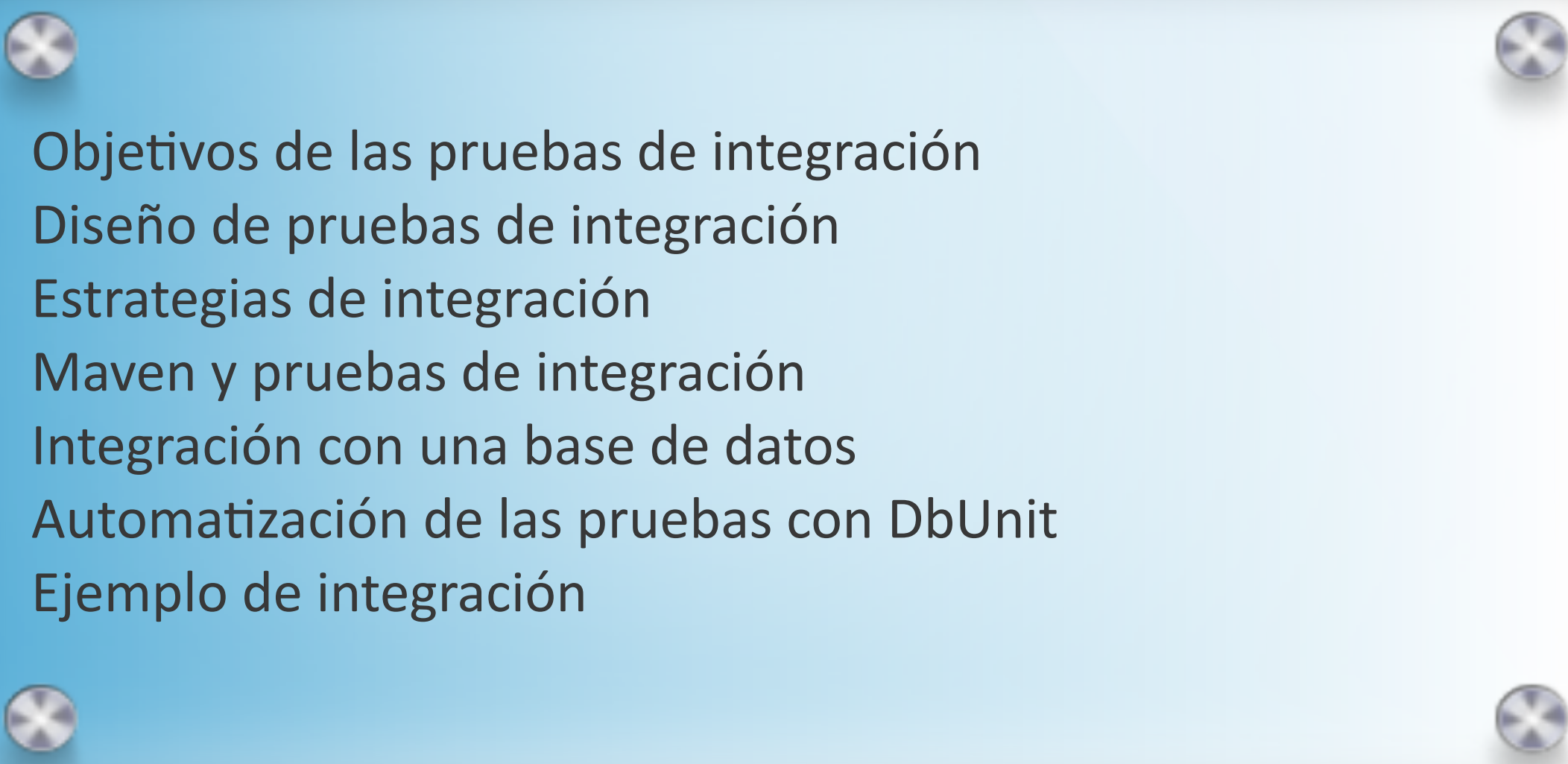




Sesión s07: Pruebas de integración

Four circular icons, each containing a stylized 'X' or cross shape, are positioned at the corners of the light blue box. They appear to be decorative elements or perhaps represent screws or fasteners.

Objetivos de las pruebas de integración
Diseño de pruebas de integración
Estrategias de integración
Maven y pruebas de integración
Integración con una base de datos
Automatización de las pruebas con DbUnit
Ejemplo de integración

NIVELES DE PRUEBAS

- Las pruebas se realizan a diferentes niveles, durante el proceso de desarrollo

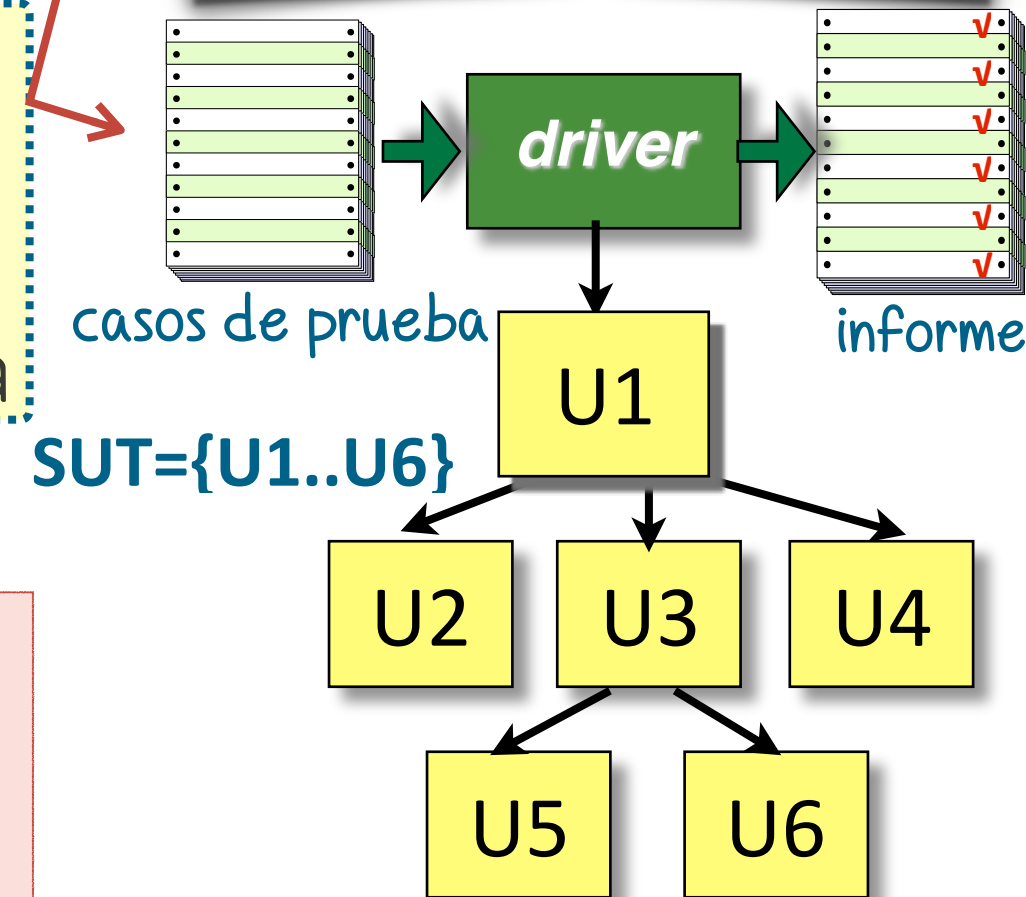


Objetivo: encontrar defectos en el código de las unidades

Cuestión fundamental:
¿cómo aislar cada unidad del resto del código?

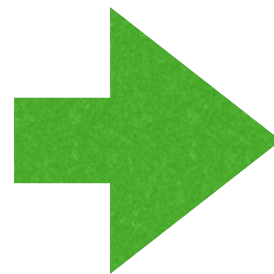
Objetivo: encontrar defectos derivados de la INTERACCIÓN entre las unidades, que PREVIAMENTE han sido probadas

Cuestión fundamental: ¿cuál es el "orden" en el que vamos a realizar dicha integración?



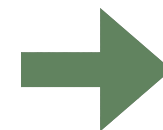
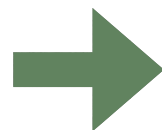
VALIDACIÓN
¿el producto es el correcto?
↓
nivel de aceptación

- A nivel de pruebas unitarias, el sistema "existe" en forma de "piezas" bajo el control de los programadores
- La siguiente tarea importante es "reunir" todas las "piezas" para construir el sistema completo
 - Un sistema es una colección de "componentes" interconectados de una determinada forma para cumplir un determinado objetivo



- Construir el sistema completo a partir de sus "piezas" no es una tarea fácil debido a los numerosos errores sobre las INTERFACES
 - A pesar de esforzarnos en realizar un buen diseño y documentación, las malinterpretaciones, errores, y descuidos son una realidad
 - Los errores de interfaz entre los diferentes componentes son provocados fundamentalmente por los programadores

- El objetivo de la integración del sistema es construir una versión "operativa" del sistema mediante:
 - la agregación, de forma **incremental**, de nuevos componentes, y
 - asegurándonos de que la adición de nuevos componentes no "perturban" el funcionamiento de los componentes que ya existen (**pruebas de regresión**)
- Las pruebas de integración del sistema constituyen un proceso sistemático para "ensamblar" un sistema software, durante el cual se ejecutan pruebas conducentes a descubrir errores asociados con las **interfaces** de dichos componentes
 - Se trata de garantizar que los componentes, sobre los que previamente se han realizado pruebas unitarias, funcionan correctamente cuando son "combinados" de acuerdo con lo indicado por el **diseño**





TIPOS DE INTERFACES Y ERRORES COMUNES



ver Bibliografía: Ian Sommerville, 9th ed. Cap. 8.1.3

- La interfaz entre componentes (unidades, módulos), puede ser de varios tipos:
 - Interfaz a través de **parámetros**: los datos se pasan de un componente a otro en forma de parámetros. Los métodos de un objeto tienen esta interfaz
 - **Memoria compartida**: se comparte un bloque de memoria entre los componentes. Los componentes escriben datos en la memoria compartida, que son leídas por otros
 - **Interfaz procedural**: un componente encapsula un conjunto de procedimientos que pueden llamarse desde otros componentes. Por ejemplo, los objetos tienen esta interfaz
 - **Paso de mensajes**: un componente A prepara un mensaje y lo envía al componente B. El mensaje de respuesta del componente B incluye los resultados de la ejecución del servicio. Por ejemplo, los servicios web tienen esta interfaz
- Errores más comunes derivados de la interacción de los componentes a través de sus interfaces:
 - Mal uso de la interfaz
 - Malentendido sobre la interfaz
 - Errores temporales
- Las pruebas para detectar defectos en las interfaces son difíciles, ya que algunos de ellos pueden sólo manifestarse bajo condiciones inusuales!!!



GUÍAS GENERALES PARA DISEÑAR LAS PRUEBAS



- Examinar el código a probar y listar de forma explícita cada llamada a un componente externo. Diseñar un conjunto de pruebas con los valores de los parámetros a componentes externos en los **extremos de los rangos**. Estos valores pueden revelar inconsistencias de la interfaz con una mayor probabilidad
- Si se pasan punteros a través de la interfaz, siempre probar con punteros nulos
- Cuando se invoca a un componente con una **interfaz procedural**, diseñar tests que provoquen, de forma deliberada, que el componente falle. Diferentes asunciones sobre los fallos son uno de los malentendidos sobre la especificación más comunes
- Utilizar pruebas de estrés en sistemas con **paso de mensajes**. Esto implica que deberíamos diseñar los tests de forma que se generen más mensajes de los que probablemente ocurran en la práctica. Esta es una forma efectiva de revelar problemas temporales
- Cuando varios componentes interaccionan con **memoria compartida**, diseñaremos los tests en el orden en los que estos componentes son activados. De esta forma revelaremos asunciones implícitas hechas por el programador sobre el orden en el que los datos son producidos y consumidos



