P06- 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 "especificado" 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
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.. Para ejecutarlos usaremos Maven, EasyMock, y Junit5.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P06-Dependencias2**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2020-Gx-apellido1-apellido2..

Recuerda que si trabajas desde los ordenadores del laboratorio primero deberás configurar git y clonar tu repositorio de Bitbucket.

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**, simplemente tendremos que realizar lo siguiente:

- ***File→New Project***. A continuación elegimos "Empty Project" y pulsamos sobre Next,
- ***Project name*** : "P06-mocks". ***Project Location:*** *"*$HOME/ppss-2020-Gx-.../P06-Dependencias2/P06-mocks". Es decir, que tenemos que crear el proyecto dentro de nuestro directorio de trabajo, y del directorio P06-Dependencias2.

De forma automática, IntelliJ nos da la posibilidad de añadir un nuevo módulo a nuestro proyecto (desde la ventana ***Project Structure***), antes de crear el proyecto. Cada ejercicio lo haremos en un módulo diferente. Recuerda que CADA módulo es un PROYECTO MAVEN.

Vamos a añadir un primer módulo que usaremos en el Ejercicio1. En la ventana que nos muestra IntelliJ, desde ***Project Settings →module***, pulsamos sobre "***+→New Module***":

- Seleccionamos ***Maven***, y nos aseguramos de elegir el ***JDK 1.8***
- ***GroupId***: "*ppss*"; ***ArtifactId:*** *"gestorLlamadasMocks*".
- ***ModuleName***: "*gestorLlamadasMocks*". ***Content Root:*** *"$HOME/ppss-2020-Gx-.../P06-Dependencias2/P06-mocks/gestorLlamadasMocks*". ***Module file location***: *"$HOME/ppss-2020-Gx-.../P06-Dependencias2/P06-mocks/gestorLlamadasMocks*".

Finalmente pulsamos sobre OK.

➪ ➪ Ejercicio 1: *drivers* para `calculaConsumo()`

Una vez que hemos creado el **módulo gestorLlamadasMocks** en nuestro proyecto IntelliJ, vamos a usarlo para hacer este ejercicio.

Se trata de 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 nuestro SUT, y los casos de prueba que queremos automatizar.

```
//paquete ppss

public class GestorLlamadas {
  static double TARIFA_NOCTURNA=10.5;
  static 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

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

➪ ➪ Ejercicio 2: drivers para *compruebaPremio()*

Para este ejercicio añadiremos un nuevo módulo (***New→Module…***) a nuestro proyecto IntelliJ. Recuerda que los valores de "***Add as a module to***" y "***Parent***", deben ser *<none>,* el ***groupId*** será "*ppss*" y el valor de ***artifactId*** será "*premio*"). El nombre del módulo será *premio*. Asegúrate de que el ***Content root*** y ***Module file location*** sean: "*$HOME/ppss-2020-Gx-.../P06-Dependencias2/**P06-mocks/premio***"

A continuación mostramos el código del 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 *calculaPrecio()*

Para este ejercicio añadiremos un nuevo módulo (***New→Module...***). El valor de "***Add as a module to***" y "***Parent***", debe ser *<none>,* el ***groupId*** será "*ppss*" y el valor de ***artifactId*** será "*alquiler*"). El nombre del módulo será "alquiler". Asegúrate de que el ***Content root*** y ***Module file location*** sean: "*$HOME/ppss-2020-Gx-.../P06-Dependencias2/**P06-mocks/alquiler***".

Proporcionamos el siguiente código del método ***ppss.Alquilacoches.calculaPrecio()***..

```
1  public class AlquilaCoches {
2    protected Calendario calendario = new Calendario();
3
4    public Ticket calculaPrecio(TipoCoche tipo, LocalDate inicio, int ndias)
5                                                        throws MensajeException {
6
7      Ticket ticket = new Ticket();
8      float precioDia,precioTotal =0.0f;
9      float porcentaje = 0.25f;
10
11    String observaciones = "";
12    IService servicio = new Servicio();
13    precioDia = servicio.consultaPrecio(tipo);
14    for (int i=0; i<ndias;i++) {
15       LocalDate otroDia = inicio.plusDays((long)i);
16       try {
17          if (calendario.es_festivo(otroDia)) {
18            precioTotal += (1+ porcentaje)*precioDia;
19          } else {
20            precioTotal += (1- porcentaje)*precioDia;
21          }
22       } catch (CalendarioException ex) {
23           observaciones += "Error en dia: "+otroDia+"; ";
24       }
25    }
26
27    if (observaciones.length()>0) {
28       throw new MensajeException(observaciones);
29    }
30
31    ticket.setPrecio_final(precioTotal);
32    return ticket;
33   }
34 }
```

```
public class Ticket {
  private float precio_final;
  //getters y setters
}
```

La especificación es la misma que la de la práctica anterior.

Recordemos que el tipo LocalDate representa una fecha y pertenece a la librería estándar de Java. La sentencia de la línea 14 devuelve la fecha resultante de añadir "i" dias a la fecha "inicio".

Podemos obtener una representación de tipo String a partir de un LocalDate, de la siguiente forma:

