P07- Pruebas de integración

Integración con una base de datos

El objetivo de esta práctica es automatizar pruebas de integración con una base de datos. Para controlar el estado de la base de datos durante los tests utilizaremos la librería DbUnit.

Recordamos que es muy importante, no solamente la implementación de los drivers, sino también dónde se sitúan físicamente dichos drivers en un proyecto maven, cómo configurar correctamente el pom y cómo ejecutar los drivers desde maven. Al fin y al cabo, lo que buscamos es obtener ese informe que nos permita responder a la pregunta inicial: "¿en nuestro SUT hay algún defecto?

En este caso, implementaremos drivers con verificación basada en el estado. En las pruebas de integración ya no estamos interesados en detectar defectos en las unidades, por lo tanto la cuestión fundamental no es tener el control de las dependencia externas (una de las cuales será la base de datos, o mejor dicho, el servidor de la base de datos). Precisamente queremos ejercitar dichas dependencias externas durante las pruebas. Nuestro objetivo es detectar defectos en las "interfaces" de las unidades que forman nuestro SUT. Dichas unidades ya han sido probadas previamente, de forma individual, y por lo tanto, ya hemos detectado defectos "dentro" de las mismas. Es decir, buscamos potenciales problemas en los puntos de "interconexión" de dichas unidades. En definitiva, la cuestión fundamental a contemplar en las pruebas de integración es el orden en el que vamos a integrar las unidades (estrategias de integración).

Dado que no podemos hacer pruebas exhaustivas, es perfectamente posible que durante las pruebas de integración se pongan de manifiesto defectos "dentro" de las unidades correspondientes, es decir, que un defecto que se nos "había pasado por alto" durante las pruebas unitarias, "se manifieste" ahora. Obviamente, habrá que depurarlo, pero debes tener muy claro que nuestro objetivo no es ese.

En la sesión S01, en la primera definición de testing, se indica de forma explícita que la "intención" con la que probamos es fundamental para conseguir nuestro objetivo, lo cual es totalmente lógico. Por lo tanto, cuando realizamos pruebas de integración, y dado que el objetivo es diferente, el conjunto de casos de prueba obtenidos también será diferente (elegiremos conjuntos diferentes de comportamientos a probar), puesto que nuestros esfuerzos estarán centrados en "sacar a la luz" problemas en "líneas de código diferentes"!!! En este caso, como hemos visto en la sesión de teoría, usaremos métodos de diseño de casos de prueba de caja negra, teniendo en cuenta las guías generales de pruebas.

Como siempre decimos: si no se tiene claro QUÉ queremos conseguir (qué problema concreto queremos que resolver), es improbable que las acciones que realicemos (el CÓMO) sean las adecuadas para solucionar dicho problema de la mejor forma y lo más eficientemente posible.

Para implementar los drivers usaremos JUnit5 y Dbunit. 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 **P07-Integracion**, 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.

Base de datos MySql

En la máquina virtual tenéis instalado un servidor de bases de datos MySQL, al que podéis acceder con el usuario "***root***" y password "***ppss***" a través del siguiente comando, desde el terminal:

```
> mysql –u root –p
```

Dicho servidor se pone en marcha automáticamente cuando arrancamos la máquina virtual.

Vamos a usar un ***driver jdbc*** para acceder a nuestra base de datos. Por lo tanto, desde el código, usaremos la siguiente cadena de conexión:

```
jdbc:mysql://localhost:3306/DBUNIT?useSSL=false
```

En donde *localhost* es la máquina donde se está ejecutando el servidor mysql. Dicho proceso se encuentra "escuchando" en el puerto *3306*, y queremos acceder al esquema *DBUNIT* (el cual crearemos automáticamente durante el proceso de construcción usando el plugin sql).

Ejecución de scripts sql con Maven: plugin sql-maven-plugin

Hemos visto en clase que se puede utilizar un plugin de maven (*sql-maven-plugin*) para ejecutar scripts sql sobre una base de datos mysql.

