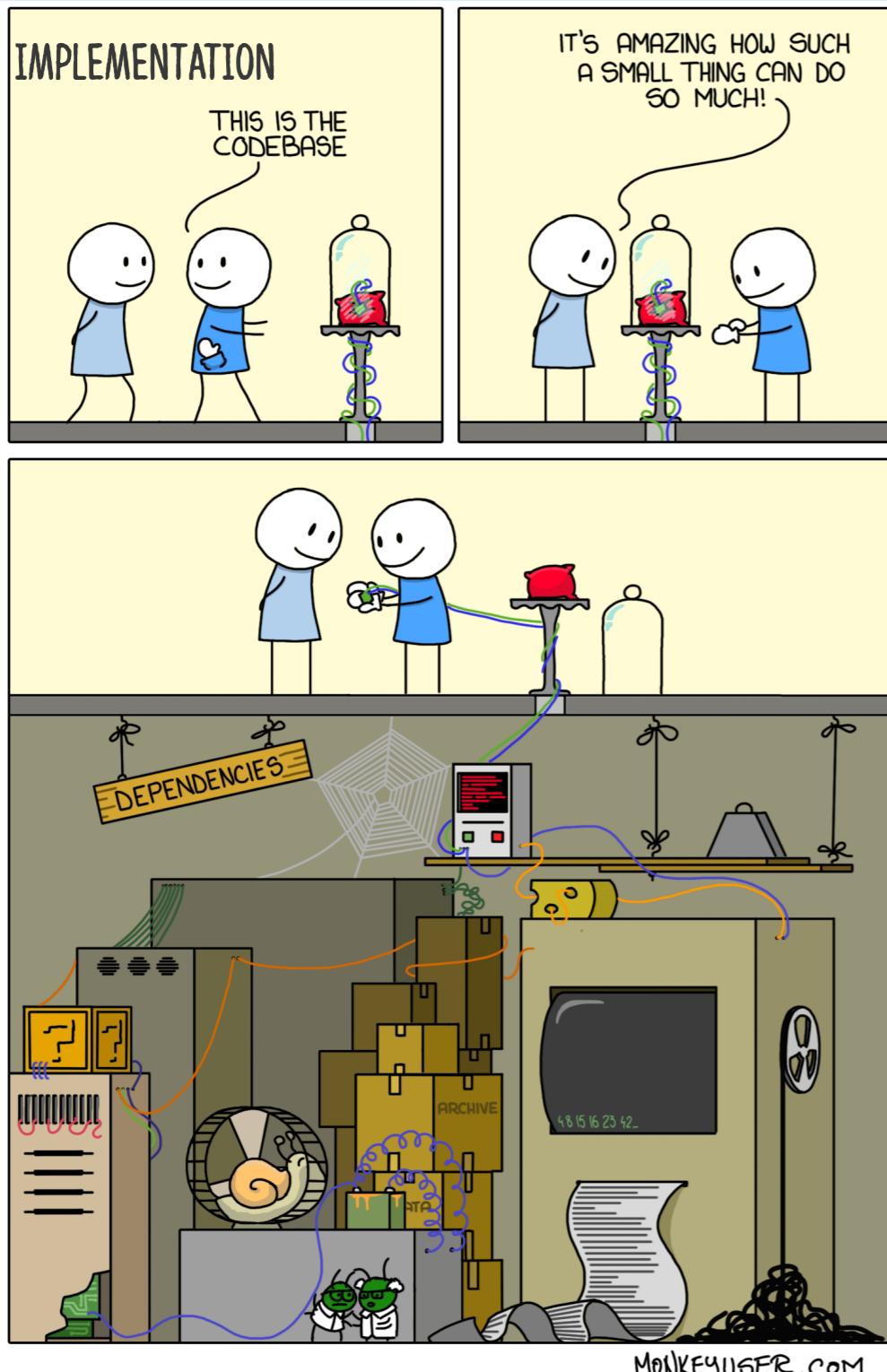


PPSS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2024-25

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

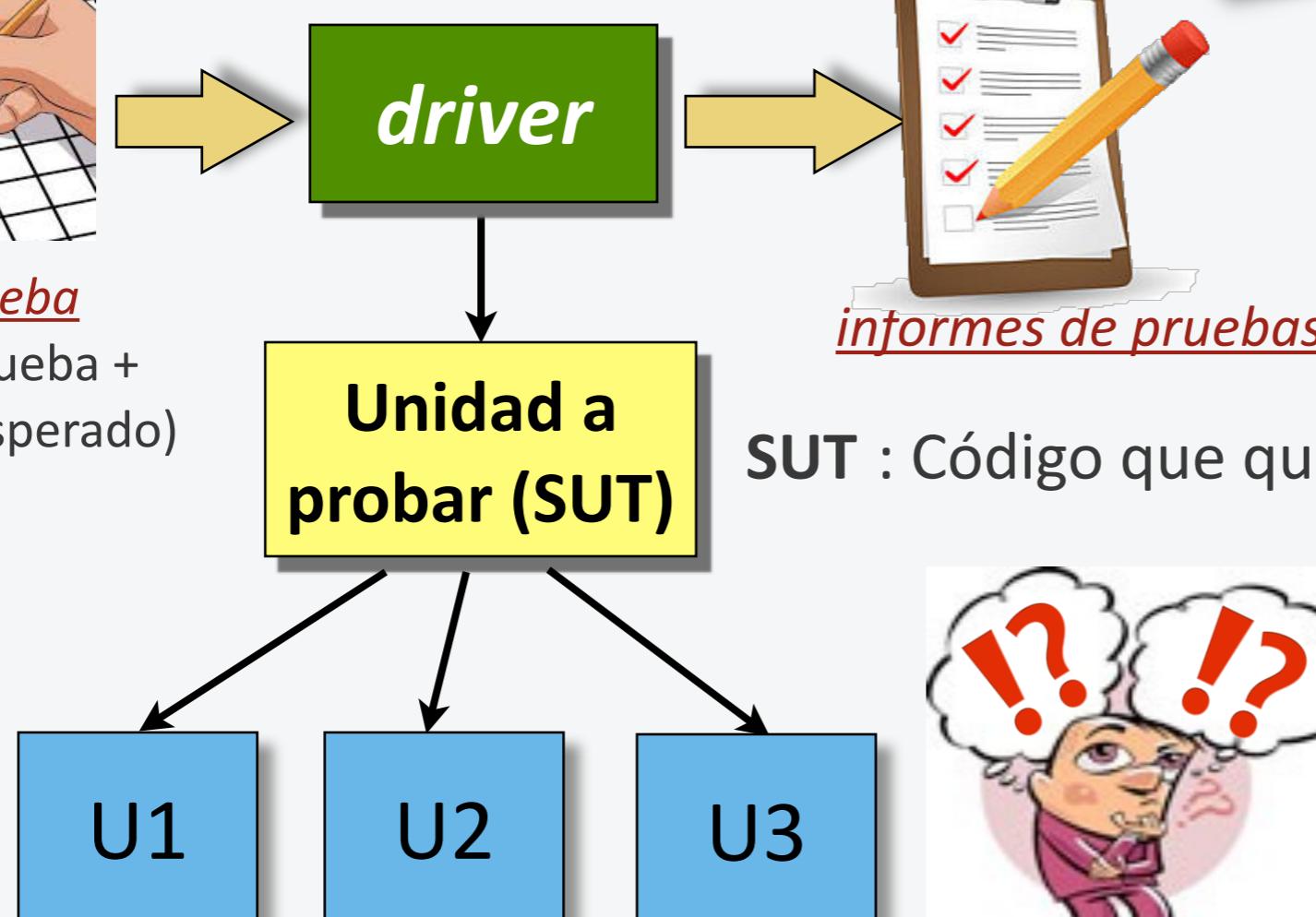
P

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



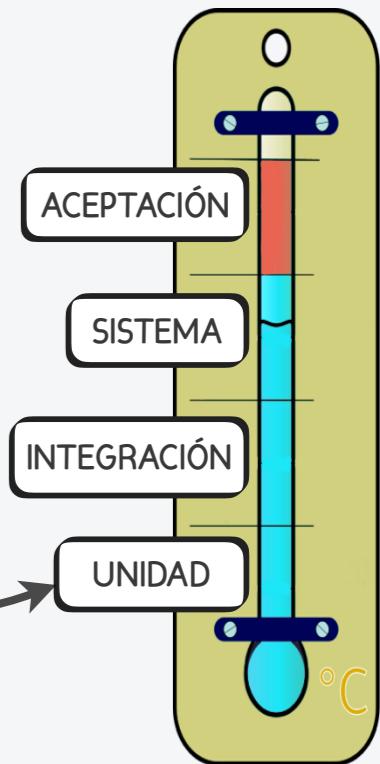
casos de prueba

(datos de prueba + resultado esperado)



SUT : Código que queremos probar.

Objetivo: encontrar DEFECTOS en el código de las UNIDADES probadas



