

Berufsakademie Mannheim

University of Cooperative Education



Independent Research Project

Object/ Relational Bridge (OBJ)
in the J2EE environment

Anne Holzappel, TIM01ANC
Matrikelnr. 171328

Claudia Buder, TIM01ANC
Matrikelnr. 132974

08.07.2004

Abstract

The following report deals with OJB - Object Relational Bridge - as an object relational mapping tool in the J2EE environment. It takes the usage and features into consideration provided by OJB. Besides the concepts and APIs that lays behind that layer, a deeper look into the used APIs, PersistenceBroker, JDO – Java Data Objects - and ODMG – Object Data Management Group – to persist objects are presented.

Furthermore, a deeper look into performance differences, advantages and disadvantages is made. Finally, the report ends in the introduction of EJB – Enterprise Java Beans - with regard to the cooperation of OJB and EJB and a detailed description of the deployment and usage of OJB in EJB based applications.

Within a two-tier architecture OJB is situated between the client application, where the client sends the requests and the database layer that can consist of different components. The OJB layer includes different features and APIs that will be explained in more detail in the following report.

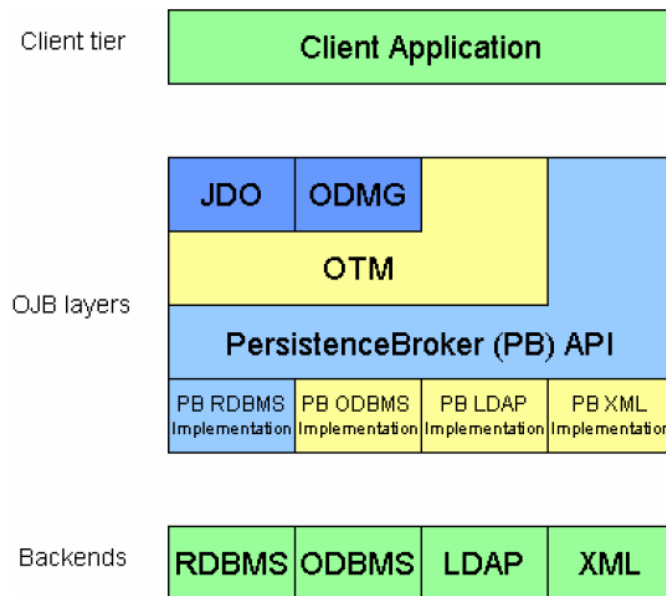


Figure 1: OJB Architektur*

* yellow fields are not yet implemented

Declaration of Academic Honesty

We hereby declare to have written the report on our own, using only the listed resources.

Location, Date

Anne Holzapfel, Matr.Nr. 171328

Location, Date

Claudia Buder, Matr.Nr. 132974

Content

ABSTRACT	2
DECLARATION OF ACADEMIC HONESTY	3
CONTENT	4
ABBREVIATIONS	5
MAIN PART	6
GENERAL DESCRIPTION	6
OBJECT/ RELATIONAL BRIDGE (OJB).....	7
<i>Deployment</i>	7
<i>Example</i>	9
<i>Mapping</i>	10
<i>Mapping inheritance hierarchies</i>	15
<i>Cascading</i>	18
<i>Proxies</i>	19
<i>Field conversion</i>	20
<i>Persistence broker</i>	21
<i>OJB queries</i>	23
<i>Sequence manager</i>	25
<i>Object cache</i>	27
<i>Nested objects</i>	28
<i>Stored procedures</i>	29
<i>Performance</i>	29
JDO & ODMG	30
DEPLOYMENT AND USAGE OF OJB IN EJB BASED APPLICATIONS	33
<i>Configuration</i>	34
<i>Using OJB within enterprise java beans</i>	36
CONCLUSION	37
ADDITION	38
ATTACHMENTS	38
BIBLIOGRAPHIES	40

Abbreviations

API	<u>A</u> pplication <u>P</u> rogram <u>I</u> nterface
EJB	<u>E</u> nterprise <u>J</u> ava <u>B</u> eans
J2EE	<u>J</u> ava <u>2</u> <u>E</u> nterprise <u>E</u> dition
JDO	<u>J</u> ava <u>D</u> ata <u>O</u> bjects
JDOQL	<u>J</u> DO <u>Q</u> uery <u>L</u> anguage
JNDI	<u>J</u> ava <u>N</u> aming <u>D</u> irectory <u>I</u> nterface
ODMG	<u>O</u> bject <u>D</u> ata <u>M</u> anagement <u>G</u> roup
OJB	<u>O</u> bject/ <u>R</u> elational <u>B</u> ridge
OQL	<u>O</u> bject <u>Q</u> uery <u>L</u> anguage
SQL	<u>S</u> imple <u>Q</u> uery <u>L</u> anguage
OTM	<u>O</u> bject <u>T</u> ransfer <u>M</u> ode
VM	<u>V</u> irtual <u>M</u> achine

Main part

General description

“OJB is an Object/Relational mapping tool that allows transparent persistence for Java Objects against relational databases.” (<http://db.apache.org/ojb>)

If you use an object oriented programming language like JAVA in the J2EE environment and a relational database to store your data you need a form of object/ relational mapping. Object/ relational mapping tools try to transform between object and relational modeling. To get an understanding of this problem a short description of relational and object modeling will be given.

Object modeling

Object modeling describes the world/ system as build out of objects. Objects are programming abstractions having identity, behavior and state. Objects include high level concepts like association, class and inheritance.

Relational modeling

Relational modeling describes information as predicate logic and truth statements. The most important concepts to the relational model are relation variables, tuples and attributes. In the relational database terminology this will correspond to databases, rows and columns.

The impedance mismatch

Object-oriented technology builds applications out of objects, containing both data and behavior. Relational technologies support the storage of data in tables, and data manipulation via data manipulation languages: internally via stored procedures, and externally via SQL calls.

Unfortunately, the fit between the two technologies is not perfect: there exists an object-relational impedance mismatch. The object-oriented paradigm is based on proven

software engineering principles. The relational paradigm, however, is based on proven mathematical principles. Since the underlying paradigms are different, the two technologies do not work together seamlessly.

The impedance mismatch becomes apparent in the context of the preferred accessing approach: the object paradigm lets you traverse objects via their relationships, while in the relational paradigm, you join data rows of tables. This fundamental difference results in a non-ideal combination of object and relational technologies.

This lack of impedance mismatch in Object DBMSs gives them a performance advantage over Relational DBMSs, especially on complex data. Impedance mismatch slows down performance on complex data because of processing needed map from one data structure (tables) to another (objects).

Object/ relational mapping tools try to bring the concepts of objects and relations together. The approach for mapping is vendor specific. O/ R tools provide a solution to the conflict between the application and the Database Management System (DBMS).