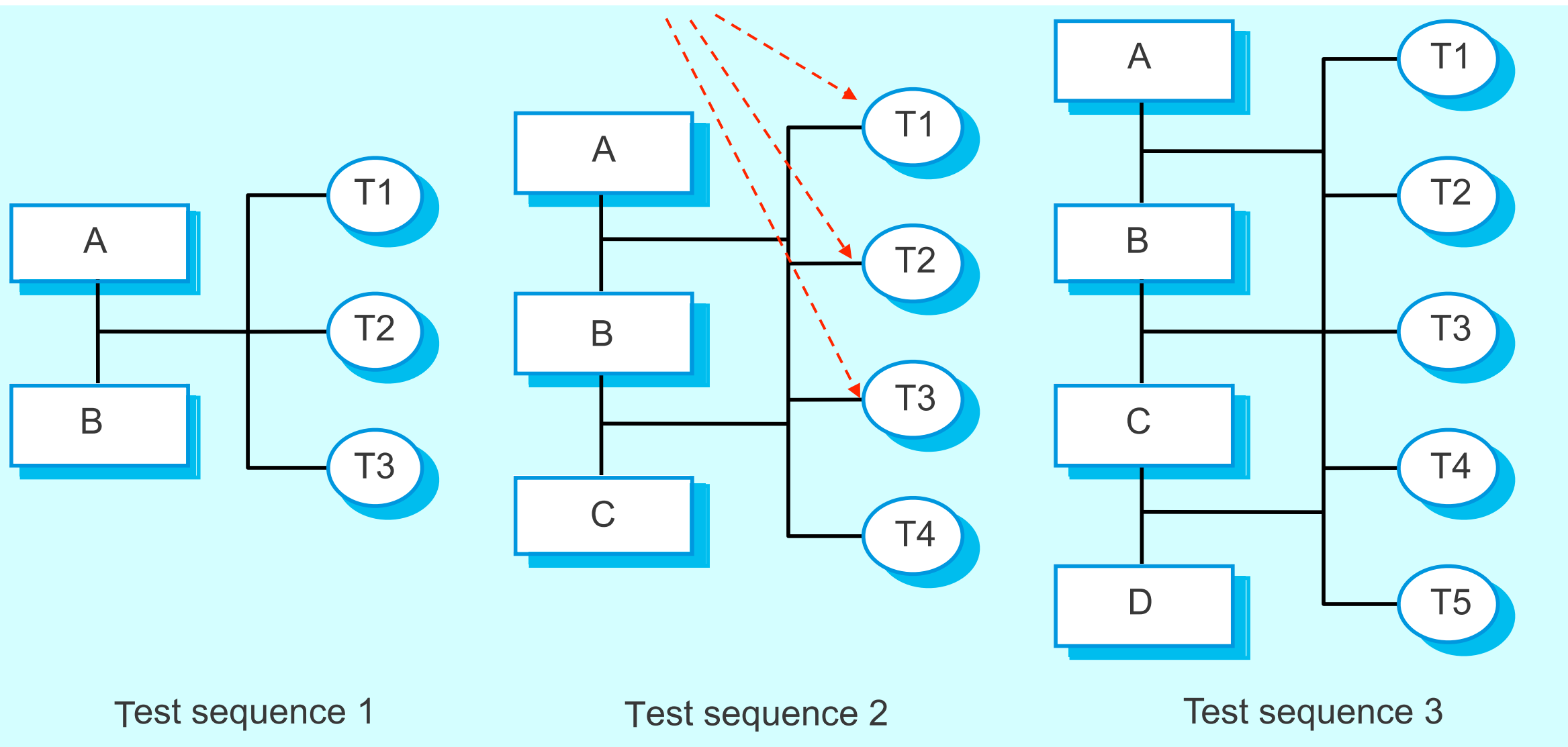
ESTRATEGIAS DE INTEGRACIÓN



- **Big Bang**: una vez realizadas las pruebas unitarias, se integran todas las unidades
- **Top-down**: integramos primero los componentes con mayor nivel de abstracción, y vamos añadiendo componentes cada vez con menor nivel de abstracción
- **Bottom-up**: integramos primero los componentes de infraestructura que proporcionan servicios comunes, como p.ej. acceso a base de datos, acceso a red,... y posteriormente añadimos los componentes funcionales, cada vez con mayor nivel de abstracción
- **Sandwich**: es una mezcla de las dos estrategias anteriores
- **Dirigida por los riesgos**: se eligen primero aquellos componentes que tengan un mayor riesgo (p.ej. aquellos con más probabilidad de errores por su complejidad)
- **Dirigida por las funcionalidades** (casos de uso, historias de usuario...)/**hilos de ejecución**: se ordenan las funcionalidades con algún criterio y se integra de acuerdo con este orden

- Las pruebas de REGRESIÓN consisten en repetir las pruebas realizadas cuando integramos un nuevo componente

PRUEBAS DE REGRESIÓN



A,B,...,D= Componentes a probar; T_i = Tests



MAVEN Y PRUEBAS DE INTEGRACIÓN



- El ciclo de vida de Maven contempla cuatro fases relacionadas con los tests de integración:
 - pre-integration-test**: en esta fase podemos iniciar cualquier servicio requerido o realizar cualquier acción, p.ej. iniciar una base de datos, un servidor web...
 - integration-test**: en esta fase se ejecutan los tests de integración (puede que se requiera desplegar el artefacto empaquetado en un entorno en el que se puedan ejecutar dichos tests)
 - * Si algún test falla NO se detiene la construcción
 - post-integration-test**: en esta fase “detendremos” todos los servicios o realizaremos las acciones que sean necesarias para volver a restaurar el entorno de pruebas
 - verify**: en esta fase se comprueba que todo está listo (no hay ningún error) para poder copiar el artefacto generado en nuestro repositorio local
 - * Si algún test ha fallado, se detiene la construcción
- En el ciclo de vida por defecto, si el empaquetado es "jar", NO hay ninguna goal asociada a ninguna de las cuatro fases relacionadas con los tests de integración

Default lifecycle

validate
initialize
generate-sources
process-sources
generate-resources
process-resources
compile
process-classes
generate-test-sources
process-test-sources
generate-test-resource
process-test-resources
test-compile
process-test-classes
test
prepare-package
package
pre-integration-test
integration-test
post-integration-test
verify
install
deploy



MAVEN Y PRUEBAS DE INTEGRACIÓN



Default lifecycle

<http://maven.apache.org/surefire/maven-failsafe-plugin/plugin-info.html>

- Ejecución de los tests de integración:
 - Se lleva a cabo mediante la goal **failsafe:integration-test**
 - * Por defecto esta goal está asociada a la fase **integration-test**
 - * se ejecutan los métodos anotados con @Test de las clases ****/IT*.java, **/*IT.java, o **/*ITCase.java**
 - * artefactos generados en **/target/failsafe-reports**
 - Los informes se generan en formato ***.xml**
- Necesitamos incluir el plugin en el fichero pom.xml:

```
<build>
<plugins>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.20</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
</plugins>
</build>
```

mvn verify

asociada a

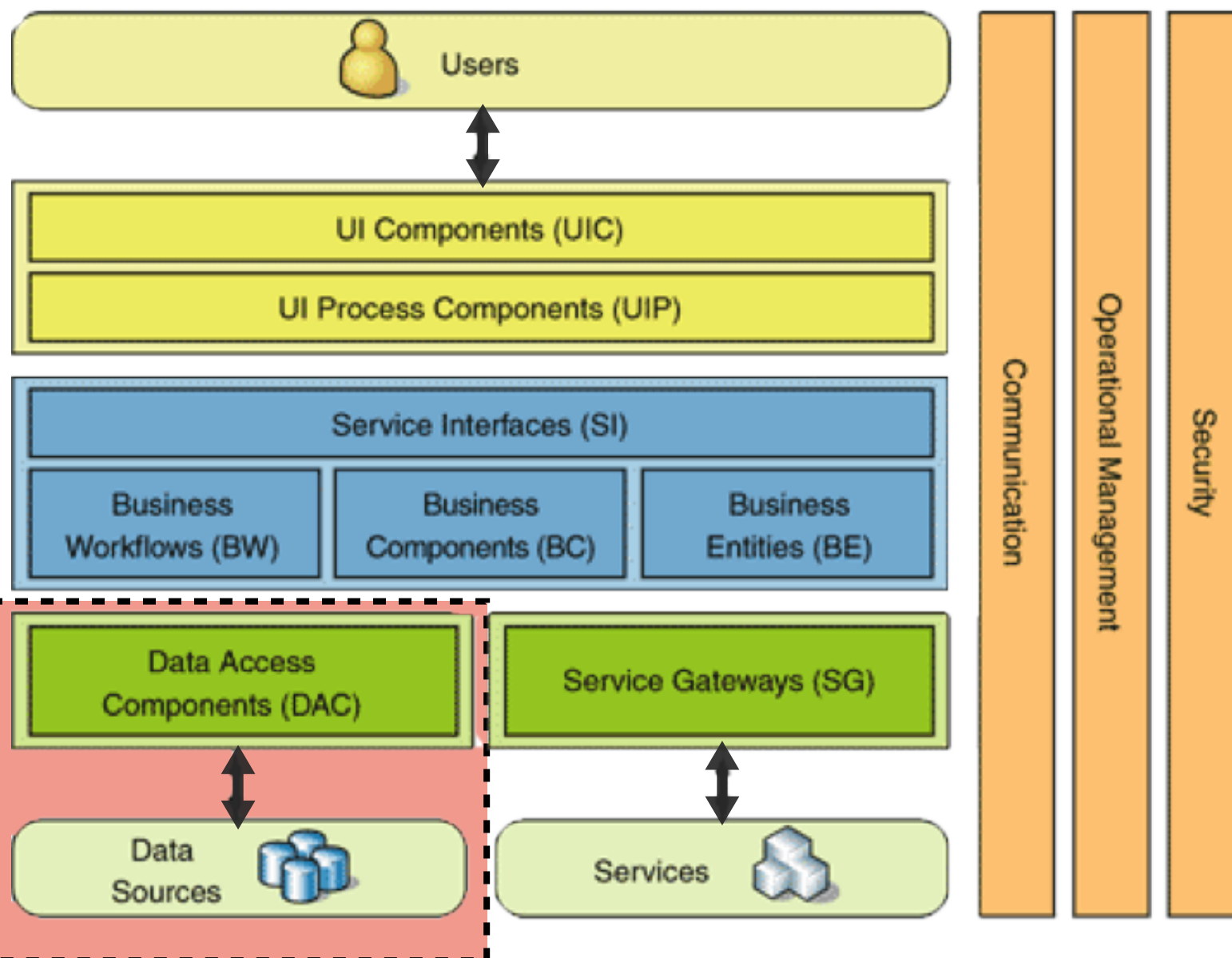
validate
initialize
generate-sources
process-sources
generate-resources
process-resources
compile
process-classes
generate-test-sources
process-test-sources
generate-test-resource
process-test-resources
test-compile
process-test-classes
test
prepare-package
package
pre-integration-test
integration-test
post-integration-test
verify
install
deploy



INTEGRACIÓN CON LA BASE DE DATOS



- La mayor parte de las aplicaciones de empresa utilizan una Base de Datos (BD) como mecanismo de persistencia
 - Una arquitectura software típica de aplicaciones de empresa es una arquitectura por capas, en donde la BD se sitúa entre las capas inferiores



- Las pruebas de integración para dichas aplicaciones requieren que existan datos en la BD para funcionar correctamente
- ¿Cómo podemos realizar pruebas de integración sobre las clases que dependen directamente de dicha BD?
- ¿Cómo podemos asegurarnos de que nuestro código está realmente leyendo y/o escribiendo los datos correctos de dicha BD?
- * La respuesta es... Utilizando **DbUnit**

¿QUÉ ES DBUNIT?

Componentes principales del API:

- IDatabaseTester → JdbcDatabaseTester
- IDatabaseConnection
- DatabaseOperation
- IDataset → FlatXmlDataSet
- ITable
- Assertion



DBUnit NO sustituye a la Base de Datos, sólo nos permite controlar su estado previo y verificar su estado después de la invocación de nuestro SUT

- DbUnit es un **framework** de código abierto creado por Manuel Laflamme basado en JUnit (es, de hecho, una extensión de JUnit)
- DbUnit proporciona una solución “elegante” para controlar la dependencia de las aplicaciones con una base de datos
 - Permite gestionar el estado de una base de datos durante las pruebas
 - Permite ser utilizado juntamente con JUnit
- El escenario típico de ejecución de pruebas con bases de datos utilizando DbUnit es el siguiente:
 1. Eliminar cualquier estado previo de la BD resultante de pruebas anteriores (siempre ANTES de ejecutar cada test)

NO restauramos el estado de la BD después de cada test
 2. Cargar los datos necesarios para las pruebas en la BD

Sólo cargaremos los datos NECESARIOS para cada test
 3. Ejecutar las pruebas utilizando métodos de la librería DbUnit para las aserciones



La ejecución de cada uno de los tests debe ser **INDEPENDIENTE** del resto!!!



