

P06B- Pruebas de integración

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P06B) termina justo ANTES de comenzar la siguiente sesión de prácticas (P07) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P06B (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

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, para así obtener el informe de pruebas de forma automática. 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 dependencias externas (una de las cuales será la base de datos, o mejor dicho, el servidor de la base de datos). Ahora 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 "salgan a la luz" 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 evidenciar problemas en "líneas de código diferentes"!!! En este caso, y tal y 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.

En esta práctica nos vamos a centrar en el problema de la integración con una base de datos.

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 **P06B-Integracion**, dentro de tu espacio de trabajo.

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 (Para salir, usa el comando exit;)
```

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/ESQUEMA_BD?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 *ESQUEMA_BD* (el cual contiene el conjunto de tablas de base de datos que usará nuestra aplicación).

Podemos crear de forma automática nuestro "esquema" mysql de forma automática, ejecutando un script sql.

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

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

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 password, 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.

```
...
<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>
...
```

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

- ❖ 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

Acceso a una base de datos desde IntelliJ

Podemos acceder a nuestro servidor de base de datos MySQL instalado en la máquina virtual, a través de IntelliJ, en concreto, desde la vista "**Database**" (Si no aparece en el lateral, se puede abrir desde el menú de IntelliJ *View→Tool Windows→Database*).

Desde la ventana *Database*, creamos una nueva "conexión" desde "+ → Data Source → MySQL".

A continuación proporcionamos los valores:

- ❖ **Name:** por defecto se muestra *@localhost* (puedes dejar este nombre o cambiarlo)
- ❖ **Host:** localhost, **Port:** 3306
- ❖ **User:** root, **Password:** ppss, **URL:** jdbc:mysql://localhost:3306
- ❖ Pulsa sobre "**Test Connection**" para asegurarte de que funciona la conexión (ver **Nota)
- ❖ Finalmente pulsamos sobre **OK**.

Ahora ya puedes ver los esquemas de base de datos creadas en el servidor MySQL, e interactuar con él a través de IntelliJ.

Pulsando sobre el segundo icono empezando por la derecha desde esa ventana (Jump To Query Console...), se abre un editor con el que podemos realizar queries sobre la base de datos. Por ejemplo, para ver el contenido de una tabla de la base de datos:

