Planificación y Pruebas de Sistemas Software

Curso 2015-16

Sesión 6: Dependencias externas (2)

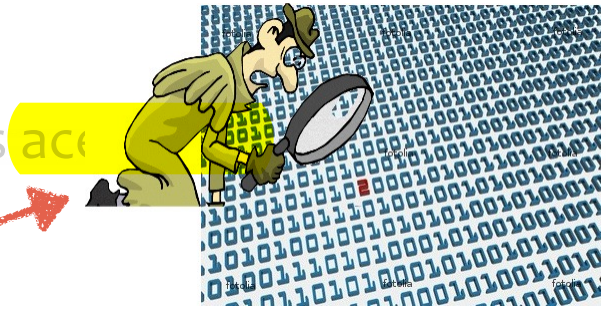
Verificación basada en el comportamiento

Librería *EasyMock*

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

Verificación y pruebas unitarias

- Fundamentalmente probamos PARA:
  - Juzgar la calidad del software o en qué grado es ac
    - 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)



nivel de unidades

nivel de integración

nivel de sistema

nos basamos en el estado

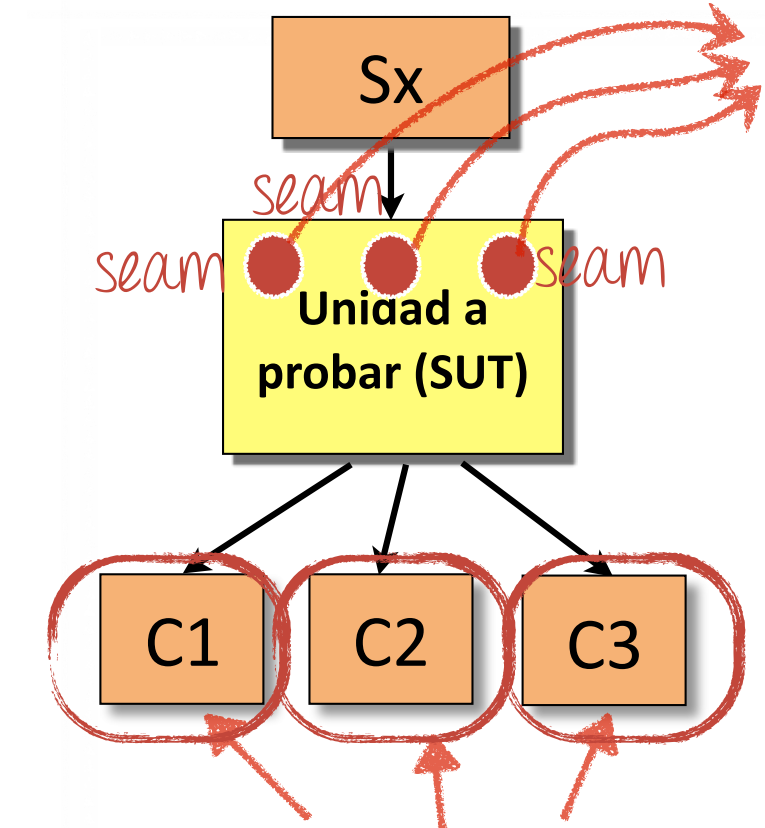
implementamos el driver considerando nuestro SUT como una caja negra

