OJB Documentation

1. OJB

1.1. ObJectRelationalBridge - OJB

1.1.1. Summary

ObJectRelationalBridge (OJB) is an Object/Relational mapping tool that allows transparent persistence for Java Objects against relational databases.

1.1.1.1. flexibility

OJB supports multiple persistence APIs to provide users with their API of choice:

- A full featured **ODMG 3.0** compliant API. (See the ODMG Tutorial for an introduction.)
- A **JDO** compliant API. We currently provide a plugin to the JDO Reference Implementation (RI). Combining the JDO RI and our plugin provides a JDO 1.0 compliant o/r solution.
  A full JDO implementation is scheduled for OJB 2.0. (See JDO tutorial for an introduction to the JDO programming model.)
- An Object Transaction Manager (OTM) layer that contains all features that JDO and ODMG have in common. (See OTM tutorial for details).
- A low-level **PersistenceBroker** API which serves as the OJB persistence kernel. The OTM-, ODMG- and JDO-implementations are build on top of this kernel.
  This API can also be used directly by applications that don't need full fledged object level transactions (See the Persistence Broker Tutorial for details).

See the FAQ for a detailed view of the OJB layering.

1.1.1.2. scalability

OJB has been designed for a large range of applications, from embedded systems to rich client application to multi-tier J2EE based architectures.

OJB integrates smoothly into J2EE Application servers. It supports JNDI lookup of datasources. It ships with full JTA and JCA Integration. OJB can be used within JSPs, Servlets and SessionBeans. OJB provides special support for Bean Managed EntityBeans (BMP).

1.1.1.3. functionality

OJB uses an XML based Object/Relational Mapping. The mapping resides in a dynamic MetaData layer which can be manipulated at runtime through a simple Meta-Object-Protocol (MOP) to change the behaviour of the persistence kernel.

OJB provides several advanced O/R features like an Object Caching, lazy materialization through virtual proxies or a distributed lock-management with configurable Transaction-Isolation Levels. Optimistic and pessimistic Locking is supported.

OJB provides a flexible configuration and plugin mechanism that allows to select from set of predefined components or to implement your own extensions and plugins.

A more complete featurelist can be found here.

Learn more about the OJB design principles in this document.

1.2. OJB - Features

1.2.1. Features

- Support of standard and non-standard API's:
  - PB api (non-standard)
  - OTM api (non-standard)
  - ODMG api (standard)
  - JDO api (standard)
- The PersistenceBroker kernel api and all top-level api (ODMG, OTM, JDO) allows Java Programmers to store and retrieve Java Objects in/from (any) JDBC-compliant RDBMS
- Transparent persistence: Persistent classes don't have to inherit from a persistent base class or to implement an interface.
- Scalable architecture that allows to build massively distributed and clustered systems.
- Configurable persistence by reachability: All Objects associated to a persistent object by references can made persitent too.
- Extremly flexible design with pluggable implementation of most service classes like *PersistenceBroker*, *ObjectCache*, *SequenceManager*, *RowReader*, *ConnectionFactory*, *ConnectionManager*, *IndirectionHandler*, *SQLGenerator*, *JdbcAccess*, ... and so on.
- Quality assurance taken seriously: More than *600 JUnit-TestCases* for regression tests. JUnit tests integrated into the build scripts.
- Mapping support for 1:1, 1:n and m:n associations.
- Configurable collection queries to control loading of relationships. See QueryCustomizer.
- Automatic and manual assignment of foreign key values.
- The Object / Relational mapping is defined in an XML Repository. The mapping is completely dynamic and can be manipulated at runtime for maximum flexibility
- Easy use of multiple databases.
- Configurable Lazy Materialization through Proxy support in the PersistenceBroker. The user can implement specific Proxy classes or let OJB generate dynamic Proxies.
- Support for Polymorphism and Extents. You can use Interface-types and abstract classes as attribute types in your persistent classes. Queries are also aware of extents: A query against a baseclass or interface will return matches from derived classes, even if they are mapped to different DB-tables
- Support for Java Array- and Collection-attributes in persistent classes. The attribute-types can be Arrays, java.util.Collection or may be user defined collections that implement the interface `ojb.broker.ManageableCollection`.
- Sequence-Managing . The SequenceManager is aware of "extents" and maintains uniqueness of ids accross any number of tables. Sequence Numbering can be declared in the mappping repository.
  Native Database based Sequence Numbering is also supported.
- Reusing Prepared Statements, internal connection pooling.
- Integrates smoothly in controlled environments like EJB containers
- Full JTA and JCA (in progress) Integration.
- Support for prefetched relationships to minimize the number of queries.
- ODMG compliant API, a Tutorial, and TestCases are included.
- JDO 1.0.1 compliant API (based on jdori, native implementation in progress), a Tutorial, and TestCases are included.
- Distributed Lockmanagement supporting four pessimistic Transaction Isolation Levels (uncommited or "dirty" reads, commited reads, repeatable reads, serializable transactions).
- Optimistic locking support. Users may declare `int` or `long` fields as version attributes or `java.sql.Timestamp` fields as timestamp attributes.
- Support of distributed caches.
- Comes along with fully functional demo applications running against HSQLDB.
- Provides Commons-Logging and Log4J logging facilities.

- 100 %: pure Java, Open Source, Apache License

> **Note:**
> - OQL is currently not fully implemented (Aggregations and Method Invocations)
> - ODMG implicit locking is partly implemented but does currently not maintain transaction isolation properly. To achieve safe transaction isolation client application must use explicit lock acquisition

1.3. Status

1.3.1. PB API (Persistence Broker API)

The PB API implementation is **stable**.

1.3.2. OTM API (Object Transaction Manager API)

The OTM Object Transaction Manager API is in beta status with this release.

1.3.3. ODMG API

The ODMG API implementation is stable, but there some known issues - see release-notes

> **Note:**
> OQL is currently not fully implemented (Aggregations and Method Invocations).
> ODMG implicit locking is partly implemented but does currently not maintain transaction isolation properly. To achieve safe transaction isolation client application must use explicit lock acquisition.

1.3.4. JDO API

By providing a plugin to the SUN JDO Reference Implementation we provide a complete JDO 1.0.1 prototype O/R mapping tool. A complete Apache licensed JDO implementation is scheduled for OJB 2.0.

1.4. OJB - Mail Lists

1.4.1. Mailing Lists

These are the mailing lists that have been established for this project. For each list, there is a subscribe, unsubscribe, and an archive link.

List Name	Subscribe	Unsubscribe	Archive
Objectbridge User List	Subscribe	Unsubscribe	Archive
Objectbridge Developer List	Subscribe	Unsubscribe	Archive

1.5. OJB - Mail Archives

1.5.1. Mail Archives

archive provider	OJB User list	OJB Developer list	searchable	remarks
GMANE	gmane.comp.jakarta.ojb.user	gmane.comp.jakarta.ojb.devel	yes	latest 600 postings available

				via web access. Unlimited access through nntp (news reader)
Apache	ojb-user@db.apache.org	ojb-dev@db.apache.org	yes	--
The Mail Archive	ojb-user	--	yes	--

1.6. OJB - References and Testimonials

1.6.1. References and Testimonials

1.6.1.1. projects using OJB

Jakarta JetSpeed
Jetspeed is an Open Source implementation of an Enterprise Information Portal, using Java and XML.
OJB will be the default persistence model within Jetspeed 2.

The Tammi project
Tammi is a JMX-based Java application development framework and run-time environment providing a service architecture for J2EE server side Internet applications that are accessible from any device that supports HTTP including mobile (wireless) handsets.
Future plans include integration of Apache OJB based persistence services to the framework.

The Object Console project
The Object Console is an open web based application meant for the administration of objects via the web. Any object that is persistable by the ObJectRelationalBridge (OJB) framework can be managed through this tool. In addition, this tool provides administration functionality for the ObJectRelationalBridge (OJB) framework itself.
Object Console uses Struts and OJB. It ships with full sourcecode and is thus a great source for learning Struts + OJB techniques.

The IntAct project
The IntAct project establishes a knowledgebase for protein-protein interaction data. It's hosted at EBI - European Bioinformatics Institute, Cambridge.
IntAct uses OJB as its persistence layer.

Network for Earthquake Engineering Simulation
The NEES program will provide an unprecedented infrastructure for research and education, consisting of networked and geographically distributed resources for experimentation, computation, model-based simulation, data management, and communication.
OJB is used as the O/R mapping layer.

The OJB.NET project
OJB.NET is an object-to-relational persistence tool for the .NET platform. It enables applications to transparently store and retrieve .NET objects using relational databases.
OJB.NET is a port ojb Apache OJB to the .NET platform

The OpenEMed project
OpenEMed is a set of distributed healthcare information service components built around the OMG distributed object specifications and the HL7 (and other) data standards and is written in Java for platform portability.
OpenEMed uses ODMG as its persistence API. OJB is used as ODMG compliant O/R tool.

1.6.1.2. user testimonials

"We're using OJB in two production applications at the Northwest Alliance for Computational Science and Engineering (NACSE). One is a data mining toolset, and the other is a massive National Science Foundation project that involves huge amounts of data, and about 20 or 25 universities and research groups like mine.
In fact, I've begun making OJB sort of a de-facto standard for NACSE java/database development. I've thrown out EJB's for the most part and I've tried JDO from Castor, but I'm sticking with OJB. Maybe we'll reconsider JDO when the OJB implementation is more complete."

"We are planning a November 2003 production deployment with OJB and WE LOVE IT!! We have been in development on a very data-centric application in the power industry for about 5 months now and OJB has undoubtedly saved us countless hours of development time. We have received benefits in the following areas:
-> Easily adapts to any data model that we've thrown at it. No problems mapping tables with compound keys, tables mapping polymorphic relationships, identity columns, etc.
-> Seemlesly switches between target DB platforms. We develop and unit test on our local workstations with HSQLDB and PostgreSQL, and deploy to DB2 using the Type 4 JDBC driver from IBM. Works great!
-> Makes querying a breeze with the PersistenceBroker API
Overall we have found OJB to be very stable (and we've really tested it out quite a bit). The only issues we've got outstanding at the moment is support for connections to multiple databases, but I've noticed in CVS that the OJB guys are already fixing this for OJB 0.9.9."

"We've been using it in "production" for a long time now, from about version 0.9.4, I believe. It has been very robust. We don't use all of its features. We've only see to failures of our persistent store in about 9 months, and I'm not sure they were due to OJB."

"So yes, we have made a quite useful mediumsized production website based on OJB (with JBoss, Jakarta Jetspeed, Jakarta Turbine and Jakarta Jelly, three Tomcats, OpenSymhony OSCache and for the database MSSQL server, all running on Win2000.) It is attracting between 600 and 9000 (peak) users a day, and runs smoothly for extended periods of time. And no, I can not actually show you the wonders of the editorial interface of the content management system, because it is hidden behind a firewall.
I feel OJB is quite useful in production, but you certainly have to know what you are doing and what you are trying to achieve with it. And there have been some tricky aspects, but these could be solved by simple workarounds and small hacks.
The main thing about OJB is that AFAIK it has an overall clean design, and it far beats making your own database abstraction layer and object/relational mapper. We certainly do not use all of it, only the Persistence Broker parts, so there was less to learn. We love the virtual proxy and collection proxy concepts, the criteria objects for building queries, and the nice little hidden features that you find when you start to learn the system."

"My Company is building medium to large scale, mission critical applications (100 - 5.000 concurrent users) for our customers. Our largest customer is KarstadtQuelle, Europes largest retail company. The next big system that will go in production (in June) is the new logistics system for the stationary logistics of Karstadt.
Of course we are using OJB in those Systems! We have several OJB based systems now in production for over a year. We never had any OJB related problems in production.
Most problems we faced during development were related to the learning curve developers had to face who were new to O/R mapping."

"I've also worked with OJB on high-load situations in J2EE environments. We're using JRun and/or Orion with OJB in a clustered/distributed environment. This is a National Science Foundation project called the Network for Earthquake Engineering Simulation (NEES).
The only major problem that we ran into was the cache. JCS just isn't good, and hasn't seemed to get much better over the last year. We ended up plugging in Tangosol's Coherence Clustered Cache into the system. We can also do write-behinds, and buffered data caching that is queued for transaction. That's important to us because we're dealing with very expensive scientific data that _can't_ get lost if a db goes down. Some of these Tsunami experiments can get pretty expensive.
Otherwise, we use mostly the PersistenceBroker, and a little of the ODMG. Performance seems better on PB, but less functional. It's not really that much of a problem anyway, because we can cheaply and quickly add app-servers to the cluster."

1.7. Links and further readings

1.7.1. Summary

This page contains interesting links and recommended readings that will help to learn more about OJB concepts, related projects, didactic material, research reports etc.

1.7.2. Design

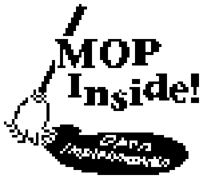
OJB is based on a variety of conceptual sources. In this section I'll give a summary about the most prominent influences.

1. **Craig Larmans Applying UML and Patterns**
2. **The Siemens Guys "Pattern-Oriented Software Architecture"**
3. **Scott Amblers classic papers on O/R mapping**
4. **The "Crossing Chasms" paper from Brown et. al.**
5. **The GOF Design Patterns**

(sorted by relevance)

1. The most important input came from Applying UML and Patterns. It contains a chapter describing the design of a PersistenceBroker based approach persistence layer. His presentation contains a lot of other good ideas (e.g. usage of Proxies, caching etc.) I implemented a lot of his things 1:1. This book is a must have for all OJB developers !

2. Larman does not cover the dynamic metadata concept. He mentiones that such a thing would be possible, but does not go into details. As I had been a fan of MetaLevel architectures for quite a while I wanted to have such a thing in OJB too !!!



mop-gif

I took the concepts from the book Pattern-Oriented Software Architecture. They have a chapter on the **Reflection** pattern (aka Open Implementation, Meta-Level Architecture).
They even provide an example how to apply this pattern to a persistence layer.
There is another Architectural pattern from this book that I am using: The **Microkernel** pattern.
My idea was to have a kernel (the PersistenceBroker) that does all the hard work (O/R mapping, JDBC access, etc.)
High Level object transaction frameworks like a ODMG or JDO implementations are clients to the PersistenceBroker kernel in this concept!

3. I read Scott Amblers papers before starting OJB. Sure! There are several things in OJB that are from his classic The design of a robust persistence layer and from his Mapping Objects To Relational Databases. Most prominent: The **PersistenceBroker** concept.
I incorporated the Query API from the OpenSource project COBRA that applies Amblers PersistentCriteria concept.
Reading Amblers paper on these topics is a must.

But IMO these are the only aspects of Amblers presentation that map directly to OJB. Here are the concepts that differ:

- Amblers concept relies on a persistent base class.
- caching is not covered by his design
- his concept of OID does not fit for legacy databases with compound PKs.
- The OJB proxy concept is quite different (Ambler has proxy functionality in his PersistentObject base class.)
- OJB does not use Insert- and UpdateCriteria
- OJB uses a different mapping approach (A full metadata layer)

4. For several detail questions (like mapping inheritance hierarchies) I consulted crossing chasms. This is also a very good source for all O/R implementors.

5. For all the "small things" I'm using the common GOF patterns like Factory, Observer, Singleton, Proxy, Adaptor, State, Command, etc.

Here is a thesis describing concepts very similar to OJB.
As I read this paper I saw a lot of thing inspired by OJB. It's giving a nice introduction into the PersistenceBroker pattern and related topics.

The PARC software design area pioneering in Metalevel computation, aspect oriented programming etc.

1.7.3. Further readings on O/R mapping

- ObjectArchitects O/R pattern page
- JavaSkyLine page on database integration
- Barry and Associates page on O/R mapping
- Portland Pattern Repository page on O/R
- Martin Fowlers book "Pattern of Enterprise Application Architecture" covers many O/R patterns that can be found in OJB. Here you will find an online catalog of these patterns.

1.7.4. Patterns

- The Hillside Pattern page
- The Portland Pattern Repository

1.7.5. OJB tutorials

- The famous Beer4All Struts/OJB tutorial by Chuck Cavaness
- A presentation on OJB held at the Atlanta Java Users Group by Chuck Cavaness
- An extensive tutorial on OJB by John Carnell
- Roberto Ghizzioli's tutorial on Struts, OJB, and nested tags
- An introductory tutorial on the O'Reilly site.

1.7.6. Books covering OJB

- The O'Reilly book on Struts programming by Chuck Cavaness has a whole chapter about how to build an applications model layers based on OJB. A must reading for everyone intending to use Struts and OJB. All source code from the book can be found here: Struts Programming sources.
- There's also a WROX book on Struts + OJB All source code from the book can be found here: Professional Struts and OJB sources.
- Enterprise Java Development on a Budget

2. Download

3. Development

3.1. Coding Standards

3.1.1. Coding Standards

This document describes a list of coding conventions that are required for code submissions to the project. By default, the coding conventions for most Open Source Projects should follow the existing coding conventions in the code that you are working on. For example, if the bracket is on the same line as the if statement, then you should write all your code to have that convention.

If you commit code that does not follow these conventions, you are responsible for also fixing your own code.

Below is a list of coding conventions that are specific to Turbine, everything else not specificially mentioned here should follow the official Sun Java Coding Conventions.

1. Brackets should begin and end on a new line and should exist even for one line statements. Examples:

```
        if ( foo )
        {
            // code here
        }

        try
        {
            // code here
        }
        catch (Exception bar)
        {
            // code here
        }
        finally
        {
            // code here
        }

        while ( true )
        {
            // code here
        }
```

2. Though it's considered okay to include spaces inside parens, the preference is to not include them. Both of the following are okay:

```
        if (foo)

        or

        if ( foo )
```

3. 4 space indent. **NO tabs**. Period. We understand that many developers like to use tabs, but the fact of the matter is that in a distributed development environment where diffs are sent to the mailing lists by both developers and the version control system (which sends commit log messages), the use tabs makes it impossible to preserve legibility.

In Emacs-speak, this translates to the following command:

```
        (setq-default tab-width 4 indent-tabs-mode nil)
```

4. Unix linefeeds for all .java source code files. Other platform specific files should have the platform specific linefeeds.

5. JavaDoc **MUST** exist on all methods. If your code modifications use an existing class/method/variable which lacks JavaDoc, it is required that you add it. This will improve the project as a whole.

6. The Jakarta/Turbine License **MUST** be placed at the top of each and every file.

7. If you contribute to a file (code or documentation), add yourself to the authors list at the top of the file. For java files the preferred Javadoc format is:

```
        @author <a href="mailto:user@domain.com">John Doe</a>
```

8. All .java files should have a @version tag like the one below.

```
        @version $Id: code-standards.xml,v 1.1 2004/06/20 09:12:35 tomdz Exp $
```

9. Import statements must be fully qualified for clarity.

```
        import java.util.ArrayList;
        import java.util.Hashtable;

        import org.apache.foo.Bar;
        import org.apache.bar.Foo;
```

And not

```
        import java.util.*;
        import org.apache.foo.*;
        import org.apache.bar.*;
```

X/Emacs users might appreciate this in their .emacs file.

```
        (defun apache-jakarta-mode ()
          "The Java mode specialization for Apache Jakarta projects."
          (if (not (assoc "apache-jakarta" c-style-alist))
              ;; Define the Apache Jakarta cc-mode style.
              (c-add-style "apache-jakarta" '("java" (indent-tabs-mode . nil))))

          (c-set-style "apache-jakarta")
          (c-set-offset 'substatement-open 0 nil)
          (setq mode-name "Apache Jakarta")

          ;; Turn on syntax highlighting when X is running.
          (if (boundp 'window-system)
              (progn (setq font-lock-support-mode 'lazy-lock-mode)
                     (font-lock-mode t))))

        ;; Activate Jakarta mode.
        (if (fboundp 'jde-mode)
            (add-hook 'jde-mode-hook 'apache-jakarta-mode)
          (add-hook 'java-mode-hook 'apache-jakarta-mode))
```

Thanks for your cooperation.

4. Documentation

4.1. Documentation

4.1.1. Introduction

This section contains all documentation about OJB (except the wiki doc).

If you're new to OJB, we recommend that you start with reading the Getting Started section and the FAQ.

There are tools for building the metadata mapping files used by OJB. Information about them can be found here.

- Tutorials
  Tutorials for the API's supported by OJB.
- Reference Guides
  OJB reference guides.
- Howto's
  Howto's provided by OJB users and committers.

- Testing
  Info about OJB's quality assurance and test writing.

4.2. Frequently Asked Questions

4.2.1. Questions

1. **General**
    - Why OJB? Why do we need another O/R mapping tool?
    - How is OJB related to ODMG and JDO?
    - What are the OJB design principals?
    - Where can I learn more about Object/Relational mapping in general?
    - How OJB performance compares to native JDBC programming?
    - How OJB performance compares to other O/R mapping tools?
    - Is OJB ready for production environments?
2. **Getting Started**
    - Help! I'm having problems installing and using OJB!
    - Help! I still have serious problems installing OJB!
    - OJB does not start?
    - Does OJB support my RDBMS?
    - What are the OJB internal tables for?
    - What does the exception *Could not borrow connection from pool* mean?
    - Any tools help to generate the metadata files?
3. **OJB api's**
    - What are the differences between the PersistenceBroker API and the ODMG API? Which one should I use in my applications?
    - I don't like OQL, can I use the PersistenceBroker Queries within ODMG?
    - The OJB JDO implementation is not finished, how can I start using OJB?
4. **Howto**
    - How to use OJB with my RDBMS?
    - What are the best settings for maximal performance?
    - How to page and sort?
    - What about performance and memory usage if thousands of objects matching a query are returned as a Collection?
    - When is it helpful to use Proxy Classes?
    - How can I convert data between RDBMS and OJB?
    - How can I trace and/or profile SQL statements executed by OJB?
    - How does OJB manage foreign keys?
    - Difference between getIteratorByQuery() and getCollectionByQuery()?
    - How can Collections of primitive typed elements be mapped?
    - How could class 'myClass' represent a collection of 'myClass' objects
    - How to lookup `PersistenceBroker` instances?
    - How to access ODMG?
    - Needed to put user/password of database connection in repository file?
    - Many different database user - How do they login?
    - How do I use multiple databases within OJB?
    - How does OJB handle connection pooling?
    - Can I directly obtain a `java.sql.Connection` within OJB?
    - Is it possible to perform my own sql-queries in OJB?
    - Start OJB without a repository file?
    - Connect to database at runtime?
    - Add new persistent objects metadata (`class-descriptor`) at runtime?
    - Global metadata changes at runtime?

4.2.2. Answers

4.2.2.1. 1. General

1.1. Why OJB? Why do we need another O/R mapping tool?

here are some outstanding OJB features:

- It's fully ODMG 3.0 compliant
- It will have a full JDO implementation
- It's higly scalable (Loadbalanced Multiserver scenario)
- It provides multiple APIs:
  - The full fledged ODMG-API,
  - The JDO API (planned)
  - and the PersistenceBroker API. This API provides a O/R persistence kernel which can be used to build higher level APIs (like the ODMG and JDO Implementations)
- It's able to handle multiple RDBMS simultaneously.
- it has a slick MetaLevel Architecture: By changing the MetaData at runtime you can change the O/R mapping behaviour. (E.G. turning on/off usage of Proxies.)
- It has a simple CacheMechanisms that is fully garbage collectable by usage of weak references.
- It has a simple and clean pattern based design.
- It uses a configurable plugin concept. This allows to replace components (e.g. the ObjectCache) by user defined Replacements.
- It has a modular architecture (you can quite easily reuse some components in your own applications if you don't want to use the whole thing:
  - The PersistenceBroker (e.g. to build your own PersistenceManager)
  - The Query Interface as an abstract query syntax
  - The OQL Parser
  - The MetaData Layer
  - The JDBC Accesslayer
- It has a very sharp focus: It's concerned with O/R mapping and nothing else.

Before making OJB an OpenSource project I had a look around at the emerging OpenSource O/R scene and was asking myself if there is really a need for yet another O/R tool. I came to the conclusion that there was a need for OJB because:

- There was no ODMG/JDO compliant opensource tool available
- There was no scalable opensource O/R tool available
- there was no tool available with the idea of a PersistenceBroker Kernel that could be easiliy extended
- The tools available had no dynamic MetaData architectures.
- The tools available were not as clearly designed as I hoped, thus extending one of them would have been very difficult.

1.2. How is OJB related to ODMG and JDO?

ODMG is a standard API for Object Persistence specified by the ODMG consortium (www.odmg.org). JDO is Sun's API specification for Object Persistence. ODMG may well be regarded as a Precursor to JDO. In fact JDO incorporates many ideas

from ODMG and several people who have been involved in the ODMG spec are now in the JDO team.
I assume JDO will have tremendous influence on OODBMS-, RDBMS-, J2EE-server and O/R-tool-vendors to provide compliant products.
OJB wants to provide first class support for JDO and ODMG APIs.

OJB currently contains of four main layers, each with its own API:

1. A low-level **PersistenceBroker** API which serves as the OJB persistence kernel. The PersistenceBroker also provides a scalable multi-server architecture that allows to used it in heavy-duty app-server scenarios.
   This API can also be used directly by applications that don't need full fledged object level transactions (see PB tutorial for details).
2. An Object Transaction Manager (OTM) layer that contains all features that JDO and ODMG have in common as Object level transactions, lock-management, instance lifecyle etc. (See OTM tutorial for details.) The OTM is work in progress.
3. A full featured **ODMG 3.0** compliant API. (See ODMG tutorial for an introduction.)
   Currently this API is implemented on top the PersistenceBroker. Once the OTM layer is finished ODMG will be implemented on top of OTM.
4. A **JDO** compliant API. This is work in progress. (See JDO tutorial for an introduction.)
   Currently this API is implemented on top the PersistenceBroker. Once the OTM layer is finished JDO will be implemented on top of OTM.

The following graphics shows the layering of these APIs. Please note that the layers coloured in yellow are not yet implemented.

OJB Layer

1.3. What are the OJB design principals?

OJB has a "pattern driven" design. Please refer to this document for more details

1.4. Where can I learn more about Object/Relational mapping in general?

We have a link list pointing to further readings.

1.5. How OJB performance compares to native JDBC programming?

See page Performance.

1.6. How OJB performance compares to other O/R mapping tools?

See page Performance.

1.7. Is OJB ready for production environments?

Depends on your production environment. If you want to program an aeroplane autopilot system you should not use Java at all. (according to the official disclaimer).
But I assume we are talking about enterprise business applications, aren't we? And for such applications it's a clear **yes**. OJB is used in production application since version 0.5. We have about 6.000 downloads each month (and growing) and a large user base using it in a wide spectrum of production scenarios.
We provide a regression test suite for Quality Assurance. You can use this testsuite to check if OJB works smoothly in your target environment. (see supported platforms documentation)
We also provide a performance testsuite that compares OJB performance against native JDBC. This test will give you an impression of the performance impact OJB will have in your target environment. (see Performance testsuite documentation)
OJB is also the persistence layer of choice in several books on programming J2EE based enterprise business systems. (see our links and references section)
Reference projects and user testimonials are listed here.

4.2.2.2. 2. Getting Started

2.1. Help! I'm having problems installing and using OJB!

Please read the Getting Started document. OJB is a powerful and complex system - installing and configuring OJB is not a trivial task. Be sure to follow **all the steps** mentioned in that document - don't skip any steps when first installing OJB on your systems.

If you are having problems running OJB against your target database, read the respective platform documentation. Before you try to deploy OJB to your environment, read the deployment guide.

2.2. Help! I still have serious problems installing OJB!

The following answer is quoted from the OJB user-list. It is from a reply to a user who had serious problems getting started with OJB.

I would say it was stupid not to understand OJB. How can you know what another programmer wrote. I've been a Java programmer for quite some time and I could show you stuff I wrote that I know you wouldn't understand. I'll just break it down the best I can on what, where and why.

OJB is a data persistence layer for Java. I'll just use an example of how I use it. I have an RDMS. I would like to save Java object states to this database and I would like to be able to search this information as well. If you serialize objects it's hard to search and if you use SQL it won't work with any different database. Plus it's a mess having to work with all that SQL in your code. And by using SQL you don't get to work with just Java objects. But, with OJB your separated from having to work outside the object world and unlike serialization you can preform SQL like searches on your data. Also, there's things like caching and connection pooling in OJB that help with performance. After setting up OJB you will use either PB-API or ODMG or JDO to access your information in a object centric manner. PB API is a non-standard O/R mapping API with many features and great flexibility. All top-level API's like ODMG or JDO build on top of the PB-api. ODMG is a standard for the api for accessing your data. That means you can use any ODMG compliant api if you don't want to use OJB. The JDO part is like ODMG except it's the SUN JDO standard. I use ODMG because the JDO interface is not ready yet.

OJB is easy to use. I'll just break it down into two sides. There's the side your writing your code for your application and there's the side that you configure to make OJB connect to your database. Starting with your application side, all that is needed is to use the interface you wish. I use ODMG because JDO is not complete yet. Here's a link to the ODMG part with some code for examples.
That's all you need on the application side. Next there's the configuration side. This is the one your fighting with. Here you need to setup the core tables for OJB and you will define the classes you wish to store in your database.

First thing to do is to build the cvs's with the default database HSQL, because you know it will work. If you get past this point you should have a working OJB compiled. Now if your using JDK 1.4 you will need to set in build.properties JDBC=+JDBC30 and do a *ant preprocess* first. Next you will do a *ant junit* and this will build OJB and test everything for you. If you get a build successful then your in business. Then you will want to run *ant jar* to create the OJB jar to put in your /lib. You will need a couple other jars in you /lib directory to make it all work. See this page for those. http://jakarta.apache.org/ojb/deployment.html

Next you will need some xml and configuration files in your class path for OJB. You will find those files under {$OJB_base_dir}/target/test/ojb. All the repository.xml's and OJB.properties for sure. With all these files in place with your application you should be ready to use OJB and start writing your application.

Finally you will want to setup your connection to your database and define your classes you will be storing in your database. In the repository.xml file you can configure your JDBC parameters so OJB can connect to your database. You will also need your JDBC jar somewhere in your class path. Then you will define your classes in the repository_user.xml file. Look here for examples. http://jakarta.apache.org/ojb/tutorial1.html Note you will want to comment out the junit part in repository.xml because it's just for testing.

The final thing to do is to make sure the OJB core tables are in your database. Look on this page for the core tables. These core tables are used by OJB to store internal data while it's running. It needs these. Then there's the tables you define. The ones you mapped in the repository_user.xml file.

Sorry if any of this is off. OJB is growing so fast that it's hard to keep up with all changes. The order I gave the steps in is just how I would think it's understood better. You can go in any order you want. The steps I've shown are mostly for deployment. Hope this helps you understand OJB a little better. I'm not sure if this is what your wanting or not.

2.3. OJB does not start?

If you carefully attended the installing hints there may be something wrong with your metadata mapping defined in the repository file or one the included sub files.

- Are you included all configuration files in classpath?
- On update to a new release, make sure you replaced all configuration files
- Check your metadata mapping - typos,... ?

If something going wrong while OJB read the metadata files you can enable *debug* log level for `org.apache.ojb.broker.metadata.RepositoryXmlHandler` and `org.apache.ojb.broker.metadata.ConnectionDescriptorXmlHandler` to get more detailed information.

Note:
If OJB default logging was used, change entries for these classes in OJB.properties file (this may change in future).

2.4. Does OJB support my RDBMS?

please refer to this document.

2.5. What are the OJB internal tables for?

Please refer to this document.

2.6. What does the exception Could not borrow connection from pool mean?

There can be several reasons

2.7. Any tools help to generate the metadata files?

Please refer to this document.

4.2.2.3. 3. OJB api's

3.1. What are the differences between the PersistenceBroker API and the ODMG API? Which one should I use in my applications?

The PersistenceBroker (PB) provides a minimal API for transparent persistence:

- O/R mapping
- Retrieval of objects with a simple query interface from RDBMS
- storing (insert, update) of objects to RDBMS
- deleting of objects from RDBMS

This is all you need for simple applications as in tutorial1.

The OJB ODMG implementation uses the PB as its persistence kernel. But it provides much more functionality to the application developer. ODMG is a full fledged API for Object Persistence, including:

- OQL Query interface

- real Object Transactions
- A Locking Mechanism for management of concurrent threads (apps) accessing the same objects
- predefined persistent capable Collections and Hashtables

Some examples explaining the implications of these functional differences:

1. Say you use the PB to query an object O that has a collection attribute col with five elements a,b,c,d,e. Next you delete Objects d and e from col and store O again with PersistenceBroker.store(O);
   PB will store the remaining objects a,b,c. But it will not delete d and e ! If you then requery object O it will again contain a,b,c,d,e !!!
   The PB keeps no transactional state of the persistent Objects, thus it does not know that d and e have to be deleted. (as a side note: deletion of d and e could also be an error, as there might be references to them from other objects !!!)
   Using ODMG for the above scenario will eliminate all trouble: Objects are registered to a transaction so that on commit of the transaction it knows that d and e do not longer belong to the collection. the ODMG collection will not delete the objects d and e but only the REFERENCES from the collection to those objects!
2. Say you have two threads (applications) that try to access and modify the same object O. The PB has no means to check whether objects are used by concurrent threads. Thus it has no locking facilities. You can get all kind of trouble by this situation. The ODMG implementation has a Lockmanager that is capable of synchronizing concurrent threads. You can even use four transaction isolation levels:
   read-uncommitted, read-committed, repeatable-read, serializable.

In my eyes the PB is a persistence kernel that can be used to build high-level PersistenceManagers like an ODMG or JDO implementation. It can also be used to write simple applications, but you have to do all management things (locking, tracking objects state, object transactions) on your own.

3.2. I don't like OQL, can I use the PersistenceBroker Queries within ODMG?

Yes you can! The ODMG implementation relies on PB Queries internally! Several users (including myself) are doing this.

If you have a look at the simple example below you will see how OJB Query objects can be used withing ODMG transactions. The most important thing is to lock all objects returned by a query to the current transaction before starting manipulating these objects.
Further on do not commit or close the obtained PB-instance, this will be done by the ODMG transaction on `tx.commit() / tx.rollback()`.

```
Transaction tx = odmg.newTransaction();
tx.begin();
....
// cast to get intern used PB instance
PersistenceBroker broker = ((HasBroker) tx).getBroker();
...
// build query
QueryByCriteria query = ...
// perform PB-query
Collection result = broker.getCollectionByQuery(query);
// use result
...

tx.commit();
...
```

3.3. The OJB JDO implementation is not finished, how can I start using OJB?

I recommend to not use JDO now, but to use the existing ODMG api for the time being.

Migrating to JDO later will be smooth if you follow the following steps. I recommend to first divide your model layer into Activity- (or Process-) classes and Entity classes.

Entity classes represent classes that must be made persistent at some point in time, say a "Customer" or a "Order" object. These persistent classes and the repsective O/R mapping in repository.xml will remain unchanged.

Activities are classes that perform business tasks and work upon entities, e.g. "edit a Customer entry", "enter a new Order"... They implement (parts of) use cases.

Activities are driving transactions against the persistent storage.

I recommend to have a Transaction interface that your Activities can use. This Transaction interface can be implemented by ODMG or by JDO Transactions (which are quite similar). The implementation should be made configurable to allow to switch from ODMG to JDO later.

The most obvious difference between ODMG and JDO are the query languages: ODMG uses OQL, JDO define JDOQL. As an OO developer you won't like both of them. I recommend to use the ojb Query objects that allow an abstract syntax representation of queries. It is possible to use these queries within ODMG transactions and it will also be possible to use them within JDO Transactions. (this is contained in the FAQ too).

Using your own Transaction interface in conjunction with the OJB query api will provide a simple but powerful abstraction of the underlying persistence layer.

We are using this concept to provide an abstract layer above OJB-ODMG, TopLink and LDAP servers in my company. Making it work with OJB-JDO will be easy!

4.2.2.4. 4. Howto

4.1. How to use OJB with my RDBMS?

please refer to this document.

4.2. What are the best settings for maximal performance?

See performance section.

4.3. How to page and sort?

Sorting can be configured by `org.apache.ojb.broker.query.Criteria::orderBy(column_name)`.

There is no paging support in OJB. OJB is concerned with Object/Relational mapping and not with application specific presentation details like presenting a scrollable page of items.

OJB returns query results as Collections or Iterators.

You can easily implement your partial display of result data by using an Iterator as returned by `ojb.broker.PersistenceBroker::getIteratorByQuery(...)`.

4.4. What about performance and memory usage if thousands of objects matching a query are returned as a Collection?

You can do two things to enhance performance if you have to process queries that produce thousands of result objects:

1. Use getIteratorByQuery() rather than getCollectionByQuery(). The returned Iterator is lazy and does not materialize Objects in advance. Objects are only materialized if you call the Iterators next() method. Thus you have total control about when and how many Objects get materialized! Please see here for proper handling.
2. You can define Proxy Objects as placeholder for your persistent business objects. Proxys are lighweight objects that contain only primary key information. Thus their materialization is not as expensive as a full object materialization. In your case this would result in a collection containing 1000 lighweight proxies. Materialization of the full objects does only occur if the objects are accessed directly. Thus you can build similar lazy paging as with the Iterator. You will find examples in the OJB test suite (src-distribution only: [db-ojb]/src/test). More info about Proxy object here.

The Perfomance of 1. will be better than 2. This approach will also work for VERY large resultsets, as there are no references to result objects that would prevent their garbage collectability.

4.5. When is it helpful to use Proxy Classes?

Proxy classes can be used for "lazy loading" aka "lazy materialization". Using Proxy classes can help you in reducing unneccessary db lookups. Example:

Say you load a ProductGroup object from the db which contains a collection of 15 Article objects.

Without proxies all 15 Article objects are immediately loaded from the db, even if you are not interested in them but just want to lookup the description-attribute of the ProductGroup object.

With a proxy class, the collection is filled with 15 proxy objects, that implement the same interface as the "real objects" but contain only an OID and a void reference.

Once you access such a proxy object it loads its "real subject" by OID and delegates the method call to it.

have a look at section proxy usage of page basic technique.

4.6. How can I convert data between RDBMS and OJB?

For Example I have a DB column of type INTEGER but a class atribute of type boolean. How can I provide an automatic mapping with OJB?

OJB provides a concept of ConversionStrategies that can be used for such conversion tasks. Have a look at the respective document.

4.7. How can I trace and/or profile SQL statements executed by OJB?

OJB ships with out of the box support for P6Spy. P6Spy is a JDBC proxy. It delegates all JDBC calls to the real JDBC driver and traces all calls to a log file.

In the file `build.properties` you have to set the switch `useP6Spy` to `true` in order to activate it:

```
# The useP6Spy switch determines if the tracing JDBC driver P6Spy
# is used.
# If you enable this switch, you must also edit the file
# jakarta-ojb/src/test/org/apache/ojb/spy.properties
# to tell P6Spy which JDBC driver to use and where to write the log.
# By default the HSQLDB driver is used.
useP6Spy=true
```

This setup uses P6Spy to trace and profile all executed SQL to a file `target/test/ojb/spy.log`. It also measures the time needed to execute each statement!

4.8. How does OJB manage foreign keys?

Automatically! you just define 1:1, 1:n or m:n associations in the repository_user.xml file. OJB does the rest!

Please refer to basic technique and xml-metadata repository for details.

4.9. Difference between getIteratorByQuery() and getCollectionByQuery()?

The first one returns an `org.apache.ojb.broker.OJBIterator` instance. The returned Iterator instance is lazy and does not materialize Objects in advance. Objects are only materialized from the underlying query result set if you call the Iterators next() method. If all objects materialized or the calling `org.apache.ojb.broker.PersistenceBroker` instance was closed or transaction demarcations ends the Iterator instance release all used resources (e.g. used Statement and

ResultSet instances).

Method `getCollectionByQuery()` use an Iterator to materialize all objects first and then return the materialized objects within the `java.util.Collection` instance.

Note:

If method `getIteratorByQuery()` was used keep in mind that the used Iterator instance is only valid as long as the used `org.apache.ojb.broker.PersistenceBroker` instance ends transaction or be closed. So it is NOT possible to get an Iterator, close the PersistenceBroker and pass the Iterator instance to a servlet or client. In that case use `getCollectionByQuery()`.

4.10. How can Collections of primitive typed elements be mapped?

The first thing to ask is: How are these primitive typed elements (Strings are also treated as primitive types here) stored in the database.
1) are they treated as ordinary domain objects and stored in a separate table?
2) are they serialized into a Varchar field?
3) are they stored as a comma separated varchar field?
4) is each element of the vector or array stored in a separate column? (this solution does only work for a fixed number of elements!)
Follow these steps for solution 3):
a) simply define ordinary collection-descriptors as for every other collection of domain objects.
b) use the Object2ByteArrFieldConversion. See jdbc-types.html for details on conversion strategies.
c) use the StringVector2VarcharFieldConversion. See jdbc-types.html for details on conversion strategies.
d) provide a field-descriptor for each element.

4.11. How could class 'myClass' represent a collection of 'myClass' objects

OJB can handle such recursive associations without problems.

- add a collection attribute 'myClasses' to the class `myClass` this collection will hold the associated `myClass` objects.
- you have to decide wether this assosciation is 1:n or m:n.
  for 1:n you just need an additional foreignkey attribute in the MY_CLASS table. Of course you'll also need a matching attribute in the class `myClass`.
  For a m:n association you'll have to define a intermediary table to hold the mapping entries.
- define a `collection-descriptor` tag in the `class-descriptor` of `myClass` in repository.xml. Follow the steps in basic technique on 1:n and m:n.

4.12. How to lookup PersistenceBroker instances?

The `org.apache.ojb.broker.PersistenceBrokerFactory` make several methods available:

```
public PersistenceBroker createPersistenceBroker(PBKey key) throws PBFactoryException;

public PersistenceBroker createPersistenceBroker(String jcdAlias, String user, String password)
       throws PBFactoryException;

public PersistenceBroker defaultPersistenceBroker() throws PBFactoryException;
```

Method `defaultPersistenceBroker()` can be used if the attribute *default-connection* is set *true* in *jdbc-connection-descriptor*. It's a convenience method, useful when only one database is used.

The standard way to lookup a broker instance is via `org.apache.ojb.broker.PBKey` by specify *jcdAlias* (specified in the repository file (or sub file)), *user* and *passwd*. If the user and password is already set in *jdbc-connection-descriptor* it is possible to lookup the broker instance only be specify the *jcdAlias* in PBKey:

```
PBKey pbKey = new PBKey("myJcdAliasName");
```

```
PersistenceBroker broker = PersitenceBrokerFactory.createPersistenceBroker(pbKey);
```

See here too.

4.13. How to access ODMG?

Obtain a `org.odmg.Implementation` instance first, then create new `org.odmg.Database` instance and open this instance by setting the used jcd-alias name:

```
Implementation odmg = OJB.getInstance();
Database database = odmg.newDatabase();
database.open("jcdAliasName#user#password", Database.OPEN_READ_WRITE);
```

The *user* and *password* separated by # hash only needed, when the user/passwd not specified in the connection metadata (jdbc-connection-descriptor).

4.14. Needed to put user/password of database connection in repository file?

There is no need to put user/password in the repository file (more exact in the `jdbc-connection-descriptor`). You can pass this information at runtime. See Many different database user - How do they login?.

Only if you want to use convenience `PersistenceBroker` lookup method of `PersistenceBrokerFactory`, OJB needs all database connection information in the configuration files. More details see repository file doc - section jdbc-connection-descriptor `default-connection` attribute

See lookup PB api.
See lookup ODMG api.

```
PBKey pbKey = new PBKey(jcdAlias, user, passwd);
PersistenceBroker broker = PersistenceBrokerFactory.createPersistenceBroker(pbKey);
// or using a convenience (when default-connection was set in jdbc-connection-descriptor)
PersistenceBroker broker = PersistenceBrokerFactory.defaultPersistenceBroker();
```

4.15. Many different database user - How do they login?

There are two ways to do that. Define for each user a `jdbc-connection-descriptor` (unattractive way, because we have to add each new user to repository file), or let OJB handle this for you.
For it define **one** `jdbc-connection-descriptor`, now you can use the same `jcdAlias` name with different `User/Password`. OJB **copy** the defined `jdbc-connection-descriptor` and replace the `username` and `password` with the given `User/Password`.

PersistenceBroker-api example:

```
PBKey user_1 = new PBKey(jcdAlias,username, passwd);
PersistenceBroker broker =
PersistenceBrokerFactory.createPersistenceBroker(user_1);
...
```

ODMG-api example:

```
Implementation odmg = OJB.getInstance();
Database db = odmg.newDatabase();
db.open("jcdAlias#username#passwd", Database.OPEN_READ_WRITE);
...
```

Keep in mind, when the `connection-pool` element enables connection pooling, every user get its separate pool. See How does OJB handle connection pooling?.

4.16. How do I use multiple databases within OJB?

Define for each database a `jdbc-connection-descriptor`, use the different `jcdAlias` names in the repositry file to match the according database.

```
<jdbc-connection-descriptor
    jcd-alias="myFirstDb"
    ...
>
    ...
</jdbc-connection-descriptor>

<jdbc-connection-descriptor
    jcd-alias="mySecondDb"
    ...
>
    ...
</jdbc-connection-descriptor>
```

> **Note:**
> OJB does not provide distributed transactions by itself. To use distributed transactions, OJB have to be integrated in an j2ee conform environment (or made work with an JTA/JTS implementation).

4.17. How does OJB handle connection pooling?

Please have a look in section Connection Handling.

4.18. Can I directly obtain a java.sql.Connection within OJB?

Please have a look in section Connection Handling.

4.19. Is it possible to perform my own sql-queries in OJB?

There are serveral ways in OJB to do that.
If you completely want to bypass the OJBquery-api see Can I directly obtain a java.sql.Connection within OJB?.
A more elegant way is to use a `QueryBySQL` object:

```
String sql =
"SELECT A.Artikel_Nr FROM Artikel A, Kategorien PG"
+ " WHERE A.Kategorie_Nr = PG.Kategorie_Nr"
+ " AND PG.Kategorie_Nr = 2";
// get the QueryBySQL
Query q2 = QueryFactory.newQuery(Article.class, sql);

Iterator iter2 = broker.getIteratorByQuery(q2);
// or
Collection col2 = broker.getCollectionByQuery(q2);
```

4.20. Start OJB without a repository file?

See section Metadata Handling.

4.21. Connect to database at runtime?

See section Metadata Handling.

4.22. Add new persistent objects metadata (class-descriptor) at runtime?

See section Metadata Handling.

4.23. Global metadata changes at runtime?

Please see section Metadata Handling.

4.24. Per thread metadata changes at runtime?

Please see section Metadata Handling.

4.25. Is it possible to use OJB within EJB's?

Yes, see deployment instructions in the docs. Additional you can find some EJB example beans in package `org.apache.ojb.ejb` under `[jakarta-ojb]/src/ejb`.

4.26. Can OJB handle ternary (or higher) associations?

Yes, that's possible. Here is an example. With a ternary relationship there are three (or more) entities 'related' to each other. An example would be `Developer`, `Language` and `Project`.

Each entity is mapped to one table (`DEVELOPER`, `LANGUAGE` and `PROJECT`). To represent the combinations of these entities we need an additional bridge table (`PROJECTRELATIONSHIP`)with three columns holding the foreign keys to the other three tables (just like an m:n association is represented by an intermediary table with 2 columns).

To handle this table with OJB we have to define a class that is mapped on it. This Relationship class can then be used to perform queries/updates as with any other persistent class. Here is the layout of this class:

```
public class ProjectRelationship {
  Integer developerId;
  Integer languageId;
  Integer projectId;

  Developer developer;
  Language lanuage;
  Project project;

  /** setters and getters not shown for brevity**/
}
```

Here is the respective extract from the repository :

```
<class-descriptor
    class="ProjectRelationship"
    table="PROJECTRELATIONSHIP"
>
    <field-descriptor
        name="developerId"
        column="DEVELOPER_ID"
        jdbc-type="INTEGER"
        primarykey="true"
    />
      <field-descriptor
          name="languageId"
          column="LANGUAGE_ID"
        jdbc-type="INTEGER"
        primarykey="true"
    />
      <field-descriptor
          name="projectId"
          column="PROJECT_ID"
        jdbc-type="INTEGER"
        primarykey="true"
    />
      <reference-descriptor
          name="developer"
          class-ref="Developer"
      >
        <foreignkey field-id-ref="developerId" />
      </reference-descriptor>
      <reference-descriptor
          name="language"
          class-ref="Language"
      >
        <foreignkey field-id-ref="languageId" />
      </reference-descriptor>
      <reference-descriptor
          name="project"
          class-ref="Project"
      >
        <foreignkey field-ref="projectId" />
      </reference-descriptor>
</class-descriptor>
```

Here is some sample code for storing a relationship :

```
Developer dev = .... ; // create or retrieve
Project  proj = .... ; // create or retrieve
Language lang = .... ; // create or retrieve

ProjectRelationship rel = new ProjectRelationship();
rel.setDeveloper(dev);
rel.setLanguage(lang);
rel.setProject(proj);

broker.store(r);
```

In the next code sample we are looking up all Projects that Developer "Bob" has done in "Java".

```
Criteria criteria = new Criteria();
criteria.addEqualTo("developer.name","Bob");
cirteria.addEquatTo("language.name","Java");

Query q = new QueryByCriteria(ProjectRelationship.class, criteria, true);
Iterator iter = Broker.getIteratorByQuery(q);

// now iterate over the collection and retrieve all projects:
while (iter.hasNext())
{
    ProjectRelationship rel = (ProjectRelationship) iter.next();
    System.out.println(rel.getProject().toString());
}
```

You could also have on the Project class-descriptor a `collection-descriptor` that returns all relationships associated with the Project. If it was call "projectRelationships" the following would give you all projects that have a relationship with "bob" and the language "java".

```
Criteria criteria = new Criteria();
criteria.addEqualTo("projectRelationships.developer.name","bob");
cirteria.addEquatTo("projectRelationships.language.name","java");

Query q = new QueryByCriteria(Project.class, criteria, true);
Collection projects = Broker.getCollectionByQuery(q);
```

This is the layout of the Project class:

```
public class Project {
  Integer id;
  String name;
  Collection projectRelationships;

  /** setters and getters not shown for brevity**/
}
```

This is the class-descriptor of the Project class:

```
<class-descriptor
```

```
    class="Project"
    table="PROJECT"
>
    <field-descriptor
        name="id"
        column="ID"
        jdbc-type="INTEGER"
        primarykey="true"
    />
      <field-descriptor
          name="name"
          column="NAME"
        jdbc-type="VARCHAR"
    />
      <collection-descriptor
          name="projectRelationships"
          element-class-ref="ProjectRelationship"
      >
        <inverse-foreignkey field-ref="projectId" />
      </collection-descriptor>
</class-descriptor>
```

4.27. How to map a list of Strings

You can not map a list of Strings with a collection descriptor. A collection descriptor can only be used if the element class is a persistent class too. But element-class-ref="java.lang.String" won't work, because it's no persistent entity class!
Follow these steps to provide a mapping for an attribute holding alist of Strings. Let's assume your persistent class has an attribute `listOfStrings` holding a list of Strings:

```
protected Collection listOfStrings;
```

The database table mapped to the persistent class has a colum `LIST_OF_STRINGS` of type `VARCHAR` that is used to hold all strings.

```
<field-descriptor
    name="listOfStrings"
    column="LIST_OF_STRINGS"
    jdbc-type="VARCHAR"
    conversion=
"o.a.ojb.broker.accesslayer.conversions.StringVector2VarcharFieldConversion"
/>
```

4.28. How to set up Optimistic Locking

Optimistic locking use an additional column (*Timestamp* or *Integer*) which is incremented each time changes are committed to the object, and is utilizied to determine whether an optimistic transaction should succeed or fail. Optimistic locking is fast, because it checks data integrity only at update time.

1. In your table you need a dedicated column of type `INTEGER` or `TIMESTAMP`. Say the column is typed as `INTEGER` and named `VERSION_MAINTAINED_BY_OJB`.
2. You then need a (possibly private) attribute in your java class corresponding to the column. Say the attribute is defined as:

   ```
   private int versionMaintainedByOjb;
   ```
3. in repository.xml you need a field-descriptor for this attribute. this field-descriptor must specify `locking="true"`
4. The resulting field-descriptor will look as follows:

   ```
   <field-descriptor
       name="versionMaintainedByOjb"
       column="VERSION_MAINTAINED_BY_OJB"
       jdbc-type="INTEGER"
       locking="true"
   />
   ```

For further reference see also the repository documentation.

4.29. How to use OJB in a cluster

Q: I'm running a web site in a load-balanced/cluster environment. Multiple servlet engines (different VMs/HTTP sessions), each running an OJB instance, against a single shared database. How should OJB be configured to get the concurrent servlet engines synchronized properly?

transactional isolation and locking
If you are using the PersistenceBroker API use optimistic locking (OL) to let OJB handle write conflicts. To use OL define a TIMESTAMP or INTEGER column and the respective Java attribute for it. In the field-descriptor of this attribute set the attribute *locking="true"*.
If you are working with the ODMG API distributed pessemistic locking should be used, by setting the respective flag in OJB.properties.

sequence numbers
Use a SequenceManager that is safe across multiple JVMs. The NextVal based SequenceManagers or any other SequenceManager based on database mechanisms will be fine.

caching
You could use different caching implementations

1. Use the EmptyCacheImpl to avoid any dirty reads. (But: The EmptyCache cannot handle cyclic structures on load!)
2. Use the PerBrokerCache Implementation to avoid dirty reads.
3. Use the OSCache cache implementation as distributed object cache.

There is also a complete howto document available that covers these topics.

4.30. How to work with the ObjectCacheEmptyImpl

Q: I just tried to turn caching off by using ObjectCacheEmptyImpl setting in ObjectCacheClass, and it seems to continuously loop through the SQL statement infinitely. The default works fine though.
Any ideas why this might be?

A: The Problem you see is due to circular references in your data. Say A references B and B has a backreference to A.
Now we load A from the DB. If autoretrieve="true" for the reference-descriptor defining the reference to B, OJB will also load B. If autoretrieve="true" for the B-reference-descriptor describing the back-reference to A, OJB must retrieve A. And here is the key point.
If we use the defaultcache A will be in the cache already, as it was loaded first. So OJB will simply lookup A from the cache. No endless recursion!
But if we use the emptycache, A will not be cached. So OJB must load A from the DB. And then again B is retrieved, etc., etc. There's you endless recursion.
In other words: A non-empty cache is needed to allow proper loading of circular references. (Other O/R tools like TopLink work the same way).
If you still want to use the EmptyCacheImpl you should set autoretrieve="false" and load references explicitely by broker.retrieveReference(...).