In OJB every class in the application is byte code post processed. The post processor changes the access code for the data object's instance variables where data objects are used as Java classes. The access and addresses are intercepted by calls to the object relational modeling engine. Within the application all classes have to be post processed. Sun JDO provides its own basic query language JDOQL . The OJB implementation supports the Sun JDO API. You will find more information about JDO in the chapter about persisting objects within OJB.

Object/ Relational Bridge (OJB)

Deployment

OJB can be used in two modes:

1. In standalone mode running in the same VM as the client application. Useful if only one instance of the client application is running (e.g. in applications with a local database or in single-server Servlet- or EJB application

2. In Client/Server mode. Multiple Clients can run against multiple PersistenceBrokerServers running in multiple VMs and/or on multiple physical machines. This is useful in typical Client/Server scenarios or when an ApplicationServer based application must be loadbalanced accross multiple servers.

Requirements

To deploy OJB several things have to be considered and set up.

The necessary OJB code is contained in the compiled OJB library within the db-ojb-
<version> .jar file (in the /lib directory). Additionally, several other jar archives are necessary that can be found in the /lib directory, too.

To deploy OJB in standalone applications additional packages are needed, which are not in the /lib directory. These should be downloaded separately (for example the j2ee.jar from SUN) and may need a license.

Furthermore, two kinds of configuration data are needed:

- configuration of the OJB runtime environment (OJB.properties)
- configuration of the MetaData layer (repository.xml)

These configuration files are read in through ClassLoader resource lookup and must therefore be placed on the classpath.

To define the JDBC connection to the runtime database, the repository.xml file and the respective jar archives in the classpath are used. The repository.xml can include several other xml files.

```
<!-- defining entities for include-files -->
<!DOCTYPE descriptor-repository SYSTEM "repository.dtd"
[
<!ENTITY user SYSTEM "repository_user_generated.xml">
<!ENTITY internal SYSTEM "repository_internal.xml">
]>
```

The repository_internal.xml includes the mapping of internal tables which ojb uses for example to do the locking and the sequence manager.

The repository_user_generated.xml file includes the mapping of the user defined tables, described in the section mapping. In this given source code the database connection is defined directly in the repository.xml file, but it also can be defined in a special file mostly called repository_database.xml which should then be included in the

repository.xml file. To separate the files for mapping and database definition is just for better clarity.

Deploying OJB for standalone and servlet based applications

There are four steps during the deployment that have to be followed:

- Deployment of the db-ojb- < version > .jar with the servlet application's WAR file (WEB-INF/lib)
- Deployment of the OJB.properties and repository.xml with the servlet application's WAR file (WEB-INF/classes)
- Adding the additional runtime jar archives to WEB-INF/lib
- Adding the JDBC driver's jar archive to WEB-INF/lib

Example

The following Entity Relationship Model (ERM) (see figure 1) describes the example we will use to describe how OJB maps database tables to objects and the other way around. All given example code will be based on this simple example to show the mechanisms of OJB.

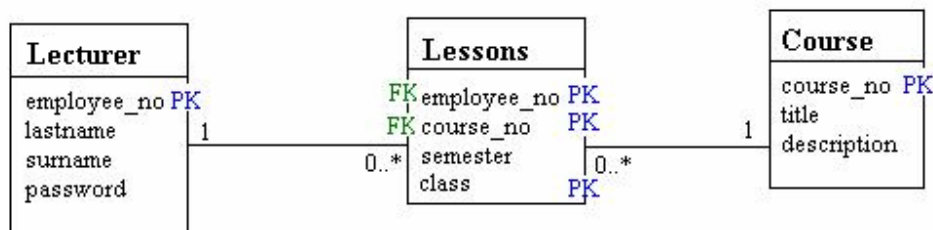


Figure 2: ERM university example – cut of

The ERM describes the relation between lecturers and courses. One lecturer can teach many courses and a course for example “Database systems” can be hold by different lecturers. Therefore we need a connection table holding the information about one course given by one lecturer in a special semester and in a special class for example “TIM01ANC”. This table is called lessons. The primary keys of the tables are marked as blue and the foreign keys are green.

Mapping

OJB uses an XML based Object/Relational Mapping. The mapping resides in a dynamic MetaData layer.

Advanced OJB Mapping techniques

- Mapping 1:1 associations
- Mapping 1:n associations
- Mapping m:n associations
- Creating load-, update- and delete-cascades
- Usage of proxy-classes
- Polymorphism
- Mapping of inheritance hierarchies

For getting the needed information on how to map classes to relational tables, the object relational metadata is used. Therefore, the repository.xml document is applied. The flexibility of OJB during the mapping of classes to the database tables is a great advantage. The mapping of a single class to a relational database is the easiest way. Therefore, the class is reproduced to the table and the attributes to a column. Finally, the rows are similar to the instances of the class.

The repository.xml is set up by the following physical components:

- Repository_user.xml: contains user-defined mappings
- Other management files: manages other metadata, like database information (connection information - username/password, connection configuration)

Additionally, every class consists of a class-descriptor composed of field-descriptors:

1. class-descriptor:
 - is build up by class (specifies the fully-qualified Java class name)
 - and table (specifies the used table for instances)
 - provides information about proxies and row-readers
2. field-descriptor:
 - is build up by name (name of the instance variable in the Java class)
 - and column (column in the table specified for this class)

- additionally, the primary-key and the autoincrement specification are set

Mapping can be done for three types of associations: for 1:1, 1:n (n:1) and n:m relations. In the following part the first two techniques will be described. The n:m association is not taken into account, because of the volition of database designs not to provide such inconsistent relations.

Mapping of 1:1 associations

1:1 relations exist for two tables where one entry of the one table fits only to one entry of the other one. OJB therefore, relies on the foreign key attributes. Following, a reference of the primary key of the first table is stored in the second table. The complete mechanism of the needed descriptors is shown in the next part, about 1:n relations. The sequence therefore is exactly the same and is described on an example.

Mapping of 1:n associations

By having a look on the chosen example it can be seen, that there is a 1:n relation between the lecturer and the lessons table. One lecturer may educate more than one or no course in one semester. But one lesson has to be given by exactly one lecturer. The primary key of the lecturer table 'employee_no' is referenced as foreign key in the lessons table.