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.

Ejemplo: queremos realizar una prueba **unitaria** sobre 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. }
```

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

↑
SUT

NO estamos interesados en ejecutar el código del que depende nuestro 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 elemento con quién interactúa nuestro código a probar (invocación a otra unidad) y sobre el que no tenemos ningún control



DOC = **Depended-On Component**

DOBLES
(sustituyen a las dependencias externas durante las pruebas)



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

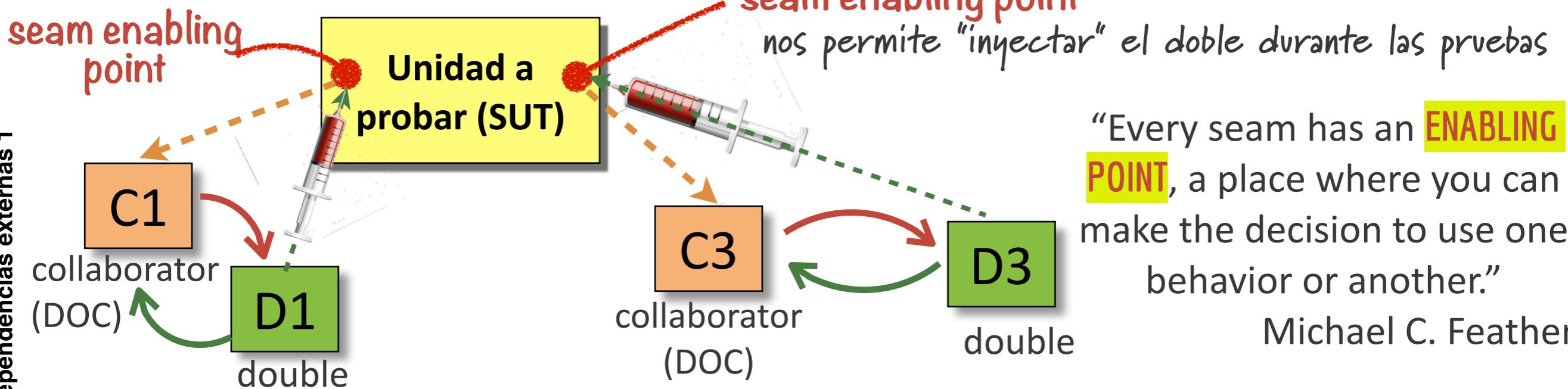
<http://www.informit.com/articles/article.aspx?p=359417&seqNum=3>

Para poder conseguir un SEAM en nuestro código PUEDE que necesitemos REFACTORIZAR nuestra 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>



