Object-relational mapping

Source http://www.datanucleus.org/

Object-relational impedance mismatch

- The **object-relational impedance mismatch** is a set of conceptual and technical difficulties that are often encountered when a RDBMS is being used by a program written in an object-oriented programming language or style.

Data type differences

- A major mismatch between existing relational and OO languages is the type system differences.
- The relational model strictly prohibits by-reference attributes (or pointers), whereas OO languages embrace and expect by-reference behavior.
- Scalar types and their operator semantics are also very often subtly to vastly different between the models, causing problems in mapping.
- For example, most SQL systems support string types with varying collations and constrained maximum lengths, while most OO languages consider collation only as an argument to sort routines and strings are intrinsically sized to available memory

Structural and integrity differences

- In OO languages, objects can be composed of other objects—often to a high degree—or specialized from a more general definition. This may make the mapping to relational schemas less straightforward.
- This is because relational data tends to be represented in a named set of global, unnested relation variables.

Manipulative differences

- The relational model has an intrinsic, relatively small and well defined set of primitive operators for usage in the query and manipulation of data, whereas OO languages generally handle query and manipulation through custom-built or lower-level physical access path specific imperative operations

Transactional differences

- Relational database transactions, as the smallest unit of work performed by databases, are much larger than any operations performed by classes in OO languages.
- Transactions in relational databases are dynamically bounded sets of arbitrary data manipulations, whereas the granularity of transactions in OO languages is typically individual assignments of primitive typed fields.
- OO languages typically have no analogue of isolation or durability as well and atomicity and consistency are only ensured for said writes of primitive typed fields.

Solving impedance mismatch

- There have been some attempts at building object-oriented database management systems (OODBMS) that would avoid the impedance mismatch problem.
- They have been less successful in practice than relational databases however, partly due to the limitations of OO principles as a basis for a data model.
- There has been research performed in extending the database-like capabilities of OO languages through such notions as transactional memory.

Object-relational mapping

- **Object-relational mapping** (ORM, O/RM, and O/R mapping) is a programming technique for solving the impedence mismatch
- The heart of the problem is translating the logical representation of the objects into an atomized form that is capable of being stored on the database, while somehow preserving the properties of the objects and their relationships so that they can be reloaded as an object when needed.
- If this storage and retrieval functionality is implemented, the objects are then said to be **persistent**.

Object-relational mapping

- Compared to traditional techniques of exchange between an object-oriented language and a relational database, ORM often reduces the amount of code that needs to be written
- Disadvantages of O/R mapping tools generally stem from the high level of abstraction obscuring what is actually happening in the implementation code.

Java Data Objects

- Java Data Objects (JDO) is a specification of Java object persistence.
- One of its features is a transparency of the persistence services to the domain model.
- JDO persistent objects are ordinary Java programming language classes (POJOs); there is no requirement for them to implement certain interfaces or extend from special classes.
- JDO was developed under the Java Community Process.
- JDO 3.0 was released in April 2010

Java Data Objects

- Object persistence is defined in external XML metafiles
- JDO vendors provide developers with enhancers, which modify compiled Java class files so they can be transparently persisted.
- Byte-code enhancement is not mandated by the JDO specification, although it is the commonly used mechanism for implementing the JDO specification's requirements.
- Currently, JDO vendors offer several options for persistence, e.g. to RDBMS, to OODB, or to files.

Java Data Objects

- JDO enhanced classes are portable across different vendors' implementation. Once enhanced, a Java class can be used with any vendor's JDO product
- JDO is both an object-relational mapping standard and a transparent object persistence standard
- JDO, from an API point of view, is agnostic to the technology of the underlying datastore

Java Persistence API

- The Java Persistence API, sometimes referred to as JPA, is a Java programming language application programming interface specification which describes the management of relational data in applications
- The JPA 2.1 specification was developed under the Java Community Process and was released 22 April 2013
- JPA uses the javax.persistence package

Java Persistence API

- JPA, is an object-relational mapping (ORM) standard, while JDO is both an object-relational mapping standard and a transparent object persistence standard.
- JDO, from an API point of view, is agnostic to the technology of the underlying datastore, whereas JPA is targeted to RDBMS datastores (although there are several JPA providers that support access to non-relational datastores through the JPA API, such as DataNucleus and ObjectDB).

