Advanced Technique

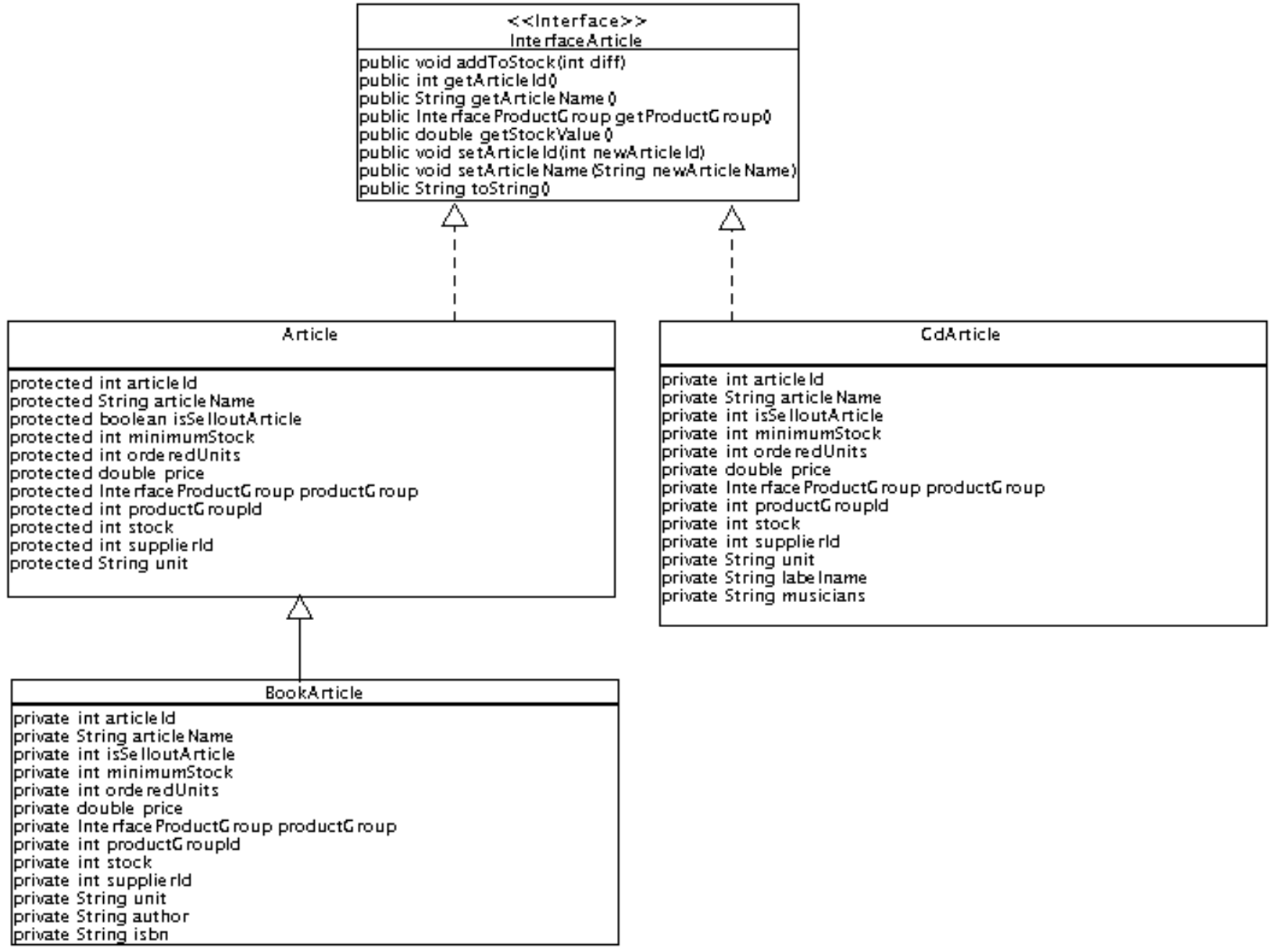
by Thomas Mahler, Jakob Braeuchli, Armin Waibel

1. Introduction

2. Extents and Polymorphism

Working with inheritance hierarchies is a common task in object oriented design and programming. Of course, any serious Java O/R tool must support inheritance and interfaces for persistent classes. To demonstrate we will look at some of the JUnit TestSuite classes.

There is a primary interface "InterfaceArticle". This interface is implemented by "Article" and "CdArticle". There is also a class "BookArticle" derived from "Article". (See the following class diagram for details)



polymorphism.gif

2.1. Polymorphism

OJB allows us to use interfaces, abstract, or concrete base classes in queries, or in type definitions of reference attributes. A Query against the interface `InterfaceArticle` must not only return objects of type `Article` but also of `CdArticle` and `BookArticle`! The following test method searches for all objects implementing `InterfaceArticle` with an `articleName` equal to "Hamlet". The Collection is filled with one matching `BookArticle` object.

```
public void testCollectionByQuery() throws Exception
{
    Criteria crit = new Criteria();
    crit.addEqualTo("articleName", "Hamlet");
    Query q = QueryFactory.newQuery(InterfaceArticle.class, crit);

    Collection result = broker.getCollectionByQuery(q);

    System.out.println(result);
```

```
    assertNotNull("should return at least one item", result);
    assertTrue("should return at least one item", result.size() > 0);
}
```

Of course it is also possible to define reference attributes of an interface or baseclass type. In all above examples Article has a reference attribute of type `InterfaceProductGroup`.

2.2. Extents

The query in the last example returned just one object. Now, imagine a query against the InterfaceArticle interface with no selecting criteria. OJB returns all the objects implementing InterfaceArticle. I.e. All Articles, BookArticles and CdArticles. The following method prints out the collection of all InterfaceArticle objects:

```
public void testExtentByQuery() throws Exception
{
    // no criteria signals to omit a WHERE clause
    Query q = QueryFactory.newQuery(InterfaceArticle.class, null);
    Collection result = broker.getCollectionByQuery(q);

    System.out.println(
        "OJB proudly presents: The InterfaceArticle Extent\n" +result);

    assertNotNull("should return at least one item", result);
    assertTrue("should return at least one item", result.size() > 0);
}
```

The set of all instances of a class (whether living in memory or stored in a persistent medium) is called an **Extent** in ODMG and JDO terminology. OJB extends this notion slightly, as all objects implementing a given interface are regarded as members of the interface's extent.