"Every seam has an **ENABLING POINT**, a place where you can make the decision to use one behavior or another."

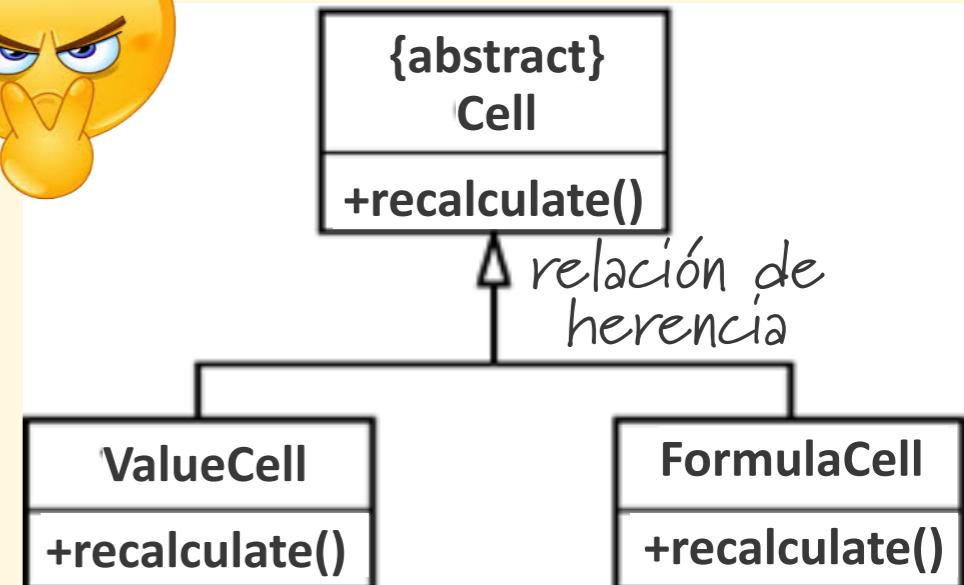
Michael C. Feathers

DOBLES: reemplazos controlados de los colaboradores del sistema utilizados durante las pruebas para aislar el código de SUT

CÓMO IDENTIFICAR UN SEAM



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



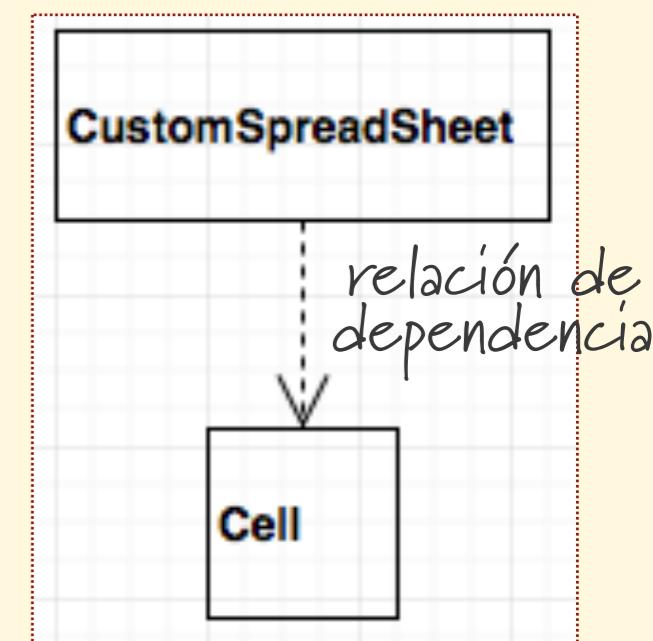
SUT