```
USE nombreEsquema;          <---- Nos situamos en el esquema correspondiente
SELECT * FROM nombreTabla; <---- Recuperamos los datos de una tabla ordenados por clave primaria
```

****Nota:** Si al probar la conexión aparece el mensaje: *"Failed. Server returns invalid timezone. Need to set 'serverTimezone' property"*, pulsa sobre *Set Time Zone*, asegúrate de que se muestra la propiedad *serverTimezone* con el valor UTC, y selecciona OK. Intenta nuevamente probar la conexión y comprueba que funciona correctamente.

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**, tendremos que realizar lo siguiente:

- **File→New Project.** A continuación elegimos "**Multi-module Project**" y pulsamos sobre Next,
- **Project name :** "*dbunit*". **Project Location:** "*\$HOME/ppss-2022-Gx-.../P06B-Integracion/dbunit*".

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

En ese caso, la carpeta **dbunit** contiene nuestro **proyecto IntelliJ**.

No vamos a añadir todavía ningún módulo, por lo que terminamos el proceso de forma que efectivamente tenemos un proyecto IntelliJ "vacío".

Nota: En versiones anteriores de IntelliJ, la opción "*Empty project*" creaba un proyecto vacío. Con la nueva versión, y como habéis podido comprobar en la práctica anterior, la opción "*Empty project*" crea un proyecto IntelliJ con un módulo, mientras que la opción "*Multi-module Project*" crea efectivamente un proyecto vacío, que es lo que queremos.

📁 Ejercicio 1: proyecto dbunitexample

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

- Seleccionamos **Maven**, y nos aseguramos de elegir el **JDK 11**
- **Parent:** *<None>*
- **Name:** *dbunitexample*
- **Location:** *\$HOME/ppss-2022-Gx-.../P06B-Integracion/dbunit/dbunitexample*
- **GroupId:** *ppss* **ArtifactId:** *dbunitexample*

Nota: Si al crear del módulo aparece una ventana advirtiéndote de que el fichero *pom.xml* no pertenece al proyecto, simplemente dejaremos marcada la opción indicando que queremos editar ese fichero igualmente.

El proyecto que acabamos de crear es un **single maven project**. Como siempre, necesitamos incluir las *<properties>* en el *pom*, que copiaremos de proyectos anteriores. Además necesitaremos añadir como dependencias la librería *junit5*, más las nuevas dependencias que hemos visto en clase (*dbunit* y *driver mysql*)

Con respecto a los **plugins**, recuerda que tienes que añadir el compilador de java (versión 3.9.0), y los nuevos plugins que hemos visto en clase para esta sesión (*failsafe* y *sql*).

En el directorio Plantillas-P06B/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.Ciente* y *ppss.IClienteDAO*,
- ❖ la implementación de dos tests de integración (*ClienteDAO_IT.java*), más una clase adicional (*MijdbcDatabaseTester.java*) en la que indicaremos a DbUnit que vamos a trabajar con una base de datos MySQL. Esta clase la usaremos para que DbUnit no nos muestre ningún *warning* al acceder a dicha base de datos, y estará en el mismo directorio que los tests.
- ❖ ficheros xml con los datos iniciales de la BD (*cliente-init.xml*), y resultado esperado de uno de los tests (*cliente-esperado.xml*). En clase hemos visto dónde ubicar estos ficheros.
- ❖ fichero con el script sql para restaurar el esquema y las tablas de la base de datos (*create-table-customer.sql*). En clase hemos visto dónde ubicar estos ficheros.

Tendrás que crear la carpeta **resources** en *src/test/*. Para que IntelliJ la "vea" como la carpeta de recursos de maven, selecciona "**Mark directory as →Test Resources Root**" desde el menú contextual de la carpeta "*src/test/resources*".

Además, crea un nuevo *DataSource* asociado al servidor de bases de datos MySQL desde la ventana Database.

Se pide:

- A) Organiza el código proporcionado en el proyecto Maven que has creado según hemos visto en clase. Revisa el *pom* para que contenga las dependencias y plugins correctos, y añade, además, la siguiente librería de *logging*. Solamente la incluimos para que no nos aparezcan advertencias (*warnings*) al construir el proyecto con Maven.

```
<!-- Si no la incluimos veremos un warning al ejecutar los tests -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.36</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 correspondiente (puedes verlo en el script sql proporcionado).

La clase que vamos a usar que representa el driver para acceder a la base de datos es: `com.mysql.cj.jdbc.Driver`. Recuerda que tenemos que acceder a la base de datos tanto desde `src/main` como desde `src/test`.

Aunque proporcionamos ya el código de dos tests de integración, debes tener claro el uso del API `dbunit` y qué es lo que estamos probando en los tests.

- B) Utiliza la ventana Maven para ejecutar los tests de integración. 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 ficheros y artefactos generados.

Verás que aparecen varios *warnings*. Para que esto no ocurra debemos indicar a DbUnit el tipo de base de datos al que estamos accediendo. Usa la clase `MysqlDatabaseTester` proporcionada, en lugar de la clase `JdbcDatabaseTester` que hemos explicado en clase.

Puedes comprobar en la ventana Database de IntelliJ que se ha creado el esquema DBUNIT con una tabla `cliente`. Para ello tendrás que "refrescar" la vista Database, pulsando sobre el tercer icono empezando por la izquierda (icono *Refresh*).

- C) Una vez ejecutados los tests, crea un nuevo test `test_insert2()` cambiando los datos de entrada y considerando que en la base de datos ya existen los datos de dos clientes, y queremos añadir uno nuevo. Puedes añadir un nuevo test `test_delete2()` con diferentes datos de entrada para familiarizarte con el código.
- D) 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 `dbunit`. Usaremos el proyecto multimódulo "`matriculacion`" de la práctica anterior (P06A).

Copia, desde un terminal, el proyecto maven multimódulo "`matriculacion`" de la práctica anterior (carpeta `matriculacion`) en la carpeta de nuestro proyecto IntelliJ, es decir: **`dbunit`**.