Java Persistence API

- A persistence entity is a lightweight Java class whose state is typically persisted to a table in a relational database.
- Instances of such an entity correspond to individual rows in the table.
- Entities typically have relationships with other entities, and these relationships are expressed through object/relational metadata.
- Object/relational metadata can be specified directly in the entity class file by using annotations, or in a separate XML descriptor file distributed with the application.

Java Persistence Query Language

- The Java Persistence Query Language (JPQL) makes queries against entities stored in a relational database. Queries resemble SQL queries in syntax, but operate against entity objects rather than directly with database tables.

Related technologies

- **Hibernate** provides an open source object-relational mapping framework for Java.
- Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions.
- Hibernate's primary feature is mapping from Java classes to database tables (and from Java data types to SQL data types).
- Many of the features originally presented in Hibernate were incorporated into the Java Persistence API
- Versions 3.2 and later provide an implementation for the Java Persistence API.

Related technologies

- **Enterprise JavaBeans**
- The EJB 3.0 specification (itself part of the Java EE 5 platform) included a definition of the Java Persistence API.
- However, javax.persistence does **not** require an EJB container, and thus will work within a Java SE environment as well
- The Java Persistence API replaces the persistence solution of EJB 2.0 CMP (Container Managed Persistence).

JDO with DataNucleus AccessPlatform

- http://www.datanucleus.org/products/accessplatform/jdo/tutorial.html
- Step 0 : Download DataNucleus AccessPlatform
- In our case, download JDO tutorial from Classroom or from the Esercitazioni section of the course home page
- Step 1 : Create your domain/model classes
- Do this as you would normally.

Tutorial in Eclipse

- Uncompress jdo.zip into a new folder jdo
- Create a new Java project
- Copy the src and lib subfolders of jdo in the root folder of the Java project

Tutorial in Eclipse

- Add all the libraries in the lib folder to the build path of the project
  - lib/datanucleus-core-5.1.7.jar;
  - lib/datanucleus-api-jdo-5.1.4.jar;
  - lib/datanucleus-rdbms-5.1.7.jar;
  - lib/javax.jdo-3.2.0-m8.jar JDO API JAR
  - lib/db2jcc4.jar; JDBC driver classes

Tutorial in Eclipse

- Copy jdo/src/META-INF/persistence.xml in bin/META-INF/persistence.xml in the Java project
- Copy jdo/src/package-db2.orm in bin/package-db2.orm

Working example: store

- Application handling products in a store

```
package org.datanucleus.samples.jdo.tutorial;
public class Product
{   protected long id;  String name = null;
    String description = null;   double price = 0.0;

    public Product(String name, String desc, double price)
    {   this.name = name;
        this.description = desc;
        this.price = price;}
……
}
```

Book

```
package org.datanucleus.samples.jdo.tutorial;
public class Book extends Product
{   String author=null;
    String isbn=null;
    String publisher=null;
    public Book(String name, String desc, double price, String
author, String isbn, String publisher)
    {   super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
….
}
```

Inheritance

- We have inheritance between 2 classes.
- Some data in the store will be of type *Product*, and some will be *Book*
- This allows us to extend our store further in the future and provide *DVD* items for example, and so on.
- JDO allows objects to be retrieved maintaining their inheritances.

Inventory

```
package org.datanucleus.samples.jdo.tutorial;
import java.util.HashSet;
import java.util.Set;
public class Inventory
{   protected String name=null;
    protected Set<Product> products = new HashSet<Product>();
    public Inventory(String name)  {       this.name = name;}
    public String getName()    {       return name;    }
    public Set<Product> getProducts() {       return products;    }
    public String toString()
    {       return "Inventory : " + name;}
}
```

Step 2 : Define the Persistence for classes

- You now need to define how the classes should be persisted, in terms of which fields are persisted etc. With JDO you could use
- XML Metadata
- Annotations
- Annotations + XML
- MetaData API at runtime
- Here we use what could be considered a best practice, specifying basic persistence info as annotations, and then adding ORM information in XML (since if we want then to persist to a different datastore later we don't need to update/recompile our classes, just change the XML file).

