

## 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...



CommitStrip.com

P

P

# VERIFICACIÓN Y PRUEBAS UNITARIAS

Verificación basada en el ESTADO

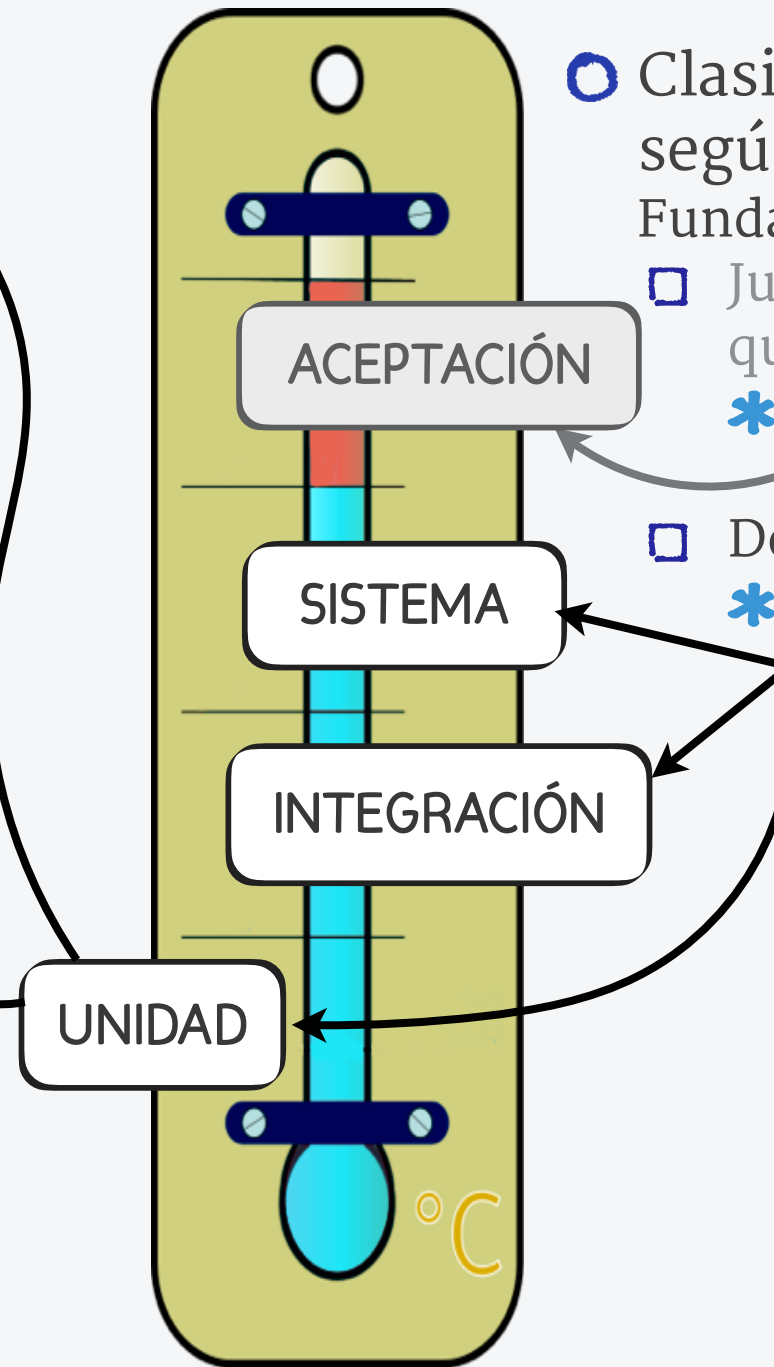


implementamos el driver considerando nuestro SUT como una caja negra

Verificación basada en el COMPORTAMIENTO



implementamos el driver teniendo en cuenta la interacción de nuestro SUT con sus dependencias externas!!!!



○ Clasificación de las pruebas según su OBJETIVO.

