Planificación y Pruebas de Sistemas Software

Curso 2015-16

Sesión 7: Pruebas de integración

Pruebas de integración
Integración con la base de datos
Automatización de las pruebas con DbUnit
Definición y componentes DbUnit
Ejemplo
Patrón DAO

María Isabel Alfonso Galipienso
Universidad de Alicante eli@ua.es

Niveles de pruebas

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

VERIFICACIÓN
¿está implementado correctamente?

VALIDACIÓN
¿el producto es el correcto?

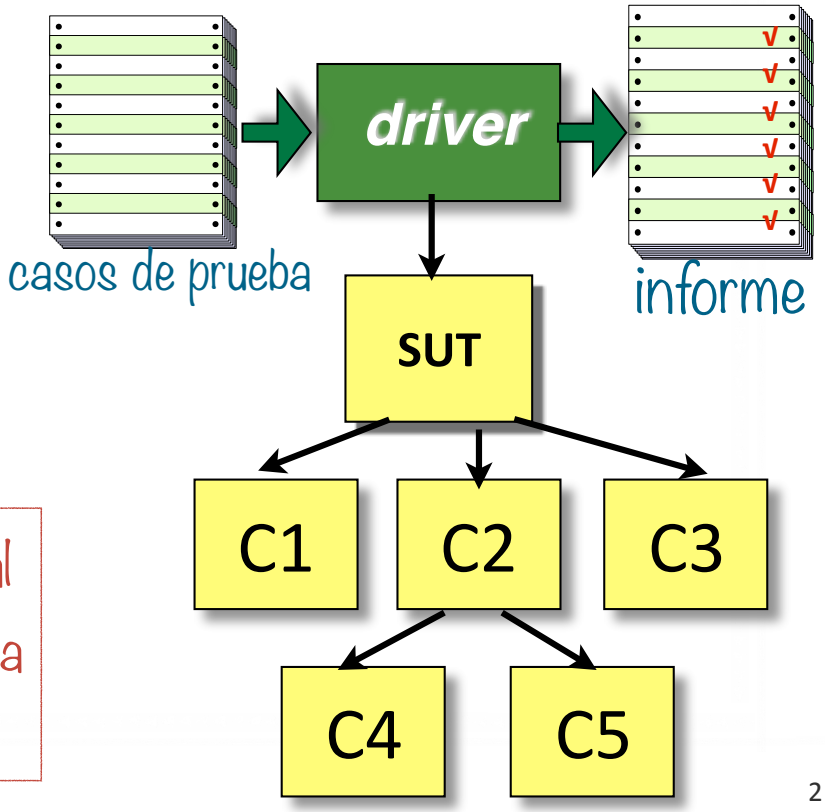
nivel de unidades | nivel de integración | nivel de sistema | nivel de aceptación

Objetivo: encontrar defectos en el código de las unidades

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

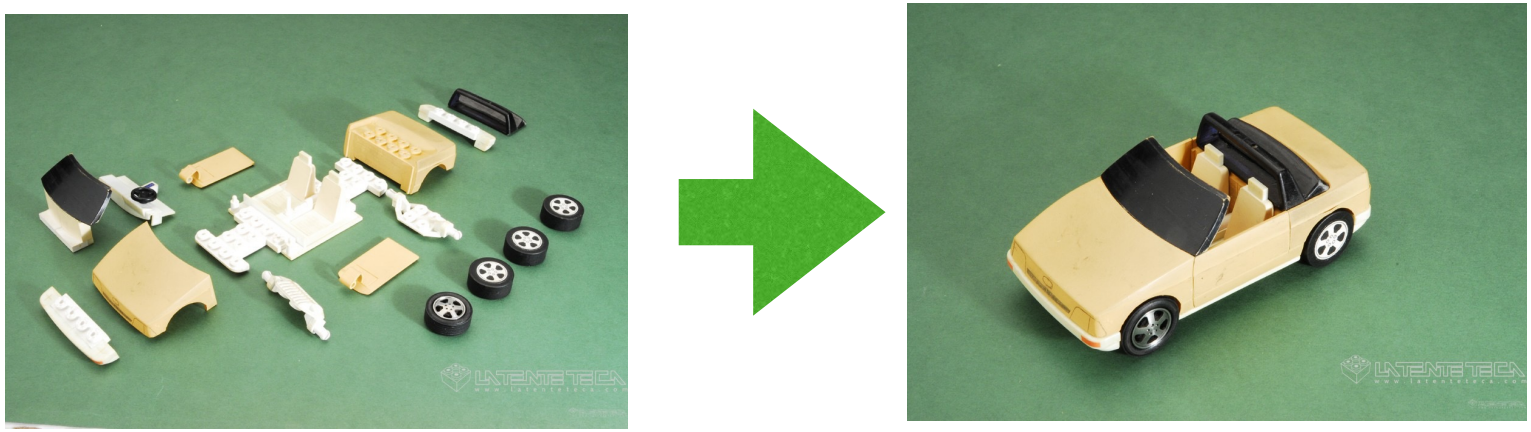
Objetivo: encontrar defectos derivados de la **interacción** entre las unidades, que previamente han sido probadas

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



Importancia de las pruebas de integración

- 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 proceso de integración

- 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 de 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