Product

```
package org.datanucleus.samples.jdo.tutorial;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;
@PersistenceCapable public class Product
{   @PrimaryKey
    @Persistent(valueStrategy=IdGeneratorStrategy.NATIVE)
    protected long id;
    String name = null;
    String description = null;   double price = 0.0;
……
}
```

Book

```
package org.datanucleus.samples.jdo.tutorial;
import javax.jdo.annotations.PersistenceCapable;
@PersistenceCapable
public class Book extends Product
{   String author=null;
    String isbn=null;
    String publisher=null;
    public Book(String name, String desc, double price, String
author, String isbn, String publisher)
    {   super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;}
….}
```

Inventory

```
package org.datanucleus.samples.jdo.tutorial;
import java.util.HashSet;
import java.util.Set;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.PrimaryKey;
@PersistenceCapable public class Inventory
{   @PrimaryKey protected String name=null;
    protected Set<Product> products = new HashSet<Product>();
    public Inventory(String name)  {       this.name = name;}
    public String getName()    {       return name;    }
    public Set<Product> getProducts() {       return products;    }
    public String toString()
    {       return "Inventory : " + name;}
}
```

Persistence information

- Note that we mark each class that can be persisted with *@PersistenceCapable* and their primary key field(s) with @PrimaryKey.
- In addition we defined a *valueStrategy* for Product field *id* so that it will have its values generated automatically.
- You now need to define which objects of these classes are actually persisted. You do this via a file *META-INF/persistence.xml* at the root of the CLASSPATH.

Persistence information

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
version="1.0">
<persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jdo.tutorial.Product</class>
        <class>org.datanucleus.samples.jdo.tutorial.Book</class>
        <exclude-unlisted-classes/>
```

persistence.xml

```
 <properties>
        <property name="javax.jdo.option.ConnectionURL"
value="jdbc:db2://server-rzf.mate.man:50000/test"/>
        <property name="javax.jdo.option.ConnectionDriverName"
value="com.ibm.db2.jcc.DB2Driver"/>
        <property name="javax.jdo.option.ConnectionUserName"
value="tbd"/>
        <property name="javax.jdo.option.ConnectionPassword"
value="TechBD"/>
        <property name="javax.jdo.option.Mapping" value="db2"/>
   </properties>
 </persistence-unit>
</persistence>
```

ORM information

- We define ORM information in an XML file package-db2.orm

```
<?xml version="1.0"?>
<!DOCTYPE orm SYSTEM "file:/javax/jdo/orm.dtd">
<orm>
   <package name="org.datanucleus.samples.jdo.tutorial">
      <class name="Inventory" table="INVENTORIES<MATR>">
         <field name="name">
            <column name="INVENTORY_NAME" length="100"/>
         </field>
         <field name="products" table=
"INVENTORY_PRODUCTS<MATR>">
            <join/>
         </field>
      </class>
```

ORM information