```

public class CustomSpreadsheet {
    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

```

public class CustomSpreadsheet {
    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注入emos. Podemos injectar cualquier **subtipo** de Cell

SEAM: MÁS EJEMPLOS

Nuestra SUT será **TESTABLE** si tiene 1 seam para cada dependencia externa!!

- 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 {
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        recalculate(cell);
        ...
    }
    protected void recalculate(Cell cell) {
        ...
    }
}
```

SÍ es un seam

CÓDIGO TESTABLE!!

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

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

Sesión 5: Dependencias externas 1

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

Inyectaremos el doble durante las pruebas

CÓDIGO TESTABLE!!

```
public class TestableCustomSpreadsheet
    extends CustomSpreadsheet {
    @Override
    public Cell getCell() {
        ...
        > return new DoubleCell();
    }
}
```

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

PASOS A SEGUIR PARA AUTOMATIZAR LAS PRUEBAS

Identificar las dependencias externas de nuestro SUT



C1 C2 C3

Colaboradores (DOCs)

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

D1 D2 D3

Dobles

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** nuestra SUT (y/o la clase a la que pertenece) para poder realizar dichos reemplazos controlados durante las pruebas

SUT testable

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 nuestra SUT (implementaremos el driver como ya hemos visto en las sesiones anteriores)

driver

verificación basada en el **COMPORTAMIENTO**:

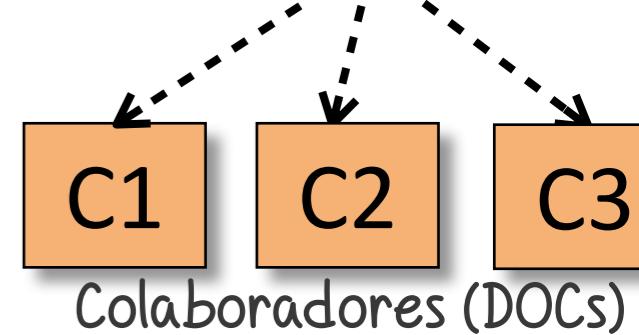
nos interesa, además, verificar que las interacciones entre nuestro SUT y las dependencias externas se realizan correctamente

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 1:

```

1. public class GestorPedidos {
2.
3.   public Factura generarFactura(Cliente cli) throws FacturaException {
4.     Factura factura;
5.     Buscador buscarDatos = new Buscador();
6.
7.     int numElems = buscarDatos.elemPendientes(cli);
8.     if (numElems>0) {
9.       //código para generar la factura
10.      factura = ...;
11.    } else {
12.      throw new FacturaException("No hay nada pendiente de facturar");
13.    }
14.    return factura;
15.  }
16. }
```

SUT

Sesión 5: Dependencias externas 1

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



NUESTRA SUT DEBE SER TESTABLE

Y si no lo es, lo REFACTORIZAREMOS



Para que sea testable, debe contener un SEAM

P

P

- Necesitamos poder INVOCAR al doble de nuestra dependencia externa durante las pruebas SIN cambiar el código de nuestra SUT.

- Esto no será posible si no tenemos un **seam** para CADA dependencia externa, que nos permita **inyectar** nuestro doble durante las pruebas.

- Formas de INYECTAR el doble en nuestra SUT ([DEPENDENCY INJECTION](#)):

- (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 **FACTORÍA LOCAL** de la clase que contiene nuestra SUT, o una **CLASE FACTORIA**

- Si nuestra SUT NO es testable, tendremos que **REFACTORIZAR** el código de nuestra SUT para poder inyectar 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 realiza 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 **FACTORÍA 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 FACTORIA** 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 nuestra SUT

Como NO es testable,
tendremos que
REFACTORIZAR

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

Sustituimos la versión
original de nuestra
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!!!