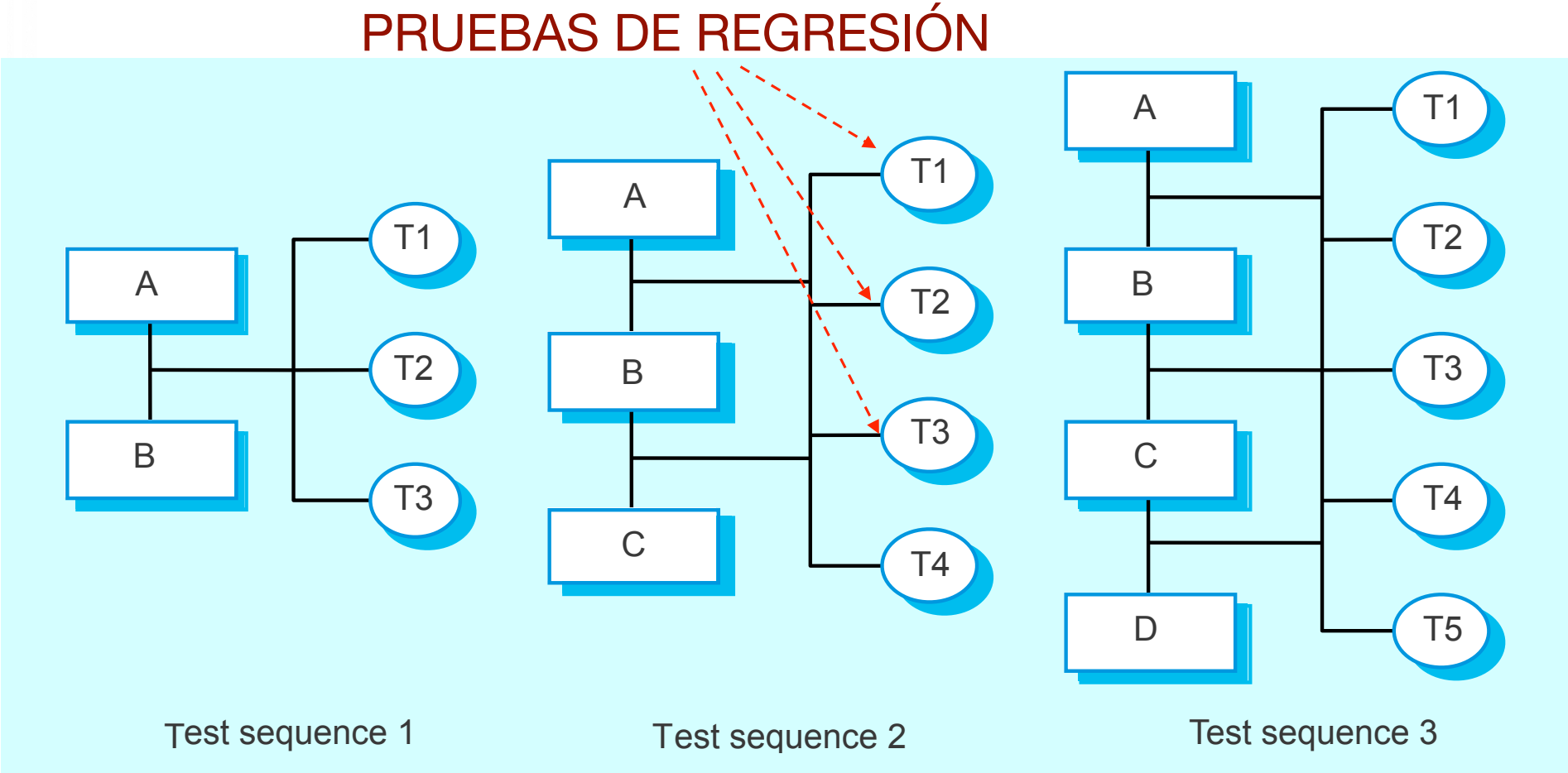
- Examinar el código a probar y listar de forma explícita cada llamada a un componente externo. Diseñar un conjunto de pruebas con los valores de los parámetros a componentes externos en los extremos de los rangos. Estos valores pueden revelar inconsistencias de la interfaz con una mayor probabilidad
- 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

Integración y pruebas de regresión

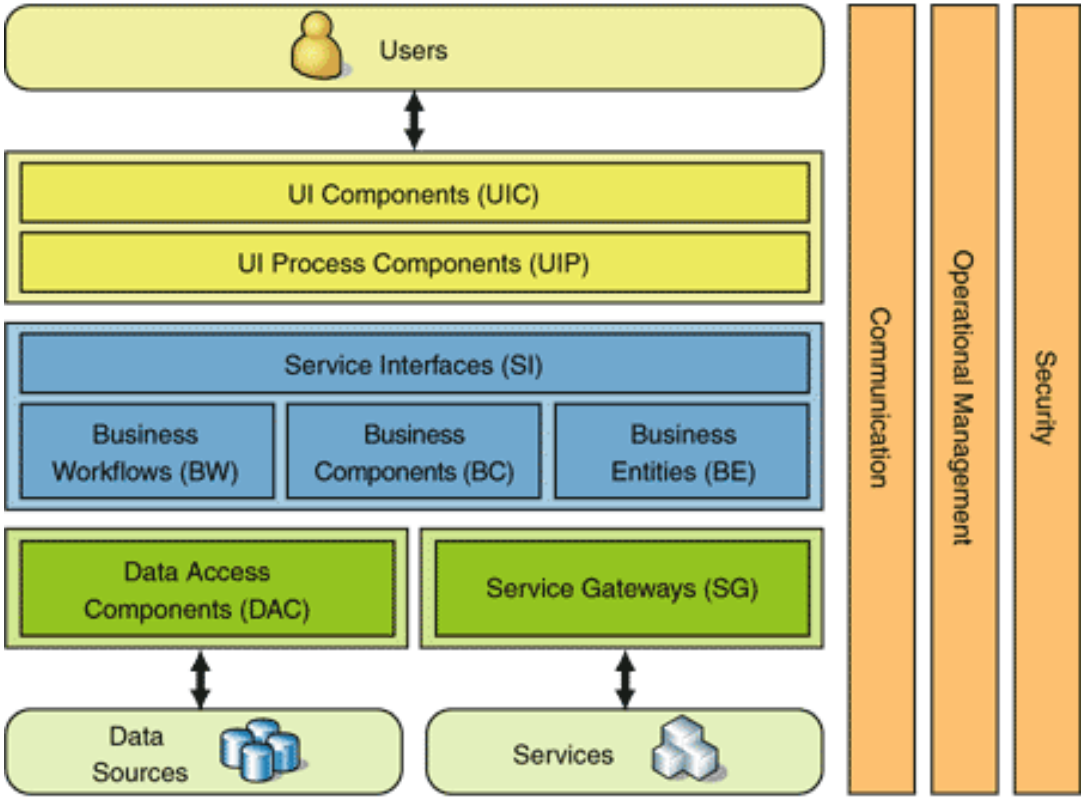
- Las pruebas de REGRESIÓN consisten en repetir las pruebas realizadas cuando integramos un nuevo componente



A,B,..,D= Componentes a probar; **Ti** = Tests

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



- 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?

- 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

Componentes principales DbUnit

- IDataBaseTester → JdbcDatabaseTester
- IDatabaseConnection
- DatabaseOperation
- IDataSet → FlatXmlDataSet
- ITable
- Assertion

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

Ejemplo de uso de IDataBaseTester

- 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`
    - Solamente realiza actualizaciones e inserciones basadas en el dataset
  - `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

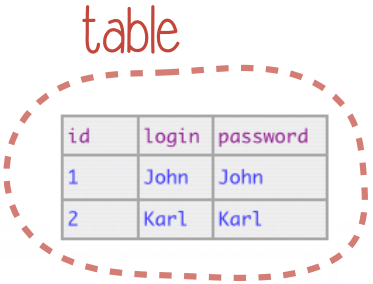
```
public class TestDBUnit {
   // databaseTester nos permitirá obtener la conexión con la BD
   private IDatabaseTester databaseTester;
   ...
   @Test
   public void testInsert() throws Exception {
       ...
       // recuperamos la conexión con la BD
       IDatabaseConnection connection = databaseTester.getConnection();

       // 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

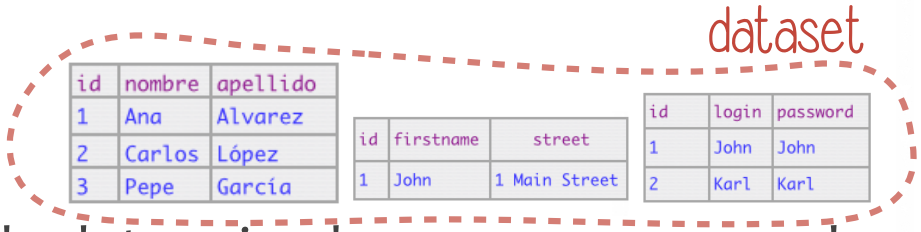
utilizamos la conexión para recuperar TODOS los datos de la BD

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



- 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

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);
}
```

