.name("password"));  
        login =  
            driver.findElement(By.name("btnLogin"));  
        pTitle =  
            driver.findElement(By.className("barone"));  
    }  
  
    public void login(String user, String pass){  
        userID.sendKeys(user);  
        password.sendKeys(pass);  
        login.click();  
    }  
  
    public String getPageTitle(){  
        return pTitle.getText();  
    }  
}
```

```
public class ManagerPage {  
    WebDriver driver;  
    WebElement homePageUserName;  
    WebElement newCustomer;  
    ...  
    WebElement logOut;  
  
    public ManagerPage(WebDriver driver){  
        this.driver = driver;  
        homePageUserName =  
            driver.findElement(By.xpath("//table//  
                tr[@class='heading3']"));  
        newCustomer =  
            driver.findElement(By.linkText("New Customer"));  
        logOut =  
            driver.findElement(By.linkText("Log out"));  
    }  
  
    public String getHomePageDashboardUserName(){  
        return homePageUserName.getText();  
    }  
    ...  
}
```



Las clases que representan cada una de las páginas contienen código Webdriver y por lo tanto, dependen del código html de nuestra aplicación a probar

CLASE TEST LOGIN PAGE

El test lo implementaremos en src/test/java

```
public class TestLoginPage {  
    WebDriver driver;  
    LoginPage poLogin;  
    ManagerPage poManagerPage;  
  
    @BeforeEach  
    public void setup(){  
        driver = new ChromeDriver();  
        poLogin = new LoginPage(driver);  
    }  
  
    @Test  
    public void test_Login_Correct(){  
  
        String loginPageTitle = poLogin.getLoginTitle();  
        Assertions.assertTrue(loginPageTitle.toLowerCase().contains("guru99 bank"));  
        poLogin.login("mngr34733", "AbEvydU");  
        poManagerPage = new ManagerPage(driver);  
  
        Assertions.assertTrue(poManagerPage.getHomePageDashboardUserName()  
            .toLowerCase().contains("manger id : mngr34733"));  
        driver.close();  
    }  
  
    @AfterEach  
    public void tearDown() {  
        driver.close();  
    }  
}
```



El test **NO** contiene código webdriver, por lo tanto, es independiente del código html de nuestra aplicación a probar



Acuérdate de cerrar el navegador después de cada test

P PAGEFACTORY (I) S

P

La clase PageFactory proporciona objetos de las clases PageObject. Para ello tendremos que:

- anotar los atributos de la clase PageObject con `@FindBy`,
- y utilizar el método estático `PageFactory.initElements()` en el test:
`initElements(WebDriver driver, java.lang.Class PageObjectClass)`

El constructor de la clase PO requiere un parámetro de tipo Webdriver

```
public class LoginPage {  
    WebDriver driver;  
    @FindBy(name="uid") WebElement userID;  
    @FindBy(name="password") WebElement password;  
    @FindBy(name="btnLogin") WebElement login;  
    @FindBy(className="barone") WebElement loginTitle;  
  
    public LoginPage(WebDriver driver){  
        this.driver = driver;  
        this.driver.get("http://demo.guru99.com/V4");  
    }  
  
    public ManagerPage login(String user, String pass) {  
        userID.sendKeys(user);  
        password.sendKeys(pass);  
        login.click();  
  
        return PageFactory.initElements(driver,  
                                         ManagerPage.class);  
    }  
  
    public String getLoginTitle(){ ... }  
}
```



El método `initElements()` invoca al constructor de la clase.

La anotación `@FindBy` inyecta los valores de los atributos de forma "lazy", localizando cada elemento en el momento en el que vamos a usarlo.

```
public class ManagerPage {  
    WebDriver driver;  
    @FindBy(xpath="//table//tr[@class='heading3']")  
    WebElement userName;  
    @FindBy(linkText="New Customer")  
    WebElement newCustomer;  
    @FindBy(linkText="Log out")  
    WebElement logOut;  
  
    public ManagerPage(WebDriver driver){  
        this.driver = driver;  
    }  
    public String getUsername(){ ... }  
}
```

PAGEFACTORY (II)

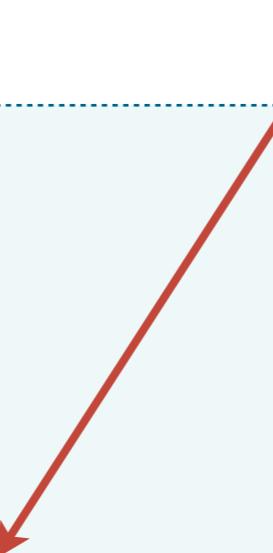
P

S

S

- Usaremos el método PageFactory.initElements en nuestro test para crear una instancia de la clase que representa la página web de la aplicación a probar

```
public class TestLoginPage {  
    WebDriver driver;  
    LoginPage poLogin;  
    ManagerPage poManagerPage;  
  
    @BeforeEach  
    public void setup(){  
        driver = new ChromeDriver();  
        poLogin = PageFactory.initElements(driver, LoginPage.class);  
    }  
  