4.31. JDO - Why must my persisten class implement javax.jdo.spi.PersistenceCapable?

As specified by JDO all persistent classe must implement the interface `javax.jdo.spi.PersistenceCapable`. If a class does not implement this interface a JDO implementation does not know how to handle it.
On the other hand the JDO spec claims to provide transaparent persistence. That is no persistence class is required to implement a specific interface or to be derived from a special base class.
Sounds like a contradiction? It is! The JDO spec resolves this contradiction by stating that a JDO implemention is responsible to add the methods required by `javax.jdo.spi.PersistenceCapable` to the the user classes. This "injection" could be achieved by Pre- or Post-processing. The strategy most implementations use is called "bytecode-enhancement". This is a postprocesing step that adds the required methods to the .class files of the persistent user classes.

The JDO Reference implementation also uses bytecode-enhancement. In order to enhance the Product class to implement the `javax.jdo.spi.PersistenceCapable` interface use the ant target "enhance-jdori" before launching the tutorial5 application. This is documentated in the first section of tutorial4.html.

4.3. ObJectRelationalBridge - Getting Started

This document will guide you through the very first steps of setting up a project with OJB. To make this easier, OJB comes with a blank project template called ojb-blank which you're encouraged to use. You can download it here.

For the purpose of this guide, we'll be showing you how to setup the project for a simple application that handles products and uses MySQL. This is continued later on in the next tutorial parts.

4.3.1. Acquiring ojb-blank

First off, OJB uses Ant to build, so please install it prior to using OJB. In addition, please make sure that the environment variables `ANT_HOME` and `JAVA_HOME` are correctly set to the top-level folders of your Ant distribution and your JDK installation, respectively.

Next download the latest ojb-blank and OJB binary distributions. You can also start with the source distribution rather than the binary as the unit tests provide excellent sample code and you can build the ojb-blank project on your own with it.

The ojb-blank project contains all libraries necessary to get running. However, there may be additional libraries required when you venture deeper into OJB's APIs. See here for a list of additional libraries.
Most notably, you'll probably want to add the jdbc driver for you database unless you plan to use the embedded Hsqldb database for which the ojb-blank project is pre-configured (including all necessary jars).

4.3.2. Contents of ojb-blank

Copy the `ojb-blank.jar` file to your project directory and unpack it via the command

```
jar xvf ojb-blank.jar
```

This will unpack it into the `ojb-blank` directory under wherever you unpacked it from. You can move things out of that directory into your project directory, or, more simply, rename the `ojb-blank` directory to be whatever you want your project directory to be named.
After you unpacked the jar, you'll get the following directory layout:

```
\ojb-blank
    .classpath
    .project
    build.properties
    build.xml
    \lib
    \src
        \java
        \resources
        \schema
        \test
```

Here's a quick rundown on what the individual directories and files are:

.classpath, .project
An Eclipse project for your convenience. You can simply import it into Eclipse via **File -> Import... -> Existing Project into Workspace**.

build.xml, build.properties
The Ant build script and the build properties. These are described in more detail below.

lib
Contains the libraries necessary to compile and run your project. If you want to use a different database than

Hsqldb, then put the jars of your jdbc driver in here.

src/java

Put your java source code here.

src/resources

Contains the runtime configuration files for OJB. For more detail see below.

src/schema

Here you will find a schema containing tables that are required by certain components of OJB such as clustered locking and OJB managed sequences. More information on these tables is available in the platform documentation. The schema is in a database-independent format that can be used by Torque or commons-sql to create the database.

The ojb-blank project contains the runtime files of Torque 3.0.2, and provides a build target that can be invoked on your schema (see below for details). Therefore, this directory also contains the build script of Torque, but you won't need to invoke it directly.

src/java

Place your unit tests in here.

4.3.2.1. Sample project

For our sample project, we should rename the directory to something more fitting, like `productmanager`.

Also, since we're using MySQL, we put the MySQL jar of the jdbc driver, which is called something like `mysql-connector-java-[version]-stable-bin.jar`, into the `lib` subdirectory.

The only other thing missing is the source code, but since that's what the other tutorials are dealing with, we will silently assume that it is already present in the `src/java` subdirectory.
If you don't want to write the code yourself, you can use the code from one of the tutorials which you can download here.

Warning:

Note that if you do not intent to use JDO, then you should delete the files in the `ojb.apache.ojb.tutorial5`, otherwise you'll get compilation errors.

4.3.3. The build files

4.3.3.1. Configuration via build.properties

The next step is to adapt the build files, especially the `build.properties` file to your environment. It basically contains two sets of information, the database settings and the build configuration. While you shouldn't have to change the latter, the database settings probably need to be adapted to suit your needs:

Property	Purpose
jcdAlias	The name of the connection. You should leave the default value, which is `default`.
databaseName	This is the name of the database, per default `ojb_blank`.
databaseUser	The user name for accessing the database (default: sa). If you're using Torque to create the database, then this user also requires sufficient rights to create databases and tables.
databasePassword	Password for the user, per default empty.
dbmsName	The type of database, which is one of the following: **Db2**, **Firebird**, **Hsqldb**, **Informix**, **MaxDB**,

	MsAccess, **MsSQL**, **MySQL**,**Oracle** (pre-9i versions), **Oracle9i**, **WLOracle9i** (Oracle 9i or above used from WebSphere), **PostgreSQL**, **Sapdb**, **Sybase** (generic), **SybaseASA**, **SybaseASE**. Please note that this setting is case-sensitive. Per default, Hsqldb is used, which is an embedded database. All files required for this database come with the ojb-blank project.
jdbcRuntimeDriver	The fully-qualified classname of the jdbc driver. For Hsqldb this is `org.hsqldb.jdbcDriver`.
jdbcLevel	The jdbc level that the driver conforms to. Please check the documentation of your jdbc driver for this value, though most jdbc drivers conform to version 2.0 at least. For the Hsqldb jdbc driver this is 2.0.
urlProtocol	The protocol of the database url (see below), usually `jdbc`.
urlSubprotocol	The sub-protocol of the database url which is database- and driver-specific. For Hsqldb, you're using `hsqldb`.
urlDbalias	This is the address that points the jdbc driver to the database. For Hsqldb this is per default the database name.
torque.database	If you're using Torque to create the database, then you have to set the database here (again). Unfortunately, this value is different from the `dbmsName` which defines the database for OJB. Currently, these values are defined: **axion**, **cloudscape**, **db2**, **db2400**, **hypersonic** (which is Hsqldb), **interbase** (use for Firebird), **mssql**, **mysql**, **oracle**, **postgresql**, **sapdb**, and **sybase**. Default value is `hypersonic` for use with Hsqldb.
torque.database.createUrl	This specifies the url that Torque will use in order to create the database. Depending on the database, this may be the same as the normal access url (the default value), but for some database this is different. Please check the manual of your database for this url.

If you know how the jdbc url for connecting to your database looks like, then you can derive the settings `databaseName`, `databaseName`, `databaseName` and `databaseName` easily:

Assume this url is given as:

```
jdbc:mysql://localhost:3306/myDatabase
```

then these properties are

Property	Value
databaseName	`myDatabase`
urlProtocol	`jdbc`
urlSubprotocol	`mysql`

urlDbalias	`//localhost/myDatabase`

4.3.3.2. Building via build.xml

After setting up the build you're probably eager to actually build the project. Here's the actions that you can perform using the Ant build file `build.xml`:

Action (target in the build.xml file)	What it does
clean	Cleans up all files from the previous build.
compile	Compiles your java source files to `build/classes`. Usually, you don't run this target, but rather the next one which includes the compilation step.
build	Compiles your java sources files (using the compile action), and prepares the runtime configuration files using the settings that you specified in the `build.properties` file, most notably the `repository_database.xml` which will be located in the `build/resources` directory after the build. After you run this action, your application is ready to go (if the action ran successfully, of course).
jar	A convenience action that packs your successfully build application into a jar.
xdoclet	Creates the runtime configuration files that describe the repository, from javadoc comments embedded in your java source files. Details on how to this are given in the tutorials and in the documentation of the XDoclet OJB module.
setup-db	Creates the database and tables from a database-independent schema using Torque. You'll find more info on this schema in the documentation of the XDoclet OJB module and on the Torque homepage.
enhance-jdori	This is a sample target that shows how a class meant to be persistent with JDO, is processed by the JDO bytecode enhancer from the JDO reference implementation. It uses the `Product` class from the JDO tutorial (tutorial 5).

So, a typical build would be achieved with this Ant call:

```
ant build
```

If you want to create the database as well, and you have javadoc comments in your source code that describe the repository, then you would call Ant this way:

```
ant build setup-db
```

This will perform in that order the actions `build`, `xdoclet` (invoked automatically from the next action) and `setup-db`. Of course, you do not need to use Torque to setup your database, but it is a convenient way to do so.

4.3.3.3. Sample project

First we change the database properties to these values (assuming that Torque will be used to setup the database):

Property	Value
jcdAlias	We leave the default value of `default`.
databaseName	Since the application manages products, we call the database `productmanager`.
databaseUser	This depends on your setup. For the purposes of this guide, let's call him `steve`.
databasePassword	Again depending on your setup. How about `secret` (you know that you should not use this password in reality ?!).
dbmsName	`MySQL`
jdbcRuntimeDriver	Its called `com.mysql.jdbc.Driver`.
jdbcLevel	For the newer Mysql drivers this is 3.0.
urlProtocol	The default of `jdbc` will do.
urlSubprotocol	For MySQL, we're using `mysql`.
urlDbalias	Assuming that the database runs locally on the default port, we have `//localhost/${databaseName}`.
torque.database	We want to use Torque, so we put `mysql` here.
torque.database.createUrl	MySQL allows to create a database via jdbc. The url that we should use to do so, is the normal url used to access the database minus the database name. So the value here is: `${urlProtocol}:${urlSubProtocol}://localhost/`. Please note that the trailing slash is important.

Ok, now we have everything configured for building. The `build.properties` file now looks like this (the comments have been removed for brevity):

```
jcdAlias=default
databaseName=productmanager
databaseUser=steve
databasePassword=secret

dbmsName=MySQL
jdbcLevel=3.0
jdbcRuntimeDriver=com.mysql.jdbc.Driver
urlProtocol=jdbc
urlSubprotocol=mysql
urlDbalias=//localhost/${databaseName}

torque.database=mysql
torque.database.createUrl=${urlProtocol}:${urlSubprotocol}://localhost/

jar.name=projectmanager.jar

source.dir=src
source.java.dir=${source.dir}/java
source.resource.dir=${source.dir}/resources
source.test.dir=${source.dir}/test
source.schema.dir=${source.dir}/schema

build.dir=build
build.lib.dir=lib
build.classes.dir=${build.dir}/classes/
build.resource.dir=${build.dir}/resources/

target.dir=target
```

Looks like we're ready for building. Again, we're assuming that the source code is already present. So we're invoking Ant now in the top-level folder `productmanager`:

```

ant build setup-db
```

which should (assuming five java classes) produce an output like this

```

Buildfile: build.xml

compile:
    [mkdir] Created dir: /home/steve/projects/productmanager/build
    [mkdir] Created dir: /home/steve/projects/productmanager/build/classes
    [javac] Compiling 5 source files to /home/steve/projects/productmanager/build/classes

build:
     [copy] Copying 10 files to /home/steve/projects/productmanager/build/resources

xdoclet:
[ojbdoclet] (XDocletMain.start                   47  ) Running <ojbrepository/>
[ojbdoclet] Generating ojb repository descriptor (build/resources//repository_user.xml)
[ojbdoclet] Type test.Project
[ojbdoclet] Processed 5 types
[ojbdoclet] Processed 5 types
[ojbdoclet] (XDocletMain.start                   47  ) Running <torqueschema/>
[ojbdoclet] Generating torque schema (build/resources//project-schema.xml)
[ojbdoclet] Processed 5 types

setup-db:

check-use-classpath:

check-run-only-on-schema-change:

sql-check:

sql:
     [echo] +------------------------------------------+
     [echo] |                                          |
     [echo] | Generating SQL for YOUR Torque project!  |
     [echo] | Woo hoo!                                 |
     [echo] |                                          |
     [echo] +------------------------------------------+

sql-classpath:
[torque-sql] Using contextProperties file:
             /home/steve/projects/productmanager/build.properties
[torque-sql] Using classpath
[torque-sql] Generating to file
             /home/steve/projects/productmanager/build/resources/report.productmanager.sql.generation
[torque-sql] Parsing file: 'ojbcore-schema.xml'
[torque-sql] (transform.DTDResolver               128 ) Resolver: used database.dtd from
             org.apache.torque.engine.database.transform package
[torque-sql] Parsing file: 'project-schema.xml'
[torque-sql] (transform.DTDResolver               140 ) Resolver: used
             http://jakarta.apache.org/turbine/dtd/database.dtd

sql-template:

create-db-check:

create-db:
[torque-data-model] Using classpath
[torque-data-model] Generating to file
```

```
                    /home/steve/projects/productmanager/build/resources/create-db.sql
[torque-data-model] Parsing file: 'ojbcore-schema.xml'
[torque-data-model] (transform.DTDResolver               128 ) Resolver: used database.dtd from
                    org.apache.torque.engine.database.transform package
[torque-data-model] Parsing file: 'project-schema.xml'
[torque-data-model] (transform.DTDResolver               140 ) Resolver: used
                    http://jakarta.apache.org/turbine/dtd/database.dtd
     [echo]
     [echo]       Executing the create-db.sql script ...
     [echo]
      [sql] Executing file:
            /home/steve/projects/productmanager/build/resources/create-db.sql
      [sql] 2 of 2 SQL statements executed successfully

insert-sql:
[torque-sql-exec] Our new url -> jdbc:mysql://localhost/productmanager
[torque-sql-exec] Executing file:
                  /home/steve/projects/productmanager/build/resources/project-schema.sql
[torque-sql-exec] Executing file:
                  /home/steve/projects/productmanager/build/resources/ojbcore-schema.sql
[torque-sql-exec] 50 of 50 SQL statements executed successfully

BUILD SUCCESSFUL
```

That was it. You now have your database setup properly. Go on, have a look:

```

mysql -u steve productmanager

mysql> show tables;
```

There, all tables for your project, as well as the tables required for some OJB functionality which we also used in the above process (you can recognize them by their names which start with `ojb_`).

4.3.4. The runtime configuration files

The last thing missing for actually running your project is to adapt the runtime configuration files used by OJB. There are basically three sets of configuration that need to be provided: configuration of the OJB runtime, description of the database connection, and description of the repository.

4.3.4.1. Configuring the OJB runtime

With the OJB.properties file and OJB-logging.properties (both located in `src/resources`), you configure and finetune the runtime aspects of OJB. For a simple application you'll probably won't have to change anything in them, though.

4.3.4.2. Configuring the database connection

For projects that use OJB, you configure the connections to the database via jdbc connection descriptors. These are usually defined in a file called `repository_database.xml` (located in `src/resources`). In the ojb-blank project, the build file will setup this file for you and place it in the `build/resources` directory.

4.3.4.3. Configuring the repository

Finally you need to configure the repository. It consists of descriptors that define which java classes are mapped in what way to which database tables, and it is typically contained in the `repository_user.xml` file. This is the most complicated configuration part which will be explained in much more detail in the rest of the tutorials.
An convenient way of creating the repository metadata is to use the XDoclet OJB module. Basically, you put specific Javadoc comments into your source code, which are then processed by the build file (`xdoclet` and `setup-db` targets) and the repository metadata and the database schema are generated.

4.3.4.4. Sample project

Actually, there is not much to do here. For our simple sample application the default properties of OJB work just fine, so we leave `OJB.properties` and `OJB-logging.properties` untouched.

Also, the build file generated the connection descriptor for us, and we were using the XDoclet OJB module and Torque to generate the repository metadata and database for us. For instance, the processed connection descriptor (file `build/resources/repository_database.xml`) looks like this:

```
<jdbc-connection-descriptor
    jcd-alias="default"
    default-connection="true"
    platform="MySQL"
    jdbc-level="3.0"
    driver="com.mysql.jdbc.Driver"
    protocol="jdbc"
    subprotocol="mysql"
    dbalias="//localhost/productmanager"
    username="root"
    password=""
    eager-release="false"
    batch-mode="false"
    useAutoCommit="1"
    ignoreAutoCommitExceptions="false"
>
    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
        <attribute attribute-name="timeout" attribute-value="900"/>
        <attribute attribute-name="autoSync" attribute-value="true"/>
    </object-cache>
    <connection-pool
        maxActive="21"
        validationQuery="" />
    <sequence-manager className="org.apache.ojb.broker.util.sequence.SequenceManagerHighLowImpl">
        <attribute attribute-name="grabSize" attribute-value="20"/>
        <attribute attribute-name="autoNaming" attribute-value="true"/>
        <attribute attribute-name="globalSequenceId" attribute-value="false"/>
        <attribute attribute-name="globalSequenceStart" attribute-value="10000"/>
    </sequence-manager>
</jdbc-connection-descriptor>
```

If you're curious as to what this stuff means, check this reference guide.

The repository metadata (file `build/resources/repository_user.xml`) starts like:

```
<class-descriptor
    class="productmanager.Product"
    table="Product"
>
    <field-descriptor
        name="name"
        column="name"
        jdbc-type="VARCHAR"
        length="32"
    >
    </field-descriptor>
    <field-descriptor
        name="price"
        column="price"
        jdbc-type="FLOAT"
    >
    </field-descriptor>
    <field-descriptor
        name="stock"
        column="stock"
        jdbc-type="INTEGER"
    >
    </field-descriptor>
    <field-descriptor
```

```
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
    >
    </field-descriptor>
</class-descriptor>
...
```

Now you should be able to run your application:

```
cd build/resources

java productmanager.Main
```

Of course, you'll need to setup the `CLASSPATH` before running your application. You'll should add all jars except the ones for Torque (`torque-[version].jar`, `velocity-[version].jar` and `commons-collections-[version].jar`) and for the XDoclet OJB module (`xdoclet-[version].jar`, `xjavadoc-[version].jar` and `xdoclet-ojb-module-[version].jar`).

It is important to note that OJB per default assumes the `OJB.properties` and `OJB-logging.properties` files in the directory where you're starting the application. Hence, we changed to the `build/resources` directory.

Per default, the same applies to the other configuration files (`repository*.xml`) but you can change this in the `OJB.properties` file.

4.3.5. Learning More

After you've have learned about building and configuring projects that use OJB, you should check out the tutorials to learn how to specify your persistent classes and how to use OJB's APIs to perform database operations. The Mapping Tutorial in particular shows you how to map your classes to tables in an RDBMS.

4.4. Tutorials

4.4.1. Tutorial Summary

4.4.1.1. Tutorials

Here can be found a summary of all tutorials.

- Object-Relational Mapping
  The Object-Relational Mapping tutorial walks though a basic metadata mapping for an object to a relational database.
- The Persistence Broker API
  The PB tutorial demonstrates how to use the `PersistenceBroker` API which forms an object persistence kernel for OJB. While it is the lowest level API provided by OJB it is also exceptionally easy to use.
- The ODMG API
  The ODMG API tutorial steps though using the ODMG 3.0 API provided by OJB. This is an industry standard API designed for Object Databases.
- The JDO API
  JDO is a standard API for accessing persistent objects in Java. This tutorial steps through how to use OJB's JDO plugin.
- The Object Transaction Manager
  The OTM is OJB's implementation of object level transactions. These are transactions independent of the underlying relational database providing more efficient resource utilisation and extremely flexible locking semantics.

Further strongly recommended documentation for all beginners:

- OJB Queries
  This document explains the usage of the query syntax.

- Basic O/R Technique
  This tutorial explains basic object-relational mapping technique in OJB like 1:1, 1:n and m:n relations, the auto-xxx settings for references and proxy objects/collections.
- Tools to build large metadata mappings
  Explains how to build large metadata mapping and present useful tools.

4.4.2. Mapping Tutorial

4.4.2.1. What is the Object-Relational Mapping Metadata?

The O/R mapping metadata is the specific configuration information that specifies how to map classes to relational tables. In OJB this is primarily accomplished through an xml document, the `repository.xml` file, which contains all of the initial mapping information.

The Product Class

This tutorial looks at mapping a simple class with no relations:

```
package org.apache.ojb.tutorials;

public class Product
{
    /** product name */
    private String name;

    /** price per item */
    private Double price;

    /** stock of currently available items */
    private int stock;

    ...
}
```

This class has three fields, `price`, `stock`, and `name`, that need to be mapped to the database. Additionally, we will introduce one artificial field used by the database that has no real meaning to the class, an artificial key primary id:

```
/** Artificial primary-key */
private Integer id;
```

Including the primary-key attribute in the class definition is mandatory, but under certain conditions anonymous keys can also be used to keep this database artifact hidden in the database. However, as access to an artifical unique identifier for a particular object instance can be useful, particularly in web-based applications, this tutorial will expose it

The Database

OJB is very flexible in terms of how it can map classes to database tables, however the simplest technique for mapping a single class to a relational database is to map the class to a single table, and each attribute on the class to a single column. Each row will then represent a unique instance of that class.

The DDL for such a table, for the `Product` class might look like:

```
CREATE TABLE Product
(
    id INTEGER PRIMARY KEY,
    name VARCHAR(100),
    price DOUBLE,
    stock INTEGER
    )
```

The individual field names in the database and class definition match here, but this is no requirement. They may vary independently of each other as the metadata will specify what maps to what.

The Metadata

The `repository.xml` document is split into several physical documents. The `repository_user.xml` xml file is used to contain user-defined mappings. OJB uses the other ones for managing other metadata, such as database information.

In general each class will be defined within a `class-descriptor` element with `field-descriptoy` child elements for each field. In addition the mapping of references and collections is described in the basic technique section. This tutorial sticks to mapping a single, simplistic, class.

The complete mapping for the `Product` class is as follows:

```
    <class-descriptor
           class="org.apache.ojb.tutorials.Product"
           table="Product"
    >
        <field-descriptor
           name="id"
           column="id"
           primarykey="true"
           autoincrement="true"
        />
        <field-descriptor
           name="name"
           column="name"
        />
        <field-descriptor
           name="price"
           column="price"
        />
        <field-descriptor
           name="stock"
           column="stock"
        />
    </class-descriptor>
```

Examine the `class-descriptor` element. It has two attributes:

- class - This attribute is used to specify the fully-qualified Java class name for this mapping.
- table - This attribute specifies which table is used to store instances of this class.

Other information can be specified here, such as proxies and custom row-readers as specified in the repository.xml documentation.

Examine now the first `field-descriptor` element. This is used to describe the `id` field of the `Product` class. Two required attributes are specified:

- name - This specifies the name of the instance variable in the Java class.
- column - This specifies the column in the table specified for this class used to store the value.

In addition to those required attributes, notice that the first element specifies two optional attributes:

- primary-key - This attribute specifies that this field is the primary key for this class.
- autoincrement - The `autoincrement` attribute specifies that the value will be automatically assigned by OJB sequence manager. This might use a database supplied sequence, or, by default, an OJB generated value.

Using the XDoclet module

OJB provides an XDoclet module to make generating the repository descriptor and the corresponding table schema easier. An

XDoclet module basically processes custom JavaDoc tags in the source code, and generates files from them. In the case of OJB, two types of files can be generated: the repository descriptor (`repository_user.xml`) and a Torque schema which can be used to create the tables in the database. This provides one important benefit: the descriptor and the database schema are much more likely in sync with the code thus avoiding errors that are usually hard to find. Furthermore, the XDoclet module contains some checks that find common mapping errors.

In the above example, the source code for Product class with JavaDoc tags would look like:

```
package org.apache.ojb.tutorials;

/**
 * @ojb.class
 */
public class Product
{
    /**
     * Artificial primary-key
     *
     * @ojb.field primarykey="true"
     *            autoincrement="ojb"
     */
    private Integer id;

    /**
     * product name
     *
     * @ojb.field length="100"
     */
    private String name;

    /**
     * price per item
     *
     * @ojb.field
     */
    private Double price;

    /**
     * stock of currently available items
     *
     * @ojb.field
     */
    private int stock;
}
```

As you can see, much of the stuff that is present in the descriptor (and the DDL) is generated automatically by the XDoclet module, e.g. the table/column names and the jdbc-types. Of course, you can also specify them in the JavaDoc tags, e.g. if they differ from the java names.

For details on OJB's JavaDoc tags and how to generate and use the mapping files please see the OJB XDoclet Module documentation.

4.4.2.2. Advanced Topics

Relations

As most object models have relationships between objects, mapping specific types of relationships (1:1, 1:Many, Many:Many) is important in mapping objects into a relational database. The basic technique tutorial discusses this in great detail.

It is important to note that this metadata mapping can be modified at runtime through the `org.apache.ojb.metadata.MetadataManager` class.

Inheritence

OJB can map inheritence hierarchies using a variety of techniques discussed in the Extents and Polymorphism section of the Advanced O/R Documentation

Anonymous Keys

This tutorial uses explicit keys mapped into the Java class. It is also possible to keep artificial keys completely hidden within the database. The Anonymous Keys HOWTO explains how this is accomplished.

Large Projects

Projects with small numbers of persistent classes can be mapped by hand, however, many projects can have hundreds, or even thousands, of distinct classes which must be mapped. In these circumstances managing the class-database mapping by hand is not viable. The How To Build Mappings HOWTO explores different tools which can be used for managing large-scale mapping.

Custom JDBC Mapping

OJB maps Java types to JDBC types according to the JDBC Types table. You can, however, define custom JDBC -> Java type mappings via custom field conversions.

4.4.3. Persistence Broker Tutorial

4.4.3.1. The PersistenceBroker API

Introduction

The PersistenceBroker API provides the lowest level access to OJB's persistence engine. While it is a low-level API compared to the OTM, ODMG, or JDO API's it is still very straightforward to use.

The core class in the PersistenceBroker API is the `org.apache.ojb.broker.PersistenceBroker` class. This class provides the point of access for all persistence operations in this API.

This tutorial operates on a simple example class:

```
package org.apache.ojb.tutorials;

public class Product
{
    /* Instance Properties */

    private Double price;
    private Integer stock;
    private String name;lean

    /* artificial property used as primary key */

    private Integer id;

    /* Getters and Setters */
    ...
}
```

The metadata descriptor for mapping this class is described in the mapping tutorial

The source code for this tutorial is available with the source distribution of OJB in the `src/test/org/apache/ojb/tutorials/` directory.

A First Look - Persisting New Objects

The most basic operation is to persist an object. This is handled very easily by just

1. obtaining a `PersistenceBroker`
2. begin the PB-transaction
3. storing the object via the `PersistenceBroker`
4. commit transaction
5. closing the `PersistenceBroker`

For example, the following function stores a single object of type `Product`.

```
public static void storeProduct(Product product)
{
    PersistenceBroker broker = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        broker.beginTransaction();
        broker.store(product);
        broker.commitTransaction();
    }
    catch(PersistenceBrokerException e)
    {
        if(broker != null) broker.abortTransaction();
        // do more exception handling
    }
    finally
    {
        if (broker != null) broker.close();
    }
}
```

Two OJB classes are used here, the `PersistenceBrokerFactory` and the `PersistenceBroker`. The `PersistenceBrokerFactory` class manages the lifecycles of `PersistenceBroker` instances: it creates them, pools them, and destroys them as needed. The exact behavior is very configurable.

In this case we used the static `PersistenceBrokerFactory.defaultPersistenceBroker()` method to obtain an instance of a `PersistenceBroker` to the default data source. This is most often how it is used if there is only one database for an application. If there are multiple data sources, a broker may be obtained by name (using a `PBKey` instance as argument in `PersistenceBrokerFactory.createPersistenceBroker(pbKey)`).

It is worth noting that the `broker.close()` call is made within a `finally {...}` block. This ensures that the broker will be closed, and returned to the broker pool, even if the function throws an exception.

To use this function, we just create a `Product` and pass it to the function:

```
Product product = new Product();
product.setName("Sprocket");
product.setPrice(1.99);
product.setStock(10);
storeProduct(product);
```

Once a `PersistenceBroker` has been obtained, its `PersistenceBroker.store(Object)` method is used to make an object persistent.

Maybe you have noticed that there has not been an assignment to `product.id`, the primary-key attribute. Upon storing `product` OJB detects that the attribute is not properly set and assigns a unique id. This automatic assignment of unique Ids for the attribute `id` has been explicitly declared in the XML repository file, as we discussed in the .

If several objects need to be stored, this can be done within a transaction, as follows.

```
public static void storeProducts(Product[] products)
{
    PersistenceBroker broker = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        broker.beginTransaction();
        for (int i = 0; i < products.length; i++)
        {
            broker.store(products[i]);
        }
        broker.commitTransaction();
    }
    catch(PersistenceBrokerException e)
    {
        if(broker != null) broker.abortTransaction();
        // do more exception handling
    }
    finally
    {
        if (broker != null) broker.close();
    }
}
```

This contrived example stores all of the passed Product instances within a single transaction via the `PersistenceBroker.beginTransaction()` and `PersistenceBroker.commitTransaction()`. These are database level transactions, not object level transactions.

Querying Persistent Objects

Once objects have been stored to the database, it is important to be able to get them back. The PersistenceBroker API provides two mechanisms for building queries, by using a template object, or by using specific criteria.

```
public static Product findByTemplate(Product template)
{
    PersistenceBroker broker = null;
    Product result = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        QueryByCriteria query = new QueryByCriteria(template);
        result = (Product) broker.getObjectByQuery(query);
    }
    finally
    {
        if (broker != null) broker.close();
    }
    return result;
}
```

This function finds a `Product` by building a query against a template `Product`. The template should have any properties set which should be matched by the query. Building on the previous example where a product was stored, we can now query for that same product:

```
Product product = new Product();
product.setName("Sprocket");
product.setPrice(new Double(1.99));
product.setStock(new Integer(10));
storeProduct(product);

Product template = new Product();
template.setName("Sprocket");
Product sameProduct = findByTemplate(template);
```

In the above code snippet, `product` and `sameProduct` will reference the same object (assuming there are no additional products in the database with the name "Sprocket").

The template `Product` has only one of its properties set, the `name` property. The others are all null. Properties with null values are not used to match.

An alternate, and more flexible, way to have specified a query via the PersistenceBroker API is by constructing the criteria on the query by hand. The following function does this.

```
public static Collection getExpensiveLowStockProducts()
{
    PersistenceBroker broker = null;
    Collection results = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();

        Criteria criteria = new Criteria();
        criteria.addLessOrEqualThan("stock", new Integer(20));
        criteria.addGreaterOrEqualThan("price", new Double(100000.0));

        QueryByCriteria query = new QueryByCriteria(Product.class, criteria);
        results = broker.getCollectionByQuery(query);
    }
    finally
    {
        if (broker != null) broker.close();
    }
    return results;
}
```

This function builds a `Criteria` object and uses it to set more complex query parameters - in this case greater-than and less-than contraints. Looking at the first constraint put on the criteria, `criteria.addLessOrEqualThan("stock", new Integer(10));` notice the arguments. The first is the property name on the object being queried for. The second is an `Integer` instance to be used for the comparison.

After the `Criteria` has been built, the `QueryByCriteria` constructor used is also different from the previous example. In this case the criteria does not know the type of the object it is being used against, so the `Class` must be specified to the query.

Finally, notice that this example uses the `PersistenceBroker.getCollectionByQuery(...)` method instead of the `PersistenceBroker.getObjectByQuery(...)` method used previously. This is used because we want all of the results. Either form can be used with either method of constructing queries. In the case of the `PersistenceBroker.getObjectByQuery(...)` style query, the first matching object is returned, even if there are multiple matching objects.

Updating Persistent Objects

The same mechanism, and method, is used for updating persistent objects as for inserting persistent objects. The same `PersistenceBroker.store(Object)` method is used to store a modified object as to insert a new one - the difference between new and modified objects is irrelevent to OJB.

This can cause some confusion for people who are very used to working in the stricter confines of SQL inserts and updates. Basically, OJB will insert a new object into the relational store if the primary key, as specified in the O/R metadata is not in use. If it is in use, it will update the existing object rather than create a new one.

This allows programmers to treat every object the same way in an object model, whether it has been newly created and made persistent, or materialized from the database.

Typically, making changes to a peristent object first requires retrieving a reference to the object, so the typical update cycle, unless the application caches objects, is to query for the object to modify, modify the object, and then store the object. The following function demonstrates this behavior by "selling" a Product.

```
public static boolean sellOneProduct(Product template)
{
    PersistenceBroker broker = null;
    boolean isSold = false;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        QueryByCriteria query = new QueryByCriteria(template);
        Product result = (Product) broker.getObjectByQuery(query);

        if (result != null)
        {
            broker.beginTransaction();
            result.setStock(new Integer(result.getStock().intValue() - 1));
            broker.store(result);
            // alternative, more performant
            // broker.store(result, ObjectModificationDefaultImpl.UPDATE);
            broker.commitTransaction();
            isSold = true;
        }
    }
    catch(PersistenceBrokerException e)
    {
        if(broker != null) broker.abortTransaction();
        // do more exception handling
    }
    finally
    {
        if (broker != null) broker.close();
    }
    return isSold;
}
```

This function uses the same query-by-template and `PersistenceBroker.store()` API's examined previously, but it uses the store method to store changes to the object it retrieved. It is worth noting that the entire operation took place within a transaction.

Deleting Persistent Objects

Deleting persistent objects from the repository is accomplished via the `PersistenceBroker.delete()` method. This removes the persistent object from the repository, but does not affect any change on the object itself. For example:

```
public static void deleteProduct(Product product)
{
    PersistenceBroker broker = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        broker.beginTransaction();
        broker.delete(product);
        broker.commitTransaction();
    }
    catch(PersistenceBrokerException e)
    {
        if(broker != null) broker.abortTransaction();
        // do more exception handling
    }
    finally
    {
        if (broker != null) broker.close();
    }
}
```

This method simply deletes an object from the database.

4.4.3.2. Notes on Using the PersistenceBroker API

Pooling PersistenceBrokers

The `PersistenceBrokerFactory` pools `PersistenceBroker` instances. Using the `PersistenceBroker.close()` method releases the broker back to the pool under the default implementation. For this reason the examples in this tutorial all retrieve, use, and close a new broker for each logical transaction.

Transactions

Transactions in the PeristenceBroker API are database level transactions. This differs from object level transactions. The broker does not maintain a collection of modified, created, or deleted objects until a commit is called -- it operates on the database using the databases transaction mechanism. If object level transactions are required, one ofthe higher level API's (ODMG, JDO, or OTM) should be used.

Exception Handling

Most `PersistenceBroker` operations throw a `org.apache.ojb.broker.PersistenceBrokerException`, which is derived from `java.lang.RuntimeException` if an error occurs. This means that no try/catch block is **required** but does not mean that it should not be used. This tutorial specifically does not catch exceptions all in order to focus more tightly on the specifics of the API, however, best usage would be to include a try/catch/finally block around persistence operations using the PeristenceBroker API.

Additionally, the closing of `PersistenceBroker` instances is best handled in `finally` blocks in order to guarantee that it is run, even if an exception occurs. If the `PersistenceBroker.close()` is not called then the application will leak broker instances. The best way to ensure that it is always called is to always retrieve and use `PersistenceBroker` instances within a `try {...}` block, and always close the broker in a `finally {...}` block attached to the `try {...}` block.

A better designed `getExpensiveLowStockProducts()` method is presented here.

```
public static Collection betterGetExpensiveLowStockProducts()
{
    PersistenceBroker broker = null;
    Collection results = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();

        Criteria criteria = new Criteria();
        criteria.addLessOrEqualThan("stock", new Integer(20));
        criteria.addGreaterOrEqualThan("price", new Double(100000.0));

        QueryByCriteria query = new QueryByCriteria(Product.class, criteria);
        results = broker.getCollectionByQuery(query);
    }
    catch (PersistenceBrokerException e)
    {
        // Handle exception
    }
    finally
    {
        if (broker != null) broker.close();
    }
    return results;
}
```

Notice first that the `PersistenceBroker` is retrieved and used within the confines of a `try {...}` block. Assuming nothing goes wrong the entire operation will execute there, all the way to the `return results;` line. Java guarantees that `finally {...}` blocks will be called before a method returns, so the `broker.close()` method is only included once, in the `finally` block. As an exception may have occured while attempting to retrieve the broker, a not-null test is first performed before closing the broker.

4.4.4. The ODMG API

4.4.4.1. Introduction

The ODMG API is an implementation of the ODMG 3.0 Object Persistence API. The ODMG API provides a higher-level API and query language based interface over the PersistenceBroker API.

This tutorial operates on a simple example class:

```
package org.apache.ojb.tutorials;

public class Product
{
/* Instance Properties */

private Double price;
private Integer stock;
private String name;

/* artificial property used as primary key */

private Integer id;

/* Getters and Setters */
...
}
```

The metadata descriptor for mapping this class is described in the mapping tutorial

The source code for this tutorial is available with the source distribution of OJB in the `src/test/org/apache/ojb/tutorials/` directory.

4.4.4.2. Initializing ODMG

The ODMG implementation needs to have a database opened for it to access. This is accomplished via the following code:

```
Implementation odmg = OJB.getInstance();
Database db = odmg.newDatabase();
db.open("default", Database.OPEN_READ_WRITE);

/* ... use the database ... */

db.close();
```

This opens an ODMG `Database` using the name specified in metadata for the database -- "default" in this case. Notice the `Database` is opened in read/write mode. It is possible to open it in read-only or write-only modes as well.

Once a `Database` has been opened it is available for use. Unlike `PersistenceBroker` instances, ODMG `Database` instances are threadsafe and can typically be used for the entire lifecycle of an application. There is no need to call the `Database.close()` method until the database is truly no longer needed.

4.4.4.3. Persisting New Objects

Persisting an object via the ODMG API is handled by writing it to the peristence store within the context of a transaction:

```
public static void storeProduct(Product product)
{
```

```
Implementation impl = OJB.getInstance();
Transaction tx = impl.newTransaction();
tx.begin();
tx.lock(product, Transaction.WRITE);
tx.commit();
}
```

The `OJB.getInstance()` function provides the ODMG `Implementation` instance required for using the ODMG API. From here on out it is straight ODMG code that should work against any compliant ODMG implementation.

Once the ODMG implementation has been obtained it is used to begin a transaction, obtain a write lock on the `Product`, and commit the transaction. It is very important to note that all changes need to be made within transactions in the ODMG API. When the transaction is committed the changes are made to the database. Until the transaction is committed the database is unaware of any changes -- they exist solely in the object model.

4.4.4.4. Querying Persistent Objects

The ODMG API uses the OQL query language for obtaining references to persistent objects. OQL is very similar to SQL, and using it is very similar to use JDBC. The ODMG implementation is used to create a query, the query is specifed, executed, and a list fo results is returned:

```
public static Product findProductByName(String name) throws Exception
{
Implementation impl = OJB.getInstance();
Transaction tx = impl.newTransaction();
tx.begin();

OQLQuery query = impl.newOQLQuery();
query.create("select products from "
             + Product.class.getName()
             + " where name = $1");
query.bind(name);
DList results = (DList) query.execute();
Product product = (Product) results.iterator().next();

tx.commit();
return product;
}
```

4.4.4.5. Updating Persistent Objects

Updating a persistent object is done by modifying it in the context of a transaction, and then committing the transaction:

```
public static void sellProduct(Product product, int number)
{
Implementation impl = OJB.getInstance();
Transaction tx = impl.newTransaction();
tx.begin();

tx.lock(product, Transaction.WRITE);
product.setStock(new Integer(product.getStock().intValue() -  number));

tx.commit();
}
```

The sample code obtains a write lock on the object, binding it to the transaction, changes the object, and commits the transaction. The newly modified `Product` now has a new `stock` value.

4.4.4.6. Deleting Persistent Objects

Deleting persistent objects requires directly addressing the `Database` which contains the persistent object. This can be obtained from the ODMG `Implementation` by asking for it. Once retrieved, just ask the `Database` to delete the object. Once again, this is all done in the context of a transaction.

```
public static void deleteProduct(Product product)
{
Implementation impl = OJB.getInstance();
Transaction tx = impl.newTransaction();

tx.begin();
Database db = impl.getDatabase(product);
db.deletePersistent(product);
tx.commit();
}
```

It is important to note that the `Database.deletePerstient()` call does not delete the object itself, just the persistent representation of it. The transient object still exists and can be used however desired -- it is simply no longer persistent.

4.4.4.7. Notes on Using the ODMG API

Transactions

The ODMG API uses object-level transactions, compared to the PersistenceBroker database-level transactions. An ODMG `Transaction` instance contains all of the changes made to the object model within the context of that transaction, and will not commit them to the database until the ODMG `Transaction` is committed. At that point it will use a database transaction to ensure atomicity of its changes.

Locks

The ODMG specification includes several levels of locks and isolation. These are explained in much more detail in the Lock Manager documentation.

In the ODMG API, locks obtained on objects are locked within the context of a thread. Any object modified within the context of a transaction will be stored with the transaction, however changes made to the same object by other threads will also be stored. The ODMG locking conventions ensure that an object can only be modified within the transaction on the locking thread.

Persisting Non-Transactional Objects

Frequently, objects will be modified outside of the context of an ODMG transaction, such as a data access object in a web application. In those cases a persistent object can still be modified, but not directly through the OMG ODMG specification. OJB provides an extension to the ODMG specification for instances such as this. Examine this code:

```
public static void persistChanges(Product product)
{
    Implementation impl = OJB.getInstance();
    TransactionExt tx = (TransactionExt) impl.newTransaction();

    tx.begin();
    tx.markDirty(product);
    tx.commit();
}
```

In this function the product is modified outside the context of the transaction, and is then the changes are persisted within a transaction. The `TransactionExt.markDirty()` method indicates to the Transaction that the passed object has been modified, even if the Transaction itself sees no changes to the object.

4.4.5. JDO Tutorial

4.4.5.1. Using the ObJectRelationalBridge JDO API

Introduction

This document demonstrates how to use ObjectRelationalBridge and the JDO API in a simple application scenario. The tutorial application implements a product catalog database with some basic use cases. The source code for the tutorial application is shipped with the OJB source distribution and resides in the directory **`[db-ojb]/src/jdori/org/apache/ojb/tutorial5`**.

This document is not meant as a complete introduction to JDO. For more information see: Sun's JDO site.

> **Note:**
> OJB does not provide it's own JDO implementation yet. A full JDO implementation is in the scope of the 2.0 release.
> For the time being we provide a plugin to the JDO reference implementation called `OjbStore`. The `OjbStore` plugin resides in the package `org.apache.ojb.jdori.sql`.
> The work on the *native OJB-JDO* implementation has started. A first beta version is announce for OJB 1.1 version.

Running the Tutorial Application

To install and run the demo application please follow the following steps:

1. Download the JDO Reference Implementation from Sun's JDO site.
   Extract the archiv to a local directory and copy the files:
   - `jdori.jar`
   - `jdo.jar`

   into the OJB **`[db-ojb]/lib`** directory.
2. Now compile the sources, setup the test database and perform bytecode enhancement by executing
   `ant with-jdori prepare-tutorials enhance-jdori`
   from the ojb toplevel directory.
3. Now you can start the tutorial application by executing `bin\tutorial5` or `bin/tutorial5.sh` from the ojb toplevel directory.

4.4.5.2. Using the JDO API in the UseCase Implementations

As shown here OJB supports four different API's. The PersistenceBroker, the OTM layer, the ODMG implementation, and the JDO implementation.

The PB tutorial implemented the sample application's use cases with the PersistenceBroker API. This tutorial will show how the same use cases can be implemented using the JDO API.

You can get more information about the JDO API at JDO javadocs.

Obtaining the JDO PersistenceManager Object

In order to access the functionalities of the JDO API you have to deal with a special facade object that serves as the main entry point to all JDO operations. This facade is specified by the Interface `javax.jdo.PersistenceManager`.

A Vendor of a JDO compliant product must provide a specific implementation of the `javax.jdo.PersistenceManager` interface. JDO also specifies that a JDO implementation must provide a `javax.jdo.PersistenceManagerFactory` implementation that is responsible for generating `javax.jdo.PersistenceManager` instances.

So if you know how to use the JDO API you only have to learn how to obtain the OJB specific PersistenceManagerFactory object. Ideally this will be the only vendor specific operation.

In our tutorial application the `PersistenceManagerFactory` object is obtained in the constructor of the Application class and reached to the use case implementations for further usage:

```
public Application()
{
    factory = null;
    manager = null;
    try
    {
        // create OJB specific factory:
        factory = new OjbStorePMF();
    }
    catch (Throwable t)
    {
        System.out.println("ERROR: " + t.getMessage());
        t.printStackTrace();
    }
    useCases = new Vector();
    useCases.add(new UCListAllProducts(factory));
    useCases.add(new UCEnterNewProduct(factory));
    useCases.add(new UCEditProduct(factory));
    useCases.add(new UCDeleteProduct(factory));
    useCases.add(new UCQuitApplication(factory));
}
```

The class `org.apache.ojb.jdori.sql.OjbStorePMF` is the OJB specific `javax.jdo.PersistenceManagerFactory` implementation.

########### TODO: Put information about the .jdo files #############

The `PersistenceManagerFactory` object is reached to the constructors of the UseCases. These constructors store it in a protected attribute `factory` for further usage.

Retrieving collections

The next thing we need to know is how this Implementation instance integrates into our persistence operations.

In the use case `UCListAllProducts` we have to retrieve a collection containing all product entries from the persistent store. To retrieve a collection containing objects matching some criteria we can use the JDOQL query language as specified by the JDO spec. In our use case we want to select *all* persistent instances of the class Products. In this case the query is quite simple as it does not need any limiting search criteria.

We use the factory to create a PersistenceManager instance in step one. In the second step we ask the PersistenceManager to create a query returning all Product instances.

In the third step we perform the query and collect the results in a collection.

In the fourth step we iterate through the collection to print out each product matching our query.

```
public void apply()
{
    // 1. get a PersistenceManager instance
    PersistenceManager manager = factory.getPersistenceManager();
    System.out.println("The list of available products:");

    try
    {
        // clear cache to provoke query against database
        PersistenceBrokerFactory.
                    defaultPersistenceBroker().clearCache();

        // 2. start tx and form query
        manager.currentTransaction().begin();
        Query query = manager.newQuery(Product.class);

        // 3. perform query
        Collection allProducts = (Collection)query.execute();
        // 4. now iterate over the result to print each
        // product and finish tx
        java.util.Iterator iter = allProducts.iterator();
        if (! iter.hasNext())
        {
            System.out.println("No Product entries found!");
        }
        while (iter.hasNext())
        {
            System.out.println(iter.next());
        }
        manager.currentTransaction().commit();
    }
    catch (Throwable t)
    {
        t.printStackTrace();
    }
    finally
    {
        manager.close();
    }
}
```

Storing objects

Now we will have a look at the use case `UCEnterNewProduct`. It works as follows: first create a new object, then ask the user for the new product's data (productname, price and available stock). These data is stored in the new object's attributes. This part is no different from the PB tutorial implementation. (Steps 1. and 2.)

Now we will store the newly created object in the persistent store by means of the JDO API. With JDO, all persistence operations must happen within a transaction. So the third step is to ask the PersistenceManager object for a fresh `javax.jdo.Transaction` object to work with. The `begin()` method starts the transaction.

We then have to ask the PersistenceManager to make the object persistent in step 4.

In the last step we commit the transaction. All changes to objects touched by the transaction are now made persistent. As you will have noticed there is no need to explicitly store objects as with the PersistenceBroker API. The Transaction object is responsible for tracking which objects have been modified and to choose the appropriate persistence operation on commit.

```
public void apply()
{
    // 1. this will be our new object
    Product newProduct = new Product();
    // 2. now read in all relevant information and fill the new object:
    System.out.println("please enter a new product");
    String in = readLineWithMessage("enter name:");
    newProduct.setName(in);
    in = readLineWithMessage("enter price:");
    newProduct.setPrice(Double.parseDouble(in));
    in = readLineWithMessage("enter available stock:");
    newProduct.setStock(Integer.parseInt(in));

    // 3. create PersistenceManager and start transaction
    PersistenceManager manager = factory.getPersistenceManager();

    Transaction tx = null;
    tx = manager.currentTransaction();
    tx.begin();

    // 4. mark object as persistent
    manager.makePersistent(newProduct);

    // 5. commit transaction
    tx.commit();
    manager.close();
}
```

Updating Objects

The UseCase `UCEditProduct` allows the user to select one of the existing products and to edit it.

The user enters the products unique id. The object to be edited is looked up by this id. (Steps 1., 2. and 3.) This lookup is necessary as our application does not hold a list of all product objects.

The product is then edited (Step 4.).

In step five the transaction is commited. All changes to objects touched by the transaction are now made persistent. Because we modified an existing object an update operation is performed against the backend database.

```
public void apply()
{
    PersistenceManager manager = null;

    // ask user which object should edited
    String in = readLineWithMessage("Edit Product with id:");
    int id = Integer.parseInt(in);

    Product toBeEdited;
    try
    {
        // 1. start transaction
        manager = factory.getPersistenceManager();
        manager.currentTransaction().begin();

        // We don't have a reference to the selected Product.
        // So we have to look it up first,


        // 2. Build a query to look up product by the id
        Query query = manager.newQuery(Product.class, "id == " + id);

        // 3. execute query
        Collection result = (Collection) query.execute();
        toBeEdited = (Product) result.iterator().next();

        if (toBeEdited == null)
        {
            System.out.println("did not find a matching instance...");
            manager.currentTransaction().rollback();
            return;
        }

        // 4. edit the existing entry
        System.out.println("please edit the product entry");
        in =
            readLineWithMessage(
                "enter name (was " + toBeEdited.getName() + "):");
        toBeEdited.setName(in);
        in =
            readLineWithMessage(
                "enter price (was " + toBeEdited.getPrice() + "):");
        toBeEdited.setPrice(Double.parseDouble(in));
        in =
            readLineWithMessage(
                "enter available stock (was "
                    + toBeEdited.getStock()
                    + "):");
        toBeEdited.setStock(Integer.parseInt(in));

        // 5. commit changes
        manager.currentTransaction().commit();
    }
    catch (Throwable t)
    {
        // rollback in case of errors
        manager.currentTransaction().rollback();
        t.printStackTrace();
    }
    finally
    {
        manager.close();
    }
}
```

Deleting Objects

The UseCase `UCDeleteProduct` allows the user to select one of the existing products and to delete it from the persistent storage.

The user enters the products unique id. The object to be deleted is looked up by this id. (Steps 1., 2. and 3.) This lookup is necessary as our application does not hold a list of all product objects.

In the fourth step we check if a Product matching to the id could be found. If no entry is found we print a message and quit the work.

If a Product entry was found we delete it in step 5 by calling the PersistenceManager to delete the persistent object. On transaction commit all changes to objects touched by the transaction are made persistent. Because we marked the Product entry for deletion, a delete operation is performed against the backend database.

```
public void apply()
{
    PersistenceManager manager = null;
    Transaction tx = null;
    String in = readLineWithMessage("Delete Product with id:");
    int id = Integer.parseInt(in);

    try
    {
        // 1. start transaction
        manager = factory.getPersistenceManager();
        tx = manager.currentTransaction();
        tx.begin();

        // 2. Build a query to look up product by the id
        Query query = manager.newQuery(Product.class, "id == " + id);

        // 3. execute query
        Collection result = (Collection) query.execute();

        // 4. if no matching product was found, print a message
        if (result.size() == 0)
        {
            System.out.println("did not find a Product with id=" + id);
            tx.rollback();
            manager.close();
            return;
        }
        // 5. if a matching product was found, delete it
        else
        {
            Product toBeDeleted = (Product) result.iterator().next();
            manager.deletePersistent(toBeDeleted);
            tx.commit();
            manager.close();
        }
    }
    catch (Throwable t)
    {
        // rollback in case of errors
        //broker.abortTransaction();
        tx.rollback();
        t.printStackTrace();
    }
}
```

4.4.5.3. Conclusion

In this tutorial you learned to use the standard JDO API as implemented by the OJB system within a simple application scenario. I hope you found this tutorial helpful. Any comments are welcome.

4.4.6. Object Transaction Manager Tutorial

4.4.6.1. The OTM API

Introduction

The Object Transaction Manager (OTM) is written as a tool on which to implement other high-level object persistence APIs. It is, however, very usable directly. It supports API's similar to the ODMG and PersistenceBroker API's in OJB. Several of its idioms are designed around the fact that it is meant to have additional, client-oriented, API's built on top of it, however.

The `OTMKit` is the initial access point to the OTM interfaces. The kit provides basic configuration information to the OTM components used in your system. This tutorial will use the `SimpleKit` which will work well under most circumstances for local transaction implementations.

This tutorial operates on a simple example class:

```
package org.apache.ojb.tutorials;

public class Product
{
    /* Instance Properties */

    private Double price;
    private Integer stock;
    private String name;

    /* artificial property used as primary key */

    private Integer id;

    /* Getters and Setters */
    ...
}
```

The metadata descriptor for mapping this class is described in the mapping tutorial.

The source code for this tutorial is available with the source distribution of OJB in the `src/test/org/apache/ojb/tutorials/` directory.

Persisting New Objects

The starting point for using the OTM directly is to look at making a transient object persistent. This code will use three things, an `OTMKit`, an `OTMConnection`, and a `Transaction`. The connection and transaction objects are obtained from the kit.

Initial access to the OTM client API's is through the `OTMKit` interface. We'll use the `SimpleKit`, an implementation of the `OTMkit` suitable for most circumstances using local transactions.

```
public static void storeProduct(Product product) throws LockingException
```

```
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        tx.begin();
        conn.makePersistent(product);
        tx.commit();
    }
    catch (LockingException e)
    {
        if (tx.isInProgress()) tx.rollback();
        throw e;
    }
    finally
    {
        conn.close();
    }
}
```

A kit is obtained and is used to obtain a connection. Connections are against a specific JCD alias. In this case we use the default, but a named datasource could also be used, as configured in the metadata repository. A transaction is obtained from the kit for the specific connection. Because multiple connections can be bound to the same transaction in the OTM, the transaction needs to be acquired from the kit instead of the connection itself. The `SimpleKit` uses the commonly seen transaction-per-thread idiom, but other kits do not need to do this.

Every persistence operation within the OTM needs to be executed within the context of a transaction. The JDBC concept of implicit transactions doesn't exist in the OTM -- transactions must be explicit.

Locks, on the other hand, are implicit in the OTM (though explicit locks are available). The `conn.makePersistent(..)` call obtains a write lock on `product` and will commit (insert) the object when the transaction is committed.

The `LockingException` will be thrown if the object cannot be write-locked in this transaction. As it is a transient object to begin with, this will probably only ever happen if it has been write-locked in another transaction already -- but this depends on the transaction semantics configured in the repository metadata.

Finally, connections maintain resources so it is important to make sure they are closed when no longer needed.