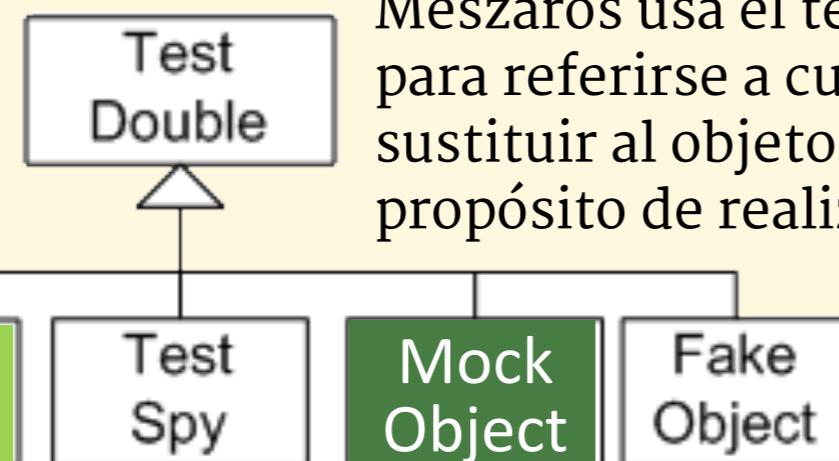


IMPLEMENTAMOS EL DOBLE



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

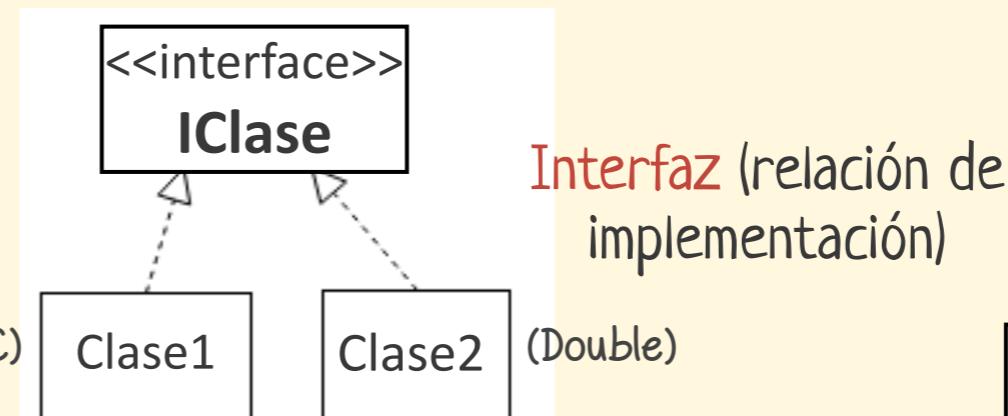
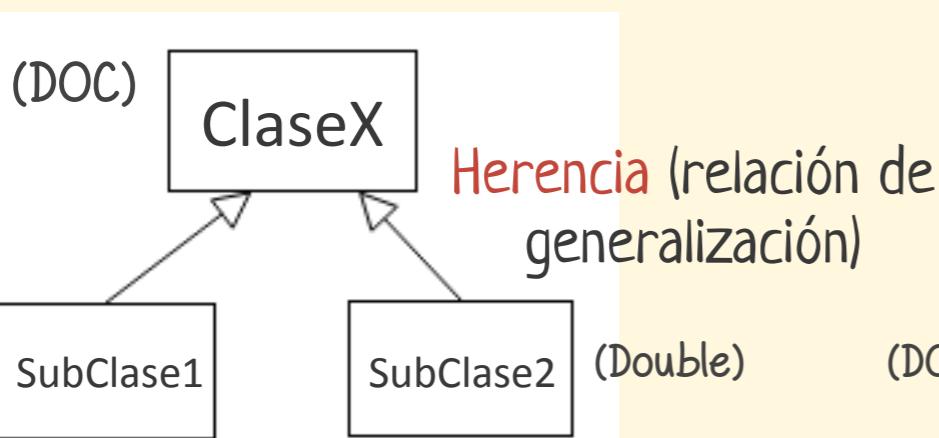
Nuestra SUT invocará al doble SOLO durante las pruebas. En producción, la SUT invocará al DOC



Nuestra dependencia externa siempre será un **método Java**. Por lo tanto nuestro doble será una implementación **ALTERNATIVA** del MISMO método Java.

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)



```
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 define metodoB()
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" al DOC o "implementará" la misma interfaz del DOC.

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

```
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)

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 nuestra SUT será idéntico en ambos casos!!!