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

- Seleccionamos la carpeta *matriculacion* que acabamos de copiar en el directorio *dbunit*.
- En la siguiente pantalla seleccionamos: **Create module from existing sources**

IntelliJ detectará automáticamente los directorios de fuentes de nuestro proyecto multimódulo (asegúrate de que estén todos "marcados". Seguiremos pulsando sobre "next" hasta que lleguemos al final.

Nos aparecerá una ventana informándonos de que se ha encontrado con los ficheros *.iml*, y nos preguntará lo que queremos hacer con ellos. Podemos decirle que queremos "reutilizarlos" (opción "reuse"). Con esto finalizamos el proceso y ya podemos trabajar con el proyecto multimódulo *matriculación*.

En la ventana Project Structure→ Project Settings → Modules, verás que IntelliJ ha "marcado" los directorios de fuentes maven, los de recursos, el directorio target...

La **Figura 1** muestra la lista de módulos y el marcado de directorios para el proyecto *matriculacion-dao*

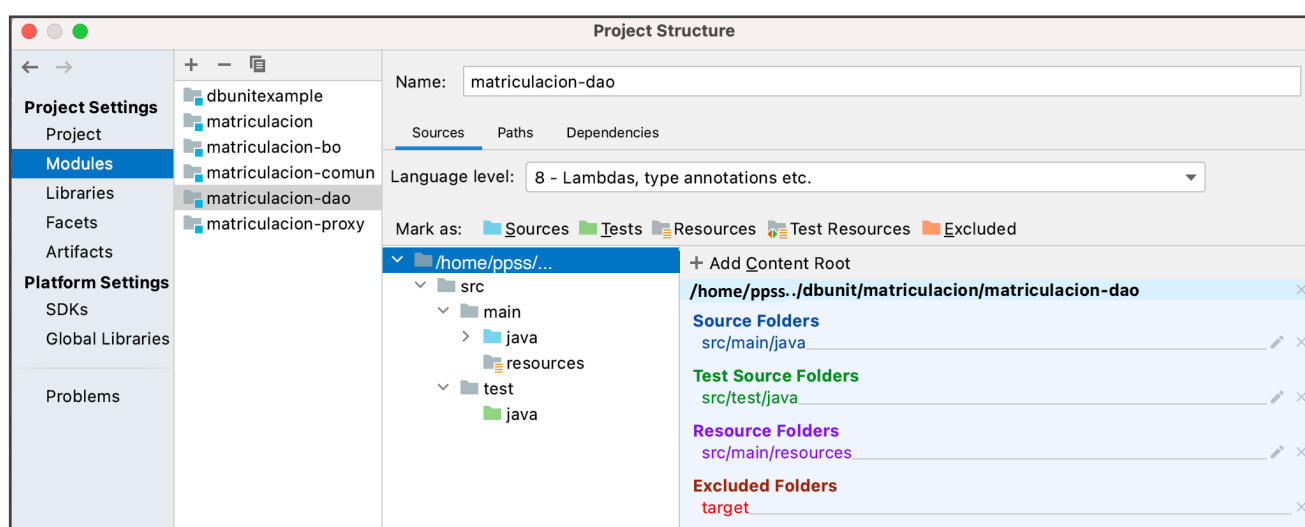
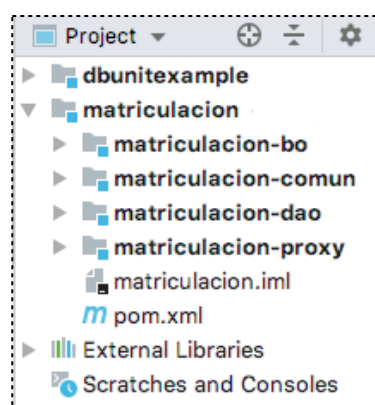


Figura 1. Lista de módulos del proyecto dbunit. Los módulos *matriculacion-xxx* son proyectos maven

Finalmente debemos ver nuestro proyecto *matriculación* como un módulo en la vista Project de IntelliJ (ver imagen de la derecha)



Vista Project del proyecto IntelliJ dbunit

En el módulo *matriculacion-dao*, crea la carpeta **resources** en *src/test/*. Para que IntelliJ la "vea" como la carpeta de recursos de maven, selecciona "**Mark directory as** → **Test Resources Root**" desde el menú contextual de la carpeta "*src/test/resources*".