Deleting Persistent Objects

Deleting a persistent object from the backing store (making a persistent object transient) is almost identical to making it persistent -- the difference is just in the `conn.deletePersistent(product)` call instead of the `conn.makePersistent(product)` call. The same notes about transactions and resources apply here.

```
public static void storeProduct(Product product) throws LockingException
{
   OTMKit kit = SimpleKit.getInstance();
   OTMConnection conn = null;
   Transaction tx = null;
   try
   {
       conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
       tx = kit.getTransaction(conn);
       tx.begin();
       conn.deletePersistent(product);
       tx.commit();
   }
   catch (LockingException e)
   {
       if (tx.isInProgress()) tx.rollback();
```

```
        throw e;
    }
    finally
    {
        conn.close();
    }
}
```

Querying for Objects

The OTM implements a transaction system, not a new client API. As such it supports two styles of query at present -- an PersistenceBroker like query-by-criteria style querying system, and an ODMG OQL query system.

Information on constructing these types of queries is available in the PersistenceBroker and ODMG tutorials respectively. Using those queries with the OTM is examined here.

A PB style query can be handled as follows:

```
public Iterator findByCriteria(Query query)
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        tx.begin();
        Iterator results = conn.getIteratorByQuery(query);
        tx.commit();
        return results;
    }
    finally
    {
        conn.close();
    }
}
```

Where, by default, a read lock is obtained on the returned objects. If a different lock is required it may be specified specifically:

```
public Iterator findByCriteriaWithLock(Query query, int lock)
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        tx.begin();
        Iterator results = conn.getIteratorByQuery(query, lock);
        tx.commit();
        return results;
    }
    finally
    {
        conn.close();
    }
}
```

The int `lock` argument is one of the integer constants on `org.apache.ojb.otm.lock.LockType`:

```
LockType.NO_LOCK
LockType.READ_LOCK
LockType.WRITE_LOCK
```

OQL queries are also supported, as this somewhat more complex example demonstrates:

```
public Iterator findByOQL(String query, Object[] bindings) throws Exception
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        OQLQuery oql = conn.newOQLQuery();
        oql.create(query);
        for (int i = 0; i < bindings.length; ++i)
        {
            oql.bind(bindings[i]);
        }
        tx.begin();
        Iterator results = conn.getIteratorByOQLQuery(oql);
        tx.commit();
        return results;
    }
    catch (QueryInvalidException e)
    {
        if (tx.isInProgress()) tx.rollback();
        throw new Exception("Invalid OQL expression given", e);
    }
    catch (QueryParameterCountInvalidException e)
    {
        if (tx.isInProgress()) tx.rollback();
        throw new Exception("Incorrect number of bindings given", e);
    }
    catch (QueryParameterTypeInvalidException e)
    {
        if (tx.isInProgress()) tx.rollback();
        throw new Exception("Incorrect type of object given as binding", e);
    }
    finally
    {
        conn.close();
    }
}
```

This function is, at its core, doing the same thing as the PB style queries, except that it constructs the OQL query, which supports binding values in a manner similar to JDBC prepared statements.

The OQL style queries also support specifying the lock level the same way:

```
Iterator results = conn.getIteratorByOQLQuery(query, lock);
```

More Sophisticated Transaction Handling

These examples are a bit simplistic as they begin and commit their transactions all in one go -- they are only good for retrieving data. More often data will need to be retrieved, used, and committed back.

Only changes to persistent objects made within the bounds of a transaction are persisted. This means that frequently a query will be executed within the bounds of an already established transaction, data will be changed on the objects obtained, and the transaction will then be committed back.

A very convenient way to handle transactions in many applications is to start a transaction and then let any downstream code be executed within the bounds of the transaction automatically. This is straightforward to do with the OTM using the `SimpleKit`! Take a look at a very slightly modified version of the query by criteria function:

```
public Iterator moreRealisticQueryByCriteria(Query query, int lock)
{
     OTMKit kit = SimpleKit.getInstance();
     OTMConnection conn = null;
     Transaction tx = null;
     try
     {
         conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
         tx = kit.getTransaction(conn);
         boolean auto = ! tx.isInProgress();
         if (auto) tx.begin();
         Iterator results = conn.getIteratorByQuery(query, lock);
         if (auto) tx.commit();
         return results;
     }
     finally
     {
         conn.close();
     }
}
```

In this case the function looks to see if a transaction is already in progress and sets a boolean flag if it is, `auto`. It then handles transactions itself, or allows the already opened transaction to maintain control.

Because connections can be attached to existing transactions the `SimpleKit` can attach the new connection to the already established transaction, allowing this function to work as expected whether there is a transaction in progress or not!

Client code using this function could then open a transaction, query for products, change them, and commit the changes back. For example:

```
public void renameWidgetExample()
{
    OTMKit kit = SimpleKit.getInstance();
    OTMConnection conn = null;
    Transaction tx = null;
    try
    {
        conn = kit.acquireConnection(PersistenceBrokerFactory.getDefaultKey());
        tx = kit.getTransaction(conn);
        tx.begin();
        Product sample = new Product();
        sample.setName("Wonder Widget");
        Query query = QueryFactory.newQueryByExample(sample);
        Iterator wonderWidgets
                    = moreRealisticQueryByCriteria(query, LockType.WRITE_LOCK);
        while (wonderWidgets.hasNext())
        {
            Product widget = (Product) wonderWidgets.next();
            widget.setName("Improved Wonder Widget");
        }
        tx.commit();
    }
    finally
    {
        conn.close();
    }
}
```

This sample renames a whole bunch of products from "Wonder Widget" to "Improved Wonder Widget" and stores them back. It must makes the changes within the context of the transaction it obtained for those changes to be stored back to the database. If the same iterator were obtained outside of a transaction, and the changes made, the changes would be made on the objects in memory, but not in the database. You can think of non-transaction objects as free immutable transfer objects.

This example also demonstrates two connections bound to the same transaction, as the `renameWidgetExample(...)` function obtains a connection, and the `moreRealisticQueryByCriteria(...)` function obtains an additional connection to the same transaction!

4.4.6.2. Notes on the Object Transaction Manager

Transactions

The Object Transaction Manager (OTM) is a transaction management layer for Java objects. It typically maps 1:1 to database transactions behind the scenes, but this is not actually required for the OTM to work correctly.

The OTM supports a wide range of transactional options, delimited in the LockManager documentation. While the lock manager is writte to the ODMG API, the same locking rules apply at the OTM layer.

4.5. Reference Guides

4.5.1. Reference Guides

4.5.1.1. Reference Guides

Here can be found a summary with a explanation of all reference guides.

- OJB Queries
  This document explains the usage of the query syntax.
- Basic O/R Technique
  This tutorial explains basic object-relational mapping technique in OJB like 1:1, 1:n and m:n relations, the auto-xxx settings for references and proxy objects/collections.
- Platforms
  What OJB requires from relational databases, and how to let it know which database to use.
- Logging configuration
  Details how to configure the logging within OJB.
- OJB.properties configuration
  The details on how to modify OJB's behaviour. This includes changing pluggable components.
- JDBC Types
  This document explains the standard mapping of JDBC types to Java classes.
- Repository Metadata
  The specific details of OJB metadata.
- Advanced O/R Technique
  This document explains some advanced O/R techniques like Polymorphism and "OJB Extents", Mapping Inheritance Hierarchies, Nested Objects and so on.
- Metadata Handling
  This document explains how the metadata xml file work and how the metadata information can be modified at runtime.
- Deployment
  Specifics on what is required to deploy OJB, including deployment to EJB containers.
- Connection Handling
  This document explains how OJB handles the `Connection` instances and how User's can step in.
- The Object Cache
  Documentation on the different object caching implementations included with OJB.
- The Sequence Manager
  How to use different sequence management strategies with OJB.
- The Lock Manager
  The ODMG API supports different lock management systems. This document explains the differences and how to make use of them.
- OJB XDoclet Module
  Documentation for the OJB XDoclet module. The module can build mappings and schema.
- OJB Performance

A look at how OJB performs and how to use OJB's performance tests.

4.5.2. Platforms

4.5.2.1. how to use OJB with a specific relational database

OJB has been designed to smoothly integrate with any relational database that provides JDBC support. OJB can be configured to use only JDBC 1.0 API calls to avoid problems with restrictions of several JDBC drivers.
It uses a limited SQL subset to avoid problems with restrictions of certain RDBMS. This design allows to keep the OJB code generic and free from database specifics.

This document explains basic concepts and shows how OJB can be configured to run against a specific RDBMS.

4.5.2.2. Basic Concepts

OJB internal tables

For certain features OJB relies on several internal tables that must be present in the target rdbms to allow a proper functioning. If those features are not needed OJB can be safely run without any internal tables.

The following table lists all tables and their specific purpose.

Tablename	Purpose
`OJB_HL_SEQ`	Table for the high/low sequence manager. If the built-in OJB sequence manager is not used, this table is not needed.
`OJB_LOCKENTRY`	This table is used to store Object locks if the LockManager is run in distributed mode. Not needed in singlevm mode.
`OJB_NRM`	The "Named Roots Map". ODMG allows to bind persistent objects to an user defined name. The Named roots map is used to store these bindings. It has NAME (String of arbitrary length) as primary key and keeps the serialized OID of the persistent object in the field OID (String of arbitrary length). If bind() and lookup() are not used in client apps, this table is not needed
`OJB_DLIST`	The table used for the ODMG persistent DList collections. If ODMG DLists are not used, this table is not needed.
`OJB_DLIST_ENTRIES`	stores the entries of DLists (a wrapper to objects stored in the DList) If ODMG DLists are not used, this table is not needed.
`OJB_DSET`	The table used to store ODMG persistent DSET collections If ODMG DSets are not used, this table is not needed.
`OJB_DSET_ENTRIES`	This table stores the entries of DSets.

	If ODMG DSets are not used, this table is not needed.
`OJB_DMAP`	The table use to store the ODMG persistent DMap tables If ODMG DMaps are not used, this table is not needed.
`OJB_DMAP_ENTRIES`	The table containing the DMap entries. The Keys and Values of the map can be arbitrary persistent objects. If ODMG DMaps are not used, this table is not needed.

OJB uses Torque to create all required tables and data. Thus there is no SQL DDL file, but an XML file describing the tables in format readable by Torque. The Torque DDL information for the internal tables resides in the file `src/schema/ojbcore-schema.xml`.

The o/r mappings for these tables are contained in the file `repository_internal.xml`.

Tables for the regression testbed

It is recommended to run the OJB JUnit regression tests against your target database. Thus you will have to provide several more tables, filled with the proper testdata.

The DDL information for these tables resides in the file `src/schema/ojbtest-schema.xml`.

The testdata is defined in the file `src/schema/ojbtest-data.xml`.

The o/r mappings for these tables are contained in the file `repository_junit.xml`.

Tables for the tutorial applications

If you intend to run the OJB tutorial applications against your target database you will have to provide one extra table.

The DDL information for this table also resides in the file `src/schema/ojbtest-schema.xml`.

The testdata is also defined in the file `src/schema/ojbtest-data.xml`.

The o/r mappings for this table is contained in the file `repository_user.xml`.

4.5.2.3. The setup process

OJB provides a setup routine to generate the target database and to fill it with the required testdata. This routine is based on Torque scripts and is driven from the build.xml file. This section describes how to use it.

Selecting a platform profile

OJB ships with support for several popular database platforms. The target platform is selected by the switch `profile` in the file build.properties. You can choose one out of the predefined profiles:

```
# With the 'profile' property you can choose the RDBMS platform OJB is using
# implemented profiles:
#
profile=hsqldb
# use the mssqldb-JSQLConnect profile for Microsoft SQL Server and
# you will automatically JSQLConnect driver, from http://www.j-netdirect.com/
# MBAIRD: This is my driver of preference for MS SQL Server, I find the OEM'd
# MS driver to have some problems.
```

```
#profile=mssqldb-JSQLConnect
#profile=mssqldb-Opta2000
#profile=mssqldb-ms
#profile=mysql
#profile=db2
#profile=oracle
#profile=oracle9i
#profile=oracle9i-Seropto
#profile=msaccess
#profile=postgresql
#profile=informix
#profile=sybase
#profile=sapdb
#profile=maxdb
```

The profile switch activated in `build.properties` is used to select a profile file from the `profile` directory.
If you set `profile=db2`, then the file `profile/db2.profile` is selected.
This file is used by the Torque scripts to set platform specific properties and to perform platform specific SQL operations.

editing the profile to point to your target db

The platform specific file `profile/xxx.profile` contains lots of information used by Torque. You can ignore most of it. The only important part in this file is the section where the url to the target db is assembled, here is an snip of the DB2 profile:

```
# -------------------------------------------------------------------
#
#  D A T A B A S E  S E T T I N G S
#
# -------------------------------------------------------------------
# JDBC connection settings. This is used by the JDBCToXML task
# that will create an XML database schema from JDBC metadata.
# These settings are also used by the SQL Ant task to initialize
# your Turbine system with the generated SQL.
# -------------------------------------------------------------------

dbmsName = Db2
jdbcLevel = 1.0
urlProtocol = jdbc
urlSubprotocol = db2
urlDbalias = OJB

createDatabaseUrl = ${urlProtocol}:${urlSubprotocol}:${urlDbalias}
buildDatabaseUrl = ${urlProtocol}:${urlSubprotocol}:${urlDbalias}
databaseUrl = ${urlProtocol}:${urlSubprotocol}:${urlDbalias}
databaseDriver = COM.ibm.db2.jdbc.app.DB2Driver
databaseUser = admin
databasePassword = db2
databaseHost = 127.0.0.1
```

These settings result in a database URL `jdbc:db2:OJB`. If your production database is registered with the name `MY_PRODUCTION_DB` you have to edit the entry `urlDBalias` to:
`urlDbalias = MY_PRODUCTION_DB`.

In this section you can also set application user name and password. You can also enter a different jdbc driver class, to activate a different driver.

Before progressing, please check that the jdbc driver class, named in the `databaseDriver` entry is located on the classpath! You can either edit the global environment variable `CLASSPATH` or place the jdbc driver jar file into the `jakarta-ojb-xxx/lib` directory.

Executing the build script

Now everything should be prepared to launch the setup routine. This routine can be invoked by calling **`ant prepare-testdb`** .

If you are prompted with a `BUILD SUCCESSFUL` message after some time, everything is OK.

If you are prompted with a `BUILD FAILED` message after some time, something went wrong. This may have several reasons:

- You entered some incorrect settings. Please check the log messages to see what went wrong.
- Torque does not work properly against your target database. Torque is very flexible and should be able to work against a wide range of databases. But the code templates for each database may not be accurate. Please check the ojb-user mailinglist archive if there are any failure reports for your specific database. Please also check if some contributed a fix already. If you don't find anything please post your problem to the ojb user-list.

As a last resort you can try the following: Switch back to the default hsqldb profile and execute `ant prepare-testdb` This will setup the default hsqldb database. And it will also generate SQL scripts that you may use to generate your database manually.

The SQL scripts are generated to `jakarta-ojb-xxx/target/src/sql`. You can touch these scripts to match your database specifics and execute them manually against your platform.

Verifying the installation

Now everything is setup to run the junit regression tests against your target database.

Execute

```
ant junit
```

to see if everything works as expected. more information about the OJB Test Suite here. If you did not manage to set up the target database with the `ant prepare-testdb` you can use **`ant junit-no-compile-no-prepare`** to run the testsuite without generation of the test database.

4.5.3. OJB.properties Configuration File

4.5.3.1. OJB Configuration

OJB provides two different configuration mechanisms:

1. An XML based **`repository.xml`** is used to define the Object/Relational Mapping. This Mapping is translated into a metadata dictionary at runtime. The metadata layer may also be manipulated at runtime through OJB API calls. Follow this link to learn more about the XML repository.
2. A properties file **`OJB.properties`** that is responsible for the configuration of the OJB runtime environment. It contains information that does not change at runtime and does not contain O/R mapping related information.

The rest of this document details on this properties file.

4.5.3.2. OJB.properties File

By default this file is named **`OJB.properties`** and is loaded from the **classpath** by a J2EE compliant resource lookup:

```
Thread.currentThread().getContextClassLoader().getResource(getFilename());
```

The filename of the properties file can be changed by setting a Java system property. This can be done programmatically:

```
System.setProperty("OJB.properties","myOwnPropertiesFile.props");
```

or by setting a -D option to the JVM:

```
 java -DOJB.properties=myOwnPropertiesFile.props my.own.ojb.Application
```

All things that can be configured by OJB.properties are commented in the file itself. Have a look at the default version of this file.

4.5.4. JDBC Types

4.5.4.1. Mapping of JDBC Types to Java Types

OJB implements the mapping conventions for JDBC and Java types as specified by the JDBC 3.0 specification. See the table below for details.

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
REF	Ref
DATALINK	java.net.URL
JAVA_OBJECT	underlying Java class

4.5.4.2. Type and Value Conversions

Introduction

A typical problem with O/R tools is mismatching datatypes: a class from the domain model has an attribute of type boolean but the corresponding database table stores this attribute in a column of type bit or int.

This example explains how OJB allows you to define **FieldConversions** that do the proper translation of types and values.

The source code of this example is included in the OJB source distribution and resides in the test package `org.apache.ojb.broker`.

The problem

The test class `org.apache.ojb.broker.Article` contains an attribute `isSelloutArticle` of type boolean:

```
public class Article implements InterfaceArticle
{
    protected int articleId;
    protected String articleName;

    // maps to db-column Auslaufartikel of type int
    protected boolean isSelloutArticle;

    ...
}
```

The coresponding table uses an int column (`Auslaufartikel`) to store this attribute:

```
CREATE TABLE Artikel (
    Artikel_Nr         INT PRIMARY KEY,
    Artikelname        CHAR(60),
    Lieferanten_Nr     INT,
    Kategorie_Nr       INT,
    Liefereinheit      CHAR(30),
    Einzelpreis        DECIMAL,
    Lagerbestand       INT,
    BestellteEinheiten INT,
    MindestBestand     INT,
    Auslaufartikel     INT
  )
```

The Solution

OJB allows to use predefined (or self-written) FieldConversions that do the appropiate mapping. The `FieldConversion` interface declares two methods: `javaToSql(...)` and `sqlToJava(...)`:

```
/**
 * FieldConversion declares a protocol for type and value
 * conversions between persistent classes attributes and the columns
 * of the RDBMS.
 * The default implementation does not modify its input.
 * OJB users can use predefined implementation and can also
 * build their own conversions that perform arbitrary mappings.
 * the mapping has to defined in the xml repository
 * in the field-descriptor.
 *
 * @author Thomas Mahler
 */
public interface FieldConversion extends Serializable
{
    /**
     * convert a Java object to its SQL
     * pendant, used for insert & update
     */
    public abstract Object javaToSql(Object source) throws ConversionException;

    /**
     * convert a SQL value to a Java Object, used for SELECT
     */
    public abstract Object sqlToJava(Object source) throws ConversionException;
}
```

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 FieldConversion is declared for a persistent classes attributes, it uses the FieldConversion to do the marshalling of this attribute.

For the above mentioned problem of mapping an int column to a boolean attribute we can use the predefined FieldConversion `Boolean2IntFieldConversion`. Have a look at the code to see how it works:

```
public class Boolean2IntFieldConversion 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);

    /**
     * @see FieldConversion#javaToSql(Object)
     */
    public Object javaToSql(Object source)
    {
        if (source instanceof Boolean)
        {
            if (source.equals(B_TRUE))
            {
                return I_TRUE;
            }
            else
            {
                return I_FALSE;
            }
        }
        else
        {
            return source;
        }
    }

    /**
     * @see FieldConversion#sqlToJava(Object)
     */
    public Object sqlToJava(Object source)
    {
        if (source instanceof Integer)
        {
            if (source.equals(I_TRUE))
            {
                return B_TRUE;
            }
            else
            {
                return B_FALSE;
            }
        }
        else
        {
            return source;
        }
    }
}
```

There are other helpful standard conversions defined in the package `org.apache.ojb.broker.accesslayer.conversions`: Of course it is possible to map between `java.sql.date` and `java.util.date` by using a Conversion. A very interesting Conversion is the `Object2ByteArrFieldConversion` it allows to store inlined objects in varchar columns!

Coming back to our example, there is only one thing left to do: we must tell OJB to use the proper FieldConversion for the Article class. This is done in the XML Repository. The field-descriptor allows to define a `conversion` attribute declaring the fully qualified FieldConversion class:

```
<!-- Definitions for test.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" />

      ...

      <field-descriptor
         name="isSelloutArticle"
         column="Auslaufartikel"
         jdbc-type="INTEGER"
         conversion="org.apache.ojb.broker.accesslayer.
                        conversions.Boolean2IntFieldConversion"
      />

      ...

   </class-descriptor>
```

4.5.5. Repository File

4.5.5.1. Introduction - repository syntax

The syntax of the OJB repository xml files is defined by the *repository.dtd*.
The repository.dtd can be found here.

The actual repository metadta declaration is split up into several separate files, here is an excerpt of the most important files:

1. the repository.xml. Main file for metadata declaration. This file is split into several sub files using xml-Entity references.
2. the repository_database.xml. This file contains the mapping information for database/connection handling.
3. the repository_internal.xml. This file contains the mapping information for the OJB internal tables. These tables are used for implementing SequenceManagers and persistent collections.
4. the repository_user.xml. This file contains mappings for the tutorial applications and may be used to hold further user defined class mappings.
5. the repository_junit.xml. This file contains mapping information for common OJB JUnit regression test suite. In production environments these tables are not needed.
6. other repository_junit_XYZ.xml
   More specific junit test mapping. In production environments these tables are not needed.
7. There are some more files, for more information see comment in appropriate xml-file.

4.5.5.2. descriptor-repository

The *descriptor-repository* is the root element of a repository.xml file. It consists of one *jdbc-connection-descriptor* and at least one *class-descriptor* element.

Elements

```
<!ELEMENT descriptor-repository (documentation?, attribute*,
        jdbc-connection-descriptor*, class-descriptor*)>
```

The *documentation* element can be used to store arbitrary information.

The *attribute* element allows to add custom attributes, e.g. for passing arbitrary properties.

The *jdbc-connection-descriptor* element specifies a jdbc connection for the repository.

The *class-descriptor* element specify o/r mapping information for persistent class.

```
<!ELEMENT descriptor-repository (
    documentation?,
    attribute*,
    jdbc-connection-descriptor*,
    class-descriptor* )
>
```

Attributes

```
<!ATTLIST descriptor-repository
        version (1.0) #REQUIRED
        isolation-level (read-uncommitted | read-committed | repeatable-read |
                         serializable | optimistic) "read-uncommitted"
        proxy-prefetching-limit CDATA "50"
>
```

version

The *version* attribute is used to bind a repository.xml file to a given version of this dtd. A given OJB release will work properly only with the repository version shipped with that relase. This strictness maybe inconvenient but it does help to avoid the most common version conflicts.

isolation

The *isolation* attribute defines the default isolation level for *class-descriptor* that do not define a specific isolation level. This isolation level is used within the ODMG-api and does not touch the isolation-level off the database.

proxy-prefetching-limit

The *proxy-prefetching-limit* attribute specifies a default value to be applied to all proxy instances. If none is specified a default value of 50 is used. Proxy prefetching specifies how many instances of a proxied class should be loaded in a single query when the proxy is first accessed.

```
<!ATTLIST descriptor-repository
    version                 ( 1.0 ) #REQUIRED
    isolation-level         ( read-uncommitted |
                              read-committed   |
                              repeatable-read  |
                              serializable     |
                              optimistic ) "read-uncommitted"
    proxy-prefetching-limit CDATA "50"
>
```

4.5.5.3. jdbc-connection-descriptor

The *jdbc-connection-descriptor* element specifies a jdbc connection for the repository. It is allowed to define more than one *jdbc-connection-descriptor*. All *class-descriptor* elements are independent from the *jdbc-connection-descriptor*s. More info about connection handling here.

Elements

```
<!ELEMENT jdbc-connection-descriptor (documentation?, attribute*,
                object-cache?, connection-pool?, sequence-manager?)>
```

The *object-cache* element specifies the object-cache implementation class associated with this class.

A *connection-pool* element may be used to define connection pool properties for the specified JDBC connection.

Further a *sequence-manager* element may be used to define which sequence manager implementation should be used within the defined connection.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT jdbc-connection-descriptor (
    documentation?,
    attribute*,
    object-cache?,
    connection-pool?,
    sequence-manager? )
>
```

Attributes

The *jdbc-connection-descriptor* element contains a bunch of required and implied attributes:

```
<!ATTLIST jdbc-connection-descriptor
        jcd-alias CDATA #REQUIRED
    default-connection (true | false) "false"
    platform (Db2 | Hsqldb | Informix | MsAccess | MsSQLServer |
                  MySQL | Oracle | PostgreSQL | Sybase | SybaseASE |
              SybaseASA | Sapdb | Firebird | Axion | NonstopSql |
              Oracle9i | MaxDB ) "Hsqldb"
        jdbc-level (1.0 | 2.0 | 3.0) "1.0"
        eager-release (true | false) "false"
    batch-mode (true | false) "false"
    useAutoCommit (0 | 1 | 2) "1"
    ignoreAutoCommitExceptions (true | false) "false"

        jndi-datasource-name CDATA #IMPLIED

        driver CDATA #IMPLIED
        protocol CDATA #IMPLIED
        subprotocol CDATA #IMPLIED
        dbalias CDATA #IMPLIED

        username CDATA #IMPLIED
        password CDATA #IMPLIED
>
```

jdbcAlias

The *jcdAlias* attribute is a shortcut name for the defined connection descriptor. OJB uses the jcd alias as key for the defined connections.

default-connection

The *default-connection* attribute used to define if this connection should used as default connection with OJB. You could define only one connection as default connection. It is also possible to set the default connection at runtime using *PersistenceBrokerFactory#setDefaultKey(...)* method. If set *true* you can use a PB-api shortcut-method of the *PersistenceBrokerFactory* to lookup PersistenceBroker instances.

> **Note:**
> If *default-connection* was not set at runtime, it is mandatory that *username and password* is set in repository file.

platform

The *platform* attribute is used to define the specific RDBMS Platform. This attribute corresponds to a *org.apache.ojb.broker.platforms.PlatformXXXImpl* class. Supported databases see here. Default was *Hsqldb*.

jdbc-level

The *jdbc-level* attribute is used to specify the Jdbc compliance level of the used Jdbc driver. Allowed values are: *1.0*, *2.0*, *3.0*. Default was *1.0*.

eager-release

The *eager-release* attribute was adopt to solve a problem occured when using OJB within JBoss (3.0 <= version < 3.2.2, seems to be fixed in jboss 3.2.2 and higher). Only use within JBoss. *DEPRECATED* attribute.

batch-mode

The *batch-mode* attribute allow to enable JDBC connection batch support (if supported by used database), 'true' value allows to enable per-session batch mode, whereas 'false' prohibits it. *PB.serviceConnectionManager.setBatchMode(...)* method can be used to switch on/off batch modus, if batch-mode was enabled. On PB.close() OJB switch off batch modus, thus you have to do '...setBatchMode(true)' on each obtained PB instance again.

useAutoCommit

The *useAutoCommit* attribute allow to set how OJB uses the autoCommit state of the used connections. The default mode is 1. When using mode 0 or 2 with the PB-api, you must use PB transaction demarcation.

- 0 - OJB ignores the autoCommit setting of the connection and does not try to change it. This mode could be helpful if the connection won't let you set the autoCommit state (e.g. using datasources within an application server).
- 1 - Set autoCommit explicitly to *true* when a connection was created and temporary set to *false* when necessary (default mode).
- 2 - Set autoCommit explicitly to *false* when a connection was created.

ignoreAutoCommitExceptions

If the *ignoreAutoCommitExceptions* attribute is set to *true*, all exceptions caused by setting autocommit state, will be ignored. Default mode is *false*.

jndi-datasource-name

If a *jndi-datasource-name* for JNDI based lookup of Jdbc connections is specified, the four attributes *driver*, *protocol*, *subprotocol*, and *dbalias* used for Jdbc DriverManager based construction of Jdbc Connections must not be declared.

username

The *username* and *password* attributes are used as credentials for obtaining a jdbc connections.
If users don't want to keep user/password information in the repository.xml file, they can pass user/password using a *PBKey* to

obtain a PersistenceBroker. More info see FAQ.

4.5.5.4. connection-pool

The *connection-pool* element specifies the connection pooling parameter. More info about the connection handling can be found here.

```
<!ELEMENT connection-pool (documentation? )
>
```

Valid attributes for the *connection-pool* element are:

```
<!ATTLIST connection-pool
    maxActive                      CDATA #IMPLIED
    maxIdle                        CDATA #IMPLIED
    maxWait                        CDATA #IMPLIED
    minEvictableIdleTimeMillis     CDATA #IMPLIED
    numTestsPerEvictionRun         CDATA #IMPLIED
    testOnBorrow                   ( true | false ) #IMPLIED
    testOnReturn                   ( true | false ) #IMPLIED
    testWhileIdle                  ( true | false ) #IMPLIED
    timeBetweenEvictionRunsMillis  CDATA #IMPLIED
    whenExhaustedAction            ( 0 | 1 | 2 ) #IMPLIED
    validationQuery                CDATA #IMPLIED
    logAbandoned                   ( true | false ) #IMPLIED
    removeAbandoned                ( true | false ) #IMPLIED
    removeAbandonedTimeout         CDATA #IMPLIED
>
```

maxActive is the maximum number of connections that can be borrowed from the pool at one time. When non-positive, there is no limit.

maxIdle controls the maximum number of connections that can sit idle in the pool at any time. When non-positive, there is no limit

maxWait - the maximum time block to get connection instance from pool, after that exception is thrown. When non-positive, block till last judgement

whenExhaustedAction

- 0 - fail when pool is exhausted
- 1 - block when pool is exhausted
- 2 - grow when pool is exhausted

testOnBorrow when *true* the pool will attempt to validate each object before it is returned from the pool.

testOnReturn set to *true* will force the pool to attempt to validate each object before it is returned to the pool.

testWhileIdle indicates whether or not idle objects should be validated. Objects that fail to validate will be dropped from the pool.

timeBetweenEvictionRunsMillis indicates how long the eviction thread should sleep before "runs" of examining idle objects. When non-positive, no eviction thread will be launched.

minEvictableIdleTimeMillis specifies the minimum amount of time that a connection may sit idle in the pool before it is eligable for eviction due to idle time. When non-positive, no connection will be dropped from the pool due to idle time alone (depends on *timeBetweenEvictionRunsMillis* > 0)

numTestsPerEvictionRun - the number of connections to examine during each run of the idle object evictor thread (if any)

validationQuery allows to specify a validation query used by the ConnectionFactory implementations using connection pooling, to test a requested connection (e.g. "select 1 from dual") before leave the pool (used by *ConnectionFactoryDBCPImpl*

and *ConnectionFactoryPooledImpl*).
If not set, only *connection.isClosed()* will have been called before the connection was delivered.

logAbandoned is only supported when using *org.apache.ojb.broker.accesslayer.ConnectionFactoryDBCPImpl* ConnectionFactory implementation. Then it is a flag to log stack traces for application code which abandoned a Statement or Connection. Defaults to false. Logging of abandoned Statements and Connections adds overhead for every Connection open or new Statement because a stack trace has to be generated.
DEPRECATED attribute!

removeAbandoned and *removeAbandonedTimeout* When using *org.apache.ojb.broker.accesslayer.ConnectionFactoryDBCPImpl* ConnectionFactory implementation, the *removeAbandoned* flag controls the removal of abandoned connections if they exceed the *removeAbandonedTimeout*. Set to true or false, default false. If set to true a connection is considered abandoned and eligible for removal if it has been idle longer than the *removeAbandonedTimeout*. Setting this to true can recover db connections from poorly written applications which fail to close a connection.
DEPRECATED attributes!

4.5.5.5. sequence-manager

The *sequence-manager* element specifies the sequence manager implementation used for key generation. All sequence manager implementations shipped with OJB can be found in the *org.apache.ojb.broker.util.sequence* package. If no sequence manager is defined, OJB uses the default one. More info about sequence key generation here.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT sequence-manager (
    documentation?,
    attribute* )
>
```

The *className* attribute represents the full qualified class name of the desired sequence manager implementation - it is mandatory when using the sequence-manager element. All sequence manager implementations you find will under *org.apache.ojb.broker.util.sequence* package named as *SequenceManagerXXXImpl*

More info about the usage of the Sequence Manager implementations can be found here.

```
<!ATTLIST sequence-manager
    className CDATA #REQUIRED
>
```

4.5.5.6. object-cache

The *object-cache* element can be used to specify the ObjectCache implementation used by OJB. There are three levels of declaration:

- in OJB.properties file, to declare the standard (default) ObjectCache implementation
- on jdbc-connection-descriptor level, to declare ObjectCache implementation on a per connection/user level
- on class-descriptor level, to declare ObjectCache implementation on a per class level

Note:

The priority of the declared object-cache elements are:
per class > per jdbc descriptor > standard

E.g. if you declare ObjectCache implementation 'my.cacheDef' as standard, set ObjectCache implementation 'my.cacheA' in class-descriptor for class A and class B does not declare an object-cache element. Then OJB use 'my.cacheA' as ObjectCache for class A and 'my.cacheDef' for class B.

```
<!ELEMENT object-cache (documentation?, attribute*)>
```

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ATTLIST object-cache
    class   CDATA   #REQUIRED
>
```

Attribute 'class' specifies the full qualified class name of the used ObjectCache implementation.

4.5.5.7. custom attribute

An *attribute* element allows arbitrary name/value pairs to be represented in the repository. See the repository.dtd for details on which elements support it.

```
<!ELEMENT attribute EMPTY>
```

The *attribute-name* identifies the name of the attribute.

The *attribute-value* identifies the value of the attribute.

```
<!ATTLIST attribute
    attribute-name CDATA #REQUIRED
    attribute-value CDATA #REQUIRED
>
```

4.5.5.8. class-descriptor

For interfaces or abstract classes a *class-descriptor* holds a sequence of *extent-class* elements which specify the types extending this class.
Concrete base classes may specify a sequence of extent-class elements, naming the derived classes.

For concrete classes it must have *field-descriptor*s that describe primitive typed instance variables. References to other persistent entity classes are specified by *reference-descriptor* elements. Collections or arrays attributes that contain other persistent entity classes are specified by *collection-descriptor* elements
A class-descriptor may contain user defined custom attribute elements.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT class-descriptor (
    (
        documentation?,
        extent-class+,
        attribute* ) |
    (
        documentation?,
        object-cache?,
        extent-class*,
        field-descriptor+,
        reference-descriptor*,
        collection-descriptor*,
        index-descriptor*,
        attribute*,
        insert-procedure?,
        update-procedure?,
        delete-procedure? )
    )
>
```

The *class* attribute contains the full qualified name of the specified class. As this attribute is of the XML type ID there can only be one *class-descriptor* per class.

The *isolation-level* attribute specifies the transactional isolation to be used for this class on ODMG-level.

Note:
The *isolation-level* does not touch the jdbc-connection isolation level. It's completely independend from the database connection setting.

If the *proxy* attribute is set, proxies are used for all loading operations of instances of this class. If set to *dynamic*, dynamic proxies are used. If set to another value this value is interpreted as the full-qualified name of the proxy class to use. More info about using of proxies here.

The *proxy-prefetching-limit* attribute specifies a limit to the number of elements loaded on a proxied reference. When the first proxied element is loaded, a number up to the proxy-prefetch-limit will be loaded in addition.

The *schema* attribute may contain the database schema owning the table mapped to this class.

The *table* attribute speciefies the table name this class is mapped to.

The *row-reader* attribute may contain a full qualified class name. This class will be used as the RowReader implementation used to materialize instances of the persistent class.

The *extends* attribute ************TODO: description*************

The *accept-locks* attribute specifies whether implicit locking should propagate to this class. Currently relevant for the ODMG layer only.

The optional *initialization-method* specifies a no-argument instance method that is invoked after reading an instance from a database row. It can be used to do initialization and validations.

The optional *factory-class* specifies a factory class that that is to be used instead of a no argument constructor when new objects are created. If the factory class is specified, then the *factory-method* also must be defined. It refers to a static no-argument method of the factory class that returns a new instance.

The *refresh* attribute can be set to *true* to force OJB to refresh instances when loaded from cache. Means all field values (except references) will be replaced by values retrieved from the database. It's set to *false* by default.

```
<!ATTLIST class-descriptor
    class ID #REQUIRED
    isolation-level (read-uncommitted | read-committed |
        repeatable-read | serializable | optimistic) "read-uncommitted"
    proxy CDATA #IMPLIED
    proxy-prefetching-limit CDATA #IMPLIED
    schema CDATA #IMPLIED
    table CDATA #IMPLIED
    row-reader CDATA #IMPLIED
    extends IDREF #IMPLIED
    accept-locks (true | false) "true"
    initialization-method CDATA #IMPLIED
    factory-class CDATA #IMPLIED
    factory-method CDATA #IMPLIED
    refresh (true | false) "false"
>
```

4.5.5.9. extent-class

An extent-class element is used to specify an implementing class or a derived class that belongs to the extent of all instances of the interface or base class.

```
<!ELEMENT extent-class EMPTY>
```

The *class-ref* attribute must contain a fully qualified classname and the repository file must contain a class-descriptor for this class.

```
<!ATTLIST extent-class
    class-ref IDREF #REQUIRED
>
```

4.5.5.10. field-descriptor

A field descriptor contains mapping info for a primitive typed attribute of a persistent class.
A field descriptor may contain custom attribute elements.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT field-descriptor (documentation?, attribute*)>
```

The *id* attribute is optional. If not specified, OJB internally sorts field-descriptors according to their order of appearance in the repository file.
If a different sort order is intended the id attribute may be used to hold a unique number identifying the decriptors position in the sequence of field-descriptors.

Note:

The order of the numbers for the field-descriptors must correspond to the order of columns in the mapped table.

The *name* attribute holds the name of the persistent classes attribute. More info about persistent field handling.

The *table* attribute may specify a table different from the mapped table for the persistent class. (**currently not implemented**).

The *column* attribute specifies the column the persistent classes field is mapped to.

The *jdbc-type* attribute specifies the JDBC type of the column. If not specified OJB tries to identify the JDBC type by inspecting the Java attribute by reflection - OJB use the java/jdbc mapping desribed here.

The *primarykey* specifies if the column is a primary key column, default value is *false*.

The *nullable* attribute specifies if the column may contain null values.

The *indexed* attribute specifies if there is an index on this column

The *autoincrement* attribute specifies if the values for the persistent attribute should be automatically generated by OJB. More info about sequence key generation here.

The *sequence-name* attribute can be used to state explicitly a sequence name used by the sequence manager implementations. Check the javadocs of the used sequence manager implementation to get information if this is a mandatory attribute. OJB standard sequence manager implementations build a sequence name by its own, if the attribute was not set. More info about sequence key generation here.

The *locking* attribute is set to *true* if the persistent attribute is used for *optimistic locking*. More about optimistic locking. The default value is *false*.

The *updatelock* attribute is set to *false* if the persistent attribute is used for optimistic locking AND the dbms should update the lock column itself. The default is *true* which means that when locking is true then OJB will update the locking fields. Can only be set for TIMESTAMP and INTEGER columns.

The *default-fetch* attribute specifies whether the persistent attribute belongs to the JDO default fetch group.

The *conversion* attribute contains a fully qualified class name. This class must implement the interface `org.apache.ojb.accesslayer.conversions.FieldConversion`. A FieldConversion can be used to implement conversions between Java- attributes and database columns. More about field conversion.

The *length* attribute can be used to specify a length setting if required by the jdbc-type of the underlying database column.

The *precision* attribute can be used to specify a precision setting, if required by the jdbc-type of the underlying database column.

The *scale* attribute can be used to specify a sclae setting, if required by the jdbc-type of the underlying database column.

The *access* attribute specifies the accessibility of the field. Fields marked as *readonly* are not to modified. *readwrite* marks fields that may be read and written to. *anonymous* marks anonymous fields.
An anonymous field has a database representation (column) but no corresponding Java attribute. Hence the name of such a field does not refer to a Java attribute of the class, but is used as a unique identifier only. More info about anonymous keys here.

```
<!ATTLIST field-descriptor
    id CDATA #IMPLIED
    name CDATA #REQUIRED
    table CDATA #IMPLIED
    column CDATA #REQUIRED
    jdbc-type (BIT | TINYINT | SMALLINT | INTEGER | BIGINT | DOUBLE |
               FLOAT | REAL | NUMERIC | DECIMAL | CHAR | VARCHAR |
               LONGVARCHAR | DATE | TIME | TIMESTAMP | BINARY |
               VARBINARY | LONGVARBINARY | CLOB | BLOB) #REQUIRED
    primarykey (true | false) "false"
    nullable (true | false) "true"
    indexed (true | false) "false"
    autoincrement (true | false) "false"
    sequence-name CDATA #IMPLIED
    locking (true | false) "false"
    update-lock (true | false) "true"
    default-fetch (true | false) "false"
    conversion CDATA #IMPLIED
    length CDATA #IMPLIED
    precision CDATA #IMPLIED
    scale CDATA #IMPLIED
    access (readonly | readwrite | anonymous) "readwrite"
>
```

4.5.5.11. reference-descriptor

A reference-descriptor contains mapping info for an attribute of a persistent class that is not primitive but references another persistent entity Object. More about 1:1 references here.

A *foreignkey* element contains information on foreign key columns that implement the association on the database level.

```
<!ELEMENT reference-descriptor ( foreignkey+)>
```

The *name* attribute holds the name of the persistent classes attribute. Depending on the used PersistendField implementation, there must be e.g. an attribute in the persistent class with this name or a JavaBeans compliant property of this name.

The *class-ref* attribute contains a fully qualified class name. This class is the Object type of the persistent reference attribute. As this is an IDREF there must be a class-descriptor for this class in the repository too.

The *proxy* attribute can be set to *true* to specify that proxy based lazy loading should be used for this attribute.

The *proxy-prefetch-limit* attribute specifies a limit to the number of elements loaded on a proxied reference. When the first proxied element is loaded, a number up to the proxy-prefetch-limit will be loaded in addition.

The *refresh* attribute can be set to *true* to force OJB to refresh object references on instance loading.

Note:

This does not mean that all referenced objects will be read from database. It only means that the reference will be refreshed, the objects itself may provided by the cache. To refresh the objects set the *refresh* attribute of *class-descriptor*.

The *auto-retrieve* attribute specifies whether OJB automatically retrieves this reference attribute on loading the persistent object. If set to *false* the reference attribute is set to null. In this case the user is responsible to fill the reference attribute.
More info about auto-retrieve here.

The *auto-update* attribute specifies whether OJB automatically stores this reference attribute on storing the persistent object. More info about the auto-XXX settings here.

Note:
This attribute must be set to false if using the OTM, ODMG or JDO layer.

The *auto-delete* attribute specifies whether OJB automatically deletes this reference attribute on deleting the persistent object. More info about the auto-XXX settings here.

Note:
This attribute must be set to false if using the OTM, ODMG or JDO layer.

The *otm-dependent* attribute specifies whether the OTM layer automatically creates the referred object or deletes it if the reference field is set to null. Also otm-dependent references behave as if auto-update and auto-delete were set to true, but the auto-update and auto-delete attributes themself must be always set to false for use with OTM layer.

```
<!ATTLIST reference-descriptor
    name CDATA #REQUIRED
    class-ref IDREF #REQUIRED

    proxy (true | false) "false"
    proxy-prefetching-limit CDATA #IMPLIED
    refresh (true | false) "false"

    auto-retrieve (true | false) "true"
    auto-update (true | false) "false"
    auto-delete (true | false) "false"
    otm-dependent (true | false) "false"
>
```

4.5.5.12. foreignkey

A *foreignkey* element contains information on a foreign-key persistent attribute that implement the association on the database level.

```
<!ELEMENT foreignkey EMPTY>
```

The *field-ref* and *field-id-ref* attributes contain the name and the id attributes of the field-descriptor used as a foreign key.

Note:
Exactly one of these attributes must be specified.

```
<!ATTLIST foreignkey
    field-id-ref CDATA #IMPLIED
    field-ref CDATA #IMPLIED
>
```

4.5.5.13. collection-descriptor

A collection-descriptor contains mapping info for a liCollection- or Array-attribute of a persistent class that contains persistent entity Objects. See more about 1:n and m:n references.

The *inverse-foreignkey* elements contains information on foreign-key attributes that implement the association on the database level.

The *fk-pointing-to-this-class* and *fk-pointing-to-element-class* elements are only needed if the Collection or array implements a m:n association. In this case they contain information on the foreign-key columns of the intermediary table.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT collection-descriptor (
    documentation?,
    orderby*,
    inverse-foreignkey*,
    fk-pointing-to-this-class*,
    fk-pointing-to-element-class*,
    attribute*)>
```

The *name* attribute holds the name of the persistent classes attribute. More info about persistent field handling.

The *collection-class* may hold a fully qualified class name. This class must be the Java type of the Collection attribute. This attribute must only specified if the attribute type is not a `java.util.Collection` (or subclass) or Array type. It is also possible to use non Collection or Array type user defined "collection" classes. More info see section manageable collection.

The *element-class-ref* attribute contains a fully qualified class name. This class is the Object type of the elements of persistent collection or Array attribute. As this is an IDREF there must be a class-descriptor for this class in the repository too.

The *orderby* attribute may specify a field of the element class. The Collection or Array will be sorted according to the specified attribute. The sort attribute may be used to specify ascending or descending order for this operation.

The *indirection-table* must specify the name of an intermediary table, if the persistent collection attribute implements a m:n association.

The *proxy* attribute can be set to true to specify that proxy based lazy loading should be used for this attribute. More about using proxy here.

The *proxy-prefetch-limit* attribute specifies a limit to the number of elements loaded on a proxied reference. When the first proxied element is loaded, a number up to the proxy-prefetch-limit will be loaded in addition.

The *refresh* attribute can be set to *true* to force OJB to refresh object references on instance loading.

Note:

This does not mean that all referenced objects will be read from database. It only means that the reference will be refreshed, the objects itself may provided by the cache. To refresh the objects use the *refresh* attribute in *class-descriptor*.

The *auto-retrieve* attribute specifies whether OJB automatically retrieves this reference attribute on loading the persistent object. If set to *false* the reference attribute is set to null. In this case the user is responsible to fill the reference attribute.
More info about auto-retrieve here.

The *auto-update* attribute specifies whether OJB automatically stores this reference attribute on storing the persistent object.
More info about the auto-XXX settings here.

Note:

This attribute must be set to false if using the OTM, ODMG or JDO layer.

The *auto-delete* attribute specifies whether OJB automatically deletes this reference attribute on deleting the persistent object.
More info about the auto-XXX settings here.

Note:

This attribute must be set to false if using the OTM, ODMG or JDO layer.

The *otm-dependent* attribute specifies whether the OTM layer automatically creates collection elements that were included into the collection, and deletes collection elements that were removed from the collection. Also otm-dependent references behave as if auto-update and auto-delete were set to true, but the auto-update and auto-delete attributes themself must be always set to false for use with OTM layer.

```
<!ATTLIST collection-descriptor
    name CDATA #IMPLIED
    collection-class CDATA #IMPLIED
    element-class-ref IDREF #REQUIRED
    orderby CDATA #IMPLIED
    sort (ASC | DESC) "ASC"

    indirection-table CDATA #IMPLIED

    proxy (true | false) "false"
    proxy-prefetching-limit CDATA #IMPLIED
    refresh (true | false) "false"

    auto-retrieve (true | false) "true"
    auto-update (true | false) "false"
    auto-delete (true | false) "false"
    otm-dependent (true | false) "false"
>
```

4.5.5.14. inverse-foreignkey

A *inverse-foreignkey* element contains information on a foreign-key persistent attribute that implement the association on the database level.

```
<!ELEMENT inverse-foreignkey EMPTY>
```

The *field-ref* and *field-id-ref* attributes contain the name and the id attributes of the field-descriptor used as a foreign key. Exactly one of these attributes must be specified.

```
<!ATTLIST inverse-foreignkey
    field-id-ref CDATA #IMPLIED
    field-ref CDATA #IMPLIED
>
```

4.5.5.15. fk-pointing-to-this-class

A *fk-pointing-to-this-class* element contains information on a foreign-key column of an intermediary table in a m:n scenario.

```
<!ELEMENT fk-pointing-to-this-class EMPTY>
```

The *column* attribute specifies the foreign-key column in the intermediary table that points to the class holding the collection.

```
<!ATTLIST fk-pointing-to-this-class
    column CDATA #REQUIRED
>
```

4.5.5.16. fk-pointing-to-element-class

A *fk-pointing-to-element-class* element contains information on a foreign-key column of an intermediary table in a m:n scenario.

```
<!ELEMENT fk-pointing-to-element-class EMPTY>
```

The *column* attribute specifies the foreign-key column in the intermediary table that points to the class of the collection elements.

```
<!ATTLIST fk-pointing-to-element-class
```

```
    column CDATA #REQUIRED
>
```

4.5.5.17. query-customizer

A query enhancer element to enhance the 1:n query, e.g. to modify the result objects of a query. More info about customizing collection queries.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT query-customizer (
    documentation?,
    attribute*)>

<!ATTLIST query-customizer
    class CDATA #REQUIRED
>
```

4.5.5.18. index-descriptor

An *index-descriptor* describes an index by listing its columns. It may be unique or not.

```
<!ELEMENT index-descriptor (documentation?, index-column+)>

<!ATTLIST index-descriptor
    name CDATA #REQUIRED
    unique (true | false) "false">
```

4.5.5.19. index-column

An *index-column* is just the name of a column in an index.

```
<!ELEMENT index-column (documentation?)>

<!ATTLIST index-column
    name CDATA #REQUIRED>
```

4.5.5.20. Stored Procedure Support

OJB supports stored procedures for insert, update and delete operations. How to use stored procedures within OJB can be found here.

insert-procedure

Identifies the procedure/function that should be used to handle insertions for a specific class-descriptor.

The nested *argument* elements define the argument list for the procedure/function as well as the source for each argument.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT insert-procedure
    (documentation?, (runtime-argument | constant-argument)?, attribute*)>
```

The *name* attribute identifies the name of the procedure/function to use

The *return-field-ref* identifies the field-descriptor that will receive the value that is returned by the procedure/function. If the procedure/ function does not include a return value, then do not specify a value for this attribute.

The *include-all-fields* attribute indicates if all field-descriptors in the corresponding class-descriptor are to be passed to the procedure/ function. If include-all-fields is 'true', any nested 'argument' elements will be ignored. In this case, values for all field-descriptors will be passed to the procedure/function. The order of values that are passed to the procedure/function will match the order of field-descriptors on the corresponding class-descriptor. If include-all-fields is false, then values will be passed to the procedure/function based on the information in the nested 'argument' elements.

```
<!ATTLIST insert-procedure
    name CDATA #REQUIRED
    return-field-ref CDATA #IMPLIED
    include-all-fields (true | false) "false"
>
```

update-procedure

Identifies the procedure/function that should be used to handle updates for a specific class-descriptor.

The nested *argument* elements define the argument list for the procedure/function as well as the source for each argument.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT update-procedure
    (documentation?, (runtime-argument | constant-argument)?, attribute*)>
```

The *name* attribute identifies the name of the procedure/function to use

The *return-field-ref* identifies the field-descriptor that will receive the value that is returned by the procedure/function. If the procedure/ function does not include a return value, then do not specify a value for this attribute.

The *include-all-fields* attribute indicates if all field-descriptors in the corresponding class-descriptor are to be passed to the procedure/ function. If include-all-fields is 'true', any nested 'argument' elements will be ignored. In this case, values for all field-descriptors will be passed to the procedure/function. The order of values that are passed to the procedure/function will match the order of field-descriptors on the corresponding class-descriptor. If include-all-fields is false, then values will be passed to the procedure/function based on the information in the nested 'argument' elements.

```
<!ATTLIST update-procedure
    name CDATA #REQUIRED
    return-field-ref CDATA #IMPLIED
    include-all-fields (true | false) "false"
>
```

delete-procedure

Identifies the procedure/function that should be used to handle deletions for a specific class-descriptor.

The nested *runtime-argument* and *constant-argument* elements define the argument list for the procedure/function as well as the source for each argument.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT delete-procedure
    (documentation?, (runtime-argument | constant-argument)?, attribute*)>
```

The *name* attribute identifies the name of the procedure/function to use

The *return-field-ref* identifies the field-descriptor that will receive the value that is returned by the procedure/function. If the procedure/ function does not include a return value, then do not specify a value for this attribute.

The *include-pk-only* attribute indicates if all field-descriptors in the corresponding class-descriptor that are identified as being part of the primary key are to be passed to the procedure/function. If include-pk-only is 'true', any nested 'argument' elements will be ignored. In this case, values for all field-descriptors that are identified as being part of the primary key will be passed to the procedure/function. The order of values that are passed to the procedure/function will match the order of field-descriptors on the corresponding class-descriptor. If include-pk-only is false, then values will be passed to the procedure/ function based on the information in the nested 'argument' elements.

```
<!ATTLIST delete-procedure
    name CDATA #REQUIRED
    return-field-ref CDATA #IMPLIED
    include-pk-only (true | false) "false"
>
```

runtime-argument

Defines an argument that is passed to a procedure/function. Each argument will be set to a value from a field-descriptor or null.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT runtime-argument
    (documentation?, attribute*)>
```

The *field-ref* attribute identifies the field-descriptor in the corresponding class-descriptor that provides the value for this argument. If this attribute is unspecified, then this argument will be set to null.

```
<!ATTLIST runtime-argument
    field-ref CDATA #IMPLIED
    return (true | false) "false"
>
```

constant-argument

Defines a constant value that is passed to a procedure/function.

Use the *custom-attribute* element to pass implementation specific properties.

```
<!ELEMENT constant-argument
    (documentation?, attribute*)>
```

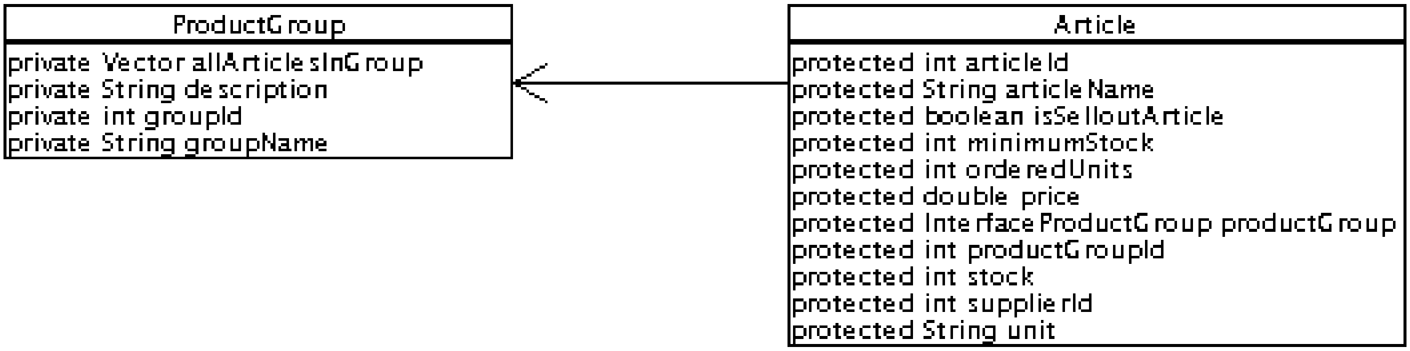
The *value* attribute identifies the value that is passed to the procedure/ function.