El plugin sql-maven-plugin requiere incluir como dependencia el conector con la base de datos (ya que podemos querer ejecutar el script sobre diferentes tipos de base de datos). En la sección <configuration> tenemos que indicar el driver, la cadena de conexión, login y passord, para poder acceder a la BD y ejecutar el script. Además podemos configurar diferentes *ejecuciones* de las goals del plugin (cada una de ellas estará en una etiqueta <execution>, y tendrá un valor de <id> diferente). Para ejecutar cada una de ellas podemos hacerlo de diferentes formas, teniendo en cuenta que:

- Si hay varias <execution> asociadas a la misma fase y ejecutamos dicha fase maven, se ejecutarán todas ellas
- Si queremos ejecutar solamente una de las <executions>, podemos hacerlo con mvn sql:execute@execution-id, siendo execution-id el identificador de la etiqueta <id> de la <execution> correspondiente
- Si ejecutamos simplemente mvn sql:execute, sin indicar ninguna ejecución en concreto, por defecto se ejecutará aquella que esté identificada como <id>default-cli</id>. Si no hay ninguna <execution> con ese nombre, no se ejecutará nada.

Ejemplo: Dado el siguiente fragmento de código con las <executions> del plugin,

```
...
<executions>
  <execution>
     <id>create-customer-table</id>
     <phase>pre-integration-test</phase>
     <goals>
        <goal>execute</goal>
     </goals>
     <configuration>
        <srcFiles>
            <srcFile>src/test/resources/sql/script1.sql</srcFile>
        </srcFiles>
    </configuration>
  </execution>
  <execution>
    <id>create-customer-table2</id>
    <phase>pre-integration-test</phase>
    <goals>
      <goal>execute</goal>
    </goals>
    <configuration>
      <srcFiles>
        <srcFile>src/test/resources/sql/script2.sql</srcFile>
      </srcFiles>
    </configuration>
  </execution>
</executions>
...
```

- La ejecución ***mvn pre-integration-test*** ejecutará los scripts script1.sql y script2.sql
- La ejecución ***mvn sql:execute@create-customer-table2*** solamente ejecutará el script2.sql
- La ejecución ***mvn sql:execute*** no ejecutará nada

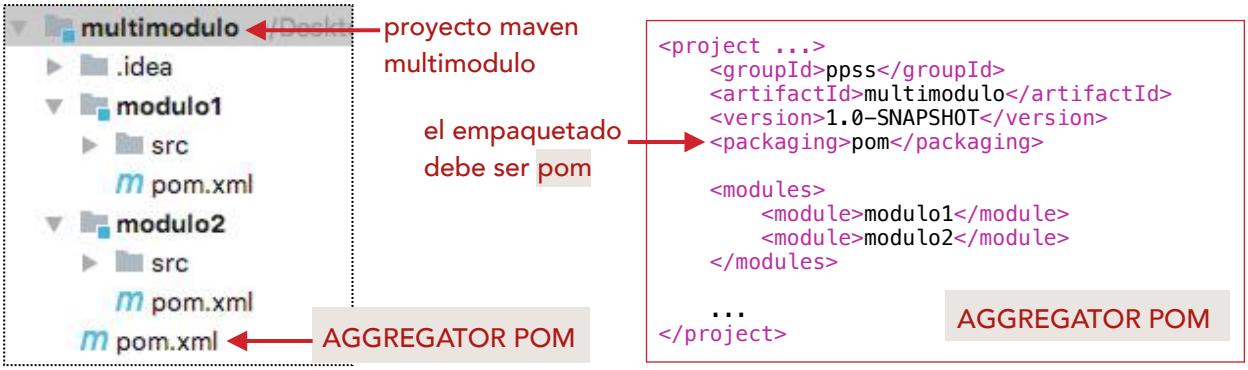
Proyectos Maven multimódulo

Dependiendo de lo "grande" que sea nuestro proyecto software, es habitual "distribuir" el código en varios "subproyectos". Hasta ahora hemos trabajado con proyectos maven independientes, de forma que cada uno contenía "todo" nuestro código.