nos basamos en el COMPORTAMIENTO

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

Ya hemos visto el proceso para automatizar pruebas de unidad

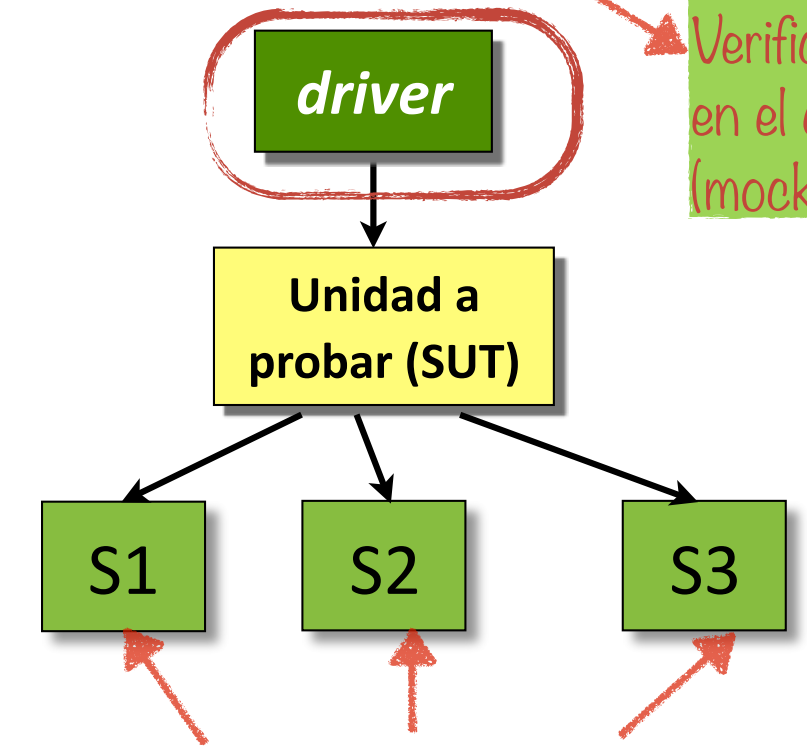
- Necesitamos:



(2) Conseguir que nuestro código sea testable

(4) Implementar los drivers

- Verificación basada en el estado (stub)
- Verificación basada en el comportamiento (mock)

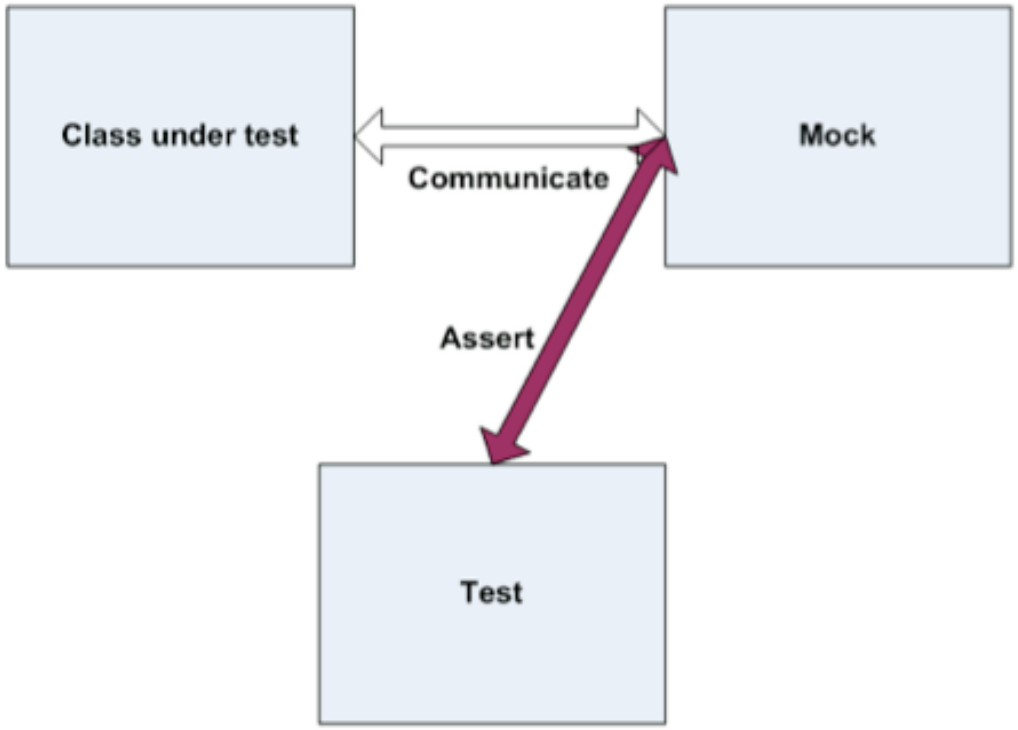


(1) Identificar las dependencias externas

(3) Proporcionar una implementación ficticia para cada colaborador (Ci). Ésta se denomina *mock* o *stub* dependiendo del tipo de verificación que realice el *driver*