```
String fechaS = fecha.toString(); //el valor de fechaS es : "aaaa-mm-dd"
```

Podemos crear un objeto de tipo LocalDate a partir de un String mediante:

```
LocalDate fecha = LocalDate.of(2020, Month.MARCH, 2);
```

Las clases Calendario y Servicio están siendo implementadas por otros miembros del equipo.

Tipo coche es un tipo enumerado:

```
public enum TipoCoche {TURISMO,DEPORTIVO,CARAVANA}; //ej. de uso: TipoCoche.TURISMO
```

Queremos automatizar las pruebas unitarias usando verificación basada en el comportamiento, a partir de los siguientes casos de prueba (y teniendo 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):

Asumimos que el precio base para cualquier día es de 10 euros, tanto para caravanas como para turismos.

Id	Datos Entrada				Resultado Esperado
	Tipo	fechaInicio	dias	festivo	Ticket (importe) o MensajeException
C1	TURISMO	2020-05-18	1o	false para todos los días	Ticket (75)
C2	CARAVANA	2020-06-19	7	true solo para los días 20 y 24	Ticket (62,5)
C3	TURISMO	2020-04-17	8	false para todos los días, y lanza excepción en 18, 21, y 22	(“Error en dia: 2020-04-18; Error en dia: 2020-04-21; Error en dia: 2020-04-22; ”)

Nota: el formato de la fecha es "aaaa-mm-dd" (año-mes-dia)

Nota 2: En la implementación de los drivers puedes "relajar" la comprobación del parámetro en la invocación al método es_festivo y usar anyObject() en los tres tests.

Nota 3: Implementa una versión adicional del driver para el caso de prueba C3, al que puedes llamar C3A, en la que no "relajes" la comprobación del valor del parámetro (no uses anyObject() en la invocación del método es_festivo()). Obviamente, necesitarás mas líneas de código para especificar las expectativas de los dobles. Como podrás comprobar, con una verificación basada en el comportamiento en la que importan "todas" las interacciones de nuestro SUT con las dependencias externas (orden de las invocaciones, los valores de los parámetros, y número de invocaciones), provocan que nuestros drivers sean más "frágiles" y dependientes de la implementación, tal y como se ha explicado en la clase de teoría.

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

Para este ejercicio añadiremos un nuevo módulo (***New→Module...***) a nuestro proyecto IntelliJ. El valor de "***Add as a module to***" y "***Parent***", debe ser *<none>,* el ***groupId*** será "*ppss*" y el valor de ***artifactId*** será "*reserva*"). El nombre del módulo será *reserva*. Asegúrate de que el ***Content root*** y ***Module file location*** sean: “*$HOME/ppss-2020-Gx-.../P06-Dependencias2/**P06-mocks/reserva***"

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

La especificación es la misma que la de la práctica anterior, aunque la versión de código sea diferente.

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");
    }
}
```

```
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);
    }
  }
}
```

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

Tabla de casos de prueba:

	login	password	ident. socio	Acceso BD	isbns	Resultado esperado
C1	"xxxx"	"xxxx"	"Pepe"	No se accede a la BD	{"22222"}	ReservaException1
C2	"ppss"	"ppss"	"Pepe"	Acceso a BD sin excepciones	{"22222", "33333"},	No se lanza excep.
C3	"ppss"	"ppss"	"Pepe"	Acceso a BD con ex. "ISBN ..."	{"11111"}	ReservaException2
C4	"ppss"	"ppss"	"Luis"	Acceso a BD con ex. "SOCIO..."	["22222"}	ReservaException3
C5	"ppss"	"ppss"	"Pepe"	Acceso a BD con ex. "CONEX.."	{"22222"}	ReservaException4

ReservaException1: Excepción de tipo ReservaException con el mensaje: "ERROR de permisos; "
ReservaException2: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:11111; "
ReservaException3: Excepción de tipo ReservaException con el mensaje: "SOCIO invalido; "
ReservaException4: Excepción de tipo ReservaException con el mensaje: "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".

Ten 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, tampoco podremos añadir ningún método público en la clase de nuestro SUT.

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, por ejemplo, ReservaMockTest.java

B) Implementa los drivers utilizando verificación basada en el estado. La clase que contiene los tests se llamará, por ejemplo, ReservaStubTest.java

Nota 2: Para la implementación del driver del caso de prueba C2, fíjate que la invocación de nuestra SUT no debe lanzar una excepción. En este caso, para verificar el resultado, y dado que el método realizaReserva() es un método void, usaremos la siguiente sentencia assert:

```
Assertions.assertDoesNotThrow(
        ()-> mockReserva.realizaReserva(login, password, idSocio, isbns)
);
```

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, éstas serán sustituidas por stubs durante las pruebas. Los dobles 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), éstos tienen que poder inyectarse en nuestro SUT a través de los "enabling seam points".

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
- EasyMock también nos permite implementar drivers usando verificación basada en el estado (usando stubs). Para ello tendremos que crear el doble (de tipo NiceMock), programar sus expectativas (relajando los valores de los parámetros) e indicar que el doble ya está listo para ser usado por nuestro SUT".