

Frameworks para Implementar Dóbles:

- 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 Mock).
- Inyectaremos los dobles en la unidad a probar antes de injectarla.
- Un framework contribuye a crear test potencialmente frágiles y difíciles de mantener.

Para crear STUBS usaremos `EasyMock.niceMock(clase.class)`. devolviendo un doble para la clase o interfaz

- El orden en el que se realizan las invocaciones a los métodos no se chequea
- Usamos a los STUBS para realizar una verificación basada en el estado por lo que no estamos interesados en verificar el orden de las invocaciones de nuestro SUT al doble solo queremos controlar las entradas indirectas.
- EasyMock implementa un doble por defecto para cada método de la clase, pero si no se define su comportamiento devolverá null, 0 o false.
- El objeto NiceMock verifica que el SUT invoca al doble usando los parámetros especificados
- Primero tendremos que generar la clase que contendrá a dicho doble. EasyMock crea un doble para cada uno de los métodos de la clase y después programaremos el comportamiento de sus métodos.

① Creamos el Stub :

```
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
```

② Programamos las expectativas y los valores de entrada

```
//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();
```

③ Indicamos que el STUB ya está listo

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

- Los parámetros que recibirá serán cualquier String y cualquier Entero
- Devolverá 9

EasyMock Programación de Expectativas

- Un Stub puede devolver resultados diferentes dependiendo de los valores de los parámetros de las invocaciones.
- O podemos "relajarnos" los valores de los parámetros de forma que devolvamos un determinado resultado independientemente de los valores de entrada del STUB. Usaremos `anyType()`

```
//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.anyType())).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");
```

Ejemplo 1 Drivers con Stub 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();

        Buscador stub = EasyMock.niceMock(Buscador.class); ①
        EasyMock.expect(stub.elemPendientes(anyObject())) ②
            .andStubReturn(10);
        EasyMock.replay(stub); ③
    }
}
```

DRIVER

La "implementación" del doble la realizamos "dentro" del driver

Inyección del doble

anyObject() → "cualquier objeto"
 anyChar() → "cualquier char"
 anyFloat() → "cualquier float"
 anyInt() → "cualquier int"
 anyString() → "cualquier objeto String" ...

- ① Generamos el Stub con NiceMock
- ② Programamos las expectativas diciéndole que da igual el objeto que reciba devolverá 10
- ③ Hacemos el replay para verificar que está listo
- ④ Por ultimo hacemos los asserts para verificar el resultado obtenido.

Ejemplo 2 Driver con Stub y EasyMock

```
public class OrderProcessor {
    private PricingService pricingService; ①
    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();

        ② PricingService stub = EasyMock.niceMock(PricingService.class);
        EasyMock.expect(stub.getDiscountPercentage(anyObject(), anyObject()))
            .andStubReturn(discount);

        ③ sut.setPricingService(stub);
        EasyMock.replay(stub);

        ④ 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

● Esta es nuestra dependencia externa
 ● Al contar con este setter podemos injectar nuestro Stub sin refactorizar

- ① Creamos el objeto que contiene nuestra SUT con todos los parámetros que necesita
- ② Generamos nuestro Stub y programamos sus expectativas
- ③ Lo injectamos mediante el setter en nuestra SUT y hacemos el replay para verificar.
- ④ Hacemos los asserts

Ejemplo 3 Inyección en el Constructor

```
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;  
        }  
    }  
  
    public boolean getPortion(CoffeeCupType coffeeCupType) throws NotEnoughException {  
        // Implementación del método  
    }  
}
```

SUT TESTABLE!!!

prototipo del método getPortion()

SUT

DRIVER

Detectamos un punto de inyección en el constructor

Dependencias Externas

① Generamos los Stubs y los注入amos en el constructor

② Dentro del test programamos su comportamiento

!! Cuidado que lanza excepción

tendremos que programar su comportamiento dentro de assertdoesnotthrow

```
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() {  
        [1] assertDoesNotThrow(() -> EasyMock.expect(coffeeContainerStub.getPortion(EasyMock.anyObject()))  
            .andStubReturn(true));  
        [2] assertDoesNotThrow(() -> EasyMock.expect(waterContainerStub.getPortion(EasyMock.anyObject()))  
            .andStubReturn(true));  
        EasyMock.replay(coffeeContainerStub, waterContainerStub);  
        assertDoesNotThrow(() -> assertTrue(coffeeMachine.makeCoffee(CoffeeCupType.LARGE)));  
    }  
}
```

STUBS

inyectamos los dobles

SUT

Diferencias entre Stubs y Mocks

- **Stubs:** Es un objeto que actúa como punto de control, para entregar entradas indirectas al SUT cuando son invocadas, se usan para la verificación basada en Estado.
- **Mock Objeto:** Es un objeto que actúa como punto de observación para las salidas indirectas del Sut, puede devolver información cuando se le invoca o no devolver nada. Además registra las llamadas recibidas desde el sut y compara las llamadas reales con las llamadas previamente definidas como expectativas, haciendo que el test falle si no se cumplen. Utiliza verificación basada en Comportamiento.