DbUnit y Maven

- Para utilizar DbUnit en un proyecto Maven tendremos que incluir en el fichero de configuración (pom.xml) las siguientes dependencias (además de JUnit 4)

```
<dependency>
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>2.5.1</version>
    <scope>test</scope>
</dependency>
```

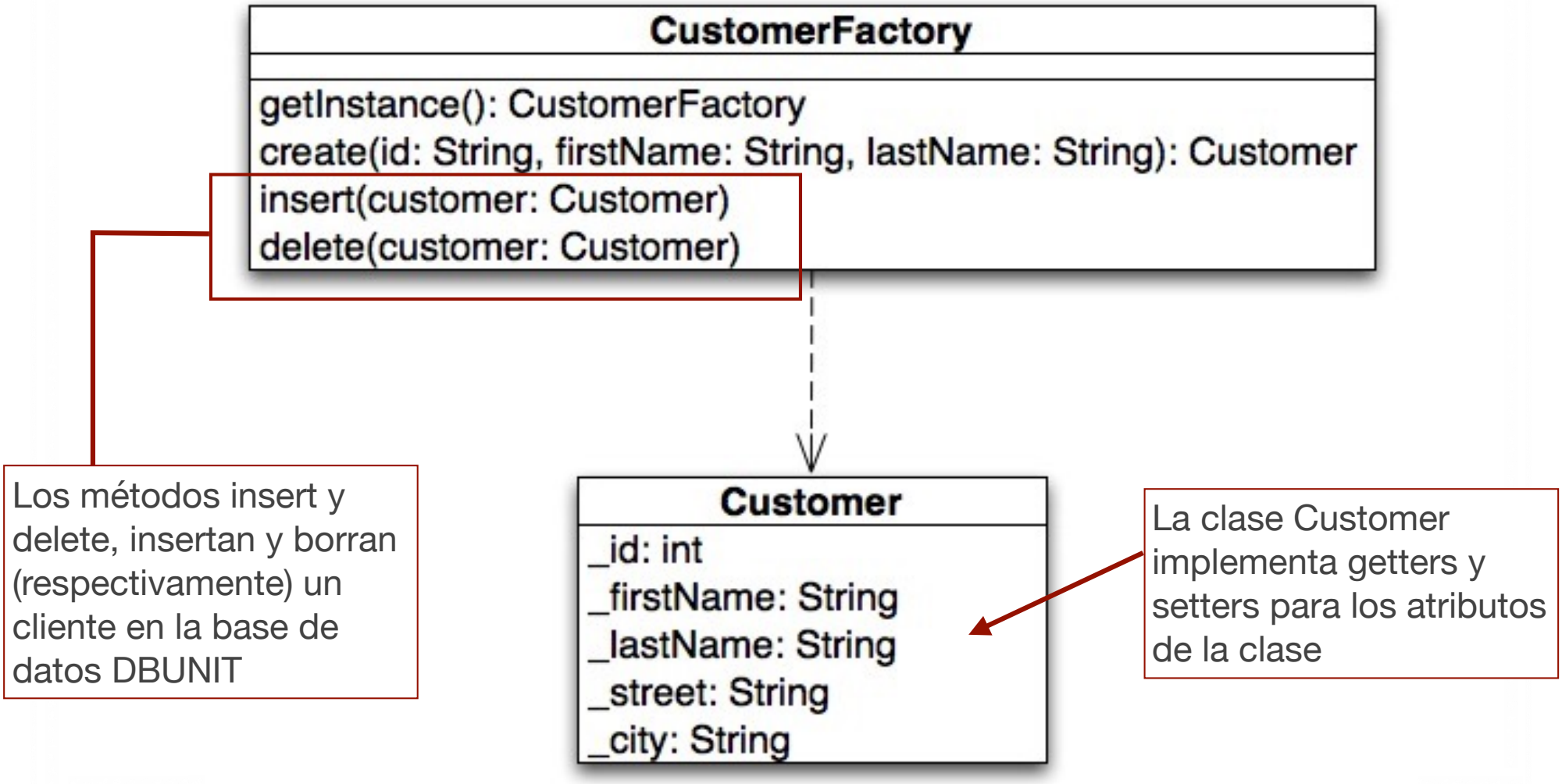
→ Librería DbUnit

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.38</version>
    <scope>test</scope>
</dependency>
```

→ Librería para acceder a una BD MySql

Ejemplo

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



Implementación de las pruebas

- 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

```
@Test
public void testInsert() throws Exception {
  Customer customer = _customerFactory.create(1,"John", "Smith");
  customer.setStreet("1 Main Street");
  customer.setCity("Anycity");

  //método a probar
  _customerFactory.insert(customer);

  // recuperamos la conexión con la BD
  IDatabaseConnection connection = databaseTester.getConnection();

  //obtenemos los valores reales de la tabla a partir de la conexión
  IDataSet databaseDataSet = connection.createDataSet();
  ITable actualTable = databaseDataSet.getTable("customer");

  //determinamos los valores esperados desde 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);