INTERFAZ IDATABASETESTER



○ **IDataBaseTester** (*org.dbunit*)

- ❑ Es la interfaz que permite el acceso a la BD, devuelve conexiones de tipo **IDatabaseConnection** con una BD

○ Implementaciones que se pueden utilizar:

- ❑ **JdbcDatabaseTester** - usa un *DriverManager* para obtener conexiones con la BD

○ Métodos que se pueden utilizar:

- ❑ **onSetUp()**, **setSetUpOperation(DatabaseOperation operacion)**
 - * El método **onSetUp()** debe llamarse desde un método anotado con **@Before**
 - * Por defecto, **onSetUp()** realiza una operación **CLEAN_INSERT**. Esto puede cambiarse con el método **setSetUpOperation**
- ❑ **onTearDown()**, **setTearDownOperation(DatabaseOperation operacion)**
- ❑ **setDataSet(IDataSet dataSet)**, **getDataSet()**
 - * Determina los datos de prueba a utilizar y recupera los datos de la BD, respectivamente
- ❑ **getConnection()**
 - * Devuelve la conexión (de tipo **IDatabaseConnection**) con la BD

- Necesitamos un objeto de tipo `IDatabaseTester` para poder obtener una "conexión" con la BD. A partir de dicha conexión podremos interactuar con la BD para realizar las pruebas

```
public class TestDBUnit {
    // databaseTester nos permitirá obtener la conexión con la BD
    private IDatabaseTester databaseTester;
    @Before
    public void setUp() throws Exception {
        // Configuramos las propiedades para poder acceder a la BD:
        // - Clase que implementa el driver
        // - Cadena de conexión con la base de datos
        // - Login y password para acceder a la base de datos
        databaseTester = new JdbcDatabaseTester(
            "jdbc:mysql://localhost:3306/DBUNIT", "root", "");
        ...
        // dataSet contiene los datos que utilizaremos para las pruebas
        databaseTester.setDataSet(dataSet);
        // El método onSetup() realiza internamente una llamada a la operación CLEAN_INSERT sobre
        // la base de datos. Dicha operación inserta en la BD el contenido de la variable dataSet
        databaseTester.onSetup();
        ...
    }
    @Test
    public void testInsert() throws Exception {
        ...
        // recuperamos la conexión con la BD
        IDatabaseConnection connection = databaseTester.getConnection();
        ...
    }
}
```

configuramos los datos de la conexión con la BD

dataset contiene los datos con los que vamos a interactuar con la BD

por defecto se borran todos los datos de las tablas del **dataset**, y se insertan los datos contenidos en el **dataset**

conexión con la BD



CLASE DATABASEOPERATION



DatabaseOperation (org.dbunit.operation)

- ❑ Clase abstracta que define el contrato de la interfaz para operaciones realizadas sobre la BD
- ❑ Utilizaremos un dataset como entrada para una **DatabaseOperation**
- ❑ **DatabaseOperation.CLEAN_INSERT**
 - * Realiza una operación DELETE_ALL, seguida de un INSERT (inserta los contenidos del dataset en la BD. Asume que dichos datos no existen en la BD). Se utiliza en el método **onSetUp()** para inicializar los datos de la BD. Es la forma más segura de garantizar que la BD se encuentre en un estado conocido
- ❑ **DatabaseOperation.REFRESH**
 - * Realiza actualizaciones e inserciones basadas en el dataset. Los datos existentes no incluidos en el dataset no se ven afectados.
- ❑ **DatabaseOperation.DELETE_ALL**
 - * Elimina todas las filas de las tablas especificadas en el dataset. Si en la BD existe alguna tabla no referenciada en el dataset, ésta no se ve afectada
- ❑ **DatabaseOperation.NONE**
 - * No hace nada

Las operaciones sobre la BD: CLEAN_INSERT, REFRESH, DELETE_ALL, NONE, ..., son variables estáticas (son accedidas a través de la clase DatabaseOperation)



INTERFAZ IDATABASECONNECTION



● **IDataBaseConnection** (*org.dbunit.database*)

- ❑ Representa una conexión con una BD (básicamente es un *wrapper* o adaptador de una conexión JDBC)

● Métodos que se pueden utilizar:

- ❑ **createDataSet()** - crea un dataset (conjunto de datos) con todos los datos existentes actualmente en la Base de datos
- ❑ **createDataSet(lista de tablas)** - crea un *dataset* conteniendo las tablas de la BD de la lista de parámetros
- ❑ **createTable(tabla)** - crea un objeto ITable con el resultado de la query "*select * from tabla*" sobre la BD
- ❑ **createQueryTable(tabla, sql)** - crea un objeto ITable con el resultado de la *query sql* sobre la BD
- ❑ **getConfig()** - devuelve un objeto de tipo DatabaseConfig, que contiene parejas de (propiedad, valor) con la configuración de la conexión
- ❑ **getRowCount(tabla)** - devuelve el número de filas de una tabla



EJEMPLO DE USO DE IDATABASECONNECTION



- Necesitamos un objeto de tipo `IDatabaseConnection` para poder interactuar con la BD para realizar las pruebas

```
public class TestDBUnit {  
    // databaseTester nos permitirá obtener la conexión con la BD  
    private IDatabaseTester databaseTester;  
    ...  
    @Test  
    public void testInsert() throws Exception {  
        ...  
        // recuperamos la conexión con la BD  
        IDatabaseConnection connection = databaseTester.getConnection();  
  
        // configuramos la conexión como de tipo mysql  
        DatabaseConfig dbconfig = connection.getConfig();  
        dbconfig.setProperty("http://www.dbunit.org/properties/datatypeFactory",  
                             new MySqlDataTypeFactory());  
  
        //recuperamos TODOS los datos de la BD y los guardamos en un IDataset  
        IDataset databaseDataSet = connection.createDataSet();  
        ...  
    }  
}
```

conexión con la BD

utilizamos la conexión para recuperar TODOS los datos de la BD

obtenemos la configuración de las propiedades de la BD.

Una de las propiedades es "datatypeFactory". Si no se configura, por defecto toma el valor "DefaultDataTypeFactory", que soporta un conjunto limitado de RDBMS



INTERFACES ITable E IDataset



ITable (org.dbunit.dataset)

- Representa una colección de datos tabulares
- Se utiliza en aserciones, para comparar tablas de bases de datos reales con esperadas
- Implementaciones que se pueden utilizar:
 - * **DefaultTable** - ordenación por clave primaria
 - * **SortedTable** - proporciona una vista ordenada de la tabla
 - * **ColumnFilterTable** - permite filtrar columnas de la tabla original

table

id	login	password
1	John	John
2	Karl	Karl

IDataset (org.dbunit.dataset)

- Representa una colección de tablas
- Se utiliza para situar la BD en un estado determinado y para comparar el estado actual de la BD con el estado esperado
- Implementaciones que se pueden utilizar:
 - * **FlatXmlDataSet** - lee/escribe datos en formato xml
 - * **QueryDataSet** - guarda colecciones de datos resultantes de una query
- Métodos que se pueden utilizar:
 - * **getTable(tabla)** - devuelve la tabla especificada

dataset

id	nombre	apellido
1	Ana	Alvarez
2	Carlos	López
3	Pepe	García

id	firstname	street
1	John	1 Main Street

id	login	password
1	John	John
2	Karl	Karl



CLASE FLATXMLDATASET



FlatXmlDataSet (org.dbunit.dataset.xml)

- Lee y escribe documentos XML que contienen conjuntos de tablas de BD con el siguiente formato:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <customer id="1"
    firstname="John"
    street="1 Main Street" />
  <user id="1"
    login="John"
    password="John" />
  <user id="2"
    login="Karl"
    password="Karl" />
</dataset>
```

Tabla customer

id	firstname	street
1	John	1 Main Street

Tabla user

id	login	password
1	John	John
2	Karl	Karl

dataset



CLASE ASSERTION



● Assertion (*org.dbunit*)

- ❑ Clase que define métodos estáticos para realizar aserciones

● Métodos que se pueden utilizar

- ❑ `assertEquals(IDataSet, IDataSet)`
- ❑ `assertEquals(ITable, ITable)`

Ejemplo de uso de IDataSet, ITable y Assertion

```
public class TestDBUnit {
    @Test
    public void testInsert() throws Exception {
        ...
        //recuperamos TODOS los datos de la BD y los guardamos en un IDataSet
        IDataSet databaseDataSet = connection.createDataSet();
        ITable actualTable = databaseDataSet.getTable("customer");

        // introducimos los valores esperados desde el fichero customer-expected.xml
        DataFileLoader loader = new FlatXmlDataFileLoader();
        IDataSet expectedDataSet = loader.load("/customer-expected.xml");

        ITable expectedTable = expectedDataSet.getTable("customer");

        //comparamos la tabla esperada con la real
        Assertion.assertEquals(expectedTable, actualTable);
    }
}
```


Para utilizar DbUnit en nuestros drivers en un proyecto Maven tendremos que incluir en el fichero de configuración (pom.xml) las siguientes dependencias (además de JUnit 4), y el plugin failsafe

```
<dependency>
  <groupId>org.dbunit</groupId>
  <artifactId>dbunit</artifactId>
  <version>2.5.4</version>
  <scope>test</scope>
</dependency>
```

Librería DbUnit

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.46</version>
  <scope>test</scope>
</dependency>
```

Librería para acceder a una BD MySql

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>2.20</version>
      <executions>
        <execution>
          <goals>
            <goal>integration-test</goal>
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Plugin *failsafe* para ejecutar los tests de integración

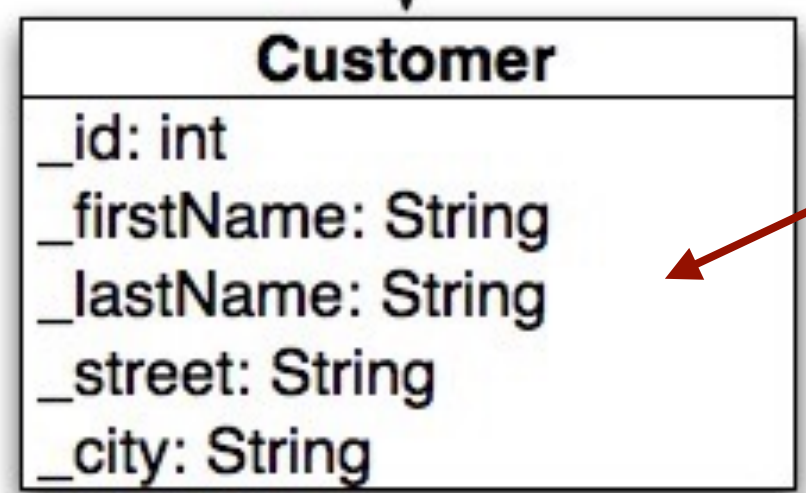
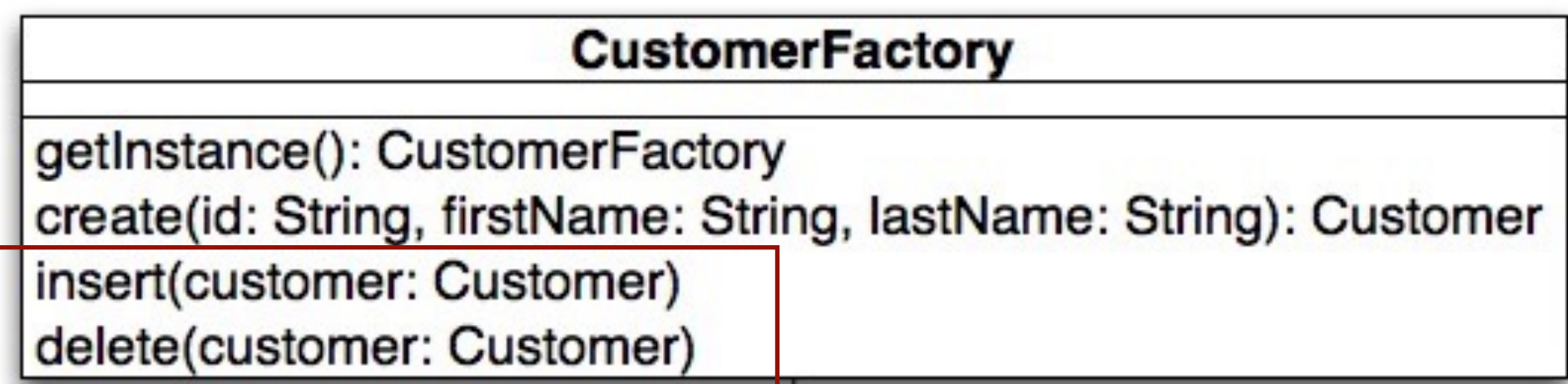


Los drivers que usan DbUnit implementan pruebas de INTEGRACIÓN, por lo tanto los tests se ejecutarán en la fase INTEGRATION-TEST
Necesitamos añadir el plugin FAILSAFE para ejecutar los tests de integración



EJEMPLO DE TEST DE INTEGRACIÓN

- La clase CustomerFactory depende de la base de datos
 - concretamente los métodos insert() y delete()



Sesión 7: Pruebas de integración

Los métodos insert y delete, insertan y borran (respectivamente) un cliente en la base de datos MySql, a la que llamaremos DBUNIT

La clase Customer implementa getters y setters para los atributos de la clase



IMPLEMENTACIÓN DE LOS DRIVERS



- Vamos a ver cómo implementar un *driver* para probar el método `CustomerFactory.insert()`
- ANTES DE CADA TEST, eliminamos cualquier estado previo de la BD utilizando el método `IDatabaseTester.onSetup()`

```
public class ITTestDBUnit {
```

```
//Clase a probar
```

```
private CustomerFactory _customerFactory;
```

```
public static final String TABLE_CUSTOMER = "customer";
```

```
private IDatabaseTester databaseTester;
```

```
@Before
```

```
public void setUp() throws Exception {
```

```
// fijamos los datos para acceder a la BD
```

```
databaseTester = new JdbcDatabaseTester("jdbc:mysql://localhost:3306/DBUNIT", "root", "");
```

```
// inicializamos el dataset
```

```
DataFileLoader loader = new FlatXmlDataFileLoader();
```

```
IDataSet dataSet = loader.load("/customer-init.xml");
```

```
databaseTester.setDataSet(dataSet);
```

```
// llamamos a la operación por defecto setUpOperation
```

```
databaseTester.onSetup();
```

```
_customerFactory = CustomerFactory.getInstance();
```

```
...
```

Necesitamos una instancia de `IDatabaseTester` para acceder a la BD

Datos para la conexión con la BD

Dataset inicial: tabla de clientes VACÍA

Borra los datos de las tablas del dataset inicial en la BD e inserta en la BD el contenido del dataset inicial