Tipos

EasyMock.niceMock (Clase.class): Stubs

- El orden que se realizan las llamadas al doble no se chequean.
- Se permiten invocaciones a todos los métodos del objeto, si no se han programado las expectativas devolverá null, 0, false.
- Todas las llamadas esperadas realizadas por el doble deben realizarse con los argumentos especificados

EasyMock.mock (Clase.class): Creación de mocks si solo hay una invocación al doble.

- El orden que se realizan las llamadas al doble no se chequean.
- El comportamiento por defecto para todos los métodos del objeto es lanzar un AssertionErroe para cualquier invocación no esperada.
- Todas las llamadas esperadas realizadas por el doble deben realizarse con los argumentos especificados

EasyMock • Strict Mock (Clase.class): Creación de Mock con cualquier nº de invocaciones

- Se comprueba el orden en el que se realizan las invocaciones al doble.
- Si se invoca a un método no esperado se lanza un AssertionErrot.
- Todas las llamadas esperadas realizadas a métodos realizados por el doble deben realizarse con los argumentos especificadas

EasyMock implementación de Mocks:

1 Creamos el Mock.

```
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);
```

2 Programamos sus expectativas

```
//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);
```

3 Activamos el Mock usando el método Replay

```
EasyMock.replay(dep1,dep2);
```

4 Despues de hacer los assert deberemos invocar al método Replay

```
EasyMock.verify(dep1,dep2);
```

EasyMock programación de expectativas: Debemos indicar como interactua nuestro Sut con el objeto Mock que hemos creado:

- 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 se producirá un fallo de Assertion Errot.
- Si es un Mock() y las invocaciones del SUT no coinciden con las expectativas programadas (nº de invocaciones parámetros) entonces se producirá un fallo

```
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)
```

Las expectativas pueden programarse de manera encadenada

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

EasyMock para comparar argumentos del tipo object utiliza por defecto el método equals de dichos argumentos

```
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"!!

Con Respeto 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 invocaciones a dicho objeto debe de coincidir exactamente con el orden establecido en las expectativas.

Con Respeto al orden de ejecución de las expectativas entre varios Mock:

- Para poder establecer un orden de invocaciones entre objetos strictMock usaremos IMockControl

Ejemplo: Con varios objetos StrictMock solo 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);
```

Ejemplo Mismo ejemplo pero esta vez se chequea el orden entre los objetos y ademas el orden de las invocaciones entre 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();
```

Importante : Siempre que usamos ctrl.mock si hemos creado un Strict control, los mocks que creamos serán del tipo Strict

Para utilizar EasyMock debemos añadir lo siguiente al POM

```
<dependency>
  <groupId>org.easymock</groupId>
  <artifactId>easymock</artifactId>
  <version>5.1.0</version>
  <scope>test</scope>
</dependency>
```

Partial Mocking:

En ocasiones podemos necesitar proporcionar una implementación ficticia, no de toda la clase sino solo de algunos métodos. Esto ocurre cuando estamos probando un método que realiza llamadas internas a otros métodos de la misma clase

```
ToMock mock = partialMockBuilder(ToMock.class)
  .addMockedMethod("mockedMethod").mock();
```

strictMock() →
F niceMock()

Ejemplo de Uso de partial Mock

```
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 Si tenemos un constructor con parámetros en vez de uno por defecto haremos uso de withConstructor:

Clase de nuestro DOC

```
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

Ejemplo Constructor Sin parámetros pero con withConstructor

Preguntar a Eli:

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"

Ejercicio Resuelto Basado Comportamientos

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

```
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!!!

Dependencia Externa
Punto de inyección

- ① Creamos el mock de nuestra dependencia
- ② Injectamos la dependencia
- ③ Programamos el comportamiento
- ④ Decimos que está listo
- ⑤ Hacemos los Assert
- ⑥ Verificamos

```
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);  
    }  
}
```

DRIVER

Exercici 1: drivers para calculaConsumo()

Añadimos un nuevo módulo (**File→New→Module...**) a nuestro proyecto IntelliJ.