Maven permite trabajar con proyectos que a su vez están formados por otros proyectos maven, a los que llamaremos "**módulos**", de forma que podemos establecer diferentes "**agrupaciones**" entre ellos, tantas como sean necesarias. Un proyecto Maven que "agrupa" a otros proyectos se denomina proyecto maven multimódulo

Un proyecto **maven multimódulo**, tiene un empaquetado (etiqueta <packaging>) con el valor "pom", y contiene una lista de módulos "agregados" (que son otros proyectos Maven). Por eso a este pom también se le denomina "***aggregator pom***".

A continuación mostramos un ejemplo de un proyecto maven multimódulo que "contiene" dos módulos. Dichos módulos pueden estar físicamente en cualquier ruta de directorios, pero lo más práctico es que los módulos se encuentren físicamente en el directorio del *pom aggregator*



Observa que el proyecto multimódulo sólo contiene el pom.xml, NO tiene código (carpeta src). En el directorio del proyecto añadiremos tantos módulos como queramos agrupar, y lo indicaremos en el pom anidando etiquetas <module>.

Los módulos agregados de nuestro proyecto multimódulo son proyectos maven "normales", y pueden construirse de forma separada o a través del *aggregator pom*.

Un **proyecto multimódulo** se **construye** a partir de su *aggregator* pom, que gestiona al grupo de módulos agregados. Cuando construimos el proyecto a través del *aggregator pom*, cada uno de los proyectos maven agrupados se construyen si tienen un empaquetado distinto de *pom*.

El mecanismo usado por maven para gestionar los proyectos multimódulo recibe el nombre de ***reactor***. Dicho mecanismo se encarga de "recopilar" todos los módulos, los ordena para construirlos en el orden correcto, y finalmente realiza la construcción de todos los módulos en ese orden. El orden en el que se construyen los módulos es importante, ya que éstos pueden tener dependencias entre ellos.

A continuación mostramos un ejemplo de construcción de nuestro proyecto multimódulo. En este caso vamos a ejecutar el comando: *mvn compile*.

Podemos observar que se ha establecido un orden entre los tres módulos: multimodulo, modulo1 y modulo2.

```
> cd multimodulo
> mvn compile
[INFO] Scanning for projects...
[INFO] Reactor Build Order:
[INFO] multimodulo                    [pom]
[INFO] modulo1                        [jar]
[INFO] modulo2                        [jar]
... (continúa en la siguiente página)
```

Y a continuación se ejecuta el comando para todos y cada uno de los módulos en el orden establecido por *reactor*.

Cuando el empaquetado es pom, por defecto solamente tiene asociadas las goals *install:install*, y *deploy:deploy* a las fases *install* y *deploy*, respectivamente. Es decir, que el proyecto que contiene el aggregator pom, no ejecutará ninguna acción en el resto de fases durante el proceso de construcción.

```
[INFO] ------------< ppss:multimodulo >-------------------------
[INFO] Building multimodulo 1.0-SNAPSHOT                  [1/3]
[INFO] ------------------[ pom ]-----------------------------------
[INFO]
[INFO] --------------< ppss:modulo1 >-----------------------------
[INFO] Building modulo1 1.0-SNAPSHOT                      [2/3]
[INFO] --------------------------------[ jar ]---------------------------
        //ejecuciones de las goals correspondientes...
[INFO] ----------------< ppss:modulo2 >-----------------------------
[INFO] Building modulo2 1.0-SNAPSHOT                      [3/3]
[INFO] --------------------------------[ jar ]---------------------------
        //ejecuciones de las goals correspondientes...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for multimodulo 1.0-SNAPSHOT:
[INFO]
[INFO] multimodulo ..................................... SUCCESS [  0.030 s]
[INFO] modulo1 ......................................... SUCCESS [  1.194 s]
[INFO] modulo2 ......................................... SUCCESS [  0.913 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Beneficios de usar multimódulos

La ventaja significativa del uso de multimódulos es la **reducción de duplicación**. Por ejemplo, podremos construir todos los módulos con **un único comando** (aplicado al aggregator pom), sin preocuparnos por el orden de construcción, ya que maven tendrá en cuenta las dependencias entre ellos y los ordenará convenientemente.

También podremos "compartir" elementos como propiedades, dependencias y plugins entre los módulos agregados usando el mecanismo de **herencia** que nos proporciona maven.

Maven soporta la relación de herencia, de forma que podemos crear un pom que nos sirva para identificar a un proyecto "padre". Podemos incluir cualquier configuración en el pom del proyecto padre, de forma que sus módulos "hijo" la hereden, evitando de nuevo duplicaciones.

Siguiendo con nuestro ejemplo, el proyecto multimodulo será nuestro proyecto "padre", y sus hijos serán los módulos *modulo1* y *modulo2*. Para ello, tendremos que referenciar al proyecto "padre" en cada uno de los módulos "hijo" usando la etiqueta <parent>.