First of all, the class for one table is shown below on the example of the lecturer table:

```
public class Lecturer
{
    private Int employee no;
    private String firstname;
    private String surname;
    private String password;

    public Int getEmployee_no()
    {
```

```

    return this.employee_no;
}
public void setEmployee_no(Int param)
{
    this.employee_no = param;
}
//other getter and setter methods not shown
}
    
```

The DDL for creating the lecturer table might look like shown next.

```

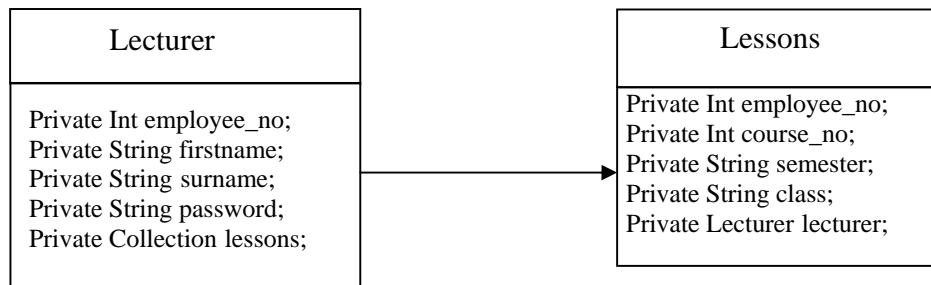
CREATE TABLE lecturer
(
    employee_no int not null,
    firstname char(50),
    surname char(100),
    password char(20),

    primary key(employee_no)
)
    
```

The field names in the database and in the class definition do not have to be the same but it simplifies the work with the names. The metadata defines which database column maps to what java class attribute.

As already mentioned, class- and field-descriptors are needed for the mapping.

Additionally, to declare the foreign key mechanics of the referenced attribute a reference-descriptor has to be added.



The association between the two tables is implemented using the Collection attribute lessons on the Lecturer class and the Lecturer attribute lecturer on the Lessons class. The lesson class contains a foreign key attribute named employee_no that identifies a lecturer from the lecturer table. To declare the foreign key mechanism of this collection attribute there has to be a collection-descriptor to the class-descriptor of the Lecturer class.

The following overview puts all the used descriptors together and specifies them shortly:

The class-descriptor element has two attributes:

- class → the Java class name for this mapping

Independent research project – OJB

- table → table which is used to store instances of this class

The element `field-descriptor` can have the following attributes, the first two are required and the other can be specified additionally:

- name → the name of the instance variable in the Java class
- column → column in the table used to store the value
- jdbc-type → type of the database column
- primary-key → primary key for the table
- autoincrement → value will be assigned by the OJB sequence manager
- conversion → custom field conversion

The `reference-descriptor` contains the following information:

- name → attribute implementing the association
- class-ref → the type of the referenced object

The `collection-descriptor` contains the following information:

- name → attribute implementing the association
- element-class-ref → the class of the element in the collection

Additionally the following attributes can be specified:

- sort → sort the retrieved collection
- order-by → order the retrieved collection
- proxy → see below

Now, the `repository.xml` including all the mentioned descriptors would look like the following for the given example:

```
<!--Definitions for table Lecturer-->
<class-descriptor
  class="com.ibm.test.objects.Lecturer"
  table="DB2INST1.LECTURER"
>
  <field-descriptor
    name="employee_no"
    column="EMPLOYEE NO"
    jdbc-type="INTEGER"
    primarykey="true"
    autoincrement="false"
  />
  <field-descriptor
    name="firstname"
    column="FIRSTNAME"
    jdbc-type="VARCHAR"
  />
</>
```

Independent research project – OJB

```
<field-descriptor
  name="lastname"
  column="LASTNAME"
  jdbc-type="VARCHAR"
/>
<field-descriptor
  name="password"
  column="PASSWORD"
  jdbc-type="VARCHAR"
/>
<reference-descriptor
  name="lesson"
  class-ref="com.ibm.test.objects.Lessons"
  >
    <foreignkey field-ref="employee_no"/>
</reference-descriptor>
</class-descriptor>

<!--Definitions for table Lessons-->
<class-descriptor
  class="com.ibm.test.objects.Lessons"
  table="DB2INST1.LESSONS"
  >
  <field-descriptor
    name="employee no"
    column="EMPLOYEE NO"
    jdbc-type="INTEGER"
    primarykey="true"
    autoincrement="false"
  />
  <field-descriptor
    name="course no"
    column="COURSE NO"
    jdbc-type="INTEGER"
    primarykey="true"
    autoincrement="false"
  />
  <field-descriptor
    name="semester"
    column="SEMESTER"
    jdbc-type="VARCHAR"
  />
  <field-descriptor
    name="class"
    column="CLASS"
    jdbc-type="VARCHAR"
    primarykey="true"
    autoincrement="false"
  />
  <collection-descriptor
    name="lecture"
    element-class-ref="com.ibm.test.objects.Lecturer"
    proxy="true"
  >
    <inverse-foreignkey field-ref="employee_no"/>
  </collection-descriptor>
</class-descriptor>
```

Writing the repository.xml file for only a few classes can easily be done. But keeping the repository synchronized with the java classes and the database gets more difficult if several hundred classes are involved.

The OJB ReverseDB tool

OJB provides a simple reverse engineering tool that allows to connect to a Relational Database Management System (RDBMS) via Java Database Connectivity (JDBC) and to take the tables from the database catalog as input.

This tool provides a GUI to generate Java classes and the matching repository.xml file. For the eclipse platform a plugin is available under <http://www.impart.ch/download.htm>, which has to be copied into the /eclipse/plugins folder and is loaded during next startup.

Mapping inheritance hierarchies

Additionally to the standard mapping mechanism described above which can be done automatically using the provided reverse DB tool you have the possibility to do advanced mappings. This includes the mapping of inheritance hierarchies, the mapping of associations, the use of proxy classes and field and type conversions.

Object oriented design and programming requires working with inheritance. This is not quite simple because we see now the so called “impedance mismatch” between object oriented programming and relational databases. OJB as an object/ relational mapping tools must now support inheritance and interfaces for persistent classes.

There are different possibilities to map inheritance hierarchies to database tables.

Consider the following UML diagram (see figure 2):

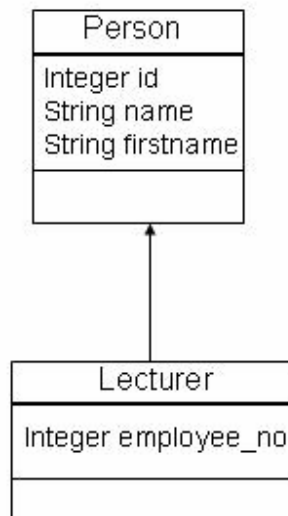


Figure 3: UML example lecturer as member of person

There are different ways to define database tables so that they contain the two classes. Firstly you can map both classes to one table holding the attributes of both classes. The second solution is to map each class to an extra table and the derived class includes all attributes from the base class. The third possibility is to map each class to a single table but the derived class does not hold the attributes of the base class. You will use joins between the tables instead and therefore store the primary key of the base class as foreign key in the derived class to materialize the objects.

OJB supports all of these solutions.