- **Name:** gestorLlamadasMocks
- **Location:** "\$HOME/ppss-2023-Gx.../P05-Dependencias2/P05-mocks/"
- Seleccionamos **Java**, y **Maven**, y nos aseguramos de elegir el **JDK 11**
- Desmarcamos la casilla **Add sample code**
- Desde **Advanced Settings**, **GroupId:** ppss.P05; **ArtifactId:** gestorLlamadasMocks

Queremos automatizar las pruebas unitarias sobre el método **GestorLlamadas.calculaConsumo()** (que pertenece al paquete **ppss**) utilizando verificación basada en el comportamiento.

A continuación indicamos el código de nuestra SUT, y los casos de prueba que queremos automatizar.

```
public class GestorLlamadas {
    private static final double TARIFA_NOCTURNA=10.5;
    private static final double TARIFA_DIURNA=20.8;

    public Calendario getCalendario() {
        Calendario c = new Calendario();
        return c;
    }

    public double calculaConsumo(int minutos) {
        Calendario c = getCalendario();*
        int hora = c.getHoraActual();*
        if(hora < 8 || hora > 20){*
            return minutos * TARIFA_NOCTURNA;
        } else {*
            return minutos * TARIFA_DIURNA;
        }
    }
}
```

	minutos	hora	Resultado esperado
C1	22	10	457,6
C2	13	21	136,5

```
public class Calendario {
    public int getHoraActual() {
        throw new
            UnsupportedOperationException
            ("Not yet implemented");
    }
}
```

① Detectamos las dependencias

* Dependencia externa de la misma clase debemos hacer un **partialMock** que será nuestro **testable**

* Dependencia externa a nuestra clase debemos hacer un **mock** del objeto, además hacer una expectativa para el método

	minutos	hora	Resultado esperado
C1	22	10	457,6
C2	13	21	136,5

② Como usamos el framework no hace falta hacer clases externas

public Class GestorLlamadasTestable {

@Test

void CalculaConsumoC1() {

//Creamos el control ya que tenemos varias dependencias

IMocksControl ctrl = EasyMock.createStrictControl();

//Creamos el mock del calendario

Calendario mock = ctrl.mock(Calendario.class);

//Creamos el testable de nuestra clase

GestorLlamadas testable = EasyMock.partialMockBuilder(GestorLlamadas.class).addMockedMethod("getCalendario").mock(ctrl);

//Programamos la expectativa para devolver nuestro Mock

EasyMock.expect(testable.getCalendario()).andReturn(mock);

//Definimos las entradas y salidas esperadas

int minutos = 22;

int hora = 10;

double salidaEsperada = 457,6;

//Definimos la salida que va a tener nuestro mock cuando le llamén

EasyMock.expect(mock.getHoraActual()).andReturn(hora);

//Indicamos que nuestras expectativas y mocks ya están listos

ctrl.replay();

//Haremos los asserts que necesitemos

Assert.assertEquals(salidaEsperada, testable.calculaConsumo(minutos));

//Realizaremos verificación

ctrl.verify();

② Test

```
void CalculaConsumoGz () {  
    // Hacemos el control  
    IMockControl ctrl = EasyMock.createStrictControl();  
    // Hacemos el mock de nuestro calendario  
    Calendario mock = ctrl.mock(Calendario.class);  
    // Hacemos el mock parcial es decir nuestra clase testable  
    GestorLlamadas testable = EasyMock.partialMockBuilder(GestorLlamadas.class).addMockedMethod("getHoraActual").  
        .mock(ctrl);  
  
    // Programamos la expectativa para que se devuelva nuestro Mock.  
    EasyMock.expect(testable.getCalendario()).andReturn(mock);  
    // Declaramos entradas y salidas  
    int minutos = 22;  
    int hora = 10;  
    double esperado = 457,6;  
    // Programamos la expectativa del mock  
    EasyMock.expect(mock.getHoraActual()).andReturn(hora);  
    // Indicamos que nuestros mocks y expectativas están listas  
    ctrl.replay();  
    // Hacemos los asserts necesarios  
    assertEquals(esperado, testable.calculaConsumo(minutos));  
    // Hacemos el verify  
    ctrl.verify();  
}
```

	minutos	hora	Resultado esperado
C1	22	10	457,6
C2	13	21	136,5

▷ Ejercicio 2: drivers para compruebaPremio()

Añadimos un nuevo módulo (*File→New→Module...*) a nuestro proyecto IntelliJ.

