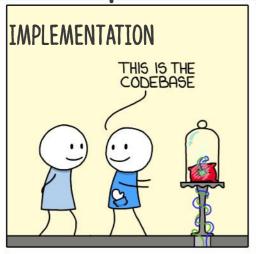
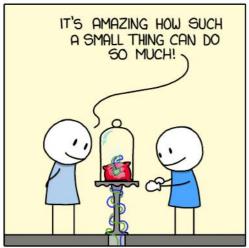
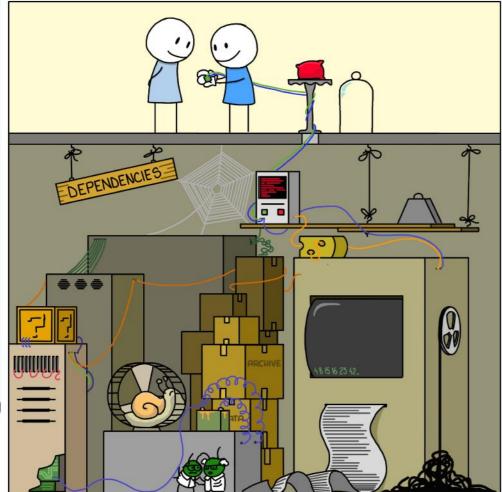
PISS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2019-20

Sesión S05: Dependencias externas







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

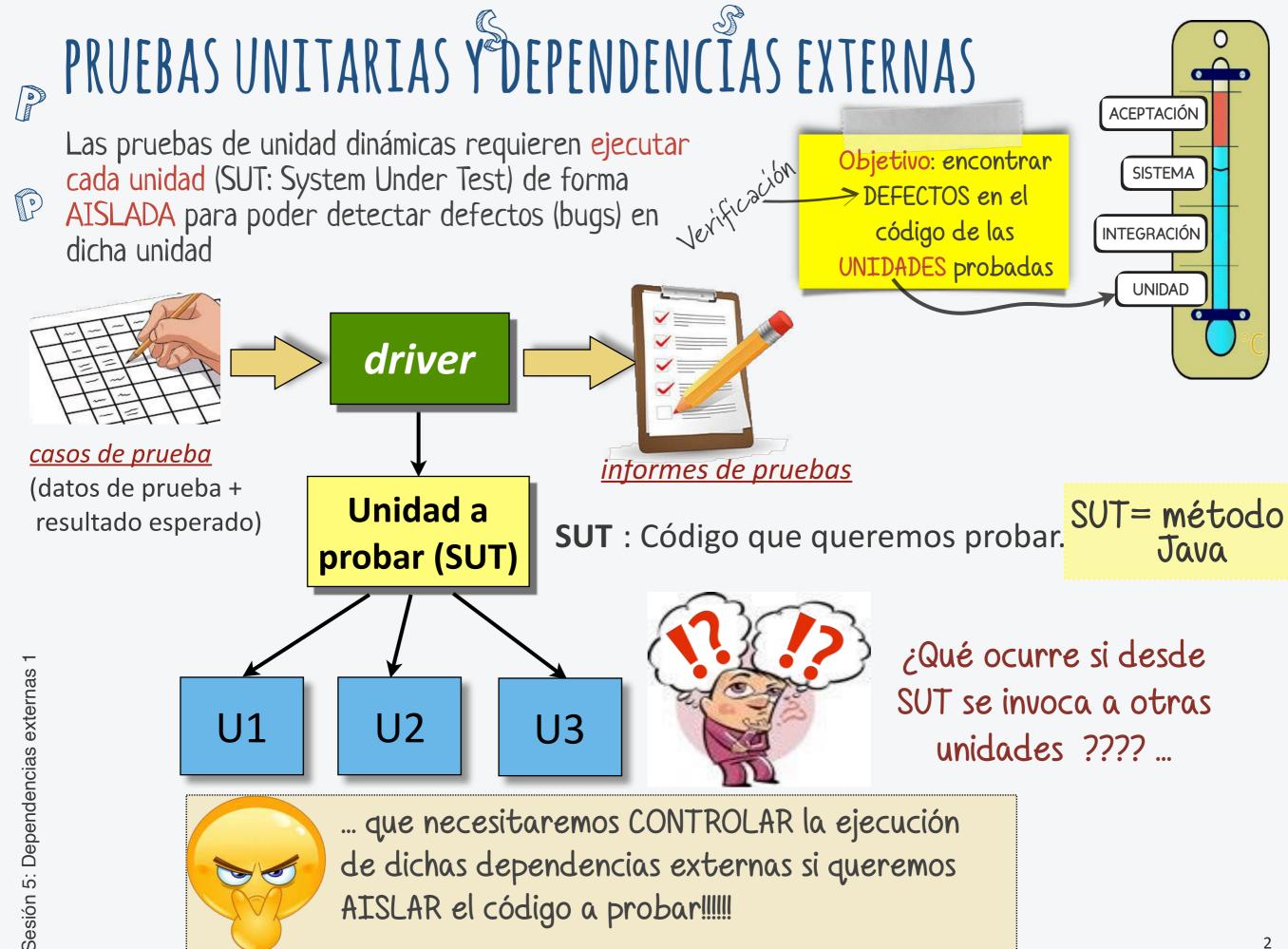
Conceptos de código testable y "seam"