Inyectaremos 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;
}
```

seam

SUT



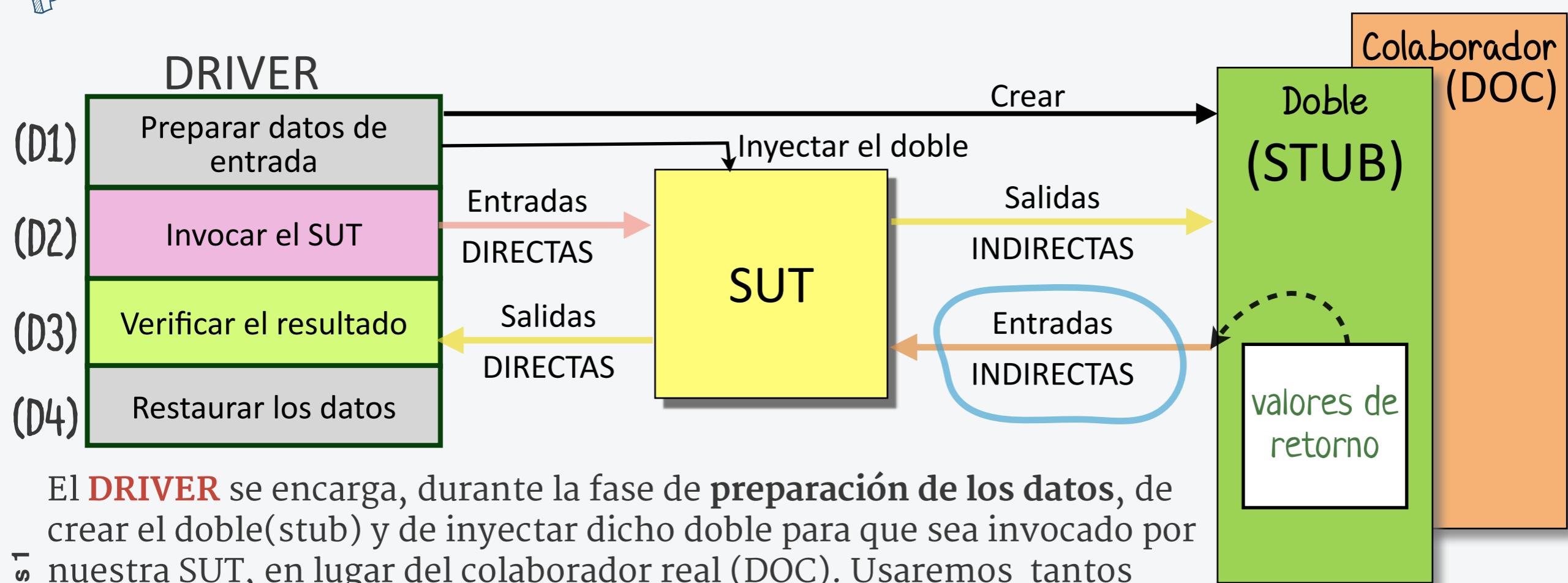
La implementación del STUB
debe ser GENÉRICA. Debe servir para todos los casos de prueba de nuestro SUT (sólo UN doble por tabla) !!!

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 nuestra 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 unidad a probar (SUT). Verificamos que el estado resultante de nuestro SUT es el esperado (**Verificación basada en el estado**)



IMPLEMENTACIÓN DEL DRIVER



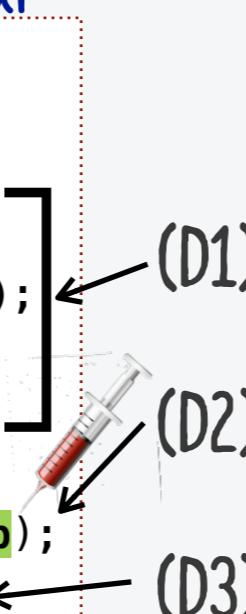
Ejemplo

- 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() {
        Cliente cli = new Cliente(...);
        GestorPedidos sut = new GestorPedidos();
        BuscadorStub buscarStub = new BuscadorStub();
        buscarStub.setResult(10);
        Factura expectedResult = new Factura(...);
        Factura realResult =
            assertDoesNotThrow(
                ()->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() {
        Cliente cli = new Cliente(...);
        GestorPedidos sut = new GestorPedidos();
        Buscador buscar = new Buscador();
        Factura expectedResult = new Factura(...);
        Factura realResult =
            assertDoesNotThrow(
                ()->sut.generarFactura(cli,buscar));
        assertEquals(expectedResult, realResult);
    }
}
```

NO tenemos control sobre las entradas indirectas!

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

Los dos tests ejecutan el MISMO CÓDIGO!!!

SUT



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



EJEMPLO 2

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

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

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

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

SUT NO TESTABLE!!!

