P05- Dependencias externas 1: stubs

Dependencias externas

En esta sesión implementaremos drivers para realizar pruebas unitarias, teniendo en cuenta que las unidades a probar pueden tener dependencias externas, y que necesitaremos controlarlas desde la unidad probada. El objetivo es realizar las pruebas aislando la ejecución de nuestra unidad. De esta forma nos aseguramos de que cualquier defecto que detectemos estará exclusivamente en el código de nuestro SUT, excluyendo así cualquier código del que dependa.

Las dependencias externas tendremos que sustituirlas por sus dobles, concretamente por stubs, por lo que la idea es que el doble controle las entradas indirectas de dicha unidad a nuestra SUT. El doble reemplazará, durante las pruebas, a la dependencia externa real, que se ejecutará en producción.

Recuerda que NO se puede alterar "temporalmente" el código a probar (SUT), pero sí se puede REFACTORIZAR, para que el código contenga un seam (uno por dependencia externa), y que sea posible inyectar el doble durante las pruebas. Es importante que tengas claro que hay diferentes refactorizaciones posibles y que cada una de ellas tiene diferentes "repercusiones" en el código en producción.

El driver que vamos a implementar realiza una verificación basada en el estado, es decir, el resultado del test depende únicamente del resultado de nuestro SUT. Observa que el algoritmo del driver es el mismo que el de sesiones anteriores, pero añadiendo más acciones en la fase de preparación de los datos, ya que tendremos que crear el doble, programar su resultado e inyectarlo en nuestra SUT, antes de ejecutarla. Para implementar los drivers usaremos JUnit5.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica, que no será código, deberá estar en el directorio **P05-Dependencias1**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-Gx-apellido1-apellido2-2019. Puedes subir los ficheros 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 las sesiones anteriores, hemos trabajado con un proyecto IntelliJ con un único módulo (nuestro proyecto Maven). En esta sesión vamos a crear también un proyecto IntelliJ, pero inicialmente estará **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*** : "P05-stubs". ***Project Location:*** "$HOME/ppss-Gx-.../P05-Dependencias1/P05-stubs". Es decir, que tenemos que crear el proyecto dentro de nuestro directorio de trabajo, y del directorio P05-Dependencias1.

De forma automática, IntelliJ nos da la posibilidad de añadir un nuevo módulo a nuestro proyecto (desde la ventana ***Project Structure***), antes de crearlo. Cada ejercicio lo haremos en un módulo diferente.

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:*** *"gestorLlamadas*".
- ***ModuleName***: "*gestorLlamadas*". ***Content Root:*** *"$HOME/ppss-Gx-.../P05-Dependencias1/P05-stubs/gestorLlamadas*". ***Module file location***: *"$HOME/ppss-Gx-.../P05-Dependencias1/P05-stubs/gestorLlamadas*".

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 gestorLlamadas** 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.ejercicio1**) utilizando verificación basada en el estado.

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

```
//paquete ppss.ejercicio1

public class GestorLlamadas {
  static double TARIFA_NOCTURNA=10.5;
  static double TARIFA_DIURNA=20.8;
  public int getHoraActual() {
    Calendar c = Calendar.getInstance();
    int hora = c.get(Calendar.HOUR);
    return hora;
    }

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

	minutos	hora	Resultado esperado
C1	10	15	208
C2	10	22	105

➪ ➪ Ejercicio 2: drivers para `calculaConsumo()`. Versión 2

Utilizando la tabla del ejercicio anterior, automatiza las pruebas unitarias sobre una implementación alternativa de ***GestorLlamadas.calculaConsumo()*** utilizando verificación basada en el estado**.** En este caso, la unidad a probar pertenece al paquete **ppss.ejercicio2,** del **módulo gestorLlamadas** que hemos creado en el ejercicio 1)

En este caso, necesitamos usar también la clase Calendario.

```
//paquete ppss.ejercicio2

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

```
//paquete ppss.ejercicio2
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;
    }
  }
}
```

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

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 "*reserva*"). El nombre del módulo será reserva. Asegúrate de que el ***Content root*** y ***Module file location*** sean: "*$HOME/ppss-Gx-.../P05-Dependencias1/**P05-stubs/reserva***"

```
//paquete ppss
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 Exception {

    ArrayList<String> errores = new ArrayList<>();
    if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
        errores.add("ERROR de permisos");
    } else {
        IOperacionBO io = new Operacion();
      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);
    }
  }
}
```

Proporcionamos el siguiente código:
Las excepciones debes implementarlas en el paquete "**ppss.excepciones**" (por ejemplo):

```
//paquete ppss.excepciones
public class JDBCException extends Exception {
  //no es necesario el constructor, excepto para ReservaException
}
```

```
//paquete ppss.excepciones
public class ReservaException extends Exception {
  public ReservaException(String message) {super(message);}
}
```

Definición de la interfaz (paquete: **ppss**):

```
//paquete ppss
public interface IOperacionBO {
  public void operacionReserva(String socio, String isbn)
              throws IsbnInvalidoException, JDBCException, SocioInvalidoException;
}
```

Definición del tipo enumerado (paquete: **ppss**):

```
//paquete ppss
public enum Usuario {
  BIBLIOTECARIO, ALUMNO, PROFESOR
}
```

Dado el código anterior, se trata de implementar los drivers (usando verificación basada en el estado) para automatizar las pruebas unitarias de la siguiente tabla de casos de prueba.

	login	password	ident. socio	Acceso BD	isbns	Resultado esperado
C1	"xxxx"	"xxxx"	"Luis"	(1)	{"11111"}	ReservaException1
C2	"ppss"	"ppss"	"Luis"	(2)	{"11111", "22222"}	No se lanza excep.
C3	"ppss"	"ppss"	"Luis"	(3)	{"33333"}	ReservaException2
C4	"ppss"	"ppss"	"Pepe"	(4)	["11111"}	ReservaException3
C5	"ppss"	"ppss"	"Luis"	(5)	{"11111"}	ReservaException4

Suponemos que el login/password del bibliotecario es "ppss"/"ppss"; que "Luis" es un socio y "Pepe" no lo es; y que los isbn registrados en la base de datos son "11111", "22222".

ReservaException1: Excepción de tipo ReservaException con el mensaje: "ERROR de permisos; "
ReservaException2: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:33333; "
ReservaException3: Excepción de tipo ReservaException con el mensaje: "SOCIO invalido; "
ReservaException4: Excepción de tipo ReservaException con el mensaje: "CONEXION invalida; "

(1): No se accede a la base de datos
(2): El acceso a la BD NO lanza ninguna excepción
(3): El acceso a la BD lanza la excepción IsbnInvalidoException
(4): El acceso a la BD lanza la excepción SocioInvalidoException
(5): El acceso a la BD lanza la excepción ConexiónInvalidaException