```
<!ATTLIST constant-argument
    value CDATA #REQUIRED
>
```

4.5.6. Basic Technique

4.5.6.1. Mapping 1:1 associations

As a sample for a simple association we take the reference from an article to its productgroup.
This association is navigable only from the article to its productgroup. Both classes are modelled in the following class diagram. This diagram does not show methods, as only attributes are relevant for the O/R mapping process.


ProductGroup
private Vector allArticlesInGroup
private String description
private int groupId
private String groupName
Article
protected int articleId
protected String articleName
protected boolean isSelloutArticle
protected int minimumStock
protected int orderedUnits
protected double price
protected InterfaceProductGroup productGroup
protected int productGroupId
protected int stock
protected int supplierId
protected String unit

1:1 association

The association is implemented by the attribute `productGroup`. To automatically maintain this reference OJB relies on foreignkey attributes. The foreign key containing the `groupId` of the referenced `productgroup` is stored in the attribute `productGroupId`. To avoid FK attribute in persistent object class see section about anonymous keys.

This is the DDL of the underlying tables:

```
CREATE TABLE Artikel
(
    Artikel_Nr          INT NOT NULL PRIMARY KEY,
    Artikelname         VARCHAR(60),
    Lieferanten_Nr      INT,
    Kategorie_Nr        INT,
    Liefereinheit       VARCHAR(30),
    Einzelpreis         FLOAT,
    Lagerbestand        INT,
    BestellteEinheiten  INT,
    MindestBestand      INT,
    Auslaufartikel      INT
)

CREATE TABLE Kategorien
(
    Kategorie_Nr        INT NOT NULL PRIMARY KEY,
    KategorieName       VARCHAR(20),
    Beschreibung        VARCHAR(60)
)
```

To declare the foreign key mechanics of this reference attribute we have to add a reference-descriptor to the class-descriptor of the Article class. This descriptor contains the following information:

- The attribute implementing the association (`name="productGroup"`) is productGroup.
- The referenced object is of type (`class-ref="org.apache.ojb.broker.ProductGroup"`) `org.apache.ojb.broker.ProductGroup`.
- A reference-descriptor contains one or more foreignkey elements. These elements define foreign key attributes. The element

  ```
                  <foreignkey field-ref="productGroupId"/>
  ```

  contains the name of the field-descriptor describing the foreignkey fields. The FieldDescriptor with the name "productGroupId" describes the foreignkey attribute productGroupId:

  ```
                  <field-descriptor
                      name="productGroupId"
                      column="Kategorie_Nr"
                      jdbc-type="INTEGER"
                  />
  ```

See the following extract from the repository.xml file containing the Article ClassDescriptor:

```
<!-- 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" />
    <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"
    />
      ...
    <reference-descriptor
       name="productGroup"
       class-ref="org.apache.ojb.broker.ProductGroup"
    >
       <foreignkey field-ref="productGroupId"/>
    </reference-descriptor>
</class-descriptor>
```

This example provides unidirectional navigation only. Bidirectional navigation may be added by including a reference from a ProductGroup to a single Article (for example, a sample article for the productgroup). To accomplish this we need to perform the following steps:

1. Add a private Article attribute named `sampleArticle` to the class `ProductGroup`.
2. Add a private int attribute named `sampleArticleId` to the *ProductGroup* class representing the foreign key. To avoid FK attribute in persistent object class see section about anonymous keys.
3. Add a column `SAMPLE_ARTICLE_ID INT` to the table `Kategorien`.
4. Add a FieldDescriptor for the foreignkey attribute to the ClassDescriptor of the Class ProductGroup:

```
<field-descriptor
  name="sampleArticleId"
  column="SAMPLE_ARTICLE_ID"
  jdbc-type="INTEGER"
/>
```

1. Add a ReferenceDescriptor to the ClassDescriptor of the Class ProductGroup:

```
<reference-descriptor
    name="sampleArticle"
    class-ref="org.apache.ojb.broker.Article"
>
    <foreignkey field-ref="sampleArticleId""/>
</reference-descriptor>
```

1:1 auto-xxx setting

General info about the `auto-xxx` and `proxy` attributes can be found here

auto-retrieve

See here

auto-update

- **none** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced object will NOT be updated by default.The reference will not be *inserted* or *updated*, the link to the reference (foreign key value to the reference) on the main object will not be assigned automatically. The user has to link the main object and to store the reference *before* the main object to avoid violation of referential integrity.
- **link** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the FK assignment on the main object was done automatic. OJB reads the PK from the referenced object and sets these values as FK in main object. But the referenced object remains untouched. If no referenced object is found, the FK will be nullified. (On insert it is allowed to set the FK without populating the referenced object)

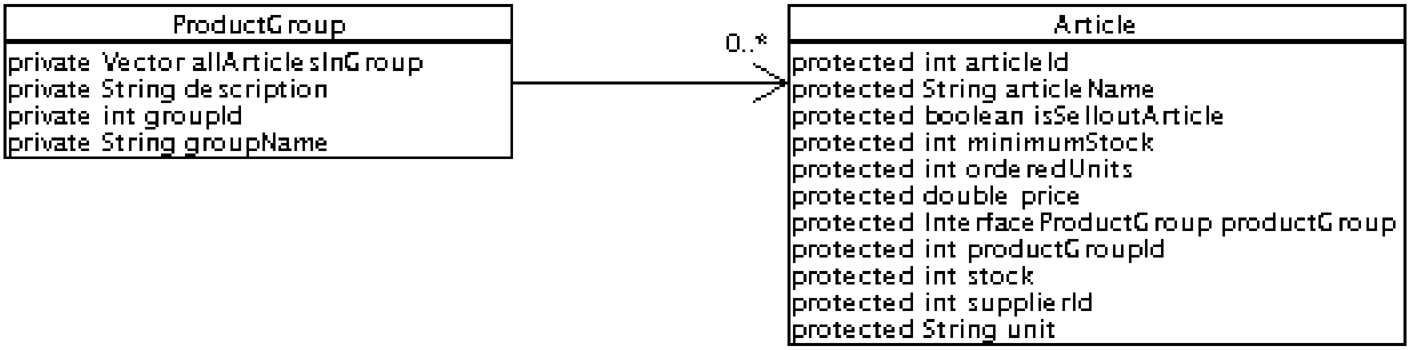
- **object** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced object will be stored first, then OJB does the same as in *link*.
- **false** Is equivalent to *link*.
- **true** Is equivalent to *object*.

auto-delete

- **none** On deleting an object with `PersistenceBroker.delete(...)` the referenced object will NOT be touched.
- **link** Is equivalent to *none*.
- **object** On deleting an object with `PersistenceBroker.delete(...)` the referenced object will be deleted too.
- **false** Is equivalent to *none*.
- **true** Is equivalent to *object*.

4.5.6.2. Mapping 1:n associations

We will take a different perspective from the previous exmaple for a 1:n association. We will associate multiple Articles with a single ProductGroup. This association is navigable only from the ProductGroup to its Article instances. Both classes are modelled in the following class diagram. This diagram does not show methods, as only attributes are relevant for the O/R mapping process.


ProductGroup
private Vector allArticlesInGroup
private String description
private int groupId
private String groupName
0..*
Article
protected int articleId
protected String articleName
protected boolean isSelloutArticle
protected int minimumStock
protected int orderedUnits
protected double price
protected InterfaceProductGroup productGroup
protected int productGroupId
protected int stock
protected int supplierId
protected String unit

1:n association

The association is implemented by the `Vector` attribute `allArticlesInGroup` on the ProductGroup class. As in the previous example, the Article class contains a foreignkey attribute named productGroupId that identifies an Article's ProductGroup. The Database table are the same as above.

To declare the foreign key mechanics of this collection attribute we must add a CollectionDescriptor to the ClassDescriptor of the ProductGoup class. This descriptor contains the following information:

1. The attribute implementing the association (`name="allArticlesInGroup"`)
2. The class of the elements in the collection (`element-class-ref="org.apache.ojb.broker.Article"`)
3. The name of field-descriptor of the element class used as foreign key attributes are defined in inverse-foreignkey elements:

   ```
   <inverse-foreignkey field-ref="productGroupId"/>
   ```
   This is again pointing to the field-descriptor for the attribute `productGoupId` in class Article.
4. optional attributes to define the sort order of the retrieved collection: `orderby="articleId" sort="DESC"`.

See the following extract from the repository.xml file containing the ProductGoup ClassDescriptor:

```
<!-- Definitions for org.apache.ojb.broker.ProductGroup -->
 <class-descriptor
    class="org.apache.ojb.broker.ProductGroup"
    table="Kategorien"
 >
    <field-descriptor
       name="groupId"
       column="Kategorie_Nr"
       jdbc-type="INTEGER"
```

```
      primarykey="true"
      autoincrement="true"
   />
   <field-descriptor
      name="groupName"
      column="KategorieName"
      jdbc-type="VARCHAR"
   />
   <field-descriptor
      name="description"
      column="Beschreibung"
      jdbc-type="VARCHAR"
   />
   <collection-descriptor
      name="allArticlesInGroup"
      element-class-ref="org.apache.ojb.broker.Article"
      orderby="articleId"
      sort="DESC"
   >
      <inverse-foreignkey field-ref="productGroupId"/>
   </collection-descriptor>
</class-descriptor>
```

With the mapping shown above OJB has two possibilities to load the Articles belonging to a ProductGroup:

1. loading all Articles of the ProductGroup immediately after loading the ProductGroup. This is done with **two** SQL-calls: one for the ProductGroup and one for all Articles.
2. if Article is a proxy (using proxy classes), OJB will only load the keys of the Articles after the ProductGroup. When accessing an Article-proxy OJB will have to materialize it with another SQL-Call. Loading the ProductGroup and all it's Articles will thus produce **n+2** SQL-calls: one for the ProductGroup, one for keys of the Articles and one for each Article.

Both approaches have their benefits and drawbacks:

- **A.** is suitable for a small number of related objects that are easily instantiated. It's efficient regarding DB-calls. The major drawback is the amount of data loaded. For example to show a list of ProductGroups the Articles may not be needed.
- **B.** is best used for a large number of related heavy objects. This solution loads the objects when they are needed ("lazy loading"). The price to pay is a DB-call for each object.

Further down a third solution using a single proxy for a whole collection will be presented to circumvent the described drawbacks.

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. But in some cases the *default* behaviour of OJB is undesired. Please read here for more information.

1:n auto-xxx setting

General info about the *auto-xxx* and *proxy* attributes can be found here.

auto-retrieve

See here

auto-update

- **none** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects are NOT updated by default. The referenced objects will not be *inserted* or *updated*, the referenced objects will not be linked (foreign key assignment on referenced objects) to the main object automatically. The user has to link and to store the referenced objects *after* storing the main object to avoid violation of referential integrity.
- **link** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects are NOT updated by default. The referenced objects will not be *inserted* or *updated*, but the referenced objects will be linked automatically (FK assignment) the main object.
- **object** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects will be linked and stored automatically.

- **false** Is equivalent to *link*.
- **true** Is equivalent to *object*.

auto-delete

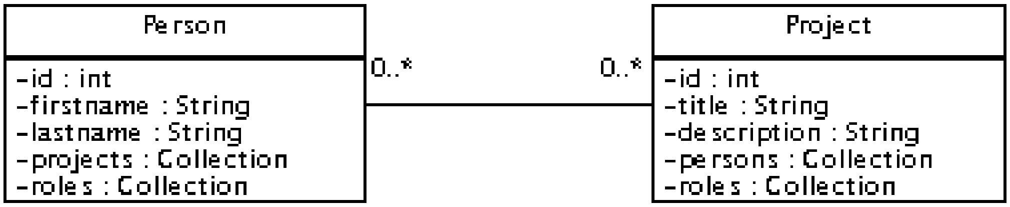
- **none** On deleting an object with `PersistenceBroker.delete(...)` the referenced objects are NOT touched. This may lead to violation of referential integrity if the referenced objects are childs of the main object. In this case the referenced objects have to be deleted manually first.
- **link** Is equivalent to *none*.
- **object** On deleting an object with `PersistenceBroker.delete(...)` the referenced objects will be deleted too.
- **false** Is equivalent to *none*.
- **true** Is equivalent to *object*.

4.5.6.3. Mapping m:n associations

OJB provides support for manually decomposed m:n associations as well as an automated support for non decomposed m:n associations.

Manual decomposition into two 1:n associations

Have a look at the following class diagram:



m:n association

We see a two classes with a m:n association. A Person can work for an arbitrary number of Projects. A Project may have any number of Persons associated to it.

Relational databases don't support m:n associations. They require to perform a manual decomposition by means of an intermediary table. The DDL looks like follows:

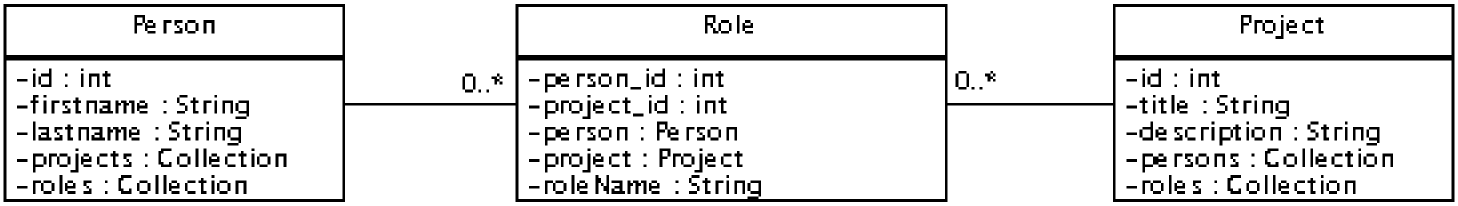
```
CREATE TABLE PERSON (
    ID          INT NOT NULL PRIMARY KEY,
    FIRSTNAME   VARCHAR(50),
    LASTNAME    VARCHAR(50)
  );

CREATE TABLE PROJECT (
    ID          INT NOT NULL PRIMARY KEY,
    TITLE       VARCHAR(50),
    DESCRIPTION VARCHAR(250)
  );

CREATE TABLE PERSON_PROJECT (
    PERSON_ID   INT NOT NULL,
    PROJECT_ID  INT NOT NULL,
    PRIMARY KEY (PERSON_ID, PROJECT_ID)
  );
```

This intermediary table allows to decompose the m:n association into two 1:n associations. The intermediary table may also hold additional information. For example, the role a certain person plays for a project:

```
    CREATE TABLE PERSON_PROJECT (
        PERSON_ID   INT NOT NULL,
        PROJECT_ID  INT NOT NULL,
        ROLENAME    VARCHAR(20),
        PRIMARY KEY (PERSON_ID, PROJECT_ID)
```

```
      );
```

The decomposition is mandatory on the ER model level. On the object model level it is not mandatory, but may be a valid solution. It is mandatory on the object level if the association is qualified (as in our example with a rolename). This will result in the introduction of a association class. A class-diagram reflecting this decomposition looks like:



m:n association

A `Person` object has a Collection attribute `roles` containing `Role` entries. A `Project` has a Collection attribute `roles` containing `Role` entries. A `Role` has reference attributes to its `Person` and to its `Project`.
Handling of 1:n mapping has been explained above. Thus we will finish this section with a short look at the repository entries for the classes `org.apache.ojb.broker.Person`, `org.apache.ojb.broker.Project` and `org.apache.ojb.broker.Role`:

```
<!-- Definitions for org.apache.ojb.broker.Person -->
   <class-descriptor
      class="org.apache.ojb.broker.Person"
      table="PERSON"
   >
      <field-descriptor
         name="id"
         column="ID"
         jdbc-type="INTEGER"
         primarykey="true"
         autoincrement="true"
      />
      <field-descriptor
         name="firstname"
         column="FIRSTNAME"
         jdbc-type="VARCHAR"
      />
      <field-descriptor
         name="lastname"
         column="LASTNAME"
         jdbc-type="VARCHAR"
      />
      <collection-descriptor
         name="roles"
         element-class-ref="org.apache.ojb.broker.Role"
      >
         <inverse-foreignkey field-ref="person_id"/>
      </collection-descriptor>
      ...
   </class-descriptor>

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

```
         name="title"
         column="TITLE"
         jdbc-type="VARCHAR"
      />
      <field-descriptor
         name="description"
         column="DESCRIPTION"
         jdbc-type="VARCHAR"
      />
      <collection-descriptor
         name="roles"
         element-class-ref="org.apache.ojb.broker.Role"
      >
         <inverse-foreignkey field-ref="project_id"/>
      </collection-descriptor>
      ...
   </class-descriptor>

<!-- Definitions for org.apache.ojb.broker.Role -->
   <class-descriptor
      class="org.apache.ojb.broker.Role"
      table="PERSON_PROJECT"
   >
      <field-descriptor
         name="person_id"
         column="PERSON_ID"
         jdbc-type="INTEGER"
         primarykey="true"
      />
      <field-descriptor
         name="project_id"
         column="PROJECT_ID"
         jdbc-type="INTEGER"
         primarykey="true"
      />
      <field-descriptor
         name="roleName"
         column="ROLENAME"
         jdbc-type="VARCHAR"
      />
      <reference-descriptor
         name="person"
         class-ref="org.apache.ojb.broker.Person"
      >
         <foreignkey field-ref="person_id"/>
      </reference-descriptor>
      <reference-descriptor
         name="project"
         class-ref="org.apache.ojb.broker.Project"
      >
         <foreignkey field-ref="project_id"/>
      </reference-descriptor>
   </class-descriptor>
```

Support for Non-Decomposed m:n Mappings

If there is no need for an association class at the object level (we are not interested in role information), OJB can be configured to do the m:n mapping transparently. For example, a Person does not have a collection of `Role` objects but only a Collection of `Project` objects (held in the attribute `projects`). Projects also are expected to contain a collection of `Person` objects (hold in attribute `persons`).

To tell OJB how to handle this m:n association the CollectionDescriptors for the Collection attributes `projects` and `roles` need additional information on the intermediary table and the foreign key columns pointing to the `PERSON` table and the foreign key columns pointing to the `PROJECT` table:

Note:

> OJB supports a multiplicity of collection implementations, inter alia `org.apache.ojb.broker.util.collections.RemovalAwareCollection` and `org.apache.ojb.broker.util.collections.RemovalAwareList`. By default the removal aware collections were used. This cause problems in m:n relations when `auto-update="true" or "object"` and `auto-delete="false" or "none"` is set, because objects deleted in the collection will be deleted on update of main object. Thus it is recommended to use a NOT removal aware collection class in m:n relations using the collection-class attribute.

Example for setting a collection class in the collection-descriptor:

```
collection-class="org.apache.ojb.broker.util.collections.ManageableArrayList"
```

An full example for a non-decomposed m:n relation looks like:

```
<class-descriptor
  class="org.apache.ojb.broker.Person"
  table="PERSON"
>
  <field-descriptor
     name="id"
     column="ID"
     jdbc-type="INTEGER"
     primarykey="true"
     autoincrement="true"
  />
  <field-descriptor
     name="firstname"
     column="FIRSTNAME"
     jdbc-type="VARCHAR"
  />
  <field-descriptor
     name="lastname"
     column="LASTNAME"
     jdbc-type="VARCHAR"
  />
  ...
  <collection-descriptor
     name="projects"
     collection-class="org.apache.ojb.broker.util.collections.ManageableArrayList"
     element-class-ref="org.apache.ojb.broker.Project"
     auto-retrieve="true"
     auto-update="true"
     indirection-table="PERSON_PROJECT"
  >
     <fk-pointing-to-this-class column="PERSON_ID"/>
     <fk-pointing-to-element-class column="PROJECT_ID"/>
  </collection-descriptor>
</class-descriptor>

<!-- Definitions for org.apache.ojb.broker.Project -->
<class-descriptor
  class="org.apache.ojb.broker.Project"
  table="PROJECT"
>
  <field-descriptor
     name="id"
     column="ID"
     jdbc-type="INTEGER"
     primarykey="true"
     autoincrement="true"
  />
  <field-descriptor
     name="title"
     column="TITLE"
     jdbc-type="VARCHAR"
  />
  <field-descriptor
     name="description"
     column="DESCRIPTION"
     jdbc-type="VARCHAR"
  />
  ...
  <collection-descriptor
     name="persons"
     collection-class="org.apache.ojb.broker.util.collections.ManageableArrayList"
     element-class-ref="org.apache.ojb.broker.Person"
     auto-retrieve="true"
     auto-update="false"
     indirection-table="PERSON_PROJECT"
  >
     <fk-pointing-to-this-class column="PROJECT_ID"/>
     <fk-pointing-to-element-class column="PERSON_ID"/>
  </collection-descriptor>
</class-descriptor>
```

That is all that needs to be configured! See the code in class `org.apache.ojb.broker.MtoNMapping` for JUnit testmethods using the classes `Person`, `Project` and `Role`.

m:n auto-xxx setting

General info about the `auto-xxx` and `proxy` attributes can be found here

auto-retrieve
See here

auto-update

- **none** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects are NOT updated by default. The referenced objects will not be *inserted* or *updated*, the referenced objects will not be linked (creation of FK entries in the indirection table) automatically. The user has to store the main object, the referenced objects and to link the m:n relation after storing of all objects. establishing the m:n relationship *before* storing main and referenced objects may violate referential integrity.
- **link** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects are NOT updated by default. The referenced objects will not be *inserted* or *updated*, but the m:n relation will be linked automatically (creation of FK entries in the indirection table).

Note:
Make sure that the referenced objects exist in database before storing the main object with auto-update set *link* to avoid violation of referential integrity.

- **object** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects will be linked and stored automatically.
- **false** Is equivalent to *link*.
- **true** Is equivalent to *object*.

auto-delete

- **none** On deleting an object with `PersistenceBroker.delete(...)` the referenced objects are NOT touched. The corresponding entries of the main object in the indirection table will not be removed. This may lead to violation of referential integrity depending on the definition of the indirection table.
- **link** On deleting an object with `PersistenceBroker.delete(...)` the m:n relation will be *unlinked* (all entries of the main object in the indirection table will be removed).
- **object** On deleting an object with `PersistenceBroker.delete(...)` all referenced objects will be deleted too.
- **false** Is equivalent to *link*.
- **true** Is equivalent to *object*.

4.5.6.4. Setting Load, Update, and Delete Cascading

As shown in the sections on 1:1, 1:n and m:n mappings, OJB manages associations (or object references in Java terminology) by declaring special Reference and Collection Descriptors. These Descriptor may contain some additional information that modifies OJB's behaviour on object materialization, updating and deletion.

The behaviour depends on specific attributes

- *auto-retrieve* - possible settings are *false*, *true*. If not specified in the descriptor the default value is *true*
- *auto-update* - possible settings are *none*, *link*, *object* and deprecated [*false*, *true*]. If not specified in the descriptor the default value is *false*
- *auto-delete* - possible settings are *none*, *link*, *object* and deprecated [*false*, *true*]. If not specified in the descriptor the default value is *false*

Note:

When using a top-level api (ODMG, OTM, JDO) it is mandatory to use the *default* auto-XXX settings (or don't specify the attributes) for proper work.
This may change in future.

The attribute *auto-update* and *auto-delete* are described in detail in the corresponding sections for 1:1, 1:n and m:n references. The *auto-retrieve* setting is described below:

auto-retrieve setting

The `auto-retrieve` attribute used in `reference-descriptor` or `collection-descriptor` elements handles the loading behaviour of references (1:1, 1:n and m:n):

- **false** If set *false* the referenced objects will not be materialized on object materialization. The user has to materialize the n-side objects (or single object for 1:1) by hand using one of the following service methods of the `PersistenceBroker` class:

  ```
  PersistenceBroker.retrieveReference(Object obj, String attributeName);
  // or
  PersistenceBroker.retrieveAllReferences(Object obj);
  ```

  The first method load only the specified reference, the second one loads all references declared for the given object.

Note:

Be careful when using "opposite" settings, e.g. if you declare a 1:1 reference with auto-retrieve="false" BUT auto-update="object" (or "true" or "link").
Before you can perform an update on the main object, you have to "retrieve" the 1:1 reference. Otherwise you will end up with an nullified reference enty in main object, because OJB doesn't find the referenced object on update and assume the reference was removed.

- **true** If set *true* the referenced objects (single reference or all n-side objects) will be automatic loaded by OJB when the main object was materialized.

If OJB is configured to use proxies, the referenced objects are not materialized immmediately, but lazy loading proxy objects are used instead.

In the following code sample, a reference-descriptor and a collection-descriptor are configured to use cascading retrieval (`auto-retrieve="true"`), cascading insert/update (`auto-update="object"` or `auto-update="true"`) and cascading delete (`auto-delete="object"` or `auto-delete="true"`) operations:

```
<reference-descriptor
 name="productGroup"
 class-ref="org.apache.ojb.broker.ProductGroup"
 auto-retrieve="true"
 auto-update="object"
 auto-delete="object"
>
 <foreignkey field-ref="productGroupId"/>
</reference-descriptor>

<collection-descriptor
 name="allArticlesInGroup"
 element-class-ref="org.apache.ojb.broker.Article"
 auto-retrieve="true"
 auto-update="object"
 auto-delete="object"
 orderby="articleId"
```

```
 sort="DESC"
>
 <inverse-foreignkey field-ref="productGroupId"/>
</collection-descriptor>
```

Link references

If in `reference-descriptor` or `collection-descriptor` the *auto-update* or *auto-delete* attributes are set to *none*, OJB does not touch the referenced objects on insert, update or delete operations of the main object. The user has to take care of the correct handling of referenced objects. When using referential integrity (who does not ?) it's essential that insert and delete operations are done in the correct sequence.

One important thing is assignment of the FK values. The assign of the FK values is transcribed with ***link references*** in OJB. In 1:1 references the main object has a FK to the referenced object, in 1:n references the referenced objects have FK pointing to the main object and in non-decomposed m:n relations a indirection table containing FK values from both sides of the relationship is used.

OJB provides some helper methods for linking references manually (assignment of the FK) in `org.apache.ojb.broker.util.BrokerHelper` class.

```
public void link(Object obj, boolean insert)
public void unlink(Object obj)
public boolean link(Object obj, String attributeName, boolean insert)
public boolean unlink(Object obj, String attributeName)
```

These methods are accessible via `org.apache.ojb.broker.PersistenceBroker`:

```
BrokerHelper bh = broker.serviceBrokerHelper();
```

Note:

The *link/unlink* methods are only useful if you set auto-update/-delete to *none*. In all other cases OJB handles the link/unlink of references internally. It is also possible to set all FK values by hand without using the link/unlink service methods.

Examples

Now we prepared for some example. Say class `Movie` has an m:n reference with class `Actor` and we want to store an Movie object with a list of Actor objects. The auto-update setting of collection-descriptor for Movie is *none*:

```
broker.beginTransaction();
// store main object first
broker.store(movie);
//now we store the right-side objects
Iterator it = movie.getActors().iterator();
while(it.hasNext())
{
    Object actor = it.next();
    broker.store(actor);
}
// now both side exist and we can link the references
broker.serviceBrokerHelper().link(movie, "actors", true);
/*
alternative call
broker.serviceBrokerHelper().link(movie, true);
*/
broker.commitTransaction();
```

First store the main object and the references, then use `broker.serviceBrokerHelper().link(movie, "actors", true)` to link the main object with the references. In case of a m:n relation linking create all FK entries in the indirection table.

In the next examples we want to manually delete a `Project` object with a 1:n relation to class `SubProject`. In the example, the Project object has load all SubProject objects and we want to delete the Project but **don't** want to delete the referenced SubProjects too (don't ask if this make sense ;-)). SubProject has an FK to Project, so we first have to *unlink* the reference from the main object to the references to avoid integrity constraint violation. Then we can delete the main object:

```
broker.beginTransaction();
// first unlink the n-side references
broker.serviceBrokerHelper().unlink(project, "subProjects");

// update the n-side references, store SubProjects with nullified FK
Iterator it = project.getSubProjects().iterator();
while(it.hasNext())
{
    SubProject subProject = (SubProject) it.next();
    broker.store(subProject);
}

// now delete the main object
broker.delete(project);
broker.commitTransaction();
```

4.5.6.5. Using Proxy Classes

Proxy classes can be used for "lazy loading" aka "lazy materialization". Using Proxy classes can help you in reducing unneccessary database lookups.
There are two kind of proxy mechanisms available:

1. **Dynamic proxies** provided by OJB. They can simply be activated by setting certain switches in repository.xml. This is the solution recommemded for **most** cases.
2. **User defined proxies**. User defined proxies allow the user to write proxy implementations.

As it is important to understand the mechanics of the proxy mechanism I highly recommend to read this section before turning to the next sections "using dynamic proxies", "using a single proxy for a whole collection" and "using a proxy for a reference", covering dynamic proxies.

As a simple example we take a ProductGroup object `pg` which contains a collection of fifteen Article objects. Now we examine what happens when the ProductGroup is loaded from the database:

Without using proxies all fifteen associated Article objects are immediately loaded from the db, even if you are not interested in them and just want to lookup the description-attribute of the ProductGroup object.

If proxies are used, the collection is filled with fifteen proxy objects, that implement the same interface as the "real objects" but contain only an OID and a void reference. The fifteen article objects are not instantiated when the ProductGroup is initially materialized. Only when a method is invoked on such a proxy object will it load its "real subject" and delegate the method call to it. Using this dynamic delegation mechanism instantiation of persistent objects and database lookups can be minimized.

To use proxies, the persistent class in question (in our case the Article class) must implement an interface (for example InterfaceArticle). This interface is needed to allow replacement of the proper Article object with a proxy implementing the same interface. Have a look at the code:

```
    public class Article implements InterfaceArticle
    {
        /** maps to db-column "Artikel-Nr"; PrimaryKey*/
        protected int articleId;
        /** maps to db-column "Artikelname"*/
        protected String articleName;
        ...

        public int getArticleId()
        {
            return articleId;
        }
        public java.lang.String getArticleName()
        {
            return articleName;
        }
        ...
    }

    public interface InterfaceArticle
    {
        public int getArticleId();
        public java.lang.String getArticleName();
        ...
    }
```

```
public class ArticleProxy extends VirtualProxy implements InterfaceArticle
{
    public ArticleProxy(ojb.broker.Identity uniqueId, PersistenceBroker broker)
    {
        super(uniqueId, broker);
    }

    public int getArticleId()
    {
        return realSubject().getArticleId();
    }

    public java.lang.String getArticleName()
    {
        return realSubject().getArticleName();
    }

    private InterfaceArticle realSubject()
    {
        try
        {
            return (InterfaceArticle) getRealSubject();
        }
        catch (Exception e)
        {
            return null;
        }
    }
}
```

The proxy is constructed from the identity of the real subject. All method calls are delegated to the object returned by `realSubject()`.
This method uses getRealSubject() from the base class VirtualProxy:

```
    public Object getRealSubject() throws PersistenceBrokerException
    {
        return indirectionHandler.getRealSubject();
    }
```

The proxy delegates the the materialization work to its `IndirectionHandler`. If the real subject has not yet been materialized, a PersistenceBroker is used to retrieve it by its OID:

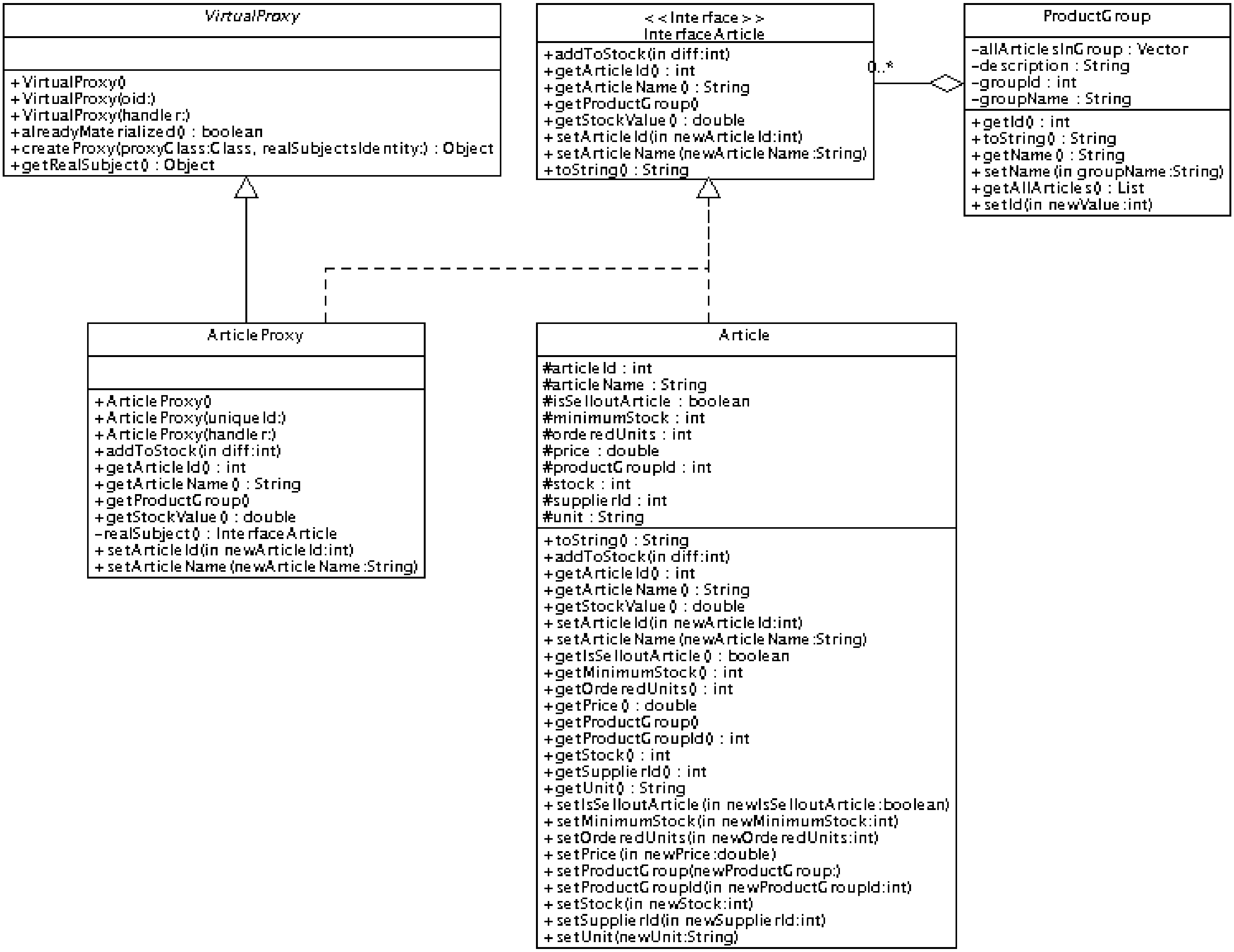
```
    public synchronized Object getRealSubject()
                    throws PersistenceBrokerException
    {
        if (realSubject == null)
        {
            materializeSubject();
        }
        return realSubject;
    }

    private void materializeSubject()
```

```
                        throws PersistenceBrokerException
    {
        realSubject = broker.getObjectByIdentity(id);
    }
```

To tell OJB to use proxy objects instead of materializing full Article objects we have to add the following section to the XML repository file:

```

    <class-descriptor
        class="org.apache.ojb.broker.Article"
        proxy="org.apache.ojb.broker.ArticleProxy"
        table="Artikel"
    >
    ...
```

The following class diagram shows the relationships between all above mentioned classes:


VirtualProxy
+VirtualProxy()
+VirtualProxy(oid:)
+VirtualProxy(handler:)
+alreadyMaterialized() : boolean
+createProxy(proxyClass:Class, realSubjectsIdentity:) : Object
+getRealSubject() : Object
<<Interface>>
InterfaceArticle
+addToStock(in diff:int)
+getArticleId() : int
+getArticleName() : String
+getProductGroup()
+getStockValue() : double
+setArticleId(in newArticleId:int)
+setArticleName(newArticleName:String)
+toString() : String
0..*
ProductGroup
-allArticlesInGroup : Vector
-description : String
-groupId : int
-groupName : String
+getId() : int
+toString() : String
+getName() : String
+setName(in groupName:String)
+getAllArticles() : List
+setId(in newValue:int)
ArticleProxy
+ArticleProxy()
+ArticleProxy(uniqueId:)
+ArticleProxy(handler:)
+addToStock(in diff:int)
+getArticleId() : int
+getArticleName() : String
+getProductGroup()
+getStockValue() : double
-realSubject() : InterfaceArticle
+setArticleId(in newArticleId:int)
+setArticleName(newArticleName:String)
Article
#articleId : int
#articleName : String
#isSelloutArticle : boolean
#minimumStock : int
#orderedUnits : int
#price : double
#productGroupId : int
#stock : int
#supplierId : int
#unit : String
+toString() : String
+addToStock(in diff:int)
+getArticleId() : int
+getArticleName() : String
+getStockValue() : double
+setArticleId(in newArticleId:int)
+setArticleName(newArticleName:String)
+getIsSelloutArticle() : boolean
+getMinimumStock() : int
+getOrderedUnits() : int
+getPrice() : double
+getProductGroup()
+getProductGroupId() : int
+getStock() : int
+getSupplierId() : int
+getUnit() : String
+setIsSelloutArticle(in newIsSelloutArticle:boolean)
+setMinimumStock(in newMinimumStock:int)
+setOrderedUnits(in newOrderedUnits:int)
+setPrice(in newPrice:double)
+setProductGroup(newProductGroup:)
+setProductGroupId(in newProductGroupId:int)
+setStock(in newStock:int)
+setSupplierId(in newSupplierId:int)
+setUnit(newUnit:String)

proxy image

Using Dynamic Proxies

The implementation of a proxy class is a boring task that repeats the same delegation scheme for each new class. To liberate the developer from this unproductive job OJB provides a dynamic proxy solution based on the JDK 1.3 dynamic proxy concept. (For JDK1.2 we ship a replacement for the required `java.lang.reflect` classes. Credits for this solution to ObjectMentor.) The basic idea of the dynamic proxy concept is to catch all method invocations on the not-yet materialized (loaded from database) object. When a method is called on the object, Java directs this call to the invocation handler registered for it (in OJB's case a class implementing the `org.apache.ojb.broker.core.proxy.IndirectionHandler` interface). This handler then materializes the object from the database and replaces the proxy with the real object. By default OJB uses the class `org.apache.ojb.broker.core.proxy.IndirectionHandlerDefaultImpl`. If you are interested in the mechanics have a look at this class.

To use a dynamic proxy for lazy materialization of Article objects we have to declare it in the repository.xml file.

```
<class-descriptor
  class="org.apache.ojb.broker.Article"
  proxy="dynamic"
  table="Artikel"
>
...
```

Just as with normal proxies, the persistent class in question (in our case the Article class) must implement an interface (for example InterfaceArticle) to be able to benefit from dynamic proxies.

Using a Single Proxy for a Whole Collection

A collection proxy represents a whole collection of objects, where as a proxy class represents a single object.
The advantage of this concept is a reduced number of db-calls compared to using proxy classes. A collection proxy only needs a **single** db-call to materialize all it's objects. This happens the first time its content is accessed (ie: by calling iterator();). An additional db-call is used to calculate the size of the collection if size() is called *before* loading the data. So collection proxy is mainly used as a deferred execution of a query.

OJB uses three specific proxy classes for collections:

1. **List proxies** are specific `java.util.List` implementations that are used by OJB to replace lists. The default set proxy class is `org.apache.ojb.broker.core.proxy.ListProxyDefaultImpl`
2. **Set proxies** are specific `java.util.Set` implementations that are used by OJB to replace sets. The default set proxy class is `org.apache.ojb.broker.core.proxy.SetProxyDefaultImpl`
3. **Collection proxies** are collection classes implementing the more generic `java.util.Collection` interface and are used if the collection is neither a list nor a set. The default collection proxy class is `org.apache.ojb.broker.core.proxy.CollectionProxyDefaultImpl`

Which of these proxy class is actually used, is determined by the `collection-class` setting of this collection. If none is specified in the repository descriptor, or if the specified class does not implement `java.util.List` nor `java.util.Set`, then the generic collection proxy is used.

The following mapping shows how to use a collection proxy for a relationship:

```
<!-- Definitions for
org.apache.ojb.broker.ProductGroupWithCollectionProxy -->
<class-descriptor
  class="org.apache.ojb.broker.ProductGroupWithCollectionProxy"
  table="Kategorien"
>
  <field-descriptor
     name="groupId"
     column="Kategorie_Nr"
     jdbc-type="INTEGER"
     primarykey="true"
  />
  ...
```

```
      <collection-descriptor
         name="allArticlesInGroup"
         element-class-ref="org.apache.ojb.broker.Article"
         proxy="true"
      >
         <inverse-foreignkey field-ref="productGroupId"/>
      </collection-descriptor>
    </class-descriptor>
```

The classes participating in this relationship do not need to implement a special interface to be used in a collection proxy.

Although it is possible to mix the collection proxy concept with the proxy class concept, it is not recommended because it increases the number of database calls.

Using a Proxy for a Reference

A proxy reference is based on the original proxy class concept. The main difference is that the ReferenceDescriptor defines when to use a proxy class and not the ClassDescriptor.
In the following mapping the class ProductGroup is **not** defined to be a proxy class in its ClassDescriptor. Only for shown relationship a proxy of ProductGroup should be used:

```
    <!-- Definitions for org.apache.ojb.broker.ArticleWithReferenceProxy -->
       <class-descriptor
          class="org.apache.ojb.broker.ArticleWithReferenceProxy"
          table="Artikel"
       >
          <field-descriptor
             name="articleId"
             column="Artikel_Nr"
             jdbc-type="INTEGER"
             primarykey="true"
             autoincrement="true"
          />
          ...
          <reference-descriptor
             name="productGroup"
             class-ref="org.apache.ojb.broker.ProductGroup"
         proxy="true"
          >
             <foreignkey field-ref="productGroupId"/>
          </reference-descriptor>
       </class-descriptor>
```

Because a proxy reference is only about the location of the definition, the referenced class must implement a special interface (see using proxy classes).

Customizing the proxy mechanism

Both the dynamic and the collection proxy mechanism can be customized by supplying a user-defined implementation.

For dynamic proxies you can provide your own invocation handler which implements the `org.apache.ojb.broker.core.proxy.IndirectionHandler` interface. See OJB's default implementation `org.apache.ojb.broker.core.proxy.IndirectionHandlerDefaultImpl` for details on how to implement such an invocation handler.

Each of the three collection proxy classes can be replaced by a user-defined class. The only requirement is that such a class implements both the corresponding interface (`java.util.Collection`, `java.util.List`, or `java.util.Set`) as well as the `org.apache.ojb.broker.ManageableCollection` interface.

Proxy implementations are configured in the ojb properties file. These are the relevant settings:

```
    ...
        #-----------------------------------------------------------------------------------------
        # IndirectionHandler
        #----------------------------------------------------------------------------------------
        # The IndirectionHandlerClass entry defines the class to be used by OJB's proxies to
        # handle method invocations
        #
        IndirectionHandlerClass=org.apache.ojb.broker.core.proxy.IndirectionHandlerDefaultImpl
        #
        #----------------------------------------------------------------------------------------
        # ListProxy
        #----------------------------------------------------------------------------------------
        # The ListProxyClass entry defines the proxy class to be used for collections that
        # implement the java.util.List interface.
        #
        ListProxyClass=org.apache.ojb.broker.core.proxy.ListProxyDefaultImpl
        #
        #----------------------------------------------------------------------------------------
        # SetProxy
        #----------------------------------------------------------------------------------------
        # The SetProxyClass entry defines the proxy class to be used for collections that
        # implement the java.util.Set interface.
        #
        SetProxyClass=org.apache.ojb.broker.core.proxy.SetProxyDefaultImpl
        #
        #----------------------------------------------------------------------------------------
        # CollectionProxy
        #----------------------------------------------------------------------------------------
        # The CollectionProxyClass entry defines the proxy class to be used for collections that
        # do not implement java.util.List or java.util.Set.
        #
        CollectionProxyClass=org.apache.ojb.broker.core.proxy.CollectionProxyDefaultImpl
    ...
```

4.5.6.6. Type and Value Conversions

Say your database column contains INTEGER values but you have to use boolean attributes in your Domain objects. You need a type and value mapping described by a FieldConversion!

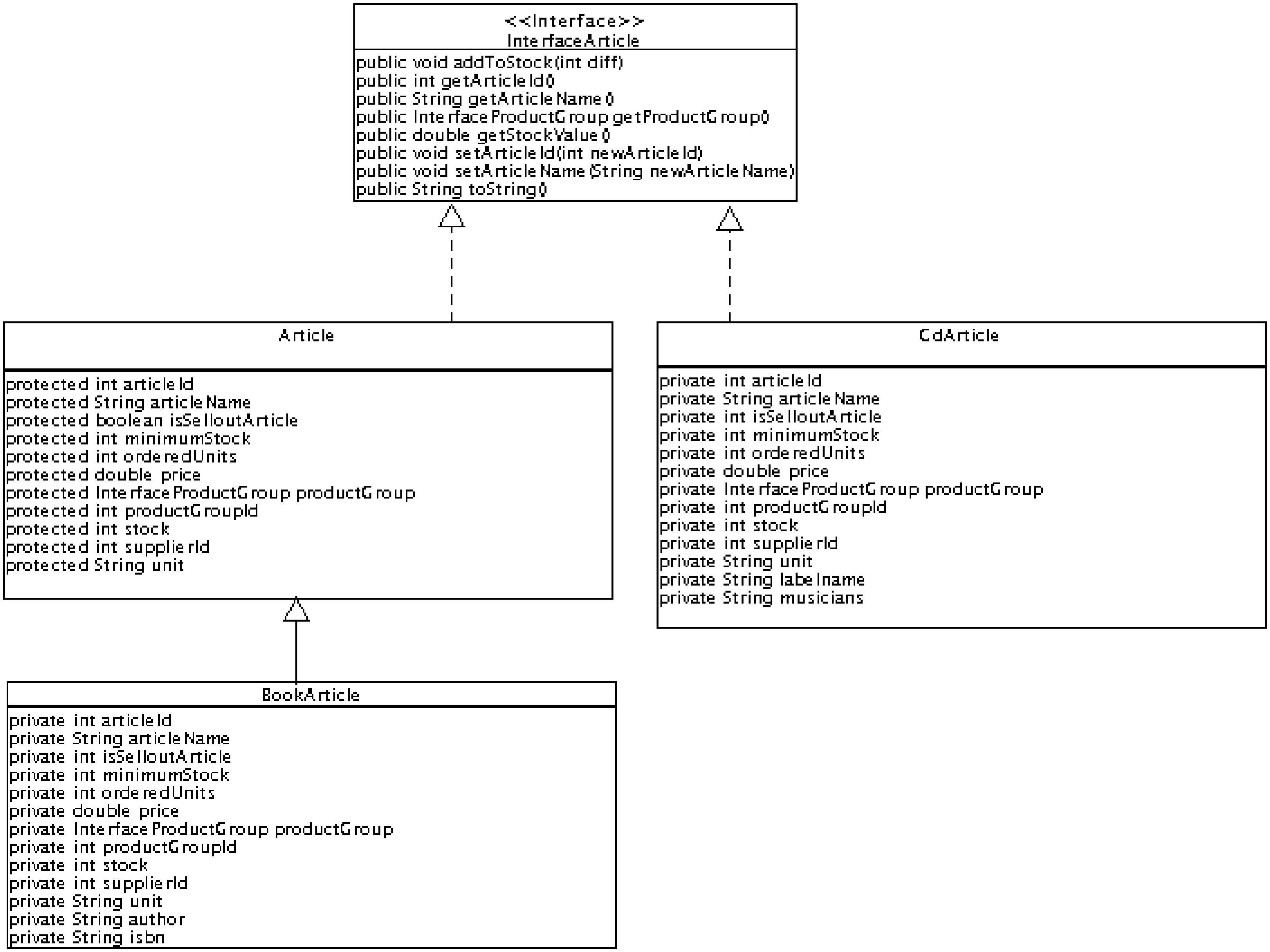
4.5.7. Advanced Technique

4.5.7.1. Introduction

4.5.7.2. 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)


<<Interface>>
InterfaceArticle
public void addToStock(int diff)
public int getArticleId()
public String getArticleName()
public InterfaceProductGroup getProductGroup()
public double getStockValue()
public void setArticleId(int newArticleId)
public void setArticleName(String newArticleName)
public String toString()
Article
protected int articleId
protected String articleName
protected boolean isSelloutArticle
protected int minimumStock
protected int orderedUnits
protected double price
protected InterfaceProductGroup productGroup
protected int productGroupId
protected int stock
protected int supplierId
protected String unit
CdArticle
private int articleId
private String articleName
private int isSelloutArticle
private int minimumStock
private int orderedUnits
private double price
private InterfaceProductGroup productGroup
private int productGroupId
private int stock
private int supplierId
private String unit
private String labelname
private String musicians
BookArticle
private int articleId
private String articleName
private int isSelloutArticle
private int minimumStock
private int orderedUnits
private double price
private InterfaceProductGroup productGroup
private int productGroupId
private int stock
private int supplierId
private String unit
private String author
private String isbn

polymorphism.gif

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`.

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.

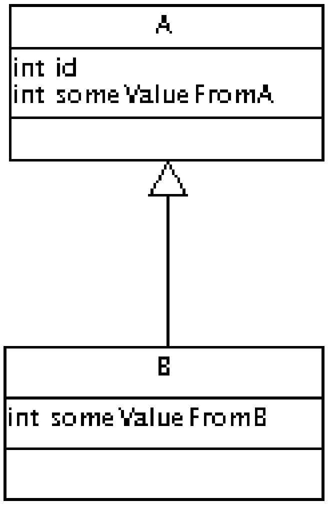
In other cases it may be neccessary to treat certain classes as implementors of an interface or as derived from a base even if they are not.

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>
```

4.5.7.3. 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.

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.

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.

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.

4.5.7.4. 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`.

4.5.7.5. 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]().

4.5.7.6. 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.

4.5.7.7. 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.

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"
>
```

4.5.7.8. 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.

4.5.7.9. 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);
    }
}
```

4.5.7.10. 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.

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>
```

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).

4.5.7.11. 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.

4.5.7.12. Metadata runtime changes

This was described in metadata section.

4.5.8. OJB Queries

4.5.8.1. Introduction

This tutorial describes the use of the different queries mechanisms. The sample code shown here is taken mainly from JUnit test classes. The junit test source can be found under `[db-ojb]/src/test` in the source distribution.

4.5.8.2. Query by Criteria

In this section you will learn how to use the query by criteria. The classes are located in the package `org.apache.ojb.broker.query`. Using query by criteria you can either query for whole objects (ie. person) or you can use report queries returning row data.

A query consists mainly of the following parts:

1. the class of the objects to be retrieved
2. a list of criteria
3. a DISTINCT flag
4. additional ORDER BY and GROUP BY

OJB offers a QueryFactory to create a new Query. Although the constructors of the query classes are public using the QueryFactory is the preferred way to create a new query.

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

To create a DISTINCT-Query, simply add **true** as third parameter.

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

Each criterion stands for a column in the SQL-WHERE-clause.

```
Criteria crit = new Criteria();
crit.addEqualTo("upper(firstname)", "TOM");
crit.addEqualTo("lastname", "hanks");
Query q = QueryFactory.newQuery(Person.class, crit);
```

This query will generate an SQL statement like this:

```
SELECT ... FROM PERSON WHERE upper(FIRSTNAME) = "TOM" AND LASTNAME = "hanks";
```

OJB supports **functions** in field criteria ie. upper(firstname). When converting a field name to a database column name, the function is added to the generated sql. OJB does not and can not verify the correctness of the specified function, an illegal function will produce an SqlException.

Query Criteria

OJB provides selection criteria for almost any SQL-comparator. In most cases you do not have to deal directly with the implementing classes like *EqualToCriteria*. The *Criteria* class provides factory methods for the appropriate classes. There are four kinds of factory methods:

- create criteria to compare a field to a value: ie. addEqualTo("firstname", "tom");
- create criteria to compare a field to another field: ie. addEqualToField("firstname", "other_field");
- create criteria to check null value: ie. addIsNull("firstname");
- create a raw sql criteria: ie: addSql("REVERSE(name) like 're%'");

The following list shows some of the factory methods to compare a field to a value:

- addEqualTo
- addLike
- addGreaterOrEqualThan
- addGreaterThan
- addLike
- addBetween , this methods has two value parameters
- addIn , this method uses a Collection as value parameter
- and of course there negative forms

This list shows some factory methods to compare a field to another field, all those methods end on ...field:

- addEqualToField
- addGreaterThanField
- and of course there negative forms

in / not in

Some databases limit the number of parameters in an IN-statement.
If the limit is reached OJB will split up the IN-Statement into multiple Statements, the limit is set to 3 for the following sample:

```
SELECT ... FROM Artikel A0 WHERE A0.Kategorie_Nr IN ( ? , ? , ? )
OR A0.Kategorie_Nr IN ( ? , ? ) ORDER BY 7 DESC
```

The IN-limit for prefetch can be defined in OJB.properties:

```
...
# The SqlInLimit entry limits the number of values in IN-sql
# statement, -1 for no limits. This hint is used in Criteria.
SqlInLimit=200
...
```

and / or

All selection criteria added to a criteria set using the above factory methods will be **AND**ed in the WHERE-clause. To get an **OR** combination two criteria sets are needed. These sets are combined using addOrCriteria:

```
Criteria crit1 = new Criteria();
crit1.addLike("firstname", "%o%");
crit1.addLike("lastname", "%m%");
Criteria crit2 = new Criteria();
crit2.addEqualTo("firstname", "hank");

crit1.addOrCriteria(crit2);
Query q = QueryFactory.newQuery(Person.class, crit1);

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

This query will generate an SQL statement like this:

```
SELECT ... WHERE (FIRSTNAME LIKE "%o%") AND LASTNAME
LIKE "%m%" OR FIRSTNAME = "hank"
```

negating the criteria

A criteria can be negated to obtain **NOT** in the WHERE-clause:

```
Criteria crit1 = new Criteria();
crit1.addLike("firstname", "%o%");
crit1.addLike("lastname", "%m%");
crit1.setNegative(true);

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

This query will generate an SQL statement like this:

```
SELECT ... WHERE NOT (FIRSTNAME LIKE "%o%" AND LASTNAME LIKE "%m%")
```

ordering and grouping

The following methods of QueryByCriteria are used for ordering and grouping:

- addOrderByAscending(String anAttributeName);
- addOrderByDescending(String anAttributeName);
- addGroupBy(String anAttributeName); this method is used for report queries

You can of course have multiple order by and group by clauses, simply repeat the addOrderBy.

```
crit = new Criteria();
query = new QueryByCriteria(Person.class, crit);
query.addOrderByDescending("id");
query.addOrderByAscending("lastname");
broker.getCollectionByQuery(query);
```

The code snippet will query all Persons and order them by **attribute** "id" descending and "lastname" ascending. The query will produce the following SQL-statement using column numbers in the ORDER BY clause:

```
SELECT A0.ID,A0.FIRSTNAME,A0.LASTNAME FROM
PERSON A0 ORDER BY 1 DESC, 3
```

When you use the **column** name "LASTNAME" instead of the **attribute** name "lastname" (query.addOrderBy("LASTNAME");), an additional column named "LASTNAME" without alias will be added.

```
SELECT A0.ID,A0.FIRSTNAME,A0.LASTNAME,LASTNAME FROM
PERSON A0 ORDER BY 1 DESC,4
```

If there are multiple tables with a column "LASTNAME" the SQL-Statement will produce an error, so it's better to always use attribute names.

subqueries

Subqueries can be used instead of values in selection criteria. The subquery should thus be a ReportQuery.
The following example queries all articles having a price greator or equal than the average price of articles named 'A%':

```
ReportQueryByCriteria subQuery;
Criteria subCrit = new Criteria();
Criteria crit = new Criteria();

subCrit.addLike("articleName", "A%");
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);
```

It's also possible to build a subquery with attributes referencing the enclosing query. These attributes have to use a special prefix **Criteria.PARENT_QUERY_PREFIX**.