} //fin testInsert()
```

Datos de entrada del caso de prueba

Invocamos al método a probar (SUT)

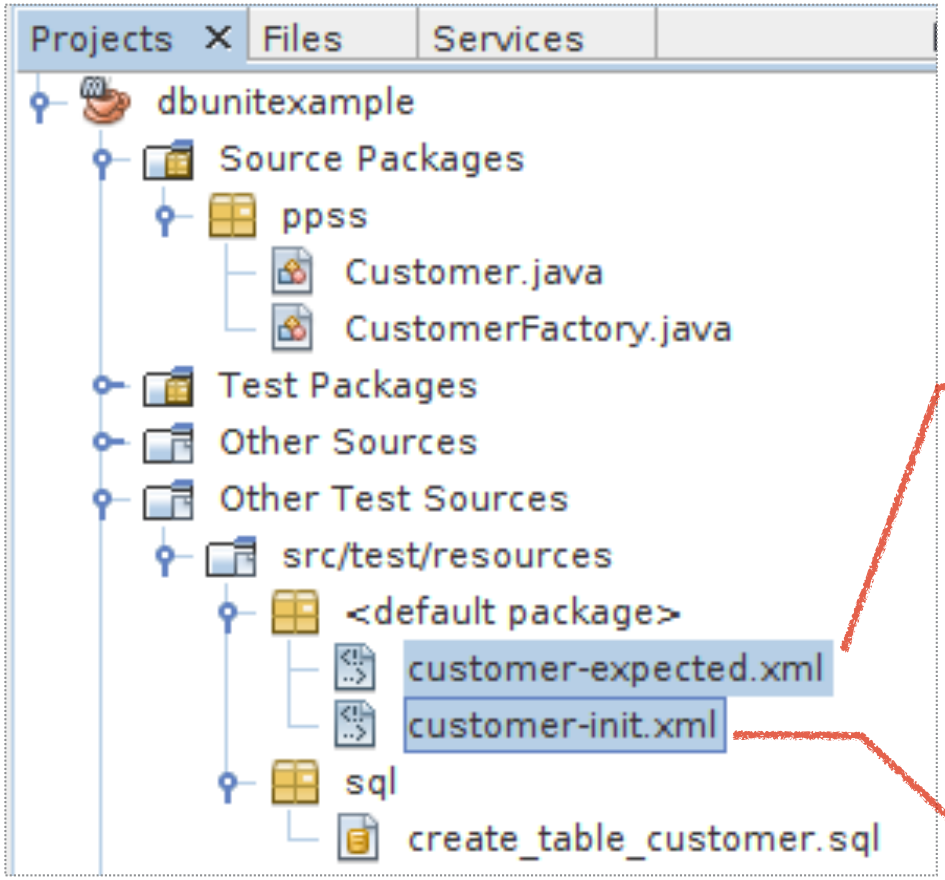
Resultado REAL

Resultado ESPERADO

Comparamos el resultado ESPERADO con el REAL

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



customer-expected.xml

Source | History

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <dataset>
3      <customer id="1"
4                firstname="John"
5                lastname="Smith"
6                street="1 Main Street"
7                city="Anycity" />
8  </dataset>
```