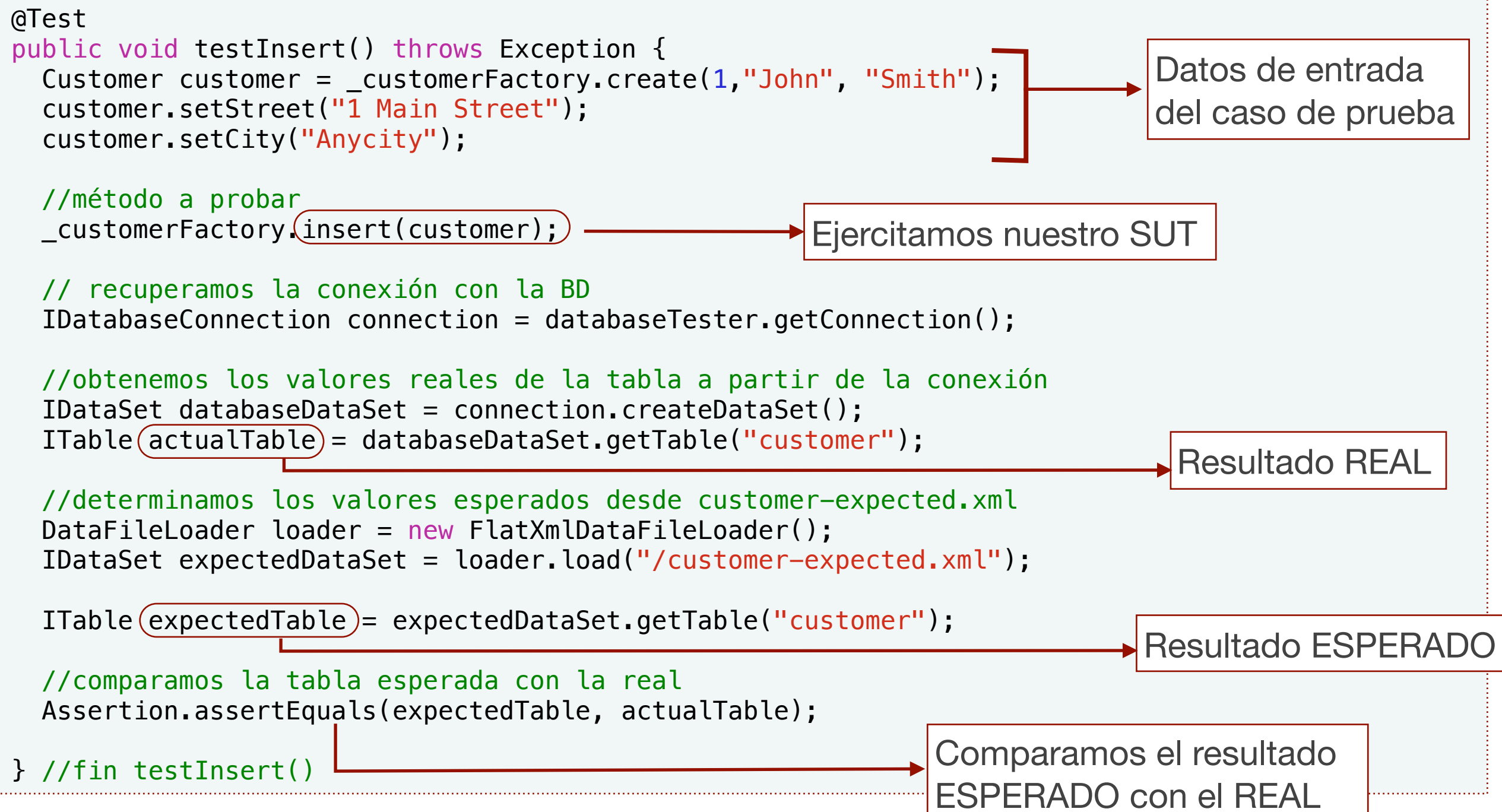
Instancia que contiene nuestro SUT



IMPLEMENTACIÓN DEL CASO DE PRUEBA



Probaremos la inserción de un cliente en una base de datos vacía

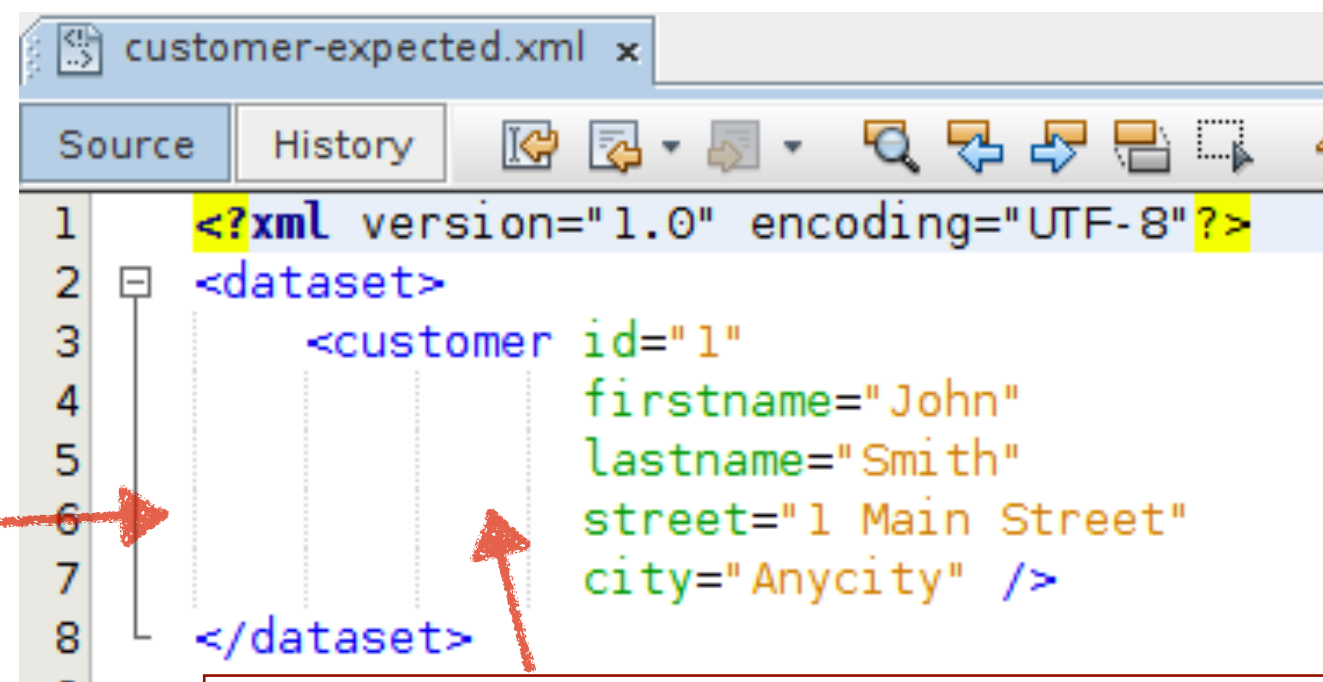
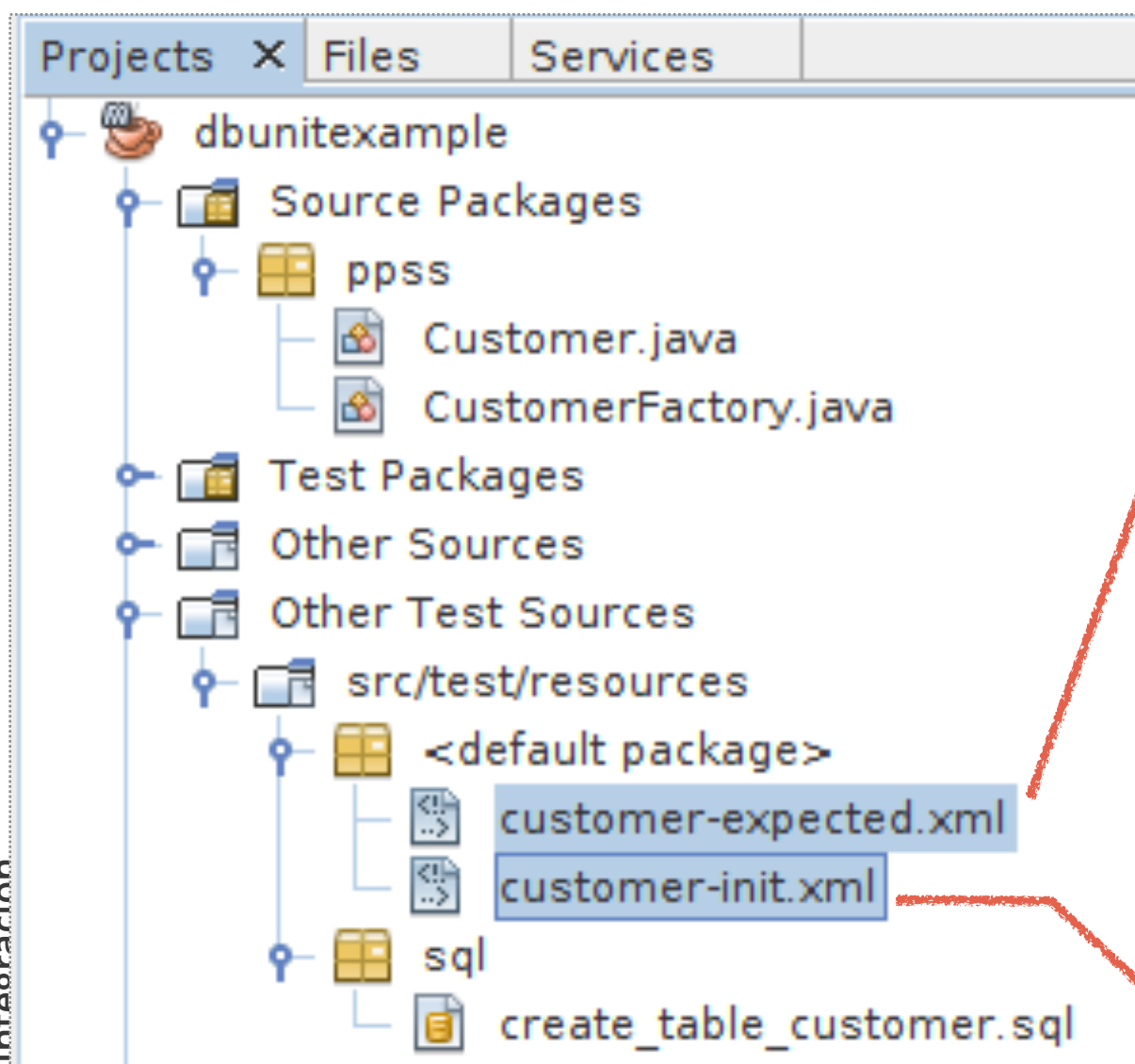




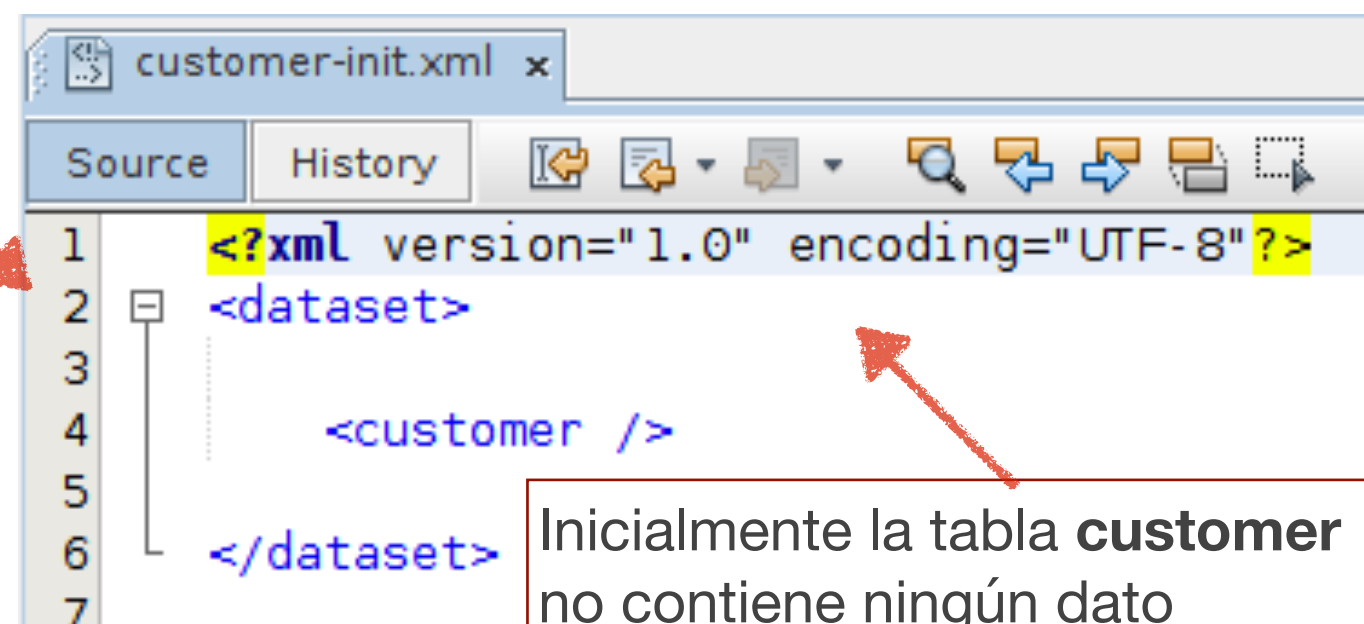
DATOS DE ENTRADA Y RESULTADO ESPERADO



- Los datos de entrada y el resultado esperado los almacenamos en ficheros de recursos xml que convertiremos en un IDataset