The following example queries all product groups having more than 10 articles:

```
ReportQueryByCriteria subQuery;
Criteria subCrit = new Criteria();
Criteria crit = new Criteria();

subCrit.addEqualToField("productGroupId", Criteria.PARENT_QUERY_PREFIX + "groupId");
subQuery = QueryFactory.newReportQuery(Article.class, subCrit);
subQuery.setAttributes(new String[] { "count(productGroupId)" });

crit.addGreaterThan(subQuery, "10"); // MORE than 10 articles
crit.addLessThan("groupId", new Integer(987654));
Query q = QueryFactory.newQuery(ProductGroup.class, crit);

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

joins

Joins resulting from **path expressions** ("relationship.attribute") in criteria are automatically handled by OJB. Path expressions are supported for all relationships 1:1, 1:n and m:n (decomposed and non-decomposed) and can be nested.

The following sample looks for all articles belonging to the product group "Liquors". Article and product group are linked by the relationship "productGroup" in class Article:

```
<!-- Definitions for org.apache.ojb.ojb.broker.Article -->
   <class-descriptor
         class="org.apache.ojb.broker.Article"
         proxy="dynamic"
         table="Artikel"
   >
      ...
      <reference-descriptor
         name="productGroup"
         class-ref="org.apache.ojb.broker.ProductGroup"
      >
         <foreignkey field-ref="productGroupId"/>
      </reference-descriptor>
   </class-descriptor>

   <class-descriptor
         class="org.apache.ojb.broker.ProductGroup"
         proxy="org.apache.ojb.broker.ProductGroupProxy"
         table="Kategorien"
   >
      ...
      <field-descriptor
         name="groupName"
         column="KategorieName"
         jdbc-type="VARCHAR"
      />
      ...
   </class-descriptor>
```

The path expression includes the 1:1 relationship "productGroup" and the attribute "groupName":

```
Criteria crit = new Criteria();
crit.addEqualTo("productGroup.groupName", "Liquors");
Query q = QueryFactory.newQuery(Article.class, crit);

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

If path expressions refer to a class having **extents**, the tables of the extent classes participate in the JOIN and the criteria is ORed. The shown sample queries all ProductGroups having an Article named 'F%'. The path expression 'allArticlesInGroup' refers to the class Articles which has two extents: Books and CDs.

```
Criteria crit = new Criteria();
crit.addLike("allArticlesInGroup.articleName", "F%");
QueryByCriteria q = QueryFactory.newQuery(ProductGroup.class, crit, true);

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

This sample produces the following SQL:

```
SELECT DISTINCT A0.KategorieName,A0.Kategorie_Nr,A0.Beschreibung
FROM Kategorien A0
INNER JOIN Artikel A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
LEFT OUTER JOIN BOOKS A1E0 ON A0.Kategorie_Nr=A1E0.Kategorie_Nr
LEFT OUTER JOIN CDS A1E1 ON A0.Kategorie_Nr=A1E1.Kategorie_Nr
WHERE A1.Artikelname LIKE  'F%'  OR
A1E0.Artikelname LIKE  'F%'  OR
A1E1.Artikelname LIKE  'F%'
```

OJB tries to do it's best to automatically use **outer** joins where needed. This is currently the case for classes having extents and ORed criteria. But you can force the SQLGenerator to use outer joins where you find it useful.
This is done by the method *QueryByCriteria#setPathOuterJoin(String)*.

```
ReportQueryByCriteria query;
Criteria crit;
Iterator result1, result2;

crit = new Criteria();

query = new ReportQueryByCriteria(Person.class, crit);
query.setAttributes(new String[] { "id", "name", "vorname", "sum(konti.saldo)" });
query.addGroupBy(new String[]{ "id", "name", "vorname" });

result1 = broker.getReportQueryIteratorByQuery(query);

query.setPathOuterJoin("konti");
result2 = broker.getReportQueryIteratorByQuery(query);
```

The first query will use an inner join for relationship "konti", the second an outer join.

user defined alias

This feature allows to have **multiple** aliases for the same table. The standard behaviour of OJB is to build one alias for one relationship.

Suppose you have two classes Issue and Keyword and there is a 1:N relationship between them. Now you want to retrieve Issues by querying on Keywords. Suppose you want to retrieve all Issues with keywords 'JOIN' and 'ALIAS'. If these values are stored in the attribute 'value' of Keyword, OJB generates a query that contains " A1.value = 'JOIN' AND A1.value = 'ALIAS' " in the where-clause. Obviously, this will not work, no hits will occur because A1.value can not have more then 1 value at the time !

For the examples below, suppose you have the following classes (pseudo-code):

```
class Container
    int id
    Collection allAbstractAttributes

class AbstractAttribute
    int id
    inf ref_id
    String name
    String value
    Collection allAbstractAttributes
```

OJB maps these classes to separate tables where it maps allAbstractAttributes using a collectiondescriptor to AbstractAttribute using ref_id as inverse foreignkey on Container for the collection descriptor.

For demo purposes : AbstractAttribute also has a collection of abstract attributes.

```
Criteria crit1 = new Criteria();
crit1.setAlias("company");                 // set an alias
crit1.addEqualTo("allAbstractAttributes.name", "companyName");
crit1.addEqualTo("allAbstractAttributes.value", "iBanx");

Criteria crit2 = new Criteria();
crit2.setAlias("contact");                 // set an alias
crit2.addEqualTo("allAbstractAttributes.name", "contactPerson");
crit2.addLike("allAbstractAttributes.value", "janssen");

Criteria crit3 = new Criteria();
crit3.addEqualTo("allAbstractAttributes.name", "size");
crit3.addGreaterThan("allAbstractAttributes.value", new Integer(500));

crit1.addAndCriteria(crit2);
crit1.addAndCriteria(crit3);

q = QueryFactory.newQuery(Container.class, crit1);
q.addOrderBy("company.value");     // user alias
```

The generated query will be as follows. Note that the alias name 'company' does not show up in the SQL.

```
SELECT DISTINCT A0.ID, A1.VALUE
FROM CONTAINER A0 INNER JOIN ABSTRACT_ATTRIBUTE A1
     ON A0.ID=A1.REF_ID
     INNER JOIN ABSTRACT_ATTRIBUTE A2
     ON A0.ID=A2.REF_ID
     INNER JOIN ABSTRACT_ATTRIBUTE A3
     ON A0.ID=A3.REF_ID
WHERE (( A0.NAME =  'companyName' ) AND  (A0.VALUE =  'iBanx' )) AND
      (( A1.NAME =  'contactPerson' ) AND  (A1.VALUE LIKE '%janssen%' )) AND
      (( A2.NAME =  'size' ) AND  (A2.VALUE =  '500' ))
ORDER BY 2
```

The next example uses a report query.

```
Criteria crit1 = new Criteria();
crit1.setAlias("ALIAS1");
crit1.addEqualTo("allAbstractAttributes.allAbstractAttributes.name", "xxxx");
crit1.addEqualTo("allAbstractAttributes.allAbstractAttributes.value", "hello");

Criteria crit2 = new Criteria();
crit2.setAlias("ALIAS2");
crit2.addEqualTo("allAbstractAttributes.name", "yyyy");
crit2.addLike("allAbstractAttributes.value", "");

crit1.addAndCriteria(crit2);

q = QueryFactory.newReportQuery(Container.class, crit1);

String[] cols = { id, "ALIAS2.name", "ALIAS2.name", "ALIAS1.name", "ALIAS1.name" };
q.setAttributes(cls);
```

The generated query will be:

```
SELECT DISTINCT A0.ID, A1.NAME, A1.VALUE, A2.NAME, A2.VALUE
FROM CONTAINER A0 INNER JOIN ABSTRACT_ATTRIBUTE A1
     ON A0.ID=A1.REF_ID
     INNER JOIN ABSTRACT_ATTRIBUTE A2
     ON A1.ID=A2.REF_ID
WHERE (( A2.NAME =  'xxxx' ) AND  (A2.VALUE =  'hello' )) AND
      (( A1.NAME =  'yyyy' ) AND  (A2.VALUE LIKE '%%' )) AND
ORDER BY 2
```

Note:

When you define an alias for a criteria, you have to make sure that *all* attributes used in this criteria belong to the *same* class. If you break this rule OJB will probably use a wrong ClassDescriptor to resolve your attributes !

class hints

This feature allows the user to specify which class of an extent to use for a path-segment. The standard behaviour of OJB is to use the base class of an extent when it resolves a path-segment.

In the following sample the path **allArticlesInGroup** points to class Article, this is defined in the repository.xml. Assume we are only interested in ProductGroups containing CdArticles performed by Eric Clapton or Books authored by Eric Clapton, a class hint can be defined for the path. This hint is defined by:
Criteria#**addPathClass**("allArticlesInGroup", CdArticle.class);

```
//
// find a ProductGroup with a CD or a book by a particular artist
//
String artistName = new String("Eric Clapton");
crit1 = new Criteria();
crit1.addEqualTo("allArticlesInGroup.musicians", artistName);
crit1.addPathClass("allArticlesInGroup", CdArticle.class);

crit2 = new Criteria();
crit2.addEqualTo("allArticlesInGroup.author", artistName);
crit2.addPathClass("allArticlesInGroup", BookArticle.class);

crit1.addOrCriteria(crit2);

query = new QueryByCriteria(ProductGroup.class, crit1);
broker.getObjectByQuery(query);
```

Note:

This feature is also available in class QueryByCriteria but using it on Criteria-level provides additional flexibility. QueryByCriteria#addPathClass is only useful for ReportQueries to limit the class of the selected columns.

prefetched relationships

This feature can help to minimize the number of queries when reading objects with relationships. In our Testcases we have ProductGroups with a one to many relationship to Articles. When reading the ProductGroups one query is executed to get the ProductGroups and for **each** ProductGroup another query is executed to retrieve the Articles.

With prefetched relationships OJB tries to read all Articles belonging to the ProductGroups in **one** query. See further down why one query is not always possible.

```
Criteria crit = new Criteria();
crit.addLessOrEqualThan("groupId", new Integer(5));

QueryByCriteria q = QueryFactory.newQuery(ProductGroup.class, crit);
q.addOrderByDescending("groupId");
q.addPrefetchedRelationship("allArticlesInGroup");

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

The first query reads all matching ProductGroups:

```
SELECT ... FROM Kategorien A0 WHERE
A0.Kategorie_Nr <= ? ORDER BY 3 DESC
```

The second query retrieves Articles belonging to the ProductGroups read by the first query:

```
SELECT ... FROM Artikel A0 WHERE A0.Kategorie_Nr
IN ( ? , ? , ? , ? , ? ) ORDER BY 7 DESC
```

After reading all Articles they are associated with their ProductGroup.

Note:

This function is not yet supported for relationships using Arrays.

Some databases limit the number of parameters in an IN-statement. If the limit is reached OJB will split up the second query into multiple queries, the limit is set to 3 for the following sample:

```
SELECT ... FROM Artikel A0 WHERE A0.Kategorie_Nr
IN ( ? , ? , ? ) ORDER BY 7 DESC
SELECT ... FROM Artikel A0 WHERE A0.Kategorie_Nr
IN ( ? , ? ) ORDER BY 7 DESC
```

The IN-limit for prefetch can be defined in OJB.properties SqlInLimit.

querying for objects

OJB queries return **complete** objects, that means all instance variables are filled and all 'auto-retrieve' relationships are loaded. Currently there's no way to retrieve partially loaded objects (ie. only first- and lastname of a person).

More info about manipulation of metadata setting here.

Report Queries

Report queries are used to retrieve row data, not 'real' business objects. A row is an array of Object. With these queries you can define what attributes of an object you want to have in the row. The attribute names may also contain path expressions like 'owner.address.street'. To define the attributes use ReportQuery **#setAttributes(String[] attributes)**.

The following ReportQuery retrieves the name of the ProductGroup, the name of the Article etc. for all Articles named like "C%":

```
Criteria crit = new Criteria();
Collection results = new Vector();
crit.addLike("articleName", "C%");
ReportQueryByCriteria q = QueryFactory.newReportQuery(Article.class, crit);
q.setAttributes(new String[] { "productGroup.groupName","articleId", "articleName", "price" });

Iterator iter = broker.getReportQueryIteratorByQuery(q);
```

The ReportQuery returns an Iterator over a Collection of Object[4] ([String, Integer, String, Double]).

Limitations of Report Queries

ReportQueries should not be used with columns referencing classes with extents. Assume we want to select all ProductGroups and summarize the amount and prize of items in stock per group. The class Article referenced by **allArticlesInGroup** has the extents Books and CDs.

```
Criteria crit = new Criteria();
Collection results = new Vector();
ReportQueryByCriteria q = QueryFactory.newReportQuery(ProductGroup.class, crit);
q.setAttributes(new String[] { "groupName", "sum(allArticlesInGroup.stock)",
"sum(allArticlesInGroup.price)" });
q.addGroupBy("groupName");

Iterator iter = broker.getReportQueryIteratorByQuery(q);
```

The ReportQuery looks quite reasonable, but it will produce an SQL not suitable for the task:

```
SELECT A0.KategorieName,sum(A1.Lagerbestand),sum(A1.Einzelpreis)
FROM Kategorien A0
```

```
LEFT OUTER JOIN artikel A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
LEFT OUTER JOIN books A1E2 ON A0.Kategorie_Nr=A1E2.Kategorie_Nr
LEFT OUTER JOIN cds A1E1 ON A0.Kategorie_Nr=A1E1.Kategorie_Nr
GROUP BY A0.KategorieName
```

This SQL will select the columns "Lagerbestand" and "Einzelpreis" from one extent only, and for ProductGroups having Articles, Books and CDs the result is wrong!

As a workaround the query can be "reversed". Instead of selection the ProductGroup we go for the Articles:

```
Criteria crit = new Criteria();
Collection results = new Vector();
ReportQueryByCriteria q = QueryFactory.newReportQuery(Article.class, crit);
q.setAttributes(new String[] { "productGroup.groupName", "sum(stock)", "sum(price)" });
q.addGroupBy("productGroup.groupName");
```

This ReportQuery executes the following three selects (one for each concrete extent) and produces better results.

```
SELECT  A1.KategorieName,sum(A0.Lagerbestand),sum(A0.Einzelpreis)
FROM artikel A0
INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
GROUP BY A1.KategorieName

SELECT  A1.KategorieName,sum(A0.Lagerbestand),sum(A0.Einzelpreis)
FROM cds A0
INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
GROUP BY A1.KategorieName

SELECT  A1.KategorieName,sum(A0.Lagerbestand),sum(A0.Einzelpreis)
FROM books A0
INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
GROUP BY A1.KategorieName
```

Of course there's also a drawback here: the same ProductGroup may be selected several times, so to get the correct sum, the results of the ProductGroup has to be added. In our sample the ProductGroup "Books" will be listed three times.

After listing so many drawbacks and problems, here's the SQL the produces the desired result. This is a manually created SQL, not generated by OJB. Unfortunately it's not fully supported by some DBMS because of "union" and sub-selects.

```
select KategorieName, sum(lagerbestand), sum(einzelpreis)
from
(
        SELECT  A1.KategorieName,A0.Lagerbestand,A0.Einzelpreis
        FROM artikel A0
        INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr

        union

        SELECT  A1.KategorieName,A0.Lagerbestand,A0.Einzelpreis
        FROM books A0
        INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr

        union

        SELECT  A1.KategorieName,A0.Lagerbestand,A0.Einzelpreis
        FROM cds A0
        INNER JOIN Kategorien A1 ON A0.Kategorie_Nr=A1.Kategorie_Nr
)
group by kategorieName
```

4.5.8.3. ODMG OQL

4.5.8.4. JDO queries

4.5.9. Metadata handling

4.5.9.1. Introduction

To make OJB proper work information about the used databases (more info see connection handling) and sequence managers is needed. Henceforth these metadata information is called **connection metadata**.

Further on OJB needs information about the persistent objects and object relations, henceforth this information is called **(persistent) object metadata**.

All metadata information need to be stored in the OJB repository file.

The *connection metadata* are completely decoupled from the *persistent object metadata*. Thus it is possible to use the same *object metadata* on different databases.
But it is also possible to use different *object metadata* profiles .

In OJB there are several ways to make metadata information available:

- using xml configuration files parsed at start up by OJB
- set metadata instances at runtime by building metadata class instances at runtime
- parse additional xml configuration files (additional repository files) and merge at runtime

All classes used for managing metadata stuff can be find under `org.apache.ojb.broker.metadata.*`-package.
The main class for metadata handling and entry point for metadata manipulation at runtime is `org.apache.ojb.broker.metadata.MetadataManager` .

4.5.9.2. When does OJB read metadata

By default all metadata is read at startup of OJB, when the first call to `PersistenceBrokerFactory` (directly or by a top-level api) or `MetadataManager` class was done.

OJB expects a repository file at startup, but it is also possible to start OJB without an repository file or only load connection metadata and object metadata at runtime or what ever combination fit your requirements.

4.5.9.3. Connection metadata

The *connection metadata* encapsulate all information referring to used database and must be declared in OJB repository file.
For each database a *jdbc-connection-descriptor* must be declared. This element encapusaltes the connection specific metadata information.

The *JdbcConnectionDescriptor* instances are managed by `org.apache.ojb.broker.metadata.ConnectionRepository`

Load and merge connection metadata

It is possible to load additional connection metadata at runtime and merge it with the existing one. The used repository files have to be valid against the repository.dtd:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE descriptor-repository SYSTEM "repository.dtd">

<descriptor-repository version="1.0" isolation-level="read-uncommitted">
    <jdbc-connection-descriptor
        jcd-alias="runtime"
        platform="Hsqldb"
        jdbc-level="2.0"
        driver="org.hsqldb.jdbcDriver"
        protocol="jdbc"
        subprotocol="hsqldb"
        dbalias="../OJB_FarAway"
        username="sa"
        password=""
        batch-mode="false"
    >

        <object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
            <attribute attribute-name="timeout" attribute-value="900"/>
            <attribute attribute-name="autoSync" attribute-value="true"/>
        </object-cache>

        <connection-pool
            maxActive="5"
            whenExhaustedAction="0"
            validationQuery="select count(*) from OJB_HL_SEQ"
        />

        <sequence-manager className="org.apache.ojb.broker.util.sequence.SequenceManagerHighLowImpl">
            <attribute attribute-name="grabSize" attribute-value="5"/>
        </sequence-manager>
    </jdbc-connection-descriptor>

    <!-- user/passwd at runtime required -->
    <jdbc-connection-descriptor
        jcd-alias="minimal"
        platform="Hsqldb"
        jdbc-level="2.0"
        driver="org.hsqldb.jdbcDriver"
        protocol="jdbc"
        subprotocol="hsqldb"
        dbalias="../OJB_FarAway"
    >
    </jdbc-connection-descriptor>
</descriptor-repository>
```

In the above additional repository file two new *jdbc-connection-descriptor* (new databases) *runtime* and *minimal* are declared, to load and merge the additional connection metadata the *MetadataManager was used:*

```
// get MetadataManager instance
MetadataManager mm = MetadataManager.getInstance();

// read connection metadata from repository file
ConnectionRepository cr = mm.readConnectionRepository("valid path/url to repository file");

// merge new connection metadata with existing one
mm.mergeConnectionRepository(cr);
```

After the merge the access to the new databases is ready for use.

4.5.9.4. Persistent object metadata

The *object metadata* encapsulate all information referring to the persistent capable java objects and the associated tables in database. *Object metadata* must be declared in OJB repository file.
Each persistence capable java object must be declared in a corresponding class-descriptor.

The *ClassDescriptor* instances are managed by `org.apache.ojb.broker.metadata.DescriptorRepository` . Per default OJB use only **one global instance** of this class - it's the repository file read at startup of OJB. But it is possible to change the global use repository:

```
// get MetadataManager instance
MetadataManager mm = MetadataManager.getInstance();

mm.setDescriptor(myGlobalRepository, true);
```

Load and merge object metadata

It is possible to load additional object metadata at runtime and merge it with the existing one. The used repository files have to be valid against the repository.dtd:

An additional repository file may look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE descriptor-repository SYSTEM "repository.dtd">

<descriptor-repository version="1.0" isolation-level="read-uncommitted">

    <class-descriptor
      class="org.my.MyObject"
      table="MY_OBJ"
    >
        <field-descriptor
         name="id"
         column="OBJ_ID"
         jdbc-type="INTEGER"
         primarykey="true"
         autoincrement="true"
        />

        <field-descriptor
         name="name"
         column="NAME"
         jdbc-type="VARCHAR"
        />
    </class-descriptor>
</descriptor-repository>
```

To load and merge the object metadata of the additional repository files first read the metadata using the *MetadataManager* .

```
// get MetadataManager instance
MetadataManager mm = MetadataManager.getInstance();

// read the additional repository file
DescriptorRepository dr = mm.readDescriptorRepository("valid path/url to repository file");

// merge the new class-descriptor with existing object metadata
mm.mergeDescriptorRepository(dr);
```

It is also possible to keep the different *object metadata* for the same classes parallel by using *metadata profiles* .

Global object metadata changes

The *MetadataManager* provide several methods to read/set and manipulate object metadata.

Per default OJB use a global instance of class *DescriptorRepository* to manage all *object metadata*. This means that all *PersistenceBroker* instances (kernel component used by all top-level api) use the same object metadata.

So changes of the object metadata (e.g. remove of a CollectionDescriptor instance from a ClassDescriptor) will be seen immediately by all *PersistenceBroker* instances. This is in most cases not the favoured behaviour and OJB supports per thread changes of object metadata.

Per thread metadata changes

Per default the manager handle one global *DescriptorRepository* for all calling threads (keep in mind PB-api is not threadsafe, thus each thread use it's own PersistenceBroker instance), but it is ditto possible to use different *metadata profiles* in a per thread manner - profiles means different instances of DescriptorRepository objects. Each thread/PersistenceBroker instance can be associated with a specific *DescriptorRepository* instance. All made object metadata changes only will be seen by the PersistenceBroker instances using the same DescriptorRepository instance. In theory each PersistenceBroker instance could be associated with a separate instance of object metadata, but the recommended way is to use metadata profiles.

To enable the use of different *DescriptorRepository* instances for each thread do:

```
MetadataManager mm = MetadataManager.getInstance();
// tell the manager to use per thread mode
mm.setEnablePerThreadChanges(true);
...
```

This can be done e.g. at start up or at runtime when it's needed. If method `setEnablePerThreadChanges` is set *false* only the *global DescriptorRepository* was used. Now it's possible to use dedicated DescriptorRepository instances per thread:

```
// e.g get a coppy of the global repository
DescriptorRepository dr = mm.copyOfGlobalRepository();
// now we can manipulate the persistent object metadata of the copy
......

// set the changed repository for current thread
mm.setDescriptor(dr);

// now let this thread lookup a PersistenceBroker instance
// with the modified metadata
// all other threads use still the global object metadata
PersistenceBroker broker = PersistenceBrokerFactory.createPersistenceBroker(myKey)
```

Note:

Set object metadata (setting of the DescriptorRepository) before lookup the PersistenceBroker instance for current thread, because the metadata was bound to the PersistenceBroker instance at lookup.

Object metadata profiles

MetadataManager was shipped with a simple mechanism to add, remove and load different *persistent objects metadata profiles* (different DescriptorRepository instances) in a per thread manner. Use method *addProfile* to add different persistent object metadata profiles, method *removeProfile* to remove profiles and *loadProfile* load a profile for the calling thread.

```
// get MetadataManager instance
MetadataManager mm = MetadataManager.getInstance();

// enable per thread mode if not done before
mm.setEnablePerThreadChanges(true);

// Load additional object metadata by parsing an repository file
DescriptorRepository dr_1 = mm.readDescriptorRepository("pathOrURLtoFile_1");
DescriptorRepository dr_2 = mm.readDescriptorRepository("pathOrURLtoFile_2");

// add  profiles
mm.addProfile("global", mm.copyOfGlobalRepository());
mm.addProfile("guest", dr_1);
mm.addProfile("admin", dr_2);

// now load a specific profile
mm.loadProfile("admin");
broker = PersistenceBrokerFactory.defaultPersistenceBroker();
```

After the *loadProfile* call all PersistenceBroker instances will be associated with the *admin* profile.

Note:

Method *loadProfile* only proper work if the per thread mode is enabled.

Reference runtime changes on per query basis

FIXME (arminw):

Changes of reference settings on a per query basis will be supported with next upcoming release 1.1

Pitfalls

OJB's flexibility of *metadata handling* demanded specific attention on object caching. If a global cache (shared permanent cache) was used, be aware of side-effects caused by runtime metadata changes.

For example, using two metadata profiles *A* and *B*. In profile A all fields of a class are showed, in profile B only the 'name filed' is showed. Thread 1 use profile A, thread 2 use profile B. It is obvious that a global shared cache will cause trouble.

4.5.9.5. Questions

Start OJB without a repository file?

It is possible to start OJB without any repository file. In this case you have to declare the `jdbc-connection-descriptor` and `class-descriptor` at runtime. See Connect to database at runtime? and Add new persistent objects (class-descriptors) at runtime? for more information.

Connect to database at runtime?

There are two possibilities to connect your database at runtime:

- load connection metadata by parsing additional repository files
- create the *JdbcConnectionDescriptor* at runtime

The first one is described in section load and merge connection metadata. For the second one a new instance of class `org.apache.ojb.broker.metadata.JdbcConnectionDescriptor` is needed. The prepared instance will be passed to class *ConnectionRepository*:

```
ConnectionRepository cr = MetadataManager.getInstance().connectionRepository();

JdbcConnectionDescriptor jcd = new JdbcConnectionDescriptor();
jcd.setJcdAlias("testConnection");
jcd.setUserName("sa");
jcd.setPassWord("sa");
jcd.setDbAlias("aAlias");
jcd.setDbms("aDatabase");
// .... the other required setter

// add new descriptor
cr.addDescriptor(jcd);

// Now it's possible to obtain a PB-instance
PBKey key = new PBKey("testConnection", "sa", "sa");
PersistenceBroker broker = PersistenceBrokerFactory.
createPersistenceBroker(key);
```

Please read this section from beginning for further information.

Add new persistent objects metadata (class-descriptor) at runtime?

There are two possibilities to add new *object metadata* at runtime:

- load object metadata by parsing additional repository files
- create new metadata objects at runtime

The first one is described in section load object metadata.

To create and add new metadata objects at runtime we create new `org.apache.ojb.broker.metadata.ClassDescriptor` instances at runtime and using the `MetadataManager` to add them to OJB:

```
DescriptorRepository dr = MetadataManager.getInstance().getRepository();
```

```
ClassDescriptor cld = new ClassDescriptor(dr);
cld.setClassOfObject(A.class);
//.... other setter

// add the fields of the class
FieldDescriptor fd = new FieldDescriptor(cld, 1);
fd.setPersistentField(A.class, "someAField");
cld.addFieldDescriptor(fd);

// now we add the the class descriptor
dr.setClassDescriptor(cld);
```

Please read this section from beginning for further information.

4.5.10. Deployment

4.5.10.1. Introduction

This section enumerates all things needed to deploy OJB in standalone or servlet based applications and j2ee-container.

4.5.10.2. Things needed for deploying OJB

1. The OJB binary jar archive

You need a `db-ojb-<version>.jar` file containing the compiled OJB library.
This jar files contains all OJB code neccessary in production level environments. It does not contain any test code. It also does not contain any configuration data. You'll find this file in the lib directory of the binary distribution. If you are working with the source distribution you can assemble the binary jar archive By calling

```
ant jar
```

This ant task generates the binary jar to the dist directory.

2. Configuration data

OJB needs two kinds of configuration data:

1. Configuration of the OJB runtime environment. This data is stored in a file named `OJB.properties` . Learn more about this file here.
2. Configuration of the MetaData layer. This data is stored in file named `repository.xml` (and several included files). Learn more about this file here.

Note:

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

3. Additional jar archives

OJB depends on several other jar archives. These jar files are shipped in the `db-ojb-<version>/lib` directory. These jar files are listed here.

Some of these jar files are only used during build-time and need not to be be deployed in runtime environments.
Apart from wasting disk space they do no harm. If you don't care you just take all jars from `db-ojb-<version>/lib`.
If you do care, here is the list of jars you can omit during runtime:

- `ant.jar`
- `antlr.debug.jar`
- `antlr_compiletime.jar`

- `junit.jar`
- `optional.jar`
- `xalan.jar`
- `ejb.jar`
- `servlet.jar`
- `jakarta-regexp-xxx.jar`
- `torque-xxx.jar`
- `velocity-xxx.jar`

4. Don't forget the JDBC driver

The repository.xml defines JDBC Connections to your runtime databases. To use the declared JDBC drivers the respective jar archives must also be present in the classpath. Refer to the documentation of your databases.

In the following sections I will describe how to deploy these items for specific runtime environments.

4.5.10.3. Deployment in standalone applications

Deploying OJB for standalone applications is most simple. If you follow these four steps your application will be up in a few minutes.

1. Add `db-ojb-<version>.jar` to the classpath
2. place `OJB.properties` and `repository.xml` files on the classpath
3. Add the additional runtime jar archives to the classpath.
4. Add your JDBC drivers jar archive to the classpath.

4.5.10.4. Deployment in servlet based applications

Generally speaking the four steps described in the previous section have to be followed also in Servlet / JSP based environments.
The exact details may differ for your specific Servlet container, but the general concepts should be quite similar.

1. Deploy `db-ojb-<version>.jar` with your servlet applications WAR file.
   The WAR format specifies that application specific jars are to be placed in a directory `WEB-INF/lib`. Place `db-ojb-<version>.jar` to this directory.
2. Deploy `OJB.properties` and `repository.xml` with your servlet applications WAR file.
   The WAR format specifies that Servlet classes are to be placed in a directory `WEB-INF/classes`. The OJB configuration files have to be in this directory.
3. Add the additional runtime jar archives to `WEB-INF/lib` too.
4. Add your JDBC drivers jar archive to `WEB-INF/lib`.

By executing `ant war` you can generate a sample servlet application assembled to a valid WAR file. The resulting `ojb-servlet.war` file is written to the dist directory. You can deploy this WAR file to your servlet engine or unzip it to have a look at its directory structure.
you can also use the target `war` as a starting point for your own deployment scripts.

4.5.10.5. Deployment in EJB based applications

The above mentioned guidelines concerning jar files and placing of the OJB.properties and the repository.xml are valid for EJB environments as well. But apart from these basic steps you'll have to perform some additional configurations to integrate OJB into a managed environment.

The instructions to make OJB running within your application server should be similar for all server. So the following instructions for JBoss should be useful for all user. E.g. most `OJB.properties` file settings are the same for all application server.

There are some topics you should examine very carefully:

- **Connection handling:** Lookup DataSource from your AppServer, only these connections will be enlisted in running transactions
- **Caching:** Do you need distributed caching?
- **Locking:** Do you need distributed locking (when using odmg-api)?

Configure OJB for managed environments considering as JBoss example

The following steps describe how to configure OJB for managed environments and deploy on a ejb conform Application Server (JBoss) on the basis of the shipped ejb-examples. In managed environments OJB needs some specific properties.

1. Adapt OJB.properties file

If the PB-api is the only persistence API being used (no ODMG nor JDO) and it is **only** being used in a managed environment, it is strongly recommended to use a special PersistenceBrokerFactory class, which enables PersistenceBroker instances to participate in the running JTA transaction (e.g. this makes PBStateListener proper work in managed environments and enables use of 'autoSync' property in ObjectCacheDefaultImpl):

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

Note:

Don't use this setting in conjunction with any other top-level api (e.g. ODMG-api).

Your `OJB.properties` file need the following additional settings to work within managed environments (apply to **all** used api):

```
...
ConnectionFactoryClass=
org.apache.ojb.broker.accesslayer.ConnectionFactoryManagedImpl

...
# set used application server TM access class
JTATransactionManagerClass=
org.apache.ojb.otm.transaction.factory.JBossTransactionManagerFactory
```

A specific *ConnectionFactory* implementation was used to by-pass all forbidden method calls in managed environments.

The *JTATransactionManagerClass* set the used implementation class for transaction manager lookup, necessary for for `javax.transaction.TransactionManager` lookup to participate in running *JTA transaction* via `javax.transaction.Synchronization` interface.

The ODMG-api needs some additional settings for use in managed environments (only needed when odmg-api was used):

```
...
# only needed for odmg-api
ImplementationClass=org.apache.ojb.odmg.ImplementationJTAImpl

...
# only needed for odmg-api
OJBTxManagerClass=org.apache.ojb.odmg.JTATxManager
```

The *ImplementationClass* specify the ODMG base class implementation. In managed environments a specific implementation is used, able to participate in *JTA transactions*.

The *OJBTxManagerClass* specify the used OJBTxManager implementation to manage the transaction synchronization in managed environments.

Note:

Currently OJB integrate in managed environments via `javax.transaction.Synchronization` interface. When the *JCA adapter* is finished (work in progress) integration will be more smooth.

2. Declare datasource in the repository (repository_database) file and do additional configuration

Do only use `DataSource` from the application server to connect to your database (Local used connections do not participate in JTA transaction).

Note:

We strongly recommend to use JBoss 3.2.2 or higher of the 3.x series of JBoss. With earlier versions of 3.x we got Statement/Connection resource problems when running some ejb stress tests. As workaround we introduce a jboss specific attribute *eager-release* for version before 3.2.2, but it seems that this attribute can cause side-effects. Again, this problem seems to be fixed in 3.2.2.

Define OJB to use a DataSource:

```
<!-- Datasource example -->
<jdbc-connection-descriptor
    jcd-alias="default"
    default-connection="true"
    platform="Sapdb"
    jdbc-level="2.0"
    jndi-datasource-name="java:DefaultDS"
    username="sa"
    password=""
    eager-release="false"
    batch-mode="false"
    useAutoCommit="0"
    ignoreAutoCommitExceptions="false"
>
    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
        <attribute attribute-name="timeout" attribute-value="900"/>
        <attribute attribute-name="autoSync" attribute-value="true"/>
     </object-cache>

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

</jdbc-connection-descriptor>
```

The attribute `useAutoCommit="0"` is mandatory in managed environments, because it's in most cases not allowed to change autoCommit state.

Note:

In managed environments you can't use the default sequence manager (SequenceManagerHighLowImpl) of OJB. For alternative sequence manager implemetation see here.

[2b. How to deploy ojb test hsqldb database to jboss]

If you use hsql database for testing you can easy setup the DB on jboss. After creating the database in OJB test directory with `ant prepare-testdb`, take the generated `.../target/test/OJB.script` file and rename it to `default.script`. Then replace the jboss default.script file in `.../jboss-3.x.y/server/default/db/hypersonic` with this file.

3. Include all OJB configuration files in classpath

Include the all needed OJB configuration files in your classpath:

- OJB.properties
- repository.dtd
- repository.xml
- repository_internal.xml

- repository_database.xml,
- repository_ejb.xml (if you want to run the ejb examples)

To deploy the ejb-examples beans we include all configuration files in a ejb jar file - more info about this see below.

The repository.xml for the ejb-example beans look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is a sample metadata repository for the ObJectBridge
System. Use this file as a template for building your own
mappings-->

<!-- 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>
```

4. Enclose all libraries OJB depend on

In most cases it is recommended to include all libraries OJB depend on in the application .ear/.sar or ejb .jar file to make OJB run and (re-)deployable. Here are the libraries needed to make the ojb sample session beans run on JBoss:

- 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
- [jakarta turbine jcs.jar from OJB /lib directory, only if ObjectCacheJCSImpl was used]

(This was tested with jboss 3.2.2)

5. Take care of caching

Very important thing is cache synchronization with the database. When using the ODMG-api or PB-api (with special PBF (see 1.) setting) it's possible to use all `ObjectCache` implementations as long as OJB doesn't run in a clustered mode. When the `ObjectCacheDefaultImpl` cache implementation was used it's recommended to enable the *autoSync* mode.
In clustered environments (OJB run on different AppServer nodes) you need a distributed ObjectCache or you should use a local/empty cache like

```
ObjectCacheClass=org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl
```

or

```
ObjectCacheClass=org.apache.ojb.broker.cache.ObjectCacheEmptyImpl
```

The cache is pluggable, so you can write your own ObjectCache implementation to accomplish your expectations.

More info you can find in clustering and ObjectCache topic.

6. Take care of locking

If the used api supports *Object Locking* (e.g. ODMG-api, PB-api does not), in clustered environments (OJB run on different AppServer nodes) a distributed lock management is mandatory.

7. Put all together

Now put all files together. We keep the examples as simple as possible, thus we deploy only a ejb .jar file. Below you can find a short instruction how to pack an ejb application .ear file including OJB.

Generate the ejb-examples described below or build your own ejb .jar file including all beans, ejb-jar.xml and appServer dependend files. Then add all OJB configuration files, the db-ojb jar file and all libraries OJB depends on into this ejb .jar file. The structure of the ejb .jar file should now look like this:

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

/all ejb classes

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

7b. Example: Deployable jar

For example the jar-file used to test the *ejb-examples* shipped with OJB, base on the *db-ojb-XY-beans.jar* file. This jar was created when the *ejb-examples* target was called.

The generated jar contains only the ejb-classes and the deployment-descriptor. We have to add additional jars (all libraries used by OJB) and files (all configuration files) to make it deployable. The deployable *db-ojb-XY-beans.jar* should look like this:

```
/OJB.properties
/repository.dtd
/repository.xml
/repository_database.xml
/repository_ejb.xml
/repository_internal.xml
/META-INF
.../Manifest.mf
.../ejb-jar.xml
.../jboss.xml

/org
.../apache (all ejb classes)

/db-ojb-1.X.jar

/antlr-XXX.jar
/commons-beanutils-XXX.jar
/commons-collections-XXX.jar
/commons-dbcp-XXX.jar
/commons-lanf-XXX.jar
/commons-logging-XXX.jar
/commons-pool-XXX.jar
/jakarta-regexp-XXX.jar
```

Please pay attention on the configuration settings to make OJB work in managed environments (especially the OJB.properties settings).

Note:

This example isn't a real world production example. Normally you will setup one or more enterprise archive files (.ear files) to bundle one or more complete J2EE (web) applications. More about how to build an *J2EE application* using OJB see here.

The described example should be re-deployable/hot-deployable in JBoss.
If you will get any problems, please let me know. All suggestions are welcome!

8. How to access OJB API?

In managed environments it is possible to access OJB in same way used in non-managed environments:

```
// PB-api
PersistenceBroker broker = PersistenceBrokerFactory.create...;

//ODMG-api
Implementation odmg = OJB.getInstance();
```

But it is also possible to bind OJB api access classes to JNDI and lookup the the api entry classes via JNDI.

9. OJB logging within JBoss

Jboss use log4j as standard logging api.
In summary, to use log4j logging with OJB within jBoss:
1) in OJB.properties set

```
LoggerClass=org.apache.ojb.broker.util.logging.Log4jLoggerImpl
```

There is no need for a separate log4j.properties file of OJB-specific log4j settings (in fact the OJB.properties setting LoggerConfigFile is ignored). Instead, the jBoss log4j configuration file must be used:

2) in JBOSS_HOME/server/default/conf/log4j.xml,
define appenders and add categories to add or filter logging of desired OJB packages, following the numerous examples in that file. For example,

```
<category name="org.apache.ojb">
    <priority value="DEBUG" />
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
</category>

<category name="org.apache.ojb.broker.metadata.RepositoryXmlHandler">
    <priority value="ERROR" />
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
</category>
```

Build example beans

Generate the sample session beans

The OJB source distribution was shipped with a bunch of sample session beans and client classes for testing. Please recognize that we don't say that these examples show "best practices" of using OJB within enterprise java beans - it's only one way to make it work.

The source code of the sample beans is stored in directory
`[db-ojb]/src/ejb/org/apache/ojb/ejb`
To generate the sample beans call

```
ant ejb-examples
```

This ant target copies the bean sources to `[db-ojb]/target/srcejb` generates all needed bean classes and deployment descriptor (by using xdoclet) to the same directory, compiles the sources and build an ejb .jar file called `[db-ojb]/dist/db-ojb-XXX-beans.jar`. Test clients for the generated beans included in the `[db-ojb]/dist/db-ojb-XXX-client.jar`.

To run xdoclet properly the following xdoclet jar files needed in `[db-ojb]/lib` directory (xdoclet version 1.2xx or higher):

```
xdoclet-xxx.jar
xdoclet-ejb-module-xxx.jar
xdoclet-jboss-module-xxx.jar
xdoclet-jmx-module-xxx.jar
xdoclet-web-module-xxx.jar
xdoclet-xjavadoc-module-xxx.jar
```

If you using a different application server than JBoss, you have to modifiy the *xdoclet* ant target in `[db-ojb]/build-ejb-examples.xml` to force xdoclet to generate the appServer specific files. See xdoclet documentation for further information.

How to run test clients for PB / ODMG - api

If the "extended ejb .jar" file was successfully deployed we need a test client to invoke the ejb-examples. As said above, the *ejb-examples* target generates a test client jar too. It's called `[db-ojb]/dist/db-ojb-XXX-client.jar` and contains junit based test clients for the PB-/ODMG-api.
The main test classes are:

- org.apache.ojb.ejb.AllODMGTests
- org.apache.ojb.ejb.AllPBTests

OJB provide an ant target to run the client side bean tests. Include all needed appServer libraries in `[db-ojb]/lib` (e.g. for JBoss jbossall-client.jar do the job, beside the "j2ee jars"). To run the PB-api test clients (access running JBoss server with default settings) call

```
ant ejb-examples-run -Dclient.class=org.apache.ojb.ejb.AllPBTests
```

To run the test clients on an arbitrary appServer pass the JNDI properties for naming context initalisation too, e.g.

- -Djava.naming.factory.initial="org.jnp.interfaces.NamingContextFactory"
- -Djava.naming.provider.url="jnp://localhost:1099"
- -Djava.naming.factory.url.pkgs="org.jboss.naming:org.jnp.interfaces"

Then the target call may looks like

```
ant ejb-examples-run -Dclient.class=org.apache.ojb.ejb.AllPBTests
 -Djava.naming.factory.initial="org.jnp.interfaces.NamingContextFactory"
  -Djava.naming.provider.url="jnp://localhost:1099"
   -Djava.naming.factory.url.pkgs="org.jboss.naming:org.jnp.interfaces"
```

Packing an .ear file

Here is an example of the .ear package structure. It is redeployable without having to restart JBoss.

The Package Structure

The package structure of the *.ear* file should look like:

```
/ejb.jar/
...EJBs
...META-INF/
......ejb-jar.xml
```

```
......jboss.xml
......MANIFEST.MF

/web-app.war/
...JSP
...WEB-INF/
......web.xml

/META-INF/
...application.xml
/ojb.jar
/[ojb required runtime jar]

/OJB.properties
/repository.dtd
/respository_internal.xml
/repository.xml
/repository_database1.xml
/repository_app1.xml
/repository_database2.xml
/repository_app2.xml
```

Make OJB API Resources available

There are two approaches to use OJB api in the ejb.jar file:

1. To create a Manifest.mf file with classpath attribute that include all the runtime jar required by OJB (Very important to include all required jar). The sample below works fine:

```
Class-Path: db-ojb-1.0.rc6.jar antlr-2.7.3.jar commons-beanutils.jar
commons-collections.jar commons-dbcp-1.1.jar commons-lang-2.0.jar
commons-logging.jar commons-pool-1.1.jar
jakarta-regexp-1.3.jar
```

Note:

If you to include the jar file under a directory of the ear file, says `/lib/db-ojb-1.0.rc6.jar` and etc. At the classpath attribute it will be something like: `Class-Path: ./lib/db-ojb-1.0.rc6.jar and etc` (The "." in front is important)

2. To add the required jar file as a "java" element in the application.xml file:

```
<module>
    <java>antlr-2.7.3.jar</java>
</module>
<module>
    <java>commons-beanutils.jar</java>
</module>
<module>
    <java>commons-collections.jar</java>
</module>
<module>
    <java>commons-dbcp-1.1.jar</java>
</module>
<module>
    <java>commons-lang-2.0.jar</java>
</module>
<module>
    <java>commons-logging.jar</java>
</module>
<module>
    <java>commons-pool-1.1.jar</java>
</module>
<module>
    <java>db-ojb-1.0.rc6.jar</java>
</module>
```

Note:

> To use this approach, all the library had to be in the root of the ear.

(This was tested on Jboss 3.2.3)

Make OJB accessible via JNDI

Current bean examples do directly use OJB main classes, but it's also possible to make OJB accessible via JNDI and use a JNDI-lookup to access OJB api's in your beans.
To make the OJB api's accessible via JNDI, you can bind them to JNDI. How to do this depends on the used environment. The main classes/method to bind are:

- Class `org.apache.ojb.broker.core.PersistenceBrokerFactoryFactory` for PB-api. Make method `PersistenceBrokerFactoryFactory.instance()` accessible.
- Class `org.apache.ojb.odmg.OJB` for ODMG-api. Make method `OJB.getInstance()` accessible.

JBoss

In JBoss you can use `mbean` classes. `org.apache.ojb.jboss.PBFactory` and `org.apache.ojb.jboss.ODMGFactory` are mbean implementations bind PB-api and ODMG-api main classes to JNDI.
Let JBoss know about the new mbeans, so declare them in a `jboss-service.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>

<server>
    <mbean code="org.apache.ojb.jboss.PBFactory"
        name="DefaultDomain:service=PBAPI,name=ojb/PBAPI">
        <depends>jboss.jca:service=RARDeployer</depends>
        <attribute name="JndiName">ojb/PBAPI</attribute>
    </mbean>

    <mbean code="org.apache.ojb.jboss.ODMGFactory"
    name="DefaultDomain:service=ODMG,name=ojb/defaultODMG">
        <depends>jboss.jca:service=RARDeployer</depends>
        <attribute name="JndiName">ojb/defaultODMG</attribute>
    </mbean>
</server>
```

Other Application Server

In other application server you can do similar steps to bind OJB main api classes to JNDI. For example in Weblogic you can use *startup class* implementation (see below).

Instructions for Weblogic

1. Add the OJB jar files and depedencies into the Weblogic classpath

2. As usual create the connection pool and the datasource.

3. Prepare the OJB.properties file. Should be similar to jboss. Expect the following entry:

```
...
# Weblogic Transaction Manager Factory
JTATransactionManagerClass=
org.apache.ojb.broker.transaction.tm.WeblogicTransactionManagerFactory
```

4. Modify the connection information in the repository.xml (specify the datasource name). SequenceManager implementation depends on the used DB, more info see here:

```
<jdbc-connection-descriptor
```

```
jcd-alias="default"
default-connection="true"
platform="Sapdb"
jdbc-level="2.0"
jndi-datasource-name="datasource_demodb"
eager-release="false"
batch-mode="false"
useAutoCommit="0"
ignoreAutoCommitExceptions="false"
>

<sequence-manager
className="org.apache.ojb.broker.util.sequence.SequenceManagerNextValImpl">
<attribute attribute-name="grabSize" attribute-value="20"/>
</sequence-manager>
</jdbc-connection-descriptor>
```

Note:

The following step is only neccessary if you want to bind OJB main api classes to JNDI.

[5.] Compile the following classes (see at the end of this section) and add them to the weblogic classpath. This allows to access the PB-api via JNDI lookup. Register via the weblogic console the startup class (see `OjbPbStartup` class below). The JNDI name and the OJB.properties file path can be specified as parameters in this startup class.

To use the ODMG-api you have to write a similar startup class. This shouldn't be too complicated. Take a look in `org.apache.ojb.jboss` package (dir `src/connector/main`). Here you could find the jboss mbeans. All you have to do is bound a similar class to JNDI in weblogic.
Implement `ODMGJ2EEFactory` Interface in your class bound this class to JNDI (in the ejb-examples the beans try to lookup the `Implementation` instance via `"java:/ojb/defaultODMG"`). Your ODMGFactory class should implement this method

```
    public Implementation getInstance()
    {
        return OJBJ2EE_2.getInstance();
    }
```

Write a session bean similar to those provided for the JBOSS samples. It is also possible to use the ejb-example beans (doing minor modifications when the JNDI lookup should be used).

Webolgic startup class
Write an OJB startup class to make OJB accessible via JNDI can look like (I couldn't test this sample class, so don't know if it will work ;-)):

```
package org.apache.ojb.weblogic;

import javax.naming.*;

import org.apache.ojb.broker.core.PersistenceBrokerFactoryFactory;
import org.apache.ojb.broker.core.PersistenceBrokerFactoryIF;

import weblogic.common.T3ServicesDef;
import weblogic.common.T3StartupDef;
import java.util.Hashtable;

/**
* This startup class created and binds an instance of a
* PersistenceBrokerFactoryIF into JNDI.
*/
public class OjbPbStartup
        implements T3StartupDef, OjbPbFactory, Serializable
{
    private String defaultPropsFile = "org/apache/ojb/weblogic/OJB.properties";
    public void setServices(T3ServicesDef services)
    {
    }

    public PersistenceBrokerFactoryIF getInstance()
    {
        return PersistenceBrokerFactoryFactory.instance();
    }

    public String startup(String name, Hashtable args)
            throws Exception
    {

        try
        {
            String jndiName = (String) args.get("jndiname");
            if(jndiName == null || jndiName.length() == 0)
                jndiName = OjbPbFactory.DEFAULT_JNDI_NAME;

            String propsFile = (String) args.get("propsfile");
            if(propsFile == null || propsFile.length() == 0)
            {
                System.setProperty("OJB.properties", defaultPropsFile);
            }
            else
            {
                System.setProperty("OJB.properties", propsFile);
            }

            InitialContext ctx = new InitialContext();
            bind(ctx, jndiName, this);

            // return a message for logging
            return "Bound OJB PersistenceBrokerFactoryIF to " + jndiName;
        }
        catch(Exception e)
        {
            e.printStackTrace();
            // return a message for logging
            return "Startup Class error: impossible to bind OJB PB factory";
        }
    }

    private void bind(Context ctx, String name, Object val)
            throws NamingException
    {
        Name n;
        for(n = ctx.getNameParser("").parse(name); n.size() > 1; n = n.getSuffix(1))
        {
            String ctxName = n.get(0);
            try
            {
                ctx = (Context) ctx.lookup(ctxName);
            }
            catch(NameNotFoundException namenotfoundexception)
            {
                ctx = ctx.createSubcontext(ctxName);
            }
        }
        ctx.bind(n.get(0), val);
    }
}
```

The used OjbPbFactory interface:

```
    package org.apache.ojb.weblogic;

    import org.apache.ojb.broker.core.PersistenceBrokerFactoryIF;
    public interface OjbPbFactory
    {
        public static String DEFAULT_JNDI_NAME = "PBFactory";
        public PersistenceBrokerFactoryIF getInstance();
    }
```

4.5.11. OJB - Connection Handling

4.5.11.1. Introduction

In this section the connection handling within OJB will be described. OJB use two classes which share the connection management:

- `org.apache.ojb.broker.accesslayer.ConnectionFactory`
- `org.apache.ojb.broker.accesslayer.ConnectionManagerIF`

4.5.11.2. ConnectionFactory

The `org.apache.ojb.broker.accesslayer.ConnectionFactory` interface implementation is a pluggable component (via the OJB.properties file - more about the OJB.properties file here) responsible for creation/lookup and release of connections.

```
public interface ConnectionFactory
{
    Connection lookupConnection(JdbcConnectionDescriptor jcd) throws LookupException;

    void releaseConnection(JdbcConnectionDescriptor jcd, Connection con);

    void releaseAllResources();
}
```

To enable a specific *ConnectionFactory* implementation class in *OJB.properties* file, set property *ConnectionFactoryClass*:

```
ConnectionFactoryClass=org.apache.ojb.broker.accesslayer.ConnectionFactoryPooledImpl
```

OJB was shipped with a bunch of different implementation classes which can be used in different situations, e.g. creation of connection instances is costly, so pooling of connection will increase performance.

To make it more easier to implement own *ConnectionFactory* classes an abstract base class called `org.apache.ojb.broker.accesslayer.ConnectionFactoryAbstractImpl` exists, most shipped implementation classes inherited from this class.

Note:

All shipped implementation with support for connection pooling only pool direct obtained connections, *DataSources* will **never** be pooled.

ConnectionFactoryPooledImpl

An *ConnectionFactory* implementation using commons-pool to pool the requested connections. On lookup call a connection was borrowed from pool and returned on the release call. This implementation was used as *default* setting in OJB.properties file.

This implementation allows a wide range off different settings, more info about the configuration properties can be found in metadata repository *connection-pool* section.

ConnectionFactoryNotPooledImpl

The name is programm, this implementation creates a new connection on each request and close it on release call. All *connection-pool* settings are ignored by this implementation.

ConnectionFactoryManagedImpl

This is a specific implementation for use in *managed environments* like J2EE conform application server. In managed environments it is **mandatory** to use *DataSource* provided by the application server.

All *connection-pool* settings are ignored by this implementation.

ConnectionFactoryDBCPImpl

An implementation using commons-dbcp to pool the connections.

This implementation allows a wide range off different settings, more info about the configuration properties can be found in metadata repository *connection-pool* section.