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

- Paso 1: Identificación de dependencias externas
- Paso 2: Refactorización de la unidad (sólo si es necesario) para conseguir inyectar 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...

MONKEYUSER COM



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()

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

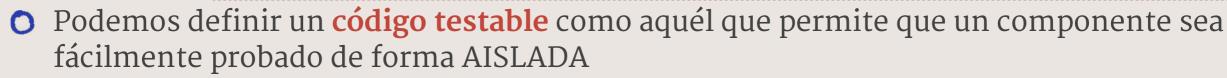
La variable "ok" es una entrada INDIRECTA de la unidad (SUT)

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!!!**

P

CÓDIGO TESTABLE Y CONTROL DE DEPENDENCIAS

(http://www.loosecouplings.com/2011/01/testability-working-definition.html)



- □ 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!!!!



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

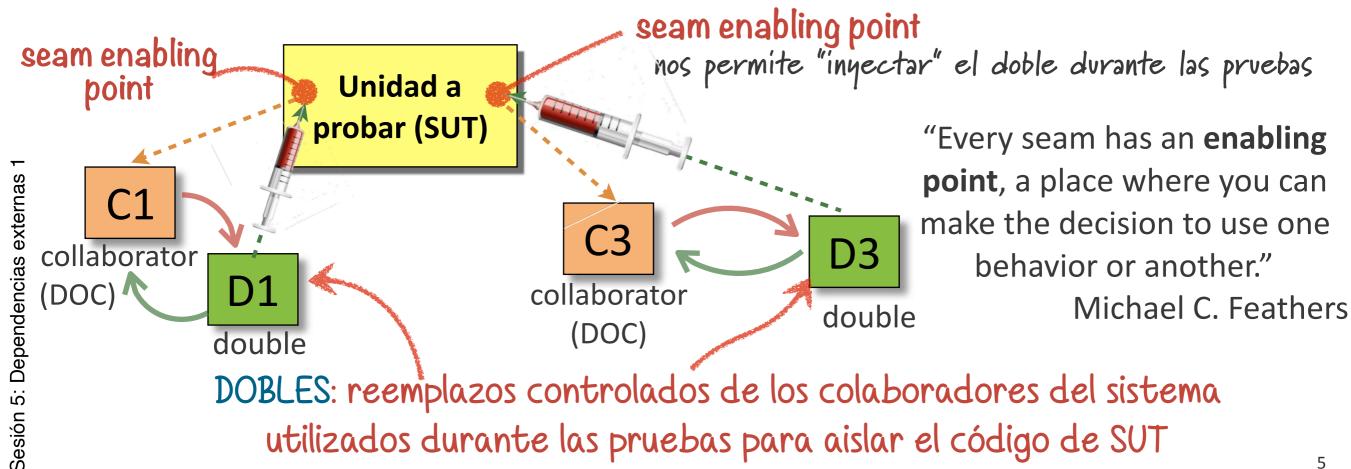
Michael Feathers

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

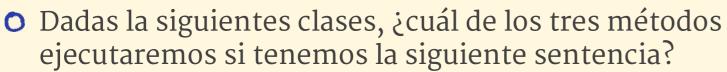
http://agile.dzone.com/articles/what-refactoring-and-what-it-0