Mapping all classes to one table

The java code for the given example (figure 2) will look like this:

```
public abstract class person
{
    protected String ojbConcreteClass;
    protected int id;
    ...
}

public class lecturer extends person
{
    int employee_no;

    public A()
    {
        ojbConcreteClass = A.class.getName();
    }
}
```

Independent research project – OJB

OJB allows using interfaces, abstract, or concrete base classes (like the class `getName()`). OJB needs a column of type `char` or `varchar` that contains the classname to be used for instantiation. This column must be mapped on a special attribute `objConcreteClass`. On loading objects from the table OJB checks this attribute and instantiates objects of this type. This means for the belonging class-definition:

```
<!-- Definitions for extent myClasses.Person -->
  <class-descriptor class="myClasses.Person">
    <extent-class class-ref="myClasses.Lecturer" />
  </class-descriptor>

<!-- Definitions for myClasses.Lecturer -->
  <class-descriptor
    class="myClasses.Lecturer"
    table="TEST_TABLE"
  >
    <field-descriptor
      name="id"
      column="ID"
      jdbc-type="INTEGER"
      primarykey="true"
      autoincrement="true"
    />
    <field-descriptor
      name="objConcreteClass"
      column="CLASS_NAME"
      jdbc-type="VARCHAR"
    />
    <field-descriptor
      name="name"
      column="NAME"
      jdbc-type="VARCHAR"
    />
    <field-descriptor
      name="employee_no"
      column="EMPLOYEE_NO"
      jdbc-type="INTEGER"
    />
  </class-descriptor>
```

Extents have to be declared in the repository file. A declaration for the baseclass `Person` is necessary, defining which classes are subclasses of it, in this case the `lecturer` class.

Mapping Each Class to a Distinct Table

This is the simplest solution. Just write a complete `ClassDescriptor` for each class that contains `FieldDescriptors` for all of the attributes, including inherited attributes.

Mapping classes on joined tables

We take the java classes from the part “mapping classes to one table”. We need the id from the person class as a foreign key in the lecturer class. To define such a field without having an attribute in the java class we can use OJB’s anonymous field feature. We keep the field descriptor for the person’s id and declare it as anonymous field just adding an attribute `access="anonymous"` to the field-descriptor.

```
<class-descriptor
  class="myClasses.Lecturer"
  table="TEST_TABLE"
>
  <field-descriptor
    name="personID"
    column="PERSON_ID"
    jdbc-type="INTEGER"
    access="anonymous"
  />
  <field-descriptor
    name="name"
    column="NAME"
    jdbc-type="VARCHAR"
  />
  <reference-descriptor name="super"
    class-ref="myClasses.person">
    <foreignkey field-ref="personID" />
  </reference-descriptor>
</class-descriptor>
```

The way OJB provides inheritance is relatively easy to use. You just have to take the decision which way of mapping you want to use and then apply the step described before.

Cascading

As OJB manages associations by declaring special reference- and collection-descriptors, there are also additional information on OJB’s behavior on object materialization, updating and deletion. The default values therefore are:

- Auto-retrieve = "true" (by using the `PersistenceBroker.getObjectByQuery()` all referenced objects are materialized)
- Auto-update = "false" (by using the `PersistenceBroker.store()` the referenced objects are not updated)
- Auto-delete = "false" (by using the `PersistenceBorker.delete` the referenced objects are not deleted)

These values can be set manually, to characterize the behavior on referenced objects:

- Auto-retrieve = "false" (the referenced attributes can be loaded manually by

Independent research project – OJB

using `PersistenceBroker.retrieveReference()` and
`PersistenceBroker.retrieveAllReferences()`

- Auto-update = “true” (the referenced objects are also persisted to the database)
- Auto-delete = “true” (the referenced objects are also deleted if an instance of the persistent class is deleted)

Proxies

There are two possibilities to load values belonging to an object:

1. Loading it immediately after loading the object. This will be done with ordinary SQL-calls.
2. Using proxies. When accessing an object-proxy OJB will have to materialize it with another SQL-call.

For ‘lazy loading’ or ‘lazy materialization’ proxy classes that reduce the number of DB calls can be used.

Without using proxies, all associated objects are immediately loaded from the database even if they are of no interest. By using them, the collection is filled with the number of objects as proxy objects that implement the same interface with real objects. But, they contain only an object ID and a void reference. Therefore, not all values are instantiated when the object is initially materialized. Only when the method is invoked on a needed proxy object it is really called. That causes a minimization of instantiation of persistent objects and database lookups.

The used interface allows the replacement of the object with a proxy, implementing the same interface. The value ‘proxy’ has to be set to “true” in the collection-descriptor.

Dynamic proxies are provided by OJB and can be set in the repository.xml.

There is the possibility to define a single proxy for a whole proxy collection. The advantage is, that the number of database calls can be reduced compared to the usage of proxy classes. Only one single database call is needed and an additional call calculates the size of the collection. So, the collection proxy is used as a deferred execution of a query.

A collection proxy class can also be customized in the OJB.properties. If there is no entry, the `org.apache.ojb.broker.accesslayer.ListProxy` is used for lists and the `org.apache.ojb.broker.accesslayer.CollectionProxy` for collections. Furthermore, a proxy can be used for a reference based on the original proxy class concept. Therefore, instead of the `ClassDescriptor` the `ReferenceDescriptor` defines when to use a proxy class. But the proxy reference only shows the location of the definition, so that the referenced class has to implement a special interface, as described above.

Field conversion

Often, the type of an attribute of a database column doesn't match with the attribute type of the domain model. To overcome this problem, field conversion is used. The table in the appendix shows which datatypes of the database can be converted into which attribute type.

An example is the conversion of the JDBC type integer into the Java type boolean. The method `FieldConversion.sqlToJava()` is a callback that is called within the OJB broker when object attributes are read in from JDBC result sets. If OJB detects that a field conversion is declared for a persistent classes attributes, it uses the field conversion to do the marshalling of this attribute.

For the example, the source code may look like the following:

```
public class Char2StringFieldConversion implements FieldConversion
{
    private static Integer I_TRUE = new Integer(1);
    private static Integer I_FALSE = new Integer(0);
    private static Boolean B_TRUE = new Boolean(true);
    private static Boolean B_FALSE = new Boolean(false);

    public Object javaToSql(Object source) {
        if (source instanceof Boolean) {
            if (source.equals(B_TRUE)) {
                return I_TRUE;
            }
            else {
                return I_FALSE;
            }
        }
        else {
            return source;
        }
    }
    public Object sqlToJava(Object source) {
        if (source instanceof Integer) {
            if (source.equals(I_TRUE)) {
```

```
        return B_TRUE;
    }
    else{
        return B_FALSE;
    }
}
else{
    return source;
}
}
}
```

Both, the integer and the boolean value can have a “true” and a “false” state, the integer value will be 0 for false and 1 for true. The source value is analyzed for its format. For the way from integer to Boolean, the source value has to be an Integer, so that the value is converted to a Boolean and set to true. It’s the same way from Boolean to integer. If you want to write your own field conversion class it has to implement FieldConversion.

