P05- Dependencias externas 2: mocks

Dependencias externas

En esta sesión implementaremos de nuevo drivers para realizar pruebas unitarias, pero en esta ocasión realizaremos una **verificación basada en comportamiento**. En este tipo de verificación el doble puede provocar que el test falle, ya que éste realiza comprobaciones para ver si la interacción de nuestra unidad con el doble es la esperada.

Ya hemos trabajado con un tipo de doble, al que hemos denominado **STUB**. Un stub controla las entradas indirectas de nuestro SUT, y se usa para realizar una **verificación basada en el estado**. En este caso, el informe de pruebas depende única y exclusivamente de si el resultado real obtenido al ejecutar nuestro SUT de forma aislada, coincide con el resultado esperado.

Ahora practicaremos con otro tipo de doble: un **MOCK**. Cuyo propósito es diferente de un stub, y por lo tanto, su implementación también lo es. Un mock es un punto de observación de las salidas indirectas de nuestro SUT, y además, y esto es importante, registra TODAS las interacciones de nuestro SUT con el doble, de forma que si el doble no es usado por nuestro SUT de la forma esperada (se llama al doble un cierto número de veces, en un determinado orden, y con unos parámetros concretos, que hemos "programado" previamente), entonces el test fallará, con independencia de que el resultado real de la ejecución de nuestro SUT coincida o no con el esperado. Por lo tanto, un mock puede hacer que nuestro test falle, mientras que un STUB nunca va a ser la causa de un informe fallido de pruebas (ya que no importa cómo interacciona nuestro SUT con el stub, ni siquiera si es invocado o no desde el SUT). Por otro lado, la implementación de un mock requerirá más líneas de código, puesto que no solamente tiene que devolver un cierto valor, sino que tiene que registrar todas las interacciones con nuestro SUT y hacer las comprobaciones oportunas que podrán provocar un fallo en nuestro informe de pruebas. Los mocks permiten realizar una **verificación basada en el comportamiento**.

Dado que la implementación de un mock no es tan simple como la de un stub, vamos a usar una **librería**, que nos permitirá crear cada doble de forma dinámica cuando estemos ejecutando nuestro test, usando un API java. La librería que vamos a usar es **EasyMock**. Y nos va a permitir implementar tanto *mocks* como *stubs*. Por lo tanto también vamos a poder usar la librería EasyMock para implementar drivers con una verificación basada en el estado.

El proceso para implementar los drivers es el MISMO, con independencia del tipo de verificación que se realice. Así, en cualquier caso tendremos que:

1. Identificar las **dependencias externas**, que serán reemplazadas por sus dobles durante las pruebas.
2. Conseguir que nuestro SUT sea testable (sólo hay que refactorizar si es estrictamente necesario)
3. Implementar los dobles
4. Implementar el driver (realizando un tipo de verificación u otro)

Recuerda que el objetivo no es solamente practicar con el framework, sino entender bien las diferencias entre los dos tipos de verificaciones de nuestros drivers, así como el papel de los dobles usados en ambos. La realización de los ejercicios también contribuirá a reforzar vuestros conocimientos sobre maven, y cómo se organiza nuestro código cuando usamos este tipo de proyectos .

Para implementar los drivers usaremos JUnit5 y EasyMock.. Para ejecutarlos usaremos Maven, y Junit5.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P05-Dependencias2**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2023-Gx-apellido1-apellido2.

Ejercicios

En esta sesión también vamos a crear un proyecto IntelliJ **vacío**, e iremos añadiendo los módulos (proyectos Maven), en cada uno de los ejercicios.

Para **crear el proyecto IntelliJ**, tendremos que (es el mismo proceso que en la práctica anterior):

- ***File→New Project***. Seleccionamos "Empty Project"
- ***Project name*** : "P05-mocks". ***Project Location:*** "$HOME/ppss-2023-Gx-.../P05-Dependencias2/".

Recuerda que debes **ELIMINAR** el módulo *P05-mocks* y el fichero *P05-mocks.iml*

NOTA: Cuando crees cada uno de los módulos maven debes **modificar el pom** convenientemente para cada uno de ellos.

OBSERVACIONES SOBRE LA IMPLEMENTACIÓN:

- Si realizamos una verificación basada en el ***comportamiento***, SIEMPRRE nos va a importar el orden en el que los dobles van a ser invocados desde la SUT (independientemente de si dichos dobles pertenecen a la misma clase o a varias).

