Using JPOX to Develop a Persistence API for Generic Objects

Victor Travassos Sarinho

UEFS - Universidade Estadual de Feira de Santana, Depto. de Exatas,
Av. Universitária, s/n - Km 03 da BR 116, Campus Universitário,
CEP: 44031-460, Feira de Santana - BA – Brasil
vsarinho@gmail.com

Abstract. JPOX is an open source project that tries to be the reference between API JDO (Java Data Objects) implementations. Using JDO specifications, Java objects can be persisted automatically, independent of database management system. However, JDO builds the automatic persistence using object type definitions (classes). In this way, specifics query and update methods must be created for each type of object. Therefore, the objective of this work is develop an automatic persistence API for generic objects, based on JPOX project, avoiding database manipulation difficulties, and improving the development of object persistence based applications.

Keywords: *Enterprise Information Systems (EIS), Enterprise application, Enterprise software solutions, Enterprise modeling and integration, Enterprise model, JDO, JPOX, Generic objects persistence*

1. INTRODUCTION

Nowadays, many software technologies based on OO (Object Oriented) concept are used in EIS (Enterprise Information System) development. In fact, OO modeling languages have been developed [1], object persistence technologies have been built [2-4], business process modeling for OO technologies has been defined [5-6], and multiplatform programming tools have been created [7].

Using these technologies, the developer can define which information must be persisted (storable), how this information could be visualized (viewable), which operations can be executed (operable), and what logic execution must be applied for each system operation (taskable) during the development of an EIS.

For the persistence information, the developer can use the persistence layer technology, which is a library that allows the object persistence process, hiding unnecessary execution details. Following this technology, many persistence frameworks and tools were developed for Java [7], such as: Hibernate [8], Castor [9], OJB (Object-Relational Java Bridge) [10], Torque [11], JPOX [12], and many others. All of them try to eliminate previous user knowledge about DBMS (Database Management System) usage necessary for EIS data persistence.

However, the necessary work to use these frameworks and tools continues being so difficult, like traditional DBMS usage, because most of object-relational operations (Create, Recover, Update and Delete [13]) must be customized for each object type defined, returning to the same maintenance problems of the traditional relational persistence EIS.

Therefore, this work defines a Java API that apply an automatic persistence configuration for any object type (generic objects), based on JPOX project, avoiding difficulties in database repositories usage and improving the development of any persistence EIS.

2. JDO/JPOX PROJECT

JDO (Java Data Objects) [14] is an API to control Java objects persistence in relational databases, which provides a well defined interface for an abstraction layer between the developed application and many types of DBMS. The persisted objects in JDO specification are objects of simple Java classes called POJOs (Plain Old Java Objects); becoming unnecessary the implementation of certain interfaces or extends special classes.

The JPOX project (Java Persistent Objects) is an open source implementation compatible with JDO 1.0 and 2.0 specifications, providing a transparent persistence for Java objects. It supports the vast majority of RDBMS products available today, working with most important object-relational mapping (ORM), and allowing JDOQL or SQL queries to persisted objects.

2.1 Using JDO/JPOX to Persist an Object

To create an application with JPOX, the following steps will be necessary: design a domain/model class; define their persistence definition using Meta-Data; and write the code to persist the objects within the DAO layer.

Step 1 : Create the domain/model class. To give a working example (Figure 1), an application handling products in a store will be shown.

```
public class Product {
   String name = null;  String description = null;  double price = 0.0;
   protected Product() {}
   public Product(String name, String desc, double price) {
      this.name = name;
      this.description = desc;
      this.price = price;
   }
}
```

Figure 1. Class Product that will be Persisted

Step 2: Define the Persistence for the class. Now is necessary to define how the classes should be persisted (Figure 2), in terms of which fields are persisted etc. This

is performed by writing a Meta-Data persistence definition for each class in a JDO MetaData file.