- **Name:** premio
- **Location:** "\$HOME/ppss-2023-Gx.../P05-Dependencias2/P05-mocks/"
- Seleccionamos Java, y Maven, y nos aseguramos de elegir el **JDK 11**
- **Parent:** <none>
- Desmarcamos la casilla *Add sample code*
- Desde **Advanced Settings**, **GroupId:** ppss.P05; **ArtifactId:** premio

El código de nuestra SUT es el método **ppss.Premio.compruebaPremio()**

```
public class Premio {  
    private static final float PROBABILIDAD_PREMIO = 0.1f;  
    public Random generador = new Random(System.currentTimeMillis());  
    public ClienteWebService cliente = new ClienteWebService();  
  
    public String compruebaPremio() {  
        if(generaNumero() < PROBABILIDAD_PREMIO) {  
            try {  
                String premio = cliente.obtenerPremio();  
                return "Premiado con " + premio;  
            } catch(ClienteWebServiceException e) {  
                return "No se ha podido obtener el premio";  
            }  
        } else {  
            return "Sin premio";  
        }  
    }  
  
    // Genera numero aleatorio entre 0 y 1  
    public float generaNumero() {  
        return generador.nextFloat();  
    }  
}
```

Se trata de implementar los siguientes tests unitarios sobre el método anterior, utilizando verificación basada en el **comportamiento**:

- A) el número aleatorio generado es de 0,07, el servicio de consulta del premio (método `obtenerPremio`) devuelve "entrada final Champions", y el resultado esperado es "Premiado con entrada final Champions"
- B) el número aleatorio generado es de 0,03, el servicio de consulta del premio (método `obtenerPremio`) devuelve una excepción de tipo `ClientWebServiceException`, y el resultado esperado es "No se ha podido obtener el premio"
- C) el número aleatorio generado es de 0,3 y el resultado esperado es: "Sin premio"

```
public class PremioTest {  
    Premio testable;  
    ClienteWebService mock;  
    IMockControl ctrl;  
    @BeforeEach  
    void setup() {  
        mock = EasyMock.createStrictControl();  
        testable = EasyMock.partialMockBuilder(Premio.class).addMockedMethod("generaNumero").mock(ctrl);  
        mock = ctrl.mock(ClienteWebService.class);  
        testable.cliente = mock;  
    }
```

@Test

```
void PremioC1 () {
```

// Definimos las entradas y salidas

```
float numeroGenerado = 0,07;
```

```
String devuelve = "entrada final Champions";
```

```
String salidaEsperada = "Premiado con entrada final Champions";
```

// Injectamos las expectativas

```
EasyMock.expect(testable.generaNumero()).andReturn(numeroGenerado);
```

// Deberemos usar un assertDoesNotThrow porque esta dentro del try

```
AssertDoesNotThrow(() → EasyMock.expect(mock.class).andReturn(devuelve));
```

// Indicamos que los mock ya estan listos

```
ctrl.replay();
```

// Hacemos los asserts

```
assertEquals(salidaEsperada, testable.compruebaPremio());
```

// Hacemos el verify

```
ctrl.verify();
```

① Detectamos las dependencias

* Dependencia externa de la misma clase tenemos que hacer un **partialMock**

* Dependencia externa de la que tenemos que hacer un **mock**, al ser el atributo público podemos injectar sin **setters** ni **getters**

@Test

void PremioC2 () {

// Definimos las entradas y salidas

float numeroGenerado = 0,03;

String salidaEsperada = "No se ha podido obtener el premio";

// Injectamos las expectativas

EasyMock.expect(testable.generaNumero()).andReturn(numeroGenerado);

// Deberemos usar un assertDoesNotThrow porque esta dentro del try

AssertDoesNotThrow().→EasyMock.expect(mock.class).andThrow(new ClientWebServiceException());

// Indicamos que los mock ya estan listos

ctrl.replay();

// Hacemos los asserts

ClientWebServiceException obtenida = testable.compruebaPremio();

assertEquals(salidaEsperada, obtenida.getMessage());

// Hacemos el verify

ctrl.verify();

C) el número aleatorio generado es de 0,3 y el resultado esperado es: "Sin premio"

@Test

void PremioC3 () {

// Programamos entradas y salidas

float numeroGenerado = 0,3

String salidaEsperada = "Sin Premio";

// Injectamos las entradas

EasyMock.expect(testable.generaNumero()).andReturn(numeroGenerado);

// Indicamos que los mocks ya estan listos

ctrl.replay();

// Hacemos los asserts

assertEquals(salidaEsperada, testable.compruebaPremio());

// Hacemos el verify

ctrl.verify();

Exercici 3: drivers para contarCaracteres()

Añadimos un nuevo módulo (**File→New→Module...**) a nuestro proyecto IntelliJ.