  Si necesitas incluir un "mock parcial" en un "StrictControl", incluye el "StrictControl" como parámetro del método para crear la clase que contiene nuestros dobles.

```
ctrl = EasyMock.createStrictControl();
partialMock = EasyMock.partialMockBuilder(...)
                .addMockedMethod(...)
                .mock(ctrl);
```

- Si al programar ***varias expectativas de un doble*** necesitas capturar una excepción, usa un único *assertDoesNotThrow*, e incluye las expectativas en la expresión lambda:

```
//doble() es un método void que puede lanzar una excepción
assertDoesNotThrow(()-> {
    mock1.doble(param1,param2);
    EasyMock.expectLastCall().andThrow(...);
    mock1.doble(param3,param4);
    mock1.doble(param5,param6);
    EasyMock.expectLastCall().andThrow(...);
});
```

- Al programar ***varias expectativas de un stub*** encadena las expectativas (NO uses bucles!!):

```
//stub() es un método void que puede lanzar una excepción
assertDoesNotThrow(()->clase.stub(anyString(), anyString()));
EasyMock.expectLastCall()
        .andThrow(...)
        .andVoid()
        .andThrow(..)
        .andVoid().anyTimes();
```

➪ ➪ Ejercicio 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 fianl 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");
  }
}
```

➪ ➪ 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"

➪ ➪ Ejercicio 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***:

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.

➪ ➪ Ejercicio 4: drivers para *reserva()*

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

- ***Name:*** *reservaMocks*
- ***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:*** *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 [] isbns) 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: isbns) {
              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("CONEXION 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***

Resumen

¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

VERIFICACIÓN BASADA EN EL ESTADO

- El driver de una prueba unitaria puede realizar una verificación basada en el estado resultante de la ejecución de la unidad a probar .
- Si la unidad a probar tiene dependencias externas que necesitemos controlar, éstas serán sustituidas por STUBS durante las pruebas. Los stubs controlarán las entradas indirectas de nuestro SUT. Un stub no puede hacer que nuestro test falle (el resultado del test no depende de la interacción de nuestro SUT con sus dependencias externas)
- Para poder usar los dobles (stubs) en nuestras pruebas, éstos tienen que poder inyectarse en nuestro SUT a través de los "enabling seam points".
- Podemos implementar los dobles de forma "manual" o usando el framework EasyMock.
- Si implementamos los stubs "manualmente", éstos deben NECESARIAMENTE implementarse en una clase separada de la clase que contiene nuestros tests. La clase que contiene la implementación de nuestros dobles tendrá el sufijo "Stub", o "Testable" (este último caso sólo si necesitamos inyectar el stub a través de dicha clase).
- Si usamos el framewkork EasyMock, la implementación del doble necesariamente estará en cada driver. Nuestros dobles estarán implementados en objetos de tipo NiceMock. Tendremos que programar las expectativas de forma que nuestro test no falle si se invoca más veces de lo esperado al doble, o con diferentes parámetros. Finalmente necesitamos indicar al framework que hemos finalizado la programacón de las expectativas (método replay()).

VERIFICACIÓN BASADA EN EL COMPORTAMIENTO

- El driver de una prueba unitaria puede realizar una verificación basada en el comportamiento, de forma que no sólo se tenga en cuenta el resultado real, sino también la interacción de nuestro SUT con sus dependencias externas (cuántas veces se invocan, con qué parámetros, y en un orden determinado).
- Los dobles usados si realizamos una verificación basada en el comportamiento se denominan mocks. Un mock constituye un punto de observación de las salidas indirectas de nuestro SUT, y además registra la interacción del doble con el SUT. Un mock sí puede provocar que el test falle.
- Para poder usar los dobles (mocks), éstos tienen que poder inyectarse en nuestro SUT a través de los "enabling seam points".
- Para implementar los dobles usaremos la librería EasyMock. Para ello tendremos que crear el doble (de tipo Mock, o StrictMock, programar sus expectativas, indicar que el doble ya está listo para ser usado y finalmente verificar la interacción con el SUT
- Si nuestros dobles pertenecen a clases diferentes, necesariamente tendremos que incluirlos en un StrictControl. para que se tenga en cuenta el orden de invocaciones entre las diferentes clases.
- NO usaremos estructuras de control para programar las expectativas (bucles, condiciones...).