```
<class name="Product" table="PRODUCTS<MATR>">
        <inheritance strategy="new-table"/>
        <field name="id"><column name="PRODUCT_ID"/></field>
        <field name="name"><column name="PRODUCT_NAME"
length="100"/></field>
    </class>
    <class name="Book" table="BOOKS<MATR>">
        <inheritance strategy="new-table"/>
        <field name="author"><column length="40"/></field>
        <field name="isbn"><column length="20" jdbc-type="CHAR"/>
        </field>
        <field name="publisher"><column length="40"/></field>
    </class>
  </package>
</orm>
```

Step 3 : Enhance your classes

- JDO relies on the classes that you want to persist being *PersistenceCapable*. That is, they need to implement this Java interface.
- You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them *PersistenceCapable* .
- There are several ways to do this, using an "enhancer" at compile time (with JDK1.6+), or at runtime, or as a post-compile step. We use the post-compile step in this tutorial.

Step 3 : Enhance your classes

- **DataNucleus JDO** provides its own byte-code enhancer for instrumenting/enhancing your classes for use by any JDO implementation.
- Use the command line from the root folder of the project
- Linux: java -cp bin:lib/datanucleus-core-5.1.7.jar:lib/datanucleus-api-jdo-5.1.4.jar:lib/datanucleus-rdbms-5.1.7.jar:lib/javax.jdo-3.2.0-m8.jar org.datanucleus.enhancer.DataNucleusEnhancer bin/package-db2.orm
- Assuming bytecode is in folder bin, libraries in folder lib and package-db2.orm in folder bin

Step 3 : Enhance your classes

- Windows cmd: java -cp bin;lib\datanucleus-core-5.1.7.jar;lib\datanucleus-api-jdo-5.1.4.jar;lib\datanucleus-rdbms-5.1.7.jar;lib\javax.jdo-3.2.0-m8.jar org.datanucleus.enhancer.DataNucleusEnhancer bin\package-db2.orm
- Windows PowerShell: java -cp "bin;lib\datanucleus-core-5.1.7.jar;lib\datanucleus-api-jdo-5.1.4.jar;lib\datanucleus-rdbms-5.1.7.jar;lib\javax.jdo-3.2.0-m8.jar" org.datanucleus.enhancer.DataNucleusEnhancer bin\package-db2.orm

Step 3 : Enhance your classes

- This command enhances the .class files that have @PersistenceCapable annotations.
- If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a *ClassNotPersistenceCapableException* thrown.
- The output of this step are a set of class files that represent *PersistenceCapable* classes.

Step 4 : Generate any schema required for your domain classes

- This step is optional, depending on whether you have an existing database schema.
- If you haven't, at this point you can use the RDBMS SchemaTool to generate the tables where these domain objects will be persisted.
- DataNucleus RDBMS SchemaTool is a command line utility

Step 4 : Generate any schema required for your domain classes

- Linux: java -cp bin:lib/datanucleus-core-5.1.7.jar:lib/datanucleus-api-jdo-5.1.4.jar:lib/datanucleus-rdbms-5.1.7.jar:lib/javax.jdo-3.2.0-m8.jar:lib/db2jcc4.jar org.datanucleus.store.schema.SchemaTool -create -pu Tutorial bin/package-db2.orm
- -pu Tutorial has the effect of indicating that the pesistence information are specified in the Tutorial persistence-unit in persistence.xml

Step 4 : Generate any schema required for your domain classes

- Windows PowerShell: java -cp "bin;lib\datanucleus-core-5.1.7.jar;lib\datanucleus-api-jdo-5.1.4.jar;lib\datanucleus-rdbms-5.1.7.jar;lib\javax.jdo-3.2.0-m8.jar;lib\db2jcc4.jar" org.datanucleus.store.schema.SchemaTool -create -pu Tutorial bin\package-db2.orm

- This will generate the required tables, indexes, and foreign keys for the classes defined in the JDO Meta-Data file.

Step 5 : Write the code to persist objects of your classes

- Now you need to define which objects of the classes are actually persisted, and when. Interaction with the persistence framework of JDO is performed via a PersistenceManager.
- This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc..

Main

```
import java.util.Iterator;
import java.util.List;

import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Extent;
import javax.jdo.Query;
import javax.jdo.JDOHelper;
import javax.jdo.Transaction;
```

Main

- The initial step is to obtain access to a PersistenceManager

```
 PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory("Tutorial");
 PersistenceManager pm =
pmf.getPersistenceManager();
```

- We are creating a PersistenceManagerFactory using the file *persistence.xml* as used above for DataNucleus RDBMS SchemaTool. This will contain all properties necessary for persistence usage.

Main

- Now that the application has a PersistenceManager it can persist objects. This is performed as follows

```
Transaction tx=pm.currentTransaction();
    Object inventoryId = null;
    try  {
        tx.begin();
        Inventory inv = new Inventory("My Inventory");
        Product product = new Product("Sony Discman","A standard discman from Sony",200.00);
        Book book = new Book("Lord of the Rings by Tolkien","The classic story",49.99,"JRR Tolkien", "12345678", "MyBooks Factory");
        inv.getProducts().add(product);
        inv.getProducts().add(book);
        pm.makePersistent(inv);
        tx.commit(); inventoryId = pm.getObjectId(inv);
}
```

Main

```
finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
```

- Please note that the *finally* step is important in that it tidies up connections to the datastore and the PersistenceManager.

Main

- To retrieve objects from persistent storage:

```
// Basic Extent of all Products
pm = pmf.getPersistenceManager();
tx = pm.currentTransaction();
try
{   tx.begin();
    Extent e = pm.getExtent(Product.class, true);
    Iterator iter = e.iterator();
    while (iter.hasNext())
    {   Object obj = iter.next();
        System.out.println(">  " + obj);  }
    tx.commit();
}
```

Main