In our class diagram we find:

1. two simple "one-class-only" extents, BookArticle and CdArticle.
2. A compound extent Article containing all Article and BookArticle instances.
3. An interface extent containing all Article, BookArticle and CdArticle instances.

There is no extra coding necessary to define extents, but they have to be declared in the repository file. The classes from the above example require the following declarations:

1. "one-class-only" extents require no declaration
2. A declaration for the baseclass Article, defining which classes are subclasses of Article:

```
<!-- Definitions for org.apache.ojb.ojb.broker.Article -->
   <class-descriptor
      class="org.apache.ojb.broker.Article"
      proxy="dynamic"
      table="Artikel"
   >
      <extent-class class-ref="org.apache.ojb.broker.BookArticle" />
      <extent-class class-ref="org.apache.ojb.broker.CdArticle" />
   ...
   </class-descriptor>
```

1. A declaration for InterfaceArticle, defining which classes implement this interface:

```
<!-- Definitions for org.apache.ojb.broker.InterfaceArticle -->
   <class-descriptor class="org.apache.ojb.broker.InterfaceArticle">
      <extent-class class-ref="org.apache.ojb.broker.Article" />
      <extent-class class-ref="org.apache.ojb.broker.BookArticle" />
      <extent-class class-ref="org.apache.ojb.broker.CdArticle" />
   </class-descriptor>
```

Why is it necessary to explicitely declare which classes implement an interface and which classes are derived from a baseclass? Of course it is quite simple in Java to check whether a class implements a given interface or extends some other class. But sometimes it may not be appropiate to treat special implementors (e.g. proxies) as proper implementors.

Other problems might arise because a class may implement multiple interfaces, but is only allowed to be regarded as member of one extent.

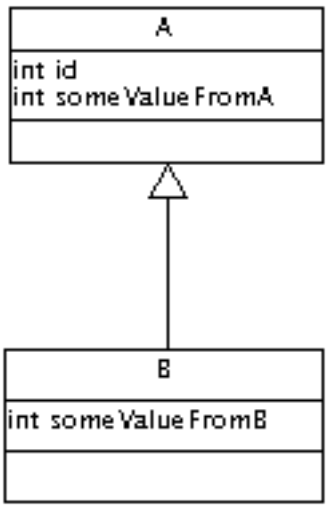
In other cases it may be neccessary to treat certain classes as implementors of an interface or as derived from a base even if they are not.

As an example, you will find that the ClassDescriptor for class org.apache.ojb.broker.Article in the repository.xml contains an entry declaring class CdArticle as a derived class:

```
<!-- Definitions for org.apache.ojb.ojb.broker.Article -->
   <class-descriptor
      class="org.apache.ojb.broker.Article"
      proxy="dynamic"
      table="Artikel"
   >
      <extent-class class-ref="org.apache.ojb.broker.BookArticle" />
      <extent-class class-ref="org.apache.ojb.broker.CdArticle" />
      ...
   </class-descriptor>
```

3. Mapping Inheritance Hierarchies

In the literature on object/relational mapping the problem of mapping inheritance hierarchies to RDBMS has been widely covered. Have a look at the following inheritance hierarchy:



inheritance-1.gif

If we have to define database tables that have to contain these classes we have to choose one of the following solutions:

1. Map all classes onto one table. A DDL for the table would look like:

```
CREATE TABLE A_EXTENT
(
    ID                  INT NOT NULL PRIMARY KEY,
    SOME_VALUE_FROM_A   INT,
    SOME_VALUE_FROM_B   INT
)
```

2. Map each class to a distinct table and have all attributes from the base class in the derived class. DDL for the table could look like:

```
CREATE TABLE A
(
    ID                  INT NOT NULL PRIMARY KEY,
```

```
    SOME_VALUE_FROM_A  INT
)
CREATE TABLE B
(
    ID                 INT NOT NULL PRIMARY KEY,
    SOME_VALUE_FROM_A  INT,
    SOME_VALUE_FROM_B  INT
)
```

3. Map each class to a distinct table, but do not map base class fields to derived classes. Use joins to materialize over all tables to materialize objects. DDL for the table would look like:

```
CREATE TABLE A
(
    ID                 INT NOT NULL PRIMARY KEY,
    SOME_VALUE_FROM_A  INT
)
CREATE TABLE B
(
    A_ID               INT NOT NULL,
    SOME_VALUE_FROM_B  INT
)
```

OJB provides direct support for all three approaches.

Note:
But it's currently not recommended to mix mapping strategies within the same hierarchy !

In the following we demonstrate how these mapping approaches can be implemented by using OJB.

3.1. Mapping All Classes on the Same Table

Mapping several classes on one table works well under OJB. There is only one special situation that needs some attention:

Say there is a baseclass AB with derived classes A and B. A and B are mapped on a table AB_TABLE. Storing A and B objects to this table works fine. But now consider a Query against the baseclass AB. How can the correct type of the stored objects be determined?

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 `ojbConcreteClass`. On loading objects from the table OJB checks this attribute and instantiates objects of this type.

Note:
The criterion for `ojbConcreteClass` is statically added to the query in class `QueryFactory` and it therefore appears in the select-statement for each extent. This means that mixing mapping strategies should be avoided.

There is sample code for this feature in the method `org.apache.ojb.broker.PersistenceBrokerTest.testMappingToOneTable()`. See the mapping details in the following Class declaration and the respective mapping:

```
public abstract class AB
{
    /** the special attribute telling OJB the object's concrete type.
     *  NOTE: this attribute MUST be called ojbConcreteClass
     */
    protected String ojbConcreteClass;

}

public class A extends AB
{
```

```
    int id;
    int someValue;

    public A()
    {
        // OJB must know the type of this object
        ojbConcreteClass = A.class.getName();
    }
}

<!-- Definitions for extent org.apache.ojb.broker.AB -->
   <class-descriptor class="org.apache.ojb.broker.AB">
      <extent-class class-ref="org.apache.ojb.broker.A" />
      <extent-class class-ref="org.apache.ojb.broker.B" />
   </class-descriptor>

<!-- Definitions for org.apache.ojb.broker.A -->
   <class-descriptor
      class="org.apache.ojb.broker.A"
      table="AB_TABLE"
   >
      <field-descriptor
         name="id"
         column="ID"
         jdbc-type="INTEGER"
         primarykey="true"
         autoincrement="true"
      />
      <field-descriptor
         name="ojbConcreteClass"
         column="CLASS_NAME"
         jdbc-type="VARCHAR"
      />
      <field-descriptor
         name="someValue"
         column="VALUE_"
         jdbc-type="INTEGER"
      />
   </class-descriptor>
```

The column CLASS_NAME is used to store the concrete type of each object.