- **Name:** ficheroTexto
- **Location:** "\$HOME/ppss-2023-Gx.../P05-Dependencias2/P05-mocks/"
- Seleccionamos **Java**, y **Maven**, y nos aseguramos de elegir el **JDK 11**
- **Parent:** <none>
- Desmarcamos la casilla **Add sample code**
- Desde **Advanced Settings**, **GroupId:** ppss.P05; **ArtifactId:** ficheroTexto

Queremos automatizar la ejecución de los siguientes casos de prueba asociados al método **ppss.FicheroTexto.contarCaracteres()**. La especificación es la misma que la de la práctica P02

	Datos de entrada		Resultado esperado	
	nombreFichero	[read()...read()]	close()	int o FicheroException
C1	src/test/resources/ficheroC1.txt	{a,b,IOException}	--	FicheroException con mensaje "src/test/resources/ficheroC1.txt (Error al leer el archivo)"
C2	src/test/resources/ficheroC2.txt	{a,b,c,-1}	IOException	FicheroException con mensaje "src/test/resources/ficheroC2.txt (Error al cerrar el archivo)"

ficheroC1.txt contiene los caracteres **abcd**

ficheroC2.txt contiene los caracteres **abc**

El código es el siguiente:

```
public class FicheroTexto {
    public int contarCaracteres(String nombreFichero) throws FicheroException {
        int contador = 0;
        FileReader fichero = null;
        try {
            fichero = new FileReader(nombreFichero);
            int i=0;
            while (i != -1) {
                i = fichero.read();
                contador++;
            }
            contador--;
        } catch (FileNotFoundException e1) {
            throw new FicheroException(nombreFichero +
                " (No existe el archivo o el directorio)");
        } catch (IOException e2) {
            throw new FicheroException(nombreFichero +
                " (Error al leer el archivo)");
        }
        try {
            System.out.println("Antes de cerrar el fichero");
            fichero.close();
        } catch (IOException e) {
            throw new FicheroException(nombreFichero +
                " (Error al cerrar el archivo)");
        }
        return contador;
    }
}
```

Necesitarás crear la clase **FicheroException**:

* El párrafo final nos dice que si necesitamos refactorizar no podemos hacer uso ni de atributos ni de clases factory por lo que tocará usar una factory local

Queremos implementar drivers para automatizar los casos de prueba anteriores usando verificación basada en el comportamiento. Debes tener en cuenta que **no podemos alterar** en modo alguno la invocación a nuestra unidad desde otras unidades, **ni tampoco podemos añadir** ningún atributo en la clase de nuestro SUT ni añadir clases adicionales en src/main/java.

① Refactorizamos

```
public class FicheroTexto | *
    public FileReader getFileReader ( String nombreFichero) throws FileNotFoundException |
        return new FileReader (nombreFichero);
|
```

```
public int contarCaracteres(String nombreFichero) throws FicheroException {
    int contador = 0;
    * FileReader fichero = null;
    try {
        fichero = getFileReader(nombreFichero);
        int i=0;
        while (i != -1) {
            i = fichero.read();
            contador++;
        }
        contador--;
    } catch (FileNotFoundException e1) {
        throw new FicheroException(nombreFichero +
            " (No existe el archivo o el directorio)");
    } catch (IOException e2) {
        throw new FicheroException(nombreFichero +
            " (Error al leer el archivo)");
    }
    try {
        System.out.println("Antes de cerrar el fichero");
        fichero.close();
    } catch (IOException e) {
        throw new FicheroException(nombreFichero +
            " (Error al cerrar el archivo)");
    }
    return contador;
}
```

② * Necesitaremos un partialMockbuilder para el método **getFileReader**
* Necesitaremos un mock para el objeto **FileReader**;

```
public class FicheroTextoTest {
```

② Test

```
void FicheroTextoC() {
```

// Creamos el control

```
IMockControl ctrl = EasyMock.createStrictControl();
```

// Creamos nuestro mock

```
FileReader mock = ctrl.mock(FileReader.class);
```

// Creamos nuestro testable

```
FicheroTexto testable = EasyMock.partialMockBuilder(FicheroTexto.class).addMockedMethod(..  
("get.FileReader")).mock(ctrl);
```

// Programamos entradas y salidas

```
String nombreFichero = "src/test/resources/ficheroC1.txt"
```

```
String salidaEsperada = nombreFichero + "Error al leer el archivo";
```

// Programamos expectativas

```
AssertDoesNotThrow(() -> EasyMock.expect(testable.get.FileReader()).andReturn(mock));
```

```
AssertDoesNotThrow(() -> EasyMock.expect(mock.read()).andReturn((int)'a').andReturn  
(int)'b').andThrow(new IOException());
```

// Hacemos el replay mocks listos

```
ctrl.replay();
```

// Hacemos los assert

```
FicheroException ex = AssertThrows(FicheroException.class, testable.contarCaracteres(nombreFichero));  
assertEquals(salidaEsperada, ex.getMessage());
```