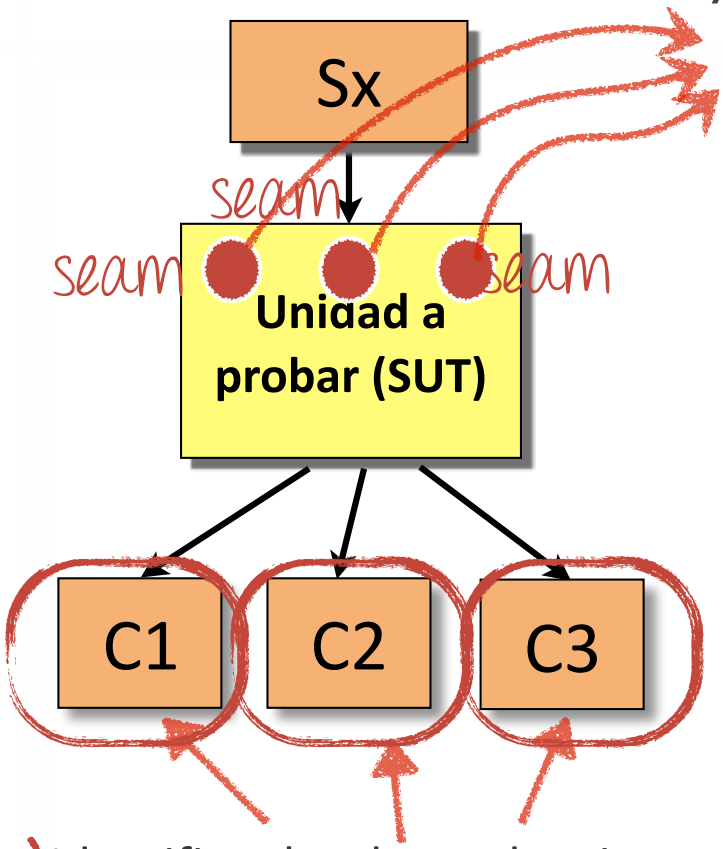
Mocks

- Los mocks, al igual que los stubs, son código que sustituye al código real (dependencia externa) durante la realización de las pruebas.
  - La diferencia fundamental con los stubs es que se realiza una verificación de la corrección basada en el **COMPORTAMIENTO**, lo que significa que se comprueba si la interacción de nuestro SUT con la dependencia externa es la correcta
- Un mock es un objeto
  - La clase a probar se comunica con el objeto mock, y toda la comunicación se registra en el mock. La prueba utiliza el objeto mock para verificar que el test "pasa"
- Utilizaremos la librería EasyMock para trabajar con objetos mock y verificar el comportamiento de nuestro SUT



Sobre las implementaciones de S1, S2, ...

- Si utilizamos la librería EasyMock:



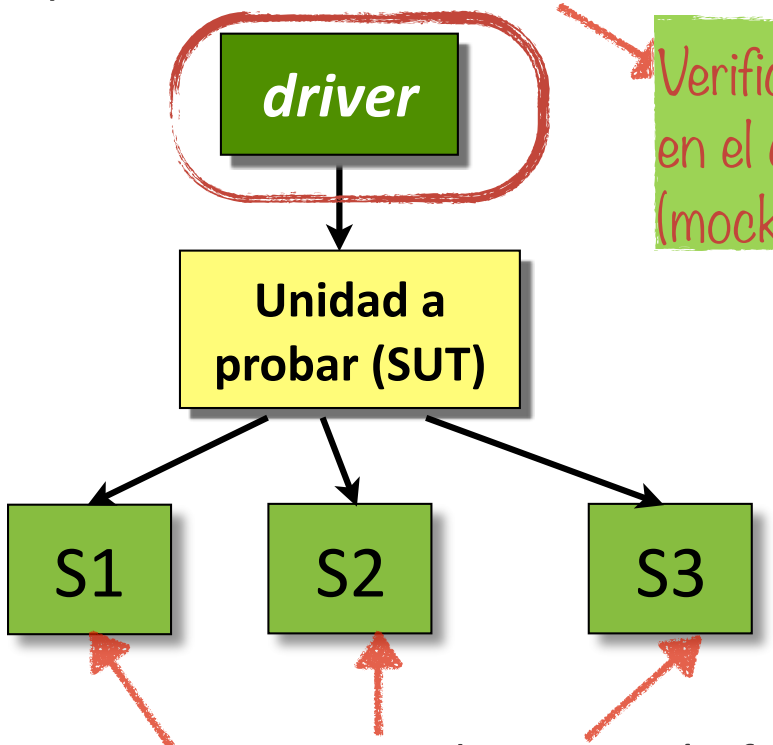
(2) Conseguir que nuestro código sea testable