4.5.11.3. ConnectionManager

The `org.apache.ojb.broker.accesslayer.ConnectionManagerIF` interface implementation is a pluggable component (via the OJB.properties file - more about the OJB.properties file here) responsible for managing the connection usage lifecycle and connection status (commit/rollback of connections).

```
public interface ConnectionManagerIF
{
    JdbcConnectionDescriptor getConnectionDescriptor();

    Platform getSupportedPlatform();

    boolean isAlive(Connection conn);

    Connection getConnection() throws LookupException;

    boolean isInLocalTransaction();

    void localBegin();

    void localCommit();

    void localRollback();

    void releaseConnection();

    void setBatchMode(boolean mode);

    boolean isBatchMode();

    void executeBatch();

    void executeBatchIfNecessary();

    void clearBatch();
}
```

The *ConnectionManager* was used by the *PersistenceBroker* to handle connection usage lifecycle.

4.5.11.4. Questions and Answers

How does OJB handle connection pooling?

OJB does connection pooling per default, expect for datasources. Datasources never will be pooled.

Responsible for managing the connections in OJB are implementations of the `org.apache.ojb.broker.accesslayer.ConnectionFactory.java` interface. There are several implementations shipped with OJB called `org.apache.ojb.broker.accesslayer.ConnectionFactoryXXXImpl.java`. You can find among other things a none pooling implementation and a implementation using jakarta-DBCP api.

To manage the connection pooling define in your *jdbc-connection-descriptor* a *connection-pool* element. Here you can specify the properties for the used *ConnectionFactory* implementation. More common info see repository section or in repository.dtd.

Can I directly obtain a java.sql.Connection within OJB?

The PB-api enabled the possibility to obtain a connection from the current used `PersistenceBroker` instance:

```
PersistenceBroker broker = PersistenceBrokerFactory.
createPersistenceBroker(myKey);
broker.beginTransaction();
// do something

Connection con = broker.serviceConnectionManager().getConnection();
// perform your connction action and do more

broker.commitTransaction();
broker.close();
```

Note:

Do not commit the connection instance, this will be done by OJB when PersistenceBroker commit-/abortTransaction was called.

If no transaction was running, it is possible to release connection after use by hand:

```
pBroker.serviceConnectionManager().releaseConnection();
```

This call cleanup the used connection and pass the instance to release method of *ConnectionFactory* (this will e.g. return connection it to pool or close it). If you don't do any connection cleanup at the latest the connection will be released on PB.close() call.

Note:

Never do a `Connection.close()` call on the obtained connection instance by hand!!
This will be handled by the *ConnectionFactory*.

Users who interested in this section also interested in 'Is it possible to perform my own sql-queries in OJB?'.

4.5.12. The Object Cache

FIXME (arminw):

this document is not finished yet.

4.5.12.1. Introduction

OJB was shipped with several `ObjectCache` implementations. All implementations can be found in **`org.apache.ojb.broker.cache`** package. To classify the different implementations we differ *local cache* and *shared/global cache* (we use both terms synonymous) implementations.

- Local cache implementation mean that each instance use its own object map to manage cached objects.
- Shared/global cache implementations share one (in most cases static) map to manage cached objects.

A distributed object cache implementation supports caching of objects across different JVM.

4.5.12.2. Why a cache and how it works?

OJB provides a pluggable object cache provided by the `ObjectCache` interface.

```
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();
}
```

Each PersistenceBroker instance using its own `ObjectCache` instance. The `ObjectCache` instances are created by the `ObjectCacheFactory` class.

Each cache implementation holds Objects previously loaded or stored by the PersistenceBroker - dependend on the implementation.
Using a Cache has several advantages:

- It increases performance as it reduces database lookups or/and object materialization. If an object is looked up by Identity the associated PersistenceBroker instance, does not perform a `SELECT` against the database immediately but first looks up the cache if the requested object is already loaded. If the object is cached it is returned as the lookup result. If it is not cached a `SELECT` is performed.
  Other queries were performed against the database, but before an object from the ResultSet was materialized the object identity was looked up in cache. If not found the whole object was materialized.
- It allows to perform circular lookups (as by crossreferenced objects) that would result in non-terminating loops without such a cache.

4.5.12.3. How to change the used ObjectCache implementation

The `object-cache` element/property can be used to specify the ObjectCache implementation used by OJB. There are three levels of declaration:

in **OJB.properties** file, to declare the standard (default) ObjectCache implementation. The declared ObjectCache implementation was initialized with default properties, it's not possible to pass additional configuration properties on this level of declaration.

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

on **jdbc-connection-descriptor level** , to declare ObjectCache implementation on a per connection/user level. Additional configuration properties can be passed by using *attribute* element entries:

```
<jdbc-connection-descriptor ...>
...
<object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
<attribute attribute-name="timeout" attribute-value="900"/>
<attribute attribute-name="useAutoSync" attribute-value="true"/>
</object-cache>
...
</jdbc-connection-descriptor>
```

on **class-descriptor level** , to declare ObjectCache implementation on a per class level. Additional configuration properties can be passed by using *attribute* element entries:

```
<class-descriptor
class="org.apache.ojb.broker.util.sequence.HighLowSequence"
table="OJB_HL_SEQ"
>
<object-cache class="org.apache.ojb.broker.cache.ObjectCacheEmptyImpl">
</object-cache>
...
</class-descriptor>
```

> **Note:**
> The priority of the declared object-cache elements are:
> *per class > per jdbc-connection-descriptor > standard*

E.g. if you declare ObjectCache 'DefaultCache' as standard and set ObjectCache 'CacheA' in class-descriptor for class A and class B does not declare an object-cache element. Then OJB use 'CacheA' as ObjectCache for class A and 'DefaultCache' for class B.

4.5.12.4. Shipped cache implementations

ObjectCacheDefaultImpl

Per default OJB use a shared reference based `ObjectCache` implementation. It's a really fast cache but there are a few drawbacks. There is no transaction isolation, when thread one modify an object, thread two will see the modification when lookup the same object or use a reference of the same object. If you rollback/abort a transaction the corrupted objects will **not** be removed from the cache (when using PB-api, top-level api may support automatic cache synchronization). You have to do this using

```
    broker.removeFromCache(obj);

    // or (using Identity object)
    ObjectCache cache = broker.serviceObjectCache();
    cache.remove(oid);
```

by your own or enable the *useAutoSync* property (more info see below).

This implementation use `SoftReference` to wrap all cached objects. If the cached object was not longer referenced by your application but only by the cache, it can be reclaimed by the garbage collector.
As we don't know when the garbage collector reclaims the freed objects, it is possible to set a `timeout` property. So an cached object was only returned from cache if it was not garbage collected and was not timed out.

To enable this `ObjectCache` implementation

```
    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
        <attribute attribute-name="timeout" attribute-value="600"/>
    </object-cache>
```

Implementation configuration properties:

Property Key	Property Values

timeout	Lifetime of the cached objects in seconds. If expired, the cached object was discarded - default was 900 sec. When set to *-1* the lifetime of the cached object depends only on GC and do never get timed out.
autoSync	If set *true* all cached/looked up objects within a PB-transaction are traced. If the the PB-transaction was aborted all traced objects will be removed from cache. Default is *false*. **NOTE:** This does not prevent "dirty-reads" by concurrent threads (more info see above). It's not a smart solution for keeping cache in sync with DB but should do the job in most cases. E.g. if you lookup 1000 objects within a transaction and modify one object and then abort the transaction, 1000 objects will be passed to cache, 1000 objects will be traced and all 1000 objects will be removed from cache. If you read these objects without tx or in a former tx and then modify one object in a tx and abort the tx, only one object was traced/removed.

Recommendation:
If you take care of cache synchronization and be aware of dirty reads, this implementation is useful for read-only or less update centric classes.

ObjectCachePerBrokerImpl

This local cache implementation allows to have dedicated caches per PersistenceBroker instance. All calls are delegated to the cache associated with the current broker instance. When the broker

- does commit a transaction
- does abort/rollback a transaction
- was closed (returned to pool)

the cache was cleared. So no dirty reads will occur, because each thread use it's own PersistenceBroker instance. No corrupted objects will be found in cache, because the cache was cleared after use.

ObjectCacheJCSImpl

A shared `ObjectCache` implementation using a JCS region for each classname. More info see turbine-JCS.

ObjectCacheEmptyImpl

This is an 'empty' ObjectCache implementation. Useful when caching was not desired.

Note:
This implementaion does not support *circular References*. Be careful when using this implementaion with references (this may change in further versions).

ObjectCacheOSCacheImpl

A implementation using OpenSymphony's OSCache. More info see in Clustering HOWTO.

More implementations ...

Additional *ObjectCache* implementations can be found in *org.apache.ojb.broker.cache* package.

4.5.12.5. Distributed ObjectCache?

If OJB was used in a clustered enviroment it is mandatory to distribute all shared cached objects across different JVM. More information how to realize such a cache see here.

4.5.12.6. Implement your own cache

The OJB cache implementations are quite simple but do a good job for most scenarios. If you need a more sophisticated cache (e.g. with MRU memory management strategies) you'll write your own implementation of the interface `ojb.broker.cache.ObjectCache`.
Integration of your implementation in OJB is easy since the object cache is a pluggable component. All you have to do, is to declare it in the `OJB.properties` file by setting the `ObjectCacheClass` property.

> **Note:**
> Of course we interested in your solutions! If you have implemented something interesting, just contact us.

4.5.12.7. CacheFilter feature

What does cache filtering mean

TODO

Default CacheFilter implementations

TODO

Implement your own filter

TODO

4.5.12.8. Future prospects

TODO

4.5.13. Sequence Manager

4.5.13.1. The OJB Sequence Manager

All sequence manager implementations you will find under the `org.apache.ojb.broker.util.sequence` package using the following naming convention `SequenceManagerXXXImpl`.

Automatical assignment of unique values

As mentioned in mapping tutorial OJB provides a mechanism to assign unique values for primary key attributes. You just have to enable the `autoincrement` attribute in the respective *FieldDescriptor* of the XML repository file as follows:

```
<class-descriptor
  class="my.Article"
  table="ARTICLE"
>
    <field-descriptor
     name="articleId"
     column="ARTICLE_ID"
     jdbc-type="INTEGER"
     primarykey="true"
     autoincrement="true"
    />
    ....
```

```
</class-descriptor>
```

This definitions contains the following information:
The attribute `articleId` is mapped on the table's column `ARTICLE_ID`. The JDBC Type of this column is `INTEGER`. This is a primary key column. OJB shall automatically assign unique values to this attribute.

This mechanism works for columns of type INTEGER, CHAR and VARCHAR. This mechanism helps you to keep your business logic free from code that computes unique Ids for primary key attributes.

Force computation of unique values

By default OJB triggers the computation of unique ids during calls to PersistenceBroker.store(...). Sometimes it will be necessary to have the ids computed in advance. This can be done by simply obtaining the Identity of the respective object as follows:

```
Identity oid = new Identity(object, targetBroker);
```

FIXME (arminw):

Fix when new Identity creation concept is implemented.

How to change the sequence manager?

To enable a specific `SequenceManager` implementation declare an `sequence-manager` within the `jdbc-connection-descriptor` element in the repository file. If no `sequence-manager` was specified in the `jdbc-connection-descriptor`, OJB use a default sequence manager implementation (default was *SequenceManagerHighLowImpl*).

Further information you could find in the repository.dtd section sequence-manager element.

Example `jdbc-connection-descriptor` using `sequence-manager` tag:

```
<jdbc-connection-descriptor
        jcd-alias="farAway"
        platform="Hsqldb"
        jdbc-level="2.0"
        driver="org.hsqldb.jdbcDriver"
        protocol="jdbc"
        subprotocol="hsqldb"
        dbalias="../OJB_FarAway"
        username="sa"
        password=""
        batch-mode="false"
    >
        <connection-pool
            maxActive="5"
            whenExhaustedAction="0"
            validationQuery="select count(*) from OJB_HL_SEQ"
        />

        <sequence-manager className="org.apache.ojb.broker.util.
                             sequence.SequenceManagerHighLowImpl">
            <attribute attribute-name="grabSize" attribute-value="5"/>
            <attribute attribute-name="globalSequenceId"
                    attribute-value="false"/>
            <attribute attribute-name="globalSequenceStart"
                    attribute-value="10000"/>
        </sequence-manager>
</jdbc-connection-descriptor>
```

The mandatory `className` attribute needs the full-qualified class name of the desired sequence-manager implementation. If a implementation needs configuration properties you pass them using `attribute` tags with `attribute-name` represents the property name and `attribute-value` the property value. Each sequence manager implementation shows all properties on the according javadoc page.

SequenceManager implementations

Source code of all `SequenceManager` implementations can be found in `org.apache.ojb.broker.util.sequence` package.
If you still think something is missing you can just write your own sequence manager implementation.

High/Low sequence manager

Per default OJB internally uses a High/Low algorithm based sequence manager for the generation of unique ids, as described in Mapping Objects To Relational Databases.
This implementation is called `ojb.broker.util.sequence.SequenceManagerHighLowImpl` and is able to generate IDs unique to a given object and all extent objects declared in the objects class descriptor.
If you ask for an uid using an interface with several implementor classes, or a baseclass with several subclasses the returned uid have to be unique accross all tables representing objects of the extent in question (more see here).
It's also possible to use this implementation in a *global mode*, generate global unique id's.

```
<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>
```

With property `grabSize` you set the size of the assigned ids (default was 20).

If property `globalSequenceId` was set `true` you will get global unique ids over all persistent objects. Default was `false`.
The attribute `globalSequenceStart` define the start value of the global id generation (default was 10000).

This sequence manager implementation supports user defined *sequence-names* to manage the sequences. The attribute `autoNaming` define if sequence names should be build automatic if none found in `field-descriptor`.
If set 'true' OJB try to build a sequence name automatic if none found in field-descriptor and set this name as `sequence-name` in field-descriptor (see more). If set 'false' OJB throws an exception if none sequence name was found in field-descriptor (default was 'true').

Limitations:
- do **not** use in **managed environments** when connections were enlisted in running transactions, e.g. when using DataSources of an application server
- if set connection-pool attribute 'whenExhaustedAction' to 'block' (wait for connection if connection-pool is exhausted), under heavy load this sequence manager implementation can block application.
- superfluously to mention, do not use if other non-OJB applications insert objects too

In-Memory sequence manager

Another sequence manager implementation is a *In-Memory* version called `ojb.broker.util.sequence.SequenceManagerInMemoryImpl`.
Only the first time an uid was requested for a object, the manager query the database for the max value of the target column - all following request were performed in memory. This implementation ditto generate unique ids across all extents, using the same mechanism as the High/Low implementation.

```
<sequence-manager className="org.apache.ojb.broker.util.
                                sequence.SequenceManagerInMemoryImpl">
    <attribute attribute-name="autoNaming"
                                    attribute-value="true"/>
</sequence-manager>
```

For attribute `autoNaming` see

This sequence manager implementation supports user defined *sequence-names* to manage the sequences (see more) or if not set in `field-descriptor` it is done automatic.

This is the fastest standard sequence manager implementation, but has some Limitations:
- do not use in clustered environments
- superfluously to mention, do not use (or handle with care) if other non-OJB applications insert objects too

Database sequences based implementation

If your database support sequence key generation (e.g. Oracle, SAP DB, PostgreSQL) you could use the `SequenceManagerNextValImpl` implementation let your database generate the requested ids.

```
<sequence-manager className="org.apache.ojb.broker.util.
                                sequence.SequenceManagerNextValImpl">
    <attribute attribute-name="autoNaming"
                                    attribute-value="true"/>
</sequence-manager>
```

Attribute `autoNaming` default was 'true'. If set 'true' OJB try to build a sequence name automatic if none found in field-descriptor and set this generated name as `sequence-name` in field-descriptor.
If set 'false' OJB throws an exception if none sequence name was found in field-descriptor, ditto OJB does NOT try to create a database sequence entry when for given sequence name no database sequence could be found.

When using this sequence manager it is possible to define a **sequence-name** `field-descriptor` attribute in the repository file for each autoincrement/pk field. If you don't specify a sequence name, the sequence manager try to build a extent-aware sequence name on its own - except you set attribute `autoNaming` to 'false', then an exception will be thrown.
Keep in mind that in this case you are responsible to be aware of extents. Thus you have to use the same `sequence-name` attribute value for all extents, even if the extents were mapped to different database tables.
See usage of the `sequence-name` attribute:

```
<class-descriptor
      class="org.apache.ojb.broker.sequence.SMDatabaseSequence"
      table="SM_TAB_DATABASE_SEQUENCE"
    >
        <field-descriptor
         name="seqId"
         column="SEQ_ID"
         jdbc-type="INTEGER"
         primarykey="true"
         autoincrement="true"
         sequence-name="TEST_SEQUENCE"
        />

        ....
    </class-descriptor>
```

Limitations:
- none known

Database sequences based high/low implementation

Based on the sequence manager implementation described above, but use a high/low algorithm to avoid database access.

```
<sequence-manager className="org.apache.ojb.broker.util.
                        sequence.SequenceManagerSeqHiLoImpl">
<attribute attribute-name="grabSize" attribute-value="20"/>
<attribute attribute-name="autoNaming"
                                attribute-value="true"/>
</sequence-manager>
```

With the property `grabSize` you set the size of the assigned ids. For attribute `autoNaming` see.

This sequence manager implementation supports user defined *sequence-names* to manage the sequences (see more) or if not set in `field-descriptor` it is done automatic.

Limitations:
- superfluously to mention, do not use (or handle with care) if other non-OJB applications insert objects too

Oracle-style sequencing

(By Ryan Vanderwerf et al.) This solution will give those seeking an oracle-style sequence generator a final answer (Identity columns really suck). If you are using multiple application servers in your environment, and your database does not support read locking like Microsoft SQL Server, this is the only safe way to guarantee unique keys (HighLowSequenceManager WILL give out duplicate keys, and corrupt your data).
The `SequenceManagerStoredProcedureImpl` implementation enabled database sequence key generation in a *Oracle-style* for all databases (e.g. MSSQL, MySQL, DB2, ...).
First add a new table `OJB_NEXTVAL_SEQ` to your database.

```

CREATE TABLE OJB_NEXTVAL_SEQ
(
    SEQ_NAME    VARCHAR(150) NOT NULL,
    MAX_KEY     INTEGER,
    CONSTRAINT SYS_PK_OJB_NEXTVAL PRIMARY KEY(SEQ_NAME)
)
```

You will also need a stored procedure called `ojb_nextval_proc` that will take care of giving you a guaranteed unique sequence number.
Here is an example for the stored procedure you need to use sequencing for MSSQL server:

```

CREATE PROCEDURE OJB_NEXTVAL_PROC
@SEQ_NAME varchar(150)
AS
declare @MAX_KEY BIGINT
-- return an error if sequence does not exist
-- so we will know if someone truncates the table
set @MAX_KEY = 0

UPDATE OJB_NEXTVAL_SEQ
SET    @MAX_KEY = MAX_KEY = MAX_KEY + 1
WHERE  SEQ_NAME = @SEQ_NAME

if @MAX_KEY = 0
select 1/0
else
select @MAX_KEY
RETURN @MAX_KEY
```

You have to adapt this script if MSSQL was not used (We are interested in scripts for other databases). Last, enable this sequence manager implementation:

```

<sequence-manager className="org.apache.ojb.broker.util.
                       sequence.SequenceManagerStoredProcedureImpl">
    <attribute attribute-name="autoNaming"
                                    attribute-value="true"/>
</sequence-manager>
```

For attribute *autoNaming* see .

This sequence manager implementation supports user defined *sequence-names* to manage the sequences (see more) or if not set in `field-descriptor` it is done automatic.

Limitations:
- currently none known

Microsoft SQL Server 'uniqueidentifier' type (GUID) sequencing

For those users you are using SQL Server 7.0 and up, the uniqueidentifier was introduced, and allows for your rows Primary Keys to be GUID's that are guaranteed to be unique in time and space.

However, this type is different than the Identity field type, whereas there is no way to programmatically retrieve the inserted value. Most implementations when using the u.i. field type set a default value of "newid()". The SequenceManagerMSSQLGuidImpl class manages this process for you as if it was any normal generated sequence/identity field.

Assuming that your PK on your table is set to 'uniqueidentifier', your field-description would be the same as using any other SequenceManager:

```
<field-descriptor
         name="guid"
         column="document_file_guid"
         jdbc-type="VARCHAR"
         primarykey="true"
         autoincrement="true"
      />
```

Note that the jdbc-type is a VARCHAR, and thus the attribute (in this case 'guid') on your class should be a String (SQL Server does the conversion from the String representation to the binary representation when retrieved/set).

You also need to turn on the SequenceManager in your jdbc-connection-descriptor like this:

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

Limitations:
-This will only work with SQL Server 7.0 and higher as the uniqueidentifier type was not introduced until then.
This works well in situations where other applications might be updated the database as well, because it guarantees (well, as much as Microsoft can guarantee) that there will be no collisions between the Guids generated.

Identity based sequence manager

This sequence manager implementation supports database Identity columns (supported by MySQL, MsSQL, HSQL, ...). When using identity columns we have to do a trick to make the sequence manager work.
OJB identify each persistence capable object by a unique ojb-Identity object. These ojb-Identity objects were created using the sequence manager instance to get UID's. Often these ojb-Identity objects were created before the persistence capable object was written to database.
When using Identity columns it is not possible to retrieve the next valid UID before the object was written to database. As recently as the real object was written to database, you can ask the DB for the last generated UID. Thus in SequenceManagerNativeImpl we have to do a trick and use a 'temporary' UID till the object was written to database.
So, if it's possible try to avoid using Identity columns in your database model. If not use this sequence manager implementation to as a workaround for the Identity problem.

To enable this sequence manager implementation set in your `jdbc-connection-descriptor`:

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

To declare the identity column in the repository.xml file add `primarykey="true"`, `autoincrement="true"` and `access="readonly"` to the field-descriptor for your table's primary key identity column.

```
<field-descriptor
        name="identifier"
        column="NATIVE_ID"
        jdbc-type="BIGINT"
        primarykey="true"
        autoincrement="true"
        access="readonly"/>
```

Limitations:
- The Identity columns have to **start with value >= 1** and should never be negative.
- Use of Identity columns is **not extent aware** (This may change in further versions). More info here.

How to write my own sequence manager?

Very easy to do, just write a implementation class of the interface `org.apache.ojb.broker.util.sequence.SequenceManager`. OJB use a factory (`SequenceManagerFactory`) to obtain sequence manager instances.

This Factory 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>
```

That's it!

If your sequence manager implementation was derived from `org.apache.ojb.broker.util.sequence.AbstractSequenceManager` it's easy to 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 get the properties in your implementation class.

Note:
Of course we interested in your solutions! If you have implemented something interesting, just contact us.

Questions

When using sequence-name attribute in field-descriptor?

Most `SequenceManager` implementations based on sequence names. If you want retain control of sequencing use your own `sequence-name` attribute in the `field-descriptor`. In that case you are reponsible to use the same name across extents (see more info about extents and polymorphism). Per default the sequence manager build its own *extent aware* sequence name with an simple algorithm (see `org.apache.ojb.broker.util.sequence.SequenceManagerHelper#buildSequenceName`) if necessary. In most cases this should be sufficient. If you have a very complex data model and you will do many metadata changes in the repository file in future, then it could be better to explicit use `sequence-names` in the `field-descriptor`. See more avoid pitfals.

What to hell does extent aware mean?

Say we have a abstract base class `Animal` and two classes `Dog` and `Cat` which extend `Animal`. For each non-abstract class we create a separate database table.
We will be able to do a query like *give me all animals*. Thus the uid's of `Dog` and `Cat` objects must be unique across the tables of both classes or else you may not get a vaild query result.
The reason for this behaviour is the `org.apache.ojb.broker.Identity` class implementation (this may change in further versions).

How could I prevent auto-build of the sequence-name?

All shipped `SequenceManager` implementations which using sequence names for UID generation, support by default auto-build (autoNaming) of the sequence name if none was found in the `field-descriptor`.
To prevent this, all relevant SM implementations support a `autoNaming` property - set via `attribute` element. If set `false` OJB doesn't try to build sequence names automatic.

```
<sequence-manager className="XYZ">
...
    <attribute attribute-name="autoNaming" attribute-value="true"/>
...
</sequence-manager>
```

Sequence manager handling using multiple databases

If you use multiple databases you have to declare a sequence manager in each `jdbc-connection-descriptor`. If you don't specify a sequence manager OJB use the default one (currently `ojb.broker.util.sequence.SequenceManagerHighLowImpl`).

One sequence manager with multiple databases?

OJB was intended to use a sequence manager per database. But it shouldn't be complicated to realize a global sequence manager solution by writing your own `SequenceManager` implementation.

Can I get direct access to the sequence manager?

That's no problem:

```
                    PersistenceBroker broker =
                    PersistenceBrokerFactory.createPersistenceBroker(myPBKey);
                    SequenceManager sm = broker.serviceSequenceManager();
                    ...
                    broker.close();
```

If you use `autoincrement=true` in your `field-descriptor`, there is no reason to obtain UID directly from the sequence manager or to handle UID in your object model.

> **Note:**
> Don't use SequenceManagerFactory#getSequenceManager(PersistenceBroker broker), this method returns a new sequence manager instance for the given broker instance and not the current used SM instance of the given PersistenceBroker instance]

Any known pitfalls?

- When enable a sequence manager implementation based on *sequence-name* attributes and if the name was not set as an attribute in the `field-descriptor` (see), an simple algorithm was used to build the sequence name.
  The algorithm try to get the top-level class of the field's enclosing class, if no top-level class was found, the table name of the field's enclosing class was used. If a top-level class was found, the first found extent class table name was used as sequence name.
  When using base classes/interfaces with extent classes based on different database tables and the `extent-class` entries in repository often change, the algorithm could be corrupted, because the first found extent class's table name could be change.
  To avoid this, remove the implementation internal sequence name entry (e.g. OJB_HL_SEQ table entry when using the Hi/Lo implementation, or remove the database sequence entry when using the 'Nextval' implementation) in that case, or use custom sequence name attributes in the field descriptor.

4.5.14. OJB logging configuration

4.5.14.1. Logging in OJB

For generating log messages, OJB provides its own, simplistic logging component PoorMansLoggerImpl, but is also able to use the two most common Java logging libraries, commons-logging (which is actually a wrapper around several logging components) and Log4j. In addition, it is also possible to define your own logging implementation.

Per default, OJB uses its own PoorMansLoggerImpl which does not require configuration and prints to `stdout`.

4.5.14.2. Logging configuration within OJB

How and when OJB determines what kind of logging to use

Logging is the first component of OJB that is initialized. If you access any component of OJB, logging will be initialized first before that component is doing anything else. Therefore, you'll have to provide for the configuration of logging before you access OJB in your program (this is mostly relevant if you plan to initialize OJB at runtime as is described below). Please note that logging configuration is independent of the configuration of other parts of OJB, namely the runtime (via OJB.properties) and the database/repository (via repository.xml).

These are the individual steps OJB performs in order to initialize the logging component:

1. First, OJB checks whether the system property `org.apache.ojb.broker.util.logging.Logger.class` is set. If specified, this property gives the fully qualified class name of the logger class (a class implementing the Logger interface). Along with this property, another property is then read which may specify a properties file for this logger class, `org.apache.ojb.broker.util.logging.Logger.configFile`.
2. If this property is not set, then OJB tries to read the file `OJB-logging.properties`. The name and path of this file can be changed by setting the runtime property of the same name. See below for the contents of this file.
3. For backwards compatibility, OJB next tries to read the logging settings from the file OJB.properties which is the normal runtime configuration file of OJB. Again, the name and path of this file can be changed by setting the runtime property of the same name. This file may contain the same entries as the `OJB-logging.properties` file.
4. If the the `OJB.properties` file does not contain logging settings, next it is checked whether the commons-logging log property `org.apache.commons.logging.Log` or the commons-logging log factory system property `org.apache.commons.logging.LogFactory` is set. If that's the case, OJB will use commons-logging for its logging purposes.
5. Next, OJB checks for the presence of the Log4j properties file `log4j.properties`. If it is found, the OJB uses Log4j directley (without commons-logging).
6. Finally, OJB tries to find the commons-logging properties file `commons-logging.properties` which when found directs OJB to use commons-logging for its logging.
7. If none of the above is true, or if the specified logger class could not be found or initialized, then OJB defaults to its `PoorMansLoggerImpl` logger which simply logs to `stdout`.

The only OJB component whose logging is not initialized this way, is the boot logger which is used by logging component itself and a few other core components. It will (for obvious reasons) always use PoorMansLoggerImpl and therefore log to `stdout`. You can define the log level of the boot logger via the `OJB.bootLogLevel` system property. Per default, **WARN** is used.

Configuration of logging for the individual components

Regardless of the logging implementation that is used by OJB, the configuration is generally similar. The individual logging implementations mainly differ in the syntax and in the configuration of the format of the output and of the output target (where to log to). See below for specific details and examples.
In general, you specify a default log level and for every component (usually a class) that should log differently, the amount and level of detail that is logged about that component. These are the levels:

DEBUG
Messages that express what OJB is currently doing. This is the most detailed debugging level
INFO
Informational messages
WARN
Warnings that may denote potentional problems (this is the default level)
ERROR
As the name says, this level is for errors which means that some action could not be completed successfully
FATAL
Fatal errors which usually prevent an application from continuing

The levels **DEBUG** and **INFO** usually result in a lot of log messages which will reduce the performance of the application. Therefore these levels should only be used when necessary.

There are two special loggers to be aware of. The **boot logger** is the logger used by the logging component itself as well as a few other core components. It will therefore always use the PoorMansLoggerImpl logging implementation. You can configure its logging level via the `OJB.bootLogLevel` system property.
The **default logger** is denoted in the `OJB-logging.properties` file by the keyword `DEFAULT` instead of the class name. It is used by components that don't require their own logging configuration (usually because they are rather small components).

4.5.14.3. Logging configuration via configuration files

OJB-logging.properties

This file usually specifies which logging implementation to use using the `org.apache.ojb.broker.util.logging.Logger.class` property, and which properties file this logger has (if any) using the `org.apache.ojb.broker.util.logging.Logger.configFile` property. You should also use this file to specify log levels for OJB's components if you're not using Log4j or commons-logging (which have their own configuration files).

A typical `OJB-logging.properties` file looks like this:

```
# Which logger to use
org.apache.ojb.broker.util.logging.Logger.class=org.apache.ojb.broker.util.logging.PoorMansLoggerImpl

# Configuration file of the logger
#org.apache.ojb.broker.util.logging.Logger.configFile=

# Global default log level used for all logging entities if not specified
ROOT.LogLevel=ERROR

# The log level of the default logger
```

```
DEFAULT.LogLevel=WARN

# Logger for PersistenceBrokerImpl class
org.apache.ojb.broker.core.PersistenceBrokerImpl.LogLevel=WARN

# Logger for RepositoryXmlHandler, useful for debugging parsing of repository.xml!
org.apache.ojb.broker.metadata.RepositoryXmlHandler.LogLevel=WARN
```

commons-logging.properties

This file is used by commons-logging. For details on its structure see here.

An example `commons-logging.properties` file would be:

```
# Use Log4j
org.apache.commons.logging.Log=org.apache.commons.logging.impl.Log4JLogger

# Configuration file of the log
log4j.configuration=log4j.properties
```

> **Note:**
> Since commons-logging provides the same function as the logging component of OJB, it will likely be used as OJB's logging component in the near future.

log4j.properties

The commons-logging configuration file. Details can be found here.

A sample log4j configuration is:

```
# Root logging level is WARN, and we're using two logging targets
log4j.rootCategory=WARN, A1, A2

# A1 is set to be ConsoleAppender sending its output to System.out
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-5r %-5p [%t] %c{2} - %m%n

# Appender A2 writes to the file "org.apache.ojb.log".
log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.File=org.apache.ojb.log

# Truncate the log file if it aleady exists.
log4j.appender.A2.Append=false

# A2 uses the PatternLayout.
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%-5r %-5p [%t] %c{2} - %m%n

# Special logging directives for individual components
log4j.logger.org.apache.ojb.broker.metadata.RepositoryXmlHandler=DEBUG
log4j.logger.org.apache.ojb.broker.accesslayer.ConnectionManager=INFO
log4j.logger.org.apache.ojb.odmg=INFO
```

Where to put the configuration files

OJB and the different logging implementations usually look up their configuration files in the classpath. So for instance, OJB searches for the `OJB-logging.properties` file directly in any of the entries of the classpath, directories and jar files. If the classpath contains in that order `some-library.jar`, `db-ojb.jar`, and `.`, then it will first search in the two jars (which themselves contain a directory structure in which OJB will search only in the root), and lastly in the current directory (which only happens if `.` is part of the classpath) but not in sub directories of it.

For applications, this classpath can easily be set either as an environment variable `CLASSPATH` or by using the commandline switch `-classpath` when invoking the java executable.

For web applications however, the server will define the classpath. There are specific folders in the webapp structure that are always part of the webapp's classpath. The one that is normally used to store configuration files, is the `classes` folder:

```
[folder containing webapps]\
    mywebapp\
        WEB-INF\
            lib\
            classes\   <-- Put your configuration files here
```

4.5.14.4. Logging configuration at runtime

Sometimes you want to configure OJB completely at runtime (within your program). How to do that for logging depends on the used logging implementation, but you can usually configure them via system properties. The only thing to keep in mind is that logging in OJB is initialized as soon as you use one of its components, so you'll have to define the properties prior to using any OJB parts.

With system properties (which are accessible via `System.getProperty()` from within a Java program) you can always define the following OJB logging settings:

`org.apache.ojb.broker.util.logging.Logger.class`	Which logger OJB shall use
`org.apache.ojb.broker.util.logging.Logger.configFile`	The config file of the logger
`OJB-logging.properties`	The path to the logging properties file, default is `OJB-logging.properties`
`OJB.properties`	The path to the OJB properties file (which may contain logging settings), default is `OJB.properties`
`org.apache.commons.logging.Log`	Use commons-logging with the specified log implementation
`org.apache.commons.logging.LogFactory`	Use commons-logging with the specified log factory
`log4j.configuration`	When using Log4j directly or via commons-logging, this is the Log4j configuration file (default is `log4j.properties`)

In addition, all Log4j properties (e.g. `log4j.rootCategory`) can be specified as system properties.

4.5.14.5. Defining your own logger

It is rather easy to use your own logger. All you need to do is to provide a class that implements the interface Logger. Besides the actual log methods (`debug, info, warn, error, fatal`) this interface defines a method `void configure(Configuration)` which is used to initialize the logger with the logging properties (as contained in `OJB-logging.properties`).

Note:

Because commons-logging performs a similar function to the OJB logging component, it is likely that it will be used as such in the near future. Therefore you're encouraged to also implement the Log interface which is nearly the same as the Logger interface.

4.5.15. The ODMG Lock Manager

4.5.15.1. What it does

The OJB ODMG implementation provides object level transactions as specified by the ODMG. This includes features like registering objects to transactions, persistence by reachability (a toplevel object is registered to a transaction, and also all its associated objects become registered implicitely) and as a very important aspect: object level locking.
Lockmanagement is needed to synchronize concurrent access to objects from multiple transactions (possibly from remote machines).
An example: There are two transactions `tx1` and `tx2` running on different physical machines. `Tx1` acquired a write lock on an object `obj` with the globally unique identity `oid`. Now also `tx2` tries to get a write lock on an object `obj'` (it's not the *same* object as it resides in a different VM!) with the *same* identity `oid` (an OJB Identity is unique accross VMs !). The OJB LockManager is responsible for detecting this conflict and doesn't allow `tx2` to obtain a write lock to prevent data inconsistency.

The ODMG Api allows transactions to lock an object `obj` as follows:

`org.odmg.Transaction.lock(Object obj, int lockMode)`,

where lockMode defines the locking mode:

```
/** Read lock mode.*/
public static final int  READ = 1;

/** Upgrade lock mode. */
public static final int  UPGRADE = 2;

/** Write lock mode. */
public static final int  WRITE = 4;
```

A sample session could look as follows:

```
// get odmg facade instance
Implementation odmg = OJB.getInstance();

//open database
Database db = odmg.newDatabase();
db.open(&quot;repository.xml&quot;, Database.OPEN_READ_WRITE);

// start a transaction
Transaction tx = odmg.newTransaction();
tx.begin();

MyClass myObject = ... ;

// lock object for read access
tx.lock(myObject, Transaction.READ);

// now perform read access on myObject ...

// lock object for write access
tx.lock(myObject, Transaction.UPGRADE);

// now perform write access on myObject ...

// finally commit transaction to make changes to myObject persistent
tx.commit();
```

The ODMG specification does not say if locks must be acquired explicitely by client applications or may be acquired implicitely. OJB provides implicit locking for the application programmers convenience: On commit of a transaction all read-locked objects are checked for modifications. If a modification is detected, a write lock is acquired for the respective object. If automatic acquisition of read- or write-lock failes, the transaction is aborted.

On locking an object to a transaction, OJB automatically locks all associated objects (as part of the *persistence by reachability feature*) with the same locking level. If application use large object nets which are shared among several transactions acquisition of write-locks may be very difficult. Thus OJB can be configured to aquire only read-locks for associated objects.

You can change this behaviour by modifying the file [OJB.properties]() and changing the entry `LockAssociations=WRITE` to `LockAssociations=READ`.

The ODMG specification does not prescribe transaction isolationlevels or locking strategies to be used. Thus there are no API calls for setting isolationlevels. OJB provides four different isolationlevels that can be configured for each persistent class in the XML repository.
The isolationlevel of a class can be configured with the following attribute to a ClassDescriptor:

```
<ClassDescriptor isolation="read-uncomitted" ...>
    ...
</ClassDescriptor>
```

The four supported values are:

- read-uncommitted
- read-committed
- repeatable-read
- serializable

The semantics of these isolationlevels is defined below.

4.5.15.2. How it works

To provide Lockmanagement in a massively distributed environment as the OJB client/server architecture, OJB implements a LockManager that allows transaction coordination accross multiple threads, multiple VMs and even multiple physical machines running OJB ODMG transactions. The Default Implementation of this LockManager uses a database table to store locks. To make locks persistent allows to make them visible to all connected ODMG clients. Thus there is no need for an additional LockManager server that is accessed from all ODMG clients.

The LockManager interface provides the following API:

```
public interface LockManager
{
    /**
     * aquires a readlock for transaction tx on object obj.
     * Returns true if successful, else false.
     */
    public abstract boolean readLock(TransactionImpl tx, Object obj);

    /**
     * aquires a writelock for transaction tx on object obj.
     * Returns true if successful, else false.
     */
    public abstract boolean writeLock(TransactionImpl tx, Object obj);

    /**
     * upgrades readlock for transaction tx on object obj to a writelock.
     * If no readlock existed a writelock is acquired anyway.
     * Returns true if successful, else false.
     */
    public abstract boolean upgradeLock(TransactionImpl tx, Object obj);

    /**
     * releases a lock for transaction tx on object obj.
     * Returns true if successful, else false.
     */
    public abstract boolean releaseLock(TransactionImpl tx, Object obj);

    /**
     * checks if there is a readlock for transaction tx on object obj.
     * Returns true if so, else false.
     */
    public abstract boolean checkRead(TransactionImpl tx, Object obj);
    /**
     * checks if there is a writelock for transaction tx on object obj.
     * Returns true if so, else false.
     */
    public abstract boolean checkWrite(TransactionImpl tx, Object obj);
}
```

The lockmanager must allow and disallow locking according to the Transaction Isolationlevel specified for `obj.getClass()`in the XML RepositoryFile. It does so by applying a corresponding LockStrategy. LockStrategies are selected by the LockStrategyFactory:

```
private static LockStrategy readUncommitedStrategy =
                                new ReadUncommittedStrategy();
private static LockStrategy readCommitedStrategy =
                                new ReadCommittedStrategy();
private static LockStrategy readRepeatableStrategy =
                                new RepeatableReadStrategy();
private static LockStrategy serializableStrategy =
                                new SerializableStrategy();

/**
 * Obtains a LockStrategy for Object obj. The Strategy to be used is
 * selected by evaluating the ClassDescriptor of obj.getClass().
 *
 * @return LockStrategy
 */
public static LockStrategy getStrategyFor(Object obj)
{
    int isolationLevel = getIsolationLevel(obj.getClass());
    switch (isolationLevel)
    {
        case IsolationLevels.RW_READ_UNCOMMITTED:
            return readUncommitedStrategy;
        case IsolationLevels.RW_READ_COMMITTED:
            return readCommitedStrategy;
        case IsolationLevels.RW_REPEATABLE_READ:
            return readRepeatableStrategy;
        case IsolationLevels.RW_SERIALIZABLE:
            return serializableStrategy;
        default:
            return readUncommitedStrategy;
    }
}
```

The four LockStrategies implement different behaviour according to the underlying isolationlevel. The semantics of the strategies are defined by the following table:

Nr.	Name of TestCase	Transactions		Transaction-Isolationlevel			
		Tx1	**Tx2**	**ReadUncommitted**	**ReadCommitted**	**RepeatableRead**	**Serializable**
1	SingleReadlock	R		True	True	True	True
18	ReadThenRead	R		True	True	True	True
		R					
2	UpgradeReadlock	R		True	True	True	True
		U					
3	ReadThenWrite	R		True	True	True	True

		W					
4	SingleWriteLock	W		True	True	True	True
5	WriteThenRead	W		True	True	True	True
		R					
6	MultipleReadLock	R	R	True	True	True	False
7	UpgradeWithExistingReader	R	U	True	True	False	False
8	WriteWithExistingReader	R	W	True	True	False	False
9	UpgradeWithMultipleReaders	R	R	True	True	False	False
			U				
10	WriteWithMultipleReaders	R	R	True	True	False	False
			W				
11	UpgradeWithMultipleReadersOn1	R	R	True	True	False	False
		W					
12	WriteWithMultipleReadersOn1	R	R	True	True	False	False
		W					
13	ReadWithExistingWriter	W	R	True	False	False	False
14	MultipleWriteLocks	W	W	False	False	False	False
15	ReleaseReadLock	R		True	True	True	True
		Rel	W				
16	ReleaseUpgradeLock	U		True	True	True	True
		Rel	W				
17	ReleaseWriteLock	W		True	True	True	True
		Rel	W				
	Acquire ReadLock	R					
	Acquire WriteLock	W					
	Upgrade	U					

	Lock						
	Release Lock	Rel					

The table is to be read as follows. The acquisition of a single read lock on a given object (case 1) is allowed (returns True) for all isolationlevels. To upgrade a single read lock (case 2) is also allowed for all isolationlevels. If there is already a write lock on a given object for tx1, it is not allowed (returns False) to obtain a write lock from tx2 for all isolationlevels (case 14).

The isolationlevels can be simply characterized as follows:

Uncommitted Reads
Obtaining two concurrent write locks on a given object is not allowed (case 14). Obtaining read locks is allowed even if another transaction is writing to that object (case 13). (Thats why this level is also called *dirty reads*)

Committed Reads
Obtaining two concurrent write locks on a given object is not allowed. Obtaining read locks is allowed only if there is no write lock on the given object (case 13).

Repeatable Reads
As commited reads, but obtaining a write lock on an object that has been locked for reading by another transaction is not allowed (case 7).

Serializable transactions
As Repeatable Reads, but it is even not allowed to have multiple read locks on a given object (case 6).

The proper behaviour of the LockStrategies is checked by JUnit TestCases that implement test methods for each of the 17 cases specified in the above table. (See code for classes `test.ojb.odmg.LockTestXXX`)

4.5.15.3. Locking in distributed environment

############ TODO ############

4.5.15.4. Implement you own lock manager

The LockManager default implementation uses a database table to make locks globally visible to all connected clients. This is a foolproof solution as it does not require a separate LockManager server. But it involves a lot of additional database traffic, as each lock check, acquisition or release results in database operations.
This may not be viable in some environments. Thus OJB allows to plug in user defined LockManagers implementing the `ojb.odmg.locking.LockManager` interface. OJB obtains its LockManager from the factory `ojb.odmg.locking.LockManagerFactory`. This Factory can be configured to generate instances of a specific implementation by changing the following entry in the configuration file OJB Properties file:

```
LockManagerClass=ojb.odmg.locking.LockManagerDefaultImpl
```

to:

```
LockManagerClass=acme.com.MyOwnLockManagerImpl.
```

Note:
Of course I'm interested in your solutions! If you have implemented something interesting, just contact me.

4.5.16. XDoclet OJB module documentation

4.5.16.1. Acquiring and building

The XDoclet OJB module is part of OJB source. As such, the source of the module is part of the OJB source tree and can be found in directory src/xdoclet. Likewise, binary versions of the module and the required libraries (xjavadoc, xdoclet) are to be found in the lib folder.

In order to build the XDoclet OJB module from source, you'll need a source distribution of XDoclet version 1.2, either a source distribution from the sourceforge download site or a CVS checkout/drop. See the XDoclet website at http://xdoclet.sourceforge.net/install.html for details.

Building with a XDoclet source distribution

Unpack the source distribution of XDoclet which is contained in a file `xdoclet-src-<version>.<archive-format>` somewhere. If you unpacked it side-by-side of OJB, you'll get a directory layout similar to:

```
\xdoclet-1.2
    \config
    \core
    \lib
    ...
\db-ojb
    \bin
    \contrib
    ...
```

The XDoclet OJB module is then build using the `build-xdoclet-module.xml` ant script:

```
ant -Dxdoclet.src.dir=../xdoclet-1.2 -f build-xdoclet-module.xml
```

The build process will take some time, and after successful compilation the three jars `xjavadoc-<version>.jar`, `xdoclet-<version>.jar`, and `xdoclet-ojb-module-<version>.jar` are copied to the library directory of OJB.

Building with a XDoclet CVS checkout

When checking out from CVS (the `xdoclet-all` target), you'll get a directory like:

```
\xdoclet-all
    \xdoclet
        \config
        \core
        ...
    \xdocletgui
    \xjavadoc
\db-ojb
    \bin
    \contrib
    ...
```

Building is XDoclet OJB module is performed by calling:

```
ant -Dxdoclet.src.dir=../xdoclet-all/xdoclet -f build-xdoclet-module.xml
```

Since this is the default structure assumed by the build script, this can be shortend to:

```
ant -f build-xdoclet-module.xml
```

Other build options

The build script for the XDoclet OJB module uses the OJB build properties so the following line added to the `build.properties` file in the OJB root directory allows to omit the `-Dxdoclet.src.dir=<xdoclet src dir>` commandline option:

```
xdoclet.src.dir=<xdoclet src dir>
```

4.5.16.2. Usage

Using the XDoclet OJB module is rather easy. Put the module jar along with the xdoclet and xjavadc jars in a place where ant will find it, and then invoke it in your build file like:

```
<target name="repository-files">
    <taskdef name="ojbdoclet"
             classname="xdoclet.modules.ojb.OjbDocletTask"
             classpathref="build-classpath">
    <ojbdoclet destdir="./build">
        <fileset dir="./src"/>
        <ojbrepository destinationFile="repository_user.xml"/>
        <torqueschema databaseName="test" destinationFile="project-schema.xml"/>
    </ojbdoclet>
</target>
```

The XDoclet OJB module has two sub tasks, `ojbrepository` and `torqueschema`, which generate the OJB repository part containing the user descriptors and the torque table schema, respectively. Please note that the XDoclet OJB module (like all xdoclet tasks) expects the directory structure of its input java source files to match their package structure. In this regard it is similar to the `javac` ant task.
Due to a bug in XDoclet, you should not call the `ojbdoclet` task more than once in the same `taskdef` scope. So, each `ojbdoclet` call should be in its own target with a leading `taskdef`.

The main `ojbdoclet` task has two attributes:

destdir
The destination directory where generated files will be placed.
checks : none | basic | strict (default)
The amount of the checks performed. Per default, `strict` checks are performed which means that for instance classes specified in an attribute (e.g. `collection-class`, `row-reader` etc.) are loaded from the classpath and checked. So in this mode it is necessary to have OJB as well as the processed classes on the classpath (using the `classpathref` attribute of the `taskdef` ant task above). If this is for some reason not possible, then use `basic` which performs most of the checks but does not load classes from the classpath. `none` does not perform any checks so use it with care and only if really necessary (in this case it would be helpful if you would post the problem to the ojb-user mailing list).

The `ojbrepository` subtask has the following attributes:

destinationFile
Specifies the output file. The default is `repository_user.xml`.
verbose : true | false (default)
Whether the task should output some information about its progress.

The `torqueschema` subtask has these attributes:

databaseName
This attribute gives the name of the database for torque (required).
destinationFile
The output file, default is `project-schema.xml`.
dtdUrl
Allows to specify the url of the torque dtd. This is necessary e.g. for XML parsers that have problems with the default dtd url (http://jakarta.apache.org/turbine/dtd/database.dtd), or when using a newer version of torque.
generateForeignkeys : true (default) | false
Whether foreignkey tags are generated in the torque database schema.
verbose : true | false (default)
Whether the task outputs some progress information.

The `classpathref` attribute in the taskdef can be used to define the classpath for xdoclet (containing the xdoclet and ojb module jars), e.g. via:

```
<path id="build-classpath">
    <fileset dir="lib">
        <include name="**/*.jar"/>
    </fileset>
</path>
```

Using the generated torque schema is a bit more tricky. The easiest way is to use the `build-torque.xml` script which is part of OJB. Include the lib subdirectory of the OJB distribution which also includes torque (e.g. in `build-classpath` as shown above). You will also want to use your OJB settings (if you're using the ojb-blank project, then only `build.properties`), so include them at the beginning of the build script if they are not already there:

```
<property file="build.properties"/>
```

Now you can create the database with ant calls similar to these:

```
<target name="init-db" depends="repository-files">
    <!-- Torque's build file -->
    <property name="torque.buildFile"
              value="build-torque.xml"/>

    <!-- The name of the database which we're taking from the profile -->
    <property name="torque.project"
              value="${databaseName}"/>

    <!-- Where the schemas (your project and, if required, ojb's internal tables) are -->
    <property name="torque.schema.dir"
              value="src/schema"/>

    <!-- Build directory of Torque -->
    <property name="torque.output.dir"
              value="build"/>

    <!-- Torque will put the generated sql here -->
    <property name="torque.sql.dir"
              value="${torque.output.dir}"/>

    <!-- Torque shall use the classpath (to find the jdbc driver etc.) -->
    <property name="torque.useClasspath"
              value="true"/>

    <!-- Which jdbc driver to use (again from the profile) -->
    <property name="torque.database.driver"
              value="${jdbcRuntimeDriver}"/>

    <!-- The url used to build the database; note that this may be different
         from the url to access the database (e.g. for MySQL) -->
    <property name="torque.database.buildUrl"
              value="${urlProtocol}:${urlSubprotocol}:${urlDbalias}"/>


    <!-- Now we're generating the database sql -->
    <ant dir="."
         antfile="${torque.buildFile}"
         target="sql">
    </ant>
    <!-- Next we create the database -->
    <ant dir="."
         antfile="${torque.buildFile}"
         target="create-db">
    </ant>
    <!-- And the tables -->
    <ant dir="."
         antfile="${torque.buildFile}"
         target="insert-sql">
    </ant>
</target>
```

As you can see, the major problem of using Torque is to correctly setup Torque's build properties.

One important thing to note here is that the latter two calls modify the database and in the process remove any existing data, so use them with care. Similar to the above targets, you can use the additional targets `datadump` for storing the data currently in the database in an XML file, and `datasql` for inserting the data from an XML file into the database.
Also, these steps are only valid for the torque that is delivered with OJB, but probably not for newer or older versions.

4.5.16.3. Tag reference

Interfaces and Classes
ojb.class
ojb.extent-class
ojb.modify-inherited
ojb.object-cache
ojb.index
ojb.delete-procedure
ojb.insert-procedure
ojb.update-procedure
ojb.constant-argument
ojb.runtime-argument
Fields and Bean properties
ojb.field
References
ojb.reference
Collections
ojb.collection
Nested objects
ojb.nested
ojb.modify-nested

4.5.16.4. Interfaces and Classes

ojb.class

The **ojb.class** tag marks interfaces and classes that shall be present in the repository descriptor. This includes types that are used as reference targets or as collection elements, but for instance not abstract base classes not used elsewhere.

Attributes:

attributes
Optionally contains attributes of the class as a comma-separated list of name-value pairs.
determine-extents : true (default) | false
When set to `true`, then the XDoclet OJB module will automatically determine all extents (ojb-relevant sub types) of this type. If set to `false`, then extents need to be specified via the ojb.extent-class class tag (see below).
documentation
Optionally contains documentation on the class.
generate-table-info : true (default) | false
This attribute controls whether the type has data and should therefore get a torque table descriptor. When set to `false`, no field, reference or collection descriptors are generated.

include-inherited : true (default) | false
Determines whether base type fields/references/collections with the appropriate tags (ojb.field, ojb.reference, ojb.collection) will be included in the descriptor and table definition of this class. Note that all base type fields/references/collections with an appropriate tag are included regardless of whether the base types have the **ojb.class** tag or not.

table
The name of the table used for this type. Is only used when table info is generated. If not specified, then the short name of the type is used.

The following `class-descriptor` attributes are also supported in the **ojb.class** tag and will be written directly to the generated class descriptor (see the repository.dtd for their meaning):

- **accept-locks**
- **factory-class**
- **factory-method**
- **initialization-method**
- **isolation-level**
- **proxy**
- **proxy-prefetching-limit**
- **refresh**
- **row-reader**

Example: (from the unit tests)

```
/**
 * @ojb.class generate-table-info="false"
 */
public abstract class AbstractArticle implements InterfaceArticle, java.io.Serializable
...