```xml
<project ...>

  <parent>
    <artifactId>multimodulo</artifactId>
    <groupId>ppss</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>modulo1</artifactId>

    ...
</project>
```

modulo1/pom.xml

```xml
<project ...>

  <parent>
    <artifactId>multimodulo</artifactId>
    <groupId>ppss</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>modulo2</artifactId>

    ...
</project>
```

modulo2/pom.xml

En este caso el proyecto maven *multimodulo* es el proyecto "padre. NO tenemos que indicar en el pom "padre" quienes son sus "hijos", sino que en cada módulo "hijo" añadiremos las coordenadas del proyecto "padre", del cual heredarán su configuración. Prácticamente se heredan la mayoría de elementos: <properties>, <dependencies>, <plugins>, <artifactId>, <version>,... Por eso no es necesario añadir las coordenadas <groupId> y <version> en los "hijos" si vana a ser las mismas que las del proyecto "padre". Entre las (pocas) etiquetas que NO se heredan están <artifactId> y <name>.

Podemos "sobreescribir" cualquier elemento heredado o usar el del proyecto "padre".

No es necesario usar los mecanismos de herencia y agregación conjuntamente. Es decir, podemos tener únicamente una relación de herencia entre nuestros proyectos maven, o sólo relaciones de agregación, o ambas, como hemos mostrado en nuestro ejemplo.

Obviamente, la reducción de duplicaciones será mayor si usamos a la vez herencia y agregación en nuestros proyectos maven.

Ejercicios

En esta sesión trabajaremos con 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*** : "*P07-dbunit*". ***Project Location:*** "$HOME/ppss-2020-Gx-.../*P07-Integracion/P07-dbunit*". Es decir, que tenemos que crear el proyecto dentro de nuestro directorio de trabajo, y del directorio *P07-Integracion*.

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.

➪ ➪ Ejercicio 1: proyecto dbunitexample

Añade un MÓDULO a nuestro proyecto IntelliJ *P07-dbunit* (***File→New Module)***:

- Seleccionamos ***Maven***, y nos aseguramos de elegir el ***JDK 1.8***
- ***GroupId***: "*ppss*"; ***ArtifactId:*** "*dbunitexample*".
- ***ModuleName***: "*dbunitexample*". ***Content Root:*** "*$HOME/ppss-2020-Gx-.../P07-Integracion/P07-dbunit/dbunitexample*". ***Module file location***: "*$HOME/ppss-2020-Gx-.../P07-Integracion/P07-dbunit/dbunitexample*".

Como siempre, necesitamos incluir las *<properties>* en el pom, que copiaremos de proyectos anteriores.

En el directorio Plantillas-P07/ejercicio1 encontraréis los ficheros que vamos a usar en este ejercicio:

- la implementación de las clases sobre las que realizaremos las pruebas: *ppss.ClienteDAO, ppss.Ciiente* y *ppss.IClienteDAO*,
- la implementación de dos tests de integración (*ClienteDAO_IT.java*),
- ficheros xml con los datos iniciales de la BD (*cliente-init.xml*), y resultado esperado de uno de los tests (*cliente-esperado.xml*),
- fichero con el script sql para restaurar el esquema y las tablas de la base de datos (*create-table-customer.sql*)

Se pide:

A) Organiza el código proporcionado en el proyecto Maven que has creado según hemos visto en clase, y modifica el pom.xml convenientemente para poder utilizar DbUnit y el plugin *sql-maven-plugin.* Añade, además, las siguientes dependencias, se trata de librerías de *logging.* Solamente las incluimos para que no nos aparezcan advertencias (*warnings)* al construir el proyecto con Maven.