// Hacemos el verify

```
ctrl.verify();
```

② Test

```
void FicheroTextoC() {
```

// Creamos el control

```
IMockControl ctrl = EasyMock.createStrictControl();
```

// Creamos nuestro mock

```
FileReader mock = ctrl.mock(FileReader.class);
```

// Creamos nuestro testable

```
FicheroTexto testable = EasyMock.partialMockBuilder(FicheroTexto.class).addMockedMethod(..  
("get.FileReader")).mock(ctrl);
```

// Programamos entradas y salidas

```
String nombreFichero = "src/test/resources/ficheroC2.txt"
```

```
String salidaEsperada = nombreFichero + "Error al cerrar el archivo";
```

// Programamos expectativas

```
AssertDoesNotThrow(() -> EasyMock.expect(testable.get.FileReader()).andReturn(mock));
```

```
AssertDoesNotThrow(() -> EasyMock.expect(mock.read()).andReturn((int)'a').andReturn  
(int)'b').andReturn((int)'c').andReturn(-1));
```

```
AssertDoesNotThrow(() -> testable.close()); // Devuelve void hay que hacerlo así
```

```
EasyMock.expectLastCall().andThrow(new IOException());
```

// Hacemos el replay mocks listos

```
ctrl.replay();
```

// Hacemos los assert

```
FicheroException ex = AssertThrows(FicheroException.class, testable.contarCaracteres(nombreFichero));  
assertEquals(salidaEsperada, ex.getMessage());
```

// Hacemos el verify

```
ctrl.verify();
```

	Datos de entrada			Resultado esperado
	nombreFichero	[read(),...read()]	close()	int o FicheroException
C1	src/test/resources/ficheroC1.txt	{a,b,IOException}	..	FicheroException con mensaje "src/test/resources/ficheroC1.txt (Error al leer el archivo)"
C2	src/test/resources/ficheroC2.txt	{a,b,c,-1}	IOException	FicheroException con mensaje "src/test/resources/ficheroC2.txt (Error al cerrar el archivo)"

Ejercicio 4: drivers para reserva()

Añadimos un nuevo módulo (**File→New→Module...**) a nuestro proyecto IntelliJ.

- **Name:** *reservaMocks*
- **Location:** *\$/HOME/psss-2023-Gx-/P05-Dependencias2/P05-mocks/*
- Seleccionamos **Java**, y **Maven**, y nos aseguramos de elegir el **JDK 11**
- **Parent:** <none>
- Desmarcamos la casilla *Add sample code*
- Desde *Advanced Settings*, **GroupId:** *psss.P05*; **ArtifactId:** *reservaMocks*

Proporcionamos el código del método **ppss.Reserva.realizaReserva()**, así como la tabla de casos de prueba asociada.

La especificación es similar a la de práctica anterior, excepto que la completamos indicando que si alguien diferente del bibliotecario intenta hacer la reserva el método devuelve la excepción *ReservaException* con el mensaje "ERROR de permisos".

Código del método **ppss.Reserva.realizaReserva()**:

```
public class Reserva {  
  
    public boolean compruebaPermisos(String login, String password, Usuario tipoUsu) {  
        throw new UnsupportedOperationException("Not yet implemented");  
    }  
  
    public void realizaReserva(String login, String password,  
                               String socio, String [] isbn) throws ReservaException {  
        ArrayList<String> errores = new ArrayList<String>();  
        if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {  
            errores.add("ERROR de permisos");  
        } else {  
            FactoriaBOs fd = new FactoriaBOs();  
            IOperacionBO io = fd.getOperacionBO();  
            try {  
                for(String isbn: isbn) {  
                    try {  
                        io.operacionReserva(socio, isbn);  
                    } catch (ISBNInvalidoException iie) {  
                        errores.add("ISBN invalido" + ":" + isbn);  
                    }  
                }  
            } catch (SocioInvalidoException sie) {  
                errores.add("SOCIO invalido");  
            } catch (JDBCEXception je) {  
                errores.add("CONEXIÓN invalida");  
            }  
        }  
        if (errores.size() > 0) {  
            String mensajeError = "";  
            for(String error: errores) {  
                mensajeError += error + "; ";  
            }  
            throw new ReservaException(mensajeError);  
        }  
    }  
}
```

Esto es un strictMock varias llamadas a una misma dependencia

Las excepciones debes implementarlas en el paquete *ppss.excepciones*

La definición de la interfaz **IOperacionBO** y del **tipo enumerado** son las mismas que en la práctica anterior.