/**
 * @ojb.class table="Artikel"
 *            proxy="dynamic"
 *            include-inherited="true"
 *            documentation="This is important documentation on the Article class."
 *            attributes="color=blue,size=big"
 */
public class Article extends AbstractArticle implements InterfaceArticle, java.io.Serializable
...
```

The `AbstractArticle` class will have an class descriptor in the repository file, but no field, reference or collection descriptors. The `Article` class however will not only have descriptors for its own fields/references/collections but also for those inherited from `AbstractArticle`. Also, its table definition in the torque file will be called "Artikel", not "Article". The resulting class descriptors look like:

```
<class-descriptor
    class="org.apache.ojb.broker.AbstractArticle"
>
    <extent-class class-ref="org.apache.ojb.broker.Article"/>
</class-descriptor>

<class-descriptor
    class="org.apache.ojb.broker.Article"
    proxy="dynamic"
    table="Artikel"
>
    <documentation>This is important documentation on the Article class.</documentation>
    ...
    <attribute attribute-name="color" attribute-value="blue"/>
    <attribute attribute-name="size" attribute-value="big"/>
</class-descriptor>
...
```

ojb.extent-class

Use the **ojb.extent-class** to explicitly specify extents (direct persistent sub types) of the current type. The **class-ref** attribute contains the fully qualified name of the class. However, these tags are only evaluated if the **determine-extents** attribute of the ojb.class tag is set to `false`.

Attributes:

class-ref
The fully qualified name of the sub-class (required).

Example:

```
/**
 * @ojb.class determine-extents="false"
 *            generate-table-info="false"
 * @ojb.extent-class class-ref="org.apache.ojb.broker.CdArticle"
 */
public abstract class AbstractCdArticle extends Article implements java.io.Serializable
...
```

which results in:

```
<class-descriptor
    class="org.apache.ojb.broker.AbstractCdArticle"
>
    <extent-class class-ref="org.apache.ojb.broker.CdArticle"/>
</class-descriptor>
```

ojb.modify-inherited

Allows to modify attributes of inherited fields/references/collections (normally, all attributes are used without modifications) for this and all sub types. One special case is the specification of an empty value which leads to a reset of the attribute value. As a result the default value is used for this attribute.

Attributes: All of ojb.field, ojb.reference, and ojb.collection (with the exception of **indirection-table** and **remote-foreignkey**), and also:

ignore : true | false (default)
Specifies that this feature will be ignored in this type (but only in the current type, not in subtypes).

name
The name of the field/reference/collection to modify (required).

Example:

```
/**
 * @ojb.class table="Artikel"
 * @ojb.modify-inherited name="productGroup"
 *                       proxy="true"
 *                       auto-update="object"
 */
public class ArticleWithReferenceProxy extends Article
```

produces the class descriptor

```
<class-descriptor
    class="org.apache.ojb.broker.ArticleWithReferenceProxy"
    table="Artikel"
>
    ...
    <reference-descriptor
        name="productGroup"
        class-ref="org.apache.ojb.broker.ProductGroup"
        proxy="true"
        auto-update="object"
    >
        <documentation>this is the reference to an articles productgroup</documentation>
        <attribute attribute-name="color" attribute-value="red"/>
        <attribute attribute-name="size" attribute-value="tiny"/>
        <foreignkey field-ref="productGroupId"/>
    </reference-descriptor>
</class-descriptor>
```

ojb.object-cache

The **ojb.object-cache** tag allows to specify the ObjectCache implementation that OJB uses for objects of this class (instead of the one defined in the jdbc connection descriptor or in the `ojb.properties` file). Classes specified with this tag have to implement the `org.apache.ojb.broker.cache.ObjectCache` interface. Note that object cache specifications are not inherited.

Attributes:

attributes
Optionally contains attributes of the object cache as a comma-separated list of name-value pairs.
class
The fully qualified name of the object cache class (required).
documentation
Optionally contains documentation on the object cache specification.

Example:

```
/**
 * @ojb.class
 * @ojb.object-cache class="org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl"
 *                   documentation="Some important documentation"
 */
public class SomeClass implements Serializable
{
    ...
}
```

and the class descriptor

```
<class-descriptor
    class="SomeClass"
    table="SomeClass"
>
    <object-cache class="org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl">
        <documentation>Some important documentation</documentation>
    </object-cache>
    ...
</class-descriptor>
```

ojb.index

The **ojb.index** tag is used to define possibly unique indices for the class. An index consists of at least one field of the class (either locally defined or inherited, anonymous or explicit). There is an default index (without a name) that is made up by all fields that have the **indexed** attribute set to `true`. All other indices have to be defined via the **ojb.index** tag. In contrast to the **indexed** attribute, indices defined via the **ojb.index** tag are not inherited.

Attributes:

documentation
Optionally contains documentation on the index.

fields
The fields that make up the index separated by commas (required).
name
The name of the index (required). If there are multiple indices with the same name, then only the first one is used (all others are ignored).
unique : true | false (default)
Whether the index is unique or not.

Example:

```
/**
 * @ojb.class table="SITE"
 * @ojb.index name="NAME_UNIQUE"
 *            unique="true"
 *            fields="name"
 */
public class Site implements Serializable
{
    /**
     * @ojb.field indexed="true"
     */
    private Integer nr;
    /**
     * @ojb.field column="NAME"
     *            length="100"
     */
    private String name;
    ...
}
```

the class descriptor

```
<class-descriptor
    class="org.apache.ojb.odmg.shared.Site"
    table="SITE"
>
    <field-descriptor
        name="nr"
        column="nr"
        jdbc-type="INTEGER"
        indexed="true"
    >
    </field-descriptor>
    <field-descriptor
        name="name"
        column="NAME"
        jdbc-type="VARCHAR"
        length="100"
    >
    </field-descriptor>
    ...
    <index-descriptor
        name="NAME_UNIQUE"
        unique="true"
    >
        <index-column name="NAME"/>
    </index-descriptor>
</class-descriptor>
```

and the torque table schema

```
<table name="SITE">
    <column name="nr"
            javaName="nr"
            type="INTEGER"
    />
    <column name="NAME"
            javaName="name"
            type="VARCHAR"
            size="100"
    />
    ...
    <index>
        <index-column name="nr"/>
    </index>
    <unique name="NAME_UNIQUE">
        <unique-column name="NAME"/>
    </unique>
</table>
```

ojb.delete-procedure

Declares a database procedure that is used for deleting persistent objects.

Attributes:

arguments
A comma-separated list of the names of constant or runtime arguments specified in the same class.
attributes
Optionally contains attributes of the procedure as a comma-separated list of name-value pairs.
documentation
Optionally contains documentation on the procedure.
include-pk-only : true | false (default)
Whether all fields of the class that make up the primary key, shall be passed to the procedure. If set to `true` then the **arguments** value is ignored.
name
The name of the procedure (required).
return-field-ref
Identifies a field of the class that will receive the return value of the procedure. Use only if the procedure has a return value.

Example:

```
/**
 * @ojb.class
 * @ojb.delete-procedure name="DELETE_PROC"
 *                       arguments="arg1,arg2"
 *                       return-field-ref="attr2"
 *                       documentation="Some important documentation"
 * @ojb.constant-argument name="arg1"
 *                        value="0"
 * @ojb.runtime-argument name="arg2"
 *                       field-ref="attr1"
 */
public class SomeClass
{
    /** @ojb.field */
    private Integer attr1;
    /** @ojb.field */
    private String attr2;
    ...
}
```

leads to the class descriptor

```
<class-descriptor
    class="SomeClass"
    table="SomeClass"
>
```

```
    <field-descriptor
        name="attr1"
        column="attr1"
        jdbc-type="INTEGER"
    >
    </field-descriptor>
    <field-descriptor
        name="attr2"
        column="attr2"
        jdbc-type="VARCHAR"
        length="254"
    >
    </field-descriptor>
    ...
    <delete-procedure
        name="DELETE_PROC"
        return-field-ref="attr2"
    >
        <documentation>Some important documentation</documentation>
        <constant-argument
            value="0"
        >
        </constant-argument>
        <runtime-argument
            field-ref="attr2"
        >
        </runtime-argument>
    </delete-procedure>
</class-descriptor>
```

ojb.insert-procedure

Identifies the database procedure that shall be used for inserting objects into the database.

Attributes:

arguments
Comma-separated list of names of constant or runtime arguments that are specified in the same class.
attributes
Contains optional attributes of the procedure in a comma-separated list of name-value pairs.
documentation
Contains optional documentation on the procedure.
include-all-fields : true | false (default)
Specifies whether all persistent fields of the class shall be passed to the procedure. If so, then the **arguments** value is ignored.
name
The name of the procedure (required).
return-field-ref
The persistent field that receives the return value of the procedure (should only be used if the procedure returns a value).

For an example see constant argument.

ojb.update-procedure

The database procedure that will be used for updating persistent objects in the database.

Attributes:

arguments
A comma-separated list of names of constant or runtime arguments in the same class.
attributes

The optional attributes of the procedure in a comma-separated list of name-value pairs.

documentation

Optional documentation on the procedure.

include-all-fields : true | false (default)

Whether all persistent fields of the class shall be passed to the procedure in which case the **arguments** value is ignored.

name

Name of the database procedure (required).

return-field-ref

A persistent field that will receive the return value of the procedure (only to be used if the procedure returns a value).

For an example see runtime argument.

ojb.constant-argument

A constant argument for a database procedure. These arguments are referenced by the procedure tags in the **arguments** attribute via their names.

Attributes:

attributes

Optionally contains attributes of the argument.

documentation

Optionally contains documentation on the argument.

value

The constant value.

name

The identifier of the argument to be used the **arguments** attribute of a procedure tag (required).

Example:

```
/**
 * @ojb.class
 * @ojb.insert-procedure name="INSERT_PROC"
 *                       arguments="arg"
 * @ojb.constant-argument name="arg"
 *                        value="Some value"
 *                        attributes="name=value"
 */
public class SomeClass
{
    ...
}
```

will result in the class descriptor

```
<class-descriptor
    class="SomeClass"
    table="SomeClass"
>
    ...
    <insert-procedure
        name="INSERT_PROC"
    >
        <constant-argument
            value="Some value"
        >
            <attribute attribute-name="name" attribute-value="value"/>
        </constant-argument>
    </insert-procedure>
```

```
</class-descriptor>
```

ojb.runtime-argument

An argument for a database procedure that is backed by a persistent field. Similar to constant arguments the name is important for referencing by the procedure tags in the **arguments** attribute.

Attributes:

attributes
Contains optionally attributes of the argument.
documentation
Optionally contains documentation on the argument.
field-ref
The persistent field that delivers the value. If unspecified, then in the procedure call `null` will be used.
name
Identifier of the argument for using it in the **arguments** attribute of a procedure tag (required).
return
If the field receives a value (?).

Example:

```
/**
 * @ojb.class
 * @ojb.update-procedure name="UPDATE_PROC"
 *                       arguments="arg"
 * @ojb.runtime-argument name="arg"
 *                       field-ref="attr"
 *                       documentation="Some documentation"
 */
public class SomeClass
{
    /** @ojb.field */
    private Integer attr;
    ...
}
```

will result in the class descriptor

```
<class-descriptor
    class="SomeClass"
    table="SomeClass"
>
    <field-descriptor
        name="attr"
        column="attr"
        jdbc-type="INTEGER"
    >
    </field-descriptor>
    ...
    <update-procedure
        name="UPDATE_PROC"
    >
        <runtime-argument
            value="attr"
        >
            <documentation>Some documentation</documentation>
        </runtime-argument>
    </update-procedure>
</class-descriptor>
```

4.5.16.5. Fields and Bean properties

ojb.field

Fields or accessor methods (i.e. get/is and set methods) for properties are marked with the **ojb.field** tag to denote a persistent field. When a method is marked, then the corresponding bean property is used for naming purposes (e.g. "value" for a method `getValue()`). The XDoclet OJB module ensures that a field is not present more than once, therefore it is safe to mark both fields and their accessors. However, in that case the three **ojb.field** tags are required to have the same attributes.

Due to a bug in XDoclet, you currently cannot process `final` or `transient` fields.

Marked fields are used for descriptor generation in the same type (if it has an ojb.class tag) and all sub types with the ojb.class tag having the **include-inherited** attribute set to `true`.
It is also possible to use the **ojb.field** tag at the class level (i.e. in the JavaDoc comment of the class). In this case, the tag is used to define an *anonymous* field, e.g. a "field" that has no counterpart in the class but exists in the database. For anonymous fields, both the **name** and the **jdbc-type** attributes are required, and the **access** attribute is ignored (it defaults to the value `anonymous`). Beside these differences, anonymous fields are handled like other fields, (e.g. they result in field-descriptor entries in the repository descriptor, and in columns in the table schema, and they are inherited and can be modified via the ojb.modify-inherited tag.

The XDoclet OJB module orders the fields in the repository descriptor and table schema according to the following rules:

1. Fields (anonymous and non-anonymous) from base types/nested objects and from the current file that have an id, sorted by the id value. If fields have the same id, then they are sorted following the rules for fields without an id.
2. Fields (anonymous and non-anonymous) from base types/nested objects and from the current file that have no id, in the order they appear starting with the farthest base type. Per class, the anonymous fields come first, followed by the non-anonymous fields.

Attributes:

access : readonly | readwrite (default)
Specifies the accessibility of the field. `readonly` marks fields that are not to modified. `readwrite` marks fields that may be read and written to. Anonymous fields do not have to be marked (i.e. `anonymous` value) as the position of the **ojb.field** tag in the class JavaDoc comment suffices.

attributes
Optionally contains attributes of the field as a comma-separated list of name-value pairs.

autoincrement : none (default) | ojb | database
Defines whether this field gets its value automatically. If `ojb` is specified, then OJB fills the value using a sequence manager. If the value is `database`, then the column is also defined as `autoIncrement` in the torque schema (i.e. the database fills the field), and in the repository descriptor, the field is marked as `access='readonly'` (if it isn't an anonymous field). The `database` value is intended to be used with the `org.apache.ojb.broker.util.sequence.SequenceManagerNativeImpl` sequence manager. For details, see the Sequence Manager documentation.
The default value is none which means that the field is not automatically filled.

column
The name of the database column for this field. If not given, then the name of the attribute is used.

conversion
The name of the class to be used for conversion between the java type of the field (e.g. `java.lang.Boolean` or `java.util.Date`) and the java type for the JDBC type (e.g. `java.lang.Integer` or `java.sql.Date`). Conversion classes must implement the `org.apache.ojb.broker.accesslayer.conversions.FieldConversion` interface. If no explicit JDBC type is defined and the java type has no defined conversion (see below), then per default the `org.apache.ojb.broker.accesslayer.conversions.Object2ByteArrFieldConversion` conversion class is used.
Default conversion is also used for the following java types when no jdbc type is given (default type is used instead), and no conversion is specified:

Java type	Default conversion
`org.apache.ojb.broker.util.GUID`	`org.apache.ojb.broker.accesslayer.conversions.GUID2StringFieldCo`

documentation
Optionally contains documentation on the field.

id
An integer specifying the position in the repository descriptor and table schema. For the placement rules see above.

jdbc-type : BIT | TINYINT | SMALLINT | INTEGER | BIGINT | DOUBLE | FLOAT | REAL | NUMERIC | DECIMAL | CHAR | VARCHAR | LONGVARCHAR | DATE | TIME | TIMESTAMP | BINARY | VARBINARY | LONGVARBINARY | CLOB | BLOB | STRUCT | ARRAY | REF | BOOLEAN | DATALINK
The JDBC type for the column. The XDoclet OJB module will automatically determine a jdbc type for the field if none is specified. This means that for anonymous fields, the **jdbc-type** attribute is required. The automatic mapping performed by the XDoclet OJB module from java type to jdbc type is as follows:

Java type	JDBC type
`boolean`	`BIT`
`byte`	`TINYINT`
`short`	`SMALLINT`
`int`	`INTEGER`
`long`	`BIGINT`
`char`	`CHAR`
`float`	`REAL`
`double`	`FLOAT`
`java.lang.Boolean`	`BIT`
`java.lang.Byte`	`TINYINT`
`java.lang.Short`	`SMALLINT`
`java.lang.Integer`	`INTEGER`
`java.lang.Long`	`BIGINT`
`java.lang.Character`	`CHAR`
`java.lang.Float`	`REAL`
`java.lang.Double`	`FLOAT`
`java.lang.String`	`VARCHAR`
`java.util.Date`	`DATE`
`java.sql.Date`	`DATE`
`java.sql.Time`	`TIME`
`java.sql.Timestamp`	`TIMESTAMP`
`java.sql.Blob`	`BLOB`
`java.sql.Clob`	`CLOB`

`java.sql.Ref`	`REF`
`java.sql.Struct`	`STRUCT`
`java.math.BigDecimal`	`DECIMAL`
`org.apache.ojb.broker.util.GUID`	`VARCHAR`

For any other type (including array types) the default mapping is to `LONGVARBINARY` using the `Object2ByteArrFieldConversion` conversion (see **conversion** attribute above).

length
The length of the column which might be required by the jdbc type in some databases. This is the reason that for some jdbc types, the XDoclet OJB module imposes default lengths if no length is specified:

Jdbc type	Default length
`CHAR`	`1`
`VARCHAR`	`254`

name
The name of the field. This attribute is required for anonymous fields, otherwise it is ignored.

precision

scale
The precision and scale of the column if required by the jdbc type. They are usually used in combination with the `DECIMAL` and `NUMERIC` types, and then specifiy the number of digits before (**precision**) and after (**scale**) the comma (excluding the plus/minus sign). Due to restrictions in some databases (e.g. MySQL), the XDoclet OJB module imposes default values for some types if none are specified:

Jdbc type	Default values for precision, scale
`DECIMAL`	`20,0` (this corresponds to the range of `long` where the longest number is -9223372036854775808).
`NUMERIC`	`20,0`

For other types, if only the precision is specified, the scale defaults to 0. If only scale is specified, precision defaults to 1.

Other attributes supported in the **ojb.field** tag that have the same meaning as in the repository descriptor (and partly in the torque table schema) are:

- **default-fetch**
- **indexed**
- **locking**
- **nullable**
- **primarykey**
- **sequence-name**
- **update-lock**

Examples:

```
/**
 * @ojb.field column="Auslaufartikel"
 *            jdbc-type="INTEGER"
 *            conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"
 *            id="10"
 *            attributes="color=green,size=small"
 */
protected boolean isSelloutArticle;
```

will result in the following field descriptor:

```
<field-descriptor
    name="isSelloutArticle"
    column="Auslaufartikel"
    jdbc-type="INTEGER"
    conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"
>
    <attribute attribute-name="color" attribute-value="green"/>
    <attribute attribute-name="size" attribute-value="small"/>
</field-descriptor>
```

The column descriptor looks like:

```
<table name="Artikel">
    ...
    <column name="Auslaufartikel"
            javaName="isSelloutArticle"
            type="INTEGER"
    />
    ...
</table>
```

An anonymous field is declared like this:

```
/**
 * @ojb.class table="TABLE_F"
 *            include-inherited="false"
 * @ojb.field name="eID"
 *            column="E_ID"
 *            jdbc-type="INTEGER"
 * @ojb.reference class-ref="org.apache.ojb.broker.E"
 *                auto-retrieve="true"
 *                auto-update="object"
 *                auto-delete="object"
 *                foreignkey="eID"
 */
public class F extends E implements Serializable
...
```

In this case an anonymous field is declared and also used as the foreignkey of an anonymous reference. The corresponding class descriptor looks like:

```
<class-descriptor
    class="org.apache.ojb.broker.F"
    table="TABLE_F"
>
    <field-descriptor
        name="eID"
        column="E_ID"
        jdbc-type="INTEGER"
        access="anonymous"
    >
    </field-descriptor>
    ...
    <reference-descriptor
        name="super"
        class-ref="org.apache.ojb.broker.E"
        auto-retrieve="true"
        auto-update="object"
        auto-delete="object"
    >
        <foreignkey field-ref="eID"/>
    </reference-descriptor>
</class-descriptor>
```

Here the anonymous field and reference (which implicitly refers to `super`) are used to establish the super-subtype relationship between `E` and `F` on the database level. For details on this see the advanced technique section.

4.5.16.6. References

ojb.reference

Similar to fields, references (java fields or accessor methods) are marked with the **ojb.reference** tag. We have a reference when the type of the java field is itself a persistent class (has an ojb.class tag) and therefore the java field represents an association. This means that the referenced type of an association (or the one specified by the **class-ref** attribute, see below) is required to be present in the repository descriptor (it has the ojb.class tag).
Foreign keys of references are also declared in the torque table schema (see example below).
OJB currently requires that the referenced type has at least one field used to implement the reference, usually some id of an integer type.
A reference can be stated in the JavaDoc comment of the class (anonymous reference), but in this case it silently refer to `super` (see the example of ojb.field) which can be used to establish an inheritance relationship. Note that anonymous references are not inherited (in contrast to anonymous fields and normal references).

Attributes:

> **attributes**
> Optionally contains attributes of the reference as a comma-separated list of name-value pairs.
> **class-ref**
> Allows to explicitly specify the referenced type. Normally the XDoclet OJB module searches the type of the field and its sub types for the nearest type with the ojb.class tag. If the type is specified explicitly, then this type is used instead. For anonymous references, the **class-ref** has to specified as there is no field to determine the type from.
> Note that the type is required to have the ojb.class tag.
> **documentation**
> Optionally contains documentation on the reference.
> **foreignkey**
> Contains one or more foreign key fields separated by commas (required). The foreign key fields are fields with the ojb.field tag in the same class as the reference, which implement the association, i.e. contains the values of the primarykeys of the referenced object.

Other supported attributes (see repository.dtd for their meaning) written directly to the repository descriptor file:

- **auto-delete**
- **auto-retrieve**
- **auto-update**
- **otm-dependent**
- **proxy**
- **proxy-prefetching-limit**
- **refresh**

Example:

```
public abstract class AbstractArticle implements InterfaceArticle, java.io.Serializable
{
    protected InterfaceProductGroup productGroup;

    /**
     * @ojb.reference class-ref="org.apache.ojb.broker.ProductGroup"
     *                foreignkey="productGroupId"
     *                documentation="this is the reference to an articles productgroup"
     *                attributes="color=red,size=tiny"
     */
    protected InterfaceProductGroup productGroup;
    /**
```

```
     * @ojb.field
     */
    protected int productGroupId;
    ...
}
```

Here the java type is `InterfaceProductGroup` although the repository reference uses the sub type `ProductGroup`. The generated reference descriptor looks like:

```
<field-descriptor
    name="productGroupId"
    column="Kategorie_Nr"
    jdbc-type="INTEGER"
>
</field-descriptor>
<reference-descriptor
    name="productGroup"
    class-ref="org.apache.ojb.broker.ProductGroup"
>
    <documentation>this is the reference to an articles productgroup</documentation>
    <attribute attribute-name="color" attribute-value="red"/>
    <attribute attribute-name="size" attribute-value="tiny"/>
    <foreignkey field-ref="productGroupId"/>
</reference-descriptor>
```

In the torque table schema for the `Article` class, the foreign key for the product group is explicitly declared:

```
<table name="Artikel">
    ...
    <column name="Kategorie_Nr"
            javaName="productGroupId"
            type="INTEGER"
    />
    ...
    <foreign-key foreignTable="Kategorien">
        <reference local="Kategorie_Nr" foreign="Kategorie_Nr"/>
    </foreign-key>
</table>
```

For an example of an anonymous reference, see the examples of ojb.field.

4.5.16.7. Collections

ojb.collection

Persistent collections which implement 1:n or m:n associations are denoted by the **ojb.collection** tag. If the collection is an array, then the XDoclet OJB module can determine the element type automatically (analogous to references). Otherwise the type must be specified using the **element-class-ref** attribute. m:n associations are also supported (collections on both sides) via the **indirection-table**, **foreignkey** and **remote-foreignkey** attributes.

Attributes:

attributes
Optionally contains attributes of the collection as a comma-separated list of name-value pairs.

collection-class
Specifies the class that implements the collection. This attribute is usually only required if the actual type of the collection object shall be different from the variable type.
Collection classes that implement `java.util.Collection` can be handled by OJB as-is so specifying them is not necessary. For the types that do not, the XDoclet OJB module checks whether the collection field type implements the `org.apache.ojb.broker.ManageableCollection` interface, and if so, generates the collection-class attribute automatically.

documentation

Optionally contains documentation on the collection.

element-class-ref

Allows to explicitly specify the type of the collection elements. Note that the type is required to have the ojb.class tag.

foreignkey

Contains one or more foreign key field or columns separated by commas (required).
If the collection implements an 1:n association, then this attribute specifies the fields in the element type that implement the association on the element side, i.e. they refer to the primary keys of the class containing this collection. Note that these fields are required to have the ojb.field tag.
When the collection is one part of an m:n association (e.g. with an indirection table), this attribute specifies the columns in the indirection table that point to the type owning this collection. This attribute is required of both collections. If the other type has no explicit collection, use the **remote-foreignkey** attribute to specify the foreign keys for the other type.

indirection-table

Gives the name of the indirection table used for m:n associations. The XDoclet OJB module will create an appropriate torque table schema. The specified foreign keys are taken from the **foreignkey** attribute in this class and the corresponding collection in the element class, or if the element class has no collection, from the **remote-foreignkey** attribute of this collection. The XDoclet OJB module associates the foreignkeys (in the order they are stated in the **foreignkey**/ **remote-foreignkey** attributes) to the ordered primarykey fields (for the ordering rules see the ojb.field tag), and use ther jdbc type (and length setting if necessary) of these primarey keys for the columns.

orderby

Contains the fields used for sorting the collection and, optionally, the sorting order (either `ASC` or `DESC` for ascending or descending, respectively) as a comma-separated list of name-value pairs. For instance, `field1=DESC,field2,field3=ASC` specifies three fields after which to sort, the first one in descending order and the other two in ascending order (which is the default and can be omitted).

query-customizer

Specifies a query customizer for the collection. The type is required to implement `org.apache.ojb.broker.accesslayer.QueryCustomizer`.

query-customizer-attributes

Specifies attributes for the query customizer. This attribute is ignored if no query customizer is specified for this collection.

remote-foreignkey

Contains one or more foreign key columns (separated by commas) in the indirection table pointing to the elements. Note that this field should only be used if the other type does not have a collection itself which the XDoclet OJB module can use to retrieve the foreign keys. This attribute is ignored if used with 1:n collections (no indirection table specified).

The same attributes as for references are written directly to the repository descriptor file (see repository.dtd) :

- **auto-delete**
- **auto-retrieve**
- **auto-update**
- **otm-dependent**
- **proxy**
- **proxy-prefetching-limit**
- **refresh**

Examples:

```
/**
 * @ojb.collection element-class-ref="org.apache.ojb.broker.Article"
 *                 foreignkey="productGroupId"
 *                  auto-retrieve="true"
 *                  auto-update="link"
 *                  auto-delete="object"
 *                  orderby="productGroupId=DESC"
 *                  query-customizer="org.apache.ojb.broker.accesslayer.QueryCustomizerDefaultImpl"
 *                  query-customizer-attributes="attr1=value1"
 */
private ArticleCollection allArticlesInGroup;
```

The corresponding collection descriptor is:

```
<collection-descriptor
    name="allArticlesInGroup"
    element-class-ref="org.apache.ojb.broker.Article"
    collection-class="org.apache.ojb.broker.ArticleCollection"
    auto-retrieve="true"
    auto-update="link"
    auto-delete="object"
>
    <orderby name="productGroupId" sort="DESC"/>
    <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>
```

An m:n collection is defined using the **indirection-table** attribute:

```
/**
 * @ojb.class generate-table-info="false"
 */
public abstract class BaseContentImpl implements Content
{
    /**
     * @ojb.collection element-class-ref="org.apache.ojb.broker.Qualifier"
     *                  auto-retrieve="true"
     *                  auto-update="link"
     *                  auto-delete="none"
     *                  indirection-table="CONTENT_QUALIFIER"
     *                  foreignkey="CONTENT_ID"
     *                  remote-foreignkey="QUALIFIER_ID"
     */
    private List qualifiers;
    ...
}

/**
 * @ojb.class table="NEWS"
 */
public class News extends BaseContentImpl
{
    ...
}

/**
 * @ojb.class generate-table-info="false"
 */
public interface Qualifier extends Serializable
{
    ...
}
```

The `BaseContentImpl` has a m:n association to the `Qualifier` interface. for the `BaseContentImpl` class, this association is implemented via the `CONTENT_ID` column (specified by the **foreignkey**) in the indirection table `CONTENT_QUALIFIER`. Usually, both ends of an m:n association have a collection implementing the association, and for both ends the **foreignkey** specifies the indirection table column pointing to the class at this end. The `Qualifier` interface however does not contain a collection which could be used to determine the indirection table column that implements the association from its side. So, this column is also specified in the `BaseContentImpl` class using the **remote-foreignkey** attribute. The class descriptors are:

```
<class-descriptor
    class="org.apache.ojb.broker.BaseContentImpl"
>
    <extent-class class-ref="org.apache.ojb.broker.News"/>
</class-descriptor>

<class-descriptor
    class="org.apache.ojb.broker.News"
    table="NEWS"
>
    ...
    <collection-descriptor
        name="qualifiers"
        element-class-ref="org.apache.ojb.broker.Qualifier"
        indirection-table="CONTENT_QUALIFIER"
        auto-retrieve="true"
        auto-update="link"
        auto-delete="none"
    >
        <fk-pointing-to-this-class column="CONTENT_ID"/>
        <fk-pointing-to-element-class column="QUALIFIER_ID"/>
    </collection-descriptor>
</class-descriptor>

<class-descriptor
    class="org.apache.ojb.broker.Qualifier"
>
    <extent-class class-ref="org.apache.ojb.broker.BaseQualifierImpl"/>
</class-descriptor>
```

As can be seen, the collection definition is inherited in the `News` class and the two indirection table columns pointing to the ends of the m:n associaton are correctly specified.

4.5.16.8. Nested objects

ojb.nested

The features of a class can be included in another class by declaring a field of that type and using this tag. The XDoclet OJB module will then add every tagged feature (i.e. fields/bean properties with ojb.field, ojb.reference or ojb.collection tag, or even with **ojb.nested**) from the type of the field to the current class descriptor. It is not required that the field's type has the ojb.class tag, though.
All attributes of the features are copied (even **primarykey**) and modified if necessary (e.g. the **foreignkey** of a reference is adjusted accordingly). For changing an attribute use the ojb.modify-nested tag.

For an example of nesting, see the example of ojb.modify-nested.

ojb.modify-nested

Similar to ojb.modify-inherited, this tag allows to modify attributes of a nested feature.

Attributes: All of ojb.field, ojb.reference, and ojb.collection (with the exception of **indirection-table** and **remote-foreignkey**), and also:

ignore : true | false (default)
Specifies that this feature will not be nested.
name
The name of the field/reference/collection to modify (required). Use here the name of the feature in the nested

type.

Example:

The two classes:

```
public class NestedObject implements java.io.Serializable
{
    /** @ojb.field primarykey="true" */
    protected int id;

    /** @ojb.field */
    protected boolean hasValue;

    /** @ojb.field */
    protected int containerId;

    /**
     * @ojb.reference foreignkey="containerId"
     */
    protected ContainerObject container;

    ...
}

/** @ojb.class */
public class ContainerObject implements java.io.Serializable
{
    /**
     * @ojb.field primarykey="true"
     *            autoincrement="ojb"
     *            id="1"
     */
    protected int id;

    /** @ojb.field id="2" */
    protected String name;

    /**
     * @ojb.nested
     * @ojb.modify-nested name="hasValue"
     *                    jdbc-type="INTEGER"
     *
conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"
     *                    id="3"
     * @ojb.modify-nested name="id"
     *                    primarykey=""
     */
    protected NestedObject nestedObj;

    ...
}
```

result in the one class descriptor

```
<class-descriptor
    class="ContainerObject"
    table="ContainerObject"
>
    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
    />
    <field-descriptor
        name="name"
        column="name"
        jdbc-type="VARCHAR"
        length="24"
    />
    <field-descriptor
        name="nestedObj::hasValue"
        column="nestedObj_hasValue"
        jdbc-type="INTEGER"
        conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"
    />
    <field-descriptor
        name="nestedObj::id"
        column="nestedObj_id"
        jdbc-type="INTEGER"
    />
    <field-descriptor
        name="nestedObj::containerId"
        column="nestedObj_containerId"
        jdbc-type="INTEGER"
    />
    <reference-descriptor
        name="nestedObj::container"
        class-ref="ContainerObject"
    >
        <foreignkey field-ref="nestedObj::containerId"/>
    </reference-descriptor>
    ...
</class-descriptor>
```

and the table descriptor

```
<table name="ContainerObject">
    <column name="id"
            javaName="id"
            type="INTEGER"
            primaryKey="true"
            required="true"
    />
    <column name="name"
            javaName="name"
            type="VARCHAR"
            size="24"
    />
    <column name="nestedObj_hasValue"
            type="INTEGER"
    />
    <column name="nestedObj_id"
            type="INTEGER"
    />
    <column name="nestedObj_containerId"
            type="INTEGER"
    />
    <foreign-key foreignTable=\"ContainerObject\">\n"+
        <reference local=\"nestedObj_containerId\" foreign=\"id\"/>\n"+
    </foreign-key>\n"+
    ...
</table>
```

Note how one **ojb.modify-nested** tag changes the type of the nested `hasValue` field, add a `conversion` and specifies the position for it. The other modification tag removes the `primarykey` status of the nested `id` field.

4.5.17. OJB Performance

4.5.17.1. Introduction

" There is no such thing as a free lunch."

(North American proverb)

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. Introducing new software layers always eats up additional computing resources.
In short: the price for using O/R tools is performance.

Software architects have to take in account this tradeoff between programming comfort and performance to decide if it is appropiate to use an O/R tool for a specific software system.

This document describes the *OJB Performance Test Suite* which was created to lighten the decision between native JDBC, OJB (the different OJB API's) and other O/R mapper.

4.5.17.2. The Performance Test Suite

The *OJB Performance Test Suite* allows to compare OJB against native JDBC programming against your RDBMS of choice and run OJB in a virtual multithreaded environment. Further on it is possible to compare OJB against any O/R mapping tool using a simple framework.

All tests are integrated in the OJB build script, you only need to perform the according ant target:

```
ant target
```

The following 'targets' exist:

- `perf-test` multithreaded performance/stress test of PB/OTM/ODMG api against native JDBC
- `performance` older single threaded test, OJB API implementations (PB, ODMG) against native JDBC
- [`performance3` multithreaded test against two different databases - developers test]

By changing the JdbcConnectionDescriptor in the configuration files you can point to your specific RDBMS. Please refer to this document for details.

4.5.17.3. Interpreting test results

Interpreting the result of these benchmarks carefully will help to decide whether using OJB is viable for specific application scenarios or if native JDBC programming should be used for performance reasons.

Take care of compareable configuration properties when run performance tests with different O/R tools.

If the decision made to use an O/R mapping tool the comparison with other tools helps to find the best one for the thought scenario. 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
- Community support
- The different licences of Open Source projects
- etcetera ...

4.5.17.4. How OJB compares to native JDBC programming?

OJB is shipped with tests compares native JDBC with ODMG and PB-API implementation. This part of the test suite is integrated into the OJB build mechanism.
A single client test you can invoke it by typing `ant performance` or `ant performance`.

If running OJB out of the box the tests will be performed against the Hypersonic SQL shipped with OJB. A typical output looks like follows:

```
performance:
      [ojb] .[performance] INFO: Test for PB-api
      [ojb] [performance] INFO:
      [ojb] [performance] INFO: inserting 2500 Objects: 3257 msec
      [ojb] [performance] INFO: updating 2500 Objects: 1396 msec
      [ojb] [performance] INFO: querying 2500 Objects: 1322 msec
      [ojb] [performance] INFO: querying 2500 Objects: 26 msec
      [ojb] [performance] INFO: fetching 2500 Objects: 495 msec
      [ojb] [performance] INFO: deleting 2500 Objects: 592 msec
      [ojb] [performance] INFO:
      [ojb] [performance] INFO: inserting 2500 Objects: 869 msec
      [ojb] [performance] INFO: updating 2500 Objects: 1567 msec
      [ojb] [performance] INFO: querying 2500 Objects: 734 msec
      [ojb] [performance] INFO: querying 2500 Objects: 20 msec
      [ojb] [performance] INFO: fetching 2500 Objects: 288 msec
      [ojb] [performance] INFO: deleting 2500 Objects: 447 msec
      [ojb] [performance] INFO:
      [ojb] [performance] INFO: inserting 2500 Objects: 979 msec
      [ojb] [performance] INFO: updating 2500 Objects: 1240 msec
      [ojb] [performance] INFO: querying 2500 Objects: 741 msec
      [ojb] [performance] INFO: querying 2500 Objects: 18 msec
      [ojb] [performance] INFO: fetching 2500 Objects: 289 msec
      [ojb] [performance] INFO: deleting 2500 Objects: 446 msec

      [ojb] Time: 18,964

      [ojb] OK (1 test)

     [jdbc] .[performance] INFO: Test for native JDBC
     [jdbc] [performance] INFO:
     [jdbc] [performance] INFO: inserting 2500 Objects: 651 msec
     [jdbc] [performance] INFO: updating 2500 Objects: 775 msec
     [jdbc] [performance] INFO: querying 2500 Objects: 616 msec
     [jdbc] [performance] INFO: querying 2500 Objects: 384 msec
     [jdbc] [performance] INFO: fetching 2500 Objects: 49 msec
     [jdbc] [performance] INFO: deleting 2500 Objects: 213 msec
     [jdbc] [performance] INFO:
     [jdbc] [performance] INFO: inserting 2500 Objects: 508 msec
     [jdbc] [performance] INFO: updating 2500 Objects: 686 msec
     [jdbc] [performance] INFO: querying 2500 Objects: 390 msec
     [jdbc] [performance] INFO: querying 2500 Objects: 360 msec
     [jdbc] [performance] INFO: fetching 2500 Objects: 46 msec
     [jdbc] [performance] INFO: deleting 2500 Objects: 204 msec
     [jdbc] [performance] INFO:
     [jdbc] [performance] INFO: inserting 2500 Objects: 538 msec
     [jdbc] [performance] INFO: updating 2500 Objects: 775 msec
     [jdbc] [performance] INFO: querying 2500 Objects: 384 msec
     [jdbc] [performance] INFO: querying 2500 Objects: 360 msec
     [jdbc] [performance] INFO: fetching 2500 Objects: 48 msec
     [jdbc] [performance] INFO: deleting 2500 Objects: 204 msec

     [jdbc] Time: 18,363

     [jdbc] OK (1 test)

     [odmg] .[performance] INFO: Test for ODMG-api
     [odmg] [performance] INFO:
     [odmg] [performance] INFO: inserting 2500 Objects: 12151 msec
     [odmg] [performance] INFO: updating 2500 Objects: 2937 msec
     [odmg] [performance] INFO: querying 2500 Objects: 4691 msec
     [odmg] [performance] INFO: querying 2500 Objects: 2239 msec
     [odmg] [performance] INFO: fetching 2500 Objects: 1633 msec
     [odmg] [performance] INFO: deleting 2500 Objects: 1815 msec
     [odmg] [performance] INFO:
     [odmg] [performance] INFO: inserting 2500 Objects: 2483 msec
     [odmg] [performance] INFO: updating 2500 Objects: 2868 msec
     [odmg] [performance] INFO: querying 2500 Objects: 3272 msec
     [odmg] [performance] INFO: querying 2500 Objects: 2223 msec
     [odmg] [performance] INFO: fetching 2500 Objects: 1038 msec
     [odmg] [performance] INFO: deleting 2500 Objects: 1717 msec
     [odmg] [performance] INFO:
     [odmg] [performance] INFO: inserting 2500 Objects: 2666 msec
     [odmg] [performance] INFO: updating 2500 Objects: 2841 msec
     [odmg] [performance] INFO: querying 2500 Objects: 2092 msec
     [odmg] [performance] INFO: querying 2500 Objects: 2161 msec
     [odmg] [performance] INFO: fetching 2500 Objects: 1036 msec
     [odmg] [performance] INFO: deleting 2500 Objects: 1741 msec

     [odmg] Time: 55,186
```

Some notes on these test results:

- You see a consistently better performance in the second and third run. this is caused by warming up effects of JVM and OJB.
- PB and native JDBC need about the same time for the three runs although JDBC performance is better for most operations. this is caused by the second run of the querying operations. In the second run OJB can load all objects from the cache, thus the time is **much** shorter. Hence the interesting result: if you have an application that has a lot of lookups, OJB can be faster than a native JDBC application!
- ODMG is much slower than PB or JDBC. This is due to the complex object level transaction management it is doing.
- You can see that for HSQLDB operations like insert and update are much faster with JDBC than with PB (60% and more). This ratio is so high, because HSQLDB is much faster than ordinary database servers (as it's inmemory). If you work against Oracle or DB2 the percentual OJB overhead is going down a lot (10 - 15 %), as the database latency is much longer than the OJB overhead.

It's easy to change target database. Please refer to this document for details.
Also it's possible to change the number of test objects by editing the ant-target in build.xml.

Another test compares PB-api,ODMG-api and native JDBC you can find next section.

4.5.17.5. OJB performance in multi-threaded environments

This test was created to check the performance and stability of the supported API's (PB-api, ODMG-api, JDO-api) in a multithreaded environment. Also this test compares the api's and native JDBC.
Running this test out of the box (a virgin OJB version against hsql) shouldn't cause any problems. To run the JDO-api test too, see JDO tutorial and comment in the test in target `perf-test` in `build.xml`

FIXME (arminw):

A test for JDO API is missed.

Per default OJB use hsql as database, by changing the JdbcConnectionDescriptor in the repository.xml file you can point to your specific RDBMS. Please refer to this document for details.

To run the multithreaded performance test call

```
ant perf-test
```

A typical output of this test looks like (OJB against hsql server, 2-tier, 100 MBit network):

```
[ojb] ========================================================
[ojb]         OJB PERFORMANCE TEST SUMMARY
[ojb] 10 concurrent threads, handle 2000 objects per thread
[ojb]         - performance mode
[ojb] ========================================================
[ojb]      API  Period   Total  Insert   Fetch  Update  Delete
[ojb]            [sec]   [sec]  [msec]  [msec]  [msec]  [msec]
[ojb] --------------------------------------------------------
[ojb]     JDBC   7.786   7.374    3951      76    2435     911
```

```
[ojb]        PB   9.807   8.333    5096      121     2192      922
[ojb]      ODMG  19.562  18.205    8432     1488     5064     3219
[ojb]       OTM  24.953  21.272   10688      223     4326     6033
[ojb] ==============================================================

[ojb] PerfTest takes 191 [sec]
```

To change the test properties go to target `perf-test` in the `build.xml` file and change the program parameter.
The test needs five parameter:
- A comma separated list of the test implementation classes (no blanks!)
- The number of test loops
- The number of concurrent threads
- The number of managed objects per thread
- The desired test mode. `false` means run in performance mode, `true` means run in stress mode (useful only for developer to check stability).

```
<target name="perf-test" depends="prepare-testdb"
            description="Simple performance benchmark and stress test for PB- and ODMG-api">
    <java fork="yes" classname="org.apache.ojb.performance.PerfMain"
          dir="${build.test}/ojb" taskname="ojb" failonerror="true" >
        <classpath refid="runtime-classpath"/>

        <!-- comma separated list of the PerfTest implementations -->
        <arg value=
        "org.apache.ojb.broker.OJBPerfTest$JdbcPerfTest,\
        org.apache.ojb.broker.OJBPerfTest$PBPerfTest,\
        org.apache.ojb.broker.OJBPerfTest$ODMGPerfTest,\
        org.apache.ojb.broker.OJBPerfTest$OTMPerfTest"
        />
        <!-- test loops, default was 3 -->
        <arg value="3"/>
        <!-- performed threads, default was 10 -->
        <arg value="10"/>
        <!-- number of managed objects per thread, default was 2000 -->
        <arg value="2000"/>
        <!-- if 'false' we use performance mode, 'true' we do run in stress mode -->
        <arg value="false"/>

        <jvmarg value="-Xms128m"/>
        <jvmarg value="-Xmx256m"/>
    </java>
    <!-- do some cleanup -->
    <ant target="copy-testdb"/>
</target>
```

4.5.17.6. How OJB compares to other O/R mapping tools?

Many user ask this question and there is more than one answer. But OJB was shipped with a simple performance *"framework"* (independend from OJB) which allows a rudimentarily comparision of OJB with other (java-based) O/R mapping tools.
All involved classes can be found in dirctory *[db-ojb]/src/test* in package `org.apache.ojb.performance`.

Call `ant perf-test-jar` to build the jar file contain all necessary classes to set up a test with an arbitrary O/R mapper. After the build, the `db-ojb-XXX-performance.jar` can be found in `[db-ojb]/dist` directory.

Steps to set up the test for other O/R frameworks:

- Implement a class derived from `PerfTest`
- Implement a class derived from `PerfHandle`
- [If persistent objects used within your mapping tool must be derived from a specific base class or must be implement a specific interface write your own persistent object class by implementing `PerfArticle` interface and **override method** `#newPerfArticle()` in your `PerfHandle` implementation class.

Otherwise a default implementation of `PerfArticle` was used]

That's it!

You can find a example implementation called `org.apache.ojb.broker.OJBPerfTest` in the test-sources directory under `[db-ojb]/src/test` (when using source-distribution). This implementation class is used to compare performance of the PB-API, ODMG-API, OTM-api and native JDBC.

See more section multi-threaded performance. `OJBPerfTest` is made up of inner classes. At each case two inner classes represent a test for one api (as described above).

Run the test

You have two possibilities to run the test:

a) Integration in the OJB build script

Add the full qualified class name of your PerfTest implementation class to the `perf-test` target of the OJB `build.xml` file, add all necessary jar files to `[db-ojb]/lib`. The working directory of the test is `[db-ojb]/target/test/ojb`.

b) Run PerfMain

It's possible to run the test using `org.apache.ojb.performance.PerfMain`.

```
java -classpath CLASSPATH org.apache.ojb.performance.PerfMain
[comma separated list of PerfTest implementation classes, no blanks!]
[number of test loops]
[number of threads]
[number of insert/fetch/delete loops per thread]
[boolean - run in stress mode if set true,
run in performance mode if set false, default false]
```

For example:

```
java -classpath CLASSPATH my.A_PerfTest,my.B_PerfTest 3 10 2000 false
```

This will use `A_PerfTest` and `B_PerfTest` and perform three loops each loop run 10 threads and each thread operate on 2000 objects. The test run in *performance* mode.

Take care of compareable configuration properties when run performance tests with different O/R tools (caching, locking, sequence key generation, connection pooling, ...).

> **Note:**
>
> **Please, don't start flame wars** by posting performance results to mailing lists made with this simple test. This test was created for OJB QA and to give a clue how good or bad OJB performs, NOT to start discussion like *XY is 12% faster then XZ*!!.

4.5.17.7. What are the best settings for maximal performance?

We don't know, that depends from the environment OJB runs (hardware, database, driver, application server, ...). But there are some settings which affect the performance:

- The API you use, e.g. PB-api is much faster then the ODMG-api. See which API for more information.
- ConnectionFactory implementation / Connection pooling. See connection pooling for more information.
- PersistentField class implementation.See OJB.properties section 'PersistentFieldClass' for more information.
- Used sequence manager implementation. See sequence manager for more information.
- Use of batch mode (when supported by the DB). See repository.dtd element 'jdbc-connection-descriptor' for more information.
- PersistenceBroker pool size. See OJB.properties for more information.

To test the different settings use the tests of the performance test suite.

4.6. Howto's

4.6.1. Howto's Summary

4.6.1.1. Howto's

Here can be found a summary of all Howto documentation submitted by OJB Users and Developers.

- How to build large metadata mappings
- Using anonymous keys for cleaner objects
- Using native database sequences
- Using Oracle LOB's
- Using OJB in a clustered environment
- Working with stored procedures

4.6.2. How to build O/R mapping meta data files

4.6.2.1. How to build O/R mapping files

Writing the repository.xml file for only a few classes can easily be done manually with the text or xml editor of your choice.

But keeping the repository in sync with the java codebase and the database gets more difficult if several hundred classes and large developer teams are involved.

This page contains tips how to integrate mapping tools and code-generators into your build process.

4.6.2.2. classification of O/R related transformations

Let's start with a classification of the source transformation problems that developers have to face in an O/R environment.

Typical development environments contain some or all of the following artefacts:

- A UML model containing at least class diagrams of the persistent classes. All modern UML tools can export to the XMI standard format.
- Other tools, such as Torque, also use a model based approach but use different model file formats (typically XML based)
- Java source code for the persistent classes. The Java source code can possibly be enhanced with xdoclet tags.
- The OJB repository.xml file. This file contains all the class-descriptors for the persistent classes.
- The database. This could be an online DB or a DDL script representing the database tables. The database contains all tables used to store instances of the persistent classes.

The technique you will use depends a lot on the problem you have to solve, on the methodology and the tool chain you have in use, which of transformations between those artefacts fit to your development process.

1. **Forward engineering from XMI:** A UML model in XMI format with class diagrams of your persistent classes exists and is used as the master source (model driven approach). Java code, repository.xml and DDL for the database tables have to be generated from this model.
2. **Forward engineering from Torque:** A model of the persistent entity classes exists in form of a Torque.XML file. Java code, repository.xml and DDL for the database tables have to be generated from this model.
3. **Forward engineering from the repository.xml:** The OJB repository.xml file is used a model format. Java code and DDL for the database tables have to be generated from this model.
4. **Xdoclet transformation from Java code:** Java code for the persistent classes exists and contains special comment tags in the Xdoclet ojb-module format. Repository.xml and DDL for the database tables have to be generated from the java files via Xdoclet transformation.
5. **Reverse engineering from database:** There is a database with existing tables or a DDL script. Java code and repository.xml have to be generated from the database.

These transformations are depicted in the following graphics. The numbers close to the arrows correspond to the numbers in the above enumeration. All related transformations have the same colour.

mapping tools image

In the following sections we will have a closer look at each of these transformations an discuss tools that provide support each approach.

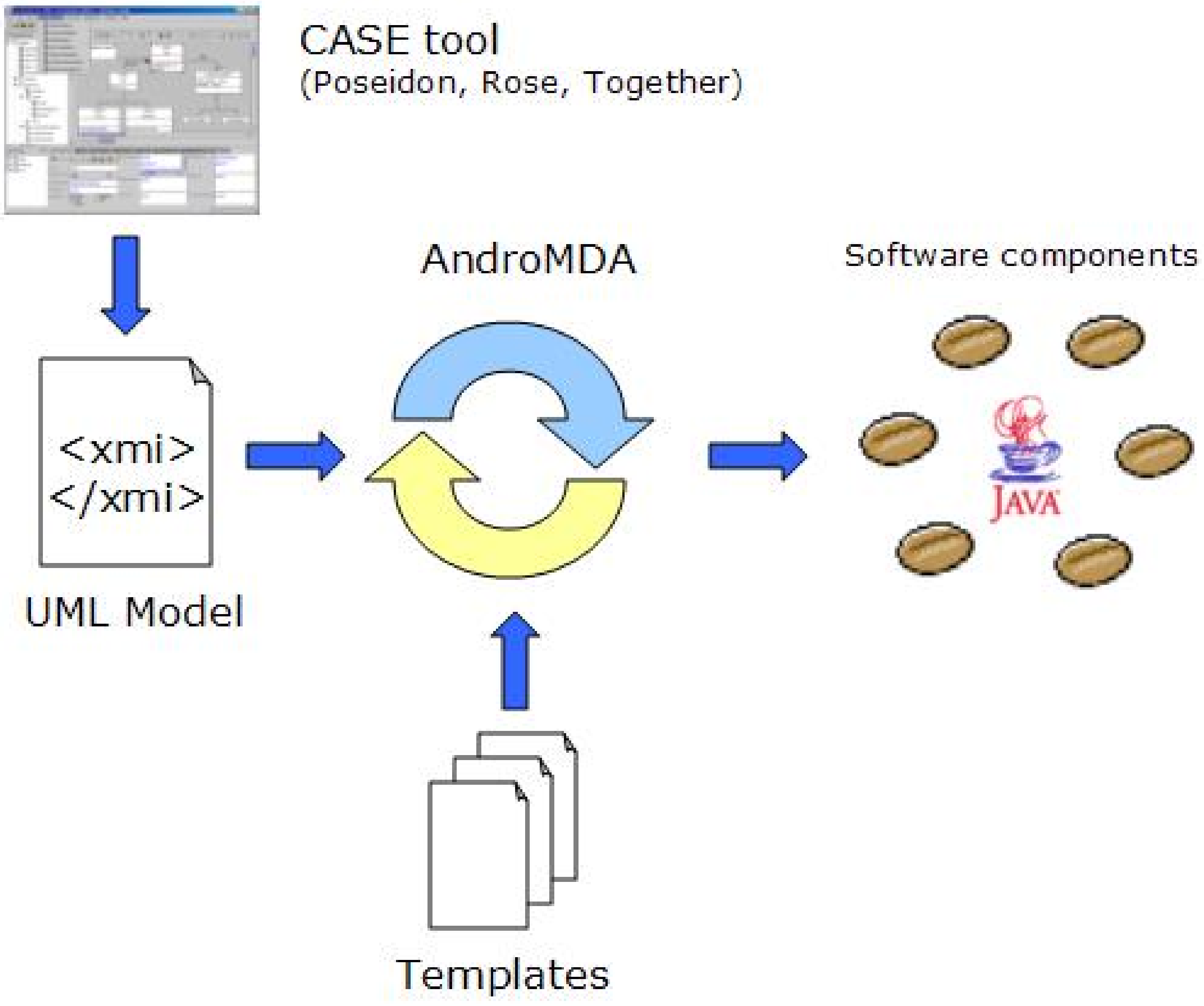
4.6.2.3. Forward engineering from XMI

This approach is recommended if you start from scratch with a new project and have to deal with a large number of persistent classes. This approach works best when there are no restrictions regarding the database, like integration of legacy tables.

Forward engineering from XMI fits perfectly into a model driven architecture (MDA) software development process.

Tool support

- **AXGen**
  AXgen is a code generator using XMI as input and Velocity templates for transformation.
  The power of AXgen is in its simplicity. You don't have to understand complicated structure of your XMI file to write an XSLT stylsheet for transformation. AXgen uses nsuml to deal with the xmi file, which gives access to the Metamodel in an objectoriented way.
  Further AXgen makes use of Jakartas Velocity. Velocity is a very sophisticated Java-based template engine. This means that inside your templates you can call Java methods. Feel free to write templates that generate anything you want.
  Our motive for AXgen is to generate Java Classes for use in a O/R Mapping tool that allows transparent persistence for Java Objects against relational databases. Therefore AXgen comes with a bundle of ready to use templates for generating ObJectRelationalBridge (OJB) specific stuff like:
  - Entity Classes
  - XML Repository
  - SQL script to build the DB scheme
- **AndroMDA**
  AndroMDA is a code generator framework - it takes a Unified Modeling Language (UML) model from a CASE-tool in XMI format and generates custom components. It comes with a set of sample templates that generate classes attributed with XDoclet tags. One build step later, the XDoclet tool generates full-blown components that can readily be deployed in the JBoss application server (and the other servers that XDoclet can feed).



andromeda image

Currently AndroMDA provides no special support for OJB. But by tagging classes with tags of the XDoclet OJB module it is possible to use it as a full forward engineering engine.

- Searching the Sourceforge project list for "XMI" returns a long list of projects dealing with code generation. It may be a good idea to check if you find a tool that matches your requirements.

4.6.2.4. Forward engineering from Torque

Torque

Torque is a persistence layer. Torque includes a generator to generate all the database resources required by your application and includes a runtime environment to run the generated classes.

Torque was developed as part of the Turbine Framework. It is now decoupled and can be used by itself. Starting with version 2.2 Turbine uses the decoupled Torque.

Torque uses a single XML database schema to generate the SQL for your target database and Torque's Peer-based object relation model representing your XML database schema.

You can use devaki-nextobjects to create the model for your application.

OJB uses Torque's generator engine to setup the testbed database and feed it with initial data.

Besides the SQL generation facilities Torque also provides special support for OJB related transformations. It provides the following two ant targets:

- **ojb-model**
  generates a simple object model for ojb
- **ojb-repository**
  generates the repository for ojb

A complete list of all availableTorque targets can be found at the Torque Generator site.

4.6.2.5. Forward engineering from repository.xml

There is currently no tool available that directly supports this model. It is not difficult to implement an XSLT stylesheet that transforms the OJB repository.xml directly into DDL Statements.

An even simpler approach could be to transform the repository.xml file into a Torque xml file. DDL can then be generated by the Torque engine.
If you write such an XSLT file please tell us about it!

4.6.2.6. XDoclet transformation from Java code

XDoclet
XDoclet is a code generation engine. It enables Attribute-Oriented Programming for java. In short, this means that you can add more significance to your code by adding meta data (attributes) to your java sources. This is done in special JavaDoc tags.

OJB was shipped with its own **xdoclet-module**.

XDoclet will parse your source files and generate many artifacts such as XML descriptors and/or source code from it. These files are generated from templates that use the information provided in the source code and its JavaDoc tags.

XDoclet lets you apply Continuous Integration in component-oriented development. Developers should concentrate their editing work on only one Java source file per component.

XDoclet originated as a tool for creating EJBs, it has evolved into a general-purpose code generation engine. XDoclet consists of a core and a constantly growing number of modules.

4.6.2.7. Reverse engineering from database

- **Druid**
  Druid is a tool that allows users to create databases in a graphical way. The user can add or import tables, fields, folders to group tables and can modify most of the database options that follow the SQL-92 standard. In addition to sql options, the user can document each table and each field with HTML information. It is distributed with modules for generating Java classes, OJB metadata, and JDO metadata.
- **Impart Eclipse Plugin for OJB**
  The Impart Eclipse plugin is based on the OJB ReverseDB Tool and provides the same functionality (and also some additional goodies). It ships as a plugin to the Eclipse IDE. It provides a very convenient GUI that integrates smoothly into the Eclipse platform.
- **RDBS2J**
  RDBS2J is a GUI based mapping tool from relational database scheme to persistent java classes which use JDO as persistence mechanism. The mapping can be modified by the GUI.
  The current version is designed to create code for OJB.
  The ODMG and the JDO interface are supported. RDBS2J creates the *.jdo files and the repository_user.xml, which are needed by OJB.
- **The OJB ReverseDB tool**
  OJB ships with a simple reverse engineering tool that allows to connect to a RDBMS via JDBC and to take the tables from

the database catalog as input.
This tool provides a nice GUI to generate Java classes and the matching repository.xml file.
You can invoke the ReverseDB tool with the ANT target `reverse-db`.

Note:
The ReverseDB tool is not up to date - any help is welcome.

4.6.3. HOWTO - Use Anonymous Keys

4.6.3.1. Why Do We Need Anonymous Keys?

The core difference between referential integrity in Java and in an RDBMS lies in where the specific referential information is maintained. Java, and most modern OO languages, maintain referential integrity information in the runtime environment. Actual object relationships are maintained by the virtual machine so that the symbolic variable used in the application is dereferenced it will in fact provide access to the object instance which it is expected to provide access to. There is no need for a manual lookup or search across the heap for the correct object instance. Entity reference integrity is maintained and handled for the programmer by the environment.

Relational databases, on the other, purposefully place the referential integrity and lookups into the problem domain - that is the problem they are designed to solve. An RDBMS presumes you can design something more efficient for your specific circumstances than the JVM does (you trust its ability to do object lookups in the heap is sufficiently efficient). As an RDBMS has a much larger heap equivalent it is designed to not operate under that assumption (mostly). So, in an RDBMS the concept of specific foreign keys exists to maintain the referential integrity.

In crossing the object to relational entity barrier there is a mismatch between the referential integrity implementations. Java programmers do not want to have to maintain both object referential integrity and key referential integrity analogous to

```
{
Foo child = new SomeOtherFooType();
Foo parent = new SomeFooType();
child.setParent(parent);
child.setParentId(parent.getId());
}
```

This is double the work required - you set up the object relationship, then set up the key relationship. OJB knows about the relationship of the objects, thus it is only needed to do

```
{
Foo child = new Foo();
Foo parent = new Foo();
child.setParent(parent);
}
```

OJB can provide transparent key relationship maintenance behind the scenes for 1:1 relations via **anonymous access fields**. As object relationships change, the relationships will be propogated into the key values without the *Java object ever being aware of a relational key* being in use. This means that the java object does not need to specify a FK field for the reference.

Without use of *anonymous keys* class `Foo` have to look like:

```
class Foo
{
    Integer id;
    Integer fkParentFoo;
    Foo parent;

    // optional getter/setter
    ....
{
```

When using *anonymous keys* the FK field will become obsolete:

```
class Foo
{
    Integer id;
    Foo parent;

    // optional getter/setter
    ....
{
```

Note:

Under specific conditions it's also possible to use anonymous keys for other relations or primary keys. More info in advanced-technique section.

4.6.3.2. How it works

To play for safety it is mandatory to understand how this feature is working. More information how it works please see here.

4.6.3.3. Using Anonymous Keys

Now we can start using of the *anonymous key* feature. In this section the using is detailed described on the basis of an example.

The Code

Take the following classes designed to model a particular problem domain. They may do it reasonably well, or may not. Presume they model it perfectly well for the problem being solved.