If you cannot provide such an additional column, but need to use some other means of indicating the type of each object you will require some additional programming:

You have to derive a Class from `org.apache.ojb.broker.accesslayer.RowReaderDefaultImpl` and overridee the method `RowReaderDefaultImpl.selectClassDescriptor()` to implement your specific type selection mechanism. The code of the default implementation looks like follows:

```
protected ClassDescriptor selectClassDescriptor(Map row)
                                        throws PersistenceBrokerException
{
    // check if there is an attribute which tells us
    // which concrete class is to be instantiated
    FieldDescriptor concreteClassFD =
        m_cld.getFieldDescriptorByName(
            ClassDescriptor.OJB_CONCRETE_CLASS);

    if (concreteClassFD == null)
        return m_cld;
    else
    {
        try
        {
            String concreteClass = (String) row.get(
                                    concreteClassFD.getColumnName());
            if (concreteClass == null ||
```

```
                                    concreteClass.trim().length() == 0)
            {
                throw new PersistenceBrokerException(
                    "ojbConcreteClass field returned null or 0-length string");
            }
            else
            {
                concreteClass = concreteClass.trim();
            }
            ClassDescriptor result = m_cld.getRepository().
                                        getDescriptorFor(concreteClass);
            if (result == null)
            {
                result = m_cld;
            }
            return result;
        }
        catch (PBFactoryException e)
        {
            throw new PersistenceBrokerException(e);
        }
    }
}
```

After implementing this class you must edit the ClassDescriptor for the respective class in the XML repository to specify the usage of your RowReader Implementation:

```
<class-descriptor
      class="my.Object"
      table="MY_OBJECT"
      ...
      row-reader="my.own.RowReaderImpl"
      ...
>
...
```

You will learn more about RowReaders in the next section.

3.2. Mapping Each Class to a Distinct Table

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

3.3. Mapping Classes on Multiple Joined Tables

Here are the definitions for the classes A and B:

```
public class A
{
    // primary key
    int id;
    // mapped to a column in A_TABLE
    int someValueFromA;
}

public class B extends A
{
    // id is primary key and serves also as foreign key referencing A.id
    int id;
    // mapped to a column in B_TABLE
    int someValueFromB;
}
```

The next code block contains the class-descriptors for the the classes A and B.

```
<!-- Definitions for org.apache.ojb.broker.A -->
<class-descriptor
    class="org.apache.ojb.broker.A"
    table="A_TABLE"
>
    <field-descriptor
        name="id"
        column="ID"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
    />
    <field-descriptor
        name="someValueFromA"
        column="VALUE_"
        jdbc-type="INTEGER"
    />
</class-descriptor>

<class-descriptor
    class="org.apache.ojb.broker.B"
    table="B_TABLE"
>
    <field-descriptor
        name="id"
        column="ID"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
    />

    <field-descriptor
        name="someValueFromB"
        column="VALUE_"
        jdbc-type="INTEGER"
    />

    <reference-descriptor name="super"
        class-ref="org.apache.ojb.broker.A"
        auto-retrieve="true"
        auto-update="true"
        auto-delete="true"
    >
        <foreignkey field-ref="id"/>
    </reference-descriptor>
</class-descriptor>
```

As you can see from this mapping we need a special reference-descriptor that advises OJB to load the values for the inherited attributes from class `A` by a JOIN using the `(B.id == A.id)` foreign key reference.

The `name="super"` is not used to address an actual attribute of the class `B` but as a marker keyword defining the JOIN to the baseclass.

Auto-update must be **true** to force insertion of A when inserting B. So have to define a *auto-update* true setting for this reference-descriptor! In most cases it's also useful to enable *auto-delete*.

Note:

Be aware that this sample does not declare `org.apache.ojb.broker.B` to be an extent of `org.apache.ojb.broker.A`. Using extents here will lead to problems (instatiating the wrong class) because the primary key is not unique within the hiearchy defined in the repository.

Attributes from the super-class A can be used the same way as attributes of B when querying for B. No path-expression is needed in this case:

```
Criteria c = new Criteria();
c.addEqualTo("someValueFromA", new Integer(1));
```

```
c.addEqualTo("someValueFromB", new Integer(2));
Query q = QueryFactory.newQuery(B.class, c);
broker.getCollectionByQuery(q);
```

The above example is based on the assumption that the primary key attribute `B.id` and its underlying column `B_TABLE.ID` is also used as the foreign key attribute.

Now let us consider a case where `B_TABLE` contains an additional foreign key column `B_TABLE.A_ID` referencing `A_TABLE.ID`. In this case the layout for class `B` could look like follows:

```
public class B extends A
{
    // id is the primary key
    int id;

    // aID is the foreign key referencing A.id
    int aID;

    // mapped to a column in B_TABLE
    int someValueFromB;
}
```

The mapping for `B` will then look like follows:

```
<class-descriptor
  class="org.apache.ojb.broker.B"
  table="B_TABLE"
>
  <field-descriptor
     name="id"
     column="ID"
     jdbc-type="INTEGER"
     primarykey="true"
     autoincrement="true"
  />

  <field-descriptor
     name="aID"
     column="A_ID"
     jdbc-type="INTEGER"
  />

  <field-descriptor
     name="someValueFromB"
     column="VALUE_"
     jdbc-type="INTEGER"
  />

  <reference-descriptor name="super"
       class-ref="org.apache.ojb.broker.A">
     <foreignkey field-ref="aID" />
  </reference-descriptor>
</class-descriptor>
```

The mapping now contains an additional field-descriptor for the `aID` attribute.

In the `"super"` reference-descriptor the foreignkey `field-ref` attribute had to be changed to `"aID"`.

