P06- Dependencias externas 2: mocks

Dependencias externas

En esta sesión implementaremos de nuevo drivers para realizar pruebas unitarias. Ya hemos visto que cuando nuestra unidad tiene dependencias externas, tenemos que sustituirlas por sus dobles durante las pruebas con el fin de tener el control sobre su ejecución y así poder aislar la unidad a probar.

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, nuestro 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 al usar unos valores concretos como entradas. También hemos visto cómo implementar dicho doble.

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 controla 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. Así pues, los mocks permiten que nuestro driver realice otro tipo de verificación, a la que hemos denominado **verificación basada en el comportamiento** (el resultado del test no solo depende de la comparación del resultado real con el esperado, sino que depende también de cómo interacciona nuestro SUT con el mock).

Dado que la implementación de un mock no es tan simple como la de un stub, vamos a usar una librería, que nos proporcionará una implementación de dicho 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 vamos a ver cómo implementar drivers basados en el estado y en el comportamiento usando la librería EasyMock.

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 .

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-Gx-apellido1-apellido2-2019. Puedes subir, además del código, los ficheros que consideres oportunos, en formato png, jpg, pdf (al margen de que también subas ficheros con extensión doc, xml, u otros formatos particulares dependiendo de las herramientas de edición que uses).

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

Ejercicios

En esta sesión crearemos 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-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:*** *"$HOME/ppss-Gx-.../P06-Dependencias2/P06-mocks/gestorLlamadasMocks*". ***Module file location***: *"$HOME/ppss-Gx-.../P06-Dependencias2/P06-mocks/gestorLlamadasMocks*".

Finalmente pulsamos sobre OK (automáticamente IntelliJ marcará los directorios de nuestro proyecto como directorios estándar de Maven, de forma que "sabrá" cuáles son los directorios de fuentes, de recursos, ...).

➪ ➪ 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-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"

➪ ➪ Ejercicio 3: 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-Gx-.../P06-Dependencias2/**P06-mocks/reserva***"

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

```
public class Reserva {

  protected FactoriaBOs getFactoriaBOs() {
     return new FactoriaBOs();
  }

  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 = getFactoriaBOs();
        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);
     }
  }
}
```

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	Acceso BD	isbns	Resultado esperado
C1	"xxxx"	"xxxx"	"Pepe"	OK	{"22222"}	ReservaException1
C2	"ppss"	"ppss"	"Pepe"	OK	{"22222", "33333"},	No se lanza excep.
C3	"ppss"	"ppss"	"Pepe"	OK	{"11111"}	ReservaException2
C4	"ppss"	"ppss"	"Luis"	OK	["22222"}	ReservaException3
C5	"ppss"	"ppss"	"Pepe"	fallo	{"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".

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