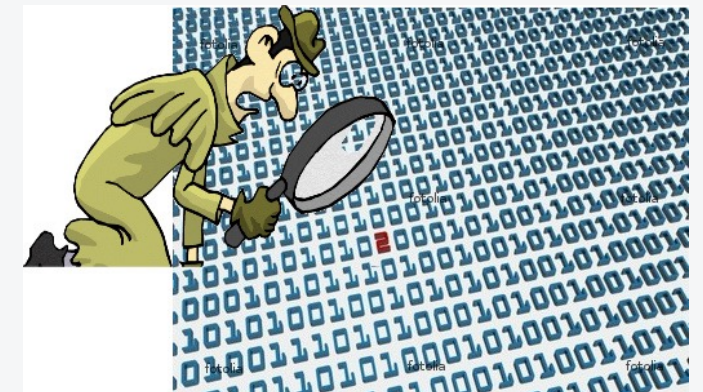
Fundamentalmente probamos PARA:

☐ Juzgar la calidad del software o en qué grado es aceptable

\* Este proceso se denomina VALIDACIÓN: ...

☐ Detectar problemas

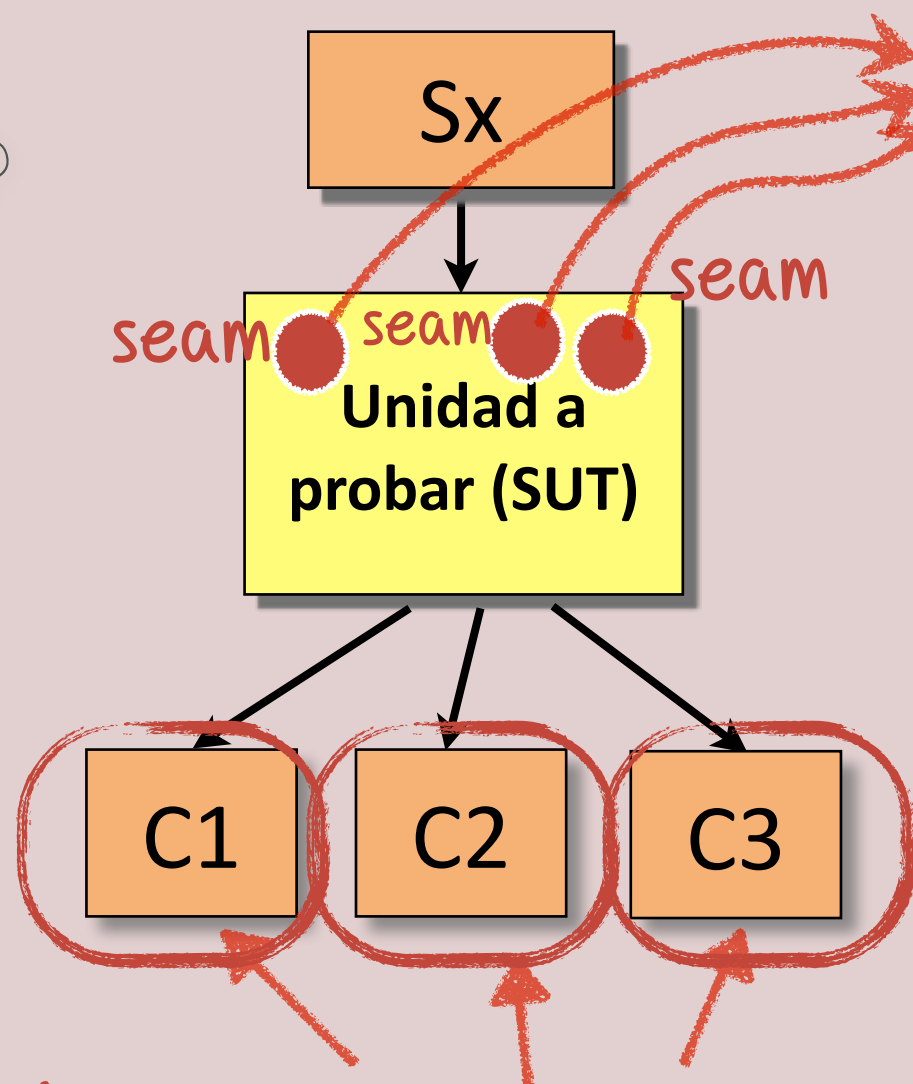
\* Este proceso se denomina **VERIFICACIÓN**: se trata de buscar defectos en el programa que provocarán que éste no funcione correctamente (según lo esperado, de acuerdo con los requerimientos especificados previamente)