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

customer-init.xml

Source | History

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <dataset>
3
4      <customer />
5
6  </dataset>
7
```

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

Uso del plugin *sql-maven-plugin*

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>sql-maven-plugin</artifactId>
  <version>1.5</version>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.38</version>
    </dependency>
  </dependencies>
  <configuration>
    <driver>com.mysql.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost:3306</url>
    <username>root</username>
    <password></password>
  </configuration>
  <executions>
    <execution>
      <id>create-customer-table</id>
      <phase>pre-integration-test</phase>
      <goals>
          <goal>execute</goal>
      </goals>
      <configuration>
        <srcFiles>
          <srcFile>src/test/resources/sql/create_table_customer.sql</srcFile>
        </srcFiles>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Plugin para ejecutar sentencias SQL

Dependencia con el *driver* JDBC para acceder a una base de datos MySQL

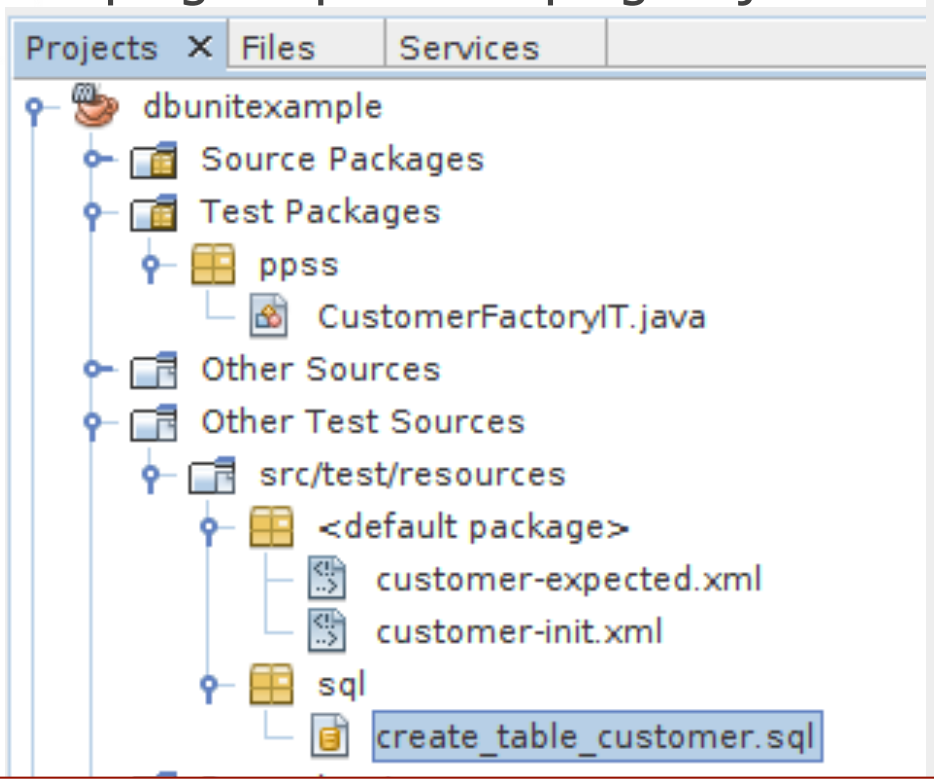
Configuración del *driver* JDBC

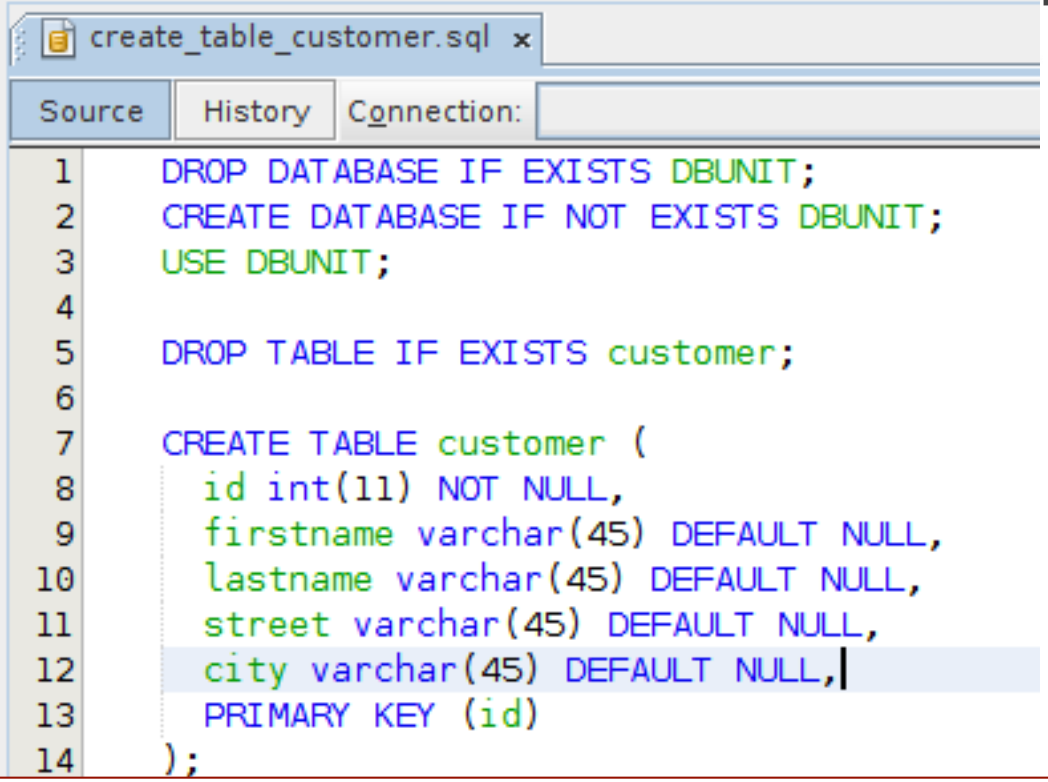
se ejecuta antes ejecutar los test de integración

script de inicialización de la tabla **customer**

Inicialización de la tabla customer

- El plugin sql-maven-plugin ejecutará el script sql **create_table_customer.sql**