    @Test  
    public void test_Login_Correct(){  
        String loginPageTitle = poLogin.getLoginTitle();  
        Assertions.assertTrue(loginPageTitle.toLowerCase()  
                            .contains("guru99 bank"));  
        poManagerPage = poLogin.login("mngr34733", "AbEvydU");  
        Assertions.assertTrue(poManagerPage.getUserName()  
                            .toLowerCase()  
                            .contains("manger id : mngr34733"));  
        driver.close();  
    }  
}
```



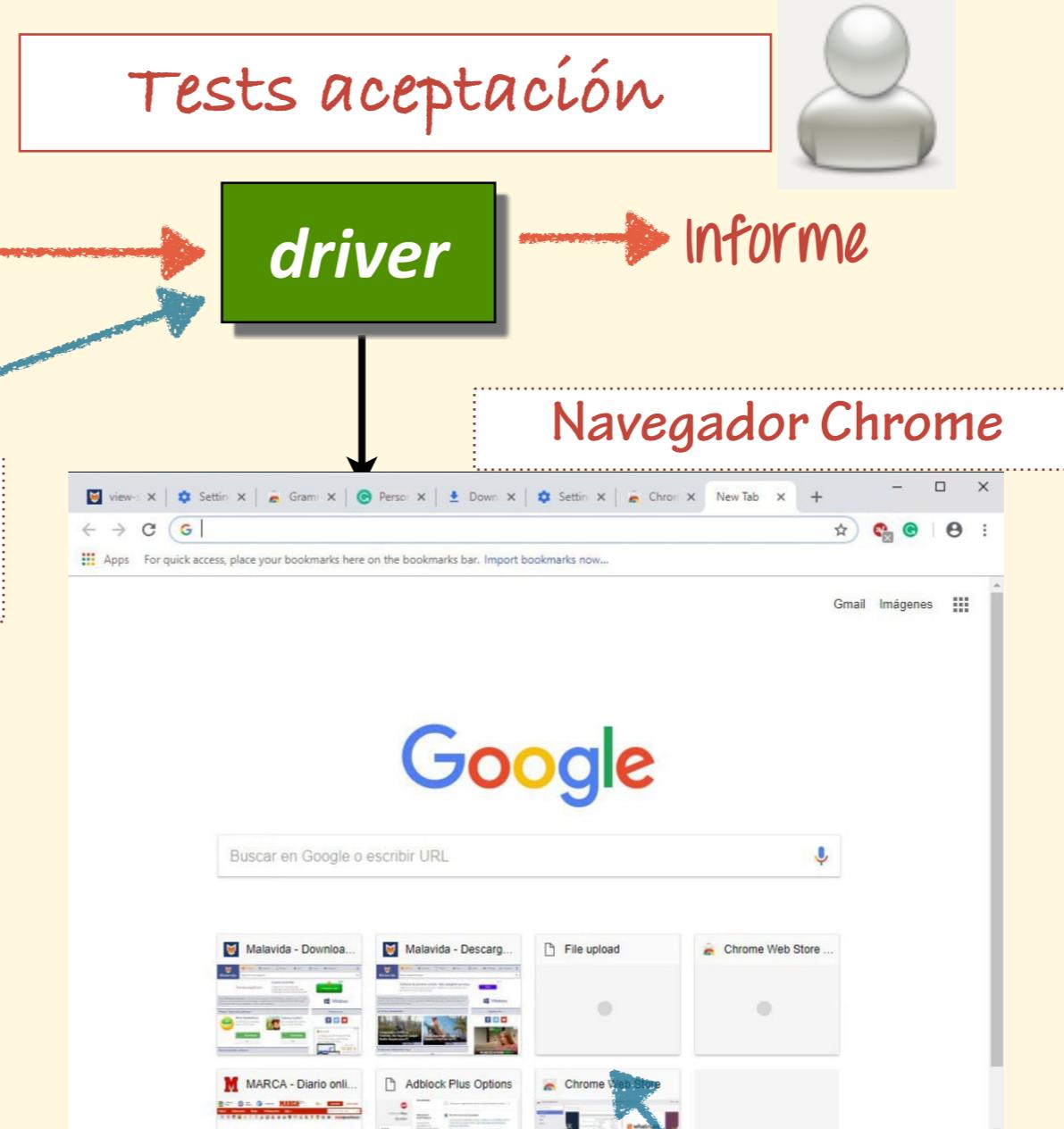
Y AHORA VAMOS AL LABORATORIO...

Vamos a implementar tests de aceptación (para validar propiedades emergentes funcionales) para una aplicación web con selenium WebDriver

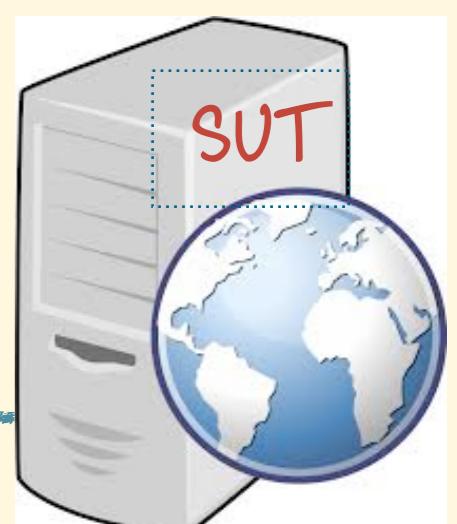
Camino	Datos Entrada	Resultado Esperado
C1	d1=... d2=... ...	r1
..		
CM	d1=... d2=... ...	rM

usaremos JUnit y el API de WebDriver para Chrome

Dado que no disponemos del código de la aplicación web a probar, usaremos un proyecto Maven que sólo va a contener el código de las pruebas de aceptación



Sevidor web



REFERENCIAS BIBLIOGRÁFICAS

- Selenium WebDriver 3 Practical Guide. Unmesh Gundecha and Satya Avasarala. Packt Publishing. 2018
 - Capítulos 1,4 y 9
- Selenium design patterns and best practices : build a powerful, stable, and automated test suite using Selenium WebDriver. Dima Kovalenko, Jim Evans, Jeremy Segal. Packt Publishing, 2014
 - Capítulo 7: The Page Object Pattern
- Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
 - Capítulo 14: Acceptance testing
- Software Engineering. 9th edition. Ian Sommerville. 2011
 - Capítulo 8.3: Release testing
- Tutorial Selenium (<http://www.guru99.com/selenium-tutorial.html>)

PPSS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2020-21



Sesión S09: Pruebas de aceptación (3)

Pruebas de aceptación

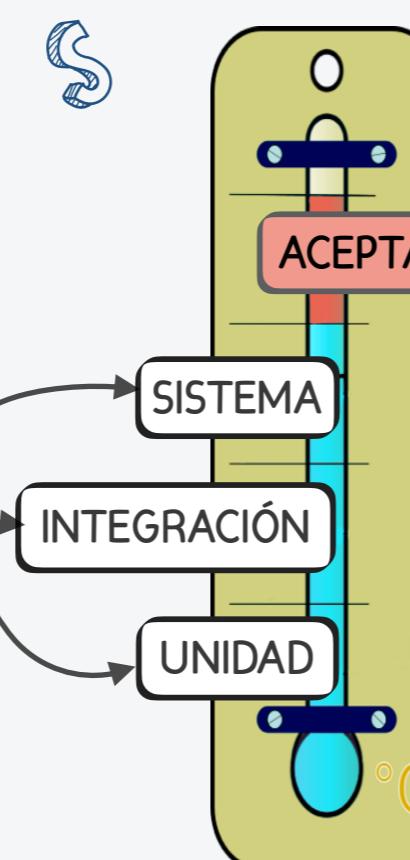
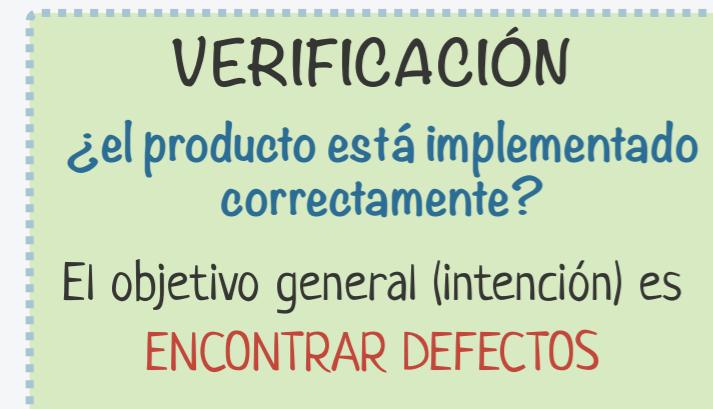
- Propiedades emergentes no funcionales
- Métricas
- Ejemplos de pruebas
 - Pruebas de carga
 - Pruebas de estrés
 - Pruebas estadísticas
- Pasos a seguir durante el proceso de pruebas
- Automatización de las pruebas: JMeter

Vamos al laboratorio...

NIVELES DE PRUEBAS

P

P



VALIDACIÓN
¿el producto es el correcto?

Las pruebas de aceptación determinan en qué **GRADO** el sistema satisface los **CRITERIOS DE ACEPTACIÓN**

Business acceptance testing

User acceptance testing (α, β)

Propiedades emergentes

Cada nivel tiene una granularidad y objetivos concretos diferentes.
Hay un **ORDEN** temporal entre ellos.

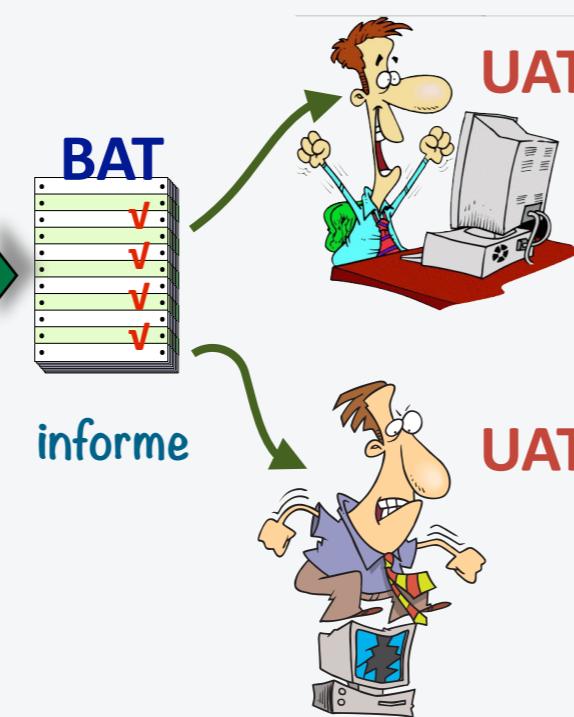
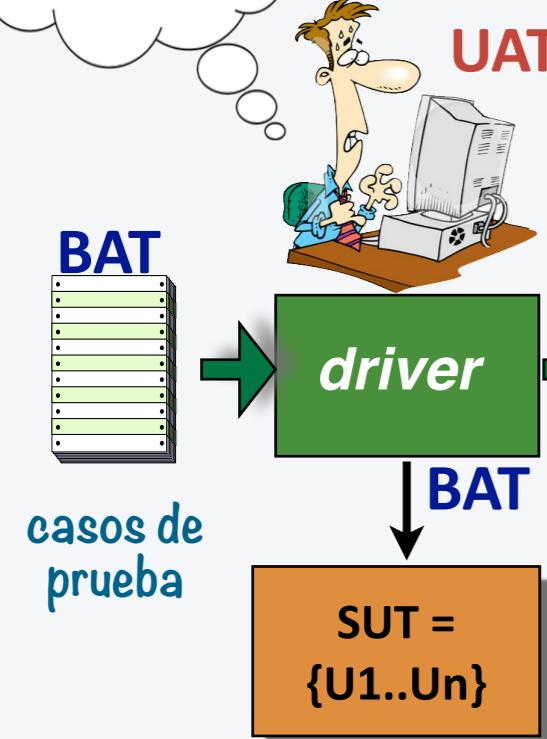
FUNCIONALES
Automatización de pruebas de aplicaciones Web

Selenium IDE

NO funcionales

Selenium WebDriver

JMeter



PROPIEDADES EMERGENTES NO FUNCIONALES

P → Solo tiene sentido aplicarlas al sistema como un TODO

- Hay dos tipos de propiedades emergentes:
 - **Funcionales**: describen lo **que** el sistema hace, o debería hacer (does view)
 - **No funcionales**: hacen referencia a "how well a system performs its functional requirements"
- Muchas de las propiedades emergentes NO FUNCIONALES se categorizan como "-ibilidades":
 - **fiabilidad**: probabilidad de funcionamiento sin fallos durante un tiempo determinado en un entorno específico
 - **disponibilidad**: tiempo durante el cual el sistema proporciona servicio al usuario. Suele expresarse como: hh/dd (p.ej 24/7: 24 horas al día, 7 días por semana)
 - **mantenibilidad**: capacidad de un sistema para soportar cambios. Hay tres tipos de cambios: correctivos, adaptativos y perfectivos
 - * **correctivos**: son provocados por errores detectados en la aplicación
 - * **adaptativos**: son provocados por cambios en el hardware y/o software (sistema operativo) sobre los que se ejecuta nuestra aplicación
 - * **perfectivos**: debidos a que se quiere añadir/modificar las funcionalidades existentes para ampliar/mejorar el "negocio" que sustenta nuestra aplicación
 - **escalabilidad**: hace referencia a la capacidad de mantener el tiempo de respuesta ante cambios en el número de usuarios que utilizan el sistema.

MÉTRICAS

Sí no podemos "medir" No podremos validar!!



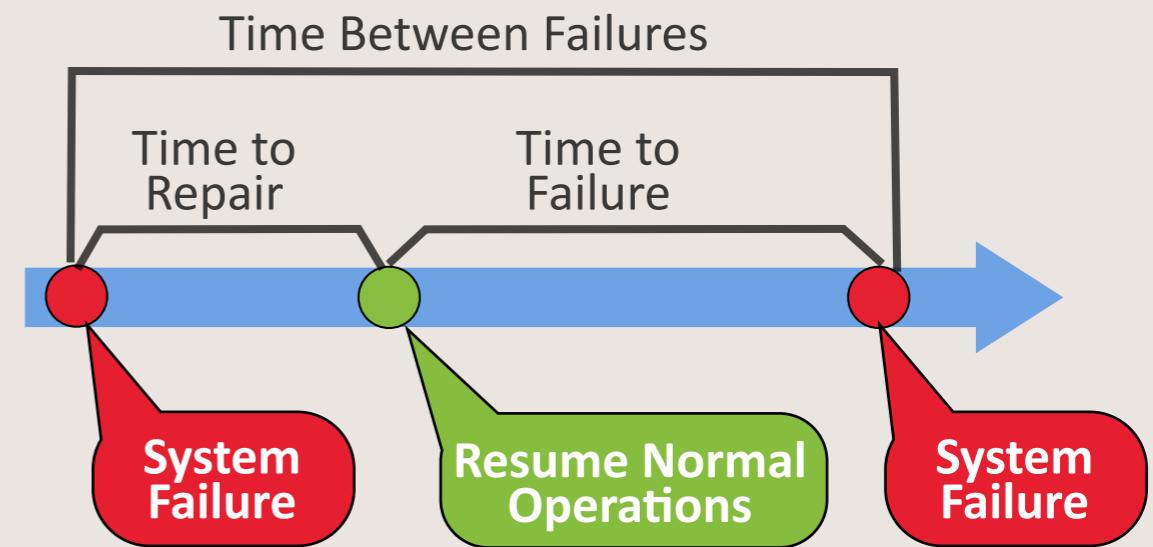
Las pruebas de aceptación determinan en qué **GRADO** el sistema satisface los **CRITERIOS DE ACEPTACIÓN**

O Los criterios de aceptación deben incluir propiedades emergentes "cuantificables"

- Hay que tener mucho cuidado con criterios de aceptación ambiguos, como "Las peticiones se tienen que servir en un tiempo razonable". Dichas sentencias son imposibles de cuantificar y por lo tanto, imposibles de probar (medir) con precisión

O Para juzgar en qué grado se satisfacen los criterios de aceptación se utilizan diferentes métricas:

- Para estimar la **fiabilidad** se utilizan pruebas aleatorias basándonos en un perfil operacional. Se utilizan las métricas MTTF (Mean Time To Failure), MTTR(Mean Time To Repair), y $MTBF = MTTF + MTTR$ (MTBF: Mean time between failures)
- Para estimar la **disponibilidad** se utiliza la métrica MTTR para medir el "downtime" del sistema. La idea es incluir medidas para minimizar el MTTR
- Para estimar la **mantenibilidad** se utiliza la métrica MTTR (que refleja el tiempo consumido en analizar un defecto correctivo, diseñar la modificación, implementar el cambio, probarlo y distribuirlo)
- La **escalabilidad** del sistema utiliza el número de transacciones (operaciones) por unidad de tiempo. Los sistemas suelen poder incrementar su escalabilidad siempre y cuando no sobrepasen limitaciones de almacenamiento de datos, ancho de banda o velocidad de procesador.

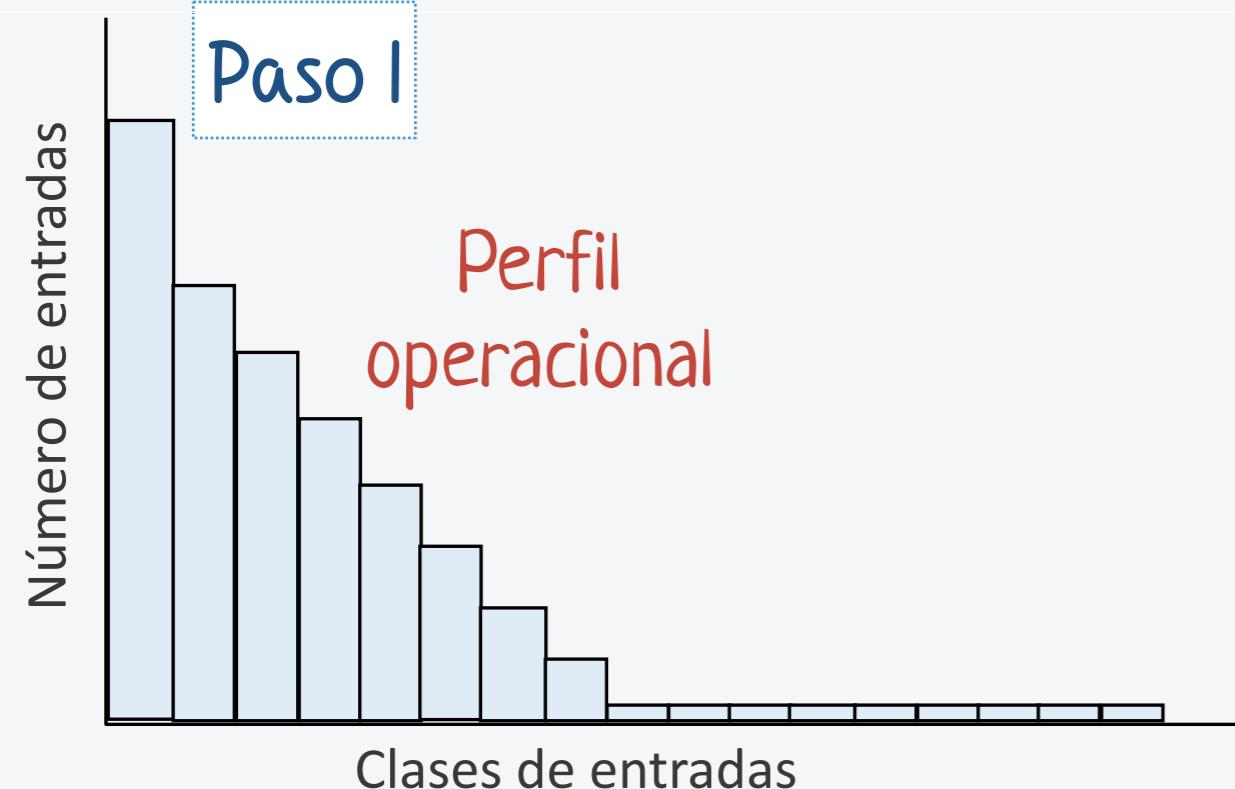


EJEMPLOS DE PRUEBAS (PROCEDIMIENTOS)

Dependiendo de la propiedad a validar,
se usan diferentes **métodos**

- Las **pruebas de carga** validan el **rendimiento** de un sistema en términos de "tratar un número específico de usuarios manteniendo un ratio de transacciones" (p.ej. "una petición del sistema se debe tratar en menos de 2 segundos cuando existen 10000 usuarios dentro del sistema")
- Las **pruebas de stress** consisten en "forzar" peticiones al sistema por encima del límite del diseño del software. Por ejemplo si el sistema se ha diseñado para permitir hasta 300 transacciones por segundo, comenzaremos por hacer pruebas con una **carga** de peticiones inferior a 300 e incrementaremos gradualmente la carga hasta sobrepasar los 300 y ver cuándo falla el sistema
 - Las pruebas de stress comprueban la **fiabilidad** y **robustez** del sistema cuando se supera la carga normal (robustez= capacidad de recuperación del sistema ante entradas erróneas u otros fallos)
- Para evaluar la **fiabilidad** de un sistema podemos utilizar lo que se denominan **pruebas estadísticas**, que consisten en:
 1. construir un "perfil operacional" (operational profile), que refleje el uso real del sistema (patrón de entradas). Como resultado se identifican las "clases" de entradas y la probabilidad de ocurrencia para cada clase, asumiendo un uso "normal" (diseño de los casos de prueba)
 2. generar un conjunto de datos de prueba que reflejen dicho perfil operacional
 3. probar dichos datos midiendo el número de fallos y el tiempo entre fallos, calculando la fiabilidad del sistema después de observar un número de fallos estadísticamente significativo

EJEMPLO DE GENERACIÓN DE PRUEBAS (DISEÑO)



Clase entrada	Distrib. Probab.	Intervalo
C1	50 %	1-49
C2	15 %	50-64
C3	15 %	65-79
C4	15 %	80-94
C5	5 %	95-99

Paso 2

Se generan números aleatorios entre 1 y 99, por ejemplo:

13-94-22-24-45-56-81-19-31-69-45-9-38-21-52-84-86-97...

Se derivan casos de prueba según su distribución de probabilidad:

C1-C4-C1-C1-C1-C2-C4-C1-C1-C3-C1-C1-C1-C2-C4-C4-C5-...



El perfil operacional es la base para el diseño de pruebas emergentes no funcionales

Paso 3

... a continuación deberíamos ejecutar las pruebas midiendo el número de fallos y el tiempo entre fallos

PRESUMEN DEL PROCESO DE PRUEBAS NO FUNCIONALES

- Escalabilidad, fiabilidad, carga... son ejemplos de propiedades emergentes no funcionales. Todas ellas influyen en el **rendimiento** del sistema.
 - En general, las propiedades emergentes no funcionales determinan el **RENDIMIENTO** de nuestra aplicación
- Para evaluar el **RENDIMIENTO**, necesitamos realizar las siguientes actividades:
 1. **Identificar** los criterios de **aceptación**: identificar y cuantificar las propiedades emergentes no funcionales que determinan cuál es el rendimiento aceptable para nuestra aplicación (p.e. tiempos de respuesta, fiabilidad, utilización de recursos...)
 2. **Diseñar** los tests: deberemos conocer el patrón de uso de la aplicación (perfil operacional) para que nuestros casos de prueba estén basados en ESCENARIOS reales de nuestra aplicación
 3. **Preparar** el entorno de pruebas: es importante que el entorno de pruebas sea lo más realista posible
 4. **Automatizar** las pruebas: utilizando alguna herramienta software, "grabaremos" los escenarios de prueba, y ejecutaremos los tests
 5. **Analizaremos** los resultados y realizaremos los cambios oportunos para conseguir nuestro objetivo

CONSIDERACIONES SOBRE EL RENDIMIENTO

- P ○ ¡¡¡ No hay que dejar las pruebas de rendimiento para el final del proyecto !!!
- P ○ Posibles fallos relacionados con el rendimiento pueden conllevar el retocar mucho código
 - Las propiedades emergentes NO funcionales están condicionadas fundamentalmente por la ARQUITECTURA del sistema
 - Normalmente la arquitectura del sistema se determina en las primeras fases del desarrollo!! Eso significa que cambiar la arquitectura puede implicar cambiar "todo".
 - Recuerda que el coste de reparar un error siempre es proporcional al intervalo de tiempo transcurrido desde que se produjo el error, hasta que éste es detectado.
- Minimizaremos los problemas derivados de la validación de las propiedades emergentes no funcionales mediante una buena **estrategia** de pruebas combinada con una arquitectura software que considere el rendimiento desde el inicio del desarrollo
 - Si usamos una metodología de desarrollo iterativa las iteraciones iniciales del proyecto son para construir prototipos exploratorios y comprobar todos los requisitos no funcionales.

JMETER



Ver <http://jmeter.apache.org/usermanual/index.html>

P

- Apache JMeter (<http://jmeter.apache.org/>) es una herramienta de escritorio 100% Java diseñada para medir el rendimiento mediante pruebas de carga
- Permite múltiples hilos de ejecución concurrente, procesando diversos y diferentes patrones de petición
 - JMeter permite trabajar con muchos tipos distintos de aplicaciones. Para cada una de ellas proporciona un **Sampler** o Muestreador, para hacer las correspondientes peticiones

S S

Apache JMeter (5.4)

File Edit Search Run Options Tools Help

Test Plan

00:00:00 ! 0 0/0

EDITOR
Aquí "escribimos" nuestros tests

Podemos cambiar el **NOMBRE** mostrado en el editor de cualquier elemento.
El icono de cada elemento indica de qué **TIPO** es ese elemento

User Defined Variables

Name:	Value

Detail Add Add from Clipboard Delete Up Down

Run Thread Groups consecutively (i.e. one at a time)
 Run tearDown Thread Groups after shutdown of main threads
 Functional Test Mode (i.e. save Response Data and Sampler Data)
Selecting Functional Test Mode may adversely affect performance.

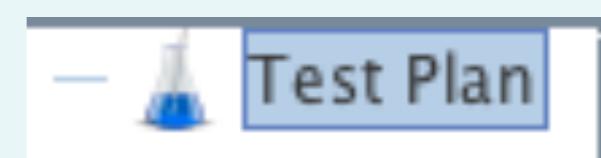
Add directory or jar to classpath Browse... Delete Clear

Library

PANTALLA DE CONFIGURACIÓN del elemento seleccionado en el Editor.
Cada elemento tiene diferentes opciones de configuración

Sesión 9: Pruebas de aceptación 3

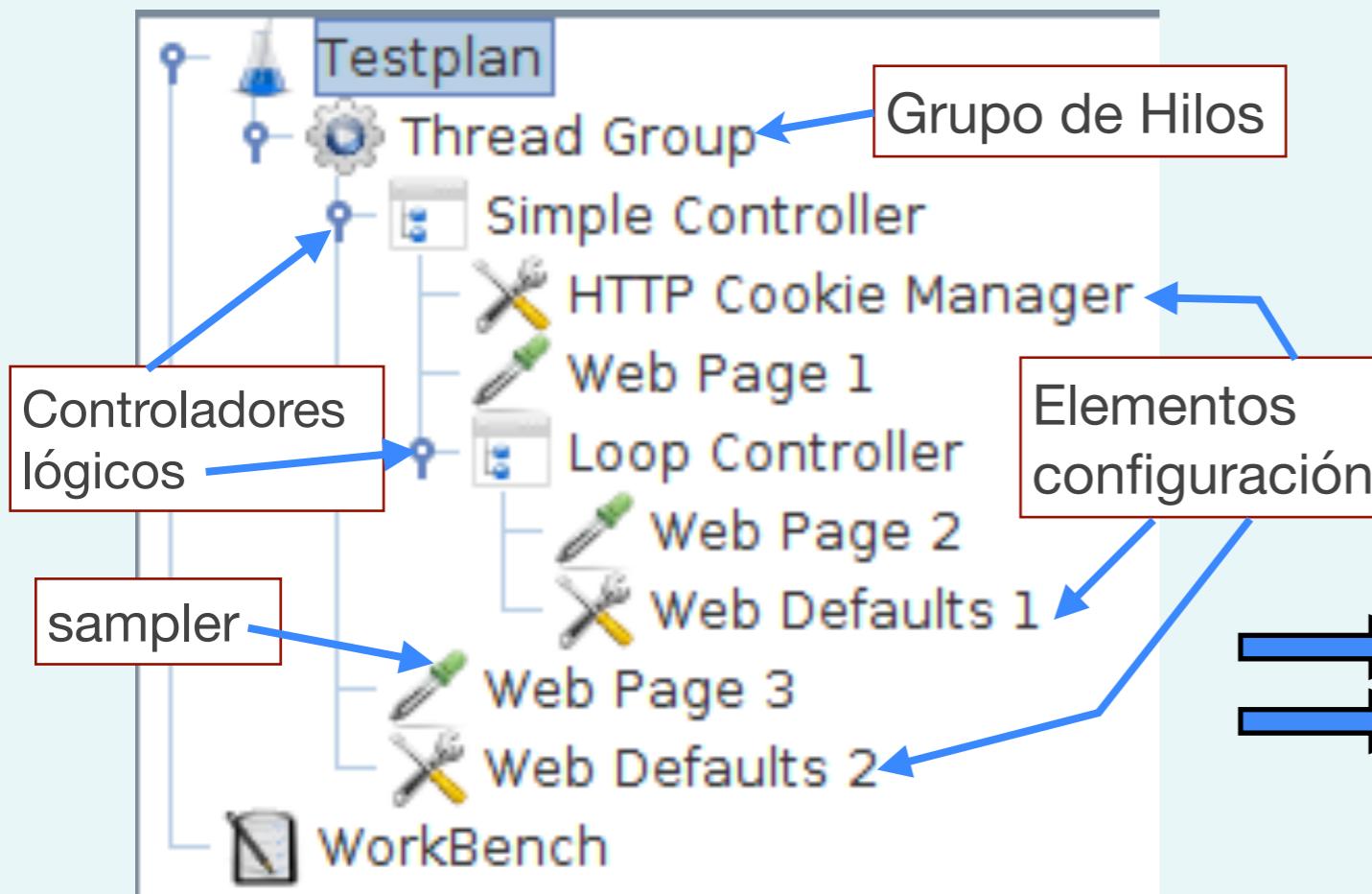
P JMeter: PLAN DE PRUEBAS (I)



https://jmeter.apache.org/usermanual/test_plan.html

- Un plan de pruebas JMeter describe una serie de "pasos" (acciones) que JMeter realizará cuando se ejecute el plan
- Un **plan de pruebas** está formado por:
 - Uno o más grupos de hilos (**Thread Groups**)
 - Controladores lógicos (**Logic Controllers**)
 - **Samplers, Listeners, Timers, Assertions, Pre-Processors, Post-Processors** y
 - Elementos de Configuración (**Configuration Elements**)

Nos centraremos en los elementos que hemos marcado con una flecha azul



El orden de ejecución de los elementos de un plan es el siguiente:

1. Configuration elements ←
2. Pre-Processors
3. Timers ←
4. Sampler ←
5. Post-Processors (unless SampleResult is null)
6. Assertions (unless SampleResult is null)
7. Listeners (unless SampleResult is null)

JMETER: HILOS DE EJECUCIÓN

S



Thread Group

- Un hilo de ejecución es el punto de partida de cualquier plan de pruebas

Thread Group

Name: Thread Group

Comments: Representa un conjunto de usuarios

Action to be taken after a Sampler error

Continue Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 50

Ramp-up period (seconds): 100

Loop Count: Infinite 1

Same user on each iteration

Delay Thread creation until needed

Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

CADA hilo ejecuta COMPLETAMENTE el plan de forma independiente de otros hilos

Cada hilo representa a un usuario



Podemos ejecutar el grupo de hilos un cierto número de veces o de forma indefinida (bucle "infinito")

RAMP-UP (periodo de subida): sirve para que los hilos se creen de forma gradual. Esto permite comprobar cómo rinde el servidor conforme crece la carga.

Por ejemplo, si el periodo de subida es de 100 segundos y el número de hilos es 50, significa que el servidor tardará 100 segundos en crear los 50 hilos, es decir, un nuevo hilo cada 2 segundos

JMETER: SAMPLERS



- Los Samplers (muestreadores) envían peticiones a un servidor. Ejemplos de samplers: HTTP request, FTP request, JDBC Request,... Se ejecutan en el orden en el que aparecen en el plan

P

HTTP Request

Podemos usar cualquier **NOMBRE** pero ten en cuenta que dicho nombre será el que se mostrará en el plan. Deberíamos elegir nombres representativos de las acciones asociadas a cada elemento

Name: HTTP Request

Comments: Esto es una petición HTTP

Basic Advanced

- Web Server

Protocol [http]: Server Name or IP: Port Number:

- HTTP Request

GET Path: Content encoding:

Redirect Automatically Follow Redirects Use KeepAlive Use multipart/form-data

Parameters Body Data Files Upload

Send Parameters With the Request:

Name:	Value	URL Encode?	Content-Type	Include Equals?
-------	-------	-------------	--------------	-----------------

Detail Add Add from Clipboard Delete Up Down

Sesión 9: Pruebas de aceptación 3

JMETER: CONTROLADORES LÓGICOS

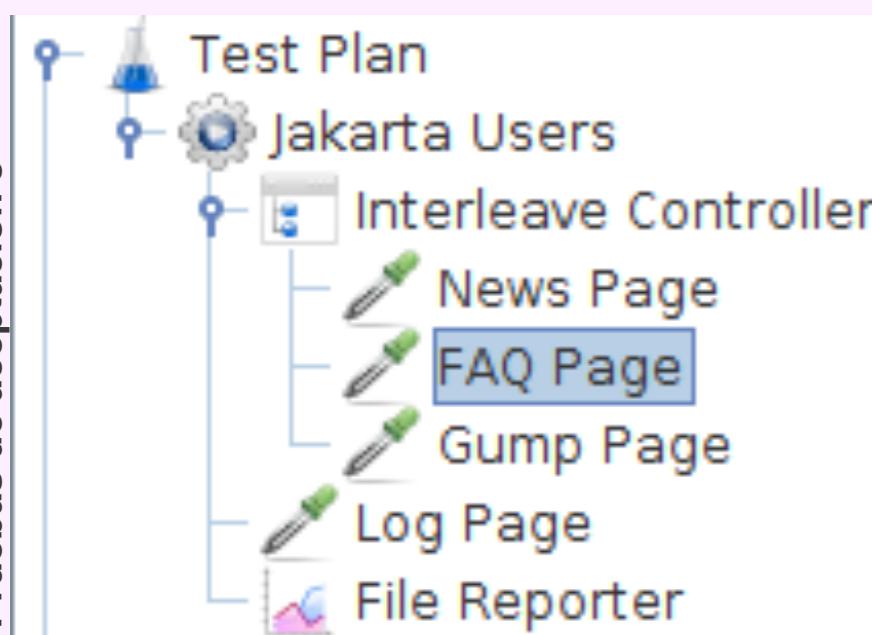


Logic Controller

http://jmeter.apache.org/usermanual/component_reference.html#Interleave_Controller

- Determinan la lógica que JMeter utiliza para decidir cuándo enviar las peticiones (orquestan el flujo de control). Actúan sobre sus elementos hijo
- Ejemplos de controladores
 - **Simple controller**: No tiene efecto sobre cómo procesa JMeter los elementos hijos de dicho controlador. Simplemente sirve para "agrupar" dichos elementos.
 - **Loop controller**: Itera sobre sus elementos hijos un cierto número de veces
 - **Only once controller**: Indica a JMeter que sus elementos hijos deben ser procesados UNA ÚNICA vez en el plan de pruebas
 - **Interleave controller**: ejecutará uno de sus subcontroladores o samplers en cada iteración del bucle de pruebas, alterándose secuencialmente a lo largo de la lista

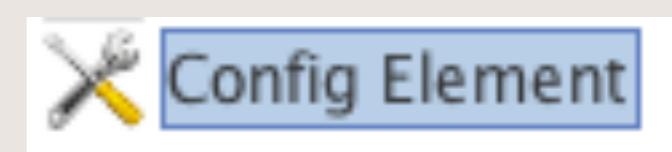
Ejemplo: supongamos que el grupo de hilos está formado por 2 hilos, y 5 iteraciones



Loop Iteration	Each JMeter Thread Sends These HTTP Requests
1	News Page
1	Log Page
2	FAQ Page
2	Log Page
3	Gump Page
3	Log Page
4	Because there are no more requests in the controller, JMeter starts over and sends the first HTTP Request, which is the News Page
4	Log Page
5	FAQ Page
5	Log Page

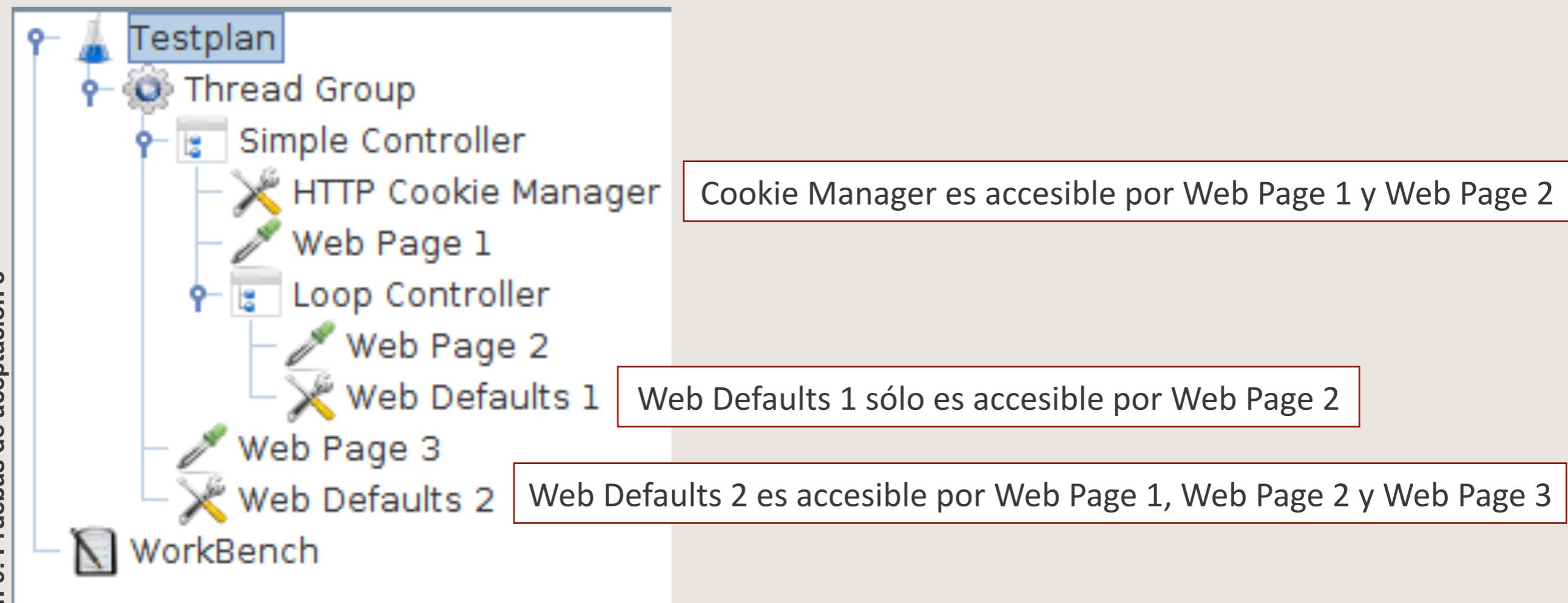
Cada hilo realiza 10 peticiones

JMETER: ELEMENTOS DE CONFIGURACIÓN



http://jmeter.apache.org/usermanual/test_plan.html

- "Trabajan" conjuntamente con un sampler. Si bien no realizan peticiones
- Un elemento de configuración es accesible sólo dentro de la rama del árbol (y sub-ramas) en la que se sitúa el elemento
- Un elemento de configuración dentro de una rama del árbol tiene mayor preferencia que el mismo elemento (mismo tipo de elemento) en una rama padre
- Ejemplos: HTTP Requests Defaults, HTTP Cookie Manager, HTTP Header Manager,...



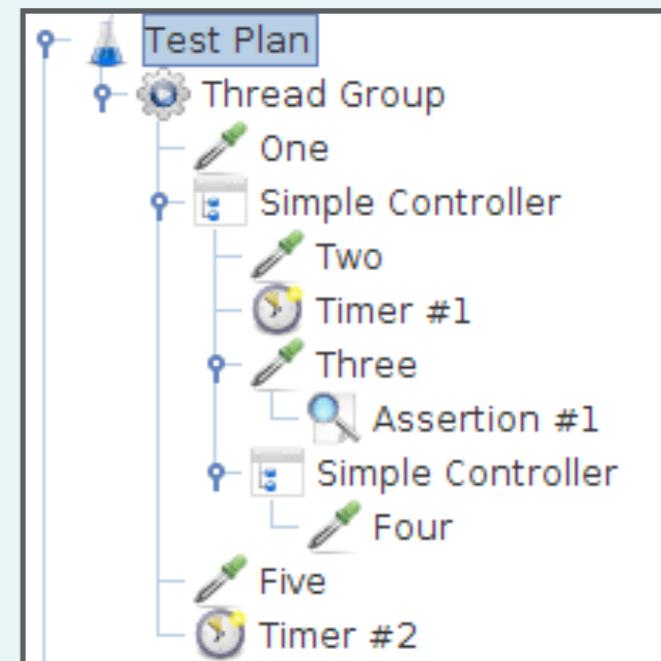
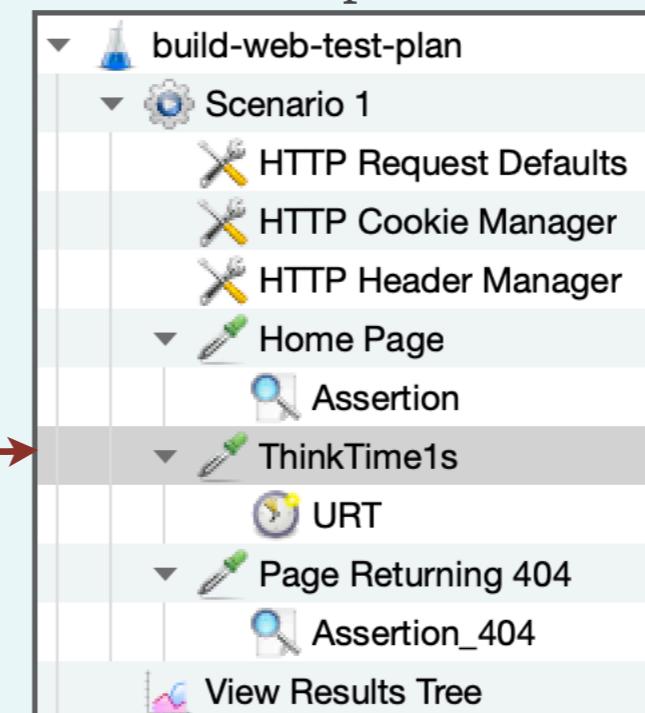
TIMERS

- Por defecto JMeter envía las peticiones sin realizar ninguna pausa entre las mismas. En este caso nuestro test podría "saturar" el servidor al enviar demasiadas peticiones en intervalos cortos de tiempo!!
- Los temporizadores permiten introducir pausas **antes** de realizar **cada** una de las peticiones de **cada** hilo
- Podemos utilizar varios tipos de temporizadores:
 - Constant timer**: Retrasa cada petición de usuario la misma cantidad de tiempo
 - Uniform random timer**: Introduce pausas aleatorias (con la misma probabilidad)
 - Gaussian random timer**: Introduce pausas según una determinada distribución (gausiana)
- Si hay varios timers en el mismo "ámbito" (nivel) del sampler se aplicarán todos ellos. En caso contrario, no se aplicará ninguno
- Si queremos que un timer afecte a un sólo sampler, debemos añadirlo como su hijo
- Si queremos que un timer se aplique después de un sampler:
 - Lo añadiremos al siguiente sampler, o
 - Lo añadiremos como hijo de un **Flow Control Action Sampler**

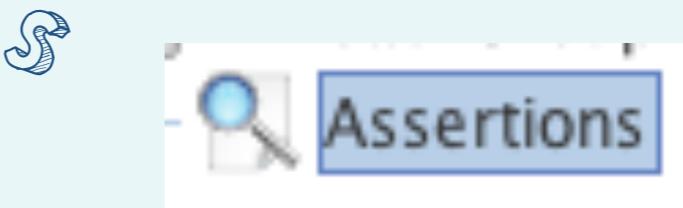


S

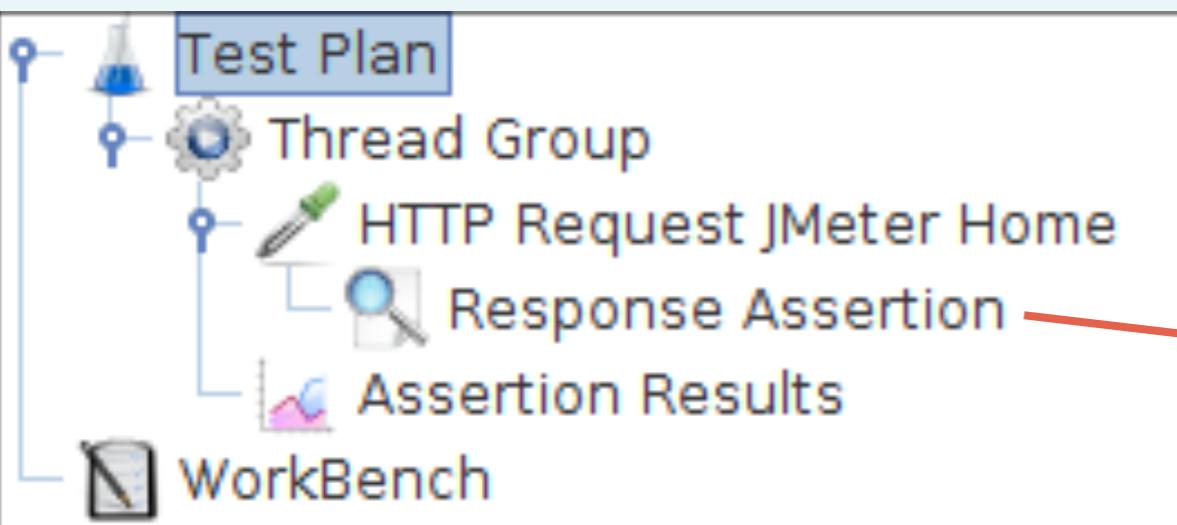
S



JMETER: ASERCIIONES



P



- Las aserciones permiten hacer afirmaciones sobre las respuestas recibidas del servidor que se está probando. Podemos añadir aserciones a cualquier sampler. Se trata de probar que la aplicación devuelve el resultado esperado

The screenshot shows the "Response Assertion" configuration dialog. The "Name" field is set to "Response Assertion". The "Comments" field is empty. Under "Apply to:", the radio button "Main sample only" is selected. Under "Field to Test", the radio button "Text Response" is selected. Under "Pattern Matching Rules", the radio button "Substring" is selected. A yellow callout box with a cartoon character and the text "Cualquier driver SIEMPRE debe incluir aserciones!!! Don't FORGET!" is overlaid on the bottom right of the dialog.

Name: Response Assertion

Comments:

Apply to:

Main sample and sub-samples Main sample only Sub-samples only JMeter Variable Name to use []

Field to Test

Text Response Response Code Response Message Response Headers
 Request Headers URL Sampled Document (text) Ignore Status
 Request Data

Pattern Matching Rules

Contains Matches Equals Substring Not Or

Patterns to Test

Add Add from Clipboard Delete

JMETER: LISTENERS (I)



http://jmeter.apache.org/usermanual/component_reference.html#listeners

- Los **listeners** se utilizan para ver y/o almacenar en el disco los resultados de las peticiones realizadas. Proporcionan acceso a la información que JMeter va acumulando sobre los casos de prueba a medida que se ejecutan los tests
 - TODOS los listeners guardan los MISMOS datos; la única diferencia es la forma en que presentan dichos datos en la pantalla
 - Ejemplos de listeners: Simple Data Writer, Aggregate Graph, Graph results, Aggregate Report, Summary Report, Response Time Graph, Assertion results,...

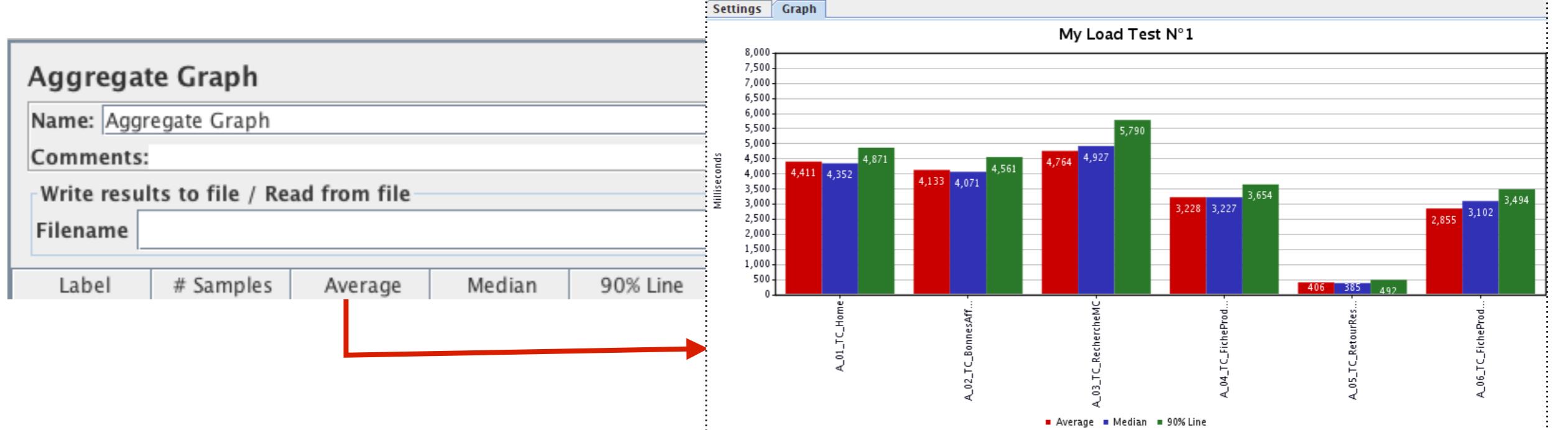
Simple Data Writer

Name: Simple Data Writer

Comments:

Write results to file / Read from file

Filename Browse... Log/Display Only: Errors Successes



JMETER: LISTENERS (II)



ver <http://www.guru99.com/jmeter-performance-testing.html>

Aggregate Report

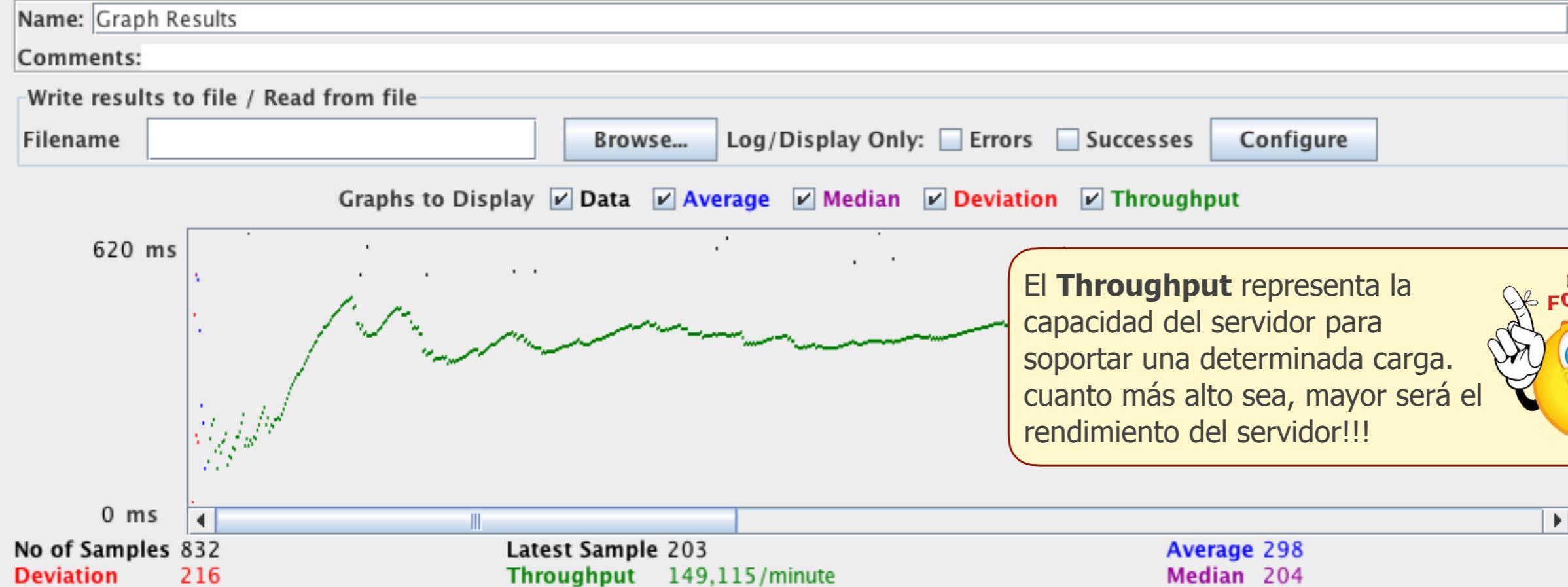
Rendimiento = num_peticiones/segundo

Rendimiento = Throughput

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	Received KB/sec	Sent KB/sec
HTTP Request-3	80	222	224	317	104	353	0.00%	1.5/sec	2.62	0.45
HTTP Request-1	80	231	233	336	108	354	0.00%	1.5/sec	10.49	0.13
HTTP Request-2	80	233	233	329	102	354	1.25%	1.5/sec	5.23	0.28
TOTAL	240	229	229	330	102	354	0.42%	4.4/sec	18.18	0.86

Include group data Save Table Header

Graph Results



SOBRE LOS DATOS REGISTRADOS POR JMETER

El tiempo se calcula en milisegundos

- Para cada muestra (sampler), JMeter calcula:

- **# Samples** - Número de muestras con la misma etiqueta
- **Average** - Tiempo medio de respuesta (en milisegundos)
- **Median** - The median is the time in the middle of a set of results. 50% of the samples took no more than this time; the remainder took at least as long.
- **Línea de 90%** (percentil): 90% of the samples took no more than this time. The remaining samples took at least as long as this
- **Min** - Tiempo mínimo de respuesta para las muestras con la misma etiqueta
- **Max** - Tiempo máximo de respuesta para las muestras con la misma etiqueta
- **% Error** - Porcentaje de peticiones con errores
- Rendimiento (**Throughput**) - número de peticiones por segundo/minuto/hora. La unidad de tiempo se elige en función de que el valor visualizado sea como mínimo 1.
- **Kb/sec** - rendimiento expresado en Kilobytes por segundo

CONSEJOS JMETER



P

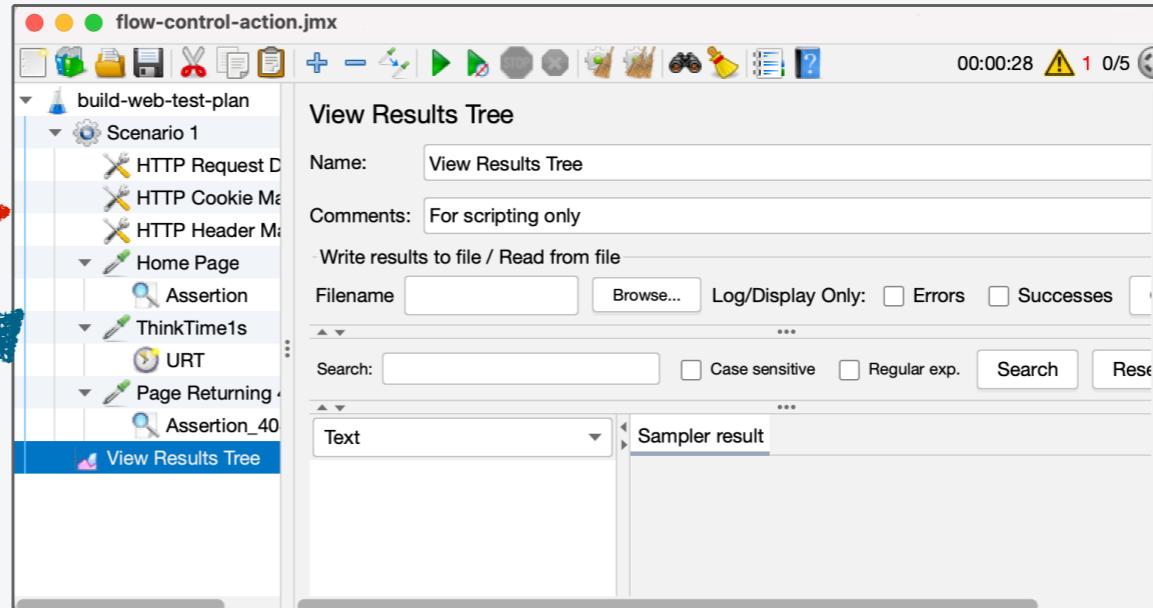
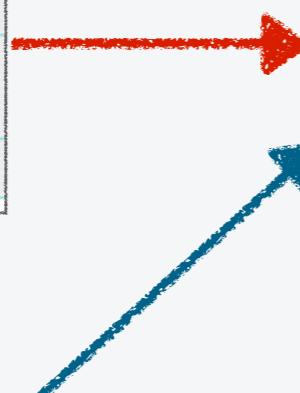
- Utiliza escenarios de prueba significativos, y construye planes de prueba que prueben situaciones representativas del mundo real
 - Los casos de uso ofrecen un punto de partida ideal. A partir de ellos debes generar un perfil operacional
- Asegúrate de ejecutar JMeter en una máquina distinta a la del sistema a probar.
 - Esto previene que JMeter afecte sobre los resultados de las pruebas
- El proceso de pruebas es un proceso científico. Todas las pruebas se deben realizar bajo condiciones completamente controladas
 - Si estas trabajando con un servidor compartido, primero comprueba que nadie más esta realizando pruebas de carga contra la misma aplicación web.
- Asegúrate de que dispones de ancho de banda en la estación que ejecuta JMeter
 - La idea es probar el rendimiento de la aplicación y el servidor, y no la conexión de la red.
- Utiliza diferentes instancias de JMeter ejecutándose en diferentes máquinas para añadir carga adicional al servidor
 - Esta configuración suele ser necesaria para realizar pruebas de stress. JMeter puede controlar las instancias JMeter de las otras máquinas y coordinar la prueba
- Deja una prueba JMeter ejecutarse durante largos periodos de tiempo, posiblemente varios días o semanas
 - Estarás probando la disponibilidad del sistema y resaltando las posibles degradaciones en el rendimiento del servidor debido a una mala gestión de los recursos

Y AHORA VAMOS AL LABORATORIO...

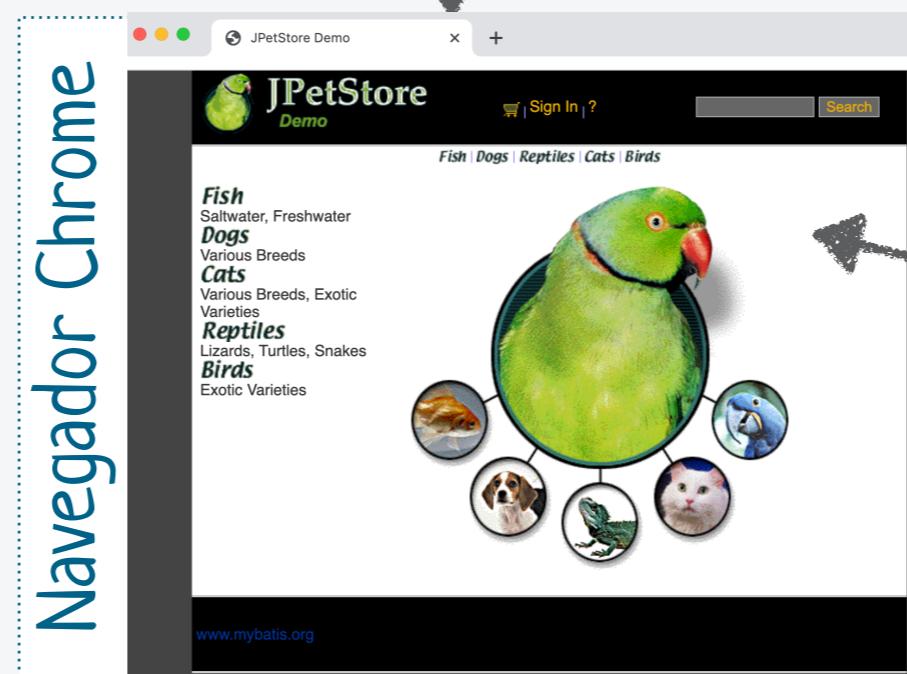
Vamos a implementar tests de aceptación (para validar propiedades emergentes NO funcionales) sobre una aplicación web con JMeter



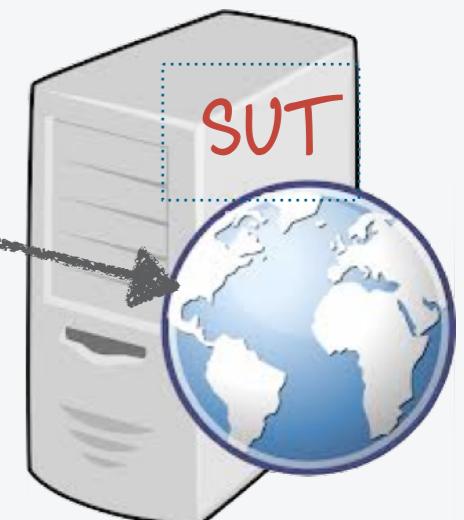
Camino	Datos Entrada	Resultado Esperado
C1	d1=... d2=...	r1
..		
CM	d1=... d2=... ...	rM



usaremos JMeter (el driver estará implementado en un fichero .jmx)



Servidor web

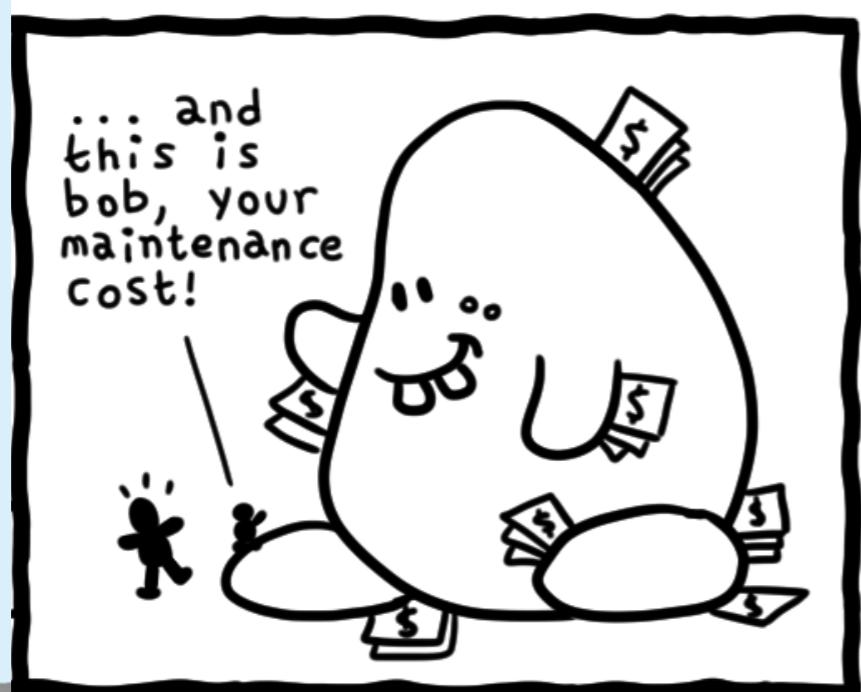
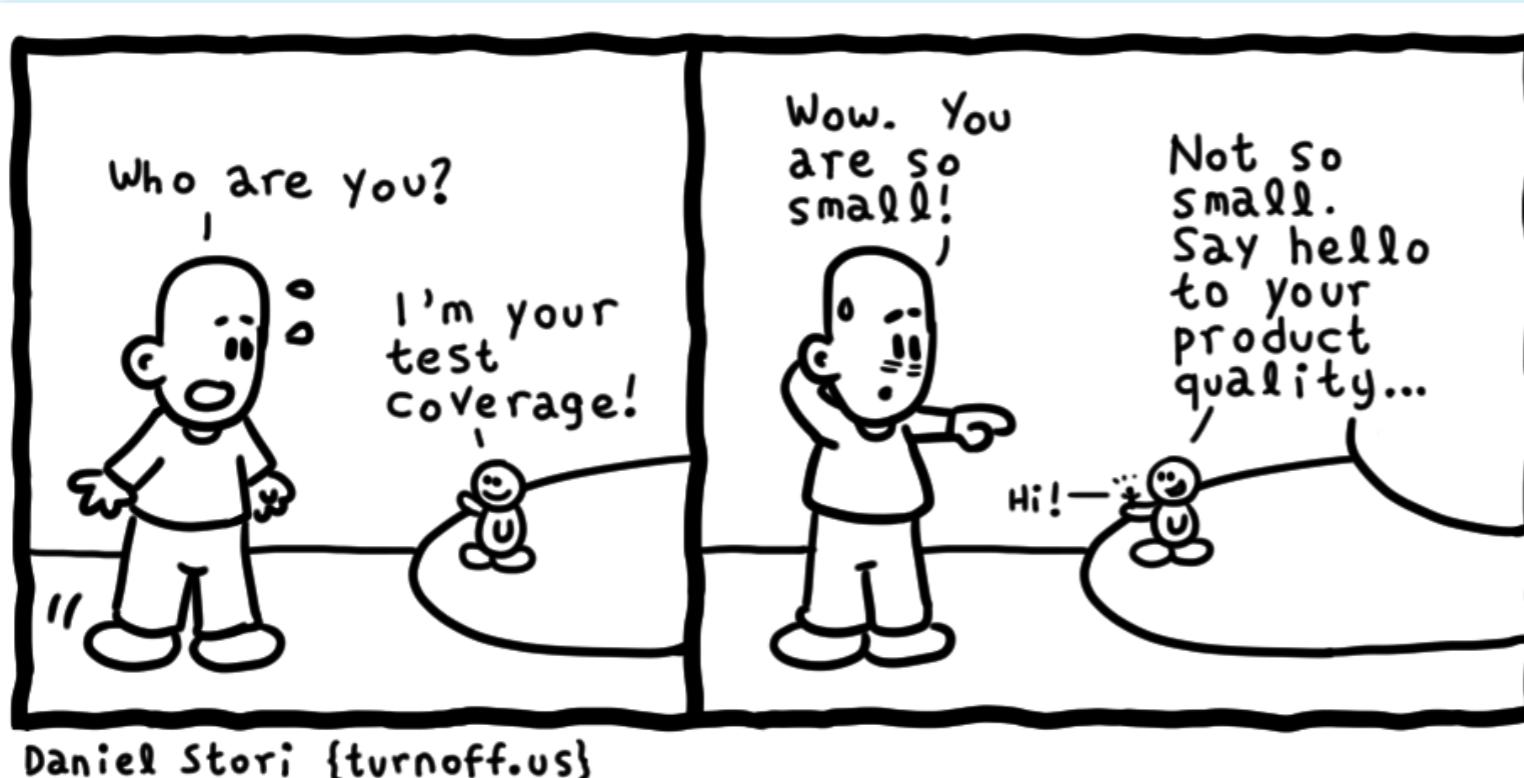


REFERENCIAS BIBLIOGRÁFICAS

- Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
 - Capítulos 8 y 15
- Página oficial JMeter:
 - <http://jmeter.apache.org/usermanual/index.html>
- Otras referencias interesantes:
 - <http://www.guru99.com/jmeter-performance-testing.html>
 - <http://www.wikishown.com/apache-jmeter-understanding-summary-report/>
 - <http://jmeterperftest.blogspot.com.es>
 - https://www.blazemeter.com/blog/understanding-your-reports-part-3-key-statistics-performance-testers-need-understand?utm_source=Blog&utm_medium=BM_Blog&utm_campaign=kpis-part1

PPSS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2020-21



Sesión S10: Análisis de pruebas

Análisis de pruebas

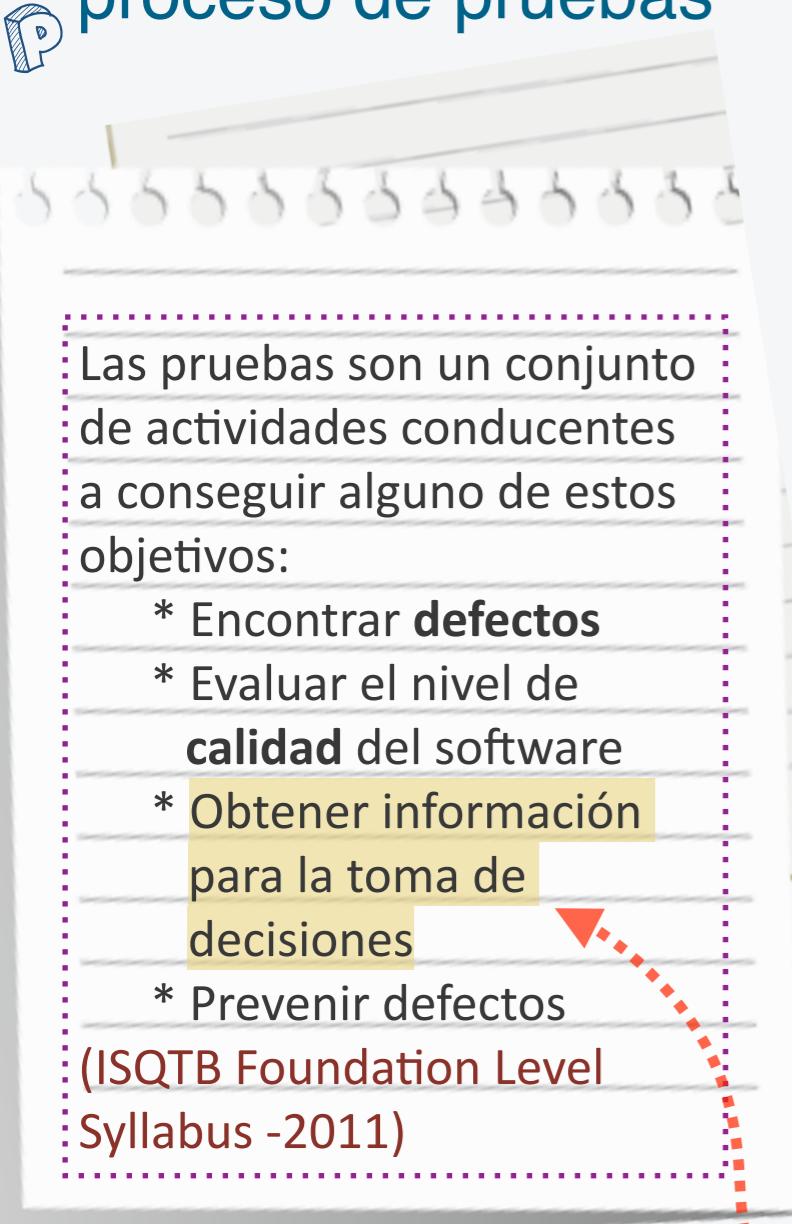
- Uso de métricas para el análisis de pruebas
- Cobertura de código
 - Formas de cuantificar la cobertura de código
 - Niveles de cobertura
- Herramienta automática de análisis: JaCoCo
 - Integración con Maven
 - Generación de informes

Vamos al laboratorio...

EL PROCESO DE PRUEBAS

P

Definición del proceso de pruebas



- ¿Qué información podemos obtener sobre las pruebas realizadas?
- ¿Para qué es útil dicha información?

Actividades del proceso de pruebas

- Planificación y control de las pruebas
- Diseño de las pruebas
- Implementación y ejecución de las pruebas
- Evaluar si hemos alcanzado las metas definidas y emitir un informe



SEGÚN ISQTB FOUNDATION LEVEL SYLLABUS (2011)

Definimos los objetivos de las pruebas (en cada nivel de pruebas tenemos objetivos diferentes). Establecemos QUÉ, CÓMO y CUÁNDΟ vamos a realizarlas

Es el proceso más importante para cumplir con los objetivos marcados. Básicamente consiste en decidir, de forma sistemática, con qué datos de entrada concretos vamos a probar el código. En cada nivel de pruebas el proceso de diseño es diferente.

La idea es poder ejecutar las pruebas diseñadas de forma automática. En cada nivel de pruebas usaremos "herramientas" diferentes.

Como resultado de la ejecución de las pruebas obtendremos información sobre el proceso realizado. En cada nivel de pruebas obtendremos informes diferentes

MÉTRICAS Y ANÁLISIS DE PRUEBAS

P

P

- Para analizar y extraer conclusiones sobre las pruebas realizadas sobre nuestro proyecto software, necesitamos “cuantificar” el proceso y resultado de las mismas, es decir, utilizar métricas que nos permitan conocer con la mayor objetividad diferentes características de nuestras pruebas.
- Una **métrica** se define como una medida cuantitativa del grado en el que un **sistema, componente o proceso**, posee un determinado atributo
 - Si no podemos medir, no podemos saber si estamos alcanzando nuestros objetivos, y lo más importante, no podremos “controlar” el proceso software ni podremos mejorarlo
 - Tiene que haber una relación entre lo que podemos medir, y lo que queremos “conocer”.
- ¿Qué podemos medir? Casi cualquier cosa: líneas de código probadas, número de errores encontrados, número de pruebas realizadas, número de clases probadas, número de horas invertidas en realizar las pruebas,...

ANÁLISIS DE LAS PRUEBAS

P

S

- independientemente del objetivo concreto de nuestras pruebas, siempre buscamos **efectividad** (no ser efectivo implica no cumplir nuestro objetivo)
- Básicamente, hay dos causas fundamentales que pueden provocar que nuestras pruebas no sean efectivas
 - Podemos ejecutar el código, pero con un mal diseño de pruebas, de forma que los casos de prueba sean tales que nuestro código esté “pobremente” probado o no probado de ninguna manera
 - Podemos, de forma “deliberada”, dejar de probar partes de nuestro código
- La primera cuestión es bastante complicada de detectar de forma automática
- Por otro lado, está claro que si parte de nuestro código no se ejecuta durante las pruebas, es que no está siendo probado
 - Vamos a detenernos en esta cuestión: en el problema de la COBERTURA de código, es decir en analizar cual es la EXTENSIÓN de nuestras pruebas
- La COBERTURA de código es la característica que hace referencia a cuánto código estamos probando con nuestros tests (porcentaje de código probado)
 - IMPORTANTE: NO proporciona un indicador la calidad del código, ni la calidad de nuestras pruebas, sino de la **extensión** de nuestros tests

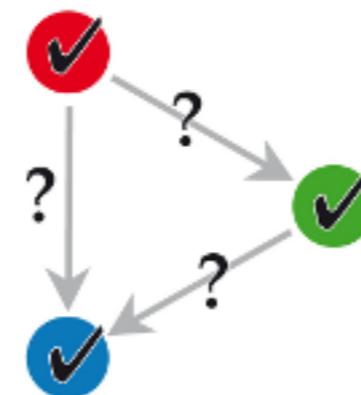
COBERTURA DE CÓDIGO

P

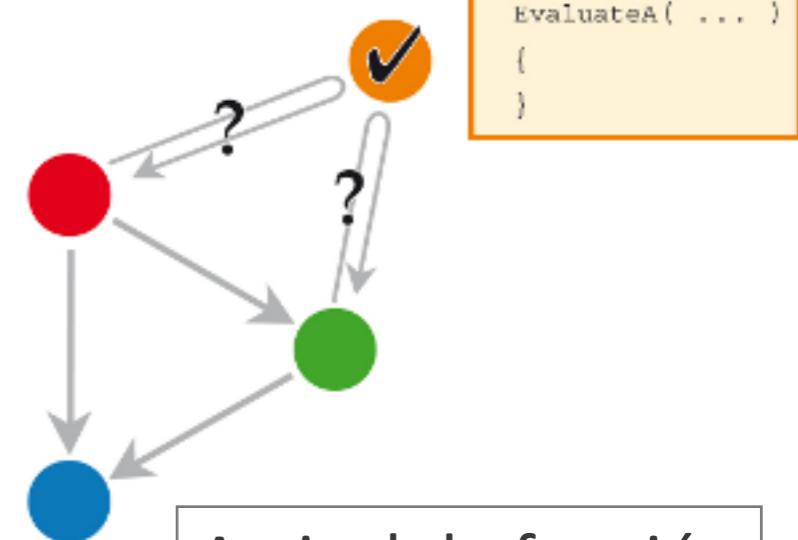
P

- El análisis de la cobertura de código se lleva a cabo mediante la “exploración”, de forma dinámica, de diferentes tipos de flujos control del programa