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

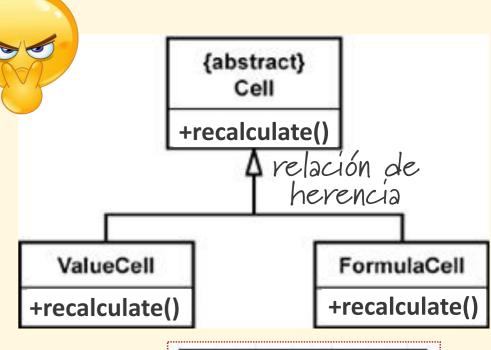
DOBLES: reemplazos controlados de los colaboradores del sistema utilizados durante las pruebas para aislar el código de SUT

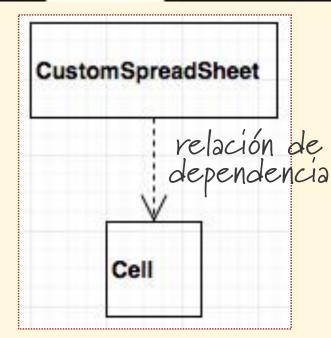
CÓMO IDENTIFICAR UN SEAM



- myCell.recalculate();
- ☐ Si no conocemos el tipo del objeto "myCell", <u>no podemos</u> 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");
                                     NO es un seam, ya que no
   myCell.recalculate();
                                   podemos cambiar el método al
                                  que se invocará sin modificar el
            CODIGO NO TESTABLE!!
                                             código
```





- seam enabling point public class CustomSpreadsheet extends(Spreadsheet){

public Spreadsheet buildMartSheet(Cell cell) { cell.recalculate(); CODIGO TESTABLE!!

Sesión 5: Dependencias externas 1

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

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



SEAM: MÁS EJEMPLOS

myCell.recalculate();

O 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!!!
                                                                Usaremos esta clase durante
  public class CustomSpreadsheet extends Spreadsheet {
                                                                las pruebas e invocaremos al
    public Spreadsheet buildMartSheet(Cell cell) {
                                                                  doble el lugar de al DOC
       recalculate(cell);
                                      SÍ es un seam
                                                        public class TestingCustomSpreadsheet
                                                        extends CustomSpreadsheet {
                                                           @Override
    -protected void recalculate(Cell cell)
                                                           protected void recalculate(Cell cell) {
                CÓDIGO TESTABLE!!
                                                            public class TestingCustomSpreadsheet
public class CustomSpreadsheet extends Spreadsheet {
                                                            extends CustomSpreadsheet {
                                                               @Override
   public Cell getCell() {
                            Inyectaremos el doble
                                                                public Cell getCell() {
       return new Cell();
                             durante las pruebas
                                                                 return new DoubleCell();
   public Spreadsheet buildMartSheet() {
                                                          pipublic class DobuleCell extends Cell {
       Cell myCell = getCell();
                                                                @Override
```

public recalculate() {



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

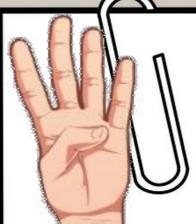
Puede que necesitemos REFACTORIZAR nuestro SUT (y/o la clase a la que pertenece) para poder realizar dichos reemplazos controlados durante las pruebas



pruebas

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

Doble



Implementar los DRIVERS correspondientes. Para ello podemos hacer una:

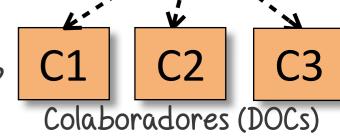
verificación basada en el **ESTADO**:

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

DEPENDENCIAS EXTERNAS Cuántas y qué dependencias externas tiene nuestra SUT??