Resultado esperado: la tabla **customer** contiene únicamente el cliente insertado



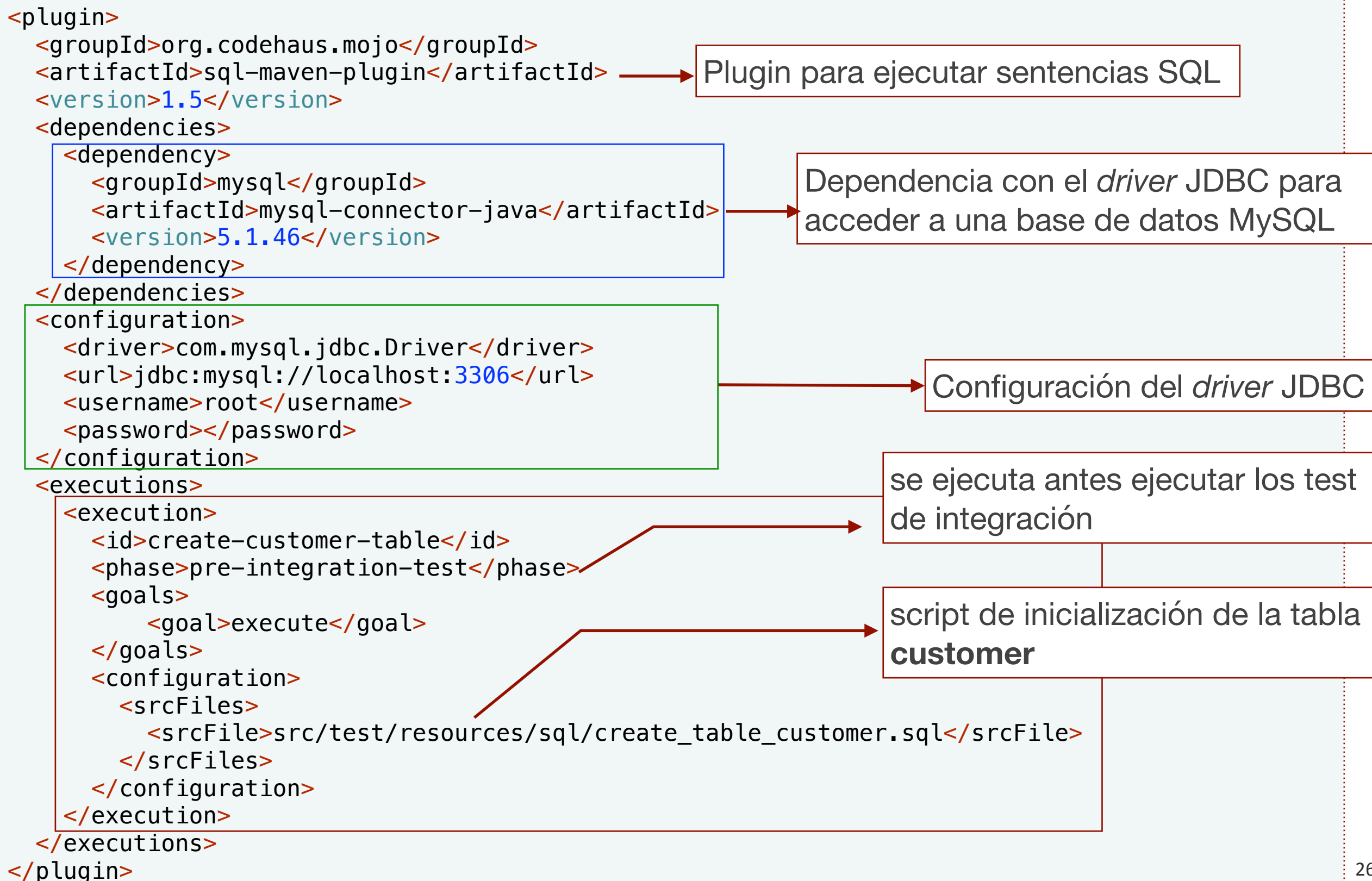
Inicialmente la tabla **customer** no contiene ningún dato



USO DEL PLUGIN SQL-MAVEN-PLUGIN



Podemos configurar el pom para inicializar las tablas de nuestra BD:





INICIALIZACIÓN DE LA TABLA CUSTOMER



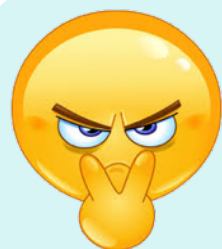
Podemos configurar el pom para inicializar las tablas de nuestra BD:

The screenshot shows an IDE with a project named 'dbunitexample'. The project structure is as follows:

- src
 - main
 - test
 - java
 - ppss
 - CustomerFactoryIT.java
 - resources
 - sql
 - create_table_customer.sql (highlighted with a red circle and an arrow pointing to the SQL editor)
 - customer-expected.xml
 - customer-init.xml
 - target
 - nbactions.xml
 - pom.xml

The SQL editor on the right shows the following script:

```
1 DROP DATABASE IF EXISTS DBUNIT;  
2 CREATE DATABASE IF NOT EXISTS DBUNIT;  
3 USE DBUNIT;  
4  
5 DROP TABLE IF EXISTS customer;  
6  
7 CREATE TABLE customer (  
8     id int(11) NOT NULL,  
9     firstname varchar(45) DEFAULT NULL,  
10    lastname varchar(45) DEFAULT NULL,  
11    street varchar(45) DEFAULT NULL,  
12    city varchar(45) DEFAULT NULL,  
13    PRIMARY KEY (id)  
14 );  
15
```



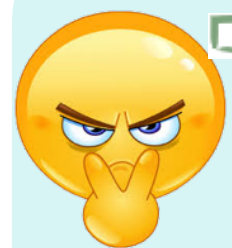
La carpeta **src/test/resources** almacenará cualquier fichero adicional (fichero no java) que necesites utilizar para ejecutar tu código de pruebas (desde src/test/java). Dentro de esta carpeta puedes crear cuantos subdirectorios consideres oportunos. Por ejemplo, hemos creado el subdirectorio **sql** con el script sql para inicializar la BD. Maven copia los recursos situados en src/test/resources al directorio target, durante la fase **process-test-resources**.

PROCESO COMPLETO PARA AUTOMATIZAR LAS PRUEBAS DE INTEGRACIÓN CON MAVEN

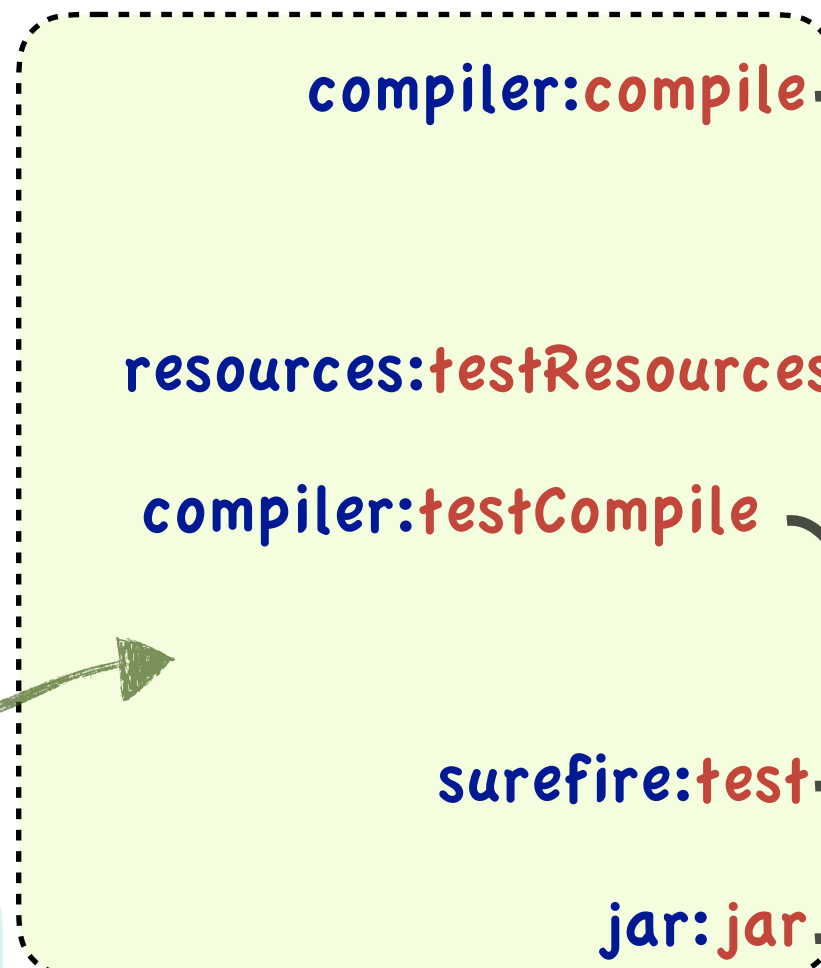


Es importante, como tester, que tengas claro qué acciones tienen

que realizarse una vez implementados los tests, para poder ejecutar de forma automática dichos drivers. Si trabajamos con Maven, utilizaremos el ciclo de vida por defecto, y aprovecharemos las goals asociadas por **defecto**