```
a = calcular(valor);  
if ((a || b) && c) {  
    contador = 0;  
    a = calcular(b);  
}  
  
contador++;
```



A nivel de bloque



A nivel de función

```
a = calcular(valor);  
if ((a || b) && c) {  
    contador = 0;  
    a = calcular(b);  
}  
  
contador++;
```

A nivel de decisiones

```
a = calcular(valor);  
if ((a || b) && c) {  
    contador = 0;  
    a = calcular(b);  
}  
  
contador++;
```

A nivel de condiciones

... entre otras

P ANÁLISIS DE LA COBERTURA DE CÓDIGO (I)

Pragmatic Software Testing. Rex Black. Chapter 21

- Hay **siete** formas principales para cuantificar la cobertura de código:
 - **Statement coverage**: un 100% significa que hemos ejecutado cada sentencia (línea)
 - **Branch (or decision) coverage**: un 100% significa que hemos ejecutado cada rama o DECISIÓN en sus vertientes verdadera y falsa.
 - Una **decisión** es una expresión booleana formada por condiciones y cero o más operadores booleanos
 - Para sentencias if, necesitamos asegurar que la expresión que controla las “ramas” se evalúa a cierto y a falso
 - Para sentencias “switch” necesitamos cubrir cada caso especificado, así como al menos un caso no especificado (o el caso por defecto)
 - Un 100% de cobertura de ramas implica un 100% de cobertura de líneas
 - **Condition coverage**: un 100% significa que hemos ejercitado cada CONDICIÓN. Cuando tenemos expresiones con múltiples condiciones, para conseguir el 100% de cobertura de las condiciones tenemos que evaluar el comportamiento para cada una de dichas condiciones en sus vertientes verdadera y falsa.
 - Una **condición** es el elemento “mínimo” de una expresión booleana (no puede descomponerse en una expresión booleana más simple)
 - Por ejemplo, para la sentencia if ((A>0) & (B>0)), necesitamos probar que (A>0) sea cierto y falso, y (B>0) sea cierto y falso. Esto lo podemos conseguir con dos tests: true & true, y false & false

PS ANÁLISIS DE LA COBERTURA DE CÓDIGO (II)

- Multicondition coverage: un 100% de cobertura significa que hemos ejercitado todas las posibles combinaciones de condiciones.
 - ❖ Siguiendo con el ejemplo anterior, para la sentencia if ((A>0) & (B>0)), necesitamos probar las cuatro posibles combinaciones: true & true, true & false, false & true y false & false
- Condition decision coverage: en lenguajes como C++ y Java, en donde las condiciones “siguientes” se evalúan dependiendo de las condiciones que las preceden, un 100% de cobertura de condiciones múltiples puede no tener sentido
 - ❖ Por ello ejercitaremos cada condición en el programa (cada una toma todos sus posibles valores al menos una vez), y cada decisión en el programa (cada una toma todos sus valores posibles al menos una vez)
 - ❖ Para la sentencia java if ((A>0) && (B>0)), probaremos con las combinaciones true && true, true && false y false && true. La combinación false && false no es necesaria debido a que la segunda condición no puede influenciar la decisión tomada por la expresión de la sentencia if
- Loop coverage: un 100% de cobertura implica probar el bucle con 0 iteraciones, una iteración y múltiples iteraciones
- Path coverage: un 100% de cobertura implica que se han probado todos los posibles caminos de control. Es muy difícil de conseguir cuando hay bucles. Un 100% de cobertura de caminos implica un 100% de cobertura de ramas

NIVELES DE COBERTURA DE CÓDIGO

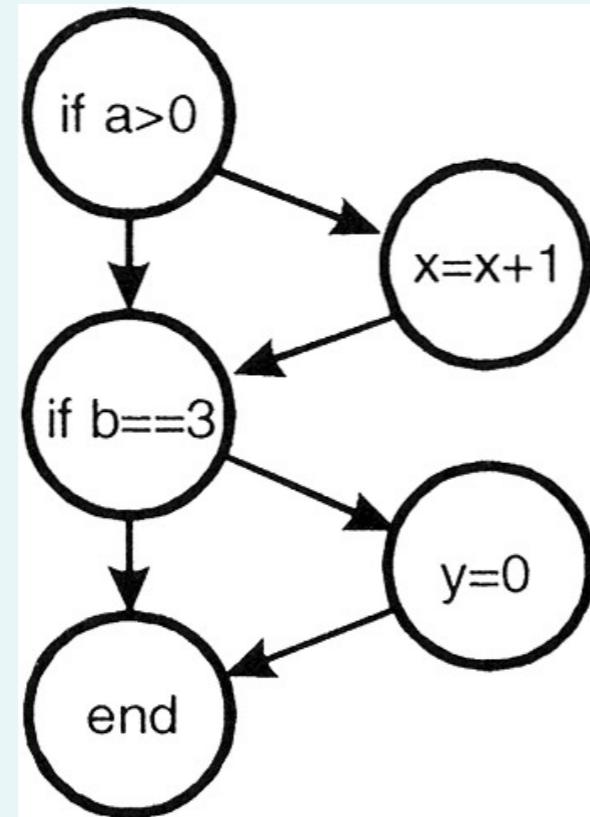
statement coverage



[Statement Coverage - Georgia Tech - Software Development Process - YouTube](#)

○ NIVEL 1: cobertura de líneas (100%)