(1) Identificar las dependencias externas

(4) Implementar los drivers

Verificación basada en el estado (stub)

Verificación basada en el comportamiento (mock)



La librería EasyMock nos permite crear e incluir esta implementación en el driver, verificando el comportamiento

(3) Proporcionar una implementación ficticia para cada colaborador (Ci). Ésta se denomina mock o stub dependiendo del tipo de verificación que realice el *driver*

Librería EasyMock

http://easymock.org/

- EasyMock utiliza lo que se denomina paradigma record/playback. Básicamente establece dos modos de funcionamiento:
  - Modo **record**: se indica qué métodos serán invocados por el objeto mock y se especifica cuál será la respuesta proporcionada por éste
  - Modo **playback**: el objeto mock espera a ser invocado por la unidad a probar, proporcionando las respuestas especificadas en el modo record
- Los secuencia básica de pasos para usar EasyMock son:
  1. Crear un objeto mock (EasyMock.createMock())
  2. Especificar (record) el comportamiento del mock (EasyMock.expect())
  3. Cambiar a modo playback (estado replay) (EasyMock.replay())
     - La invocación del método replay() indica a EasyMock que ya hemos terminado de especificar el comportamiento del mock y que pasamos al modo playback
  4. Invocar a la unidad a probar
  5. Verificar que el mock se utiliza realmente por el SUT (EasyMock.verify())
     - Si el mock realmente NO es invocado desde el SUT se genera un java.lang.AssertionError

Extraído de: http://www.shaunabram.com/easymock-2/

EasyMock: verificación del comportamiento

- El paso 5 anterior consiste invocar a EasyMock.verify(). La verificación realizada depende de cómo se haya creado el mock:
  - Verificación "normal" (creación mediante EasyMock.createMock())
    - Todas las llamadas esperadas a métodos (especificadas en el paso 2) realizadas por el mock deben realizarse con los argumentos especificados. El orden en el que se realiza la llamada a los métodos no importa. Si se realizan llamadas a métodos no especificados, el test fallará
  - Verificación "strict" ((creación mediante EasyMock.createStrictMock())
    - Todas las llamadas esperadas a métodos realizadas por el mock deben realizarse con los argumentos especificados, en el orden especificado. Si se realizan llamadas a métodos no especificados, el test fallará
  - Verificación "nice" ((creación mediante EasyMock.createNiceMock())
    - Todas las llamadas esperadas a métodos realizadas por el mock deben realizarse con los argumentos especificados, en cualquier orden. Si se realizan llamadas a métodos no especificados, no fallará el test. Se proporcionan valores por defecto "razonables" para los métodos no especificados (0, null o false)

API *EasyMock*

Lista completa de métodos que se pueden utilizar:
http://easymock.org/api/
(ver *EasyMock* e *IExpectationSetters*)

Statement	Description
createMock(nombreClase.class)	Crea un objeto mock con la misma interfaz que nombreClase.
** expectedOperation()	Almacena una llamada a un método llamado expectedOperation(), sin parámetros y que devuelve void
** expectedOperation(argValue)	Almacena una llamada a un método llamado expectedOperation(), con el parámetro argValue y que devuelve void
expect(expectedOperation(argValue)). andReturn (expectedReturnValue)	Almacena una llamada a un método llamado expectedOperation(), con el parámetro argValue y que devuelve expectedReturnValue
expect(expectedOperation(argValue)). times(x). andReturn (expectedReturnValue)	Almacena x llamadas consecutivas a expectedOperation(), con el parámetro argValue y que devuelve expectedReturnValue
expect(expectedOperation(argValue)). andThrow(Exception)	Almacena una llamada a expectedOperation(argValue), que devuelve la excepción Exception
verify(mock)	Verifica que se invoca al SUT desde la unidad a probar