- O 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 no puede controlar lo que dicha dependencia devuelve a nuestro código a probar ni cómo se comporta
 - Utilizaremos dobles para asegurarnos de que probamos nuestra unidad de forma AISLADA

O Ejemplo:

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

generarFactura depende de... Buscador.elemPendientes

dependencia externa

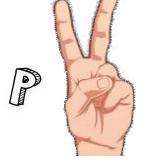


Sesión 5: Dependencias externas

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 Y SI NO 10 es, 10 KEFAIL Para que sea testable, debe contener un SEAM

- O 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.
- O 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)
- O Nuestra SUT será **TESTABLE** si podemos "inyectar" dicho doble en nuestra SUT durante las pruebas de alguna de las siguientes formas:
 - (1) como un <u>parámetro</u> de nuestra SUT
 - (2) a través del <u>constructor</u> de la clase que contiene nuestra SUT
 - (3) a través de un <u>método setter</u> de la clase que contiene nuestra SUT
 - (4) a través de un método de <u>factoría local</u> de la clase que contiene nuestra SUT, o una <u>clase factoria</u>
- O Si nuestra SUT NO es testable, entonces tendremos que REFACTORIZAR el código de nuestra SUT para que podamos 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 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 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 factoría implica añadir código en src/main/java que puede ser innecesario en producción.

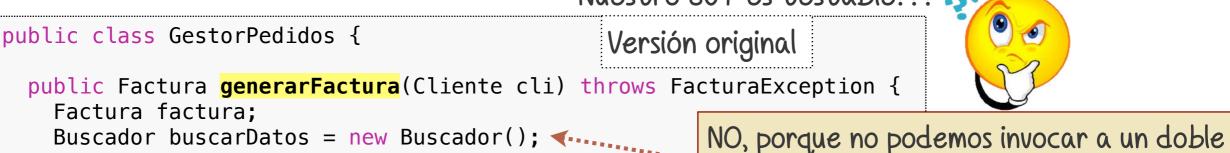


NUESTRO SUT DEBE SER TESTABLE

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

Refactorizaremos para poder inyectar el doble durante las pruebas

Nuestro SUT es testable???



```
de elemPendientes() SIN cambiar el
int numElems = buscarDatos.elemPendientes(cli);
                                                            código de nuestro SUT
if (numElems>0) {
  //código para generar la factura
                                                                      Como NO es testable,
                                         SUT
 factura = ...;
} else {
```

throw new FacturaException("No hay nada pendiente de facturar");

return factura;

Si, por ejemplo, decidimos usar la opción (1), nuestra SUT quedaría así:

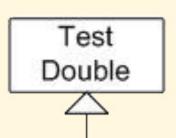
Sustituimos la versión original de nuestro SUT por la versión refactorizada!!!

```
tendremos que
                                             REFACTORIZARLO
                                             para que sea testable
public Factura generarFactura(Cliente cli, Buscador buscar)
                                            throws FacturaException {
```

```
Factura factura:
                                              Versión
int numElems = buscar.elemPendientes(cli);
if (numElems>0) {
                                              refactorizada (en
  //código para generar la factura
                                     SUT
                                              src/main/java)
  factura = ...;
} else {
  throw new FacturaException("No hay nada pendiente de facturar");
return factura;
                 SUT REFACTORIZADA. AHORA SÍ ES TESTABLE!!!
```

IMPLEMENTAMOS EL DOBLE http://xunitpatterns.com/Using%20Test%20Doubles.html

hay varios tipos de dobles



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 Spy

Mock Object

Fake Object

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

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
- Acelerar la velocidad de la ejecución de los tests (un doble tiene mucho menos código que el objeto al que sustituye)
- Conseguir ejecuciones deterministas cuando el comportamiento depende de situaciones aleatorias o dependientes del tiempo
- Simular condiciones especiales. Por ejemplo, simular que hay una caída en la red
- Conseguir acceder a información oculta. Por ejemplo, comprobar si se ha invocado un determinado método dentro del SUT