```
if (a>0) {  
    x=x+1;  
}  
if (b==3) {  
    y=0;  
}
```



- Para este código podemos conseguir un 100% de cobertura de líneas con un único caso de prueba (por ejemplo a=6, y b=3). Sin embargo estamos dejando de probar 3 de los cuatro caminos posibles.
- Un 100% de cobertura de líneas no suele ser un nivel aceptable de pruebas. Podríamos clasificarlo como de NIVEL 1
- A pesar de constituir el nivel más bajo de cobertura, en la práctica puede ser difícil de conseguir

Realmente hay un **NIVEL 0**, el cual se define como: “**test whatever you test; let the users test the rest**” [Lee Copeland, 2004]

Boris Beizer, en una ocasión escribió: “**testing less than this [100% statement coverage] for new software is unconscionable and should be criminalized. ... In case I haven't made myself clear, ... untested code in a system is stupid, shortsighted, and irresponsible.**” [Beizer, 1990]

NIVELES DE COBERTURA DE CÓDIGO

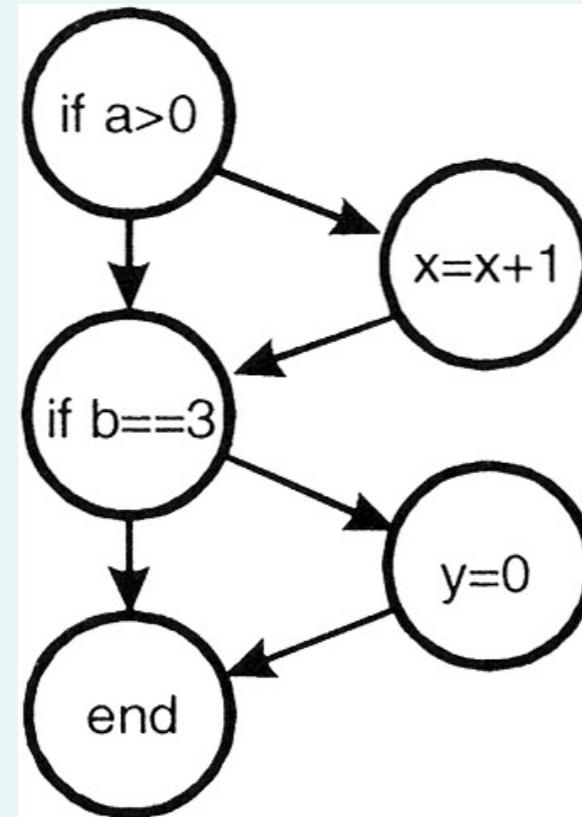
branch (decision) coverage



Branch Coverage - Georgia Tech - Software Development Process - Youtube

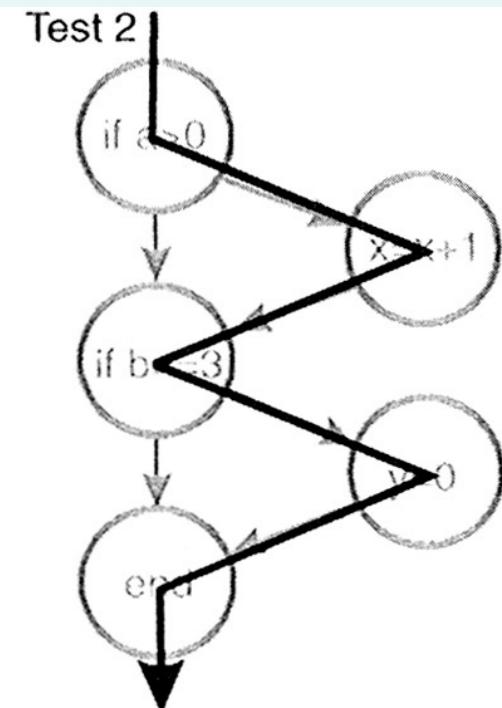
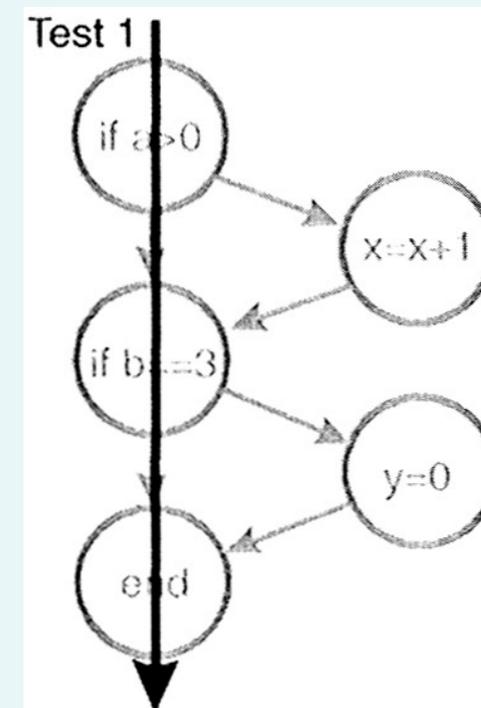
○ NIVEL 2: cobertura de ramas (100%)

```
if (a>0) {  
    x=x+1;  
}  
if (b==3) {  
    y=0;  
}
```



Un 100% de cobertura de ramas (decisiones) implica un 100% de cobertura de líneas

- Para este código podemos conseguir un 100% de cobertura de ramas con dos casos de prueba (por ejemplo a=0, b=2; y a=4, b=3). Sin embargo estamos dejando de probar 2 de los cuatro caminos posibles.



- Observa que para conseguir la cobertura de ramas (el 100%), necesitamos diseñar casos de prueba de forma que cada DECISIÓN que tenga como resultado true/false sea evaluada al menos una vez

NIVELES DE COBERTURA DE CÓDIGO

condition coverage



Condition Coverage - Georgia Tech - Software Development Process - YouTube

ONIVEL 3: cobertura de condiciones (100%)

```
if (a>0 & c==1) {  
    x=x+1;  
}  
if (b==3 | d<0) {  
    y=0;  
}
```

- Para que la primera sentencia sea cierta, a tiene que ser >0 y $c = 1$. La segunda requiere que $b= 3$ ó $d<0$
- En la primera sentencia, si el valor de a lo fijamos a 0 para hacer las pruebas, probablemente la segunda condición ($c==1$) no será probada (dependerá del lenguaje de programación)

- Podemos conseguir el nivel 3 con dos casos de prueba:

- ($a=7, c=1, b=3, d = -3$)
- ($a=-4, c=2, b=5, d = 6$)

Un 100% de cobertura de condiciones NO garantiza un 100% de cobertura de líneas

- La cobertura de condiciones es mejor que la cobertura de decisiones debido a que cada CONDICIÓN INDIVIDUAL es probada (en sus vertientes verdadera y falsa) al menos una vez, mientras que la cobertura de decisiones puede conseguirse sin probar cada condición

NIVELES DE COBERTURA DE CÓDIGO

condition + decision coverage



Branch and Condition Coverage - Georgia Tech - Software Development Process - youtube

ONIVEL 4: cobertura de condiciones (100%) y decisiones (100%)

```
if(x & y) {  
    sentenciaCond;  
}
```

- En este ejemplo podemos conseguir el nivel 3 (cobertura de condiciones), con dos casos: (x =TRUE, y = FALSE) y (x=FALSE, y =TRUE). Pero observa que en este caso “sentenciaCond” nunca será ejecutada. Podemos ser más completos si buscamos también una cobertura de decisiones

□ Podemos conseguir el nivel 4 creando casos de prueba para cada condición y cada decisión:

- (x=TRUE, y=FALSE),
- (x=FALSE, y=TRUE),
- (x=TRUE, y=TRUE)

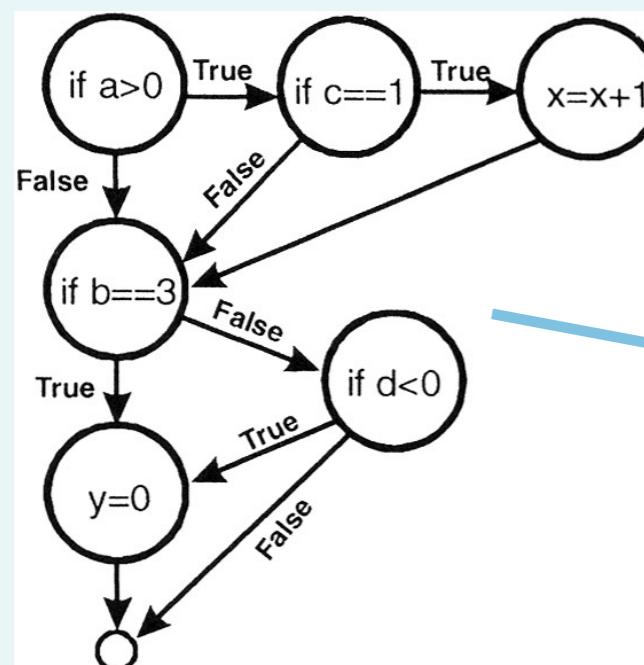
NIVELES DE COBERTURA DE CÓDIGO

multicondition coverage

ONIVEL 5: cobertura de condiciones múltiples (100%)

```
if (a>0 & c==1) {  
    x=x+1;  
}  
if (b==3 | d<0) {  
    y=0;  
}
```

operador && de java



Un 100% de cobertura de condiciones múltiples implica un 100% de cobertura de condiciones, decisiones, y condiciones/decisiones

- En este ejemplo con código Java (sin cortocircuito), este nivel de cobertura podemos conseguirlo con cuatro casos de prueba:
 - a= 5, c=1, b=3, d= -3
 - a=-4, c=1, b=3, d= 7
 - a= 5, c=2, b=5, d= -4
 - a=-1, c=2, b=5, d=0
- Si el lenguaje evalúa las condiciones en "cortocircuito" (&&, ||) podemos conseguir un 100% de cobertura de condiciones múltiples, considerando cómo el compilador evalúa realmente las condiciones múltiples de una decisión.
 - a= 5, c=1, b=5, d=0
 - a= 5, c=2, b=5, d= -4
 - a= -1, c=2, b=3, d= -3
- Si conseguimos un 100% de cobertura de condiciones múltiples, también conseguimos cobertura de decisiones, condiciones y condiciones+decisiones
- Una cobertura de condiciones múltiples no garantiza una cobertura de caminos

NIVELES DE COBERTURA DE CÓDIGO

loop coverage

P

P

ONIVEL 6: cobertura de bucles (100%)

- Cuando un módulo tiene bucles, de forma que el número de caminos hacen impracticable las pruebas, puede conseguirse una reducción significativa de esfuerzo limitando la ejecución de los bucles a un pequeño número de casos
 - ✿ Ejecutar el bucle 0 veces
 - ✿ Ejecutar el bucle 1 vez
 - ✿ Ejecutar el bucle n veces, en donde n es un número que representa un valor típico
 - ✿ Ejecutar el bucle en su máximo número de veces m (adicionalmente se pueden considerar la ejecución (m-1) y (m+1))



[Test Criteria Subsumption - Georgia Tech - Software Development Process](#) - YouTube



[Other Criteria - Georgia Tech - Software Development Process](#) - YouTube

NIVELES DE COBERTURA DE CÓDIGO

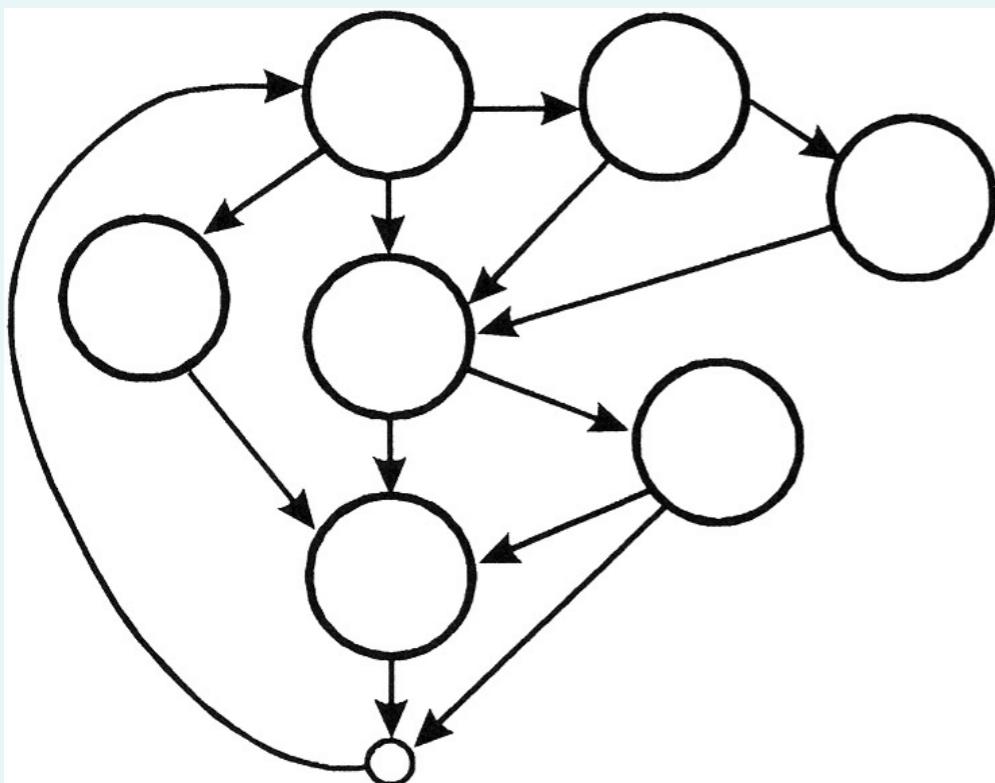
path coverage



Path Coverage - Software Testing - Youtube

ONIVEL 7: cobertura de caminos (100%)

- Si conseguimos una cobertura del 100% de caminos garantizamos que recorremos todas las posibles combinaciones de caminos de control
- Para códigos de módulos sin bucles, generalmente el número de caminos es lo suficientemente “pequeño” como para que pueda construirse un caso de prueba para cada camino. Para módulos con bucles, el número de caminos puede ser “enorme”, de forma que el problema se haga intratable



Un 100% de cobertura de caminos implica un 100% de cobertura de bucles, ramas y sentencias

Un 100% de cobertura de caminos NO garantiza una cobertura de condiciones múltiples (ni al contrario)

P ANALIZADORES AUTOMÁTICOS DE COBERTURA S

- P
 - Los analizadores de código que utilizan las herramientas de cobertura se basan en la instrumentación de dicho código.
 - Instrumentar el código consiste en añadir código adicional para poder obtener una métrica de cobertura (el código adicional añade algún contador de código que devuelve un resultado después de la ejecución del mismo)
 - Si hablamos de Java, podemos encontrar tres categorías:
 - Inserción de código de instrumentación en el código fuente
 - Inserción de código de instrumentación en el byte-code de Java
 - * Esta aproximación es la que utiliza JaCoCo
 - Ejecución de código en una máquina virtual de Java modificada
 - Vamos a ver como ejemplo una herramienta automática que analiza la cobertura de código:
 - JaCoCo

MÉTRICAS QUE CALCULA JACOCO

<https://www.jacoco.org/jacoco/trunk/doc/counters.html>

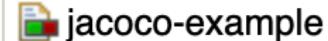


- JaCoCo instrumenta el bytecode de Java (de forma off-line, y también on-the-fly) de forma que cuando se ejecutan los tests sobre dicho código recopila información sobre la cobertura conseguida
 - Puede utilizarse desde línea de comandos, integrada con Ant, o también con Maven
- JaCoCo genera un informe con las siguiente métricas:
 - Instructions
 - ✿ Número de instrucciones byte code de Java
 - Branches (para cada sentencia if, switch)
 - ✿ ¿Se ejecuta cada decisión y condición en sus vertientes verdadera y falsa? CUIDADO!! Jacoco llama branch tanto a una condición como a una decisión. P.ej. Si en un if hay 2 decisiones y 4 condiciones posibles, entonces ese if tiene 6 branches.
 - ✿ Las sentencias try...catch no se consideran como condiciones
 - Complejidad ciclomática (para cada método no abstracto)
 - ✿ ¿Cuántos casos de prueba son necesarios para cubrir todos los caminos linealmente independientes?
 - Líneas
 - ✿ Una línea se considera ejecutada cuando se ejecuta al menos una de las instrucciones bytecode asociadas a esa línea
 - Métodos
 - ✿ Cada método no abstracto contiene una instrucción como mínimo. Un método se considera ejecutado, cuando se ejecuta al menos una de sus instrucciones. Los constructores e inicializadores estáticos se cuentan como métodos
 - Clases
 - ✿ Una clase se considera ejecutada, cuando se ejecuta al menos uno de sus métodos.

P INFORMES QUE GENERA JACOCO

Para cada métrica se muestra la columna "Missed"

Niveles de análisis: proyecto, paquete, clase y método

 jacoco-example

Paquete

jacoco-example

indicación del número de tests que faltan para una cobertura del 100% de condiciones + líneas

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cqty	Missed	Lines	Missed	Methods	Missed	Classes
ppss	50%		25%		2	4	4	8	0	2	0	1
Total	11 of 22	50%	3 of 4	25%	2	4	4	8	0	2	0	1

Análisis a nivel de proyecto

 jacoco-example > ppss

ppss

Podemos "navegar" por los hiperenlaces

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cqty	Missed	Lines	Missed	Methods	Missed	Classes
App	50%		25%		2	4	4	8	0	2	0	1
Total	11 of 22	50%	3 of 4	25%	2	4	4	8	0	2	0	1

Sesión 10: Análisis de pruebas

 jacoco-example > ppss > App

App

Método

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cqty	Missed	Lines	Missed	Methods
two_ifs(int)	42%		25%		2	3	4	7	0	1
App()	100%		n/a		0	1	0	1	0	1
Total	11 of 22	50%	3 of 4	25%	2	4	4	8	0	2

P INFORMES QUE GENERA JACOCO

Podemos visualizar el código fuente

Constructor por defecto

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxt	Missed	Lines	Missed	Methods
two_ifs(int)	42%		25%		2	3	4	7	0	1
App()		100%		n/a	0	1	0	1	0	1
Total	11 of 22	50%	3 of 4	25%	2	4	4	8	0	2

Se resaltan en VERDE las sentencias ejecutadas

No se han ejecutado todas las "branches" (falta 1)

No se han ejecutado ninguna de las dos "branches"

Se muestran en ROJO las sentencias no ejecutadas

Resultado de ejecutar two_ifs(11)

```
jacoco-example > ppss > App.java
App.java
1. package ppss;
2.
3.
4. public class App {
5.
6.     public int two_ifs(int i) {
7.         if (i<20) {
8.             System.out.println("Soy un numero menor que 20");
9.             return 1;
10.        }
11.        if (i % 2 == 0) {
12.            System.out.println("Soy numero par");
13.            return 2;
14.        }
15.        return 0;
16.    }
17. }
```

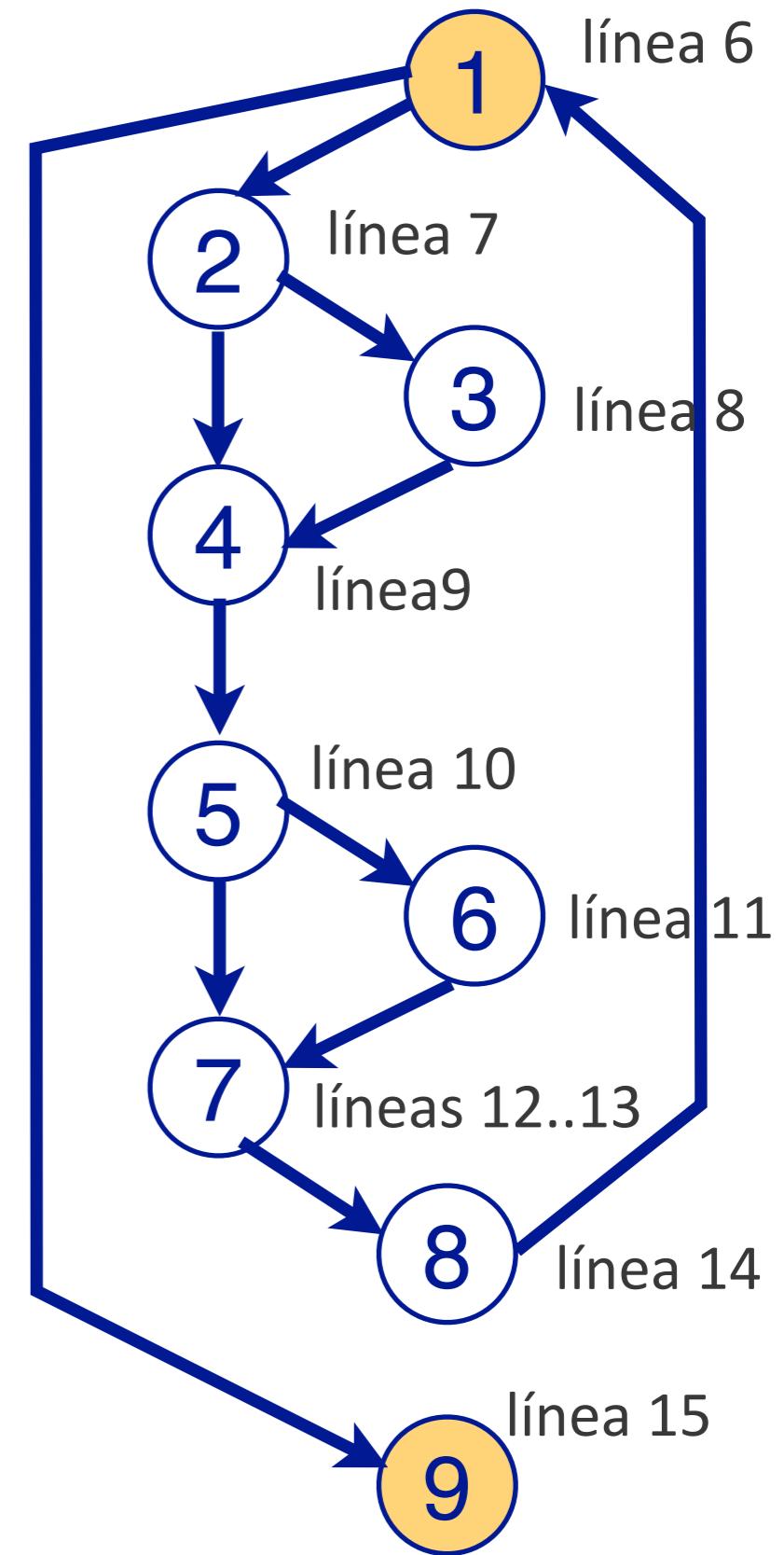
GRAFO DE FLUJO DE UN PROGRAMA

TODAS las sentencias deben estar representadas en el grafo

Cada NODO representa como máximo una condición y/o cualquier número de sentencias secuenciales