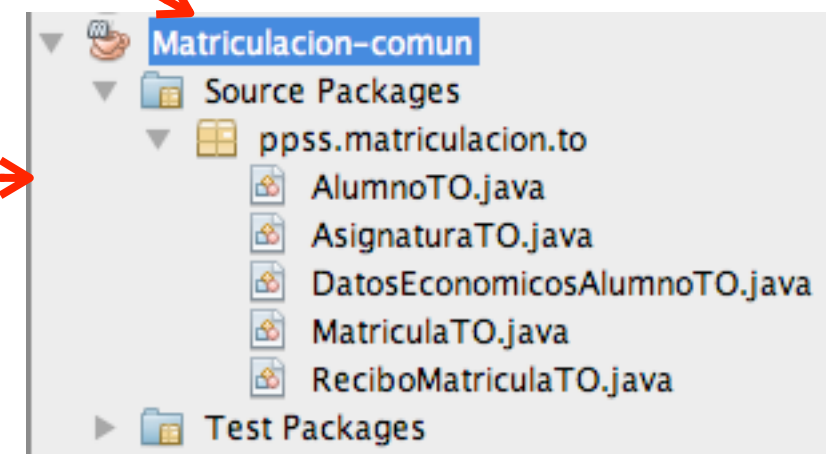
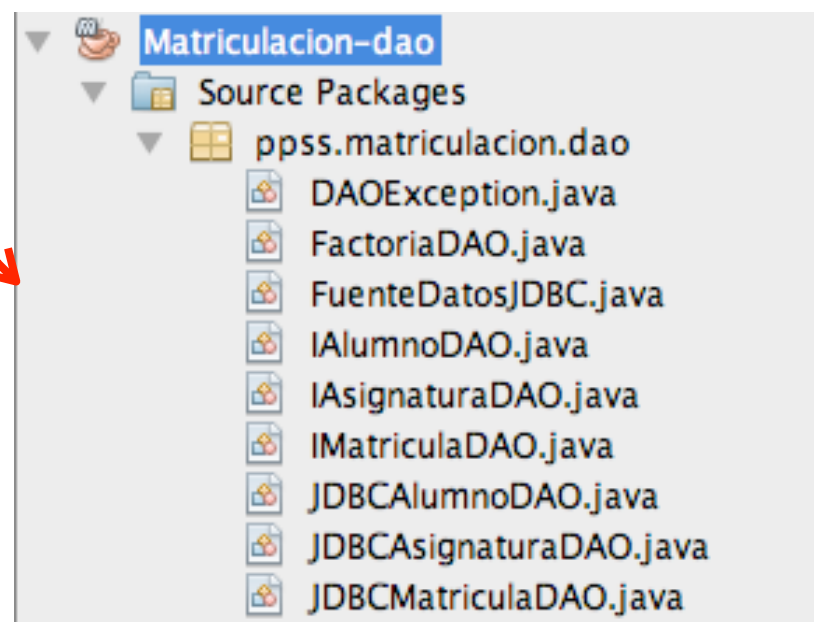
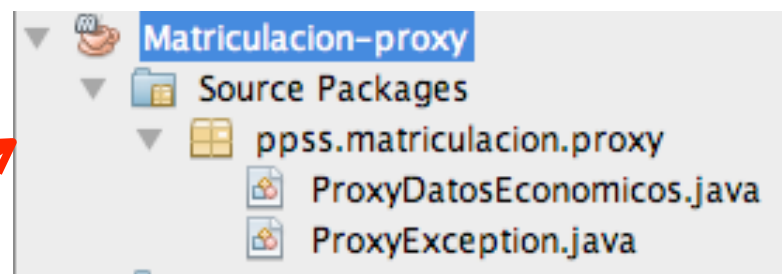
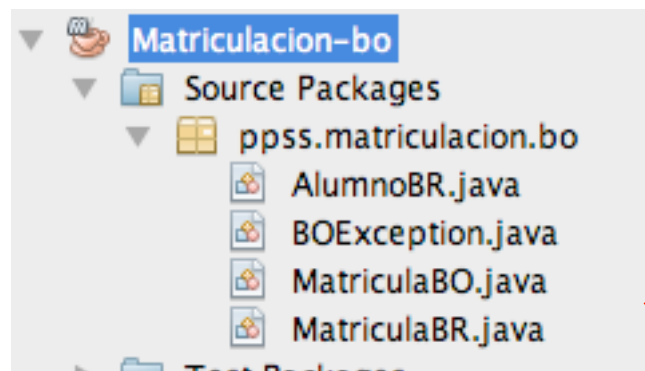
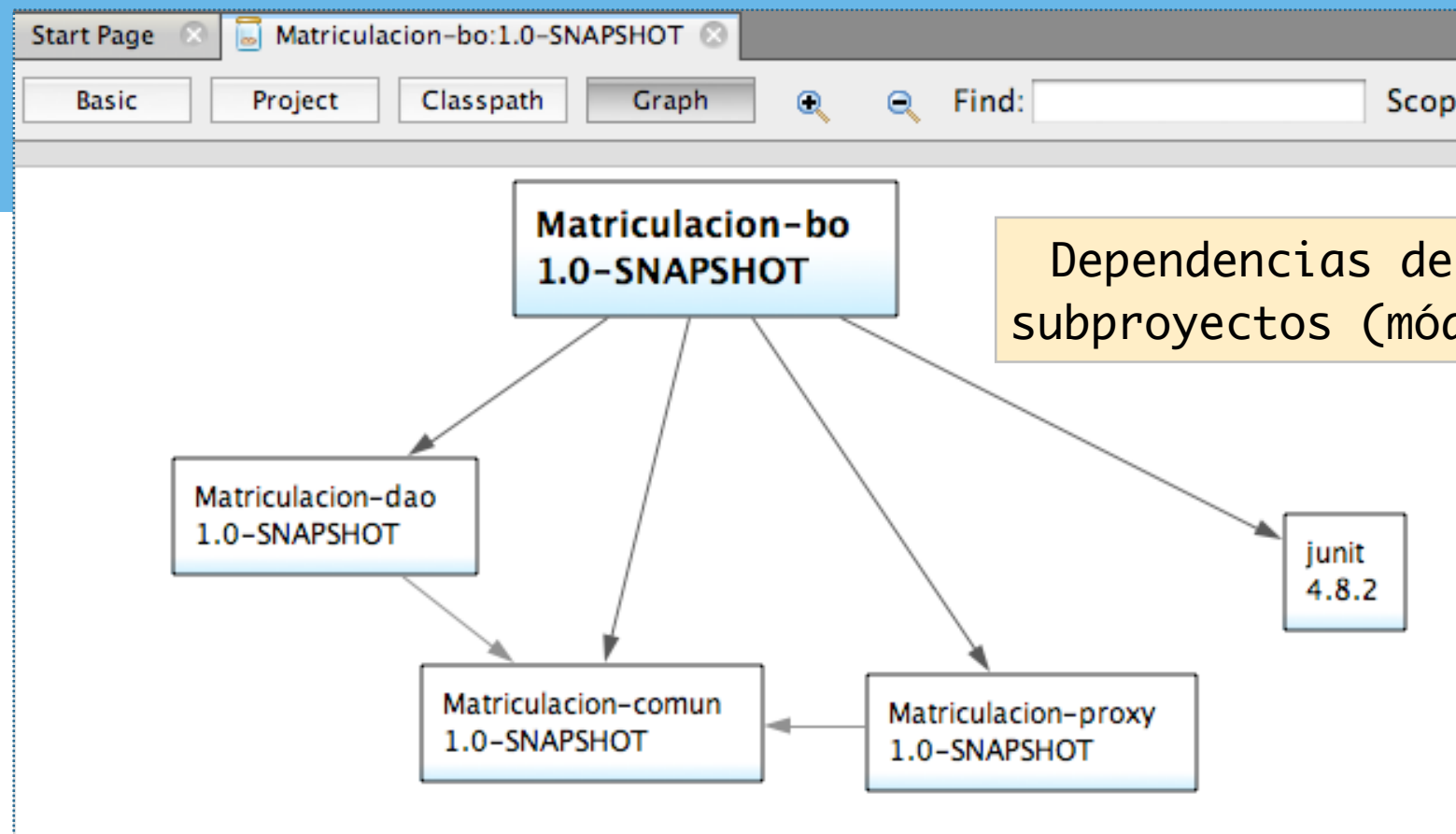
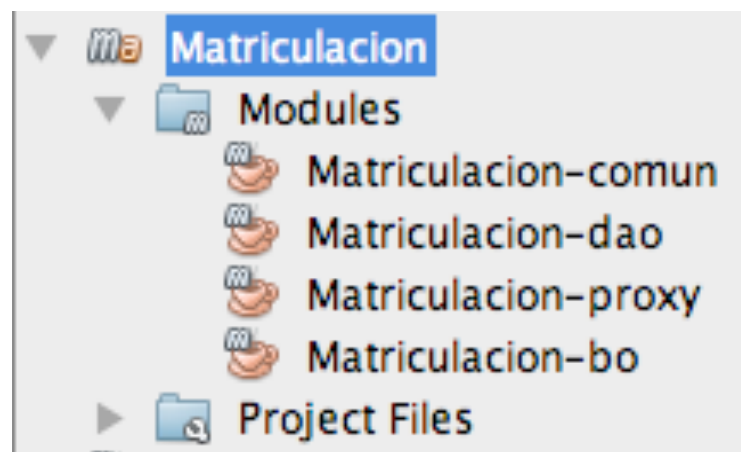
Las fases pre-integration-test ... verify, por defecto **NO** tienen asociada ninguna goal cuando el empaquetado del proyecto es jar, por lo que tendremos que incluir y configurar los plugins necesarios en el **pom** del proyecto



validate
initialize
generate-sources
process-sources
generate-resources
process-resources
compile
process-classes
generate-test-sources
process-test-sources
generate-test-resources
process-test-resources
test-compile
process-test-classes
test
prepare-package
package
pre-integration-test
integration-test
post-integration-test
verify
install
deploy

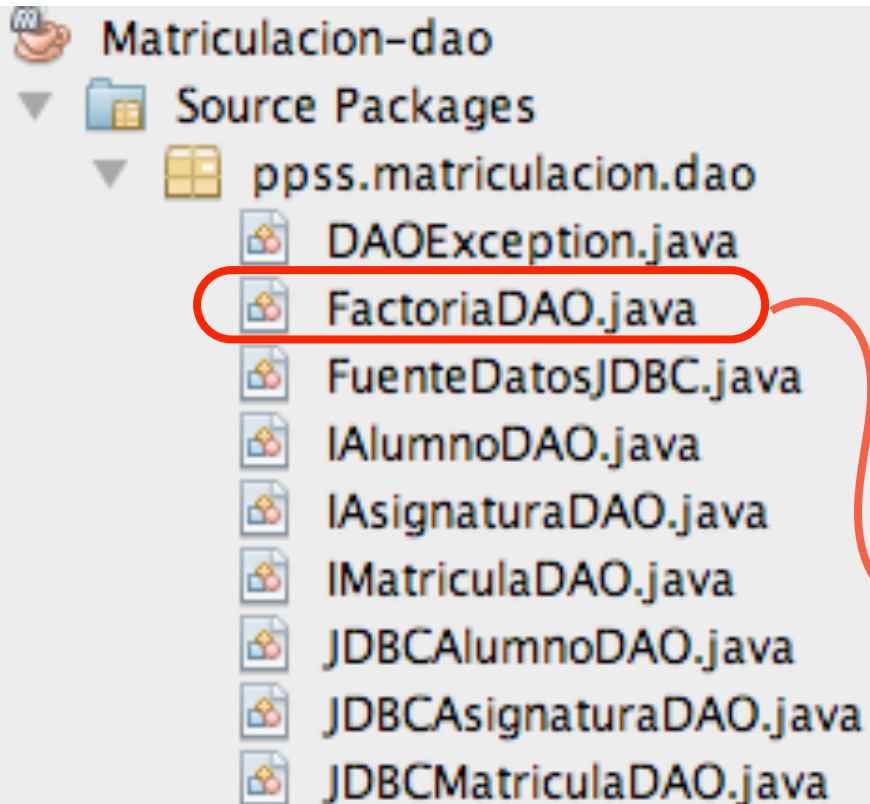
EJEMPLO

Proyecto Matriculacion



CAPA DE ACCESO A DATOS: FACTORIA DE DAOS

Usamos una clase factoría que proporciona los DAOs utilizados por los objetos de la capa de negocio (*Business Objects*)



FactoriaDAO proporciona objetos DAOs JDBC de los siguientes tipos:

- ❑ JDBCAlumnoDAO
- ❑ JDBCAsignaturaDAO
- ❑ JDBCMatriculaDAO

```
Start Page x FactoriaDAO.java x
Source History
1 package ppss.matriculacion.dao;
2
3 /**
4  * La clase <code>GestorDAO</code> se utilizará como factoría de los ob
5  * a datos de la aplicación.
6  */
7
8 public class FactoriaDAO {
9     /**
10      * Devuelve al DAO para acceder a los datos de los alumnos.
11      * @return DAO que da acceso a los alumnos.
12      */
13     public IAlumnoDAO getAlumnoDAO() {
14         return new JDBCAlumnoDAO();
15     }
16
17     /**
18      * Devuelve al DAO para acceder a los datos de las asignaturas.
19      * @return DAO que da acceso a las asignaturas.
20      */
21     public IAsignaturaDAO getAsignaturaDAO() {
22         return new JDBCAsignaturaDAO();
23     }
24
25     /**
26      * Devuelve al DAO para acceder a los datos de matriculación.
27      * @return DAO que da acceso a las matriculaciones.
28      */
29     public IMatriculaDAO getMatriculaDAO() {
30         return new JDBCMatriculaDAO();
31     }
32 }
33
```


CAPA DE ACCESO A DATOS: IMPLEMENTACIÓN DE DAOS

Cada DAO implementa una interfaz (operaciones CRUD sobre la BD)

JDBCAlumnoDAO implementa la interfaz IAlumnoDAO

```
Start Page x FactoriaDAO.java x JDBCAlumnoDAO.java x FuenteDatosJDBC.java x
Source History
1 package ppss.matriculacion.dao;
2
3 import java.sql.Connection;
4 import java.sql.PreparedStatement;
5 import java.sql.ResultSet;
6 import java.sql.SQLException;
7 import java.util.ArrayList;
8 import java.util.List;
9
10 import ppss.matriculacion.to.AlumnoTO;
11
12 /**
13  * Implementación del acceso a los datos de los alumnos almacenados en una base de
14  * datos.
15  */
16
17 public class JDBCAlumnoDAO implements IAlumnoDAO {
18
19     public void addAlumno(AlumnoTO a) throws DAOException {
20         Connection con = null;
21         PreparedStatement ps = null;
22
23         if(a==null) {
24             throw new DAOException("El alumno a insertar no puede ser nulo")
25         }
26
27         try {
28             con = FuenteDatosJDBC.getInstance().getConnection();
29             ps = con.prepareStatement("INSERT INTO alumnos(nif, nombre, dir, email, fechaNacimiento) VALUES(?, ?, ?, ?, ?)");
30             ps.setString(1, a.getNif());
31             ps.setString(2, a.getNombre());
32             ps.setString(3, a.getDireccion());
33             ps.setString(4, a.getEmail());
34             ps.setDate(5, new java.sql.Date(a.getFechaNacimiento().getTime()));
35         } catch (SQLException e) {
36             throw new DAOException(e.getMessage());
37         } finally {
38             con.close();
39             ps.close();
40         }
41     }
42 }
```

JDBCAlumnoDAO implementa operaciones CRUD de los objetos TO, p.ej. addAlumno

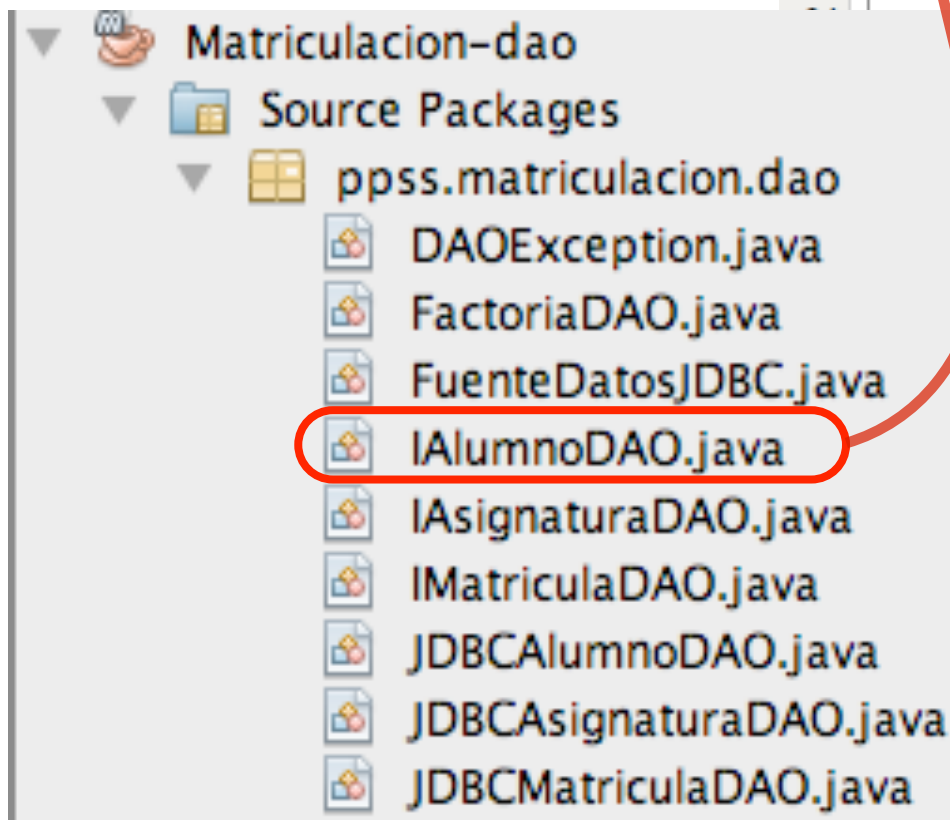
CAPA DE ACCESO A DATOS: INTERFACES DE LOS DAOS

La interfaz IAlumnoDAO permite que los cambios en la implementación de la fuente de datos subyacente NO afecten a los componentes de negocio.