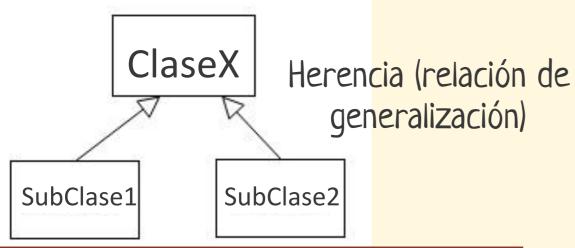


nuestro doble pertenecerá a una clase que "extienda" o "implemente" la clase que contiene nuestro SUT

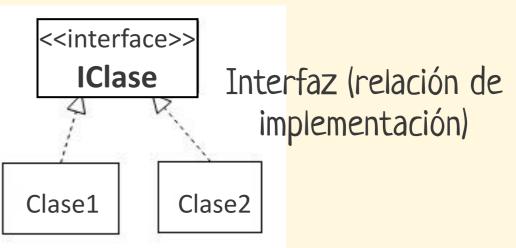
FORGET!

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.

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

IMPLEMENTAMOS EL DOBLE S



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

```
public class Buscador {
    //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!!!

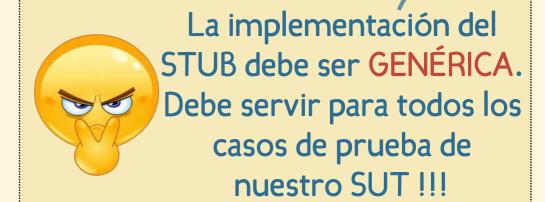
```
public class BuscadorStub extends Buscador {
   int result;

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)
```

Inyectaremos el doble (stub) durante las pruebas



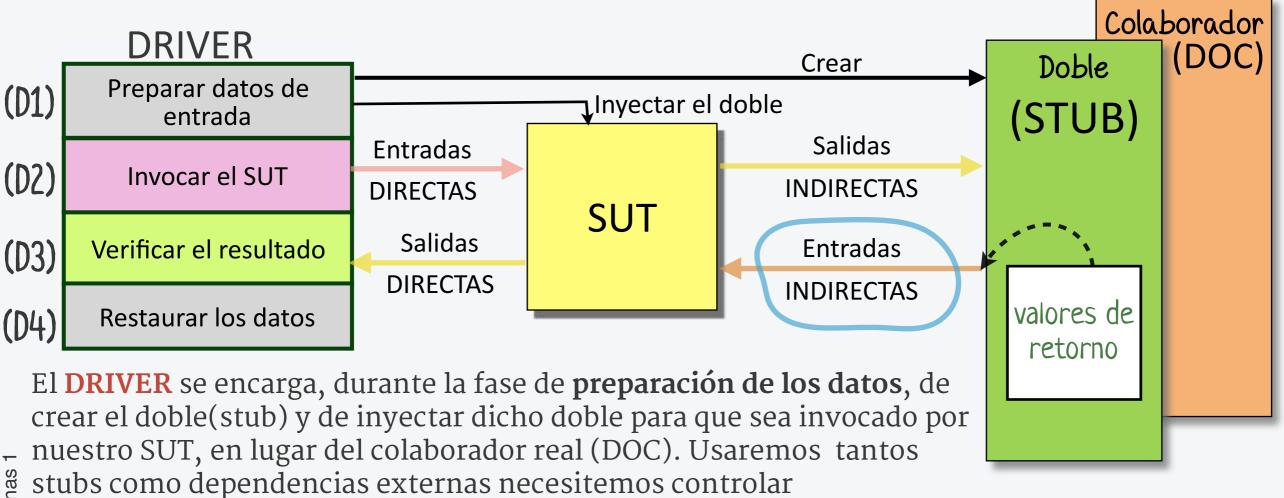


IMPLEMENTACIÓN DEL DRIVER



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)



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 S





O Nuestro driver se encargará de crear el STUB e inyectarlo en nuestro SUT antes de invocarlo con los datos de entrada diseñados.



O Un driver para pruebas de integración tendrá una implementación diferente.