Cuando ejecutemos **mvn verify**, para el proyecto DbUnitExample, se realizarán, en este orden, las siguientes fases del ciclo de vida:

- **compile**: se compila el código fuente (src/main)
- **test-compile**: se compilan los fuente de pruebas (src/test)
- **test**: se ejecutan los tests unitarios (de las clases TestXXX)
- **pre-integration-test**: se ejecuta el script create_table_customer
- **integration-test**: se ejecutan los tests de integración (de las clases ITxxx)
- **verify**: se detiene la construcción si algún test de integración ha fallado

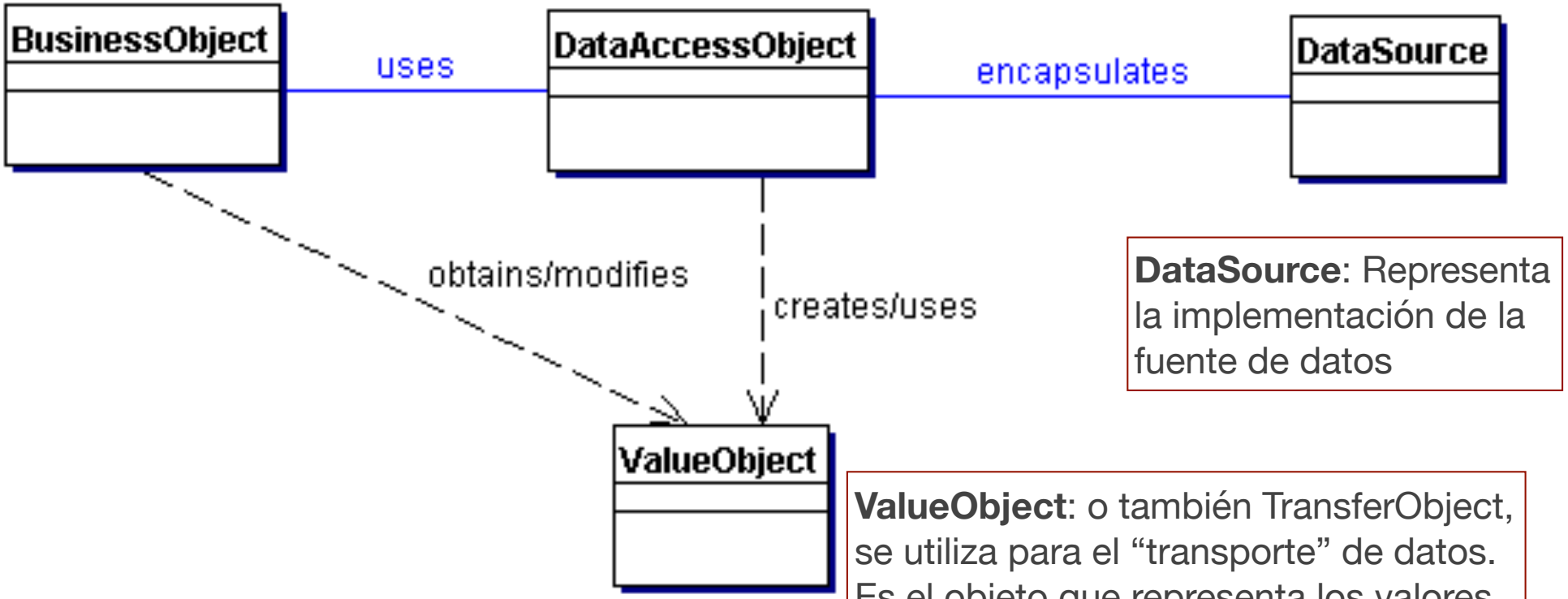
Acceso a datos: patrón DAO

- El patrón DAO (Data Access Object) se utiliza para abstraer el acceso a los datos de una BD
  - Generalmente, la aplicaciones de empresa harán uso de objetos DAO para acceder a la BD
- Los objetos DAO son usados por las clases de la capa de negocio para conseguir depender de forma indirecta de dicha BD
  - Los objetos de negocio (Business object, o también denominados Service Object) utilizan los objetos de acceso a datos para leer y escribir datos desde y en la BD
  - Los objetos DAO exponen una interfaz y ocultan completamente los detalles de implementación de la fuente de datos
  - Como la interfaz del DAO no cambia, cuando cambia la implementación de la fuente de datos subyacente, este patrón permite adaptarse a diferentes esquemas de almacenamiento sin que esto afecte a sus clientes o componentes de negocio

Consultar el siguiente enlace: http://www.programacion.com/articulo/catalogo_de_patrones_de_diseno_j2ee__y_ii:_capas_de_negocio_y_de_integracion_243/8