```
<!-- Si no la incluiumos veremos un warning al ejecutar los tests -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.30</version>
</dependency>
```

Nota: Asegúrate de que la cadena de conexión del ***plugin sql*** NO contenga el esquema de nuestra base de datos DBUNIT, ya que precisamente el plugin sql es el que se encargará de crearlo cada vez que construyamos el proyecto. Por lo tanto, la cadena de conexión que usa el plugin sql debe ser: *jdbc:mysql://localhost:3306/?useSSL=false*. Cuando uses la cadena de conexión en el código del proyecto ésta SÍ deberá incluir el esquema DBUNIT.

B) En el código de pruebas, explica qué representan las variables *clienteDAO* y *databaseTester* indicando para qué y dónde se utilizan. (puedes responder a las preguntas en un fichero con nombre "respuestas.txt" en el directorio raiz de este proyecto (mismo directorio que el pom.xml).

C) En el código de pruebas, explica la diferencia entre un *dataset* y una *table*. Identifica dónde y para qué se utilizan en el código proporcionado.

D) Utiliza Maven para ejecutar los tests, para ello debes asegurarte previamente de que tanto la cadena de conexión, como el login y el password sean correctos, tanto en el *pom.xml*, como en el código de pruebas y en el código fuente. Observa la consola que muestra la salida de la construcción del proyecto y comprueba que se ejecutan todas las goals que hemos indicado en clase. Fíjate bien en los artefactos (ficheros) generados (**Nota**: las librerías descargadas por maven en nuestro repositorio local son necesarios para construir el proyecto, pero NO son artefactos generados por Maven durante la construcción del proyecto).

E) Una vez ejecutados los tests, cambia los datos de prueba del método test_insert() y realiza las modificaciones necesarias adicionales para que los tests sigan en "verde". Puedes probar a realizar cambios en los datos de entrada del otro test (test_delete()) para familiarizarte con el código.

F) Implementa dos tests adicionales. Un test para actualizar los datos de un cliente (*testUpdate()*) y otro para recuperar los datos de un cliente (*testRetrieve()*). Utiliza los siguientes casos de prueba:

Tabla cliente inicial	Datos del cliente a modificar	Resultado esperado (tabla)
id =1 nombre="John" apellido= "Smith" direccion = "1 Main Street" ciudad = "Anycity"	id =1 nombre= "John" apellido= "Smith" direccion = "Other Street" ciudad = "NewCity"	id =1 nombre = "John" apellido= "Smith" direccion = "Other Street" ciudad = "NewCity"

Tabla cliente inicial	id del cliente a recuperar	Resultado esperado
id =1 nombre ="John" apellido = "Smith" direccion = "1 Main Street" ciudad = "Anycity"	id =1	id =1 nombre ="John" apellido = "Smith" direccion = "1 Main Street" ciudad = "Anycity"

➪ ➪ Ejercicio 2: proyecto multimódulo matriculacion

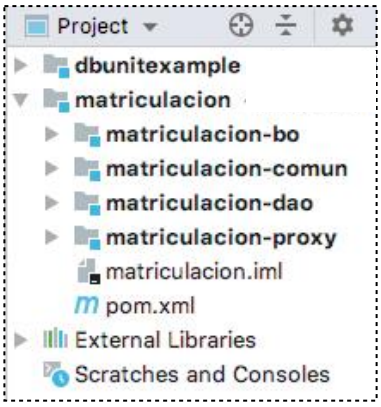
Vamos a añadir un segundo módulo a nuestro proyecto IntelliJ *P07-dbunit*. Usaremos el proyecto multimódulo "*matriculacion*", que encontraréis en la carpeta: Plantillas-P07/ejercicio2.

Copia, desde un terminal, el proyecto maven multimódulo "matriculacion" de la práctica anterior (carpeta matriculacion del directorio Plantillas-P07/ejercicio2 y todo su contenido) en el directorio "*.../P07-dbunit*".

Ahora añadiremos el proyecto como un nuevo módulo al nuestro proyecto IntelliJ ***(File→New →Module from Existing Sources...)***:

- Seleccionamos la carpeta matriculación que acabamos de copiar.
- En la siguiente pantalla seleccionamos: ***Import module from external model →Maven***

Finalmente debemos ver nuestro proyecto matriculacion como un módulo en la vista Project de IntelliJ, tal y como mostramos en la imagen de la derecha:



Contenido de nuestro proyecto IntelliJ P07-dbunit