It is also possible to have the extra foreign key column `B_TABLE.A_ID` but without having a foreign key attribute in class `B`:

```
public class B extends A
{
    // id is the primary key
    int id;

    // mapped to a column in B_TABLE
    int someValueFromB;
```

```
}
```

We can use OJB's anonymous field feature to get everything working without the `"aID"` attribute. We keep the field-descriptor for aID, but declare it as an anonymous field. We just have to add an attribute `access="anonymous"` to the field-descriptor:

```
<class-descriptor
  class="org.apache.ojb.broker.B"
  table="B_TABLE"
>
  <field-descriptor
     name="id"
     column="ID"
     jdbc-type="INTEGER"
     primarykey="true"
     autoincrement="true"
  />

  <field-descriptor
     name="aID"
     column="A_ID"
     jdbc-type="INTEGER"
     access="anonymous"
  />

  <field-descriptor
     name="someValueFromB"
     column="VALUE_"
     jdbc-type="INTEGER"
  />

  <reference-descriptor name="super"
       class-ref="org.apache.ojb.broker.A">
     <foreignkey field-ref="aID" />
  </reference-descriptor>

</class-descriptor>
```

You can learn more about the anonymous fields feature in this howto and how it work here.

4. Using interfaces with OJB

Sometimes you may want to declare class descriptors for interfaces rather than for concrete classes. With OJB this is no problem, but there are a couple of things to be aware of, which are detailed in this section.

Consider this example hierarchy :

```
public interface A
{
    String getDesc();
}

public class B implements A
{
    /** primary key */
    private Integer id;
    /** sample attribute */
    private String desc;

    public String getDesc()
    {
        return desc;
    }
    public void setDesc(String desc)
    {
        this.desc = desc;
```

```
    }
}

public class C
{
    /** primary key */
    private Integer id;
    /** foreign key */
    private Integer aId;
    /** reference */
    private A obj;

    public void test()
    {
        String desc = obj.getDesc();
    }
}
```

Here, class C references the interface A rather than B. In order to make this work with OJB, four things must be done:

- All features common to all implementations of A are declared in the class descriptor of A. This includes references (with their foreignkeys) and collections.
- Since interfaces cannot have instance fields, it is necessary to use bean properties instead. This means that for every field (including collection fields), there must be accessors (a get method and, if the field is not marked as `access="readonly"`, a set method) declared in the interface.
- Since we're using bean properties, the appropriate `org.apache.ojb.broker.metadata.fieldaccess.PersistentField` implementation must be used (see below). This class is used by OJB to access the fields when storing/loading objects. Per default, OJB uses a direct access implementation (`org.apache.ojb.broker.metadata.fieldaccess.PersistentFieldDirectAccessImpl`) which requires actual fields to be present.
  In our case, we need an implementation that rather uses the accessor methods. Since the `PersistentField` setting is (currently) global, you have to check whether there are accessors defined for every field in the metadata. If yes, then you can use the `org.apache.ojb.broker.metadata.fieldaccess.PersistentFieldIntrospectorImpl`, otherwise you'll have to resort to the `org.apache.ojb.broker.metadata.fieldaccess.PersistentFieldAutoProxyImpl`, which determines for every field what type of field it is and then uses the appropriate strategy.
- If at some place OJB has to create an object of the interface, say as the result type of a query, then you have to specify `factory-class` and `factory-method` for the interface. OJB then uses the specified class and (static) method to create an uninitialized instance of the interface.

In our example, this would result in:

```
public interface A
{
    void setId(Integer id);
    Integer getId();
    void setDesc(String desc);
    String getDesc();
}

public class B implements A
{
    /** primary key */
    private Integer id;
    /** sample attribute */
    private String desc;

    public String getId()
    {
        return id;
    }
    public void setId(Integer id)
```

```
    {
        this.id = id;
    }
    public String getDesc()
    {
        return desc;
    }
    public void setDesc(String desc)
    {
        this.desc = desc;
    }
}

public class C
{
    /** primary key */
    private Integer id;
    /** foreign key */
    private Integer aId;
    /** reference */
    private A obj;

    public void test()
    {
        String desc = obj.getDesc();
    }
}

public class AFactory
{
    public static A createA()
    {
        return new B();
    }
}
```

The class descriptors would look like:

```
<class-descriptor
    class="A"
    table="A_TABLE"
    factory-class="AFactory"
    factory-method="createA"
>
    <extent-class class-ref="B"/>
    <field-descriptor
        name="id"
        column="ID"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
    />
    <field-descriptor
        name="desc"
        column="DESC"
        jdbc-type="VARCHAR"
        length="100"
    />
</class-descriptor>

<class-descriptor
    class="B"
    table="B_TABLE"
>
    <field-descriptor
        name="id"
        column="ID"
        jdbc-type="INTEGER"
        primarykey="true"
```

```
        autoincrement="true"
    />
    <field-descriptor
        name="desc"
        column="DESC"
        jdbc-type="VARCHAR"
        length="100"
    />
</class-descriptor>

<class-descriptor
    class="C"
    table="C_TABLE"
>
    <field-descriptor
        name="id"
        column="ID"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
    />
    <field-descriptor
        name="aId"
        column="A_ID"
        jdbc-type="INTEGER"
    />
    <reference-descriptor name="obj"
        class-ref="A">
        <foreignkey field-ref="aId" />
    </reference-descriptor>
</class-descriptor>
```

One scenario where you might run into problems is the use of interfaces for nested objects. In the above example, we could construct such a scenario if we remove the descriptors for `A` and `B`, as well as the foreign key field `aId` from class `C` and change its class descriptor to:

```
<class-descriptor
    class="C"
    table="C_TABLE"
>
    <field-descriptor
        name="id"
        column="ID"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
    />
    <field-descriptor
        name="obj::desc"
        column="DESC"
        jdbc-type="VARCHAR"
        length="100"
    />
</class-descriptor>
```

The access to `desc` will work because of the usage of bean properties, but you will get into trouble when using dynamic proxies for `C`. Upon materializing an object of type `C`, OJB will try to create the instance for the field `obj` which is of type `A`. Of course, this is an interface but OJB won't check whether there is class descriptor for the type of `obj` (in fact there does not have to be one, and usually there isn't) because `obj` is not defined as a reference. As a result, OJB tries to instantiate an interface, which of course fails.
Currently, the only way to handle this is to write a custom invocation handler that knows how to create an object of type `A`.