Afterwards, the attribute `conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"` has to be set in the `field-descriptor` and the conversion is done when materializing the Object from the database.

There are other helpful standard conversions defined in the package `org.apache.ojb.broker.accesslayer.conversions`.

Persistence broker

To delete, update or insert objects into the database the PersistenceBroker API is needed that provides the lowest level access to OJB’s persistence engine.

The point of access for the operations is the

`org.apache.ojb.broker.PersistenceBroker` class. Three steps are needed to insert or update an object:

- obtain a persistence broker
- store the object via the PersistenceBroker
- close the PersistenceBroker

The following function stores an object in the database

```
public static void store(Object object) {
    PersistenceBroker broker=null;
    try{
        broker =
```

Independent research project – OJB

```
        PersistenceBrokerFactory.defaultPersistenceBroker();
        broker.store(object);
    }
    catch(PersistenceBrokerException e) {
        e.printStackTrace();
    }
    finally {
        if (broker != null) broker.close();
    }
}
```

The two main classes that are used are the `PersistenceBrokerFactory` and the `PersistenceBroker`.

The `PersistenceBrokerFactory` class manages the lifecycle of `PersistenceBroker` instances: it creates them, pools them, and destroys them as needed. By using the static `PersistenceBrokerFactory.defaultPersistenceBroker()` method an instance of the `PersistenceBroker` is obtained (mostly due to one database for an application). As it can be seen it is very easy to insert new objects into the database. The `broker.close()` call is made within a `finally` block, so that the broker is closed and returned back to the broker pool even if there was an exception. OJB makes no difference between inserts and updates. Both use the same function `PersistenceBroker.store(Object)` to make an object persistent. If the primary key of the submitted objects exists in the database an update is performed otherwise a new object is inserted. The deletion of objects is as easy as the insertion/ update. Only the `PersistenceBroker.delete(Object)` has to be called to delete an object from the database. The persistent object is deleted from the repository but doesn't affect any change on the object itself.

To get objects back from the database the `PersistenceBroker` provides two mechanisms to build queries using template objects or specifying criteria (see chapter OJB queries). There should be any properties set which should be matched by the query. Properties with null values are not used to match. The more flexible way is to construct the criteria on the query by hand.

```
public static Object getEntryByPrimarykey(String id) {
    //Init some vars
    Object object = null;

    //Create Criteria
    Criteria crit = new Criteria();
    crit.addColumnEqualTo("ID", id);

    //Create Query
```

```
Query query = new QueryByCriteria(Object.class, crit);
try{
    //Execute Query
    res = (Object) broker.getObjectByQuery(query);
}
catch (PersistenceBrokerException e){
    e.printStackTrace();
}
return res;
}
```

The Criteria object provides several methods described in the chapter OJB queries .

After the criteria have been build a query object has to be created. To get the results from the database the `getObjectByQuery(query)` or `getCollectionByQuery(query)` is used. The first method returns an object and the second a collection.

Which one to use has to be decided on the number of results that is needed, because `getObjectByQuery` only returns the first matching result.

Transactions are also supported from the `PersistenceBroker` API. It uses the database transaction mechanism. The `broker.beginTransaction()` has to be called and when finishing the database queries `broker.commitTransaction()`.

It is also possible to pool `PersistenceBrokers`. Therefore, the

`PersistenceBrokerFactory` pools `PersistenceBroker` instances. As mentioned before, the `PersistenceBroker.close()` method releases the broker back to the pool.

For exception handling, the `org.apache.ojb.broker.PersistenceBrokerException` is thrown. No try/catch block is required therefore.

OJB queries

The computation of queries against the database is kept very simple in OJB when using the persistence broker, described in the section before. The classes for building queries by criteria are located in the package `org.apache.ojb.broker.query`. OJB offers a query factory to create new criteria.

```
Query q = QueryFactory.newQuery(Class.class, crit);
```

For distinct queries just add "true" as third parameter to the `newQuery` method. There are selection criteria provided for nearly every SQL-comperator. Each criterion stands for a column in the SQL WHERE clause. The different criteria are added by an AND between the statements.

Independent research project – OJB

```
Criteria crit = new Criteria();  
crit.addEqualTo("Column name", "value");  
crit.addLike("Column name", "pattern");  
Query q = QueryFactory.newQuery(Class.class, crit);
```

The resulting SQL-statement will be:

```
SELECT * FROM Class WHERE Column name = "value" AND Column name LIKE  
"pattern";
```

To get an OR combination two criteria sets are needed. These sets are combined using `addOrCriteria`.

```
Criteria crit1 = new Criteria();  
crit1.addLike("column1", "%o%");  
Criteria crit2 = new Criteria();  
crit2.addEqualTo("column2", "zero");  
  
crit1.addOrCriteria(crit2);  
Query q = QueryFactory.newQuery(Class.class, crit1);  
Collection results = broker.getCollectionByQuery(q);
```

Ordering and grouping is also done easily with

`query.addOrderByDescending("column")` or

`query.addOrderByAscending("column")` and `query.addGroupBy("column")`.

Multiple order and group by clauses are also possible by just repeating `addOrderBy` or `addGroupBy`. The having clause does not have its own statement but can be applied when using `addSQL` (see below).

Subqueries can be used instead of values in the selection criteria. Thus they should be report queries which do not query for whole objects but return row data.

The following example taken from the OJB tutorial queries all articles having a price greater or equal than the average price of articles named 'A%':

```
ReportQueryByCriteria subQuery;  
Criteria subCrit = new Criteria();  
Criteria crit = new Criteria();  
  
subCrit.addSQL(UPPER(name) like 'RE%');  
subQuery = QueryFactory.newReportQuery(Article.class, subCrit);  
subQuery.setAttributes(new String[] { "avg(price)" });  
  
crit.addGreaterOrEqualThan("price", subQuery);  
Query q = QueryFactory.newQuery(Article.class, crit);
```

```
Collection results = broker.getCollectionByQuery(q);
```

The example shows how report queries are build and that they just are used as an attribute in the `addGreaterOrEqualThan` Criteria.

As shown above also complex queries can be build very easily using the PersistenceBroker API. You don't have to program SQL anymore but have the possibility to add sql with the `addSQL` statement.

Sequence manager

The OJB sequence manager is used to assign unique values to primary key attributes. The automatic assignment only works for columns of type integer, char or varchar. The computation is done during the call to `PersistenceBoker.store(...)`. If you need to have the ID before storing the object you can get the primary key value like this

```
String primary_key;  
Identity id= new Identity(Object,PersistenceBroker);  
primary_key= id.getPrimaryKeyValues()[0].toString();
```

OJB provides different implementations for the sequence manager. You can specify the sequence manager implementation in the repository file (see above) by just adding a sequence manager within the connection descriptor.

```
<sequence-manager className="org.apache.ojb.broker.util.  
    sequence.SequenceManagerHighLowImpl">  
</sequence-manager>
```

If no sequence manager is defined the default sequence manager implementation `SequenceManagerHighLowImpl` is used.