```
J Example.java X
1 public class Example {
2     boolean var= false;
3
4     public void nothing(int number) {
5
6         while (var == false) {
7             if (number %2 == 0)
8                 System.out.println("Even number");
9
10            if (number >20)
11                System.out.println("Greater than 20");
12
13            var = true;
14        }
15    }
16 }
```

Cada ARISTA representa el flujo de control de las sentencias



P GRAFO DE FLUJO Y CÁLCULO DE CC

Hay varias formas de calcular la CC

P

P

- La complejidad ciclomática (CC) es una cota superior del número de caminos linealmente independientes en el grafo
- Hemos visto que se puede calcular de varias formas:
 - (1) $CC = \text{arcos} - \text{nodos} + 2 = 11 - 9 + 2 = 4$
 - (2) $CC = \text{número de condiciones} + 1 = 3 + 1 = 4$
 - (3) $CC = \text{número de regiones cerradas en el grafo, incluyendo la externa} = 4$

Las alternativas (2) y (3) SÓLO se pueden utilizar si el programa NO tiene saltos incondicionales

- **JaCOCO** calcula la CC utilizando una cuarta alternativa:

□ $CC = B - D + 1 = 6 - 3 + 1 = 4$

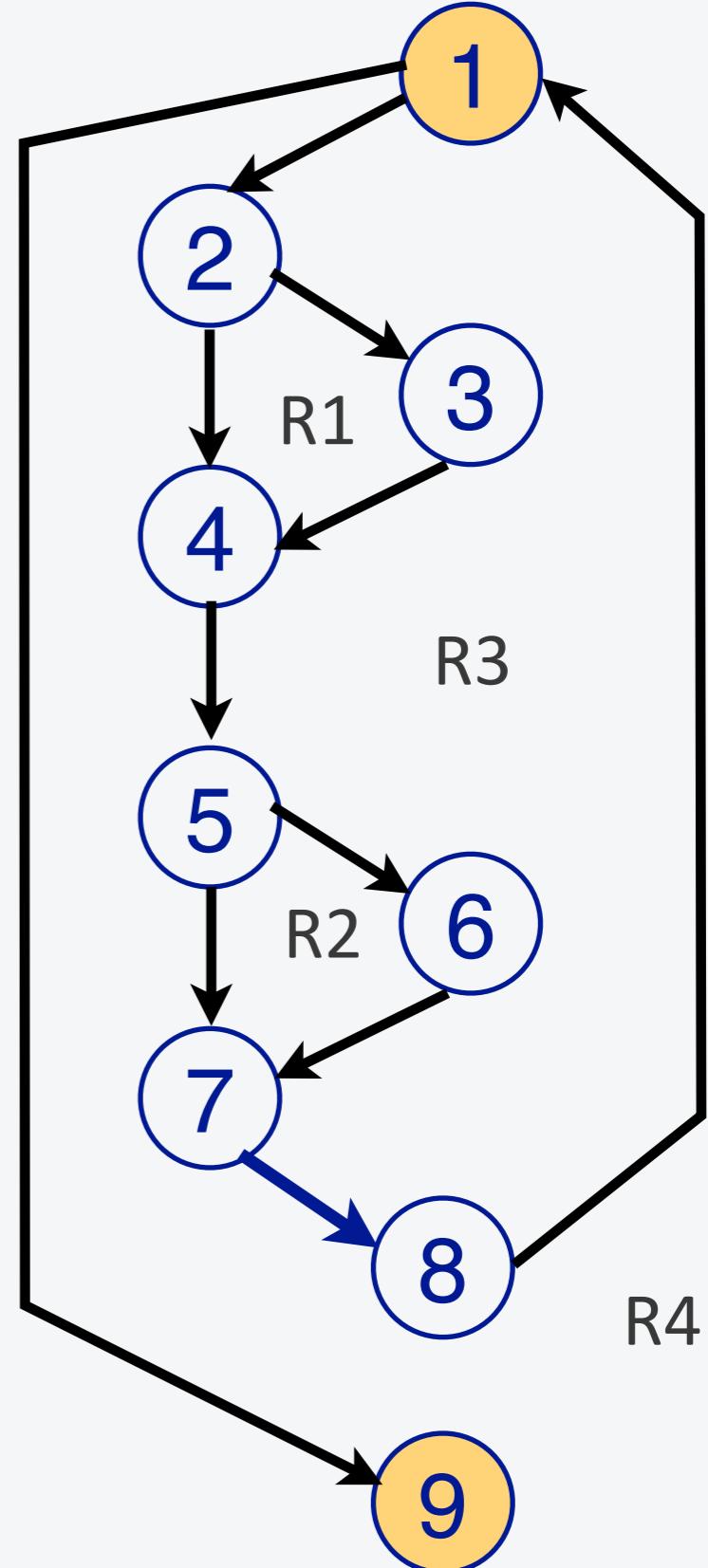
siendo:

B = número de "branches"

Las aristas: 1-9, 1-2, 2-3, 2-4, 5-6, 5-7 son "branches"

D = número de "Decision points"

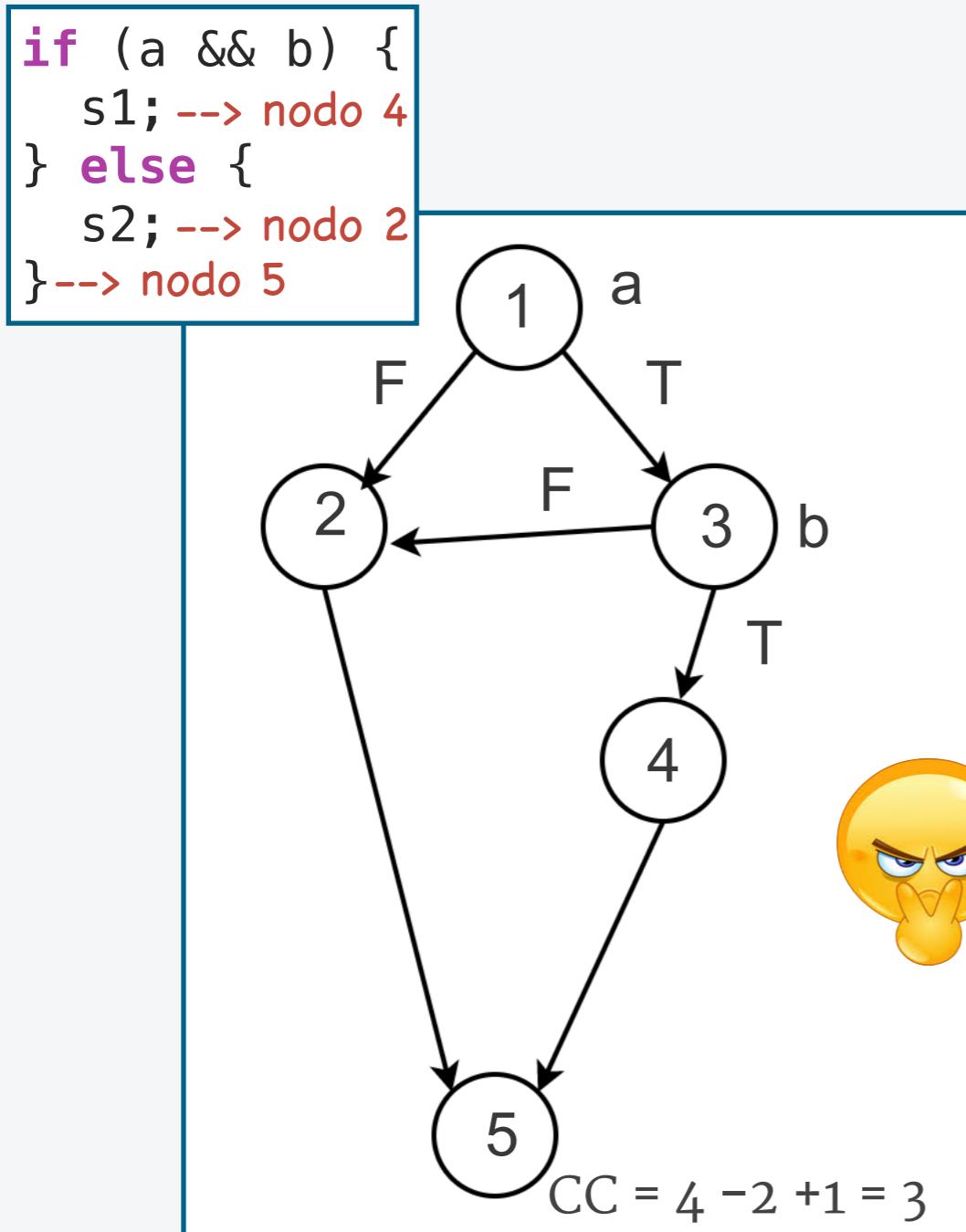
Los nodos: 1, 2 y 5 son "decision points"



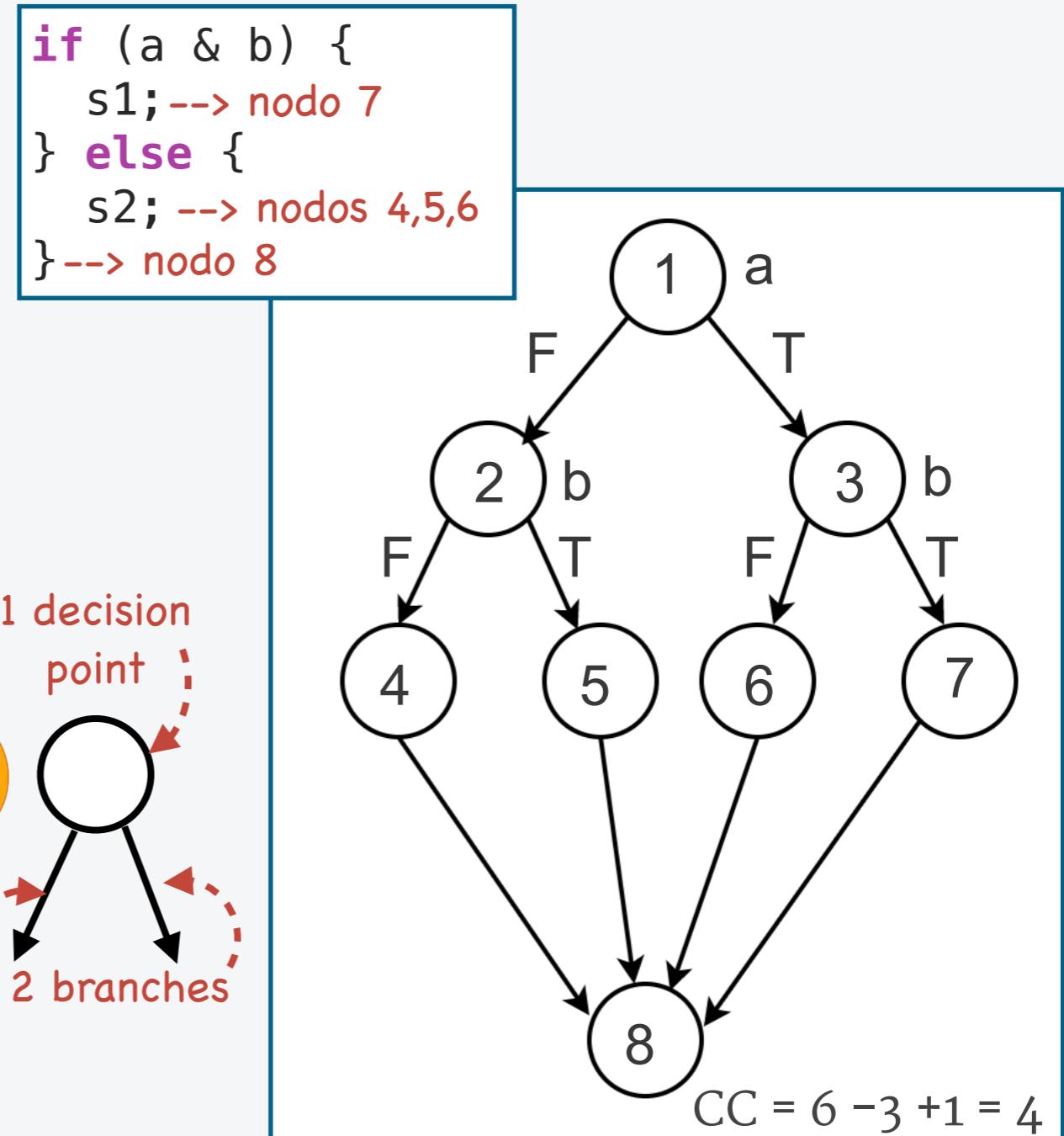
EJEMPLOS DE CÁLCULO DE CC CON JACOCO

JaCoCo cuenta Branches y Decision Points para calcular el valor de CC

Operador lógico AND con cortocircuito



Operador lógico AND sin cortocircuito



Las aristas: 1-2, 1-3, 3-2, 3-4 son "branches"
Los nodos 1, 3 son "decision points"

Las aristas: 1-2, 1-3, 2-4, 2-5, 3-6, 3-7
son "branches"
Los nodos 1, 2, 3 son "decision points"

P

¿PARA QUÉ SIRVE LA CC?

S

CC es una métrica muy útil, no sólo para testing

P

○ Mide la complejidad lógica del código:

- Cuanto mayor sea su valor el código resulta mucho más difícil de mantener (y de probar), con el consiguiente incremento de coste así como el riesgo de introducir nuevos errores
- Si el valor es muy alto (> 15) puede llevarnos a tomar la decisión de refactorizar el código para mejorar la mantenibilidad del mismo

○ Nos da una cota superior del **número máximo de tests** a realizar de forma que se garantice la ejecución de **TODAS las líneas de código** al menos una vez, y también nos garantiza que **TODAS las condiciones** se ejecutarán en sus vertientes verdadera y falsa (cuidado: cada nodo del grafo debe representar como máximo una única condición)

○ La herramienta JaCoCo calcula de forma automática el valor de CC

PLUGIN JACOCO PARA MAVEN

<https://www.jacoco.org/jacoco/trunk/doc/maven.html>

- JaCoCo instrumenta los ficheros .class para obtener los datos de cobertura. La instrumentación tiene lugar "on-the-fly" usando un mecanismo denominado "Java Agent"
 - Los ficheros .class se pre-procesan cuando la máquina virtual procede a cargar dichas clases.
 - Los datos recopilados durante la ejecución de los tests se guardan en un fichero
- **jacoco:prepare-agent**
 - Prepara el valor de la propiedad "argLine" para para pasarla como argumento a la JVM durante la ejecución de las pruebas unitarias.
 - Por defecto se asocia a la fase "**initialize**"
 - Entre otras cosas, indica qué clases deben ser instrumentadas, y cuál será el fichero con los datos resultantes del análisis de cobertura (**target/jacoco.exe**)
- **jacoco:prepare-agent-integration**
 - Es similar a la anterior, pero para las pruebas de integración. Por defecto se asocia a la fase "**pre-integration-test**", y los resultados del análisis se guardan en **target/jacoco-it.exe**
- **jacoco:report**
 - A partir de los datos obtenidos con el análisis de cobertura durante la ejecución de las pruebas unitarias (**target/jacoco.exe**), genera un informe en formato html, xml y cvs (en el directorio **target/site/jacoco**). Esta goal está asociada por defecto a la fase "**verify**"
- **jacoco:report-integration**
 - A partir de los datos obtenidos con el análisis de cobertura durante la ejecución de las pruebas de integración (**target/jacoco-it.exe**), genera un informe en formato html, xml y cvs (en el directorio **target/site/jacoco-it**). Esta goal está asociada por defecto a la fase "**verify**"

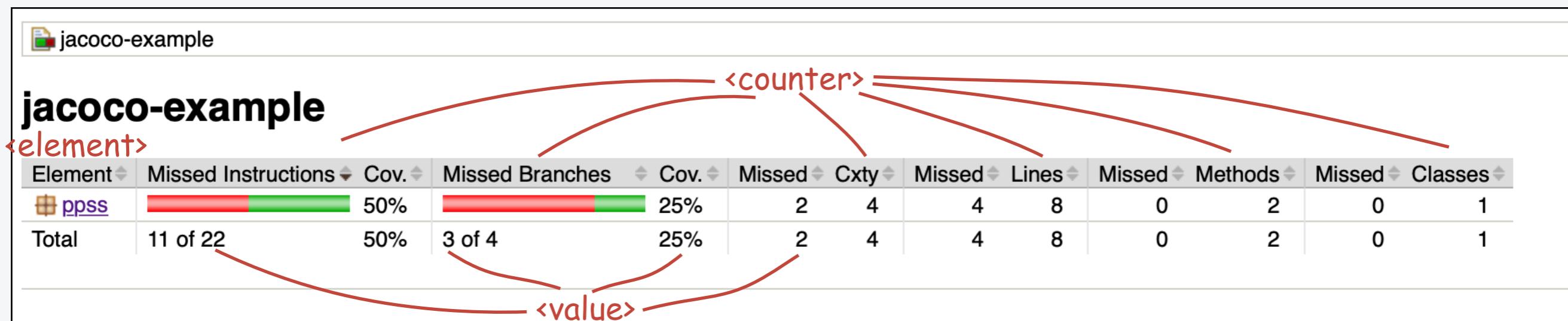
PLUGIN JACOCO PARA MAVEN

P

○ jacoco:check

P

- Comprueba si se ha alcanzado un determinado valor de cobertura, y detiene el proceso de construcción si no se alcanza dicho valor
- Por defecto se asocia a la fase "verify", y usa los datos del fichero jacoco.exe
- Es posible configurar la cobertura a diferentes **niveles** (etiqueta <element>):
 - * BUNDLE, PACKAGE, CLASS, SOURCEFILE o METHOD
 - * BUNDLE es a nivel de aplicación, PACKAGE es a nivel de paquete, ...
- Para cada nivel, se puede indicar el **contador** a configurar (etiqueta <counter>):
 - * INSTRUCTION, LINE, BRANCH, COMPLEXITY, METHOD, CLASS



- Para cada contador, se pueden establecer **valores** máximos o mínimos, sobre los diferentes valores calculados (etiqueta <value>):
 - * TOTALCOUNT, COVEREDCOUNT, MISSEDCOUNT, COVEREDRATIO, MISSED RATIO

EJEMPLO: CLASE A PROBAR Y CLASE DE TEST

- Nuestra clase a probar tiene un método denominado two_ifs

```
1 package ppss;
2
3 public class App
4 {
5     public int two_ifs(int i) {
6         if (i<20) {
7             System.out.println("Soy un numero menor que 20");
8             return 1;
9         }
10        if (i % 2 == 0) {
11            System.out.println("Soy numero par");
12            return 2;
13        }
14        return 0;
15    }
16 }
```

src/main/java/ppss/App.java

- Escribimos un test unitario para el método two_ifs()

```
6 public class AppTest {
7
8     @Test
9     public void testSimpleTwo_ifs() {
10        App app = new App();
11        //Ejecutamos un metodo de la clase
12        int n = app.two_ifs(11);
13        Assert.assertEquals(1,n);
14    }
15 }
```

src/test/java/ppss/AppTest.java

ANÁLISIS DE COBERTURA Y GENERACIÓN DE INFORMES (I)

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.7</version>

  <executions>
    <execution>
      <id>default-prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>default-report</id>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
    <execution>
      <id>default-check</id>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
    <configuration>
      <rules>
        <rule>
          <element>BUNDLE</element>
          <limits>
            <limit>
              <counter>COMPLEXITY</counter>
              <value>COVEREDRATIO</value>
              <minimum>0.60</minimum>
            </limit>
          </limits>
        </rule>
      </rules>
    </configuration>
  </executions>
</plugin>
```

podemos incluir todas las "reglas" que consideremos necesarias

Preparamos la instrumentación y el análisis de cobertura

Generamos el informe

Establecemos unos niveles de cobertura

La construcción se detendrá si, a nivel de proyecto, la complejidad ciclomática no alcanza el 60%

- Comando maven:
 - mvn verify
- Obtenemos como resultado:

```
[INFO] --- jacoco-maven-plugin:0.8.7:check
(default-check) @ jacoco-example ---
[INFO] Loading execution data file /Users/ppss/
jacoco-example/target/jacoco.exec
[INFO] Analyzed bundle 'jacoco-example' with 1
classes
[WARNING] Rule violated for bundle jacoco-
example: complexity covered ratio is 0.50, but
expected minimum is 0.60
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
```
- Hasta que no consigamos los valores de cobertura especificados, nuestra construcción no terminará con éxito.

Y AHORA VAMOS AL LABORATORIO...

Practicaremos la generación y análisis de informes de cobertura de nuestros tests

P

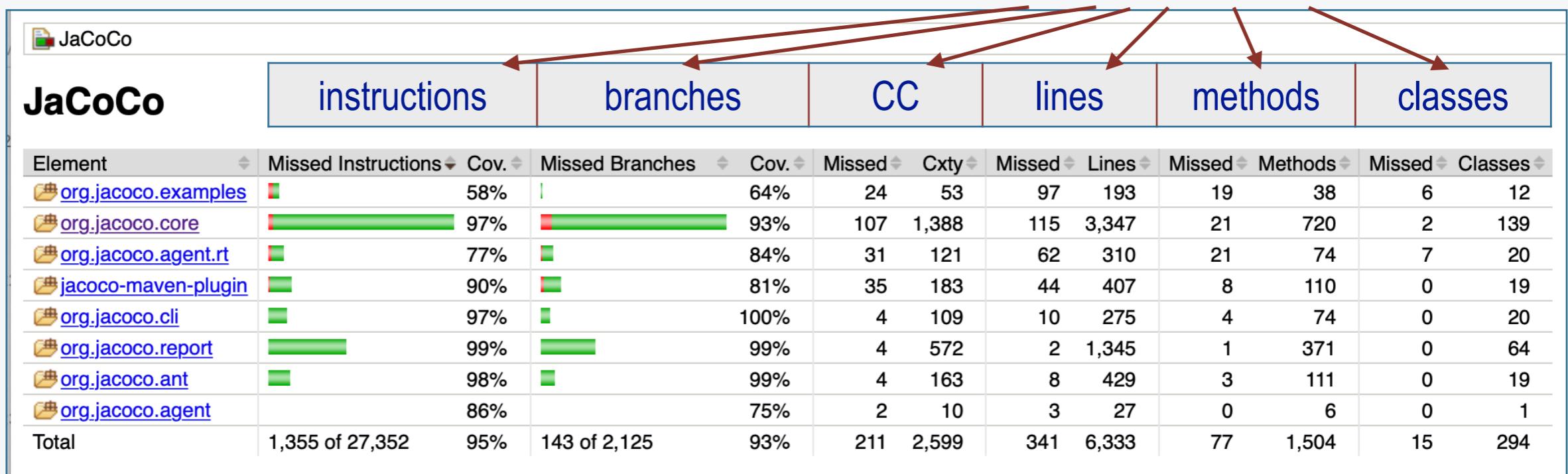
Camino	Datos Entrada	Resultado Esperado
C1	d1=... d2=... ...	r1
..		
CM	d1=... d2=... ...	rM



JaCoCo: instrumenta los .class antes de ejecutar los tests.

Analiza la cobertura después de ejecutar los tests y genera un informe

Informe de cobertura



REFERENCIAS BIBLIOGRAFICAS

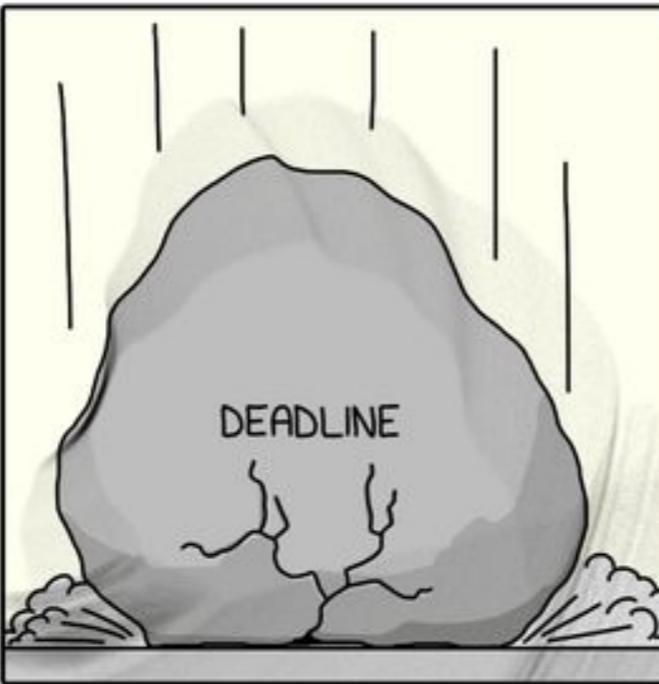
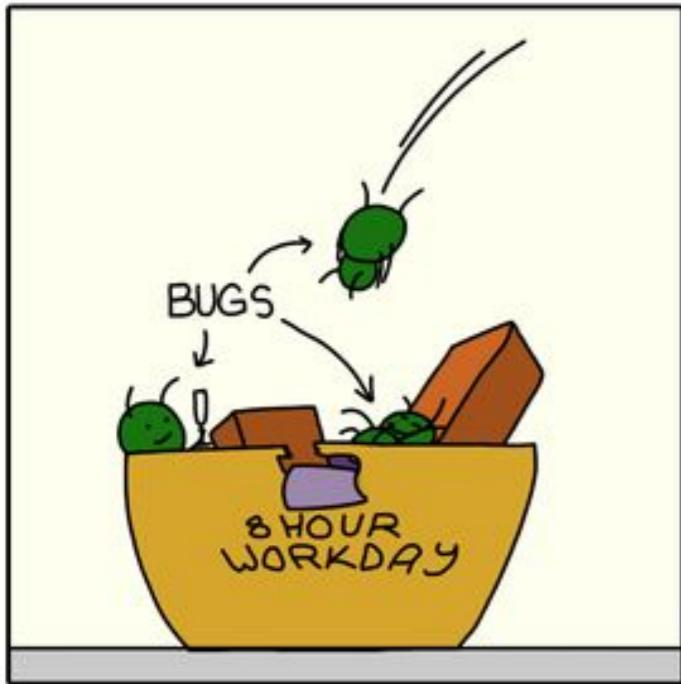
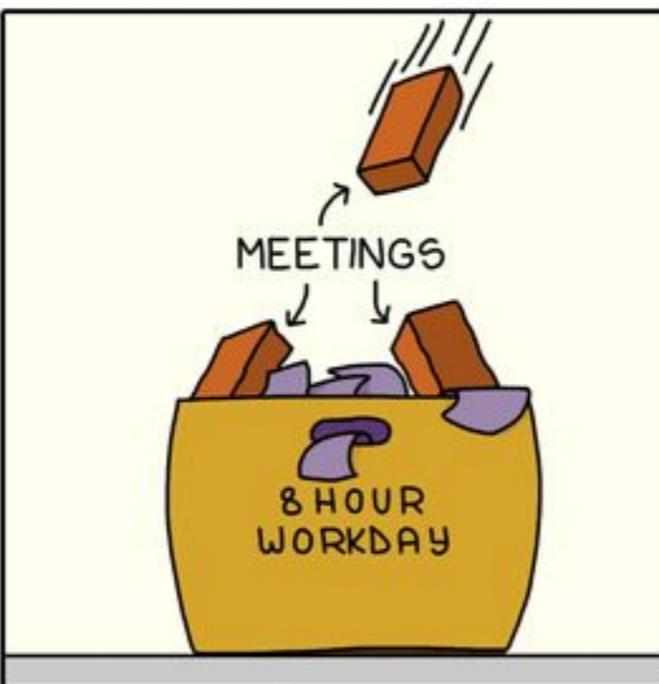
P

S

P

- “So you think you're covered?” (Keith Gregory)
 - <http://www.kdgregory.com/index.php?page=junit.coverage>
- Pragmatic Software Testing. Rex Black. John Wiley & Sons. 2007
 - Capítulo 21
- A practitioner's guide to software test design. Lee Coopeland. Artech House. 2004
 - Capítulo 10
- JaCoCo (<https://www.jacoco.org/jacoco/trunk/doc/index.html>)

9 TO 5



MONKEYUSER.COM

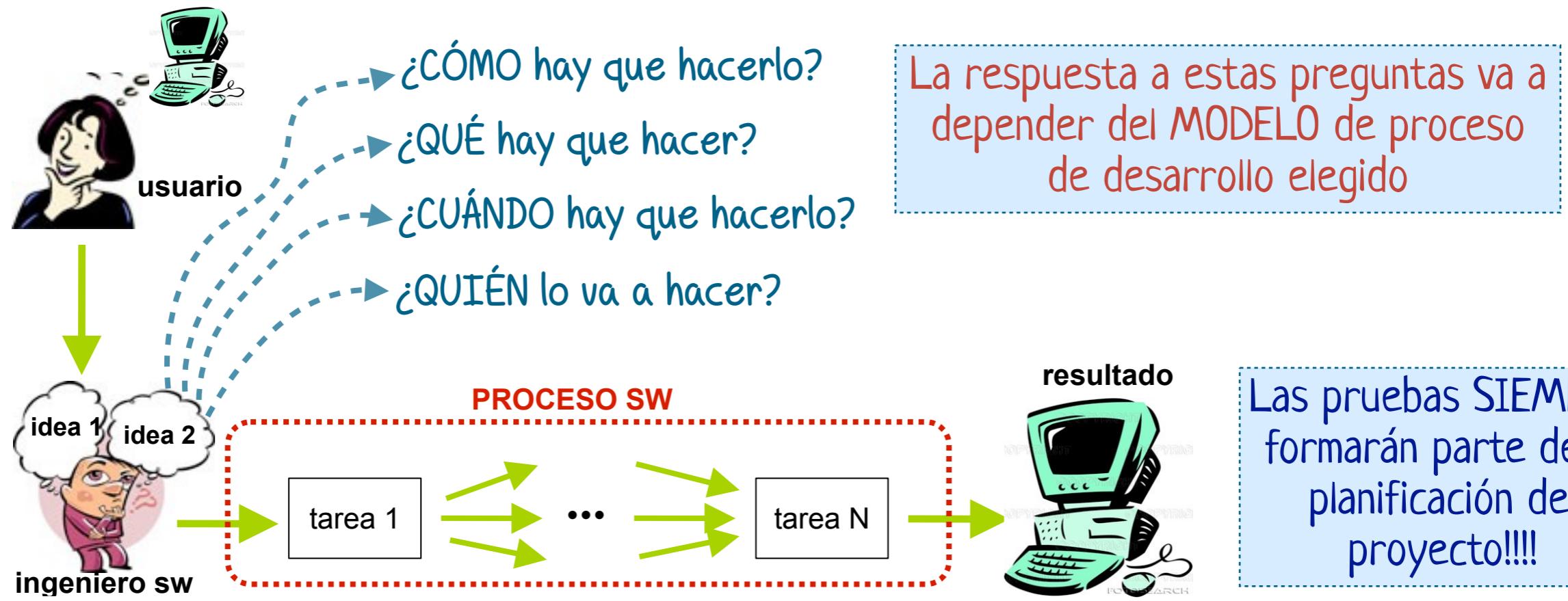
Sesión S11: Planificación de pruebas

Planificación

- Planificación del proyecto
- Efectividad y alcance de las pruebas
- Proceso de desarrollo y niveles de pruebas
- Proceso de pruebas
- Las pruebas en diferentes modelos de proceso
- Pruebas y diseño: TDD vs BDD
- Otras prácticas de pruebas: Integraciones continuas (CI)

LA IMPORTANCIA DE LA PLANIFICACIÓN

- Cuando se **planifica** un proyecto, aunque el propósito siempre es el mismo...

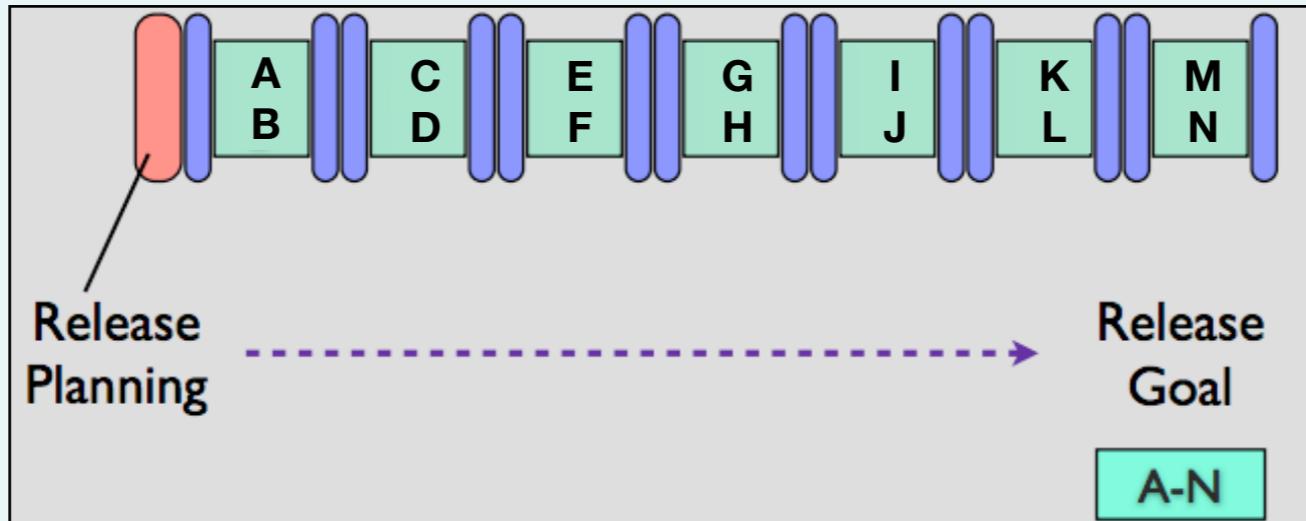


- ... se hace de forma diferente dependiendo del **modelo de proceso**:
 - Planificación predictiva vs. adaptativa
 - Los modelos iterativos y ágiles realizan una planificación adaptativa
 - Se establecen diferentes **niveles de planificación**: cada modelo de proceso considera determinados **horizontes**:
 - Los modelos ágiles planifican como mínimo a nivel de día, iteración y release

P PLANIFICACIÓN PREDICTIVA VS. ADAPTATIVA S

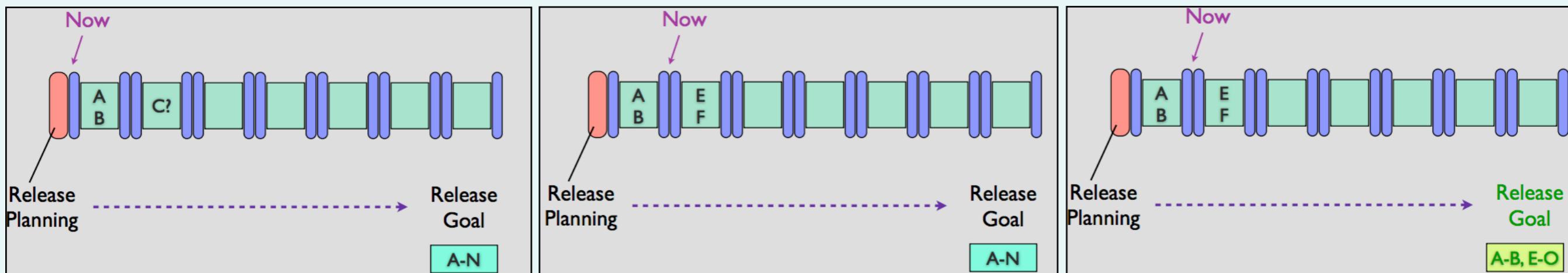
Imágenes adaptadas de: http://www.odd-e.com/material/2012/07_mda_nanyang/short_intro_agile.pdf

P Predictiva



Soporta muy mal los cambios!!!

A Adaptativa



No es posible tener claras todas las "variables" que intervienen en el desarrollo de un proyecto al comienzo del mismo (cono de incertidumbre), como por ejemplo requerimientos específicos, los detalles de la solución, cuestiones sobre el personal del proyecto, etc.



Un plan ADAPTATIVO intenta encontrar un equilibrio entre el esfuerzo invertido en realizar el plan, frente a la información (conocimiento) disponible en cada momento. Proporciona "agilidad": resulta "sencillo" incluir cambios!!!

MÚLTIPLES NIVELES DE PLANIFICACIÓN (I)

Un plan con muy poca exactitud es incontrolable

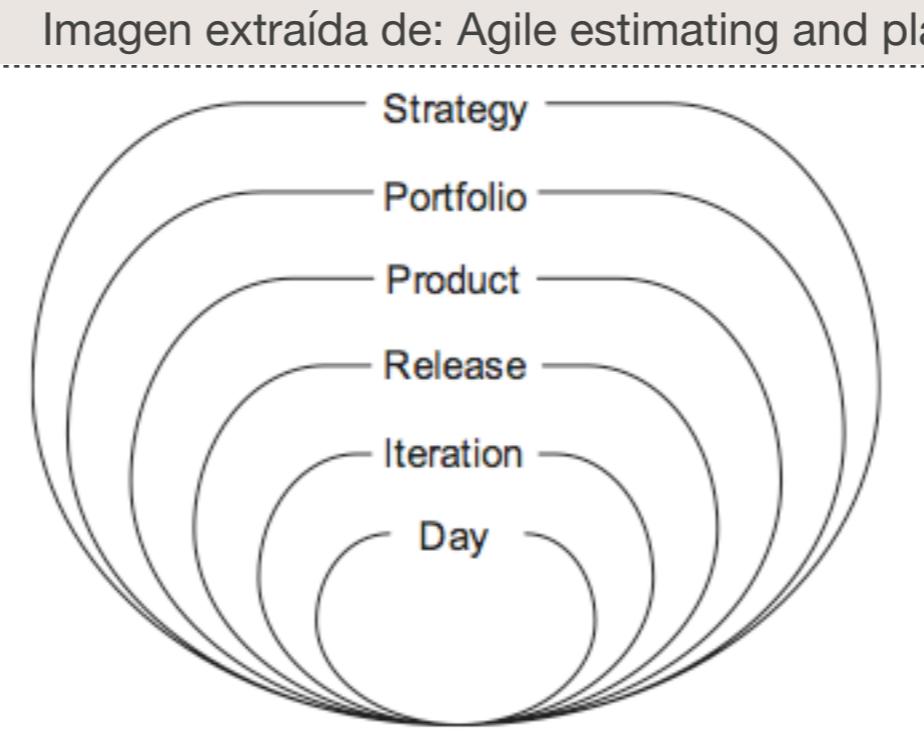
- Cuando nos marcamos objetivos, es importante recordar que no podemos "ver más allá del horizonte" y que la **exactitud** en nuestro plan decrecerá rápidamente tanto más cuanto más sobrepasemos dicho horizonte
 - Un proyecto "está en riesgo" si su planificación se extiende más allá del horizonte del planificador y no incluye tiempo para que éste "levante la cabeza", vuelva a mirar al nuevo horizonte, y realice los ajustes necesarios.