** Si expectedOperation() devuelve void, no utilizaremos el método expect(), simplemente invocaremos al método expectedOperation() directamente

Ejemplo de uso de la librería *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>3.4</version>
      <scope>test</scope>
</dependency>
```

- Queremos probar el 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* para poder invocar el método *ExchangeRate.getRate()*

Implementación de un test

- La librería nos proporciona la implementación del código que sustituirá a la dependencia externa durante las pruebas. Implementaremos un test Junit

```
public class CurrencyUnitTest {

    @Test
    public void testToEuros() throws IOException {
        Currency testObject = new Currency(2.50, "USD");
        Currency expected = new Currency(3.75, "EUR");
        ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
        EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);
        EasyMock.replay(mock);
        Currency actual = testObject.toEuros(mock);
        assertEquals(expected, actual);
        EasyMock.verify(mock);
    }
}
```

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

Ejecutamos el método a probar utilizando el *mock*

Paso 4

Se verifica que realmente se invoca al *mock* desde nuestro SUT

Paso 5

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 sóla vez

Paso 3

Especificación del comportamiento

- El caso más simple es cuando la dependencia externa no devuelve ningún valor y no tiene argumentos. En este caso llamamos al método directamente:
  - P.ej. mock.expectedOperation();
- Si la dependencia devuelve un valor (o genera una excepción), podemos utilizar el método "andReturn()" (o "andThrow()"):
  - P.ej. expect(mock.expectedOperation()).andReturn(expectedReturnValue)
  - P.ej. expect(mock.expectedOperation()).andThrow(new MyException())
- Podemos indicar el número de veces que se ejecutará la llamada:
  - P.ej. expect(mock.operation()).times(3).andReturn(expectedReturnValue)
  - P.ej. expect(mock.operation()).atLeastOnce().andReturn(expectedReturnValue)
- Podemos especificar varias llamadas a la dependencia externa, y encadenarlas en la misma sentencia:
  - P.ej. expect(mock.operation()).andReturn(true).anyTimes().andThrow(new RuntimeException("message"));
  - ... o no encadenarlas: expect(mock.operation().andReturn(true).times(1,5); expectLastCall().andThrow(new RuntimeException("message"));

Especificación de argumentos de la invocación

- El caso más simple es cuando especificamos un único valor en la llamada
  - P.ej. mock.expectedOperation(argValue);
- EasyMock, para comparar argumentos de tipo Object, utiliza por defecto el método equals() de dichos argumentos. Esto puede plantear problemas en algunos casos:
  - P.ej. String[] data = new String[] { "Data 1", "Data 2" };
    expect(mock.operation(data)).andReturn(42);

    Si el método es invocado con otro array, con los mismos datos, se lanzará una excepción, ya que el método equals() devolverá false. En su lugar utilizaremos el siguiente matcher:

    expect(mock.operation(aryEq(data)).andReturn(42)
- Podemos utilizar diferentes matchers para flexibilizar los argumentos de la interacción con la dependencia externa:
  - eq(X value): compara que los valores sean iguales, X puede ser cualquier tipo primitivo u objeto
  - anyBoolean(), anyFloat(), isNull(), notNull(), startsWith(String prefix), ...

Partial mocking

- En ocasiones podemos necesitar proporcionar una implementación ficticia no de toda la clase, sino sólo de algunos métodos.
  - Esto ocurre normalmente cuando estamos probando un método que realiza llamadas a otros de su misma clase
- La librería EasyMock nos permite realizar un mocking parcial de una clase, utilizando el método createMockBuilder(), de la siguiente forma:
  - ToMoc mock = createMockBuilder(ToMock.class)
    .addMockedMethod("mockedMethod").createMock()
    En este caso, sólo los métodos añadidos con "addMockedMetod()" serán "mocked"

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

  //getters y setters

    public int getArea() {
        return getX() * getY();
    }
}
```

```
public class RectanglePartialMockingTest {
  private Rectangle re;

  @Test
    public void testGetArea() {
      re = createMockBuilder(Rectangle.class)
           .addMockedMethods("getX", "getY").createMock();
      expect(re.getX()).andReturn(4);
      expect(re.getY()).andReturn(5);
      replay(re);
      assertEquals(20, re.getArea());
      verify(re);
    }
}
```