```
Test unitario: aislamos la unidad a probar
                                                         Test de integración: incluye varias unidades
 public class GestorPedidosTest {
                                                           public class GestorPedidosIT {
  @Test
                                                           @Test
  public void testGenerarFactura()
                                                            public void testGenerarFactura()
                             throws Exception {
                                                                                       throws Exception {
                                                             Cliente cli = new Cliente(...);
    Cliente cli = new Cliente(...);
    GestorPedidos sut = new GestorPedidos();
                                                              GestorPedidos sut = new GestorPedidos();
    Buscador buscarStub = new BuscadorStub();
                                                              Buscador buscar = new Buscador();
                                                             Factura expectedResult = new Factura(...);
    buscarStub.setResult = 10;
    Factura expectedResult = new Factura(...);
                                                              Factura realResult =
    Factura realResult =
                                                                      → sut.generarFactura(cli, buscar);
               sut.generarFactura(cli, buscarStub);
                                                             assertEquals(expectedResult, realResult);
                                                    (D3)
    assertEquals(expectedResult, realResult);←
                                                           NO tenemos control sobre las entradas indirectas!
          Aquí controlamos 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!!!
```



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

EJEMPLO 2

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



O 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 interface IService {/Src/main/java
    public int elemPendientes(Cliente cli);
}
```

- Tenemos que <u>refactorizar</u> nuestra unidad para poder inyectar 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 inyectar nuestro doble
 - Los dobles y los drivers siempre se implementarán en /src/test/java



Don't

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 BuscadorSTUB implements IService {
   int resultado;

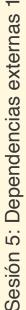
   public BuscadorSTUB(int salida) {
      this.resultado = salida;
   }

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

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

Necesitamos la clase

GestorPedidosTestable para inyectar
el stub durante las pruebas





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

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
   19
```

EJEMPLO 3 Implementamos el driver (los drivers unitarios y de integración son DIFERENTES)

Figuramos puestro SUT

Test unitario 🔦

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

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);
```



Pon't 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



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 {
    private PricingService pricingService;
    public void setPricingService(PricingService service) {
4.
       this.pricingService = service;
6.
    public void process(Order order) {
8.
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. }
17.
```

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
                        NO IMPLEMENTAMOS DOBLES
                        PARA ESTAS DEPENDENCIAS!!
public class Customer {
 private String name;
  public Customer(String name)
    this.name = name;
 //getters y setters
```

¿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

```
código real, utilizado en el SUT
public class PricingService {
                                                                       en src/main
  public float getDiscountPercentage (Customer c, Product p) {
    //calcula el porcentaje de descuento
    return ...;
public class PricingServiceStub extends PricingService {
    private float discount;
                                                          código utilizado durante las
    public PricingServiceStub(float discount) {
                                                                    pruebas
        this.discount = discount;
                                                                          en src/test
    @Override
    public float getDiscountPercentage(Customer c, Product p) {
        return discount;
```

IEST UNITARIO 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 ESPERADAO
0rder	Customer	Product	% descuento	0rder
o.customer = cus o.product = pro o.balance = 30.0	cus.name = "Pedro Gomez"	pro.name="TDD in Action" pro.price = 30.0	10 %	o.customer = cus o.product = pro o.balance = 27.0

NOTA:

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

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

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

TEST DE INTEGRACIÓN



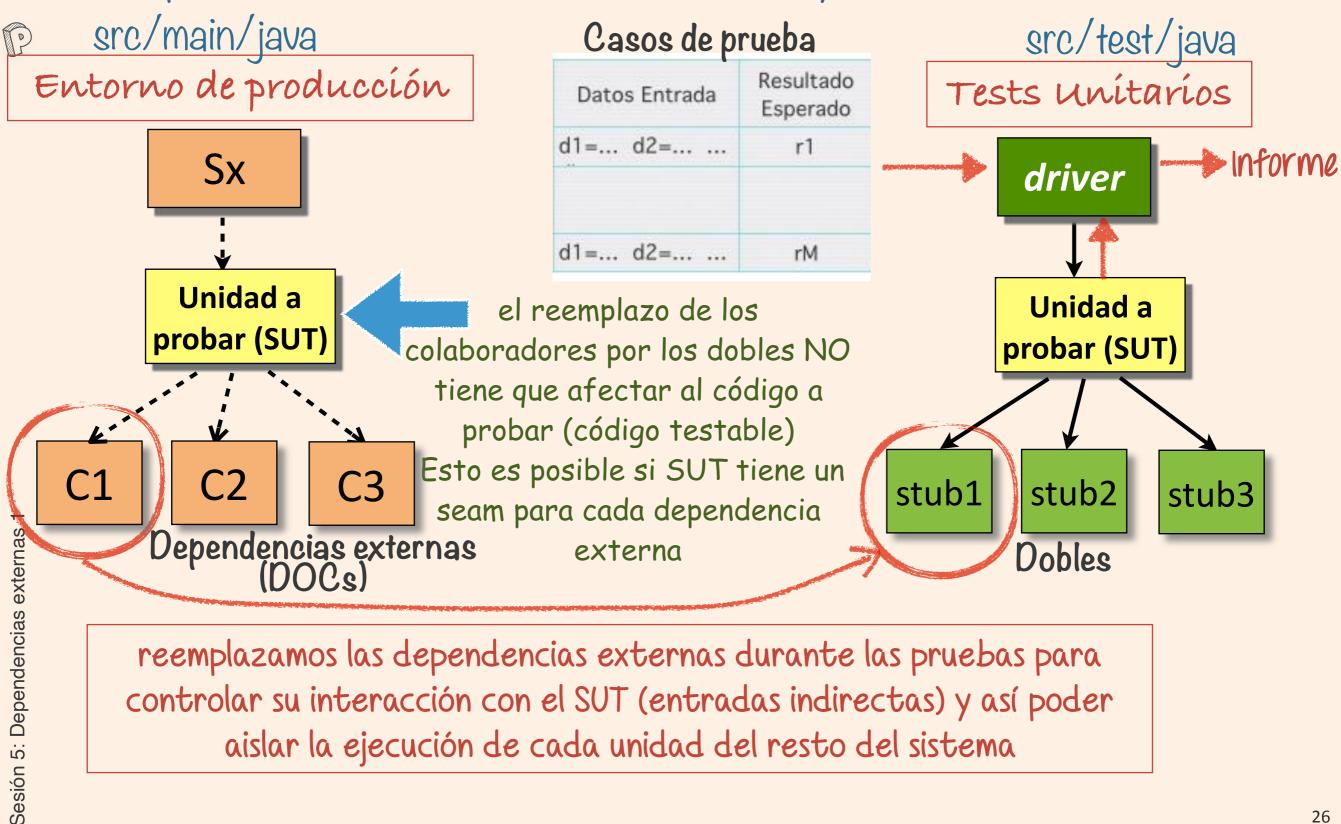
Ejecutamos el código REAL de nuestro DOC durante las pruebas!!!

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

```
public class OrderProcessorIT {
   @Test
   public void test_processOrder() {
     float listPrice = 30.0f;
     float expectedBalance = 27.0f;
     Customer customer = new Customer("Pedro Gomez");
     Product product = new Product("TDD in Action", listPrice);
     OrderProcessor processor = new OrderProcessor();
                                                          Ejecutamos nuestro SUT.
     Order order = new Order(customer, product);
                                                          No tenemos ningún control
     processor.process(order); 
                                                          sobre su dependencia externa
     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));
```

Y AHORA VAMOS AL LABORATORIO...

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



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

REFERENCIAS BIBLIOGRÁFICAS





- 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
- o xUnit Test Patterns. Refactoring test code. Gerard Meszaros
 - * Using test doubles: http://xunitpatterns.com/
 Using%20Test%20Doubles.html
- O Testing with JUnit. Frank Appel. Packt Publishing, 2015.
 - Capítulo 3. Developing independently testable units