Diagrama de clases patrón DAO

- **BusinessObject**: Es el objeto que requiere acceso a la fuente de datos
- **DataAccessObject**: Abstrae la implementación del acceso a datos subyacente al BusinessObject para permitirle un acceso transparente a la base de datos



DataSource: Representa la implementación de la fuente de datos

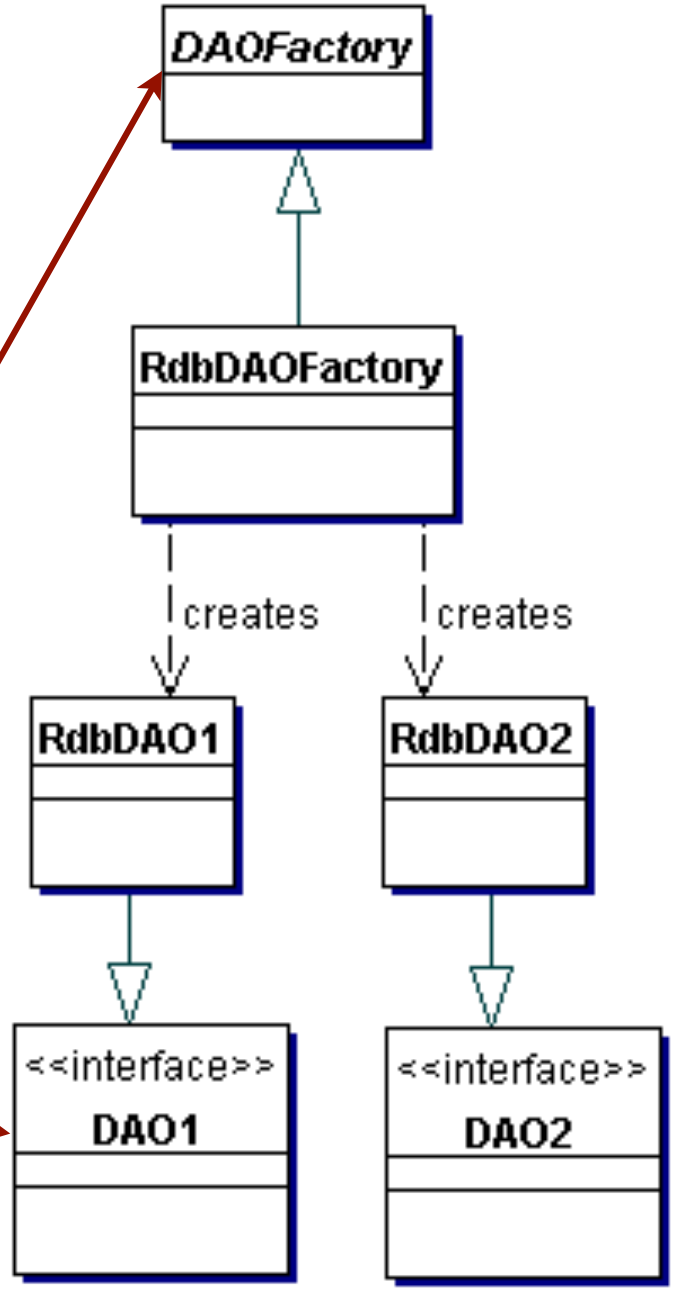
ValueObject: o también TransferObject, se utiliza para el "transporte" de datos. Es el objeto que representa los valores obtenidos (o a insertar) en la base de datos

Flexibilización del patrón DAO

- Introducción del patrón **Factory Method**, para "producir" el número de DAOs que necesita la aplicación
  - Se utiliza cuando el almacenamiento subyacente no está sujeto a cambios de una implementación a otra

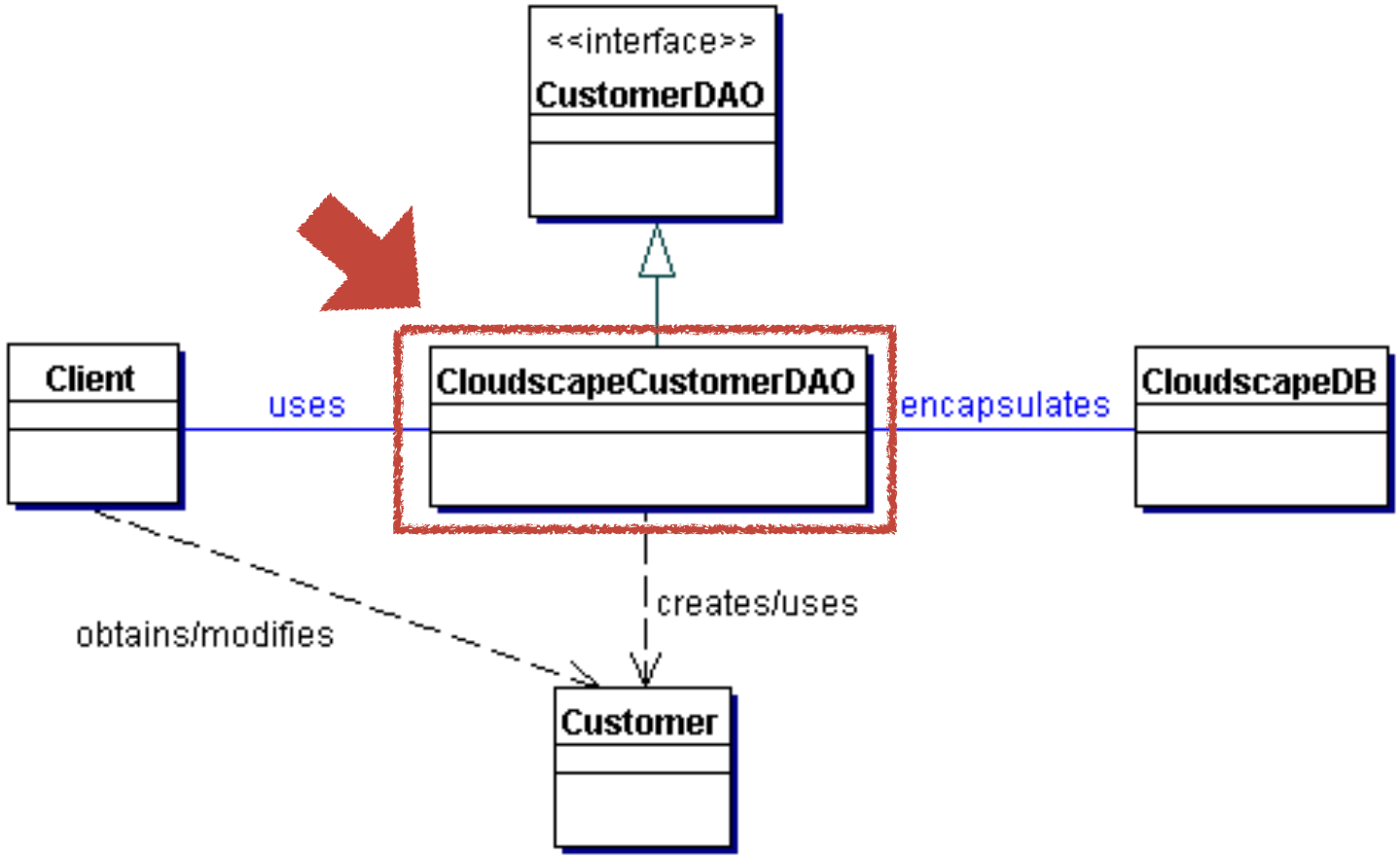
Se puede invocar una implementación específica a través de una Factory

Todos los DAO implementan interfaces CRUD: Create, Retrieve, Update, Delete



Ejemplo de diagrama de clases de un DAO

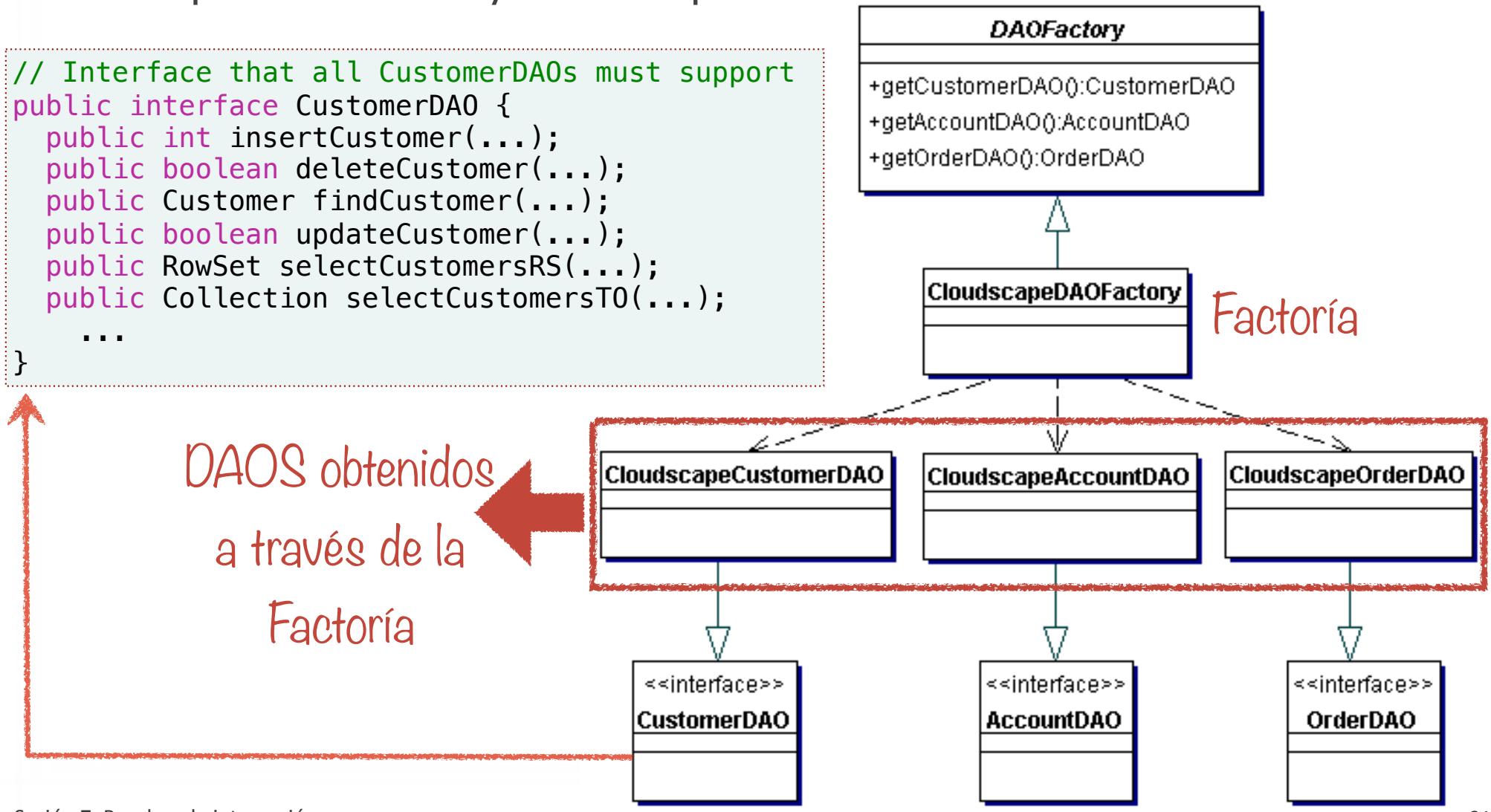
- Los objetos de tipo CloudscapeCustomerDAO se encargan de acceder a la información de un cliente (clase Customer) en una base de datos Cloudscape
- Diagrama de clases de la implementación del patrón DAO



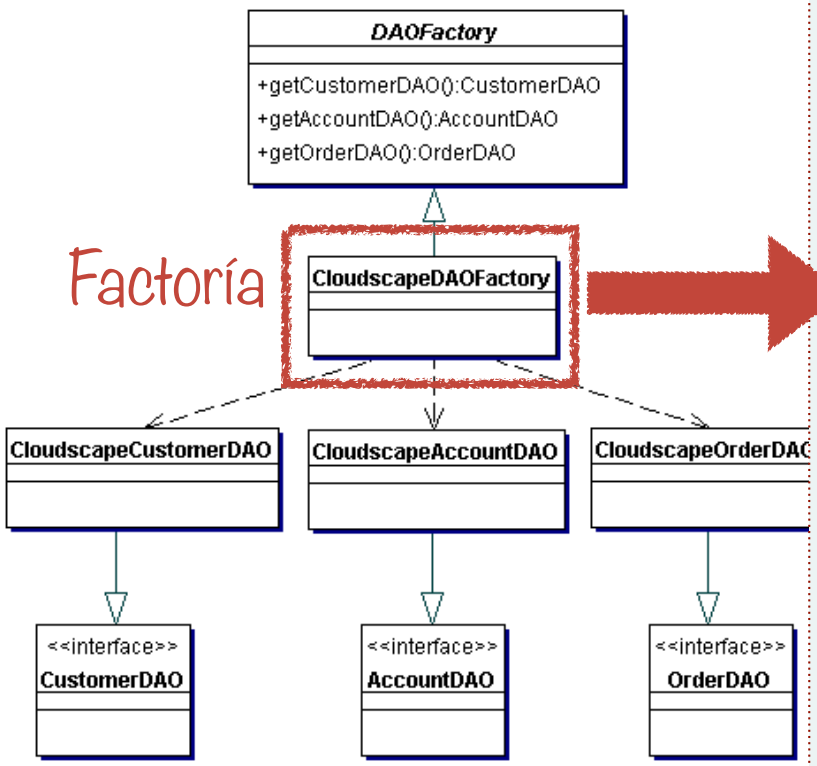
Ejemplo de una Factoría de DAOs

- Una factoría de DAOs produce varios DAOs para una única implementación de una base de datos. Los DAOS CloudscapeCustomerDAO, CloudscapeAccountDAO y CloudscapeOrderDAO acceden a la BD