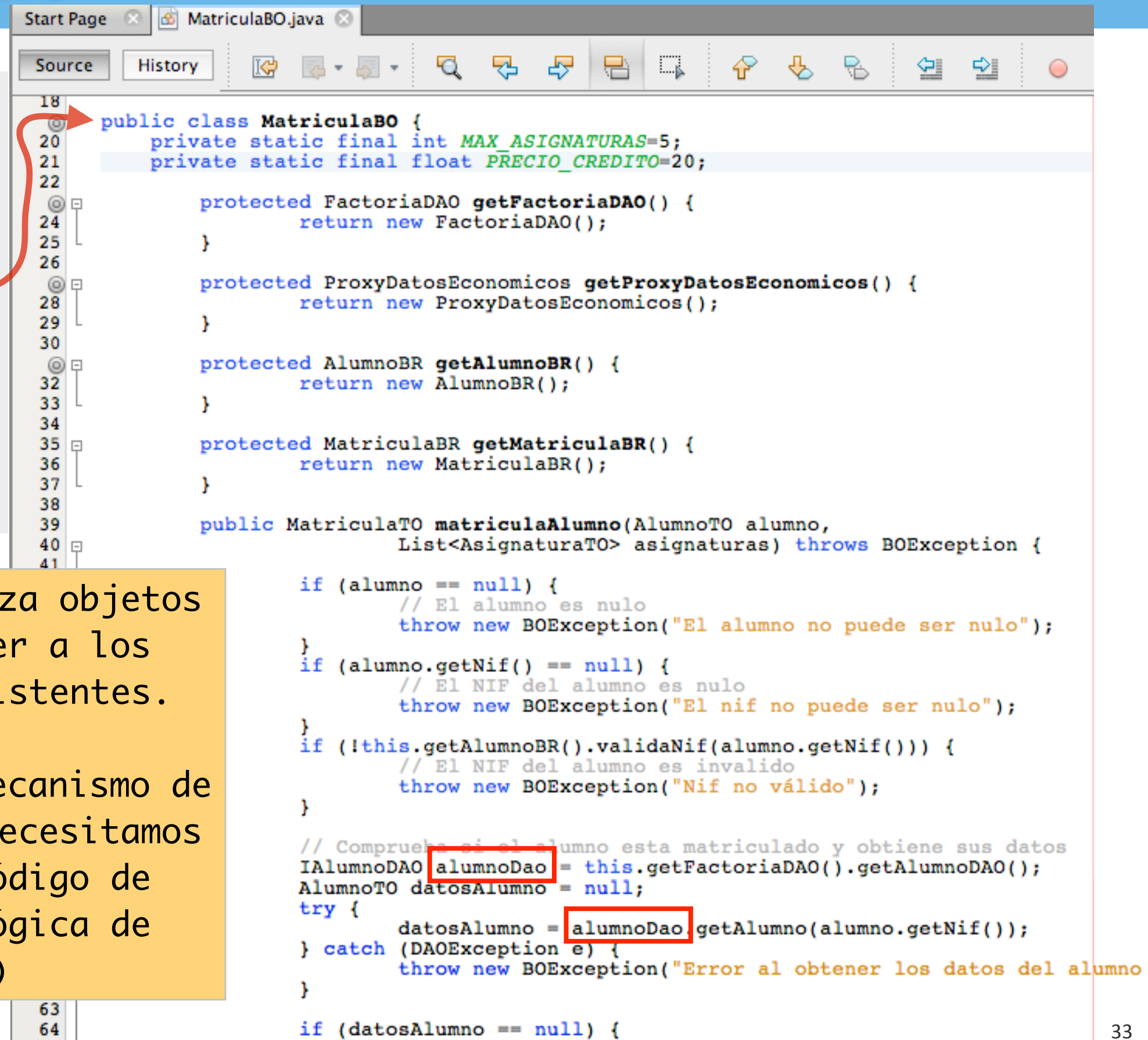
Los objetos de la capa de negocio son CLIENTES de la capa de acceso a datos

```
Start Page x FactoriaDAO.java x JDBCAlumnoDAO.java x FuenteDatosJDBC.java x IAlumnoDAO.java
Source History
1 package ppss.matriculacion.dao;
2
3 import java.util.List;
4 import ppss.matriculacion.to.AlumnoTO;
5
6
7 /**
8  * Interfaz común de los objetos que dan acceso a los datos de los alumnos.
9  *
10  */
11 public interface IAlumnoDAO {
12
13     /**
14      * De de alta una alumno.
15      * @param a Alumno que se añadirá. Se producirá un error si el alumno
16      * ya existe, o si el parámetro o su campo <code>nif</code> es <code>nu
17      * @throws DAOException Si ocurre un error al añadir al alumno
18      */
19     public abstract void addAlumno(AlumnoTO a) throws DAOException;
20
21     /**
22      * De de baja un alumno.
23      * @param nif Nif del alumno a eliminar. Se producirá un error si el al
24      * existe, o si el parámetro es <code>null</code>.
25      * @throws DAOException Si ocurre un error al eliminar al alumno
26      */
27     public abstract void delAlumno(String nif) throws DAOException;
28
29     /**
30      * Obtiene los datos de un alumno.
31      * @param nif Nif del alumno del cual queremos obtener los datos
32      * @return Datos del alumno, o <code>null</code> si el alumno no existe
33      * @throws DAOException Si ocurre un error al recuperar los datos
34      */
35     public abstract AlumnoTO getAlumno(String nif) throws DAOException;
```



IAlumnoDAO especifica las operaciones que se pueden realizar sobre la BD. Operaciones CRUD (Create, Retrieve, Update, Delete)

CAPA DE NEGOCIO



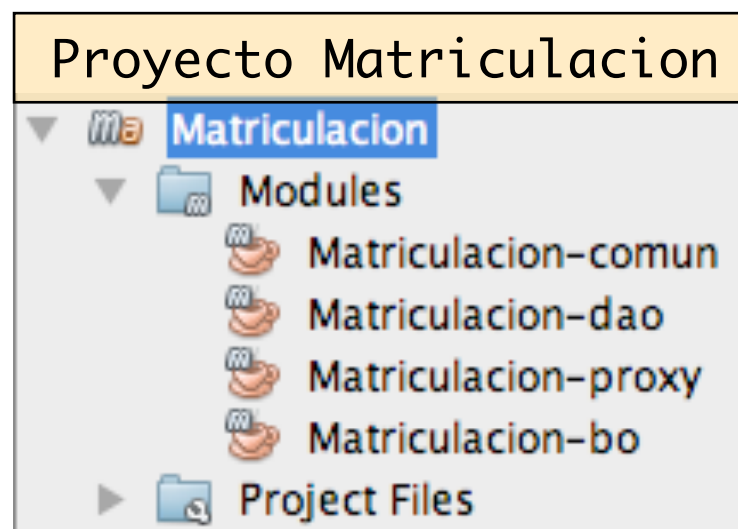
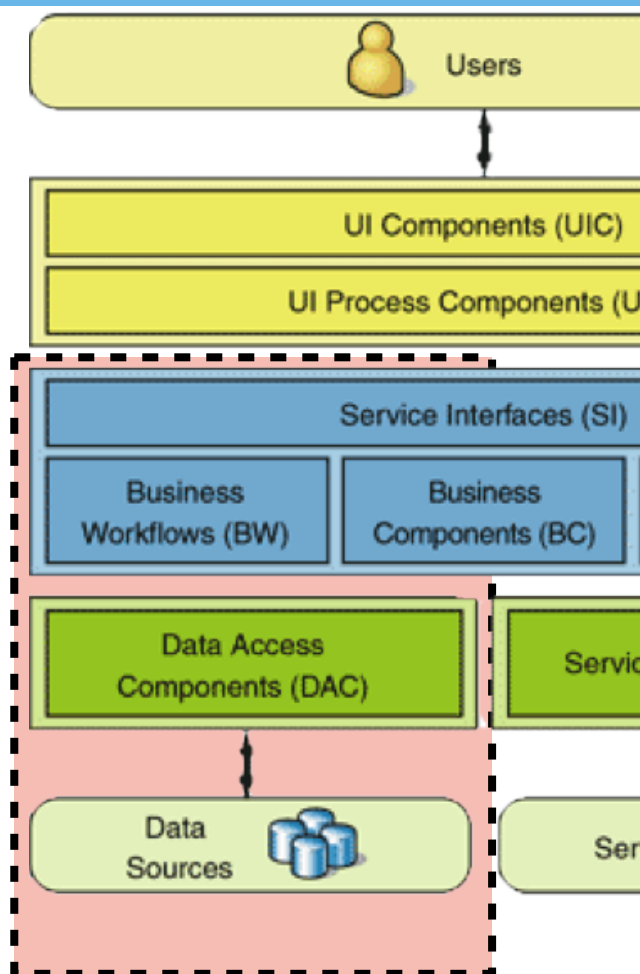
```
18 public class MatriculaBO {
19     private static final int MAX_ASIGNATURAS=5;
20     private static final float PRECIO_CREDITO=20;
21
22     protected FactoriaDAO getFactoriaDAO() {
23         return new FactoriaDAO();
24     }
25
26     protected ProxyDatosEconomicos getProxyDatosEconomicos() {
27         return new ProxyDatosEconomicos();
28     }
29
30     protected AlumnoBR getAlumnoBR() {
31         return new AlumnoBR();
32     }
33
34     protected MatriculaBR getMatriculaBR() {
35         return new MatriculaBR();
36     }
37
38     public MatriculaTO matriculaAlumno(AlumnoTO alumno,
39         List<AsignaturaTO> asignaturas) throws BOException {
40
41         if (alumno == null) {
42             // El alumno es nulo
43             throw new BOException("El alumno no puede ser nulo");
44         }
45         if (alumno.getNif() == null) {
46             // El NIF del alumno es nulo
47             throw new BOException("El nif no puede ser nulo");
48         }
49         if (!this.getAlumnoBR().validaNif(alumno.getNif())) {
50             // El NIF del alumno es invalido
51             throw new BOException("Nif no válido");
52         }
53
54         // Comprueba si el alumno esta matriculado y obtiene sus datos
55         IAlumnoDAO alumnoDao = this.getFactoriaDAO().getAlumnoDAO();
56         AlumnoTO datosAlumno = null;
57         try {
58             datosAlumno = alumnoDao.getAlumno(alumno.getNif());
59         } catch (DAOException e) {
60             throw new BOException("Error al obtener los datos del alumno");
61         }
62
63         if (datosAlumno == null) {
64
```

MatriculaBO utiliza objetos DAO para acceder a los objetos TO persistentes.