La definición de la clase **FactoriaBOs** es la siguiente

```
public class FactoriaBOs {  
    public IOperacionBO getOperacionBO(){  
        throw new UnsupportedOperationException("Not yet implemented");  
    }  
}
```

Tabla de casos de prueba:

	login	password	ident. socio	isbns	{reserva()}	Resultado esperado
C1	"xxxx"	"xxxx"	"Pepe"	{"33333"}	--	ReservaException1
C2	"ppss"	"ppss"	"Pepe"	{"22222", "33333"}	{NoExcep, NoExcep.}	No se lanza excep.
C3	"ppss"	"ppss"	"Pepe"	{"11111", "22222", "55555"}	{IsbnEx, NoExcep, IsbnEx}	ReservaException2
C4	"ppss"	"ppss"	"Luis"	{"22222"}	{SocioEx}	ReservaException3
C5	"ppss"	"ppss"	"Pepe"	{"11111", "22222", "33333"}	{IsbnEx, NoExcep, JDBCEx}	ReservaException4

ReservaException1: Excepción de tipo ReservaException con el mensaje: "ERROR de permisos;"

ReservaException2: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:11111; ISBN invalido:55555;"

ReservaException3: Excepción de tipo ReservaException con el mensaje: "SOCIO invalido;"

ReservaException4: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:11111; CONEXION invalida;"

Nota: Suponemos que el login/password del bibliotecario es "ppss"/"ppss"; que "Pepe" es un socio y "Luis" no lo es; y que los isbn registrados en la base de datos son "22222", "33333".

Tienes que tener en cuenta que si necesitas refactorizar **no podemos alterar** en modo alguno la **invocación a nuestra unidad desde otras unidades, ni tampoco podemos añadir** ninguna factoría local, **ni añadir código adicional** en el directorio de fuentes, **ni añadir/modificar ningún constructor, ni incluir atributos que no sean "private"**.

A partir de la información anterior, y utilizando la librería EasyMock, se pide lo siguiente:

- A) Implementa los drivers utilizando verificación basada en el **comportamiento**. La clase que contiene los tests se llamará *ReservaMockTest.java*
- B) Implementa los drivers utilizando verificación basada en el **estado**. La clase que contiene los tests se llamará *ReservaStubTest.java*

① Identificamos las dependencias

- * Dependencia externa en la misma clase por lo que hay que hacer un **partial Mock**
- * Dependencia externa de la que habrá que hacer un mock para modificar el comportamiento de los métodos a los que llama.
- * Dependencia externa de la que habrá que hacer un mock para modificar el comportamiento de los métodos a los que llama.

② Refactorizar la sut

Según pone en el texto no podemos usar una clase factoría, ni una factoría local, ni modificar la llamada a la sut por lo que deberemos usar un atributo privado y un setter

②

```

public class Reserva {
    private FactoriaB0s fac = New FactoriaB0s();
    public void setFactoriaB0s ( FactoriaB0s facto) {
        fac = facto;
    }

    public boolean compruebaPermisos(String login, String password, Usuario tipoUsu) {
        throw new UnsupportedOperationException("Not yet implemented");
    }

    public void realizaReserva(String login, String password,
                               String socio, String [] isbn) throws ReservaException {
        ArrayList<String> errores = new ArrayList<String>();
        if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
            errores.add("ERROR de permisos");
        } else {
            FactoriaB0s fd = fac;
            IOperacionB0 io = fd.getOperacionB0();
            try {
                for(String isbn: isbn) {
                    try {
                        io.operacionReserva(socio, isbn);
                    } catch (IsbnInvalidoException iie) {
                        errores.add("ISBN invalido" + ":" + isbn);
                    }
                }
                } catch (SocioInvalidoException sie) {
                    errores.add("SOCIO invalido");
                } catch (JDBCEception je) {
                    errores.add("CONEXION invalida");
                }
            }
        if (errores.size() > 0) {
            String mensajeError = "";
            for(String error: errores) {
                mensajeError += error + "; ";
            }
            throw new ReservaException(mensajeError);
        }
    }
}

```

③ Empezamos hacer los test

```

public class ReservaTest {
    Reserva reservaTestable;
    FactoriaB0s mockFactoria;
    IOperacion mockOperacion;
    IMockControl ctrl;
}

```

@BeforeEach

void setup ()

ctrl = EasyMock.createStrictControl();

reservaTestable = EasyMock.partialMockBuilder(Reserva.class).addMockedMethod("compruebaPermisos").
mock(ctrl);

MockFactoria = ctrl.mock(FactoriaB0s.class);

mockOperacion = ctrl.mock(IOperacion.class);