```
 catch (Exception e)
        {
            System.out.println("Exception thrown during retrieval of
Extent : " + e.getMessage());
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
```

Main

```
 // Perform some query operations
        pm = pmf.getPersistenceManager();
        tx = pm.currentTransaction();
        try {  tx.begin();
            System.out.println("Executing Query for Products with price below
150.00");
            Query q=pm.newQuery("SELECT FROM " +
Product.class.getName() +
                " WHERE price < 150.00 ORDER BY price ASC");
            List<Product> products = (List<Product>)q.execute();
            Iterator<Product> iter = products.iterator();
```

Main

```
while (iter.hasNext())
        {
            Product p = iter.next();
            System.out.println(">  " + p);

            // Give an example of an update
            if (p instanceof Book)
            {
                Book b = (Book)p;
                b.setDescription("This book has been reduced in price!");
            }
        }

        tx.commit();
    }
```

Main

```
finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
        System.out.println("");
```

Main

```
// Clean out the database
      pm = pmf.getPersistenceManager();
      tx = pm.currentTransaction();
      try
      {
         tx.begin();

         System.out.println("Retrieving Inventory using its id");
         Inventory inv = (Inventory)pm.getObjectById(inventoryId);

         System.out.println("Clearing out Inventory");
         inv.getProducts().clear();
```

Main

```
 System.out.println("Deleting Inventory");
        pm.deletePersistent(inv);

        System.out.println("Deleting all products from persistence");
        Query q = pm.newQuery(Product.class);
        long numberInstancesDeleted = q.deletePersistentAll();
        System.out.println("Deleted " + numberInstancesDeleted + "
products");

        tx.commit();
    }
```

Main

```
finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }

        System.out.println("");
        System.out.println("End of Tutorial");
    }
}
```

Step 6 : Run your application

- Linux: java -cp bin:lib/datanucleus-core-5.1.7.jar:lib/datanucleus-api-jdo-5.1.4.jar:lib/datanucleus-rdbms-5.1.7.jar:lib/javax.jdo-3.2.0-m8.jar:lib/db2jcc4.jar org.datanucleus.samples.jdo.tutorial.Main
- Windows PowerShell: java -cp "bin;lib\datanucleus-core-5.1.7.jar;lib\datanucleus-api-jdo-5.1.4.jar;lib\datanucleus-rdbms-5.1.7.jar;lib\javax.jdo-3.2.0-m8.jar;lib\db2jcc4.jar" org.datanucleus.samples.jdo.tutorial.Main
- Run Main by right clicking on the Main.java file and selection Run As..->Java application

Output

```
DataNucleus AccessPlatform with JDO
====================================
Persisting Inventory of products
Inventory, Product and Book have been persisted
Retrieving Extent for Products
>  Book : JRR Tolkien - Lord of the Rings by Tolkien
>  Product : 2 name=Sony Discman [A standard discman from Sony]
Executing Query for Products with price below 150.00
>  Book : JRR Tolkien - Lord of the Rings by Tolkien
Retrieving Inventory using its id
Clearing out Inventory
Deleting Inventory
Deleting all products from persistence
Deleted 2 products
End of Tutorial
```

Delete the schema

- Linux: java -cp bin:lib/datanucleus-core-5.1.7.jar:lib/datanucleus-api-jdo-5.1.4.jar:lib/datanucleus-rdbms-5.1.7.jar:lib/javax.jdo-3.2.0-m8.jar:lib/db2jcc4.jar org.datanucleus.store.schema.SchemaTool -delete -pu Tutorial bin/package-db2.orm
- Windows PowerShell: java -cp "bin;lib\datanucleus-core-5.1.7.jar;lib\datanucleus-api-jdo-5.1.4.jar;lib\datanucleus-rdbms-5.1.7.jar;lib\javax.jdo-3.2.0-m8.jar;lib\db2jcc4.jar" org.datanucleus.store.schema.SchemaTool -delete -pu Tutorial bin\package-db2.orm

JPA

- Similar to JDO:
  - Annotations
  - XML files
  - Class enhancement