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

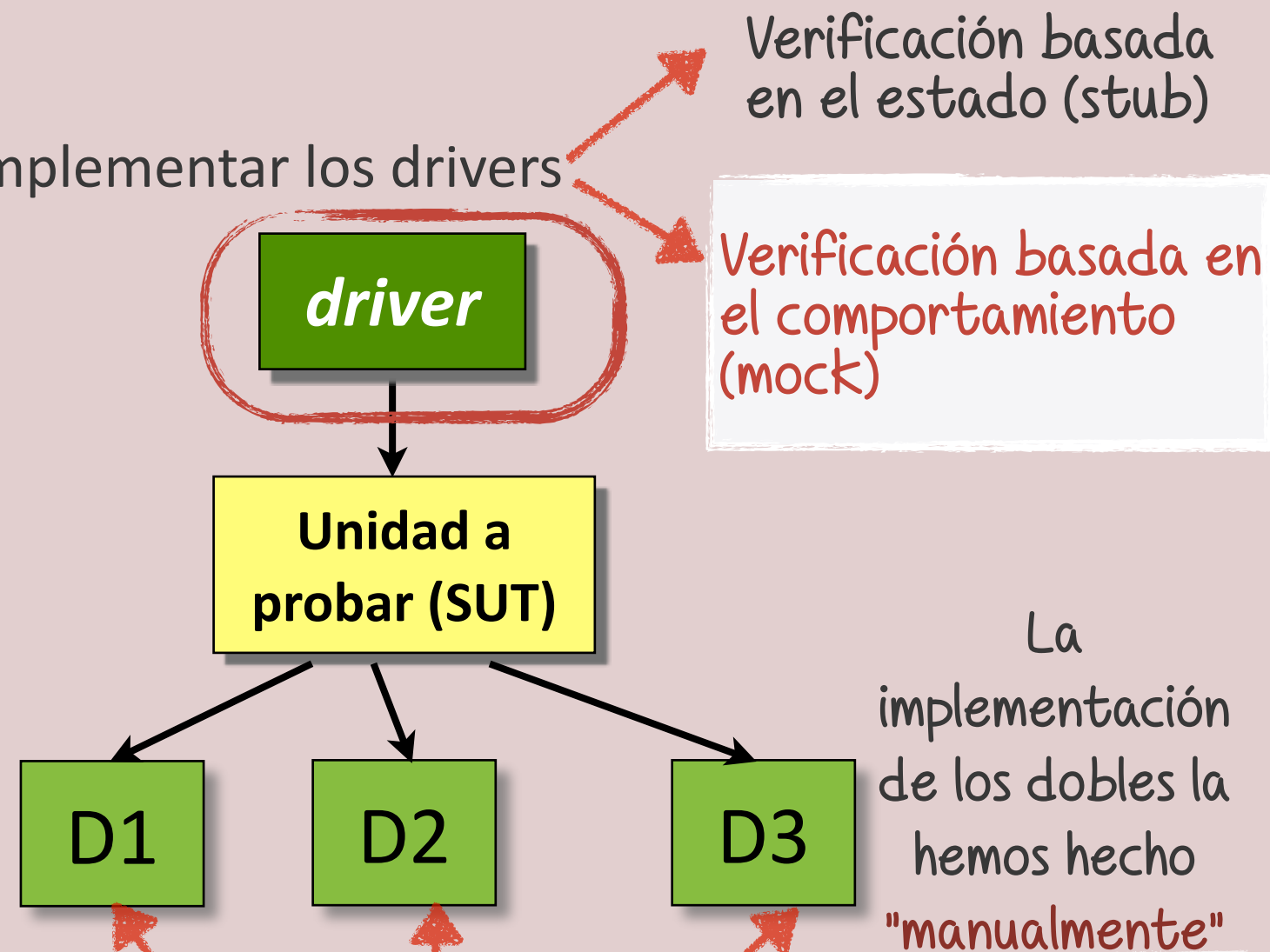


(1) Identificar las dependencias externas



(2) Conseguir que nuestro código sea testable

(4) Implementar los drivers



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

P

# 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

P

# EASYMOCK Y TIPOS DE DOBLES

EasyMock crea de forma automática la clase de nuestra dependencia externa.

Usaremos el siguiente método estático para crear STUBS

P

EasyMock.niceMock(Clasa.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

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

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)

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).

Todas las llamadas esperadas a métodos del doble deben realizarse con los argumentos especificados

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

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**



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





# EASYMOCK: IMPLEMENTACIÓN DE STUBS

Usaremos verificación basada en el estado

 <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 a la SUT)
3. Indicamos al framework que el stub ya está listo para ser invocado por nuestra SUT

 **Don't FORGET!** EasyMock nos "ahorra" **SÓLO** la implementación del doble (stub y/o mock) 

Sesión 5: Dependencias externas 2

- 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
```

1

- 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

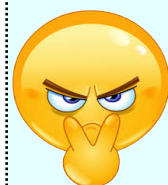
```
//metodo1() devolverá 9 si es invocado por nuestro SUT
//independientemente de cuándo o cuántas veces sea invocado
//y con qué valores de parámetros sea invocado
EasyMock.expect(dep1.metodo1(anyString(),anyInt()))
    .andStubReturn(9);
//metodo2() devolverá una excepción cuando se invoque desde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

2