// Injectamos la factoria con el setter

reservaTestable.setFactoriaB0s(mockFactoria);

{}

@Test

```
void ReservaC1()
```

// Definimos entradas y salidas

```
String login = "xxxx";
```

```
String password = "xxxx";
```

```
String idenUsuario = "Pepe";
```

```
String ISBNs[] = {"33333"};
```

```
String esperada = "ERROR de permisos";
```

// Definimos las expectativas

```
EasyMock.expect(reservaTestable.compruebaPermisos(login, password, Usuario.Bibliotecario)).  
andReturn(false);
```

// Hacemos el replay

```
ctrl.replay();
```

// Hacemos los asserts

```
ReservaException ex = AssertThrows(ReservaException.class, () -> reservaTestable.realizaReserva(  
login, password, idenSocio, ISBNs));
```

```
assertEquals(esperada, ex.getMessage());
```

```
ctrl.verify();
```

@Test

```
void ReservaC2()
```

// Definimos entradas y salidas

```
String login = "ppss";
```

```
String password = "ppss";
```

```
String idenUsuario = "Pepe";
```

```
String ISBNs[] = {"zzzzz", "33333"};
```

```
String esperada = "";
```

// Definimos las expectativas

```
EasyMock.expect(reservaTestable.compruebaPermisos(login, password, Usuario.Bibliotecario)).  
andReturn(false);
```

```
EasyMock.expect(mockFactoria.getOperacion()).andReturn(mockOperacion);
```

```
AssertDoesNotThrow(() -> mockFactoria.operacionReserva(idenUsuario, ISBNs[0]));
```

```
EasyMock.expectLastCall().andVoid();
```

```
AssertDoesNotThrow(() -> mockFactoria.operacionReserva(idenUsuario, ISBNs[1]));
```

```
EasyMock.expectLastCall().andVoid();
```

// Hacemos el replay

```
ctrl.replay();
```

// Hacemos los asserts

```
AssertDoesNotThrow(() -> reservaTestable.realizaReserva(login, password, idenSocio, ISBNs));
```

```
ctrl.verify();
```

```
}
```

Mismos test pero con verificación basada en Estado

```
public class ReservaStubsTest {
    Reserva reservaTestable;
    FactoriaBOs factoriaBOStub;
    IOperacionBO operacionStub;
    @BeforeEach
    void setup() {
        reservaTestable = EasyMock.partialMockBuilder(Reserva.class).addMockedMethod("compruebaPermisos").niceMock();
        factoriaBOStub = EasyMock.niceMock(FactoriaBOs.class);
        operacionStub = EasyMock.niceMock(IOperacion.class);
        reservaTestable.setFactoriaBOs(factoriaBOStub);
    }
    @Test
    void ReservaC1() {
        // Definimos entradas y salidas
        String login = "xxxx";
        String password = "xxxx";
        String idenUsuario = "Pepe";
        String ISBNs[] = {"33333"};
        String esperada = "ERROR de permisos";
        // Definimos las expectativas
        EasyMock.expect(reservaTestable.compruebaPermisos(login, password, Usuario.Bibliotecario)).andReturn(false);
        // Hacemos el replay
        EasyMock.replay(reservaTestable);
        // Hacemos los asserts
        ReservaException ex = AssertThrows(ReservaException.class, () -> reservaTestable.realizaReserva(
            login, password, idenUsuario, ISBNs));
        assertEquals(esperada, ex.getMessage());
        EasyMock.verify(reservaTestable);
    }
}
```

② Test

```
void ReservaC2 () {
    // Definimos entradas y salidas
    String login = "ppss";
    String password = "pprs";
    String idenUsuario = "Pepe";
    String ISBNs [] = {"22222", "33333"};
    String esperada = "/";

    // Definimos las expectativas
    EasyMock.expect(reservaTestable.compruebaPermisos(login, password, Usuario.Bibliotecario))
        .andReturn(false);

    EasyMock.expect(factoriaBOStub.getOperacion()).andReturn(operacionStub);
    AssertDoesNotThrow(() -> operacionStub.operacionReserva(idenUsuario, ISBNs[0]));
    EasyMock.expectLastCall().andVoid();
    AssertDoesNotThrow(() -> operacionStub.operacionReserva(idenUsuario, ISBNs[1]));
    EasyMock.expectLastCall().andVoid();

    // Hacemos el replay
    EasyMock.replay(reservaTestable, factoriaBOStub, operacionStub);

    // Hacemos los asserts
    AssertDoesNotThrow(() -> reservaTestable.realizaReserva(login, password, idenSocio, ISBNs));
    EasyMock.verify(reservaTestable, factoriaBOStub, operacionStub);
}
```