En el directorio Plantillas-P06B/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:

- 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*, y los *plugins compiler, surefire y failsafe*. Deberías tener claro para qué sirven cada uno de ellos.
- 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*). Acuérdate de marcar la carpeta *resources* como "Test Resources Root":
- 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 (usaremos la misma base de datos del ejercicio anterior pero el esquema de BD será diferente, puedes ver cuál es en el script sql).
- Copia el fichero ***matriculacion.dtd*** del directorio de plantillas en *src/test/resources* (del módulo *matriculacion-dao*). El fichero ***dataset.xml*** tiene un ejemplo de cómo definir nuestros datasets en formato xml. Puedes copiar también dicho fichero en *src/test/resources*.
- 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
22222222B	Laura Martinez Perez	Maisonnavé, 5	laura@ppss.ua.es	1980-02-22

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
22222222B	Laura Martinez Perez	Maisonnavé, 5	laura@ppss.ua.es	1980-02-22
33333333C	Elena Aguirre Juarez			1985-02-22

Tabla 4. Base de datos esperada como salida en testB1

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

- 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.
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 *LocalDate*. Usa el constructor *LocalDate.of()* tal y como ya hemos hecho en prácticas anteriores.

Nota: en los drivers, deberías usar la clase *MiJdbcDatabaseTester* (igual que hemos hecho en el ejercicio anterior) para evitar los *warnings* al ejecutar los tests.

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	Tabla 3
testA2	void addAlumno (AlumnoTO p)	p.nif = "11111111A" p.nombre = "Alfonso Ramirez Ruiz" p.fechaNac = 1982-02-22	DAOException (1)
testA3	void addAlumno (AlumnoTO p)	p.nif = "44444444D" p.nombre = null p.fechaNac = 1982-02-22	DAOException (1)
testA4	void addAlumno (AlumnoTO p)	p = null	DAOException (2)
testA5	void addAlumno (AlumnoTO p)	p.nif = null p.nombre = "Pedro Garcia Lopez" p.fechaNac = 1982-02-22	DAOException (1)
testB1	void delAlumno (String nif)	nif = "11111111A"	Tabla 4
testB2	void delAlumno (String nif)	nif = "33333333C"	DAOException (3)

DAOException(1) = mensaje "Error al conectar con BD"

DAOException(2)= mensaje "Alumno nulo"

DAOException(3)= mensaje "No se ha borrado ningun alumno"

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

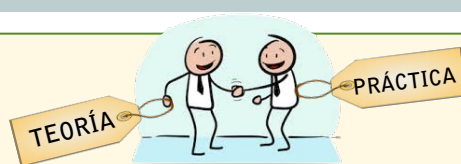
Nota: para ver el nuevo esquema creado, no es suficiente con refrescar el DataSource de la ventana Database. Edita el DataSource, y en la pestaña "**Schemas**", marca el nuevo esquema (matriculacion).

- H) Añade 3 **tests unitarios** en una nueva clase AlumnoDAOTest (deberías tener claro por qué la hemos llamado así). NO es necesario que implementes los drivers, 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. Deberías saber qué estrategia de integración estamos 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 de forma incremental, 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,...)
- Recuerda que no podemos integrar dos unidades si no hay ninguna relación entre ellas, ya que el objetivo es encontrar errores en las interfaces de las unidades (en la interconexión entre ellas). Por eso es indispensable tener en cuenta el diseño de nuestra aplicación a la hora de determinar la estrategia de integración (qué componentes van a integrarse cada vez.)

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 durante las pruebas unitarias (ya 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). El proyecto maven multimódulo agrupa todo el código de nuestra aplicación en una única carpeta, permitiendo también lanzar la construcción de todos los módulos con un único comando. 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).
- Si usamos un proyecto maven multimódulo,, el mecanismo reactor ejecuta el comando maven en todos los submódulos ordenándolos previamente atendiendo a las dependencias entre ellos. Por lo tanto, si ejecutamos las pruebas de integración desde el proyecto multimódulo, estaremos integrando en orden ascendente