Step 3: Write the code to persist objects of the class. Now is necessary to define which objects of this class is actually persisted, and when. Interaction with the persistence framework of JDO is performed using the PersistenceManager class (Figure 3). It provides methods for persisting, updating, deleting and querying persisted objects. Two examples (Figures 4 and 5) of typical scenarios in an application using these operations will be shown.

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
   "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
   "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
   <package name="org.jpox.tutorial">
      <class name="Product" identity-type="datastore">
         <inheritance strategy="new-table"/>
         <field name="name" persistence-modifier="persistent"/>
         <field name="description" persistence-modifier="persistent"/>
         <field name="price" persistence-modifier="persistent"/>
      </class>
   </package>
</jdo>
```

Figure 2. JDO Specification of Product

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

Figure 3. Getting access to PersistenceManager.

```
Transaction tx=pm.currentTransaction();
try {
   tx.begin();
   Product product = new Product("Sony Discman","A standard discman",49.99);
   pm.makePersistent(product);
   tx.commit();
}
finally {
   if (tx.isActive())
      tx.rollback();
   pm.close();
}
```

Figure 4. Persisting a Product Object

```
Transaction tx=pm.currentTransaction();
try {
   tx.begin();
   Extent e = pm.getExtent(org.jpox.tutorial.Product.class,true);
   Query q = pm.newQuery(e,"price < 150.00");
   q.setOrdering("price ascending");

   Collection c = (Collection)q.execute();
   Iterator iter = c.iterator();
   while (iter.hasNext()){
      Product p = (Product)iter.next();
      // ... (use the retrieved objects)
   }
   tx.commit();
}
finally {
   if (tx.isActive()) {
      tx.rollback();
   }
   pm.close();
}
```

Figure 5. Querying a Collection of Product Objects

3. PERSISTENCE API FOR GENERIC OBJECTS

The base of the developed API in this project is an interface called Storable, which must be implemented for all classes with objects to be persisted:

```
public interface Storable {
  // for each query an object clone must be produced to be returned
  public Storable getClone();
  // like the default constructor, returning the default object instance
  public Storable getDefaultClone();
  // object fields must be “refreshed” before DBMS operations
  public Collection getFieldsToRefresh();
  // which methods are used during fields update
  public Collection getMethodsForUpdate();
  // condition filter to identify this object as unique in DBMS
  public String getKeyFilter();
}
```

Figure 6. Storable Interface Definition

Any Storable object can be automatically persisted, and obtained by application queries too. These query and maintenance operations can be executed by the following generic operations described bellow in Figure 7:

```
public static DatabaseResponse executeTransaction(DatabaseOperation operation);
public static Collection executeTransaction(Collection operations);
public static DatabaseResponse executeQuery(Class theClass, Condition condition).
```

Figure 7. Static Methods of the StorableController Class

All EIS database operation uses the executeTransaction method, passing DatabaseOperation as a parameter. This DatabaseOperation object describe: the database operation to be executed (INSERT, UPDATE, DELETE, EXIST and NOT_EXIST), the object to be worked (Storable), an user level error message used if the DatabaseOperation fails during execution, and a condition filter object (Condition) which is necessary to define which objects the operation will work.

Condition uses a string to describe the condition filter, but other information can be provided, such as: variable declarations to be used during the query, imports for this variables, parameters with values, ordering of query results (ORDER BY) and a range of objects for the query return (rangeBegin and rangeEnd).

Collection of DatabaseOperation is worked too, allowing the user to create database transactions in the application. If one operation in the collection fails, all of them will fail too.

To execute query operations using the API, the executeQuery operation is the solution, which uses a Class reference to identify what object types are desired in the result dataset, and a condition filter which will define the objects that must be returned as query result.

The maintenance and query operations uses the DatabaseResponse object as a general result of these executions, which shows: whether the transaction execution complete, any error messages during the operation execution, and a collection of objects returned by a query operation execution.

3.1 Implementation Details

Each maintenance operation has a responsible execution method for data insert (storingObject), update (updatingObject) and removal (removingObject). EXIST and NOT_EXIST operations execute queries and, depending of the result data, exceptions can be risen showing that the operation condition (exist or not exist data in query execution) was not satisfied, and rollbacking the executed transaction from the API client application.

These maintenance operations use the Storable and the Condition objects to query an initial dataset that will be used during the operation execution. This object dataset is obtained by the current DatabaseOperation, which is used by the getObjects operation, responsible for all object queries in DBMS.

getObjects works only in the JDO transaction context, where each persistence/query operation must be executed after start a transaction (using the begin method of Transaction object) and before the end of this transaction (using the commit or rollback methods in Transaction object).

In this way, all objects in a JDO result query will be available only in the JDO transaction context. To allow any kind of operation in a persisted object out of this transaction context, the operation getClone available in Storable objects must be used. This cloning process is executed by executeQuery operation, which creates a clone for each object returned by getObjects and returns a collection of cloned Storable objects for the API client application.

During the insert operation execution, the API implementation verifies if the object was not previously stored in the DBMS, allowing the execution of this operation. For update and removal operations, the API implementation verifies if the object was previously stored in the DBMS, it allows updating or removing.

After insert/update operation validation, the API implementation must "refresh" the attributes of the object that will be persisted /modified. This step verify if exists in the object attribute list some Storable object that was previously stored in DBMS. If exists, this Storable object must be recovery and stored in the object attribute that will be persisted, changing the previous attribute value. If the attribute value is a collection of objects, the operation will verify each object in the collection, repeating the "refresh" recursively.

This "refresh" process is only necessary to avoid data duplication, using the condition filter to find any previous persisted object with the same value.

After update operation validation, the API implementation must execute the update method list in the Storable interface. Unfortunately, JDO does not have a default update operation (similar to insert and delete operations), because the JDO update process follow these steps: search for a persisted object, update the necessary attributes, and execute a commit to consolidate the operation. The API implementation uses a list of "methods for update" that's obtained by the getMethodsForUpdate method defined in Storable interface, and execute each one changing the desired attributes of the persisted object.

After delete operation validation, the API implementation executes the deletePersistentAll operation, which is the default JDO operation to remove persisted objects in the DBMS. If a Storable have any Storable type attribute, the Storable attribute reference must be deleted. But if the Storable type attribute were identified as an "autolife" attribute (an object that could exist without other objects dependencies) it will not be deleted. For Collections of Storables, the same process must be executed for each object.

3.2 Usage Example

Some code examples are listed bellow to show the User class definition implementing the Storable interface (Figure 8), and some persistence operations of a Storable object using the defined persistence API (Figure 9):

```
public class User implements Storable {
  // list of User class attributes
  private String login = "";
  private String password = "";
  private ArrayList actions = new ArrayList(); // list of Action objects   // Default
constructors for persistence operations
  public User(String login, String password, ArrayList actions) {
    this.login = login;
    this.password = password;
    this.actions = actions;
  }
  public User(){}

  // Getters and setters for object data manipulation
  ...
  public Storable getClone() {
    ArrayList actionsClone = new ArrayList();
    Iterator iter = this.actions.iterator();
    while (iter.hasNext())
      actionsClone.add((Action) iter.next().getClone());

    return new User(this.login, this.password, actionsClone);
  }
  public Storable getDefaultClone(){
    return new User();
  }
  public String getKeyFilter(){
    return "login == \"" + getLogin() + "\"";
  }
  public Collection getFieldsToRefresh(){
    ArrayList fields = new ArrayList();
    fields.add(new StorableItem("java.util.ArrayList", "actions", "Actions", true));
    return fields;
  }
  public Collection getMethodsForUpdate(){
    ArrayList methods = new ArrayList();
    methods.add(new MethodItem("Login","java.lang.String"));
    methods.add(new MethodItem("Password","java.lang.String"));
    methods.add(new MethodItem("Actions","java.util.ArrayList"));
    return methods;
  }
}
```

Figure 8. Creating an User Class to be Persisted Automatically

The StorableItem and MethodItem classes mentioned in Figure 9 are wrapper classes, containing informations about the attributes that must be “refreshed” in DBMS and which methods must be called to update the object during database update operations respectively.