```
// Interface that all CustomerDAOs must support
public interface CustomerDAO {
  public int insertCustomer(...);
  public boolean deleteCustomer(...);
  public Customer findCustomer(...);
  public boolean updateCustomer(...);
  public RowSet selectCustomersRS(...);
  public Collection selectCustomersTO(...);
    ...
}
```



Implementación concreta (*CloudscapeDAOFactory*)




```
// Cloudscape concrete DAO Factory implementation
import java.sql.*;

public class CloudscapeDAOFactory extends DAOFactory {
   public static final String DRIVER=
    "COM.cloudscape.core.RmiJdbcDriver";
   public static final String DBURL=
    "jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";

   // method to create Cloudscape connections
   public static Connection createConnection() {
    // Use DRIVER and DBURL to create a connection
    // Recommend connection pool implementation/usage
   }
   public CustomerDAO getCustomerDAO() {
    // CloudscapeCustomerDAO implements CustomerDAO
    return new CloudscapeCustomerDAO();
   }
   public AccountDAO getAccountDAO() {
    // CloudscapeAccountDAO implements AccountDAO
    return new CloudscapeAccountDAO();
   }
   public OrderDAO getOrderDAO() {
    // CloudscapeOrderDAO implements OrderDAO
    return new CloudscapeOrderDAO();
   }
   ...
}
```

Implementación concreta de un *DAO*

```
// CloudscapeCustomerDAO implementation of the CustomerDAO interface. This class can contain all Cloudscape
// specific code and SQL statements. The client is thus shielded from knowing these implementation details.

import java.sql.*;

public class CloudscapeCustomerDAO implements CustomerDAO {
   public CloudscapeCustomerDAO() {
    // initialization
   }
   //The following methods can use CloudscapeDAOFactory.createConnection()
   //to get a connection as required
   public int insertCustomer(...) {
    // Implement insert customer here. Return newly created customer
        number or a -1 on error
   }
   public boolean deleteCustomer(...) {
    // Implement delete customer here. Return true on success, false
        on failure
   }
   public Customer findCustomer(...) {
    // Implement find a customer here using supplied argument values as search criteria
    // Return a Transfer Object if found, return null on error or if not found
   }
   public boolean updateCustomer(...) {
    // implement update record here using data from the customerData Transfer Object
    // Return true on success, false on failure or  error
   }
   public RowSet selectCustomersRS(...) {
    // implement search customers here using the  supplied criteria. Return a RowSet.
   }
   public Collection selectCustomersTO(...) {
    // implement search customers here using the supplied criteria.
    // Alternatively, implement to return a Collection of Transfer Objects.
   }
   ...
}
```

Ejemplo de código cliente

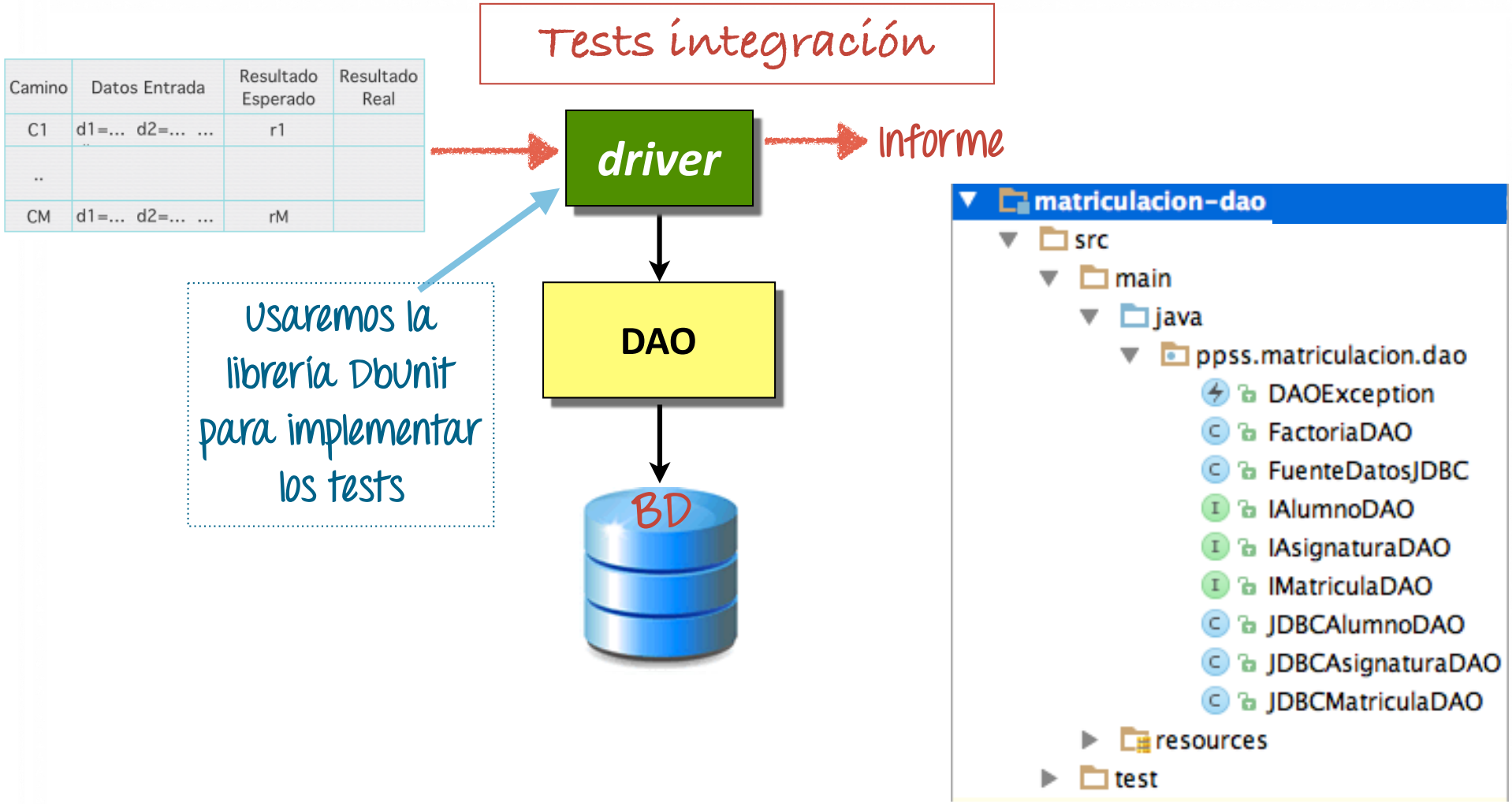
```
...
// create the required DAO Factory
DAOFactory cloudscapeFactory =
DAOFactory.getDAOFactory(DAOFactory.DAOCLOUDSCAPE);

// Create a DAO
CustomerDAO custDAO =
cloudscapeFactory.getCustomerDAO();

// create a new customer
int newCustNo = custDAO.insertCustomer(...);

// Find a customer object. Get the Transfer Object.
Customer cust = custDAO.findCustomer(...);

// modify the values in the Transfer Object.
cust.setAddress(...);
cust.setEmail(...);
// update the customer object using the DAO
custDAO.updateCustomer(cust);

// delete a customer object
custDAO.deleteCustomer(...);
// select all customers in the same city
Customer criteria=new Customer();
criteria.setCity("New York");
Collection customersList =
custDAO.selectCustomersTO(criteria);
// returns customersList - collection of Customer
// Transfer Objects. iterate through this collection to get values.

...
```

Los "clientes" de los objetos DAO son los objetos de la capa de negocio de la aplicación

el DAO devuelve objetos Customer, que son utilizados por la capa de negocio

Y ahora vamos al laboratorio…

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



Referencias bibliográficas

- 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
- Core J2EE Patterns - Data Access Object
  - http://www.oracle.com/technetwork/java/dataaccessobject-138824.html