SequenceManagerHighLowImpl

The `HighLowSequenceManager` generates unique IDs to a given object and all extend objects declared in the objects class descriptor.

This implementation can also be used to assign global unique values.

```
<sequence-manager className=  
    "org.apache.ojb.broker.util.sequence.SequenceManagerHighLowImpl">  
  
    <attribute attribute-name="grabSize" attribute-value="20"/>  
    <attribute attribute-name="globalSequenceId"  
        attribute-value="false"/>  
    <attribute attribute-name="globalSequenceStart"  
        attribute-value="10000"/>  
    <attribute attribute-name="autoNaming"  
        attribute-value="true"/>
```

```
</sequence-manager>
```

If you want to assign global unique values over all persistent objects in a database you should set the attribute `globalSequenceId` to `true`. With `grabSize` the size of the assigned ids is set. The attribute `globalSequenceStart` defines the start value for the ids.

There are several other implementations available. Another interesting implementation is a database sequenced implementation.

SequenceManagerNextValImpl

This sequence manager lets your database create the unique id. This is possible if your database provides sequence key generation like Oracle does.

You also have the possibility to write your own sequence manager implementation. This may be of interest if you want to generate unique ids also when using multiple databases.

Write your own sequence manager

You write a class of the interface

`org.apache.ojb.broker.util.sequence.SequenceManager`. OJB can be configured to generate instances of your specific implementation by adding a `sequence-manager` tag in the `jdbc-connection-descriptor`.

```
<sequence-manager className="my.SequenceManagerMYImpl" >
</sequence-manager>
```

You can pass configuration properties to your implementation using `attribute` tags.

```
<sequence-manager className="my.SequenceManagerMYImpl" >
  <attribute attribute-name="myProperty" attribute-value="test" />
</sequence-manager>
```

With

```
public String getConfigurationProperty(String key, String defaultValue)
```

method you get the properties in your implementation class.

Independent research project – OJB

Most `SequenceManager` implementations are based on sequence names. To control the sequencing use the `sequence-name` attribute in the `field-descriptor`. In that case you are responsible to use the same name across extents to make sure that you're the generated key is unique over all tables you want.

Using the sequence manager can ease the handling of key generation but you have to be careful when deciding which implementation to use. The implementations are not applicable when you have for example other non OJB applications which insert objects or you have to be really careful in using this.

You may also get into trouble when using the sequence manager in clustered environments but this will be discussed later.

Object cache

Using caching mechanism can increase the performance of you application. They reduce database look-ups and object materialization. Each Instance of the interface `PersistenceBroker` (will be described in one of the next chapters) use its own `ObjectCache` instance. This instance will hold objects previously loaded by the `PersistenceBroker`. When looking up an object the `PersistenceBroker` will not persorm the `SELECT` against the database but will first look if the object is already loaded into the object cache. The same is done when materializing an object, first the object cache is checked and only if the object has not been loaded the query is performed against the database.

There are different `ObjectCache` implementations available within OJB and like the sequence manager you can also write your own object cache. To change the used implementation you have to change the `OJB.properties` file.

```
#-----  
# Object cache  
#-----  
# The ObjectCacheClass entry tells OJB which concrete instance Cache  
# implementation is to be used.  
ObjectCacheClass=org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl  
#
```

Additionally you have to add an `object-cache` element within the `jdbc-connection-descriptor`.

```
<object-cache  
    class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">  
</object-cache>
```

You also can declare the object cache implementation for every java class. If a caching mechanism is defined at the `class-descriptor` then it will have the higher priority than the cache declaration on the `jdbc-connection-descriptor`.

The interface of the `ObjectCache` is quite simple.

```
public interface ObjectCache  
{  
    /**  
     * Write to cache.  
     */  
    public void cache(Identity oid, Object obj);  
  
    /**  
     * Lookup object from cache.  
     */  
    public Object lookup(Identity oid);  
  
    /**  
     * Removes an Object from the cache.  
     */  
    public void remove(Identity oid);  
  
    /**  
     * Clear the ObjectCache.  
     */  
    public void clear();  
}
```

If you want to write your own implementation you only have to use this interface and declare it in the `OJB.properties` file as mentioned above. Writing your own implementation is only useful if the given ones are too simple for what you want to do. All shipped implementations are listed in the `org.apache.ojb.broker.cache` package. If you want to use OJB in a clustered environment you should distribute all shared cached objects across different JVM. This will also be described in the chapter OJB in clustered environments.

Nested objects

If you want to use nested objects OJB also supports this just by changing some lines in the repository file. The difference is that you do not have to declare the referenced class in the repository file. Instead of doing this you just add nested fields in the superior class' `field-descriptor` by using `::` to specify attributes of the nested object. All aspects of

storing and retrieving the nested object are managed by OJB (for a detailed example see <http://db.apache.org/ojb/tutorial3.html#Nested%20Objects>).

Stored procedures

Within OJB stored procedures are supported to handle the basic Data Manipulation Language (DML) operations Insert, Update and Delete. Some entries have to be made to the repository file in the class-descriptor to utilize stored procedures instead of the DML statements. Stored procedures can in this context be used to use triggers on certain events. (see <http://db.apache.org/ojb/howto-work-with-stored-procedures.html> for details).

Performance

Object/relational mapping tools hide the details of relational databases from the application developer. The developer can concentrate on implementing business logic and is liberated from caring about RDBMS related coding with JDBC and SQL.

O/R mapping tools allow to separate business logic from RDBMS access by forming an additional software layer between business logic and RDBMS.

The advantage of OJB is that the SQL generation is made by the O/R-Tool. The test runs are extremely reduced and the application becomes more stable. The development time can be reduced after the first orientation time.

The disadvantage of OJB is the performance because introducing new software layers needs additional computing resources. Software architects have to take into account this tradeoff between programming comfort and performance to decide if it is appropriate to use an O/R tool for a specific software system.

But performance shouldn't be the only reason to take a specific O/R tool. There are many other points to consider:

- Usability of the supported API's
- Flexibility of the framework
- Scalability of the framework
- The different licenses of Open Source projects

There are some settings which affect the performance:

The API you use, e.g. PersistenceBroker-API is much faster than the ODMG-API

Independent research project – OJB

- ConnectionFactory implementation / Connection pooling
- PersistentField class implementation
- Used sequence manager implementation
- Use of batch mode (when supported by the DB)
- PersistenceBroker pool size

The use of OJB as a layer between the RDBMS and the object oriented programming language should be decided according to the specific project. In some applications it may reduce development time and will therefore be of advantage, in other applications the increased performance will be the main factor not to use OJB.