Imagen extraída de: Agile estimating and planning. Chap 3

Release: considera las historias de usuario que serán desarrolladas en la SIGUIENTE entrega del desarrollo al cliente

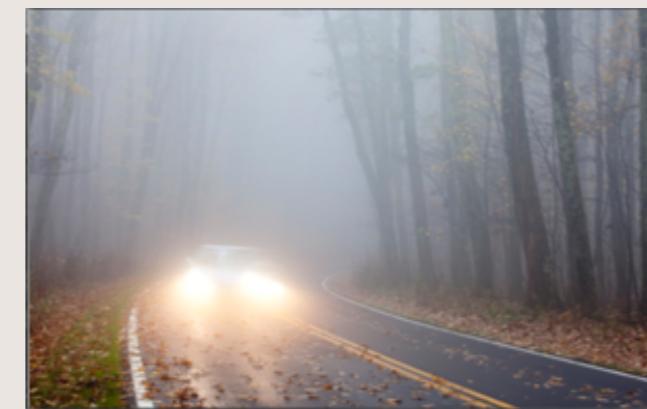
Product: considera la evolución de posteriores "releases"

Portfolio: considera la selección de desarrollos dentro de la estrategia de la empresa (desarrolladora)



Los modelos ágiles planifican como mínimo a nivel de **día, iteración y release**

- Cada modelo de proceso considera determinados horizontes. Cada horizonte proporciona una "visibilidad" adecuada para el correcto progreso del desarrollo del proyecto



Con una mala "visibilidad" no es posible planificar correctamente!!!



MÚLTIPLES NIVELES DE PLANIFICACIÓN (II)

Cada modelo de proceso requiere planes con diferentes horizontes

Imágenes extraídas de: The art of agile development. James Shore. 2008. Cap. 3

- P
- Cuando nos marcamos objetivos, es importante recordar que no podemos "ver más allá del horizonte" y que la exactitud en nuestro plan decrecerá rápidamente tanto más cuanto más sobrepasemos dicho horizonte

(a) *Waterfall lifecycle*



Modelo secuencial
Horizonte: release

(b) *Iterative lifecycle*



Modelo iterativo
Horizontes: release,
iteración

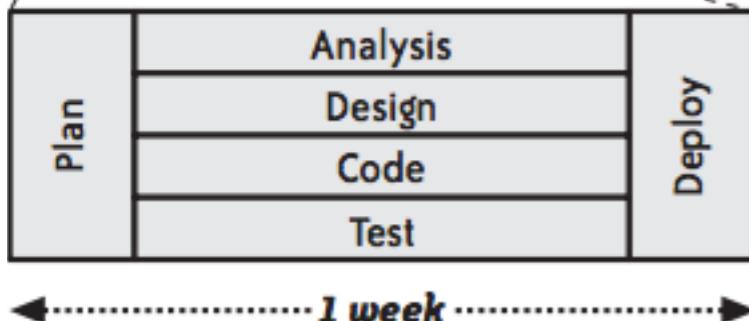


Modelo ágil (XP)
Horizontes: release,
iteración, dia

\$ = Potential release

Sesión 11: Planificación de pruebas

Una aproximación iterativa no necesariamente implica una mayor productividad. Significa que el equipo tiene una retroalimentación mucho más frecuente. Como consecuencia el equipo fácilmente "relaciona" los éxitos y fallos con sus causas subyacentes. Es mucho menos costoso REPARAR LOS FALLOS



ITERACIONES Y TIME-BOXING

S En un modelo iterativo es FUNDAMENTAL que las iteraciones sean del mismo "tamaño" y que sean time-boxed

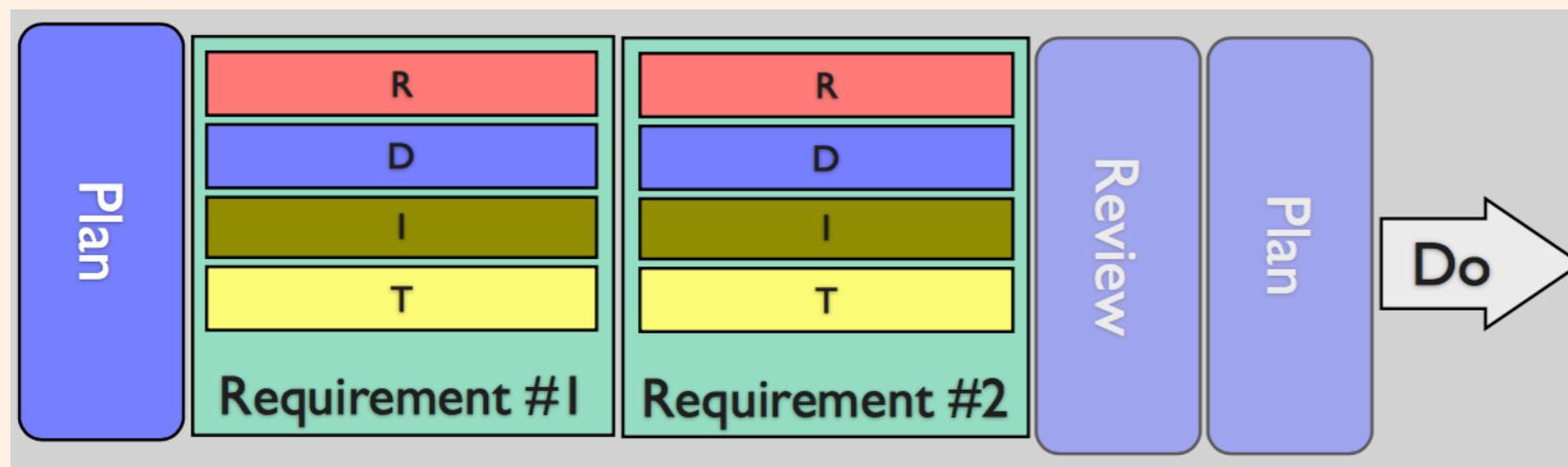
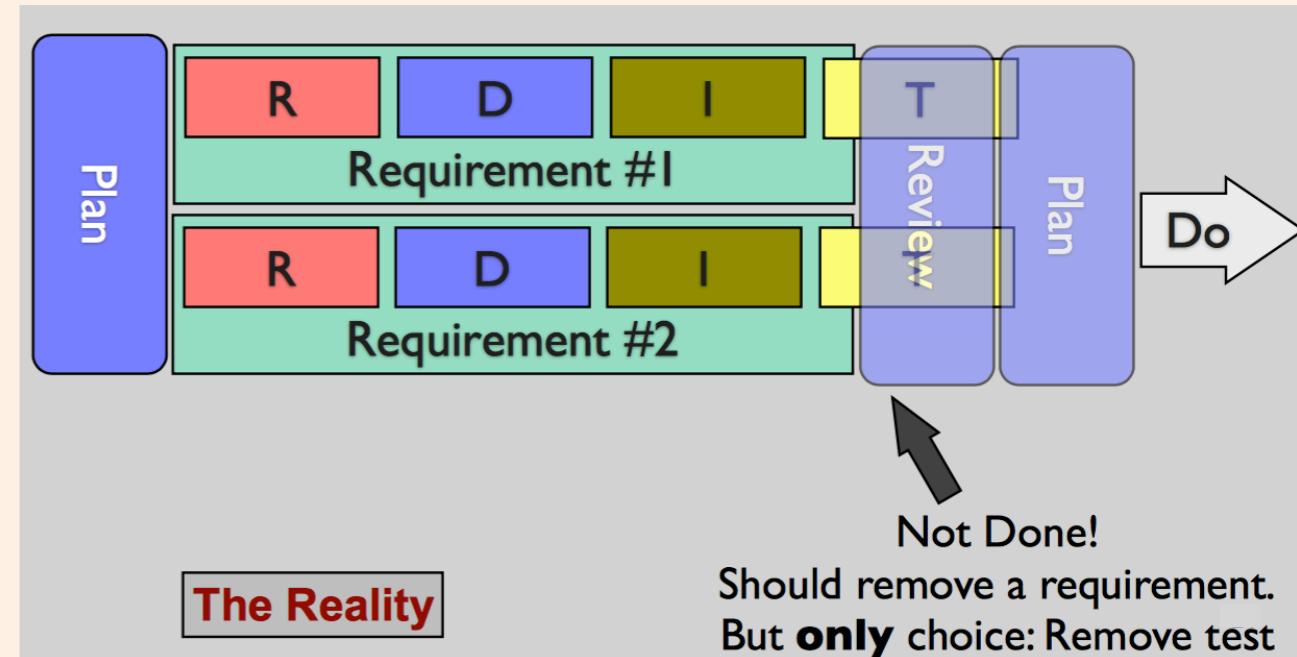
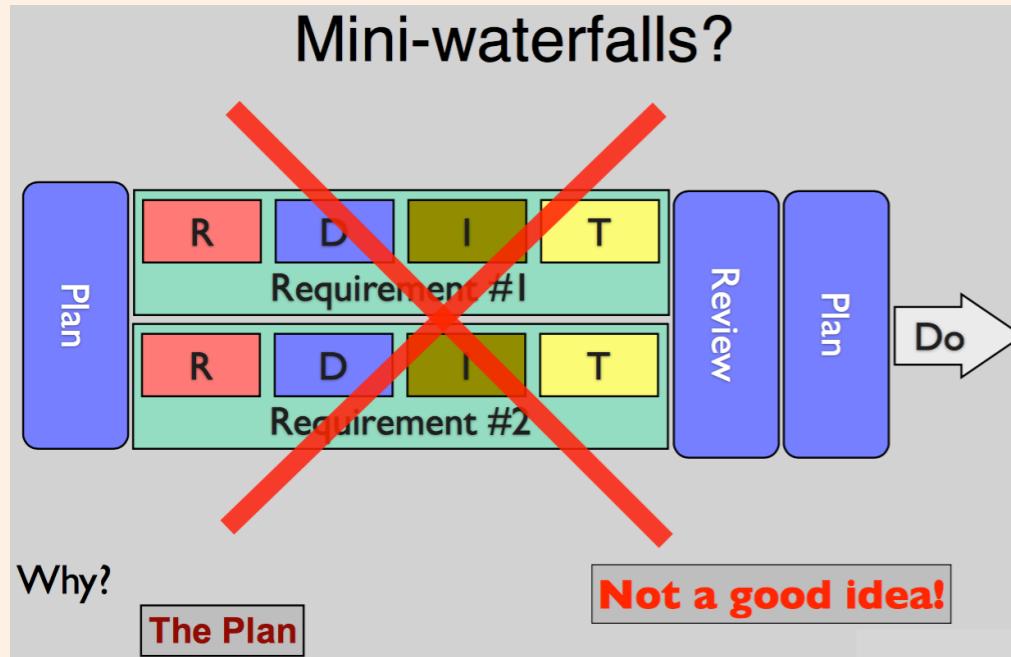
P

- ¿Quién decide lo que hay que hacer en la siguiente iteración?

P

- Los modelos iterativos, en general, están "conducidos" por el cliente (driven by the customer). Esto es importante puesto que el cliente puede validar sus expectativas cuanto antes

- Las iteraciones deben ser SIEMPRE time-boxed: nunca se retrasa el tiempo de entrega, es preferible y es mucho más sencillo cambiar el "scope"



Es un ERROR el planificar una iteración como una mini-cascada. Si lo hacemos así, no será posible cambiar el "scope" sin repercutir negativamente en el éxito del proyecto

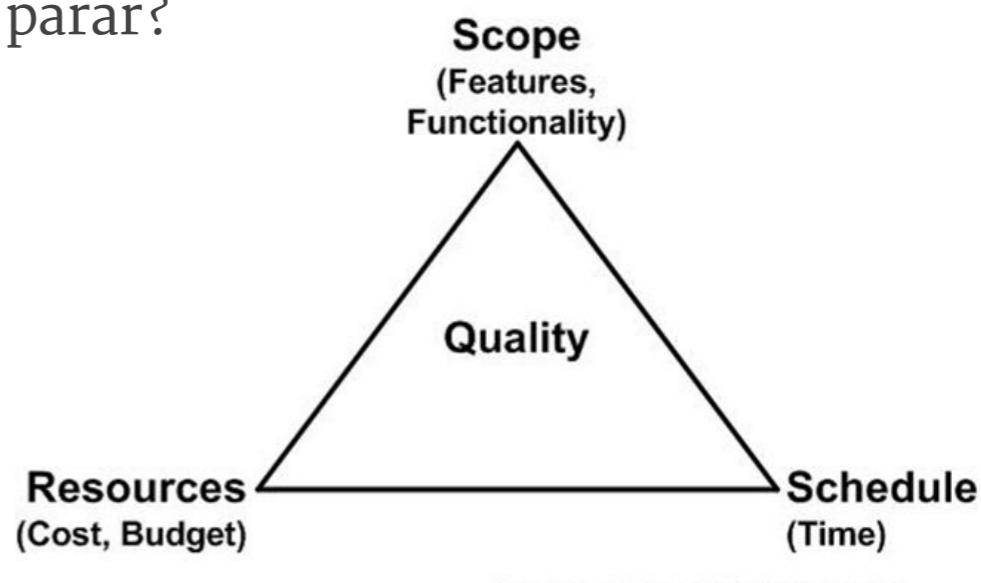
ALCANCE Y EFECTIVIDAD DE LAS PRUEBAS

Para planificar las pruebas es importante tener en cuenta las siguientes cuestiones:

- ¿Cuántas pruebas son suficientes, y cómo decidir cuándo parar?

- Factores para tomar la decisión: triángulo de recursos

El desarrollo del software debe equilibrar los tres vértices del triángulo: tiempo, dinero y funcionalidad. Estos tres recursos INFLUENCIAN la calidad (propiedades emergentes) que se incluyen (o no) en el software entregado



Copyright 2003-2006 Scott W. Ambler

- Para conseguir un resultado aceptable hay que:

- **Priorizar**: decidir qué tests son más importantes
 - Fijar **criterios** para conseguir unos objetivos de pruebas de forma que sepamos cuándo parar. P.ej. qué áreas van a ser más probadas y con qué cobertura, qué nivel de defectos se van a tolerar en un producto entregado... (**completion criteria**)

- El objetivo final es asegurar que las pruebas son **EFFECTIVAS**

- Un buen test es aquél que encuentra un defecto. Si encontramos un defecto estamos creando una oportunidad de mejorar la calidad del producto

- El proceso de pruebas debe ser **EFICIENTE**:

- Encontrar el mayor número de defectos con el menor número de pruebas posibles
 - Para ello se deben utilizar buenas técnicas de **DISEÑO** de casos de prueba

EL PROCESO DE PRUEBAS

P

OBJETIVOS DEL PROCESO DE PRUEBAS

P

- Demostrar que el software satisface las expectativas del cliente
- Descubrir situaciones en las que el comportamiento del software es incorrecto, indeseable, o no conforme a las especificaciones

Test planning and control



Test analysis and design



Test implementation and execution



Evaluating exit criteria and reporting



Test closure activities

S
Recuerda que los objetivos tienen que ser cuantificables !!



- Se determina ¿QUÉ? se va a probar, ¿CÓMO?, ¿QUIÉN? y ¿CUÁNDO? (**planning**)
- Se determina QUÉ se va a hacer si los planes no se ajustan a la realidad (**control**)
- Se determina qué CASOS DE PRUEBA se van a utilizar
- Un caso de prueba es un DATO CONCRETO + RESULTADO ESPERADO
- Se implementan y ejecutan los tests. Es la parte más visible, pero no es posible que los test sean efectivos sin realizar los pasos anteriores
- Se verifica que se alcanzan los *completion criteria* (p.ej 85% de cobertura) y se genera un informe
- En este punto las pruebas ya han finalizado. Se trata de asegurar que los informes están disponibles, ...

NIVELES DE PRUEBAS

Tenemos que integrar las actividades de pruebas con el resto de actividades de desarrollo en el plan del proyecto

Secuencia temporal de niveles de pruebas (DINÁMICAS)



P OBJETIVO (cuantificable)	VERIFICACIÓN. Objetivo: encontrar defectos Realizada por los desarrolladores			VALIDACIÓN. Objetivo: ver en qué grado se satisfacen las expectativas del cliente. Requieren al usuario
	UNIDAD	INTEGRACIÓN	SISTEMA	ACEPTACIÓN
DISEÑO	Encontrar defectos en las unidades. Deben de probarse de forma AISLADA	Encontrar defectos en la interacción de las unidades. Debe establecerse un ORDEN de integración	Encontrar defectos derivados del comportamiento del sistema como un todo.	Valorar en qué GRADO se satisfacen las expectativas del cliente. Basadas en criterios de ACEPTACIÓN (prop. emergentes) cuantificables
AUTOMATIZACIÓN (implement. + ejecución)	Camino básico (CB) Particiones equivalentes (CN)	Guías (consejos) en función de los tipos de interfaces (CN)	Basado en casos de uso (CN) Transición de estados (CN)	Basado en requerimientos (CN) Basado en escenarios (CN) Pruebas de rendimiento (CN) Pruebas α y β (CN)

en un plan de pruebas hay que decidir QUÉ, CÓMO, CUÁNDO Y QUIÉN, para cada celda de la tabla (actividades de diseño + implementación)

PLANIFICACIÓN TEMPORAL DE LOS NIVELES DE PRUEBAS

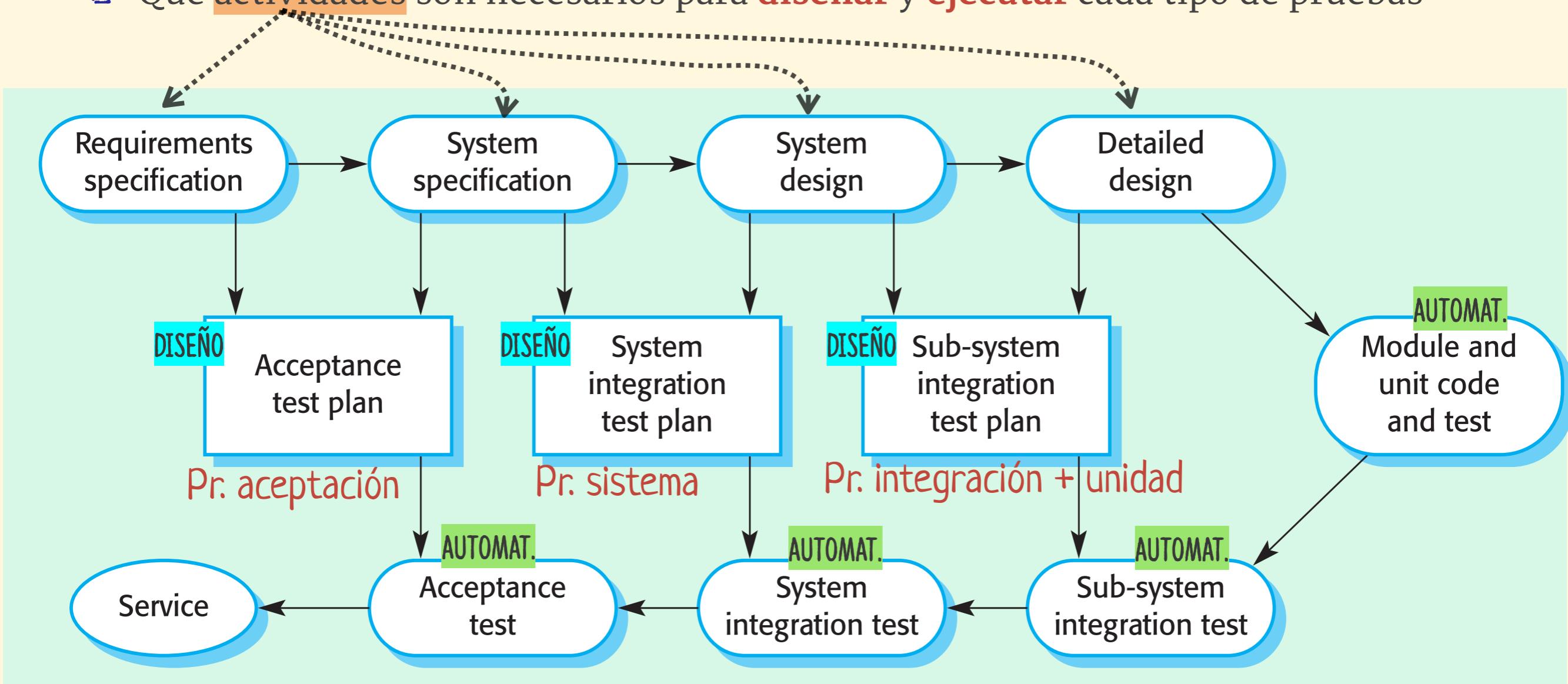
PROCESO

RESULTADO
DEL PROCESO

- Es importante conocer:

- CUÁNDO** se debe realizar cada **TIPO** de prueba

- Qué **actividades** son necesarios para **diseñar** y **ejecutar** cada tipo de pruebas



Tiempo
ESPECIF.
REQUERIM.

PUESTA EN
SERVICIO

Los procesos de **DISEÑO** y **AUTOMATIZACIÓN** de cada nivel de pruebas pueden estar muy "separados" en el tiempo



PRUEBAS EN MODELOS SECUENCIALES

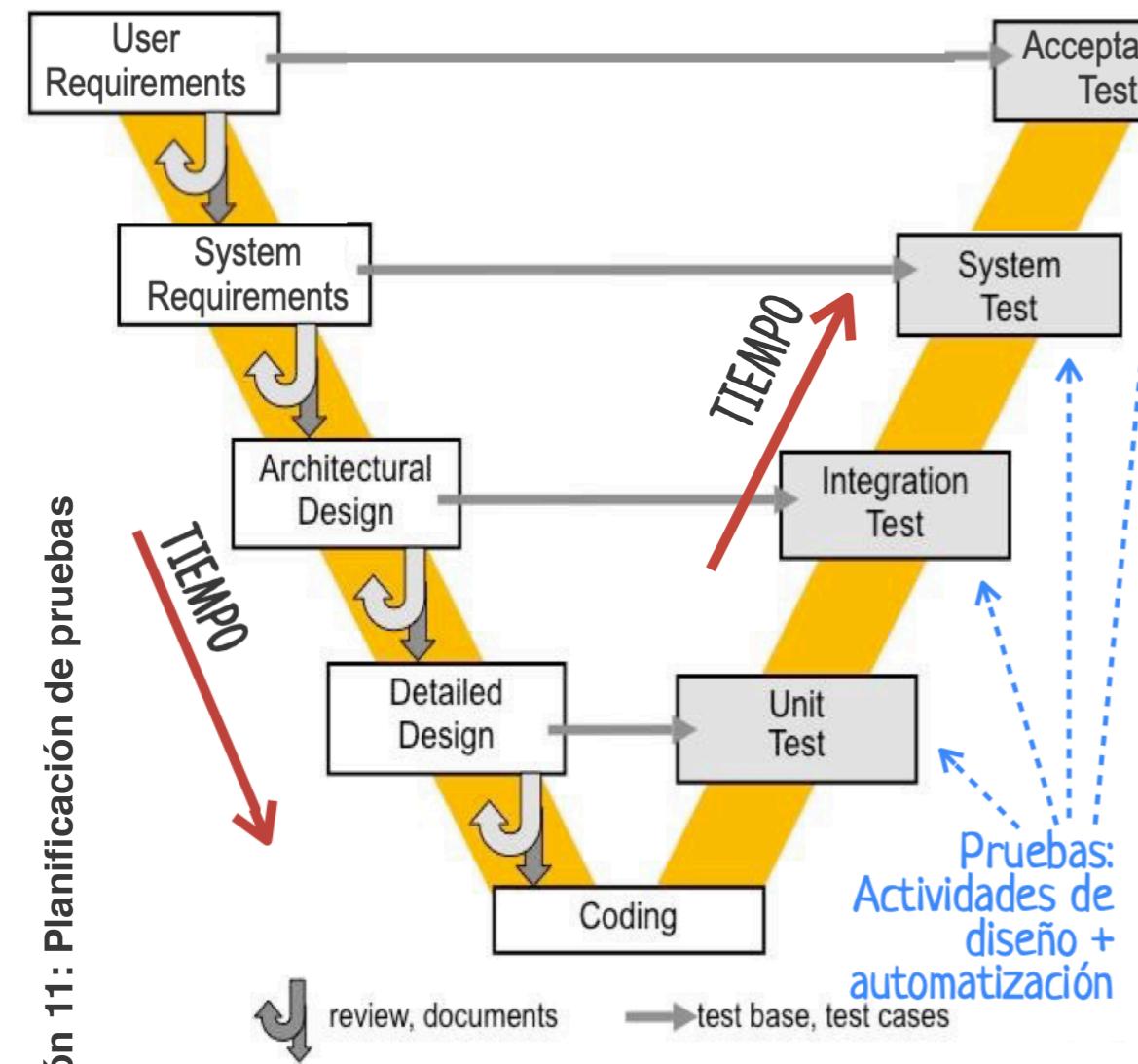
Un modelo secuencial
usa un único plan



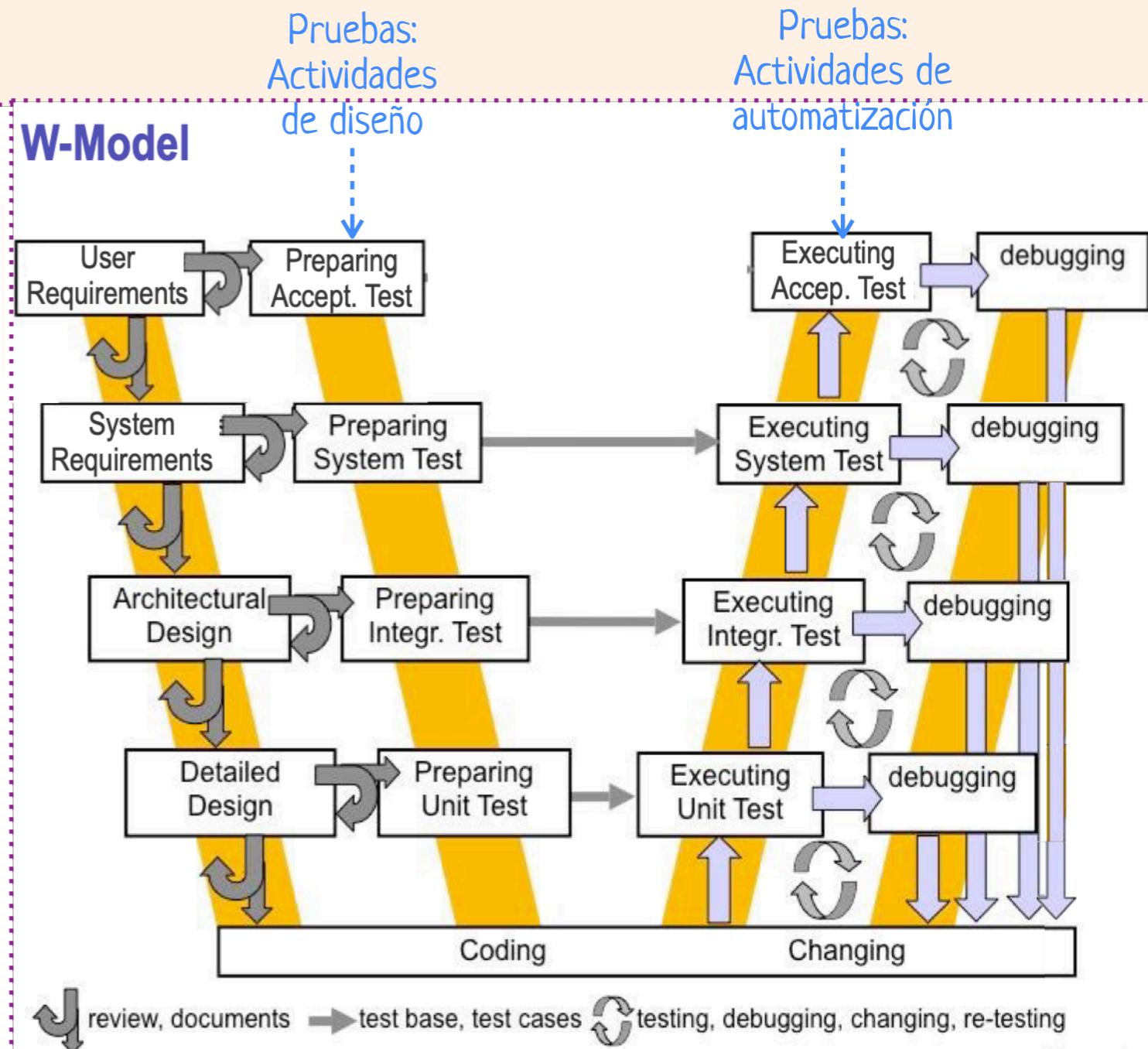
- Se trata de incluir en el plan de desarrollo las actividades de pruebas. Se debe contemplar un equilibrio entre las pruebas estáticas y las dinámicas