```
// Creating an User object without Action objects
ArrayList users = new ArrayList();
User users = new User("admin", "admin", null);
users.add(user);

// Creating an insert DatabaseOperation
DatabaseOperation operation =
                    new DatabaseOperation(DatabaseOperation.INSERT, user,
                                                new Condition(user.getKeyFilter()));
// Persisting the User object
DatabaseResponse response = StorableController.executeTransaction(operation);
if (!response.isTransactionOk())
   System.out.println(response.getMessages().toString());
```

Figure 9. Persisting an User Object Automatically

Three steps can describes the above persistence example: the instantiation of the User object (straightforward EIS step); the instantiation of the DatabaseOperation object (this is the necessary "DBMS" knowledge to persist an object); and the execution of the DatabaseOperation (using the executeTransaction method) generating an DatabaseResponse object, which is the result of a successfull transaction or not (error messages will be printed if the transaction fails).

4. CONCLUSIONS

The persistence API for generic objects developed is a simple way to persist any object type that implements the Storable interface. It provides excellent results in development and maintenance activities during software creation, increasing the software production speed and omitting unnecessary details about persistence tecnologies.

Some Enterprise Information Systems have been developed using this persistence API by a local software company at Feira de Santana – Bahia, getting excellent results of robustness and efficiency in this process development and final EIS products. The validation of this project, developing real (not academic) EIS was necessary to consolidate it.

5. FUTURE WORKS

Some API simplifications must be done for the "refresh" and "update" operations, putting together the information of StorableItem and MethodItem wrapper classes, creating an unique wrapper containing all necessary information to persist the attribute data.

The evolution of the persisted object model must be treated because any change in the object model implies in a different object type, and all previous data stored in the

DBMS must be updated for this new object type (a very slow operation) to avoid object incompatibilities.

An AMDB (Attribute Management Database System), another persistence layer to store attributes instead of objects is in development too. Its objective is increase the execution speed (diminishing the number of software layers between the EIS and the DBMS, eliminating the JPOX dependency), and facilitate the software evolution (changes in the object attribute data) of any EIS developed using this persistence API, because only the object version will be changed (not the attribute data).

REFERENCES

1. J. Rumbaugh, UML Guia do Usuário (Ed. Campus, 2000).
2. ODMG, Object Database Management Group. http://www.odmg.org (Accessed May 20, 2007).
3. ORM, Object Relational Mapping. http://en.wikipedia.org/wiki/Objectrelational_mapping (Accessed May 20, 2007).
4. S. Ambler, The Design of a Robust Persistent Layer for Relational Databases. http://www.AmbySoft.com/persistenceLayer.pdf (Accessed May 20, 2007)
5. BPMN, Business Process Modeling Notation. http://en.wikipedia.org/wiki/Business_Process_Modeling_Notation (Accessed May 20, 2007).
6. WfMC, Workflow Management Coalision. http://www.wfmc.org (Accessed May 20, 2007).
7. Java, Java 2 Plataform. http://java.sun.com (Accessed May 20, 2007).
8. Hibernate, Object/Relational Mapping and Transparent Object Persistence for Java and SQL Databases. http://www.hibernate.org/ (Accessed May 20, 2007)
9. CASTOR, The CASTOR Project. http://www.castor.org/ (Accessed May 20, 2007).
10. OJB, Object Relacional Bridge. http://db.apache.org/ojb/ (Accessed May 20, 2007).
11. TORQUE, Torque. http://db.apache.org/torque/index.html (Accessed May 20, 2007).
12. JPOX, Java Persistent Objects. http://www.jpox.org (Accessed May 20, 2007).
13. CRUD, Create, Read, Update and Delete Acronym. http://en.wikipedia.org/wiki/Create%2C_read%2C_update_and_delete (Accessed May 20, 2007).
14. JDO, Java Data Objects. http://java.sun.com/products/jdo/ (Accessed May 20, 2007).