JDO & ODMG

OJB supports multiple APIs:

- A full featured ODMG 3.0 compliant API.
- A JDO compliant API
- A low-level PersistenceBroker API which serves as the OJB persistence kernel.

The main feature projected for OJB 2.0 is a JDO(R) implementation (Java Data Objects, the Java standard extension for object persistence as proposed by SUN).

As of now OJB provides a full featured ODMG (Object Data Management Group) 3.0 implementation that is build on top of a persistence kernel, the so called PersistenceBroker. The ODMG implementation resides in the package `objb.odmg`. The PersistenceBroker kernel resides in the package `objb.broker`. The ODMG implementation is concerned with providing object-level transactions coordination, performing OQL Queries ("Object Query Language") and supporting specific persistent collections.

During an ODMG transaction commit it uses the PersistenceBroker kernel to persist objects.

JDO(R) is similar to ODMG in many respects (in fact ODMG may even be considered as a precursor of JDO). In the following, the similarities between JDO and ODMG are taken into consideration as examples for other possible object relational APIs.

Independent research project – OJB

Both APIs support the transparent persistence of Java object models. Instances can be made persistent explicitly and implicitly by using persistence-by-reachability. In both APIs the method for explicitly making an instance persistent is called `makePersistent`, defined in JDO's `javax.jdo.PersistenceManager` and ODMG's `org.odmg.Database` interfaces. They also share the same method name for explicit deletion, `deletePersistent`, defined in the same interfaces. Both support an interface called `Transaction`, with methods named `begin` and `commit`.

Those who have criticized JDO as being just like ODMG justify their claims by pointing out these method names as examples of the similarities of the APIs. But these are the only methods in common between the two APIs. This is hardly a basis for the criticisms.

Both ODMG and JDO support the management of a cache of instances from the database, ensuring only a single copy in the cache of each persistent object. They allow the application to traverse Java references, bringing objects into the cache on an as-needed basis. Furthermore, both automatically mark objects that have been modified and propagate all updates to the database at commit. These are the kind of features one expects in a transparent persistence technology. So, naturally, both APIs support these features.

Otherwise, there are some fundamental differences between JDO and ODMG. In particular, the data model for each is substantially different. Other differences exist in the areas of metadata syntax, identity, collections, query language, and concurrency control. Both APIs require metadata to indicate which classes are persistent and to specify additional persistence-related information. JDO uses XML as its representation for metadata, whereas ODMG uses a non-standard text format.

ODMG only provides support for data store identity. JDO defines three forms of identity: data storage, application and non-durability. Implementations must implement at least one of data store and application identity. Many implementations are supporting both. ODMG defines its own collection interfaces in package `org.odmg`. These include `DBag`, `DSet`, `DList`, `DArray`, and `DMap`. These interfaces extended the collection interfaces defined in package `java.util`. ODMG applications must call a factory method defined in the `org.odmg.Implementation` interface to construct instances of vendor-specific classes that implement these ODMG collection interfaces. This means that the persistent

classes need to import ODMG-specific interfaces and use them throughout their persistent data model.

JDO allows applications to directly use the collections defined in package `java.util`. JDO does not require you to import any JDO-specific types into the source of your persistent classes.

ODMG allows an implementation to use either pessimistic or optimistic concurrency control. JDO requires support of pessimistic and allows optimistic as an optional feature. JDO provides a very detailed description of the semantics of these two concurrency control policies, whereas ODMG gave implementations the flexibility to do whatever best suited their architecture. Using JDO will yield more consistent program behavior across implementations.

The query language facilities of ODMG and JDO are substantially different. ODMG has a query language called the Object Query Language (OQL). OQL provides support for computing the Cartesian product of arbitrary collection expressions (this is more general than the Cartesian product computed in SQL). JDOQL does not compute a Cartesian product result. It provides a means to filter out elements of an extent or collection, returning those elements for which the filter expression is true. OQL also allowed one to create projections or views of the data. And it supports aggregate functions. These query features are not possible in JDOQL.

The most fundamental difference between the two APIs is availability of implementations. JDO has only been an adopted standard since March 2002, but it is already far more successful than ODMG. Whereas ODMG has only a few implementations, there are already over three times as many JDO implementations available and more are due out soon. ODMG was an API designed by object database companies without much regard for relational implementations. Extra care was taken designing JDO to ensure it could be supported for a wide variety of data stores. The availability of many relational JDO implementations shows the greater applicability of JDO. The fact that JDO was defined through the Java Community Process has greatly increased its credibility within the industry.

JDO also has a number of features that ODMG does not provide. These include the following:

Independent research project – OJB

- Retaining objects across transactions
- Transient transactional instances
- Transient transactional fields
- InstanceCallbacks for managing instances during certain cache events
- Management of null fields
- Cache management methods like evict, refresh and retrieve
- Embedded objects
- Query compilation
- Application server integration via the Java Connector API

One of the required steps in the Java Community Process is the development of a compatibility test suite. This is used to ensure implementations are compatible with the standard. ODMG does not have a test suite. Consequently, there are incompatibilities among the few implementations.

Perhaps the most important feature unique to JDO is binary compatibility. JDO has a standard enhancement contract that all implementations must adhere to. A class enhanced with any JDO enhancer must work with any other JDO implementation. In ODMG class enhancement is vendor-specific, so there is no application binary portability across implementations.

The attached table (see attachment 2) shows a summary of the similarities and differences between the two approaches to transparent persistence: ODMG 3.0 and JDO.

While all three APIs support transparent persistence of object models, JDO has a far richer feature set, provides application portability, and supports a wider variety of data stores. Those developing new applications should consider JDO and those that have been using ODMG or other proprietary object persistence technologies should consider migrating to the JDO standard.

Deployment and usage of OJB in EJB based applications

OJB can be used for stand-alone applications and servlet based applications. But furthermore, OJB also can work together with EJB (Entity Java Beans). For connecting

these two concepts, OJB has to be deployed in the EJB application. An insight of how that works gives the next section.

First of all, the ordinary deployment steps for deploying OJB have to be applied. Additionally, some other configurations have to be made that are similar for different application servers. The main topics that have to be regarded in detail are connection handling, caching and locking. The following step by step description gives an overview on how to deploy OJB within EJB.

Configuration

For the JBoss environment the following configurations have to be made.

Adaption of the OJB.properties file

If the PersistenceBroker API is used a special PersistenceBrokerFactory class can be used.

```
PersistenceBrokerFactoryClass=  
org.apache.ojb.broker.core.PersistenceBrokerFactorySyncImpl
```

Furthermore, the OJB.properties file needs additional settings:

```
ConnectionFactoryClass=  
org.apache.ojb.broker.accesslayer.ConnectionFactoryManagedImpl  
  
OJBTxManagerClass=org.apache.ojb.odmg.JTATxManager  
  
# set used application server TM access class  
JTATransactionManagerClass=  
org.apache.ojb.otm.transaction.factory.JBossTransactionManagerFactory
```

Declaration of the datasource in the repository file

The DataSource from the application server should be defined to be used for the connection to the database.