5. Change PersistentField Class

OJB supports a pluggable strategy to read and set the persistent attributes in the persistence capable classes. All strategy

implementation classes have to implement the interface `org.apache.ojb.broker.metadata.fieldaccess.PersistentField`. OJB provide a few implementation classes which can be set in OJB.properties file:

```
# The PersistentFieldClass property defines the implementation class
# for PersistentField attributes used in the OJB MetaData layer.
# By default the best performing attribute/refection based implementation
# is selected (PersistentFieldDirectAccessImpl).
#
# - PersistentFieldDirectAccessImpl
#   is a high-speed version of the access strategies.
#   It does not cooperate with an AccessController,
#   but accesses the fields directly. Persistent
#   attributes don't need getters and setters
#   and don't have to be declared public or protected
# - PersistentFieldPrivilegedImpl
#   Same as above, but does cooperate with AccessController and do not
#   suppress the java language access check.
# - PersistentFieldIntrospectorImpl
#   uses JavaBeans compliant calls only to access persistent attributes.
#   No Reflection is needed. But for each attribute xxx there must be
#   public getXxx() and setXxx() methods.
# - PersistentFieldDynaBeanAccessImpl
#   implementation used to access a property from a
#   org.apache.commons.beanutils.DynaBean.
# - PersistentFieldAutoProxyImpl
#   for each field determines upon first access how to access this particular field
#   (directly, as a bean, as a dyna bean) and then uses that strategy
#
PersistentFieldClass=org.apache.ojb.broker.metadata.fieldaccess.PersistentFieldDirectAccessImpl
#PersistentFieldClass=org.apache.ojb.broker.metadata.fieldaccess.PersistentFieldPrivilegedImpl
#PersistentFieldClass=org.apache.ojb.broker.metadata.fieldaccess.PersistentFieldIntrospectorImpl
#PersistentFieldClass=org.apache.ojb.broker.metadata.fieldaccess.PersistentFieldDynaBeanAccessImpl
#PersistentFieldClass=org.apache.ojb.broker.metadata.fieldaccess.PersistentFieldAutoProxyImpl
#
```

E.g. if the PersistentFieldDirectAccessImpl is used there must be an attribute in the persistent class with this name, if the PersistentFieldIntrospectorImpl is used there must be a JavaBeans compliant property of this name. More info about the individual implementation can be found in javadoc.

6. How do anonymous keys work?

To play for safety it is mandatory to understand how this feature is working. In the HOWTO section is detailed described how to use anoymous keys.

All involved classes can be found in `org.apache.ojb.broker.metadata.fieldaccess` package. The classes used for *anonymous keys* start with a `AnonymousXYZ.java` prefix.
Main class used for provide anonymous keys is `org.apache.ojb.broker.metadata.fieldaccess.AnonymousPersistentField`. Current implementation use an object identity based weak HashMap. The persistent object identity is used as key for the anonymous key value. The (Anonymous)PersistentField instance is associated with the *FieldDescriptor* declared in the repository.

This means that all anonymous key information will be lost when the object identity change, e.g. the persistent object will be de-/serialized or copied. In conjuction with 1:1 references this will be no problem, because OJB can use the referenced object to re-create the anonymous key information (FK to referenced object).

Warning:
The use of anonymous keys in 1:n references (FK to main object) or for PK fields is only valid when object identity does not change, e.g. use in single JVM without persistent object serialization and without persistent object copying.

7. Using Rowreader

RowReaders provide a callback mechanism that allows to interact with the OJB load mechanism. All implementation classes have to implement interface `RowReader`.

You can specify the `RowReader` implementation in

- the `OJB.properties` file to set the standard used `RowReader` implementation

```
#-------------------------------------------------------------------------------
# RowReader
#-------------------------------------------------------------------------------
# Set the standard RowReader implementation. It is also possible to specify the
# RowReader on class-descriptor level.
RowReaderDefaultClass=org.apache.ojb.broker.accesslayer.RowReaderDefaultImpl
```

- within the class-descriptor to set the `RowReader` for a specific class.

RowReader setting on *class-descriptor* level will override the standard reader set in `OJB.properties` file. If neither a RowReader was set in OJB.properties file nor in class-descriptor was set, OJB use an default implementation.

To understand how to use them we must know some of the details of the load mechanism. To materialize objects from a rdbms OJB uses RsIterators, that are essentially wrappers to JDBC ResultSets. RsIterators are constructed from queries against the Database.

The method `RsIterator.next()` is used to materialize the next object from the underlying ResultSet. This method first checks if the underlying ResultSet is not yet exhausted and then delegates the construction of an Object from the current ResultSet row to the method `getObjectFromResultSet()`:

```
protected Object getObjectFromResultSet() throws PersistenceBrokerException
{
    if (getItemProxyClass() != null)
    {
        // provide m_row with primary key data of current row
        getQueryObject().getClassDescriptor().getRowReader()
                    .readPkValuesFrom(getRsAndStmt().m_rs, getRow());
        // assert: m_row is filled with primary key values from db
        return getProxyFromResultSet();
    }
    else
    {
        // 0. provide m_row with data of current row
        getQueryObject().getClassDescriptor().getRowReader()
                    .readObjectArrayFrom(getRsAndStmt().m_rs, getRow());
        // assert: m_row is filled from db

        // 1.read Identity
        Identity oid = getIdentityFromResultSet();
        Object result = null;

        // 2. check if Object is in cache. if so return cached version.
        result = getCache().lookup(oid);
        if (result == null)
        {
            // 3. If Object is not in cache
            // materialize Object with primitive attributes filled from
            // current row
            result = getQueryObject().getClassDescriptor()
                                     .getRowReader().readObjectFrom(getRow());
            // result may still be null!
            if (result != null)
            {
                synchronized (result)
                {
                    getCache().enableMaterializationCache();
```

```
                    getCache().cache(oid, result);
                    // fill reference and collection attributes
                    ClassDescriptor cld = getQueryObject().getClassDescriptor()
                            .getRepository().getDescriptorFor(result.getClass());
                    // don't force loading of reference
                    final boolean unforced = false;
                    // Maps ReferenceDescriptors to HashSets of owners
                    getBroker().getReferenceBroker().retrieveReferences(result, cld, unforced);
                    getBroker().getReferenceBroker().retrieveCollections(result, cld, unforced);
                    getCache().disableMaterializationCache();
                }
            }
        }
        else // Object is in cache
        {
            ClassDescriptor cld = getQueryObject().getClassDescriptor()
                    .getRepository().getDescriptorFor(result.getClass());
            // if refresh is required, update the cache instance from the db
            if (cld.isAlwaysRefresh())
            {
                getQueryObject().getClassDescriptor()
                                .getRowReader().refreshObject(result, getRow());
            }
            getBroker().refreshRelationships(result, cld);
        }
        return result;
    }
}
```