SECUENCIACIÓN TEMPORAL DE PRUEBAS EN UN MODELO DE DESARROLLO SECUENCIAL

V-Model



W-Model

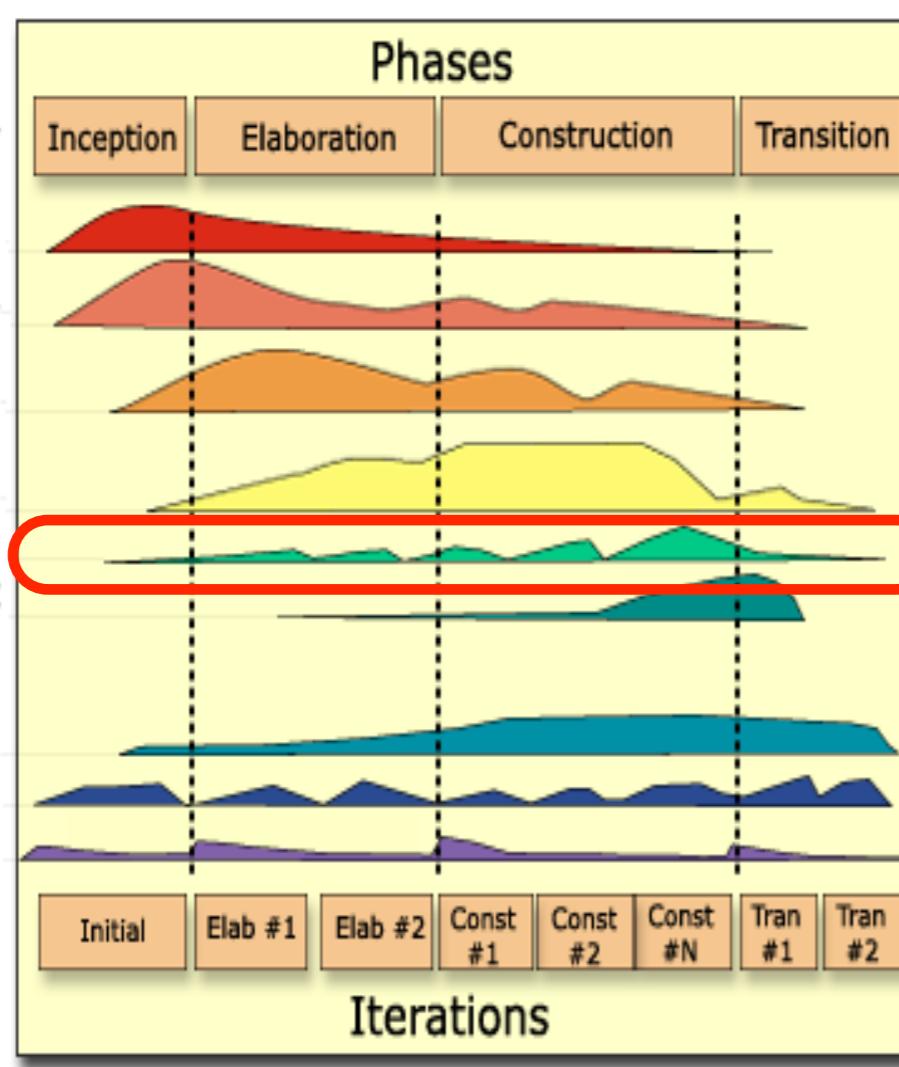


PRUEBAS EN MODELOS ITERATIVOS Y ÁGILES (I)

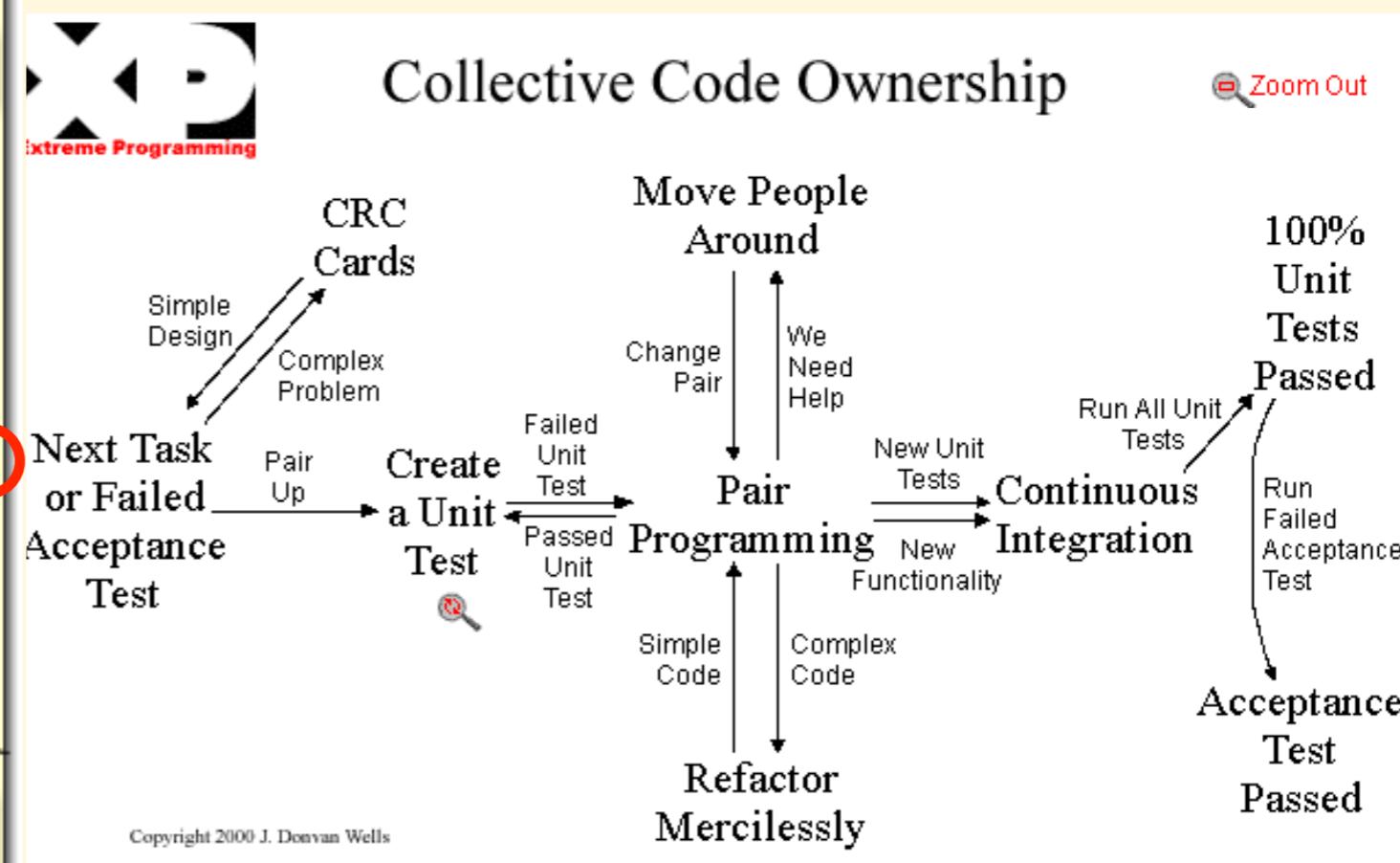
- En modelos **iterativos**, las pruebas se planifican a nivel de **ITERACIÓN** y **RELEASE** (una release está formada por un conjunto de iteraciones)

PRUEBAS EN MODELOS ITERATIVOS

Modelo UP



Modelo XP



Cada RELEASE se divide en 4 fases. Cada fase contiene una o más ITERACIONES

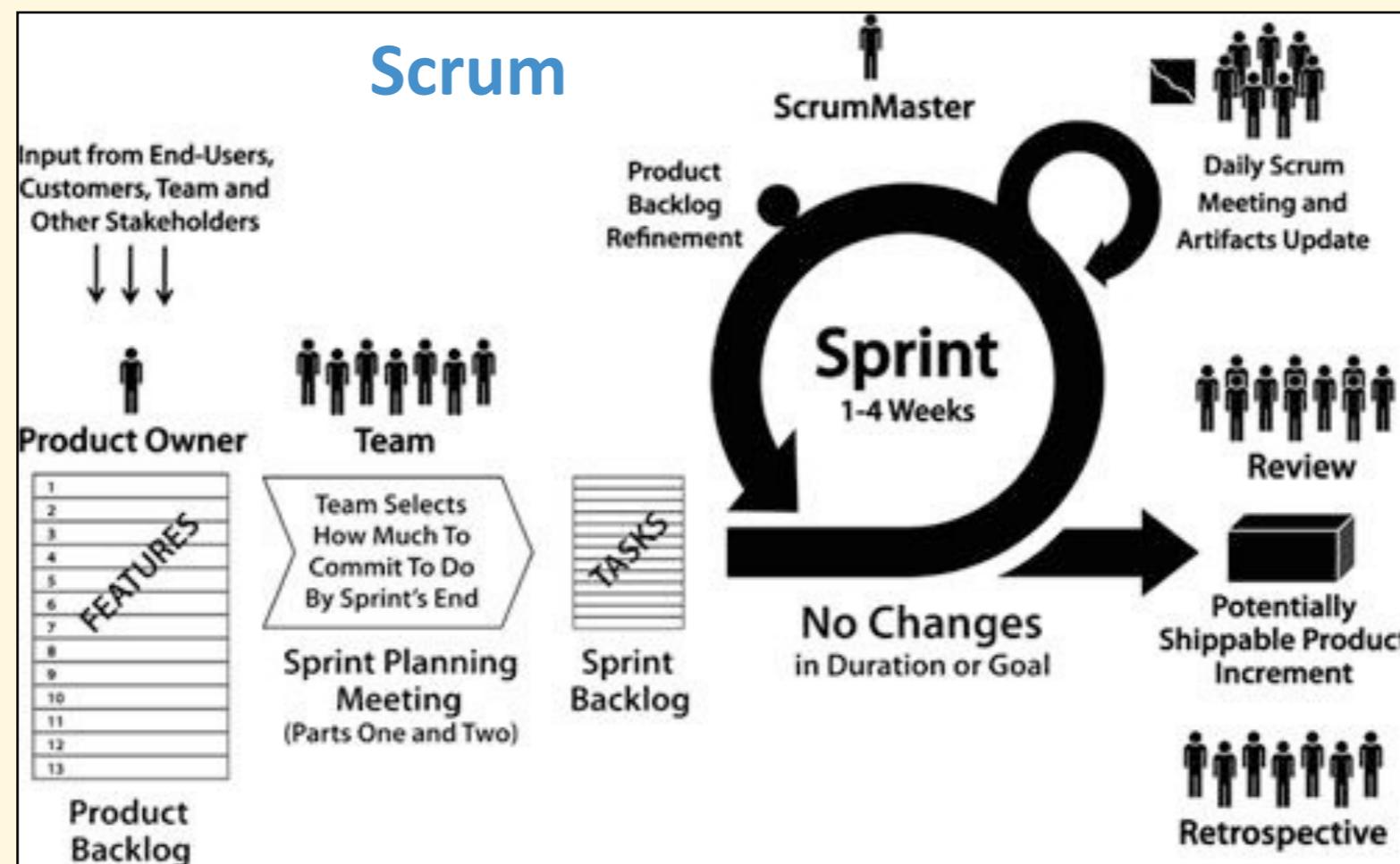
En modelos ágiles, las pruebas se planifican a nivel de DÍA, ITERACIÓN y RELEASE

PRO PRUEBAS EN MODELOS ITERATIVOS Y ÁGILES (II)

El **product owner** es responsable de conseguir el máximo valor de negocio del producto

Product backlog:

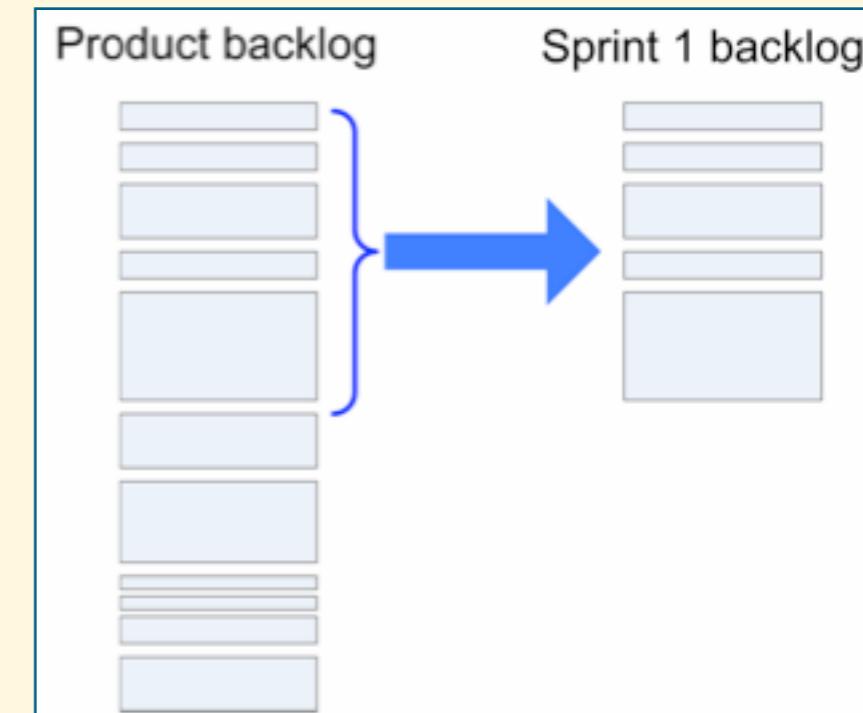
Lista de requerimientos priorizados por su valor de negocio (los valores más altos al principio de la lista).



En un proceso Scrum NO hay una fase de testing "separada" del resto de actividades del desarrollo.

Cuando un desarrollador termina una historia de usuario, los tests tienen que estar preparados para su ejecución. Si el test pasa, la historia es aceptada y se pasa a la siguiente. Una vez que se han probado todas las historias y han pasado los tests, se da por concluido el sprint y se pasa al siguiente

El **scrumMaster** es el que actúa como guía del grupo durante el proceso, "protege" al grupo del exterior, y sirve como ayuda al mismo. NO es un gestor de proyectos



¿QUÉ ASPECTO TIENE UN PLAN EN XP? (I)

Se planifica en varios niveles

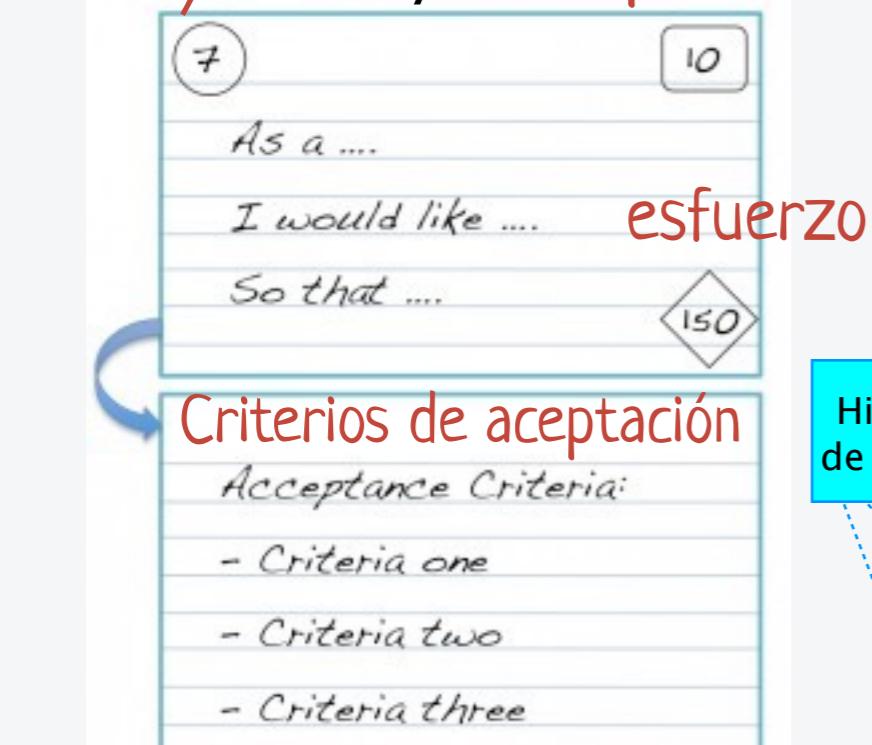
- Story cards: contienen historias de usuario. Representan características requeridas por el usuario (no casos de uso o escenarios). Se centran en el "beneficio" o resultado (value) que se pretende obtener. Deben describir un objetivo que permita a los testers evaluar una implementación exitosa de la misma. El formato suele ser:

As an [actor] I want [action] so that [achievement]. For example: As a Facebook member, I want to set different privacy levels on my page so that I can control who sees my page.

- Task list: lista de tareas para las story cards de una iteración

ITERACIÓN (= 1 semana)

nº story Story card prioridad



Las historias de usuario se **priorizan** según su valor de negocio. El conjunto de historias de usuario proporcionan la visión del producto.

Task board

Story	To Do	Tests Ready	In Process	To Verify	Hours
As a user, I can...	Code the... 8 Code the... 5 Test the... 6		Code the... SC 6 Code the... DC 4	Code the... LC 4	33
As a user, I can...	Code the... 8 Code the... 5				13
As a user, I can...	Code the... 3 Code the... 6		Code the... MC 4		13

Historias de usuario → Tareas

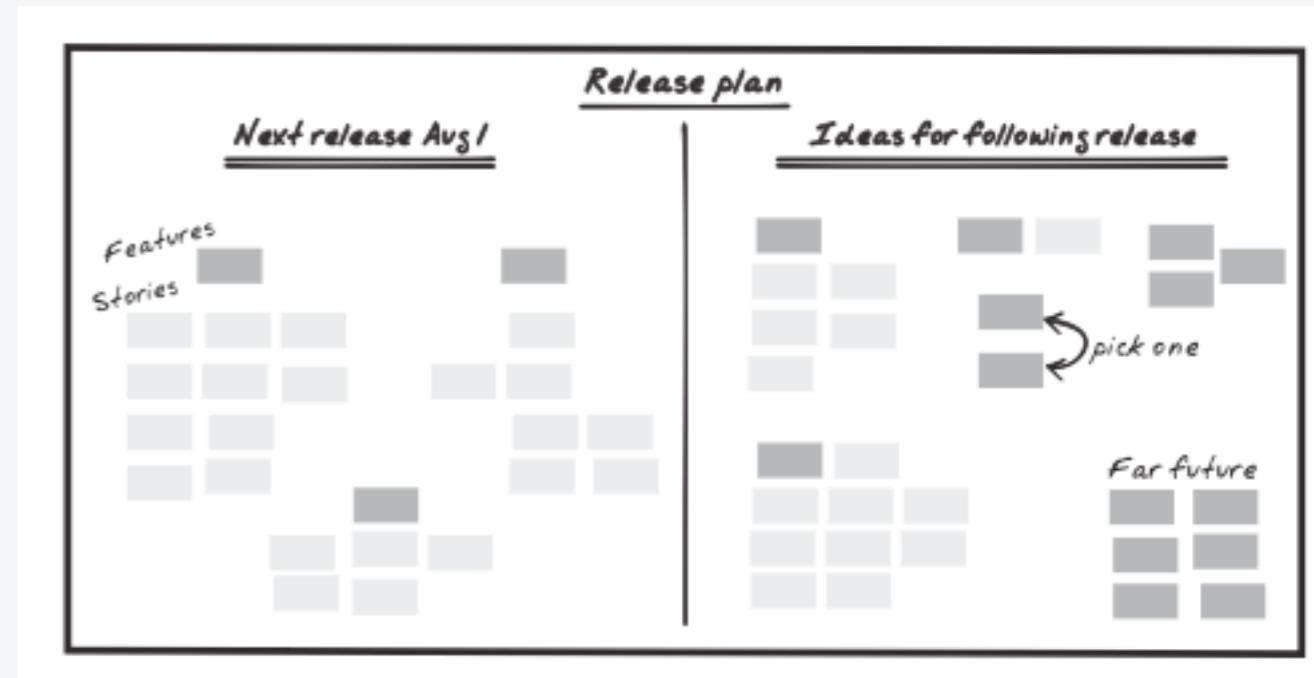
P ¿QUÉ ASPECTO TIENE UN PLAN EN XP? (II)

entrega = release

P O Release plan board

Primero se crea el plan de entregas, consistente en una lista de **historias de usuario** priorizadas por su valor de negocio (los valores más altos al principio de la lista). Se obtiene como resultado del proceso denominado *planning game*

No se asignan recursos (*¿quién?*), ni tiempo (*¿cuándo?*) hasta que comience cada iteración

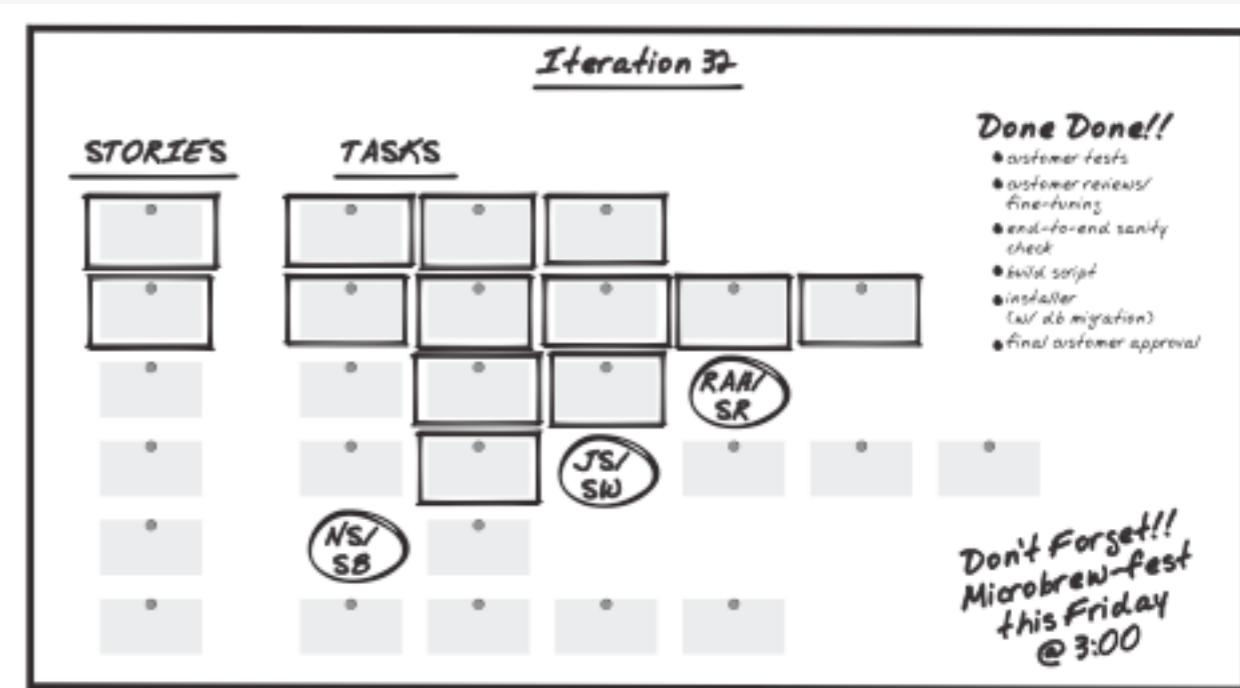


O Iteration plan board

Las historias de usuario se dividen en **tareas** concretas y son desarrolladas por los programadores.

Cuando se empieza a trabajar en una tarea, el programador "pega" la tarjeta del tablero en su ordenador, dejando sus iniciales. Cuando la tarea se termina, se vuelve a colocar en el tablero y se marca en verde

Cada TASK tiene asociado un conjunto de TESTS (en el reverso de la tarjeta)



¿QUÉ ASPECTO TIENE UN PLAN EN SCRUM?

Gráfico de seguimiento

SPRINT (3-4 semanas)

Scrum task board

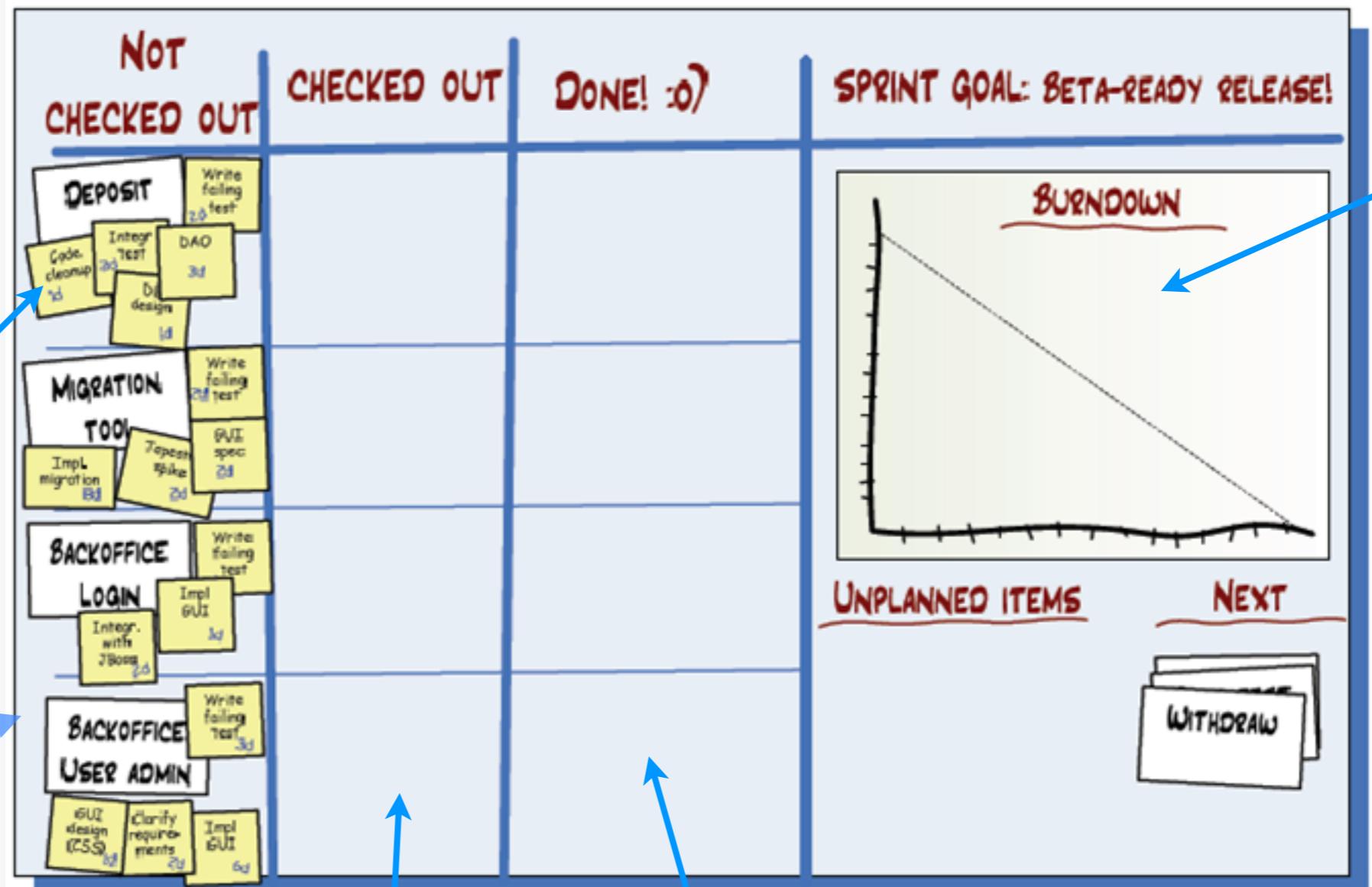
Cada TASK tiene
asociado un
conjunto de TESTS
(en el reverso de la
tarjeta)

Pila sprint: Lista de
historias de
usuario + tareas

Tareas en
proceso

Una tarea se
marca como
DONE cuando
pasa los tests

Dichos tests constituyen los
criterios de aceptación o
requerimientos de alto nivel
necesarios para que la tarea
pase al estado DONE



PRUEBAS Y DISEÑO: TDD

S

P

- TDD (Test Development Driven) es una práctica de pruebas utilizada en desarrollos ágiles, como por ejemplo XP

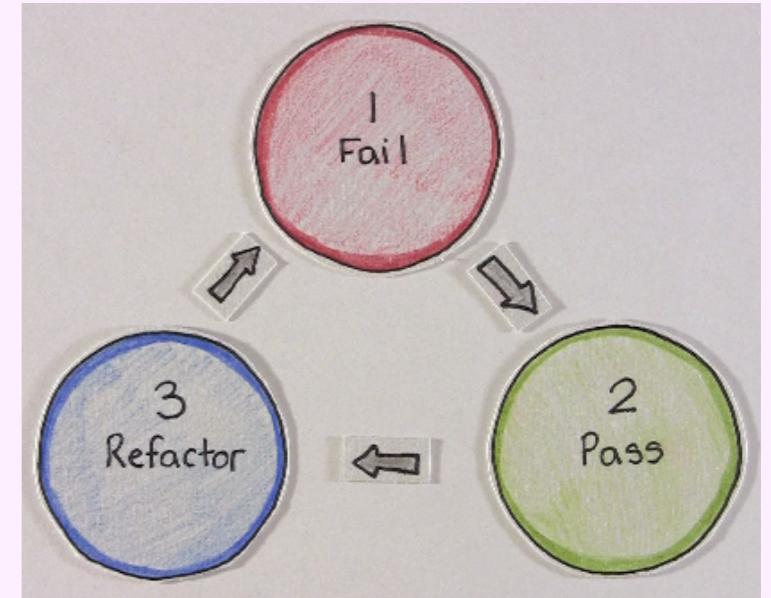
P

- Consiste en desarrollar aplicando estos tres pasos:

Paso 1. Escribir una prueba para el nuevo código y ver cómo falla

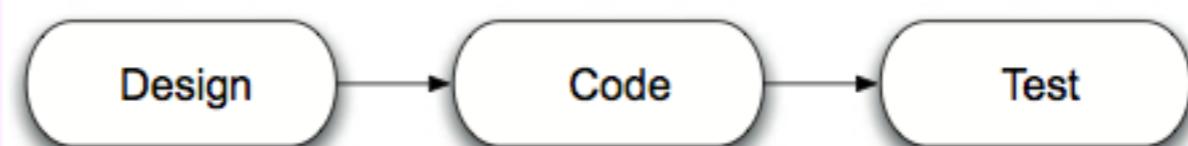
Paso 2. Implementar el nuevo código, haciendo “la solución más simple que pueda funcionar”

Paso 3. Comprobar que la prueba es exitosa y refactorizar el código



- Con TDD el **diseño de la aplicación** “evoluciona” a medida que vamos desarrollando la aplicación

→ Aproximación tradicional

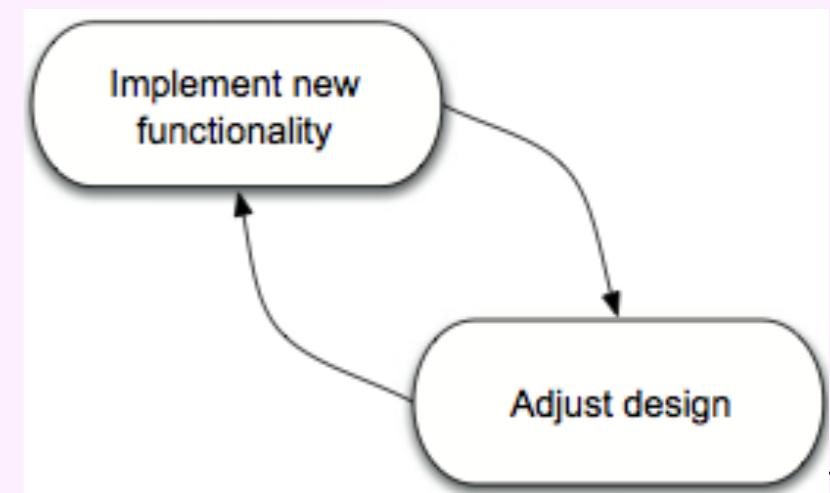


→ Aproximación TDD



- El diseño de la aplicación se realiza de forma incremental ajustando la estructura del código en pequeños incrementos a medida que se añade más comportamiento.