En el directorio Plantillas-P07/ejercicio2 encontraréis los ficheros que vamos a usar en este ejercicio:

- fichero con el script sql para restaurar el esquema y las tablas de la base de datos (*matriculacion.sql*)
- fichero xml con un dataset ejemplo (*dataset.xml*) y fichero dtd con la gramática de los ficheros XML para nuestro esquema de base de datos (*matriculacion.dtd*)

Se pide:

A) Dado que hay una relación de herencia entre *matriculación* y sus módulos agregados, vamos a modificar el **pom de *matriculacion***, para añadir aquellos elementos comunes a los submódulos. Concretamente, estos deben heredar lo siguiente: la librería para usar junit5 (*junit-jupiter-engine*), y los *plugins surefire* y *failsafe*. Indica para qué sirven exactamente cada una de estas dependencias.

B) Incluye en el **pom del módulo *matriculacion-dao*** las librerías dbunit, slf4j-simple, y mysql-connector-java (igual que en el ejercicio anterior). Debes añadir también el plugin sql-maven-plugin. El *script* sql será el fichero *matriculacion.sql* que encontrarás en el directorio de plantillas (deberás copiarlo en el directorio *src/test/resources/sql*).

C) En la clase ***FuenteDatosJDBC*** (en src/main/java del módulo matriculacion-dao) se configura la conexión con la BD. Debes cambiar el driver hsqldb por el driver jdbc (*com.mysql.cj.jdbc.Driver*), así como los parámetros del método getConnection, es decir, la cadena de conexión, login y password (debes usar los valores que ya hemos usado para el ejercicio anterior).

D) Copia el fichero ***matriculacion.dtd*** del directorio de plantillas en src/test/resources (del módulo matriculacion-dao). Como ya hemos indicado, contiene la gramática de los datasets de nuestro esquema de base de datos. El fichero ***dataset.xml*** tiene un ejemplo de cómo definir nuestros datasets en formato xml. Puedes copiar también dicho fichero en src/test/resources.

E) Queremos implementar tests de integración para los métodos AlumnoDAO.addAlumno() y AlumnoDAO.delAlumno(). Para ello vamos a necesitar crear los ficheros ***tabla2.xml***, ***tabla3.xml***, y ***tabla4.xml***, que contienen los ***datasets* iniciales** y ***datasets esperados*** necesarios para implementar los casos de prueba de la **tabla 1** (dichos datasets se definen en las **tablas 2, 3 y 4**). Crea los ficheros xml correspondientes en la carpeta src/test/resources, con la opción New→File (desde el directorio src/test/resources), indicando el nombre del fichero: por ejemplo *tabla2.xml*. Puedes copiar el contenido de dataset.xml y editar los datos convenientemente.

Tabla 2. Base de datos de entrada. Contenido tabla ALUMNOS

NIF	Nombre	Dirección	E-mail	Fecha nacimiento
11111111A	Alfonso Ramirez Ruiz	Rambla, 22	alfonso@ppss.ua.es	1982-02-22 00:00:00.0
22222222B	Laura Martinez Perez	Maisonnave, 5	laura@ppss.ua.es	1980-02-22 00:00:00.0

Tabla 3. Base de datos esperada como salida en testA1

NIF	Nombre	Dirección	E-mail	Fecha nacimiento
11111111A	Alfonso Ramirez Ruiz	Rambla, 22	alfonso@ppss.ua.es	1982-02-22 00:00:00.0
22222222B	Laura Martinez Perez	Maisonnave, 5	laura@ppss.ua.es	1980-02-22 00:00:00.0
33333333C	Elena Aguirre Juarez			1985-02-22 00:00:00.0

Tabla 4. Base de datos esperada como salida en testB1

NIF	Nombre	Dirección	E-mail	Fecha nacimiento
22222222B	Laura Martinez Perez	Maisonnave, 5	laura@ppss.ua.es	1980-02-22 00:00:00.0

F) **Implementa**, usando *DbUnit,* **los casos de prueba de la tabla 1** en una clase `AlumnoDAOIT`. Cada caso de prueba debe llevar el nombre indicado en la columna ID de la tabla 1. Ejecuta los *tests*. Hazlo desde el proyecto matriculacion-dao (comando mvn verify).

Para hacer las pruebas, tendremos que utilizar un objeto que implemente la interfaz IAlumnoDAO. Así, por ejemplo para cada test Ax de la tabla 1, utilizaremos una sentencia del tipo:

```
new FactoriaDAO().getAlumnoDAO().addAlumno(alumno);
```

Para especificar la fecha, debes utilizar la clase Calendar. Por ejemplo, para indicar la fecha de nacimiento "1985-02-22 00:00:00.0" de un objeto alumno, puede hacerse de la siguiente forma:

```
Calendar cal = Calendar.getInstance();
cal.set(Calendar.HOUR, 0);
cal.set(Calendar.MINUTE, 0);
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.MILLISECOND, 0);
cal.set(Calendar.YEAR, 1985);
cal.set(Calendar.MONTH, 1); //Nota: en la clase Calendar, el primer mes es 0
cal.set(Calendar.DATE, 22);
alumno.setFechaNacimiento(cal.getTime());
```

Tabla 1. Casos de prueba para AlumnoDAO

ID	Método a probar	Entrada	Salida Esperada
testA1	void addAlumno (AlumnoTO p)	p.nif = "33333333C" p.nombre = "Elena Aguirre Juarez" p.fechaNac = 1985-02-22 00:00:00.0	Tabla 3
testA2	void addAlumno (AlumnoTO p)	p.nif = "11111111A" p.nombre = "Alfonso Ramirez Ruiz" p.fechaNac = 1982-02-22 00:00:00.0	DAOException
testA3	void addAlumno (AlumnoTO p)	p.nif = "44444444D" p.nombre = null p.fechaNac = 1982-02-22 00:00:00.0	DAOException
testA4	void addAlumno (AlumnoTO p)	p = null	DAOException
testA5	void addAlumno (AlumnoTO p)	p.nif = null p.nombre = "Pedro Garcia Lopez" p.fechaNac = 1982-02-22 00:00:00.0	DAOException
testB1	void delAlumno (String nif)	nif = "11111111A"	Tabla 4
testB2	void delAlumno (String nif)	nif = "33333333C"	DAOException

G) Ejecuta los tests anteriores teniendo en cuenta que si los algún test falla debemos interrumpir la construcción del proyecto. El test "testA4" falla. Repara el error en el fuente, de forma que devuelva el resultado correcto.

H) Añade 3 **tests unitarios** en una clase AlumnoDAOTest. NO es necesario que los implementes, pueden contener simplemente un assertTrue(true). Ejecuta sólo los tests unitarios usando el comando mvn test y comprueba que efectivamente sólo se lanzan los tests unitarios

I) Añade también tests unitarios y de integración en los proyectos *matriculacion-bo* y *matriculacion-proxy*. Luego ejecuta *mvn verify* desde el proyecto padre. Anota la secuencia en la que se ejecutan los diferentes tipos de tests de todos los módulos. ¿Qué estrategia de integración estás siguiendo?

J) Finalmente añade un nuevo **elemento de configuración**, dentro del proyecto *matriculacion* con el comando maven "*mvn verify -Dgroups="Integracion-fase1"*. Dado que los plugins *failsafe* y *surefire* comparten la variable *groups*, etiqueta tanto los tests unitarios como los de integración del proyecto *matriculacion-dao* con la etiqueta "*Integracion-fase1"*. Ejecútala y comprueba que ahora sólo se ejecutan los tests unitarios y de integración del proyecto *matriculacion-dao.*

Resumen

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

PRUEBAS DE INTEGRACIÓN

- Nuestro SUT estará formado por un conjunto de unidades. El objetivo principal es detectar defectos en las INTERFACES de dicho conjunto de unidades. Recuerda que dichas unidades ya habrán sido probadas individualmente
- Son pruebas dinámicas (requieren la ejecución de código), y también son pruebas de verificación (buscamos encontrar defectos en el código). Se realizan de forma INCREMENTAL siguiendo una ESTRATEGIA de integración, la cual determinará EL ORDEN en el que se deben seleccionar las unidades a integrar