This method first uses a RowReader to instantiate a new object array and to fill it with primitive attributes from the current ResultSet row.
The RowReader to be used for a Class can be configured in the XML repository with the attribute `row-reader`. If no RowReader is specified, the standard `RowReader` is used. The method `readObjectArrayFrom(...)` of this class looks like follows:

```
public void readObjectArrayFrom(ResultSet rs, ClassDescriptor cld, Map row)
{
    try
    {
        Collection fields = cld.getRepository().
                        getFieldDescriptorsForMultiMappedTable(cld);
        Iterator it = fields.iterator();
        while (it.hasNext())
        {
            FieldDescriptor fmd = (FieldDescriptor) it.next();
            FieldConversion conversion = fmd.getFieldConversion();
            Object val = JdbcAccess.getObjectFromColumn(rs, fmd);
            row.put(fmd.getColumnName() , conversion.sqlToJava(val));
        }
    }
    catch (SQLException t)
    {
        throw new PersistenceBrokerException("Error reading from result set",t);
    }
}
```

In the second step OJB checks if there is already a cached version of the object to materialize. If so the cached instance is returned. If not, the object is fully materialized by first reading in primary attributes with the RowReader method `readObjectFrom(Map row, ClassDescriptor descriptor)` and in a second step by retrieving reference- and collection-attributes. The fully materilized Object is then returned.

```
public Object readObjectFrom(Map row, ClassDescriptor descriptor)
                    throws PersistenceBrokerException
{
    // allow to select a specific classdescriptor
    ClassDescriptor cld = selectClassDescriptor(row, descriptor);
```

```
    return buildWithReflection(cld, row);
}
```

By implementing your own RowReader you can hook into the OJB materialization process and provide additional features.

7.1. Rowreader Example

Assume that for some reason we do not want to map a 1:1 association with a foreign key relationship to a different database table but read the associated object 'inline' from some columns of the master object's table. This approach is also called 'nested objects'. The section nested objects contains a different and much simpler approach to implement nested fields.

The class `org.apache.ojb.broker.ArticleWithStockDetail` has a `stockDetail` attribute, holding a reference to a `StockDetail` object. The class StockDetail is not declared in the XML repository. Thus OJB is not able to fill this attribute by ordinary mapping techniques.

We have to define a RowReader that does the proper initialization. The Class `org.apache.ojb.broker.RowReaderTestImpl` extends the RowReaderDefaultImpl and overrides the `readObjectFrom(...)` method as follows:

```
public Object readObjectFrom(Map row, ClassDescriptor cld)
{
    Object result = super.readObjectFrom(row, cld);
    if (result instanceof ArticleWithStockDetail)
    {
        ArticleWithStockDetail art = (ArticleWithStockDetail) result;
        boolean sellout = art.isSelloutArticle;
        int minimum = art.minimumStock;
        int ordered = art.orderedUnits;
        int stock = art.stock;
        String unit = art.unit;
        StockDetail detail = new StockDetail(sellout, minimum,
                                    ordered, stock, unit, art);
        art.stockDetail = detail;
        return art;
    }
    else
    {
        return result;
    }
}
```

To activate this RowReader the ClassDescriptor for the class ArticleWithStockDetail contains the following entry:

```
<class-descriptor
  class="org.apache.ojb.broker.ArticleWithStockDetail"
  table="Artikel"
  row-reader="org.apache.ojb.broker.RowReaderTestImpl"
>
```

8. Nested Objects

In the last section we discussed the usage of a user written RowReader to implement nested objects. This approach has several disadvantages.

1. It is necessary to write code and to have some understanding of OJB internals.
2. The user must take care that all nested fields are written back to the database on store.

This section shows that nested objects can be implemented without writing code, and without any further trouble just by a few settings in the repository.xml file.

The class `org.apache.ojb.broker.ArticleWithNestedStockDetail` has a `stockDetail` attribute, holding a reference to a `StockDetail` object. The class StockDetail is not declared in the XML repository as a first class entity class.

```
public class ArticleWithNestedStockDetail implements java.io.Serializable
{
    /**
     * this attribute is not filled through a reference lookup
     * but with the nested fields feature
     */
    protected StockDetail stockDetail;

    ...
}
```

The *StockDetail* class has the following layout:

```
public class StockDetail implements java.io.Serializable
{
    protected boolean isSelloutArticle;

    protected int minimumStock;

    protected int orderedUnits;

    protected int stock;

    protected String unit;

    ...
}
```

Only precondition to make things work is that *StockDetail* needs a default constructor.
The nested fields semantics can simply declared by the following class- descriptor:

```
<class-descriptor
  class="org.apache.ojb.broker.ArticleWithNestedStockDetail"
  table="Artikel"
>
  <field-descriptor
     name="articleId"
     column="Artikel_Nr"
     jdbc-type="INTEGER"
     primarykey="true"
     autoincrement="true"
  />
  <field-descriptor
     name="articleName"
     column="Artikelname"
     jdbc-type="VARCHAR"
  />
  <field-descriptor
     name="supplierId"
     column="Lieferanten_Nr"
     jdbc-type="INTEGER"
  />
  <field-descriptor
     name="productGroupId"
     column="Kategorie_Nr"
     jdbc-type="INTEGER"
  />
  <field-descriptor
     name="stockDetail::unit"
     column="Liefereinheit"
     jdbc-type="VARCHAR"
  />
  <field-descriptor
     name="price"
     column="Einzelpreis"
     jdbc-type="FLOAT"
  />
  <field-descriptor
```

```
     name="stockDetail::stock"
     column="Lagerbestand"
     jdbc-type="INTEGER"
  />
  <field-descriptor
     name="stockDetail::orderedUnits"
     column="BestellteEinheiten"
     jdbc-type="INTEGER"
  />
  <field-descriptor
     name="stockDetail::minimumStock"
     column="MindestBestand"
     jdbc-type="INTEGER"
  />
  <field-descriptor
     name="stockDetail::isSelloutArticle"
     column="Auslaufartikel"
     jdbc-type="INTEGER"
     conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"
  />
</class-descriptor>
```

That's all! Just add nested fields by using `::` to specify attributes of the nested object. All aspects of storing and retrieving the nested object are managed by OJB.