EasyMock: objetos mock vs. objetos stub

http://jblewitt.com/blog/?p=316

- Los objetos mock son diferentes de los stubs, ya que los objetos mock "registran" el comportamiento en lugar de devolver valores pre-establecidos
- EasyMock nos permite trabajar con objetos mock y realizar una verificación basada en el comportamiento, es decir:
  - determinar las operaciones y expectativas sobre las clases
  - comparar las expectativas con las operaciones realmente realizadas, y
  - verificar que lo que realmente ha ocurrido es lo que se esperaba
- Una verificación basada en el estado, utilizando STUBS:
  - Podemos tener que implementar un elevado número de objetos stub, para cada uno de los cuales tenemos que "predefinir" las respuestas que proporciona
  - Resulta más complejo el preparar todos los stubs, pero los tests son más robustos ante cambios en la implementación de nuestro SUT
- Una verificación basada en el comportamiento, utilizando MOCKS (y EasyMock):
  - Podemos implementar el test de forma muy sencilla, 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

Uso de objetos stub con EasyMock

http://jblewitt.com/blog/?p=316

- Es posible, utilizando EasyMock, implementar drivers que realizan una verificación basada en el estado
  - Haciendo uso de los métodos anyTime(), anyObject(), checkOrder(false)
  - Haciendo uso de los métodos asStub(), andStubReturn(), andStubThrow()
- Utilizando un niceMock

```
...
/* Preparamos los ojetos mock */
this.myStubObject = EasyMock.createMock(IMyService.class);
/* Generamos las respuestas predefinidas */
EasyMock.expect(this.myStubObject.getData())
        .andReturn(new Integer(5)).anyTimes();
  EasyMock.expect(this.myStubObject
                      .convert(EasyMock.anyObject()))
          .andReturn("ABC").anyTimes();
  EasyMock.checkOrder(this.myStubObject, false);
  EasyMock.replay(this.myStubObject);
}
...
```

```
public interface IMyService {
  public Integer getData();
  public String convert(Object obj);
}
```

Uso de anyTime( ), anyObject( ) y checkOrder(false)

No utilizamos verify()

Uso de andStubReturn( ) y anyObject( )

```
...
/*Otra forma de generar las respuestas predefinidas */
EasyMock.expect(this.myStubObject.getData())
        .andStubReturn(new Integer(5));
EasyMock.expect(this.myStubObject
        .convert(EasyMock.anyObject()))
        .andStubReturn("ABC");
EasyMock.replay(this.myStubObject);
...
```

Ejercicio

- Usa la librería EasyMock para implementar un test con verificación basada en el comportamiento para el método GestorLlamadas.calculaConsumo()

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

	minuto	hora	Resultado esperado
C1	10	15	208

```
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;
    }
  }
}
```