- Las pruebas de integración se realizan en varias "fases", añadiendo cada vez un determinado número de unidades al conjunto, hasta que al final tengamos probadas TODAS las unidades integradas. En cada una de las "fases" tendremos que REPETIR TODAS las pruebas anteriores. A este proceso de "repetición" se le denomina PRUEBAS DE REGRESIÓN. Las pruebas de regresión provocan que las últimas unidades que se añaden al conjunto sean las MENOS probadas. Este hecho debemos tenerlo en cuenta a la hora de tomar una decisión por una estrategia de integración u otra, además del tipo de proyecto que estemos desarrollando (interfaz compleja o no, lógica de negocio compleja o no, tamaño del proyecto, riesgos,...)

DISEÑO DE PRUEBAS DE INTEGRACIÓN

- Se usan técnicas de caja negra. Básicamente se seleccionan los comportamientos a probar (en cada una de las fases de integración) siguiendo unas guías generales en función del tipo de interfaces que usemos en nuestra aplicación.
- Recuerda que si nuestro test de integración da como resultado "failure" buscaremos la causa del error en las interfaces, aunque es posible que en este nivel de pruebas, salgan a la luz errores no detectados "dentro" de las unidades. Esto es así porque recuerda que NO podemos hacer pruebas exhaustivas, por lo tanto, nunca podremos demostrar que el código probado no tiene más defectos de los que hemos sido capaces de encontrar.

AUTOMATIZACIÓN DE LAS PRUEBAS DE INTEGRACIÓN CON UNA BD

- Usaremos la librería dbunit para automatizar las pruebas de integración con una BD. Esta librería es necesaria para controlar el estado de la BD antes de las pruebas, y comprobar el estado resultante después de ellas..
- Los tests de integración deben ejecutarse después de realizar todas la pruebas unitarias. Necesitamos disponer de todos los .class de nuestras unidades (antes de decidir un ORDEN de integración), por lo que los tests requerirán del empaquetado en un .jar de todos los .class de nuestro código.
- Los tests de integración son ejecutados por el plugin failsafe. Debemos añadirlo al pom, asociando la goal integration-test a la fase con el mismo nombre.
- El plugin sql permite ejecutar scripts sql. Es interesante para incluir acciones sobre la BD durante el proceso de construcción del proyecto. Por ejemplo, "recrear" las tablas de nuestra BD en cada ejecución, e inicializar dichas tablas con ciertos datos. La goal "execute" es la que se encarga de ejecutar el script, que situaremos físicamente en la carpeta "resources" de nuestro proyecto maven. Necesitaremos configurar el driver con los dato de acceso a la BD, y también asociar la goal a alguna fase previa (por ejemplo pre-integration-test) a la fase en la que se ejecutan los tests de integración.
- Es habitual que el código a integrar esté distribuido en varios proyectos. Los proyectos maven multimódulo nos permiten agrupar y trabajar con múltiples proyectos maven. En este caso nuestro proyecto maven multimódulo contendrá varios subproyectos maven.
- El proyecto maven multimódulo (que agrupa al resto de subproyectos), únicamente contiene el fichero de configuración pom.xml, es decir, NO tiene código (carpeta src), simplemente sirve para agrupar todo el código de nuestra aplicación en una única carpeta. El pom del proyecto multimódulo permite reducir duplicaciones, ya que se puede aplicar el mismo comando a todos los submódulos (misma configuración a todos los submódulos (relación de agregación, y/o aplicar la misma configuración a los submódulos (relación de herencia)