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

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

3



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

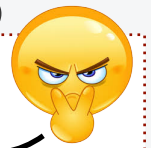
# EASYMOCK: PROGRAMACIÓN DE EXPECTATIVAS ② Comportamiento de STUBS

<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:

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



```
//creamos el doble
Dependencia dep = EasyMock.niceMock(Dependencia.class);

//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: DRIVER CON STUBS Y EASYMOCK

**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!!!**

```
public class GestorPedidosTest {  
    @Test  
    public void testGenerarFactura() {  
        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 driver



```
        Factura expResult = new Factura(...);  
        Factura result = assertDoesNotThrow(  
            ()->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: 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);
```

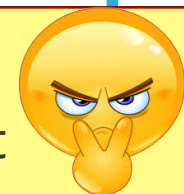
```
        Order order = new Order(customer, product);
```

```
        sut.process(order);
```

```
        assertEquals(expectedBalance, order.getBalance(), 0.001f);
```

DRIVER

Implementación del  
doble dentro del test



Inyección del doble

# EJEMPLO 3: DRIVER CON STUBS Y EASYMOCK

SUT TESTABLE!!!

```
public class CoffeeMachine {
    private Container coffeeC, waterC;

    public CoffeeMachine(Container coffee, Container water) {
        coffeeC = coffee; waterC = water;
    }

    public boolean makeCoffee(CoffeeCupType type) throws NotEnoughException {
        boolean isEnoughCoffee = coffeeC.getPortion(type);
        boolean isEnoughWater = waterC.getPortion(type);

        if (isEnoughCoffee && isEnoughWater) {
            return true;
        } else {
            return false;
        }
    }
}
```

prototipo del método getPortion()