9. Instance Callbacks

OJB does provide transparent persistence. That is, persistent classes do not need to implement an interface or extent a persistent baseclass.

For certain situations it may be neccesary to allow persistent instances to interact with OJB. This is supported by a simple instance callback mechanism.

The interface `org.apache.ojb.PersistenceBrokerAware` provides a set of methods that are invoked from the PersistenceBroker during operations on persistent instances:

```
public interface PersistenceBrokerAware
{
    /**
     * this method is called as the first operation within a call to
     * PersistenceBroker.store(Object pbAwareObject), if
     * the persistent object needs insert.
     */
    public void beforeInsert(PersistenceBroker broker)
                        throws PersistenceBrokerException;

    /**
     * this method is called as the last operation within a call to
     * PersistenceBroker.store(Object pbAwareObject), if
     * the persistent object needs insert.
     */
    public void afterInsert(PersistenceBroker broker)
                        throws PersistenceBrokerException;

    /**
     * this method is called as the first operation within a call to
     * PersistenceBroker.store(Object pbAwareObject), if
     * the persistent object needs update.
     */
    public void beforeUpdate(PersistenceBroker broker)
                        throws PersistenceBrokerException;

    /**
     * this method is called as the last operation within a call to
     * PersistenceBroker.store(Object pbAwareObject), if
     * the persistent object needs update.
```

```
     */
    public void afterUpdate(PersistenceBroker broker)
                        throws PersistenceBrokerException;

    /**
     * this method is called as the first operation within a call to
     * PersistenceBroker.delete(Object pbAwareObject).
     */
    public void beforeDelete(PersistenceBroker broker)
                        throws PersistenceBrokerException;

    /**
     * this method is called as the last operation within a call to
     * PersistenceBroker.delete(Object pbAwareObject).
     */
    public void afterDelete(PersistenceBroker broker)
                        throws PersistenceBrokerException;

    /**
     * this method is called as the last operation within a call to
     * PersistenceBroker.getObjectByXXX() or
     * PersistenceBroker.getCollectionByXXX().
     */
    public void afterLookup(PersistenceBroker broker)
                        throws PersistenceBrokerException;
}
```

If you want your persistent entity to perform certain operations after it has been stored by the PersistenceBroker you have to perform the following steps:

1. let your persistent entity class implement the interface `PersistenceBrokerAware`.
2. provide empty implementations for all required mthods.
3. implement the method `afterUpdate(PersistenceBroker broker)` and `afterInsert(PersistenceBroker broker)` to perform your intended logic.

In the following "for demonstration only code" you see a class `DBAutoIncremented` that does not use the OJB sequence numbering (more info here), but relies on a database specific implementation of autoincremented primary key values.
When the broker is storing such an instance the DB assigns an autoincrement value to the primary key column mapped to the attribute `m_id`. The `afterStore(PersistenceBroker broker)` instance callback is used to update the the attribute `m_id` with this value.

```
public abstract class DBAutoIncremented
                    implements PersistenceBrokerAware
{
    private static final String ID_ATTRIBUTE_NAME = "m_id";

    public void afterDelete(PersistenceBroker broker)
    {
    }

    public void afterLookup(PersistenceBroker broker)
    {
    }

    public void afterUpdate(PersistenceBroker broker)
    {
    }

    /**
     * after storing a new instance reflect the
     * autoincremented PK value
     * back into the PK attribute.
     */
    public void afterInsert(PersistenceBroker broker)
    {
        try
```

```
        {
            // remove object from cache to ensure we are retrieving a
            // copy that is in sync with the database.
            broker.removeFromCache(this);

            Class clazz = getClass();
            ClassDescriptor cld = broker.getClassDescriptor(clazz);
            PersistentField idField = cld
                        .getFieldDescriptorByName(ID_ATTRIBUTE_NAME)
                        .getPersistentField();
            // retrieve the object again with a query
            // on all non-id attributes.
            Object object =
                broker.getObjectByQuery(
                    buildQueryOnAllNonIdAttributes(clazz, cld));

            if (object == null)
            {
                throw new PersistenceBrokerException(
                    "cannot assign ID to "
                        + this
                        + " ("
                        + clazz
                        + ")"
                        + " because lookup by attributes failed");
            }

            // set id attribute with the value
            // assigned by the database.
            idField.set(this, idField.get(object));
        }
    }

    public void beforeDelete(PersistenceBroker broker)
    {
    }

    public void beforeStore(PersistenceBroker broker)
    {
    }

    /**
     * returns a query that identifies an object by all its non-
     * primary key attributes.
     * NOTE: This method is only safe, if these values are unique!
     */
    private Query buildQueryOnAllNonIdAttributes(
        Class clazz,
        ClassDescriptor cld)
    {

        // note: these are guaranteed to be in the same order
        FieldDescriptor[] fields = cld.getFieldDescriptions();
        Object[] values = cld.getAllValues(this);
        Criteria crit = new Criteria();

        for (int i = 0; i < fields.length; i++)
        {
            if (!fields[i].getAttributeName().
                            equals(ID_ATTRIBUTE_NAME))
            {
                if (values[i] == null)
                {
                    crit.addIsNull(fields[i].getAttributeName());
                }
                else
                {
                    crit.addEqualTo(fields[i].getAttributeName(),
                                                        values[i]);
```

```
                }
            }
        }
        return QueryFactory.newQuery(clazz, crit);
    }
}
```

10. Manageable Collection

In 1:n or m:n relations, OJB can handle `java.util.Collection` as well as user defined collection classes as collection attributes in persistent classes. See collection-descriptor.collection-class attribute for more information.