Afterwards, OJB has to be defined to use this DataSource.

```
<jdbc-connection-descriptor  
  jcd-alias="default"  
  default-connection="true"  
  platform="DB2"  
  jdbc-level="2.0"  
  jndi-datasource-name="java:DefaultDS"  
  username="test"  
  password="welcome"
```

```
batch-mode="false"  
useAutoCommit="0"  
ignoreAutoCommitExceptions="false">  
</jdbc-connection-descriptor>
```

Including all OJB configuration files in classpath

The needed OJB configuration files are the following:

- OJB.properties
- repository.dtd
- repository.xml
- repository_internal.xml
- repository_database.xml,
- repository_ejb.xml

Resulting, the repository.xml file the use of EJB beans looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>  
  
  <!-- defining entities for include-files -->  
  <!DOCTYPE descriptor-repository SYSTEM "repository.dtd" [  
  <!ENTITY database SYSTEM "repository_database.xml">  
  <!ENTITY internal SYSTEM "repository_internal.xml">  
  <!ENTITY ejb SYSTEM "repository_ejb.xml">  
  ]>  
  
  <descriptor-repository version="1.0"  
    isolation-level="read-uncommitted">  
    <!-- include all used database connections -->  
    &database;  
    <!-- include ojb internal mappings here -->  
    &internal;  
    <!-- include mappings for the EJB-examples -->  
    &ejb;  
  </descriptor-repository>
```

Enclosurement of all libraries OJB depends on

To make the OJB sample session beans run on the following libraries have to be included:

- The Jakarta commons libraries files (all commons-xxx.jar) from OJB /lib directory
- The antlr jar file (antlr-xxx.jar) from OJB /lib directory
- jakarta-regexp-xxx.jar from OJB /lib directory

Consideration of caching

Caching has to be synchronized with the used database. By using the `PersistenceBroker` API, the given `ObjectCache` implementations can be used in non-clustered environments. In the `ObjectCacheDefaultImpl` cache implementation the `autoSynch` mode has to be enabled.

For clustered environments a distributed `ObjectCache` or a local/empty cache is used. An own `ObjectCache` implementation can be written (as mentioned in the chapter object cache).

Consideration of locking

Unfortunately, the `PersistenceBroker` API does not support Object Locking (ODMG API in contrast does).

For clustered environments a distributed lock management is necessary.

Bringing all files together

All the files have to be put together

The structure of the `ejb.jar` file should for an example look like this:

```
/OJB.properties
/repository.dtd
/repository.xml
/all used repository-XYZ.xml
/META-INF
.../Manifest.mf
.../ejb-jar.xml

/all ejb classes

/db-ojb-1.X.jar
/all used libraries
```

Using OJB within enterprise java beans

Bean classes directly use OJB main classes. Furthermore it is possible to make OJB accessible via JNDI (Java Naming D Interface) and use a JNDI-lookup to access OJB API's in the beans.

To bind the OJB API's to JNDI to make them accessible these main classes/method should be bound:

Independent research project – OJB

- Class `org.apache.ojb.broker.core.PersistenceBrokerFactoryFactory` for PersistenceBroker API. Make method `PersistenceBrokerFactoryFactory.instance()` accessible

It is possible to bring OJB and EJB together. For a JBOSS example how to combine session beans with the classes OJB provide have a look at:

<http://db.apache.org/ojb/deployment.html#OJB%20sample%20session%20beans>

The question is whether this really makes sense. EJB and OJB both provide mechanism to persist objects and therefore it is not necessary to use both concepts in one application. Maybe OJB simplifies the high complexity of EJBs, especially of bean managed persistence (BMP).

Conclusion

OJB all in all is a simple to use O/R tool and provides mechanism for nearly all database functionality. The use of OJB is recommendable for rapid development of Web Applications within the J2EE environment because it is less complex than EJB. It is independent from the underlying RDBMS and supports a three-tier-architecture. It keeps the logic of the application server independent from the database server. The development time is decreased due to the simplicity of OJB. Also the easiness of building queries supports the usability of OJB. You do not have to have much knowledge in programming SQL to use OJB.

Another advantage of OJB is the persistency. Persistent layers are needed for overcoming the problem of impedance that cause high costs in development and maintenance. The persistence layer supports the access on the database, provides the programmer to concentrate on the real problem without knowing the database schema in detail. Simple changes of the database don't need changes of the application's source code and the source code size will be reduced.

Transparent persistency supports the possibility to provide persistence without implementing a special interface or a basis table.

Addition

Attachments

JDBC Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
BOOLEAN	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	Clob
BLOB	Blob
ARRAY	Array
DISTINCT	mapping of underlying type
STRUCT	Struct

Attachment 1: Standard Field Conversion

Independent research project – OJB

ODMG 3.0	JDO
Object Model	
Transparent Persistence	Transparent Persistence
Persistence by Reachability	Persistence by Reachability
Java, C++, and Smalltalk Language Bindings on top of ODMG object model	Tight integration with Java object model
Vendor specific interfaces	PersistenceCapable interface
Collection factory for five basic collections	Collection factory “by Example”, Second Class Objects
Object identity managed by database system	Three identity models
Life Cycle	
Objects live until transaction completes	Objects can live as long as PersistenceManager
Objects can not be accessed after transaction completes	API allows the setting of properties
Callbacks not defined	Callbacks for load, store, delete, and clear
Databases and Transactions	
Easy to use but proprietary database and transaction management	Databases, connections and transactions fit into other Java APIs
No distributed transactions defined	Support for distributed transactions
Vendor specific extensions for managed environments (EJB)	Support for managed environments (EJB)
Vendor specific extension for optimistic locking	Optimistic locking is optional API
Explicit locking of objects	No explicit locking -- locks set automatically
Query	
OQL: an extensive object query language	Queries based on Java programming language -- JDOQL
Complex query expressions	Simple filter strings
Results can be compositions, projections or just an integer	Results are always collections of persistence capable objects
Class name is part of query string	Class objects explicitly used as parameters
No imports or name scope	Imports, "this", named parameters

Attachment 2: Comparison JDO and ODMG

Bibliographies

Internet

- <http://db.apache.org/ojb>
- http://www.jdocentral.com/JDO_Commentary_DavidJordan_4.html
- http://www.service-architecture.com/database/articles/detailed_comparison_of_odmg_3_0_and_jdo.html
- http://www.lots.ch/dcs/div/slides_2004.1.R.08.pdf