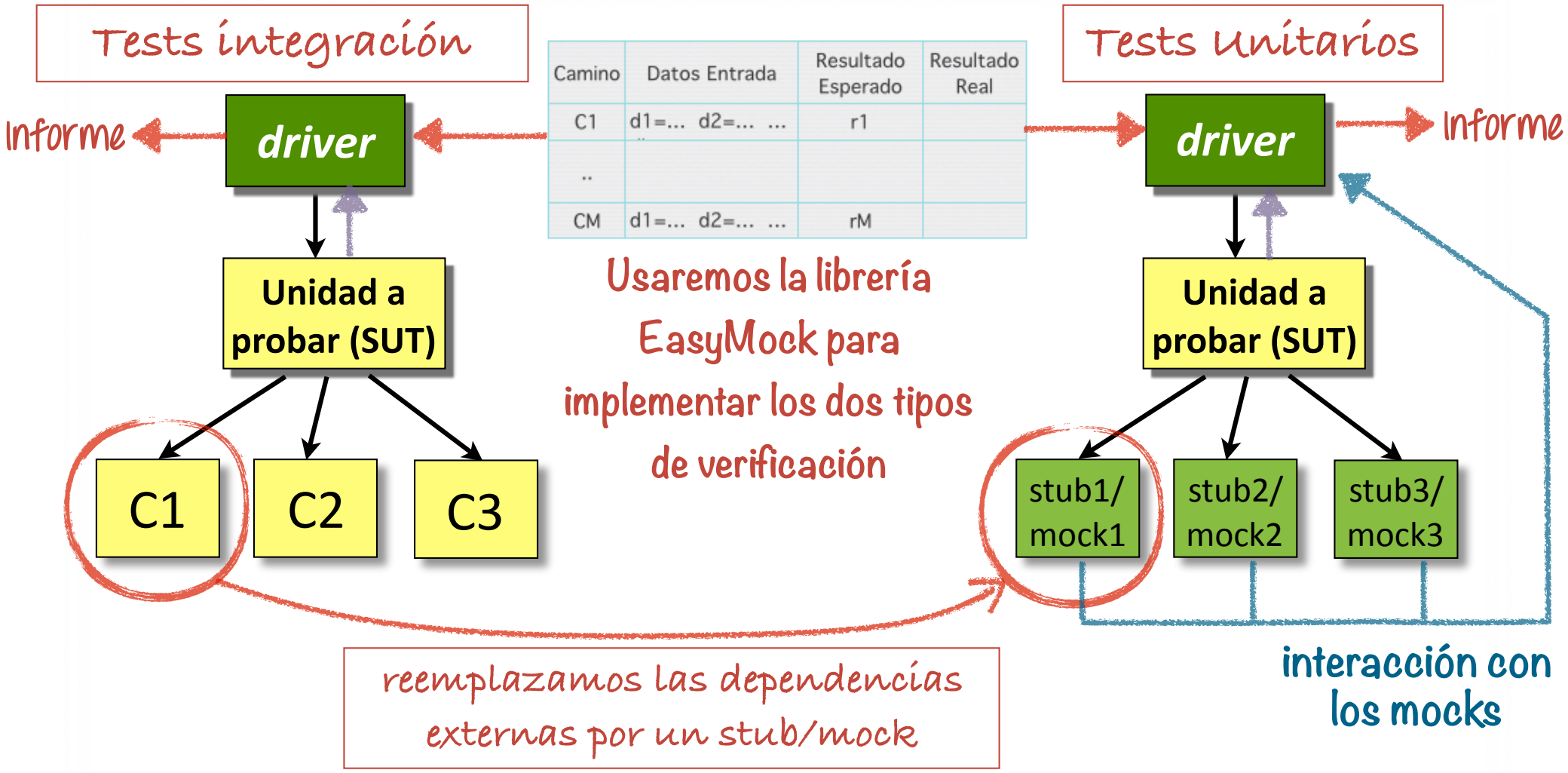
```
public class GestorLlamadasMockTest {
  private ServicioHorario mock;
  private GestorLlamadas gll;

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

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

Razones para usar STUBS/MOCKS

- En una prueba, los stubs/mocks pueden simular el comportamiento de objetos complejos y por lo tanto, resultan útiles cuando no es práctico o resulta imposible incorporar objetos reales en dicha prueba
  - Cuando estamos hablando de pruebas de unidad, por otro lado, es necesario usar stubs/mocks para controlar las dependencias externas
- Es conveniente utilizar stubs/mocks cuando el objeto real:
  - Proporciona resultados no deterministas (p.ej. temperatura actual, hora actual)
  - Contiene estados difíciles de crear o reproducir (p.ej. un error en la red)
  - Es lento (p.ej. una base de datos completa que tiene que inicializarse antes de realizar las pruebas)
  - No existe o puede cambiar su comportamiento
  - Puede tener que incluir información y métodos exclusivamente para las pruebas (que no estarán en el objeto real)

Y ahora vamos al laboratorio…

Vamos a implementar tests unitarios y de integración utilizando verificación basada en el COMPORTAMIENTO y/o en el ESTADO



Referencias bibliográficas

- The art of unit testing: with examples in .NET. Roy Osherove. Manning, 2009.
  - 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
- Using EasyMock to create stub objects. James Blewitt. 2009
  - http://jblewitt.com/blog/?p=316