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

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

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

- Tenemos que refactorizar nuestra unidad para poder injectar nuestro doble a la hora de hacer las pruebas sin cambiar el código de nuestra SUT. Podemos usar cualquiera de las opciones indicadas. En este caso, elegiremos la opción (4) usando un método de **factoría LOCAL**
 - El método de factoría local será una nueva dependencia externa, pero que pertenece a la misma clase que nuestra 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() {
        Cliente cli = new Cliente(...);
        BuscadorSTUB stub = new BuscadorSTUB(10);
        GestorPedidosTestable sut = new
            GestorPedidosTestable();
        sut.setBuscador(stub);
        Factura expectedResult = new Factura(...);
        Factura realResult = assertDoesNotThrow(
            ()->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 **CONTIENE NUESTRA SUT**

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 Factoria {
    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() {  
        Cliente cli = new Cliente(...);  
        GestorPedidos sut = new GestorPedidos();  
        BuscadorStub buscadorStub = new BuscadorStub();  
        buscadorStub.setResult(10);  
        Factoria.setServicio(buscadorStub);  
        Factura expResult = new Factura(...);  
        Factura result = assertDoesNotThrow(  
            ()->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() {  
        Cliente cli = new Cliente(...);  
        GestorPedidos sut = new GestorPedidos();  
        Factura expResult = new Factura(...);  
        Factura result = assertDoesNotThrow(  
            ()->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 nuestra SUT sin ningún control de su dependencia externa

EJEMPLO 4

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

SUT TESTABLE!!!

SUT

DOC

(Pasos 1 y 2)

EJEMPLO 4: DEFINICIÓN DE CLASES UTILIZADAS POR NUESTRA SUT

P

Clases utilizadas por OrderProcessor:

PricingService, Customer, Product y Order:

```
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  
    ...  
}
```

**NO IMPLEMENTAMOS DOBLES
PARA ESTAS DEPENDENCIAS!!**

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

Don't
FORGET!



EJEMPLO 4: IMPLEMENTAMOS EL DOBLE (Paso 3)

Necesitamos implementar un doble para getDiscountPercentaje(), de tipo PricingService.

Nuestro código en producción ejecutará al código real de dicho método, mientras que nuestros tests invocarán al doble durante la ejecución de los tests unitarios

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

código real, utilizado en la 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

EJEMPLO 4: TEST UNITARIO

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

Implementmos 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(); ← SUT
        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

P



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

P

- 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();  
        processor.setPricingService(new PricingService());  
  
        Order order = new Order(customer, product);  
        processor.process(order); ← Ejecutamos nuestro SUT.  
        assertAll -> ("Error en pedido",  
                      () -> assertEquals(expectedBalance, order.getBalance(), 0.001f),  
                      () -> assertEquals("Pedro Gomez", order.getCustomer().getName()),  
                      () -> assertEquals("TDD in Action", order.getProduct().getName()),  
                      () -> assertEquals(listPrice, order.getProduct().getPrice(), 0.001f));  
    }  
}
```

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

Y AHORA VAMOS AL LABORATORIO...

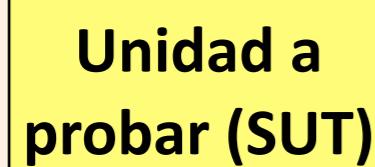
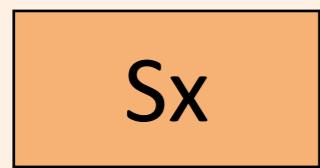
P

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

P

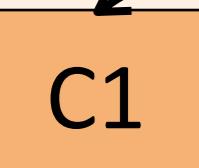
src/main/java

Entorno de producción



Dependencias externas
(DOCs)

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



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

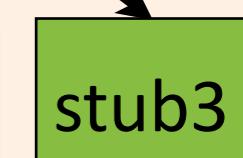
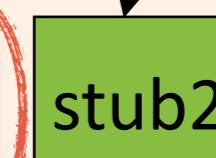
Entorno de producción

src/test/java

Tests Unitarios



Unidad a probar (SUT)



Dobles

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

P REFERENCIAS BIBLIOGRÁFICAS

- P
 - The art of unit testing: with examples in .NET. Roy Osheroove. Manning, 2009.
 - Capítulo 3. Using stubs to break dependencies.
 - * <https://www.manning.com/books/the-art-of-unit-testing>
 - 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