Si cambiamos el mecanismo de persistencia NO necesitamos modificar el código de MatriculaBO (lógica de negocio)



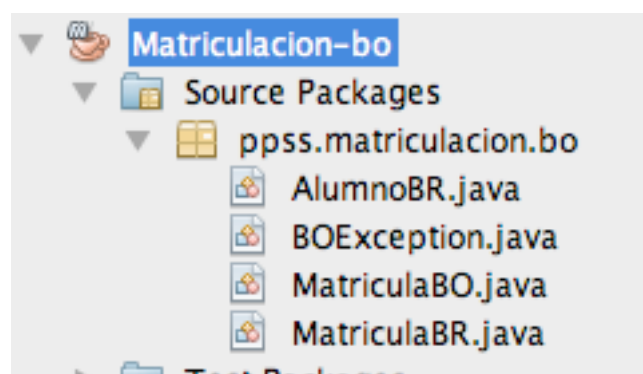
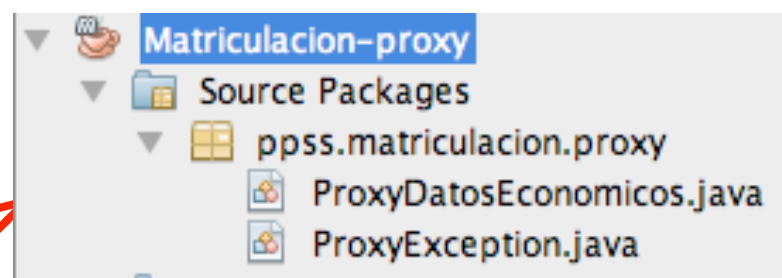
EJEMPLO DE ESTRATEGIA DE INTEGRACIÓN



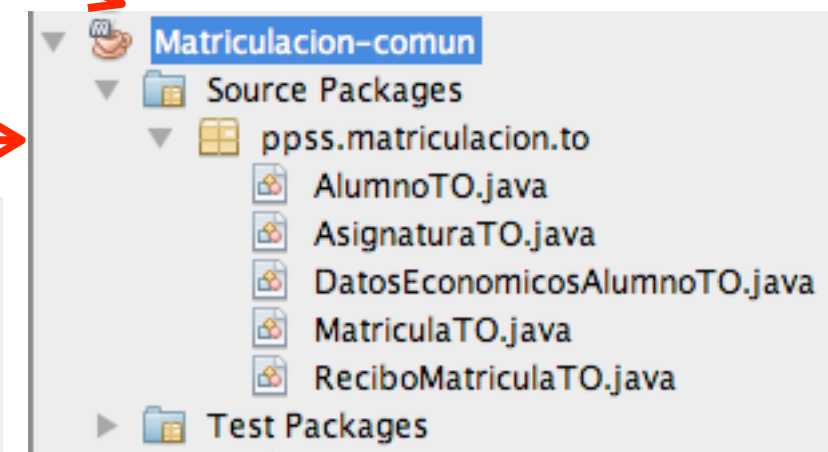
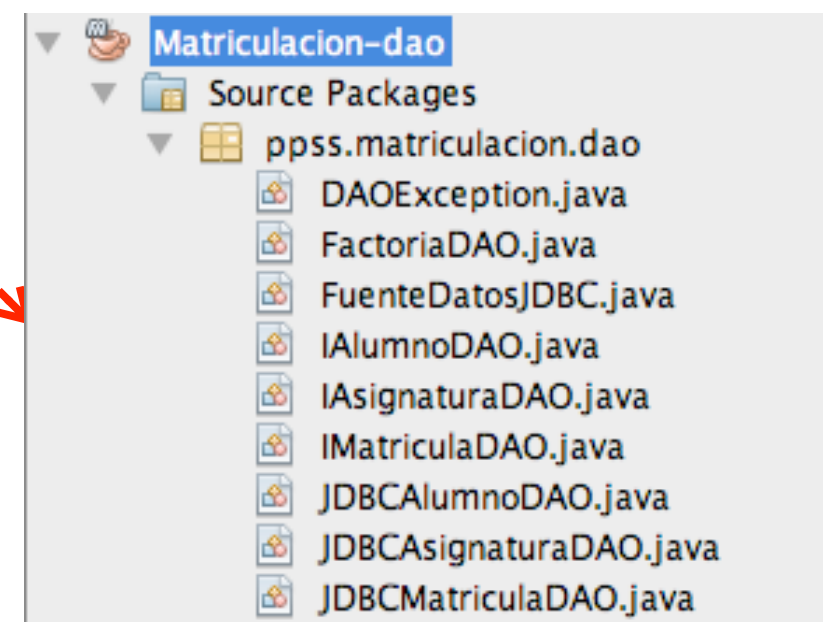
Posible estrategia de integración

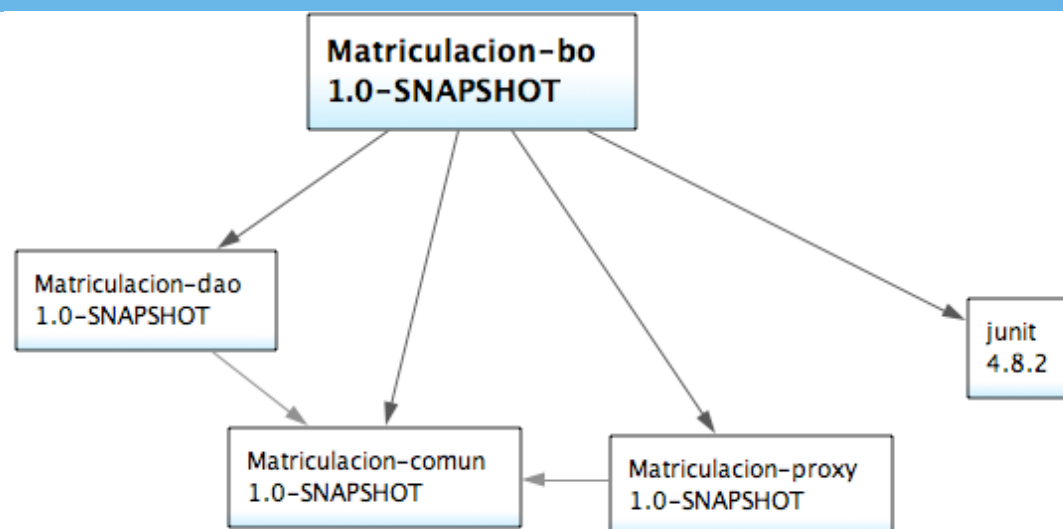
Bottom up

- 1 Matriculacion-dao
- 2 Matriculacion-bo + Matriculacion-dao
- 3 Matriculacion-bo + Matriculacion-dao + Matriculacion-proxy



Dependencias de los subproyectos





El orden de ejecución de los tests debería ser: 1, 2, 3
Para ello necesitaríamos agrupar en categorías los tests de Matriculación-bo

`mvn verify -Dgroups="Integracion-fase1"`
`mvn verify -Dgroups="Integracion-fase2"`
`mvn verify -Dgroups="Integracion-fase3"`

Para automatizar las pruebas necesitaremos implementar en cada una de las fases de la integración:

1

Matriculacion-dao

Utilizamos DbUnit para implementar Tests de integración con la BD:

`/src/test/java/AlumnoDAOIT.java, /src/test/java/AsignaturaDAOIT.java,`
`/src/test/java/MatriculaDAOIT.java`

implementados en el proyecto Matriculacion-dao
 categoría Integracion-fase1

`mvn verify -Dgroups="Integracion-fase1"`

2

Matriculacion-bo + Matriculacion-dao

Implementamos:

- stubs para Matriculación-proxy
- tests DBUnit:

implementados en el proyecto Matriculacion-bo

categoría Integracion-fase2

`/src/test/java/MatriculaB0-daoIT.java`

`mvn verify -Dgroups="Integracion-fase2"`

3

Matriculacion-bo + Matriculacion-dao +
 Matriculacion-proxy

Implementamos:

- tests DBUnit:

implementados en el proyecto Matriculacion-bo

categoría Integracion-fase3

`/src/test/javaMatriculaB0-dao-proxyIT.java`

`mvn verify -Dgroups="Integracion-fase3"`



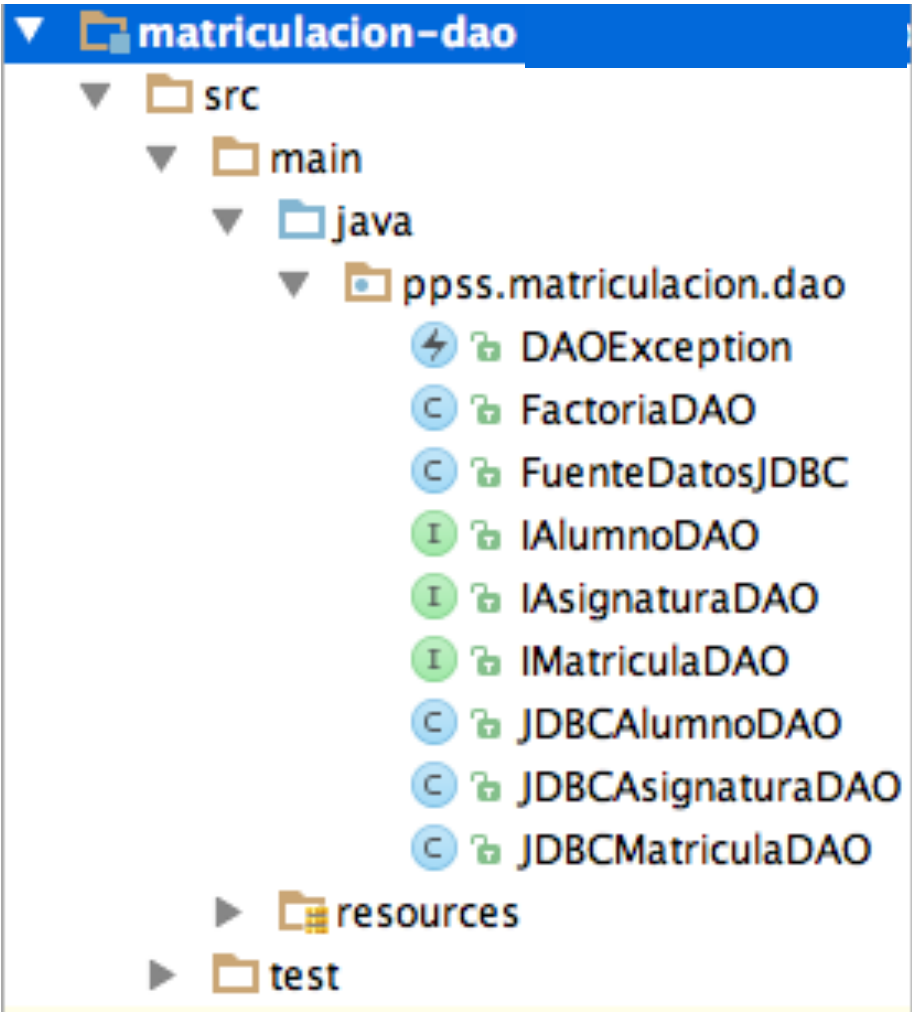
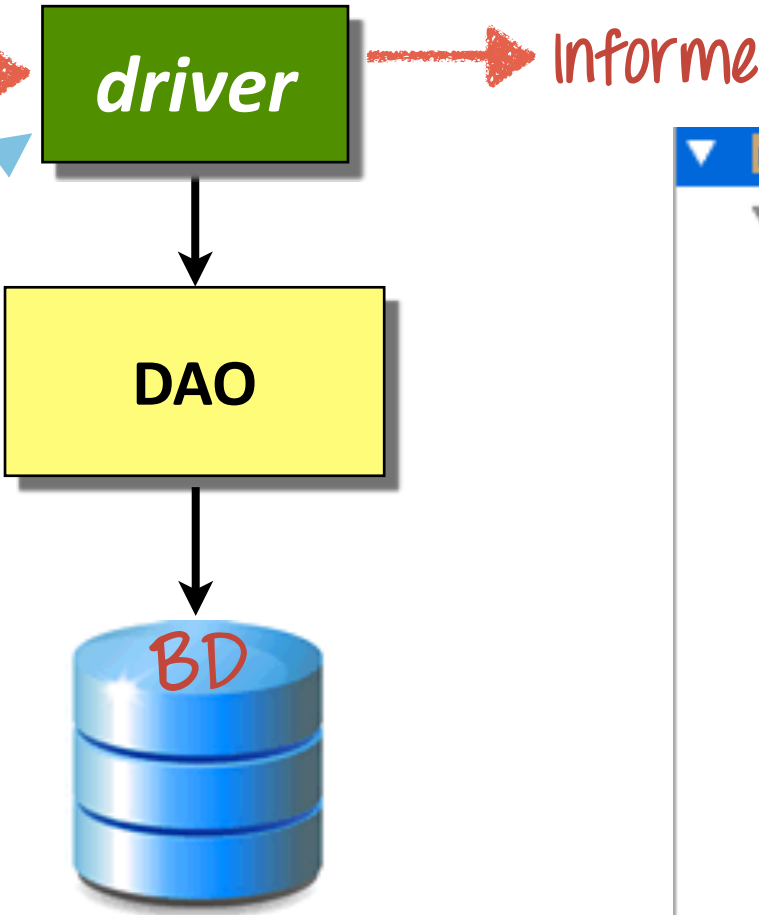
Y AHORA VAMOS AL LABORATORIO...

Vamos a implementar tests de integración con una BD MySql utilizando DbUnit

Tests integración

Camino	Datos Entrada	Resultado Esperado	Resultado Real
C1	d1=... d2=... ..	r1	
..			
CM	d1=... d2=... ..	rM	

Usaremos la librería DbUnit para implementar los tests





- Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
 - Capítulo 7: System Integration Testing
- Software Engineering. 9th edition. Ian Sommerville. 2011
 - Capítulo 8.1.3: Component testing
- DUnit (sitio oficial)
 - <http://dbunit.sourceforge.net/>
- Core J2EE Patterns - Data Access Object
 - <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>