□ En **cualquier momento del desarrollo**, el código exhibe el mejor diseño que los desarrolladores pueden concebir para soportar la funcionalidad actual



P
Ver <http://xunitpatterns.com/Philosophy%20Of%20Test%20Automation.html>

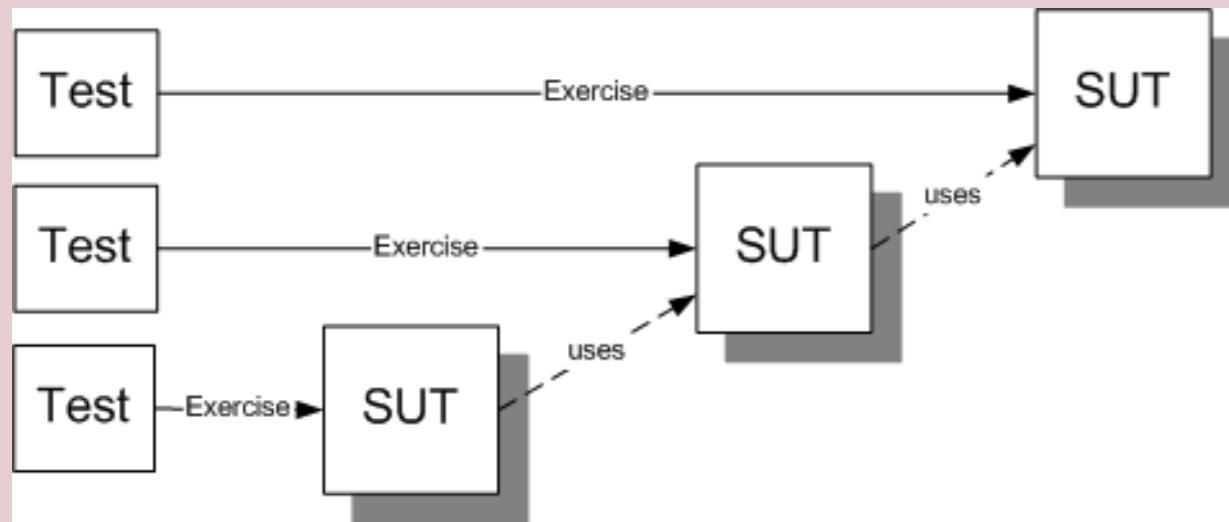
CLASSICAL VS MOCKIST TDD

S
Ver <http://martinfowler.com/articles/mocksArentStubs.html>

- La aproximación “clásica” de TDD consiste en utilizar objetos reales siempre que sea posible, y utilizar stubs (u otro “doble”) si es “complicado” utilizar el objeto real.

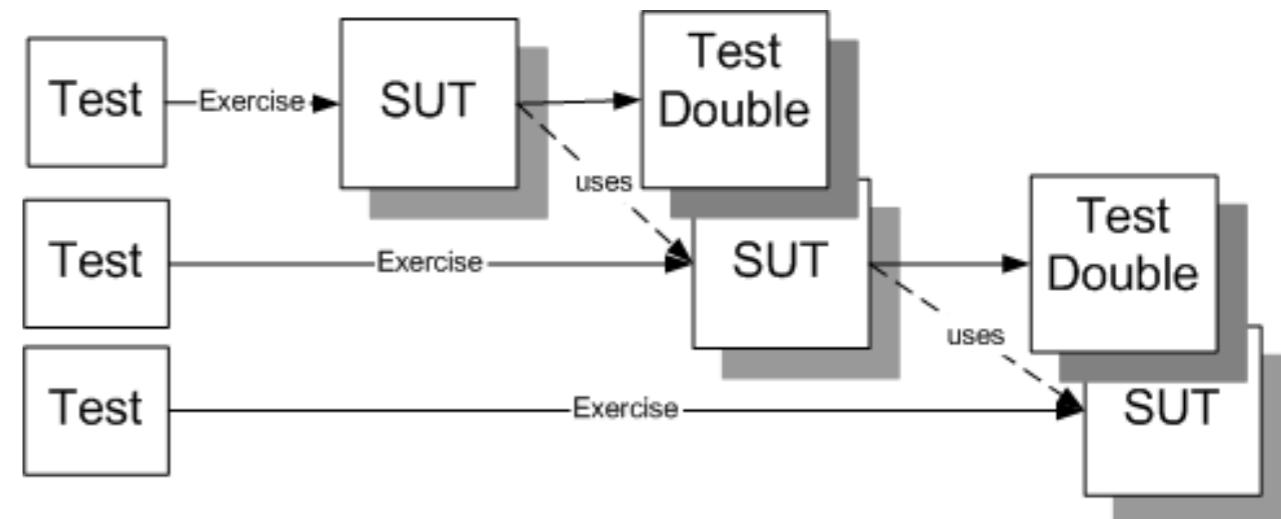
Por ejemplo, cuando estamos probando un método que como parte de su cometido debe enviar un correo

- Esta aproximación soporta un estilo de diseño **INSIDE-OUT**: comenzamos por los componentes de bajo nivel. No necesitamos stubs



minimizamos el problema de las dependencias

- La aproximación “mockist” utilizará siempre un mock para cualquier objeto con un comportamiento interesante
 - Como consecuencia de esta aproximación surge lo que se denomina BDD (Behaviour Driven Development). Fué originalmente desarrollado por Dan North como una técnica para ayudar al aprendizaje de TDD, centrando su atención en cómo TDD contribuye a generar el diseño: “El diseño del sistema evoluciona a través de iteraciones a medida que se escriben los tests”
 - La aproximación “mockist” soporta un estilo de diseño **OUTSIDE-IN**: comenzamos por los componentes de alto nivel. Utilizaremos mocks para sustituir los componentes de capas inferiores



- El diseño OUTSIDE-IN utiliza verificación basada en el comportamiento (no sólo es necesario especificar el estado inicial y final del objeto a probar, sino también la interacción con sus dependencias)

INTEGRACIONES CONTINUAS (CI)

- El término “integración continua” nace con el proceso de desarrollo XP, como una de sus doce prácticas fundamentales
- Las integraciones continuas (**CI**: Continuous Integration) consisten en **INTEGRAR** el código del proyecto de forma ininterrumpida (en ciclos cortos) en una máquina aparte de la de cada desarrollador, la cual debe estar funcionando 24/7
- Si bien la práctica de CI no requiere de una herramienta específica, es habitual utilizar un **Servidor de Integraciones Continuas** para automatizar todo el proceso
- Las integraciones continuas realizan **CONSTRUCCIONES PLANIFICADAS** del sistema
 - Ejecutan el proceso de construcción (a partir de comandos Maven, por ejemplo) tantas veces como queramos y con la frecuencia que deseemos, sin mover ni un dedo.
 - Es importante que se ejecuten a **intervalos regulares lo más cortos posible**. Por ejemplo, supongamos que integramos el proyecto cada hora. Cada 60 minutos sabremos si nuestra construcción funciona o no. Esto hace que la búsqueda de errores sea más fácil (sólo hemos de mirar en los cambios que han ocurrido durante dicho intervalo). Además, estos problemas serán fáciles de resolver, porque en una hora no hemos tenido oportunidad de realizar grandes cambios que se habrían convertido en grandes problemas.

COMPONENTES Y FUNCIONAMIENTO DE UN SISTEMA DE CI

Imagen extraída de: "Continuous integration". Paul M. Duval. Addison Wesley (Capítulo 1)

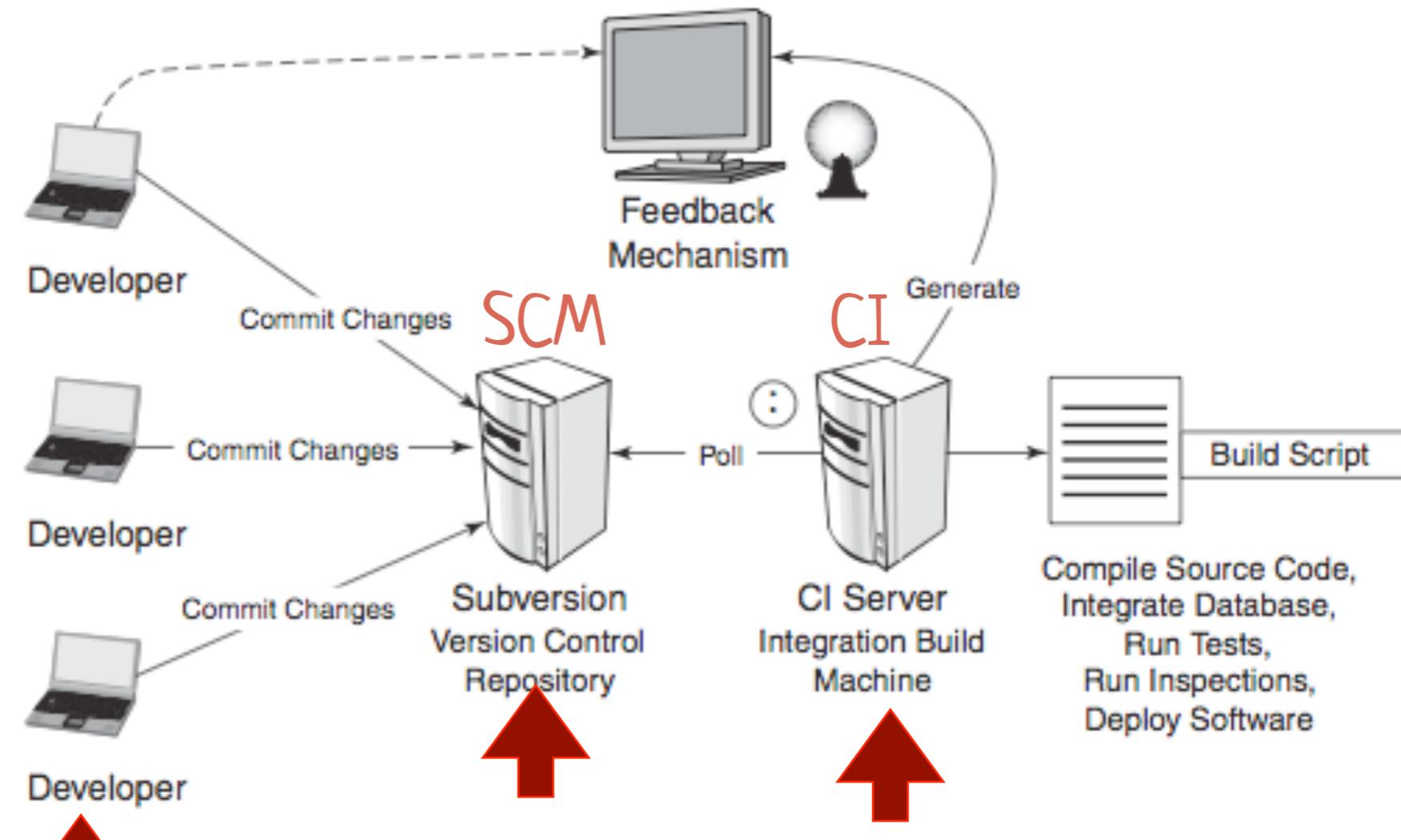
Funcionamiento.

- P 1. Un desarrollador "sube" el código en el que ha estado trabajando al repositorio **SCM** (después de hacer un build local), y sólo si las pruebas unitarias han "pasado".

Mientras tanto el servidor CI consulta el SCM para detectar cambios

2. El **CI** recupera la última versión del SCM y ejecuta un script de construcción del proyecto (**construcción planificada**)

3. El CI genera un feedback. En el momento en el que se "rompe" el sistema, hay que reparar el error



Sesión 11: Planificación de pruebas

La integración continua no evita los fallos, pero reduce drásticamente el esfuerzo de encontrar los errores y repararlos. Se agiliza el proceso de desarrollo mediante la automatización de las construcciones planificadas del sistema. La tarea de construir el sistema cada poco tiempo interfiere en el trabajo de los programadores (la construcción puede llevar desde unos pocos minutos a unas pocas horas). Un servidor de CI no tiene otra cosa mejor que hacer que construir el sistema, probarlo e informar de los resultados

BDD: BEHAVIOR DRIVEN DEVELOPMENT

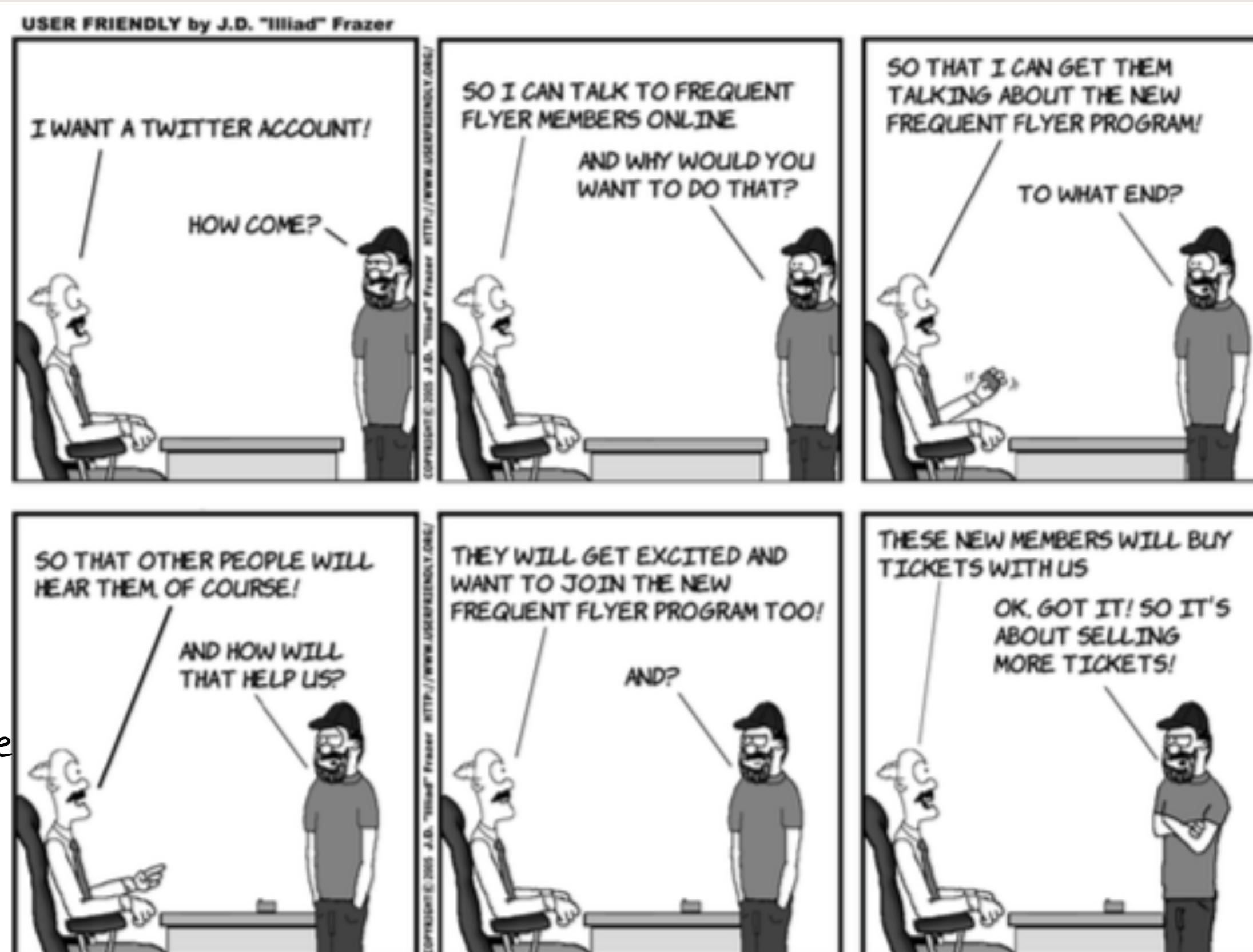
Hunting out value!!!!

Cualquier sistema software proporciona un **VALOR** de NEGOCIO.

Las expectativas del cliente dependen de dicho valor

BDD nos permite centrarnos en el valor de negocio de nuestra aplicación

S
Es muy importante entender por qué estamos llevando a cabo un proyecto (tener claros los objetivos del mismo)



BDD: HUNTING OUT VALUE

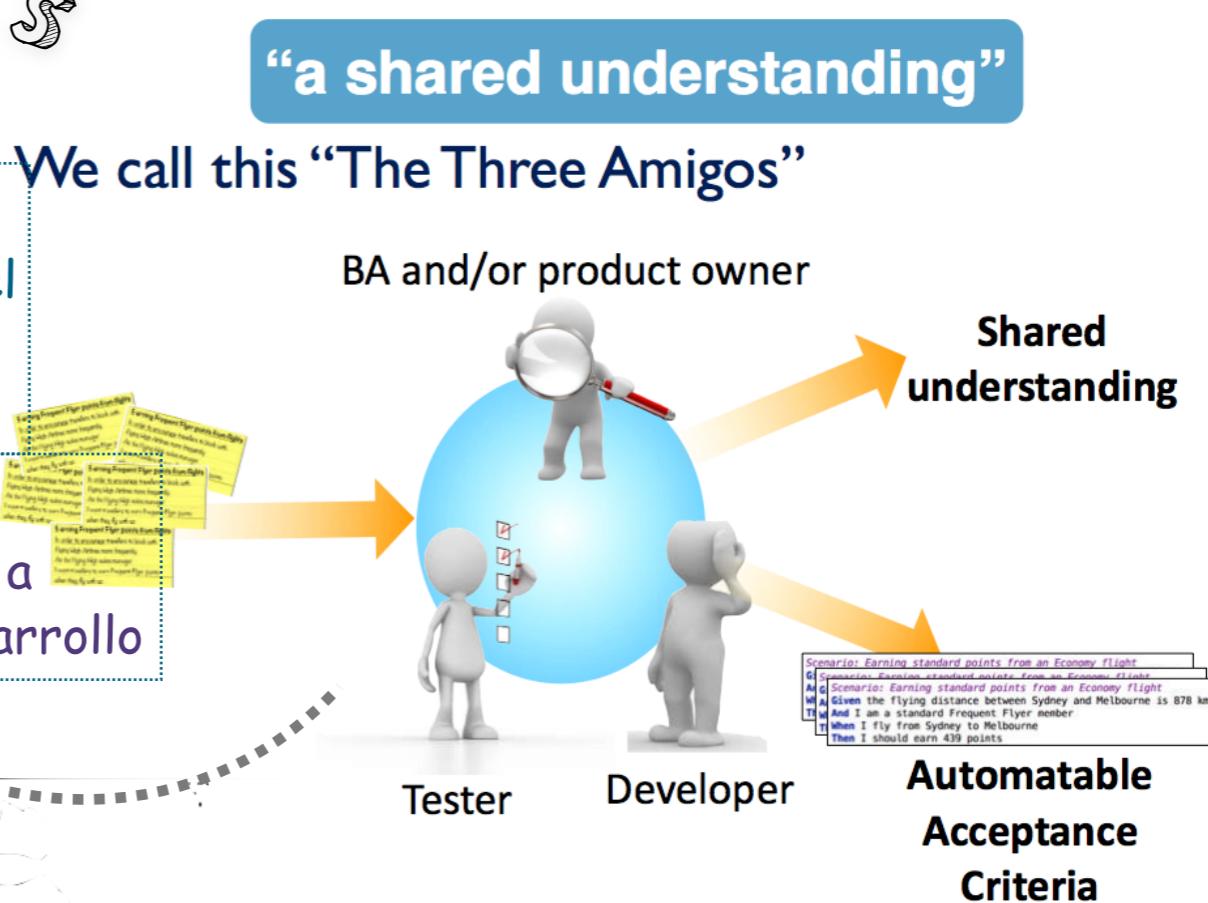
- Si preguntamos a los usuarios "qué quieren", generalmente obtendremos por respuesta un conjunto de requerimientos detallados sobre cómo imaginan la solución,
 - es decir, los usuarios no nos dicen lo que necesitan, diseñan una solución por nosotros
- Debemos centrarnos en las "features" (funcionalidades) que proporcionan un valor de negocio para el usuario,
 - es decir, funcionalidades que nos ayudan a conseguir los OBJETIVOS del proyecto
- Ejemplos de objetivos (goals):
 - Incrementar en un 10% el número de clientes proporcionando una forma sencilla de que ellos mismos gestionen sus cuentas
 - Incrementar las ventas de libros en un 5% animando a los usuarios a que compren en nuestra tienda

Behavior Driven Development, or BDD, is a set of software engineering practices designed to help teams build and deliver more valuable, higher quality software faster.

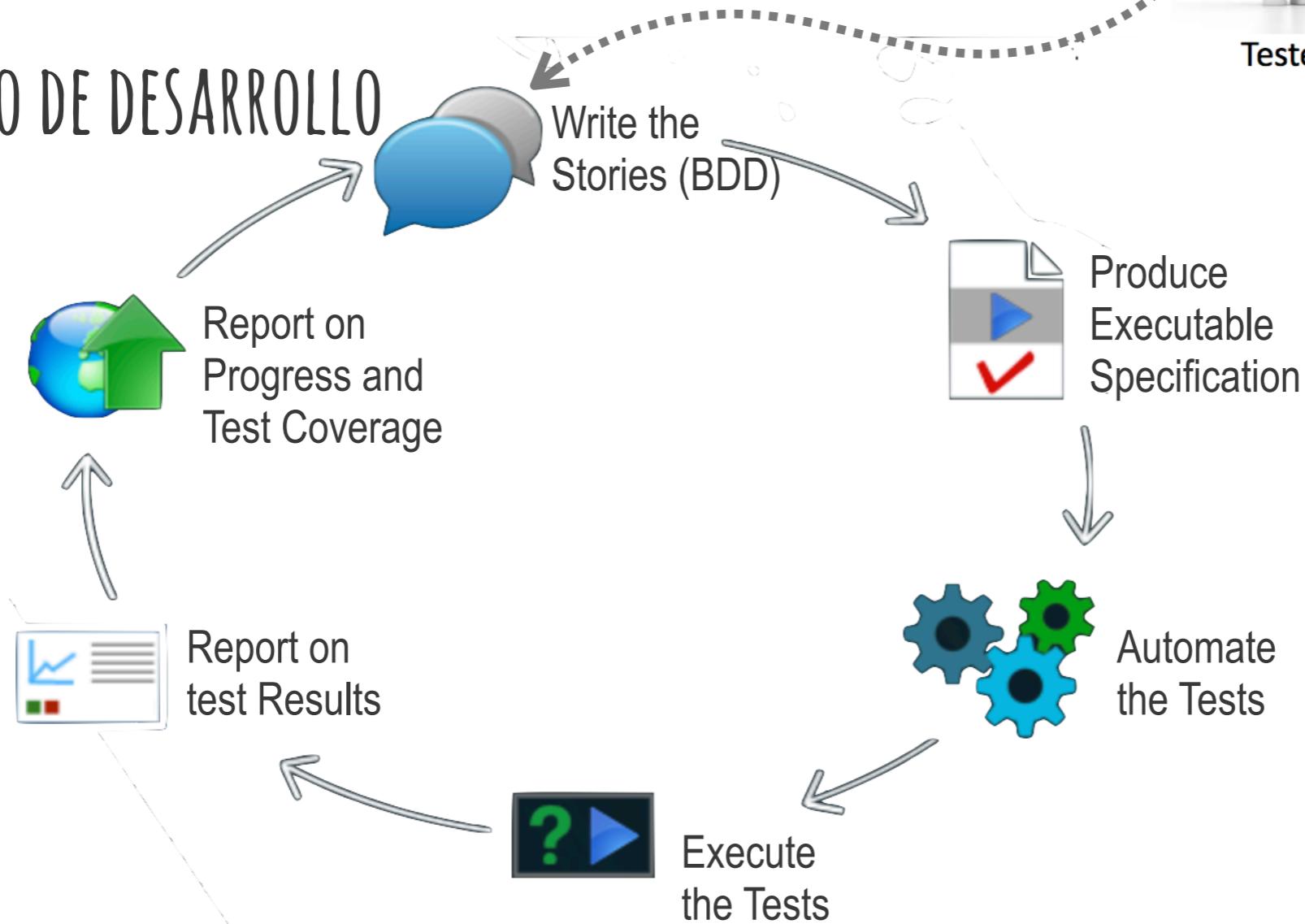
BDD Y LOS BENEFICIOS DE LA COLABORACIÓN

P Comenzaremos aclarando exactamente cómo esperamos que el sistema proporcione un valor de negocio, y focalizaremos el desarrollo en aquellas características que permitan maximizar la "rentabilidad" de nuestra aplicación

P La colaboración permite identificar y compartir claramente los objetivos que "importan" (dan valor a nuestra aplicación), reduciendo el esfuerzo de desarrollo



BDD: CICLO DE DESARROLLO



BDD utiliza conversaciones basadas en ejemplos, y expresadas de forma que sean fácilmente automatizables, reduciendo así la pérdida de información y los malentendidos

BDD NO IMPLEMENTA TESTS UNITARIOS, IMPLEMENTA ESPECIFICACIONES DE BAJO NIVEL

P

- Simplemente cambiando la forma de escribir los tests, centrándonos en lo que "debería" hacer el sistema, podemos escribir tests más "focalizados" en los objetivos concretos de nuestra aplicación, convirtiéndolos así en "especificaciones". Por ejemplo, si pensamos en una aplicación bancaria:

```
public class BankAccountTest {  
    @Test  
    public void testTransfer() {...}  
    @Test  
    public void testDeposit() {...}  
}
```

Test unitario "tradicional" (sin BDD)

vs.

Especificación de bajo nivel

Ejemplo de herramienta
que soporta BDD:
Serenity

```
public class WhenTransferringInternationalFunds {  
    @Test  
    public void should_transfer_funds_to_destination_account() {...}  
    @Test  
    public void should_deduct_fees_as_a_separateTransation() {...}  
    ...  
}
```

Los test se centran en el comportamiento de la aplicación, constituyendo una forma de expresar y verificar dicho comportamiento

Escenario

```
Given a customer has a current account  
When the customer transfers funds from this account to an overseas account  
Then the funds should be deposited in the overseas account  
And the transaction fee should be deducted from the current account
```

PS REFERENCIAS BIBLIOGRÁFICAS

- PS ○ Software Testing: An ISTQB-ISEB Foundation Guide. Brian Hambling. British Computer Society; 2nd New edition. 2010
 - Capítulo 1
- Agile estimating and planning. Mike Cohn. Prentice Hall. 2006
 - Capítulo 3
- The art of agile development. James Shore. O'Reilly. 2008
 - Capítulo 3
- The Scrum primer. An introduction to project manager with scrum. Pete Deemer. 2007
 - <http://www.scrumprimer.com/>
- Continuous integration. Martin Fowler. 2006
 - <http://www.martinfowler.com/articles/continuousIntegration.html>
- BDD in action. Behavior-Driven Development for the whole software lifecycle. John Ferguson Smart. Manning. 2015
 - Capítulo 1