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

DRIVER

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

@BeforeEach

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

@Test

```
public void makeCoffee_NotException() {
    assertDoesNotThrow(() -> EasyMock.expect(coffeeContainerStub.getPortion(EasyMock.anyObject()))
        .andStubReturn(true));

    assertDoesNotThrow(() -> EasyMock.expect(waterContainerStub.getPortion(EasyMock.anyObject()))
        .andStubReturn(true));

    EasyMock.replay(coffeeContainerStub, waterContainerStub);

    assertDoesNotThrow(() -> assertTrue(coffeeMachine.makeCoffee(CoffeeCupType.LARGE)));
}
```

programamos las expectativas de cada doble y los "activamos"

STUBS

inyectamos los dobles

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

Invocar el SUT

Verificar el resultado

Restaurar los datos

Entradas DIRECTAS

Salidas DIRECTAS

SUT

Salidas INDIRECTAS

Entradas INDIRECTAS

STUB

El stub NO puede ser la causa de 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

Invocar el SUT

Verificar que se invoca al colaborador

Verificar el resultado

Restaurar los datos

Entradas DIRECTAS

Salidas DIRECTAS

SUT

Salidas INDIRECTAS

Entradas INDIRECTAS

MOCK

Verificar expectativas

Un mock PUEDE decidir si nuestro test falla o no





# VERIFICACIÓN BASADA EN EL ESTADO VS.

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 inyectamos 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

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

P

# EASYMOCK Y TIPOS DE DOBLES

EasyMock genera de forma automática diferentes clases para nuestros dobles

P

## Creación de STUBS

`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 se 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).

# EASYMOCK: IMPLEMENTACIÓN DE MOCKS

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

El orden de invocaciones a dep1 NO importa!

1. Creamos el mock

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

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

4. Verificamos las expectativas del mock

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

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

1

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 sólo una vez desde nuestro SUT con los
//parámetros indicados y devolverá 9
EasyMock.expect(dep2.metodo1("xxx",7)).andReturn(9);
//metodo1() será invocado 1 vez desde SUT y devolverá una excepción
EasyMock.expect(dep2.metodo1("yy",4)).andThrow(new MyException("message"));
//metodo2() será invocado 1 vez desde nuestra SUT.
//metodo2() es un método que devuelve void
dep1.metodo2(15);
```

2

- 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

Si no invocamos a alguna expectativa desde SUT se lanzará la excepción **AssertionError!!!**



# EASYMOCK: PROGRAMACIÓN DE EXPECTATIVAS

②

Número de invocaciones

- 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...)
- 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

- 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):

**X** `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(5).
    andThrow(new RuntimeException("message"));
```

... en lugar de:

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



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

# EASYMOCK: 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"!!

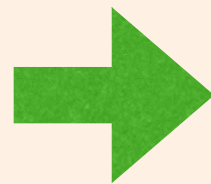
# EASYMOCK: PROGRAMACIÓN DE EXPECTATIVAS

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**

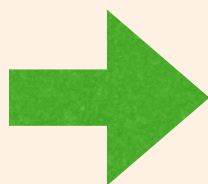
Dos dependencias en clases distintas

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 nuestra 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.mock(Doc1.class);
Doc2 mock2 = ctrl.mock(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 EASYMOCK

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>5.1.0</version>
  <scope>test</scope>
</dependency>
```

El código de nuestros tests DEPENDE de las clases de easymock-5.1.0.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() {
    ...
  }
}
```

```
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
```

SUT

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

# IMPLEMENTACIÓN DEL DRIVER

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

```
public class CurrencyUnitTest {
```

```
@Test
```

```
public void testToEuros() {
```

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

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

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

```
    Assertions.assertDoesNotThrow(() ->
```

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

```
    );
```

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

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

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

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

```
}
```

```
}
```

Resultado  
esperado

Paso 1

Creamos el *mock*

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

Paso 2

Comparamos el resultado real con el esperado

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

19

# PARTIAL MOCKING

- 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").mock();
```

→ `strictMock()`  
→ `niceMock()`

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")
            .strictMock();
        expect(rec.convertX()).andReturn(4);
        expect(rec.convertY()).andReturn(5);
        replay(rec);
        Assertions.assertEquals(20, rec.getArea());
        EasyMock.verify(rec);
    }
}
```