In order to collaborate with the OJB mechanisms these collection must provide a minimum protocol as defined by this interface `org.apache.ojb.broker.ManageableCollection`.

```
public interface ManageableCollection extends java.io.Serializable
{
    /**
     * add a single Object to the Collection. This method is used during reading Collection elements
     * from the database. Thus it is is save to cast anObject to the underlying element type of the
     * collection.
     */
    void ojbAdd(Object anObject);

    /**
     * adds a Collection to this collection. Used in reading Extents from the Database.
     * Thus it is save to cast otherCollection to this.getClass().
     */
    void ojbAddAll(ManageableCollection otherCollection);

    /**
     * returns an Iterator over all elements in the collection. Used during store and delete
Operations.
     * If the implementor does not return an iterator over ALL elements, OJB cannot store and delete
all
     * elements properly.
     */
    Iterator ojbIterator();

    /**
     * A callback method to implement 'removal-aware' (track removed objects and delete
     * them by its own) collection implementations.
     */
    public void afterStore(PersistenceBroker broker) throws PersistenceBrokerException;
}
```

The methods have a prefix "ojb" that indicates that these methods are "technical" methods, required by OJB and not to be used in business code.

In package **`org.apache.ojb.broker.util.collections`** can be found a bunch of pre-defined implementations of `org.apache.ojb.broker.ManageableCollection`.

More info about which collection class to used here.

10.1. Types Allowed for Implementing 1:n and m:n Associations

OJB supports different Collection types to implement 1:n and m:n associations. OJB detects the used type automatically, so there is no need to declare it in the repository file. There is also no additional programming required. The following types are supported:

1. `java.util.Collection, java.util.List, java.util.Vector` as in the example above. Internally OJB uses `java.util.Vector` to implement collections.
2. Arrays (see the file `ProductGroupWithArray`).

3. User-defined collections (see the file `ProductGroupWithTypedCollection`). A typical application for this approach are typed Collections.
   Here is some sample code from the Collection class `ArticleCollection`. This Collection is typed, i.e. it accepts only InterfaceArticle objects for adding and will return InterfaceArticle objects with `get(int index)`. To let OJB handle such a user-defined Collection it **must** implement the callback interface `ManageableCollection` and the typed collection class must be declared in the *collection-descriptor* using the *collection-class* attribute.
   `ManageableCollection` provides hooks that are called by OJB during object materialization, updating and deletion.

```
public class ArticleCollection implements ManageableCollection,
                                     java.io.Serializable
{
    private Vector elements;

    public ArticleCollection()
    {
        super();
        elements = new Vector();
    }

    public void add(InterfaceArticle article)
    {
        elements.add(article);
    }

    public InterfaceArticle get(int index)
    {
        return (InterfaceArticle) elements.get(index);
    }

    /**
     * add a single Object to the Collection. This method is
     * used during reading Collection elements from the
     * database. Thus it is is save to cast anObject
     * to the underlying element type of the collection.
     */
    public void ojbAdd(java.lang.Object anObject)
    {
        elements.add((InterfaceArticle) anObject);
    }

    /**
     * adds a Collection to this collection. Used in reading
     * Extents from the Database.
     * Thus it is save to cast otherCollection to this.getClass().
     */
    public void ojbAddAll(
          ojb.broker.ManageableCollection otherCollection)
    {
        elements.addAll(
          ((ArticleCollection) otherCollection).elements);
    }

    /**
     * returns an Iterator over all elements in the collection.
     * Used during store and delete Operations.
     */
    public java.util.Iterator ojbIterator()
    {
        return elements.iterator();
    }
}
```

And the collection-descriptor have to declare this class:

```
<collection-descriptor
name="allArticlesInGroup"
```

```
element-class-ref="org.apache.ojb.broker.Article"
collection-class="org.apache.ojb.broker.ArticleCollection"
auto-retrieve="true"
auto-update="false"
auto-delete="true"
>
<inverse-foreignkey field-ref="productGroupId"/>
</collection-descriptor>
```

10.2. Which collection-class type should be used?

Earlier in this section the `org.apache.ojb.broker.ManageableCollection` was introduced. Now we talk about which type to use.

By default OJB use a *removal-aware* collection implementation. These implementations (classes prefixed with *Removal...*) track removal and addition of elements.
This tracking allow the PersistenceBroker to **delete elements** from the database that have been removed from the collection before a PB.store() operation occurs.

This default behaviour is **undesired** in some cases:

- In m:n relations, e.g. between *Movie* and *Actor* class. If an Actor was removed from the Actor collection of a Movie object expected behaviour was that the Actor be removed from the indirection table, but not the Actor itself. Using a removal aware collection will remove the Actor too. In that case a simple manageable collection is recommended by set e.g. `collection-class="org.apache.ojb.broker.util.collections.ManageableArrayList"` in collection-descriptor.
- In 1:n relations when the n-side objects be removed from the collection of the main object, but we don't want to remove them itself (be careful with this, because the FK entry of the main object still exists - more info about linking here).

11. Customizing collection queries

Customizing the query used for collection retrieval allows a **developer** to take full control of collection mechanism. For example only children having a certain attribute should be loaded. This is achieved by a QueryCustomizer defined in the collection-descriptor of a relationship:

```
<collection-descriptor
    name="allArticlesInGroup"
    ...
>
    <inverse-foreignkey field-ref="productGroupId"/>

    <query-customizer
    class="org.apache.ojb.broker.accesslayer.QueryCustomizerDefaultImpl">
        <attribute
            attribute-name="attr1"
            attribute-value="value1"
        />
    </query-customizer>

</collection-descriptor>
```

The query customizer must implement the interface `org.apache.ojb.broker.accesslayer.QueryCustomizer`. This interface defines the single method below which is used to customize (or completely rebuild) the query passed as argument. The interpretation of attribute-name and attribute-value read from the collection-descriptor is up to your implementation.

```
/**
* Return a new Query based on the original Query, the
* originator object and the additional Attributes
*
* @param anObject the originator object
```

```
* @param aBroker the PersistenceBroker
* @param aCod the CollectionDescriptor
* @param aQuery the original 1:n-Query
* @return Query the customized 1:n-Query
*/
public Query customizeQuery(Object anObject,
                PersistenceBroker aBroker,
                CollectionDescriptor aCod, Query aQuery);
```

The class `org.apache.ojb.broker.accesslayer.QueryCustomizerDefaultImpl` provides a default implentation without any functionality, it simply returns the query.

12. Metadata runtime changes

This was described in metadata section.