# CONSTRUCTOR CON PARÁMETROS

<https://easymock.org/api/>

- Si tenemos que usar un constructor con parámetros, en lugar del constructor por defecto, usaremos el método: `withConstructor()` definido en la interfaz `IMockBuilder`

Clase de nuestro DDC

```
public class MyClass {  
    ...  
    public MyClass(A a, B b) {  
        ...  
    }  
}
```

```
public class oneTest {  
    @Test  
    public void testC1() {  
        A a = new A();  
        B b = new B();  
        MyClass myClass = createMockBuilder(MyClass.class)  
            .withConstructor(a, b).strictMock();  
        //Expectativas  
    }  
}
```

devuelve una instancia de `IMockBuilder`

usa el constructor con los parámetros indicados

constructor sin parámetros

```
public class OtherClass {  
    public OtherClass() {...}  
    public void sut(int a) {  
        m1(a);  
        m2();  
    }  
    public void m1(int a) {...}  
    public void m2() {...}  
}
```

```
public class OtherClassTest {  
    @Test  
    public void testC1() {  
        OtherClass class1 =  
            partialMockBuilder(OtherClass.class)  
                .withConstructor()  
                .addMockedMethod("m1", int.class)  
                .addMockedMethod("m2").strictMock();  
        // These are the expectations.  
        class1.m1(10);  
        class1.m2();  
        replay(class1);  
  
        class1.sut(10);  
        verify(class1);  
    }  
}
```

podemos indicar los tipos de parámetros, separados por "comas"

# TEN EN CUENTA QUE ...



- 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

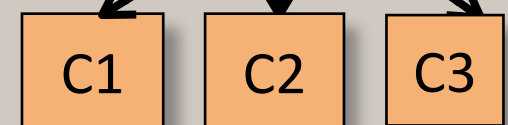


- Si nuestras dependencias externas no proporcionan entradas indirectas al SUT que debemos controlar, nuestro doble NO será un stub, usaremos un mock y por tanto tendremos que utilizar verificación basada en el comportamiento si queremos comprobar que el doble ha sido invocado desde el SUT
- Si los colaboradores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




Identificar las **dependencias** externas de nuestro SUT



Colaboradores (DOCs)



Asegurarnos de que nuestro código (SUT) es **TESTABLE**:  
Si no lo es necesitamos **REFACTORIZAR** nuestro SUT (y/o la clase a la que pertenece) para poder realizar dichos reemplazos controlados durante las pruebas




Implementar un **DOBLE** para reemplazar a cada colaborador durante las pruebas



Dobles

Los dobles se pueden implementar manualmente o con EasyMock



Implementar los **DRIVERS** correspondientes. Para ello podemos hacer una: verificación basada en el **ESTADO**: **driver**  
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)  
verificación basada en el **COMPORTAMIENTO**:  
nos interesa, además, verificar que las interacciones entre nuestro SUT y las dependencias externas se realizan correctamente

## 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



P

# EJERCICIO RESUELTO

S

P

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

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

	minutos	hora	Resultado
C1	10	15	208

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!!!

DRIVER

```
public class GestorLlamadasMockTest {  
    private ServicioHorario mock;  
    private GestorLlamadas gll;  
  
    @Before  
    public void inicializacion() {  
        mock = EasyMock  
            .mock(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

- 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

```
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;  
    }  
}
```

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!!!**

P

# EJERCICIO PROPUESTO



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

P

```
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:

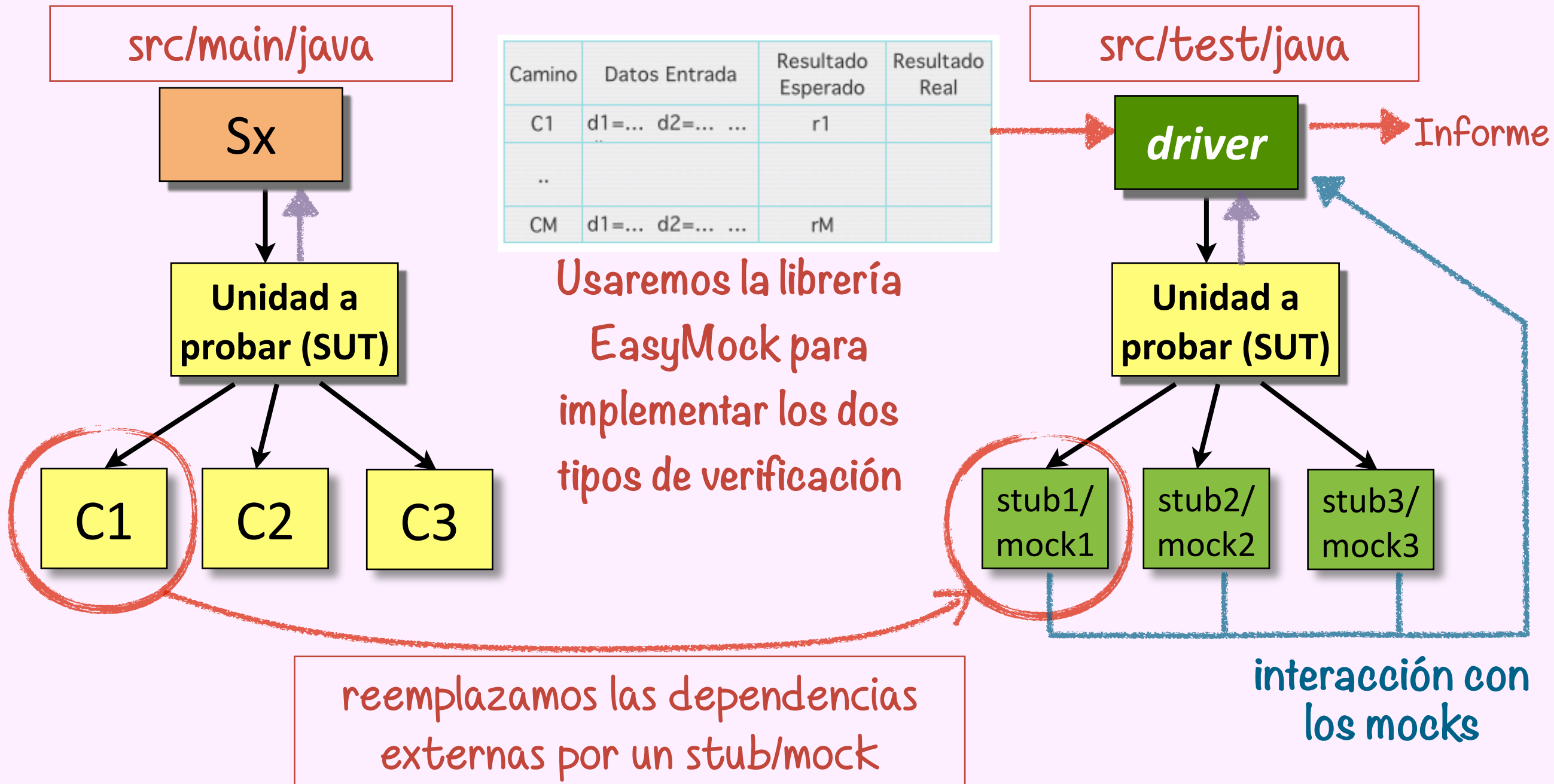
- ❑ dónde se ubican física y lógicamente cada uno de los ficheros.
- ❑ cómo debes configurar el pom del proyecto
- ❑ cómo lanzar la ejecución de los tests con maven
- ❑ las diferencias entre un stub y un mock
- ❑ las diferencias de los drivers cuando verificamos basándonos en el estado y en el comportamiento

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



# REFERENCIAS BIBLIOGRÁFICAS



- The art of unit testing: with examples in C#. 2nd edition Roy Oshero. 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>