```
public class Desk
{
    private Finish finish;
    /** Contains Drawer instances */
    private List drawers;
    private int numberOfLegs;
    private Integer id;

    public Desk()
    {
        this.drawers = new ArrayList();
    }

    public List getDrawers()
    {
        return this.drawers;
    }

    public int getNumberOfLegs()
    {
        return this.numberOfLegs;
    }

    public void setNumberOfLegs(int num)
    {
        this.numberOfLegs = num;
    }

    public Finish getFinish()
    {
        return this.finish;
    }

    public void setFinish(Finish finish)
    {
        this.finish = finish;
    }
}


public class Drawer
{
    /** Contains Thing instances */
    private List stuffInDrawer;
    private Integer id;

    public List getStuffInDrawer()
    {
        return this.stuffInDrawer;
    }

    public Drawer()
    {
        this.stuffInDrawer = new ArrayList();
    }
}


public class Finish
{
    private String wood;
    private String color;
    private Integer id;

    public String getWood()
    {
        return this.wood;
    }

    public void setWood(String wood)
    {
        this.wood = wood;
    }

    public String getColor()
    {
        return this.color;
    }

    public void setColor(String color)
    {
        this.color = color;
    }
}

public class Thing
{
    private String name;
    private Integer id;

    public String getName()
    {
        return this.name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
}
```

A Desk will typically reference multiple drawers and one finish.

The Database

When we need to store our instances in a database we use a fairly typical table per class persistance model.

```
CREATE TABLE finish
(
    id           INTEGER PRIMARY KEY,
    wood         VARCHAR(255),
    color        VARCHAR(255)
);

CREATE TABLE desk
(
    id           INTEGER PRIMARY KEY,
    num_legs     INTEGER,
    finish_id    INTEGER,
    FOREIGN KEY (finish_id) REFERENCES finish(id)
);

CREATE TABLE drawer
(
    id           INTEGER PRIMARY KEY,
    desk_id      INTEGER,
    FOREIGN KEY (desk_id) REFERENCES desk(id)
);

CREATE TABLE thing
(
    id           INTEGER PRIMARY KEY,
    name         VARCHAR(255),
    drawer_id    INTEGER,
    FOREIGN KEY (drawer_id) REFERENCES drawer(id)
);
```

At the database level the possible relationships need to be explicitely defined by the foreign key constraints. These model all the possible object relationships according to the domain model (until generics enter the Java language for the collections API, this is technically untrue for the classes used here).

The Repository Configuration

When we go to map the classes to the database, it is almost a one-to-one property to field mapping. The exception here is the primary key on each entity. This is meaningless information in Java, so we would like to keep it out of the object model. Anonymous access keys allow us to do that.

The repository.xml must know about the database columns used for referential integrity, but OJB can maintain the foreign key relationships behind the scenes - freeing the developer to focus on more accurate modeling of her objects to the problem, instead of the the persistance mechanism. Doing this is also very simple - in the repository.xml file mark the field descriptors with a `access="anonymous"` attribute.

```
<class-descriptor
    class="Desk"
    table="desk">

    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
    <field-descriptor
        name="numberOfLegs"
        column="num_legs"
        jdbc-type="INTEGER"
        />
    <field-descriptor
```

```
        name="finishId"
        column="finish_id"
        jdbc-type="INTEGER"
        access="anonymous" />

    <collection-descriptor
        name="drawers"
        element-class-ref="Drawer"
        >
        <inverse-foreignkey field-ref="deskId"/>
    </collection-descriptor>

    <reference-descriptor
        name="finish"
        class-ref="Finish">
            <foreignkey field-ref="finishId"/>
    </reference-descriptor>
</class-descriptor>

<class-descriptor
    class="Finish"
    table="finish">

    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
    <field-descriptor
        name="wood"
        column="wood"
        jdbc-type="VARCHAR"
        size="255"
        />
   <field-descriptor
        name="color"
        column="color"
        jdbc-type="VARCHAR"
        size="255"
        />
</class-descriptor>

<class-descriptor
    class="Drawer"
    table="drawer">

    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
    <field-descriptor
        name="deskId"
        column="desk_id"
        jdbc-type="INTEGER"
        access="anonymous"
        />
    <collection-descriptor
        name="stuffInDrawer"
        element-class-ref="Thing"
        >
        <inverse-foreignkey field-ref="drawerId"/>
    </collection-descriptor>
</class-descriptor>
```

```
<class-descriptor
    class="Thing"
    table="thing">

    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
    <field-descriptor
        name="name"
        column="name"
        jdbc-type="VARCHAR"
        size="255"
        />
    <field-descriptor
        name="drawerId"
        column="drawer_id"
        jdbc-type="INTEGER"
        access="anonymous"
        />
</class-descriptor>
```

Look first at the class descriptor for the Thing class. Notice the field-descriptor with the name attribute "drawerId". This field is labeled as anonymous access. Because it is anonymous access OJB will not attempt to assign the value here to a "drawerId" field or property on the Thing class. Normally the name attribute is used as the Java name for the attribute, in this case it is not. The name is still required because it is used as an indicated for references to this anonymous field.

In the field descriptor for Drawer, look at the collection descriptor with the name *stuffInDrawer*. This collection descriptor references a foreign key with the `field-ref="drawerId"`. This reference is to the anonymous field descriptor in the Thing descriptor. The field-ref matches to the name in the descriptor whether or not the name also maps to the Java attribute name. This dual use of `name` can be confusing - be careful.

The same type mapping that is used for the collection descriptor in the Drawer descriptor is also used for the 1:1 reference descriptor in the Desk descriptor.

The primary keys are populated into the objects as it is generally a good practice to not implement primary keys as anonymous access fields. Primary keys may be anonymous-access but references will get lost if the cache is cleared or the persistent object is serialized.

4.6.3.4. Benefits and Drawbacks

There are both benefits and drawbacks to using anonymous field references for maintaining referential integrity between Java objects and database relations. The most immediate benefit is avoiding semantic code duplication. The second major benefit is avoiding cluttering class definitions with persistance mechanism artifacts. In a well layered application, the persistance mechanism will not really need to be so obvious in the object model implementation. Anonymous fields helpt o achieve this - thereby making persistence mechanisms more flexible. Moving to a different one becomes easier.

4.6.4. HOWTO - Use DB Sequences

4.6.4.1. Introduction

It is easy to use OJB with with database generated sequences. Typically a table using database generated sequences will autogenerate a unique id for a field as the default value for that field. This can be particularly useful if multiple applications access the same database. Not every application will be using OJB and find it convenient to pull unique values from a high/low table. Using a database managed sequence can help to enforce unique id's across applications all adding to the same database. All of that said, care needs to be taken as using database generated sequences imposes some portability problems.

OJB includes a sequence manager implementation that is aware of database sequences and how to use them. It is known to work against Oracle, SAP DB, and PostgreSQL. MySQL has its own sequence manager implementation because it is special. This tutorial will build against PostgreSQL, but working against Oracle or SAP will work the same way.

Additional information on sequence managers is available in the Sequence Manager documentation.

4.6.4.2. The Sample Database

Before we can work with OJB against a database with a sequence, we need the database. We will create a simple table that pulls its primary key from a sequence named 'UniqueIdentifier'.

```
CREATE TABLE thingie
(
    name VARCHAR(50),
    id INTEGER DEFAULT NEXTVAL('UniqueIdentifier')
)
```

We must also define the sequence from which it is drawing values:

```
CREATE SEQUENCE UniqueIdentifier;
```

So that we have the following table:

```
                       Table "public.thingie"
Column |         Type          |                 Modifiers
--------+-----------------------+---------------------------------------------
name   | character varying(50) |
id     | integer               | default nextval('UniqueIdentifier'::text)
```

If we manually insert some entries into this table they will have their `id` field set automagically.

```
INSERT INTO thingie (name) VALUES ('Fred');
INSERT INTO thingie (name) VALUES ('Wilma');
SELECT name, id FROM thingie;

 name  | id
-------+----
 Fred  |  0
 Wilma |  1
(2 rows)
```

4.6.4.3. Using OJB

The Database Repository Descriptor

The next step is to configure OJB to access our `thingie` table. We need to configure the corrct sequence manager in the `repository-database.xml`.

The default `repository-database.xml` uses the High/Low Sequence manager. We will delete or comment out that entry, and replace it with the `org.apache.ojb.broker.util.sequence.SequenceManagerNextValImpl` manager. This manager will pull the next value from a named sequence and use it. The entry for our sequence manager in the repository is:

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

This needs to be declared within the JDBC Connection descriptor, so an entire `repository-database.xml` might look like:

```
<jdbc-connection-descriptor
```

```
    jcd-alias="default"
    default-connection="true"
    platform="PostgreSQL"
    jdbc-level="2.0"
    driver="org.postgresql.Driver"
    protocol="jdbc"
    subprotocol="postgresql"
    dbalias="test"
    username="tester"
    password=""
    eager-release="false"
    batch-mode="false"
    useAutoCommit="1"
    ignoreAutoCommitExceptions="false"
    >

    <connection-pool
        maxActive="21"
        validationQuery=""/>

    <sequence-manager
        className="org.apache.ojb.broker.util.sequence.SequenceManagerNextValImpl" />
</jdbc-connection-descriptor>
```

Defining a Thingie Class

For the sake of simplicity we will make a very basic Java Thingie:

```
public class Thingie
{
     /** thingie(name) */
    private String name;

    /** thingie(id) */
    private int id;

    public String getName()           { return this.name; }
    public void setName(String name)  { this.name = name; }

    public int getId()                { return this.id; }
}
```

We also need a class descriptor in `repository-user.xml` that appears as follows:

```
<class-descriptor
    class="Thingie"
    table="THINGIE"
    >
    <field-descriptor
        name="id"
        column="ID"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        sequence-name="UniqueIdentifier"
        />
    <field-descriptor
        name="name"
        column="NAME"
        jdbc-type="VARCHAR"
        />
</class-descriptor>
```

Look over the `id` field descriptor carefully. The `autoincrement` and `sequence-name` attributes are important for getting our desired behavior. These tell OJB to use the sequence manager we defined to auto-increment the the value in `id`, and they also tell the sequence manager which database sequence to use - in this case `UniqueIdentifier`

We could allow OJB to create an extent-aware sequence and use it, however as we are working against a table that defaults to a specific named sequence, we want to make sure to pull values from that same sequence. Information on allowing OJB to create its own sequences is available in the Sequence Manager documentation.

Using Thingie

Just to demonstrate that this all works, here is a simple application that uses our Thingie.

```
import org.apache.ojb.broker.PersistenceBroker;
import org.apache.ojb.broker.PersistenceBrokerFactory;

public class ThingieDriver
{
    public static void main(String [] args)
    {
        PersistenceBroker broker = PersistenceBrokerFactory.defaultPersistenceBroker();

        Thingie thing = new Thingie();
        Thingie otherThing = new Thingie();

        thing.setName("Mabob");
        otherThing.setName("Majig");

        broker.beginTransaction();
        broker.store(thing);
        broker.store(otherThing);
        broker.commitTransaction();

        System.out.println(thing.getName() + " : " + thing.getId());
        System.out.println(otherThing.getName() + " : " + otherThing.getId());
        broker.close();
    }
}
```

When it is run, it will create two Thingie instances, store them in the database, and report on their assigned `id` values.

```
java -cp [...] ThingieDriver

Mabob : 2
Majig : 3
```

4.6.5. HOWTO - Work with LOB Data Types

4.6.5.1. Using Oracle LOB Data Types with OJB

Introduction

In a lot of applications there is a need to store large text or binary objects into the database. The definition of large usually means that the object's size is beyond the maximum length of a character field. In Oracle this means the objects to be stored can grow to > 4 KB each.

Depending on the application you are developing your "large objects" may either be in the range of some Kilobytes (for example when storing text-only E-Mails or regular XML documents), but they may also extend to several Megabytes (thinking of any binary data such as larger graphics, PDFs, videos, etc.).

In practice, the interface between your application and the database used for fetching and storing of your "large objects" needs to be different depending on the expected size. While it is probably perfectly acceptable to handle XML documents or E-Mails in memory as a string and always completely retrieve or store them in one chunk this will hardly be a good idea for video files for example.

This HOWTO will explain:

1. Why you would want to store large objects in the database
2. Oracle LARGE versus LOB data types
3. LOB handling in OJB using JDBC LOB types

This tutorial presumes you are familiar with the basics of OJB.

4.6.5.2. Backgrounder: Large objects in databases

This section is meant to fill in non-DBA people on some of the topics you need to understand when discussing large objects in databases.

Your database: The most expensive file system?

Depending on background some people tend to store anything in a database while others are biased against that. As databases use a file system for physical storage anyway, why would it make sense to store pictures, videos and the like as a large object in a database rather that just create a set of folders and store them right into the database.

When listening to Oracle's marketing campaingns one might get the impression that there is no need to have plain filesystems anymore and that they all will vanish and be replaced by Oracle database servers. If that happened this would definitely boast Oracle's revenues, but at the same time make IT cost in companies explode.

But there are applications where it in fact makes sense to have the database take care of unstructured data that you would otherwise just store in a file. The most common criteria for storing non-relational data in the database instead of storing it directly into the file system is whenever there is a strong link between this non-relatinal and some relational data.

Typical examples for that would be:

1. Pictures or videos of houses in a real estate agent's offer database
2. E-Mails related to a customer's order

If you are not storing these objects into the database you would need to create a link between the relational and the non-relational data by saving filenames in the database. This means that you application is responsible for managing this weak link in any respect. In order to make sure your application will be robust you need to make sure in your own code that

1. When creating a new record you create a valid and unique filename for storing the object.
2. When deleting a record you delete the corresponding file as well
3. When accessing the file referred to in the record you double-check the file is there and no locked

(There might be other, more subtle implications.)

All this is done for you by the database in case you choose to store your objects there. In addition to that, when discussing text data, a database might come with an option to automatically index the stored text documents for easy retrievel. This would allow you to perform an SQL seach such as "give me all customers that ever referred to the project foo in an e-mail". (In Oracle you need to install the InterMedia option, aka Oracle Text in order to get this extra functionality. Several vendors have also worked on technologies that allowed to seach rich content such as PDFs files, pictures or even sound or video stored in a database from SQL.)

Oracle LARGE versus LOB datatypes

Some people are worried about the efficiency of storing large objects in databases and the implications on performance. They are not necessarily entirely wrong in fearing that storing large objects in databases might be problematic the best or might require a lot of tweaks to parameters in order to be able to handle the large objects. It all depends on how a database implements storing large objects.

Oracle comes with two completely different mechanisms for that:

1. LARGE objects
2. LOB objects

When comparing the LARGE datatypes such as (*fixme*) to the LOB datatypes such as CLOB, BLOB, NCLOB (*fixme*) they don't read that different at first. But there is a huge difference in how they are handled both internally inside the database as well when storing and retrieving from the database client.

LARGE fields are embedded directly into the table row. This has some consequences you should be aware of:

1. If your record is made up of 5 VARCHAR fields with a maximum length of 40 bytes each and one LONGVARCHAR and you store 10 MB into the LONGVARCHAR column, your database row will extent to 10.000.200 bytes or roughly 10 MB.
2. The database always reads or writes the entire row. So if you do a SELECT on the VARCHAR fields in order to display their content in a user interface as a basis for the user to decide if he or she will need the content of the LONGVARCHAR at all the database will already have fetched all the 10 MB. If you SELECT and display 25 records each with a 10 MB object in it this will mean about 250 MB of I/O.
3. When storing or fetching such a row you need to make sure your fetch buffer is sized appropriately.

In practice this cannot be efficient. It might work as long as you stay in the KB range, but you will most likely run into trouble as soon as it gets into the MBs per record. Additionally, there are more limitations to the concept of LONG datatypes such as limiting the number of them you can have in one row and how you can index them. This is probably why Oracle decided to deprecate LONG datatypes in favor of LOB columns.

A lot of non-Oracle-DBA people believe that LOB means "large OBject" because some other vendors have used the term BLOB for "Binary Large OBject" in their products. This is not only wrong but - even worse - misleading, because people are asking: "What's the difference between large and long?" (Bear with all non native English speakers here, please!)

Instead, LOB stands for Locator OBject which exactly describes what is is. It is a pointer to the place where the actual data itself is stored. This locator will need only occupy some bytes in the row thus not harming row size at all. So all the issues discussed above vanish immediatelly. For the sake of simplicity think of a LOB as a pointer to a file on the databases internal file system that stores the actual content of that field. (Oracle might use plain files or different mechanisms in their implementation, we don't have to care.)

But as there is always a trade-off while LOBs are exstremely handy inside a row, they are more complex to store and retrieve. As opposed to all other column types their actual content stays where it is even if you transfer the row from the database to the client. All that goes over the wire in that case will be a token representing the actual LOB column content.

In order to read the content or to write LOB content it needs to open a separate stream connection over the network that can be read from or written to similar to a file on a network file system. JDBC (starting at version *fixme*) comes with special objects such as java.sql.Blob and java.sql.Clob to access the content of LOBs that do not represent character arrays or strings but streams!

4.6.5.3. Large Objects in OJB

After having skipped the above Backgrounder (in case you do Oracle administration for a living) of having read and understood it (hopefully applies to the rest of us) now that you've most likely decided to go for LOBs and forget about LONGs how is this handled with OJB?

Strategy 1: Using streams for LOB I/O

########## to be written #########

Strategy 2: Embedding OJB content in Java objects

########## to be written #########

Querying CLOB content

########## to be written #########

4.6.6. HOWTO - Use OJB in clustered environments

4.6.6.1. How to use OJB in clustered environments

Object/Relational Bridge will work fine in environments that require robustness features such as load-balancing, failover, and clustering. However, there are some important steps that you need to take in order for your data to be secure, and to prevent isolation failures. These steps are outlined below.

I have tested this in a number of environments, and with Servlet engines and J2EE environments. If you run into problems, please post questions to the OJB users mail list.

This outline for clustering is based on an email from the OJB Users Mail List: This is that mail.

4.6.6.2. Three steps to clustering your OJB application

A lot of people keep asking for robust ways to run their OJB engines in multi-VM or clustered environments. This email covers how to do that safely and securely using Open Symphony's OSCache caching product.

OSCache is a high-performance, robust caching product that supports clustering. I've been using it for a while in production and it is excellent.

Back to the Topic: There are three main things that you should do in order to make your OJB application ready for using a cache in a multi-VM or distributed environment.

First: Take care of the sequence manager

that you define within *jdbc-connection-descriptor* element in your repository.xml file. If none was set OJB use per default the `SequenceManagerHighLowImpl` sequence manager implementation.

> **Note:**
> As of Release Candidate 5 (rc5), you can use SequenceManagerHighLowImpl in distributed (non-managed) environments. The SequenceManagerHighLowImpl now supports its own optimistic locking that makes the implementation cluster aware by versioning an entry in the OJB_HL_SEQ table.

However, the SequenceManagerHighLowImpl has not been heavily tested in clustered environments, so if you want absolute security use an sequence manager implementation which delegates key generation to database.

If your database supports database based sequence key generation (like PostgreSQL, Oracle, ...) it's recommended to use `SequenceManagerNextValImpl` (supports database based sequence keys). Using this sequence manager will prevent conflicts in multi-vm or clustered environments. More about sequence manager here.

Handling sequence names

If you are using `SequenceManagerNextValImpl` you have two possibilities:

- Do it by your own:
  - Create a sequence object in your database.
    - An Oracle sequence lookslike: "create sequence ackSequence increment by 1 start with 1;"
    - A Postgres sequence looks like: "CREATE SEQUENCE ackSequence START 1";
  - For other databases you're on your own.
  - To tell OJB to use that sequence for your table add in your repository.xml the sequence name to the field-descriptor for your table's primary key field:

```
<field-descriptor
name="ackID"
column="ACKID"
jdbc-type="INTEGER"
```

```
primarykey="true"
autoincrement="true"
sequence-name="ackSequence"
/>
```

- Let OJB do that job for you:
  The `SequenceManagerNextValImpl` implementation create the sequence in database automatic if none was found. If you don't want to declare a `sequence-name` attribute in your *field-descriptor*, you can enable an automatic sequence name building by setting a specific *custom-attribute* , then `SequenceManagerNextValImpl` build an internal sequence name if none was found.

```
<sequence-manager className="org.apache.ojb.broker.util.sequence.SequenceManagerNextValImpl">
    <attribute attribute-name="autoNaming" attribute-value="true"/>
</sequence-manager>
```

More about sequence manager here.

Second: Enable optimistic locking

You need to secure the data at the database. Thomas Mahler (lead OJB developer and considerable ORM guru) recommended in one email that you use the Optimistic Locking feature that is provided by OJB when using the **PB API** in a clustered environment. Sounds good to me. To do this you need to do three small steps:

> **Note:**
> When using one of the top-level API in most cases Pessimistic (Object) Locking is supported. In that case it is recommended to use a *distributed lock management* instead of optimistic locking. More information about ODMG API and Locking here.

- Add a database column to your table that is either an INTEGER or a TIMESTAMP
- Add the field to your java class, and getter/setter methods (depends on the used PersistentField implementation):

```
private Integer ackOptLock;

public Integer getAckOptLock()
{
return ackOptLock;
}

public void setAckOptLock(Integer ackOptLock)
{
this.ackOptLock = ackOptLock;
}
```

- Add the column to your table in your repository:

```
    <field-descriptor
    name="ackOptLock"
    column="ACKOPTLOCK"
    jdbc-type="INTEGER"
    locking="true"/>
```

Now OJB will handle the locking for you. No explicit transactional code necessary!

Do The Cache

You're basically in good shape at this point. Now you've just got to set up OSCache to work with OJB. Here are the steps for that:

- Download OSCache from OSCache. Add the oscache-2.0.x.jar to your project so that it is in your classpath (for Servlet/J2EE users place in your WEB-INF/lib directory). You will also need commons-collections.jar and commons-logging.jar, if you don't already have them.
- Download JavaGroups from JavaGroups. Add the javagroups-all.jar to your classpath (for Servlet/J2EE users place in your WEB-INF/lib directory).

- In your OJB.properties file change the ObjectCacheClass property to be the following:
  `ObjectCacheClass=org.nacse.jlib.ObjectCacheOSCacheImpl`
  To make OSCache the default used cache implementation. More info about [object caching here](#).
- Add oscache.properties from your OSCache distribution to your project so that it is in the classpath (for Servlet/J2EE users place in your WEB-INF/classes directory). Open the file and make the following changes:
  1. Add the following line to the CACHE LISTENERS section of your oscache.properties file:
     `cache.event.listeners=com.opensymphony.oscache.plugins.clustersupport.JavaGroups`
  2. Add the following line at the end of the oscache.properties file (your network must support multicast):
     `cache.cluster.multicast.ip=231.12.21.132`
- Add the following class to your project (feel free to change package name, but make sure that you specify the full qualified class name in OJB.properties file). You can find source of this class under
  `db-ojb/contrib/src/ObjectCacheOSCacheImpl`.
  Source for `ObjectCacheOSCacheImpl`:

```
public class ObjectCacheOSCacheImpl implements ObjectCache
{
    private static GeneralCacheAdministrator admin = new GeneralCacheAdministrator();
    private static final int NO_REFRESH = CacheEntry.INDEFINITE_EXPIRY;

    public ObjectCacheOSCacheImpl()
    {
    }

    public ObjectCacheOSCacheImpl(PersistenceBroker broker, Properties prop)
    {
    }

    public void cache(Identity oid, Object obj)
    {
        try
        {
            this.remove(oid);
            admin.putInCache(oid.toString(), obj);
        }
        catch(Exception e)
        {
            throw new RuntimeCacheException(e.getMessage());
        }
    }

    public Object lookup(Identity oid)
    {
        try
        {
            return admin.getFromCache(oid.toString(), NO_REFRESH);
        }
        catch(Exception e)
        {
            admin.cancelUpdate(oid.toString());
            return null;
        }
    }

    public void remove(Identity oid)
    {
        try
        {
            admin.flushEntry(oid.toString());
        }
        catch(Exception e)
        {
            throw new RuntimeCacheException(e.getMessage());
        }
    }
    public void clear()
    {
        if(admin != null)
        {
            try
            {
                admin.flushAll();
            }
            catch(Exception e)
            {
                throw new RuntimeCacheException(e);
            }
        }
    }
}
```

You're ready to go! Now just create two instances of your application and see how they communicate at the cache level. Very cool.

4.6.6.3. Notes

- For J2EE/Servlet users: I have tested this on a number of different application servers. If you're having problems with your engine, post an email to the OJB Users mail list.
- OSCache also supports JMS for clustering here, which I haven't covered. If you either don't have access to a multicast network, or just plain like JMS, please refer to the OSCache documentation for help with that, see OSCache Clustering with JMS).
- I have also tested this with Tangosol Coherence. Please refer to this Blog entry for that setup: Coherence Setup
- OJB also has ships with JCS. Feel free to try that one out on your own.

4.6.7. HOWTO - Stored Procedure Support

4.6.7.1. Introduction

OJB supports the use of stored procedures to handle the basic DML operations (INSERT, UPDATE, and DELETE). This document will describe the entries that you'll need to add to your repository in order to get OJB to utilize stored procedures instead of 'traditional' INSERT, UPDATE or DELETE statements.

Please note that there will be references to 'stored procedures' throughout this document. However, this is just a simplification for the purposes of this document. Any place you see a reference to 'stored procedure', you can assume that either a stored procedure or function can be used.

Information presented in this document includes the following:

- Basic repository entries
- Common attributes for all procedure descriptors
- An overview of the insert procedure, update procedure and delete procedure descriptors.
- Information about the argument descriptors that are supported for all procedure
- A simple example and a more complex example

4.6.7.2. Repository entries

For any persistable class (i.e. "com.myproject.Customer") where you want to utilize stored procedures to handle persistence operations instead of traditional DML statements (i.e. INSERT, UPDATE or DELETE), you will need to include one or more of the following descriptors within the corresponding class-descriptor for the persistable class:

- `insert-procedure` - identifies the stored procedure that is to be used whenever a class needs to be inserted into the database.
- `update-procedure` - identifies the stored procedure that is to be used whenever a class needs to be updated in the database.

- `delete-procedure` - identifies the stored procedure that is to be used whenever a class needs to be removed from the database.

All of these descriptors must be nested within the class-descriptor that they apply to. Here is an example of a simple class-descriptor that includes each of the procedure descriptors listed above:

```
<class-descriptor  class="com.myproject.Customer" table="CUSTOMER">
<field-descriptor column="ID" jdbc-type="DECIMAL" name="id" primarykey="true"/>
<field-descriptor column="NAME" jdbc-type="VARCHAR" name="name"/>
<insert-procedure name="CUSTOMER_PKG.ADD">
    <runtime-argument field-ref="id" return="true"/>
    <runtime-argument field-ref="name"/>
</insert-procedure>
<update-procedure name="CUSTOMER_PKG.CHG">
    <runtime-argument field-ref="id"/>
    <runtime-argument field-ref="name"/>
</update-procedure>
<delete-procedure name="CUSTOMER_PKG.CHG">
    <runtime-argument field-ref="id"/>
</delete-procedure>
</class-descriptor>
```

4.6.7.3. Common attributes

All three procedure descriptors have the following attributes in common:

- `name` - This is the name of the stored procedure that is to be used to handle the specific persistence operation.
- `return-field-ref` - This identifies the field in the class where the return value from the stored procedure will be stored. If this attribute is blank or not specified, then OJB will assume that the stored procedure does not return a value and will format the SQL command accordingly.

The basic syntax that is used to call a procedure that has a return value looks something like this:

```
{?= call &lt;procedure-name&gt;[&lt;arg1&gt;,&lt;arg2&gt;, ...]}
```

The basic syntax that is used to call a procedure that **does not** include a return value looks something like this:

```
{call &lt;procedure-name&gt;[&lt;arg1&gt;,&lt;arg2&gt;, ...]}
```

When OJB assembles the SQL to call a stored procedure, it will use the value of the 'name' attribute in place of 'procedure-name' in these two examples.

In addition, if the procedure descriptor includes a value in the 'return-field-ref' attribute that is 'valid', then the syntax that OJB builds will include the placeholder for the result parameter.

The previous section referred to the idea of a 'valid' value in the 'return-field-ref' attribute. A value is considered to be 'valid' if it meets the following criteria:

- The value is not blank
- There is a field-descriptor with a 'name' that matches the value in the 'return-field-ref' attribute.

If the 'return-field-ref' attribute is not 'valid', then the placeholder for the result parameter will not be included in the SQL that OJB assembles.

4.6.7.4. insert-procedure

The insert-procedure descriptor identifies the stored procedure that should be used whenever a class needs to be inserted into the database. In addition to the common attributes listed earlier, the insert-procedure includes the following attribute:

- include-all-fields

This attribute provides an efficient mechanism for passing all attributes of a persistable class to a stored procedure. If this attribute is set to true, then OJB will ignore any nested argument descriptors. Instead, OJB will assume that the argument list for the stored procedure includes arguments for all attributes of the persistable class and that those arguments appear in the same order as the field-descriptors for the persistable class.

The default value for this attribute is 'false'.

> **Note:**
> If the field-descriptors in your repository do not 'align' exactly with the argument list for the stored procedure, or you want to maintain explicit control over the values that are passed to the stored procedure, then either set the 'include-all-fields' attribute to 'false' or leave it off the insert-procedure descriptor.

4.6.7.5. update-procedure

The update-procedure descriptor identifies the stored procedure that should be used whenever a class needs to be updated in the database. In addition to the common attributes listed earlier, the update-procedure includes the following attribute:

- include-all-fields

This attribute provides the same capabilities and has the same caveats as the include-all-fields attribute on the insert-procedure descriptor.

4.6.7.6. delete-procedure

The delete-procedure descriptor identifies the stored procedure that should be used whenever a class needs to be deleted from the database. In addition to the common attributes listed earlier, the delete-procedure includes the following attribute:

- include-pk-only
  This attribute provides an efficient mechanism for passing all of the attributes that make up the primary key for a persistable class to the specified stored procedure. If this attribute is set to true, then OJB will ignore any nested argument descriptors. Instead, OJB will assume that the argument list for the stored procedure includes arguments for all attributes that make up the primary key for the persistable class (i.e. those field-descriptors where the 'primary-key' attribute is set to 'true'). OJB will also assume that those arguments appear in the same order as the corresponding field-descriptors for the persistable class.
  The default value for this attribute is 'false'.

> **Note:**
> If the field-descriptors in your repository that make up the primary key for a persistable class do not 'align' exactly with the argument list for the stored procedure, or you want to maintain explicit control over the values that are passed to the stored procedure, then either set the 'include-pk-only' attribute to 'false' or leave it off the delete-procedure descriptor.

4.6.7.7. Argument descriptors

Argument descriptors are the mechanism that you will use to tell OJB two things:

1. How many placeholders should be included in the argument list for a stored procedure?
2. What value should be passed for each of those arguments?

There are two types of argument descriptors that can be defined in the repository:

- runtime arguments used to set a stored procedure argument equal to a value that is only known at runtime.
- constant arguments used to set a stored procedure argument equal to constant value.

You may notice that there is no argument descriptor specifically designed to pass a *null* value to the procedure. This capability is provided by the runtime argument descriptor.

The argument descriptors are essentially the 'mappings' between stored procedure arguments and their runtime values. Each procedure descriptor can include 0 or more argument descriptors in it's definition.

After reading that last comment, you may wonder why OJB allows you to configure a procedure descriptor with no argument descriptors since the primary focus of OJB is to handle object persistence. How could OJB perform any sort persistence operation using a stored procedure that did not involve the passage of at least one value to the stored procedure? To be honest, it is extremely unlikely that you would ever set up a procedure descriptor with no argument descriptors. However, since there is no minimum number of arguments required for a stored procedure, we did not want to implement within OJB a requirement on the number of arguments that was more restrictive than the limits imposed by most/all database systems.

runtime-argument descriptors

A runtime-argument descriptor is used to set a stored procedure argument equal to a value that is only known at runtime.

Two attributes can be specified for each runtime-argument descriptor:

- field-ref
  The 'field-ref' attribute identifies the specific field descriptor that will provide the argument's value. If this attribute is not specified or does not resolve to a valid field-descriptor, then a null value will be passed to the stored procedure.
- return
  The 'return' attribute is used to determine if the argument is used by the stored procedure as an 'output' argument.
  If this attribute is set to true, then the corresponding argument will be registered as an output parameter. After execution of the stored procedure, the value of the argument will be 'harvested' from the CallableStatement and stored in the attribute identified by the field-ref attribute.
  If this attribute is not specified or set to false, then OJB assumes that the argument is simply an 'input' argument, and it will do nothing special to the argument.

constant-argument descriptors

A constant-argument descriptor is used to set a stored procedure argument equal to constant value.

There is one attribute that can be specified for the constant-argument descriptor:

- value
  The 'value' attribute identifies the value for the argument.

4.6.7.8. A simple example

This section provides background information and a simple example that illustrates how OJB's support for stored procedures can be utilized.

The background information covers the following topics:

- The basic requirements
- The database objects including the table that will be manipulated, the sequence that will be used by the stored procedures to assign primary key falues, the insert and update triggers that maintain the four 'audit' columns and the package that provides the stored procedures that will handle the persistence operations.

Click here to skip the background information and go straight to the implementation.

The basic requirements

These are the requirements that must be satisfied by our example

1. All insert, update and delete operations are to be performed by stored procedures.
2. All primary key values are to be by the stored procedure that handles the insert operation. The value that is assigned should be reflected in the object that 'triggered' the insert operation.
3. For auditing purposes, all tables will include the following set of columns:
   - `USER_CREATED` - This will contain the id of the user who created the record
   - `DATE_CREATED` - The date on which the record was created created
   - `USER_UPDATED` - The id of the user who last modified the record

- `USER_UPDATED` - The date on which the record was last modified

In addition to the inclusion of these columns on each table, the following requirements related to these columns had to be supported:

1. The values of the two date-related audit columns were to be maintained at the database level via insert and update triggers.
    - The insert trigger will set both `DATE_CREATED` and `DATE_UPDATED` to the current system date.
    - The update trigger will set `DATE_UPDATED` to the current system date. The update trigger will also ensure that the original value of `DATE_CREATED` is never modified.
2. The values of the two user-related audit columns are to be maintained at the database level via insert and update triggers.
    - The insert and update triggers will ensure that `USER_CREATED` and `USER_UPDATED` are appropriately populated.
    - The update trigger will ensure that the original value of `USER_CREATED` is never modified.
3. Any changes that are made by the insert or update triggers to any of the four 'audit' columns had to be reflected in the object that caused the insert or update operation to occur.

The database objects

The database objects that are described in this section utilize Oracle specific syntax. However, you should not infer from this that the stored procedure support provided by OJB can only be used to access data that is stored in an Oracle database. In reality, stored procedures can be used for persistence operations in any database that supports stored procedures.

- The table that will be manipulated,
- The sequence that will be used by the stored procedures to assign primary key values
- The insert and update triggers that maintain the four 'audit' columns
- The package that provides the stored procedures that will handle the persistence operations.

Click here to skip the information about the database objects and go straight to the implementation.

The CUSTOMER table

This example will deal exclusively with persistence operations related to the a table named 'CUSTOMER' that is built using the following DDL:

```
CREATE TABLE CUSTOMER
( ID NUMBER(18) NOT NULL
, NAME VARCHAR2(50) NOT NULL
, USER_CREATED VARCHAR2(30)
, DATE_CREATED DATE
, USER_UPDATED VARCHAR2(30)
, DATE_UPDATED DATE
, CONSTRAINT PK_CUSTOMER PRIMARY KEY (ID)
);
```

The sequence

This sequence will be used to assign unique values to `CUSTOMER.ID`.

```
              CREATE SEQUENCE CUSTOMER_SEQ;
```

The insert and update triggers

These two triggers will implement all of the requirements listed above that are related to the four audit columns:

```
CREATE OR REPLACE TRIGGER CUSTOMER_ITR
BEFORE INSERT ON CUSTOMER
FOR EACH ROW
BEGIN
--
-- Populate the audit dates
```

```
--
SELECT SYSDATE, SYSDATE
INTO :NEW.DATE_CREATED, :NEW.DATE_UPDATED
FROM DUAL;
--
-- Make sure the user created column is populated.
--
IF :NEW.USER_CREATED IS NULL
THEN
SELECT SYS_CONTEXT('USERENV','TERMINAL')
  INTO :NEW.USER_CREATED
  FROM DUAL;
END IF;
--
-- Make sure the user updated column is populated.
--
IF :NEW.USER_UPDATED IS NULL
THEN
SELECT SYS_CONTEXT('USERENV','TERMINAL')
  INTO :NEW.USER_UPDATED
  FROM DUAL;
END IF;
END;
/

CREATE OR REPLACE TRIGGER CUSTOMER_UTR
BEFORE UPDATE ON CUSTOMER
FOR EACH ROW
BEGIN
--
-- Populate the date updated
--
SELECT SYSDATE
INTO :NEW.DATE_UPDATED
FROM DUAL;
--
-- Make sure the user updated column is populated.
--
IF :NEW.USER_UPDATED IS NULL
THEN
SELECT SYS_CONTEXT('USERENV','TERMINAL')
  INTO :NEW.USER_UPDATED
  FROM DUAL;
END IF;
--
-- Make sure the date/user created are never changed
--
SELECT :OLD.DATE_CREATED, :OLD.USER_CREATED
INTO :NEW.DATE_CREATED, :NEW.USER_CREATED
FROM DUAL;
END;
/
```

The package

This Oracle package will handle all INSERT, UPDATE and DELETE operations involving the `CUSTOMER` table.

```

CREATE OR REPLACE PACKAGE CUSTOMER_PKG AS
--
-- This procedure should be used to add a record to the CUSTOMER table.
--
PROCEDURE ADD ( AID           IN OUT CUSTOMER.ID%TYPE
            , ANAME         IN     CUSTOMER.NAME%TYPE
            , AUSER_CREATED IN OUT CUSTOMER.USER_CREATED%TYPE
            , ADATE_CREATED IN OUT CUSTOMER.DATE_CREATED%TYPE
            , AUSER_UPDATED IN OUT CUSTOMER.USER_UPDATED%TYPE
            , ADATE_UPDATED IN OUT CUSTOMER.DATE_UPDATED%TYPE );
--
```

```
-- This procedure should be used to change a record on the CUSTOMER table.
--
PROCEDURE CHANGE ( AID           IN     CUSTOMER.ID%TYPE
               , ANAME         IN     CUSTOMER.NAME%TYPE
               , AUSER_CREATED IN OUT CUSTOMER.USER_CREATED%TYPE
               , ADATE_CREATED IN OUT CUSTOMER.DATE_CREATED%TYPE
               , AUSER_UPDATED IN OUT CUSTOMER.USER_UPDATED%TYPE
               , ADATE_UPDATED IN OUT CUSTOMER.DATE_UPDATED%TYPE );
--
-- This procedure should be used to delete a record from the CUSTOMER table.
--
PROCEDURE DELETE ( AID IN CUSTOMER.ID%TYPE );
END CUSTOMER_PKG;
/
CREATE OR REPLACE PACKAGE BODY CUSTOMER_PKG AS
--
-- This procedure should be used to add a record to the CUSTOMER table.
--
PROCEDURE ADD ( AID           IN OUT CUSTOMER.ID%TYPE
            , ANAME         IN     CUSTOMER.NAME%TYPE
            , AUSER_CREATED IN OUT CUSTOMER.USER_CREATED%TYPE
            , ADATE_CREATED IN OUT CUSTOMER.DATE_CREATED%TYPE
            , AUSER_UPDATED IN OUT CUSTOMER.USER_UPDATED%TYPE
            , ADATE_UPDATED IN OUT CUSTOMER.DATE_UPDATED%TYPE )
IS
  NEW_SEQUENCE_1 CUSTOMER.ID%TYPE;
BEGIN
  SELECT CUSTOMER_SEQ.NEXTVAL
    INTO NEW_SEQUENCE_1
    FROM DUAL;
  INSERT INTO CUSTOMER ( ID, NAME, USER_CREATED, USER_UPDATED )
     VALUES ( NEW_SEQUENCE_1, ANAME, AUSER_CREATED, AUSER_UPDATED )
    RETURNING ID, USER_CREATED, DATE_CREATED, USER_UPDATED, DATE_UPDATED
         INTO AID, AUSER_CREATED, ADATE_CREATED, AUSER_UPDATED, ADATE_UPDATED;
END ADD;
--
-- This procedure should be used to change a record on the CUSTOMER table.
--
PROCEDURE CHANGE ( AID           IN     CUSTOMER.ID%TYPE
               , ANAME         IN     CUSTOMER.NAME%TYPE
               , AUSER_CREATED IN OUT CUSTOMER.USER_CREATED%TYPE
               , ADATE_CREATED IN OUT CUSTOMER.DATE_CREATED%TYPE
               , AUSER_UPDATED IN OUT CUSTOMER.USER_UPDATED%TYPE
               , ADATE_UPDATED IN OUT CUSTOMER.DATE_UPDATED%TYPE )
IS
BEGIN
  UPDATE CUSTOMER
     SET NAME         = ANAME
       , USER_CREATED = USER_CREATED
       , USER_UPDATED = AUSER_UPDATED
   WHERE ID           = AID
    RETURNING USER_CREATED, DATE_CREATED, USER_UPDATED, DATE_UPDATED
         INTO AUSER_CREATED, ADATE_CREATED, AUSER_UPDATED, ADATE_UPDATED;
END CHANGE;
--
-- This procedure should be used to delete a record from the CUSTOMER table.
--
PROCEDURE DELETE ( AID IN CUSTOMER.ID%TYPE )
IS
BEGIN
  DELETE
    FROM CUSTOMER
   WHERE ID    = AID;
END DELETE;
END CUSTOMER_PKG;
/
```

Please note the following about the structure of the `CUSTOMER_PKG` package:

- The `AID` argument that is passed to the the `ADD` procedure is defined as `IN OUT`. This allows the procedure to return the newly assigned `ID` to the caller.
- In the `ADD` and `CHANGE` procedures, the arguments that correspond to the four 'audit' columns are defined as `IN OUT`. This allows the procedure to return the current value of these columns to the 'caller'.

The implementation

Getting OJB to utilize the stored procedures described earlier in this document is as simple as adding a few descriptors to the repository. Here is a class-descriptor related to the `CUSTOMER` table that includes all of the necessary descriptors.

```
<class-descriptor class="com.myproject.Customer" table="CUSTOMER">
    <field-descriptor column="ID" jdbc-type="DECIMAL" name="id" primarykey="true"/>
    <field-descriptor column="NAME" jdbc-type="VARCHAR" name="name"/>
    <field-descriptor column="USER_CREATED" jdbc-type="VARCHAR" name="userCreated"/>
    <field-descriptor column="DATE_CREATED" jdbc-type="TIMESTAMP" name="dateCreated"/>
    <field-descriptor column="USER_UPDATED" jdbc-type="VARCHAR" name="userUpdated"/>
    <field-descriptor column="DATE_UPDATED" jdbc-type="TIMESTAMP" name="dateUpdated"/>
    <insert-procedure name="CUSTOMER_PKG.ADD">
      <runtime-argument field-ref="id" return="true"/>
      <runtime-argument field-ref="name"/>
      <runtime-argument field-ref="userCreated" return="true"/>
      <runtime-argument field-ref="dateCreated" return="true"/>
      <runtime-argument field-ref="userUpdated" return="true"/>
      <runtime-argument field-ref="dateUpdated" return="true"/>
    </insert-procedure>
    <update-procedure name="CUSTOMER_PKG.CHG">
      <runtime-argument field-ref="id"/>
      <runtime-argument field-ref="name"/>
      <runtime-argument field-ref="userCreated" return="true"/>
      <runtime-argument field-ref="dateCreated" return="true"/>
      <runtime-argument field-ref="userUpdated" return="true"/>
      <runtime-argument field-ref="dateUpdated" return="true"/>
    </update-procedure>
    <delete-procedure name="CUSTOMER_PKG.CHG">
      <runtime-argument field-ref="id"/>
    </delete-procedure>
</class-descriptor>
```

Some things to note about this class-descriptor:

1. In the insert-procedure descriptor, the first runtime-argument descriptor correspnds to the "AID" argument that is passed to the CUSTOMER_PKG.ADD routine. The "return" attribute on this runtime-argument is set to "true". With this configuration, OJB will 'harvest' the value that is returned by the CUSTOMER_PKG.ADD stored procedure and store the value in the "id" attribute on the com.myproject.Customer class.
2. In both the insert-procedure and update-procedure descriptors, the runtime-argument descriptors that correspond to the four 'audit' columns all have the "return" argument set to "true". This allows any updates that are made by the procedure or the insert/update triggers to be reflected in the "Customer" object that caused the insert/update operation to occur.

4.6.7.9. A complex example

This example builds upon the simple example that was presented earlier by introducing some additional requirements beyond those that were specified in the simple example. Some of these additional requirements may seem a little contrived. To be honest, they are. The only purpose of these additional requirements is to create situations that illustrate how the additional capabilities provided by OJB's support for stored procedures can be utilized.

The additional requirements for this example include the following:

- All procedures will include two additional arguments. These two new arguments will be added to the end of the argument list for all existing procedures.
  - `ASOURCE_SYSTEM` - identifies the system that initiated the persistence operation. This will provide a higher level of audit tracking capability. In our example, this will always be "SAMPLE".
  - `ACOST_CENTER` - identifies the 'cost center' that should be charged for the persistence operation. In our example, this

argument will always be null.

- For all "ADD" and "CHG" stored procedures, the value that was assigned to the "DATE_UPDATED" column will no longer be returned to the caller via an "IN OUT" argument. Instead, it will be returend to the caller via the procedure's return value.

Based on these new requirements, the class-descriptor for the "com.myproject.Customer" class will look like this. The specific changes are detailed below.

```
<class-descriptor class="com.myproject.Customer" table="CUSTOMER">
    <field-descriptor column="ID" jdbc-type="DECIMAL" name="id" primarykey="true"/>
    <field-descriptor column="NAME" jdbc-type="VARCHAR" name="name"/>
    <field-descriptor column="USER_CREATED" jdbc-type="VARCHAR" name="userCreated"/>
    <field-descriptor column="DATE_CREATED" jdbc-type="TIMESTAMP" name="dateCreated"/>
    <field-descriptor column="USER_UPDATED" jdbc-type="VARCHAR" name="userUpdated"/>
    <field-descriptor column="DATE_UPDATED" jdbc-type="TIMESTAMP" name="dateUpdated"/>
    <insert-procedure name="CUSTOMER_PKG.ADD"
          return-field-ref="dateUpdated"> <!-- See note 1 -->
      <runtime-argument field-ref="id" return="true"/>
      <runtime-argument field-ref="name"/>
      <runtime-argument field-ref="userCreated" return="true"/>
      <runtime-argument field-ref="dateCreated" return="true"/>
      <runtime-argument field-ref="userUpdated" return="true"/>
      <runtime-argument field-ref="dateUpdated"/> <!-- See note 2 -->
      <constant-argument value="SAMPLE"/> <!-- See note 3 -->
      <runtime-argument/> <!-- See note 4 -->
    </insert-procedure>
    <update-procedure name="CUSTOMER_PKG.CHG"
          return-field-ref="dateUpdated"> <!-- See note 1 -->
      <runtime-argument field-ref="id"/>
      <runtime-argument field-ref="name"/>
      <runtime-argument field-ref="userCreated" return="true"/>
      <runtime-argument field-ref="dateCreated" return="true"/>
      <runtime-argument field-ref="userUpdated" return="true"/>
      <runtime-argument field-ref="dateUpdated"/> <!-- See note 2 -->
      <constant-argument value="SAMPLE"/> <!-- See note 3 -->
      <runtime-argument/> <!-- See note 4 -->
    </update-procedure>
    <delete-procedure name="CUSTOMER_PKG.CHG">
      <runtime-argument field-ref="id"/>
      <constant-argument value="SAMPLE"/> <!-- See note 3 -->
      <runtime-argument/> <!-- See note 4 -->
    </delete-procedure>
</class-descriptor>
```

Here are an explanation of each modification:

- **Note 1:** The value that is returned by the "ADD" and "CHG" stored procedures will now be stored in the "dateUpdated" attribute on the "com.myproject.Customer" class.
- **Note 2:** Since the ADATE_UPDATED argument is no longer defined as an "IN OUT" argument, we have removed the "return" attribute from the corresponding runtime-argument descriptor.
- **Note 3:** This is the first of two new arguments that were added to the argument list of each procedure. This argument represents the 'source system', the system that initiated the persistence operation. In our example, we will always pass a value of 'SAMPLE'.
- **Note 4:** This is the second of two new arguments that were added to the argument list of each procedure. This argument represents the 'cost center' that should be charged for the persistence operation. In our example, we have no cost center, so we need to pass a null value. This is accomplished by including a 'runtime-argument' descriptor that has no 'field-ref' specified.

4.7. Testing

4.7.1. Testing Summary

4.7.1.1. Testing

Here can be found a summary of all (maybe nearly all) documentation about how OJB does testing (a JUnit baseed test suite) and how to write new tests.

- The OJB test suite
- How to write tests

4.7.2. OJB JUnit Test Suite

4.7.2.1. Introduction

Building an Object/Relational mapping tool with support for multiple API's is really error prone. To create a solid and stable software, the most awful thing in programmers life has to be done - Testing.

Quality assurance taken seriously! OJB and provide specific tests for each supported API. Currently more than 600 test cases for regression tests exist. As testing framework JUnit was used.

4.7.2.2. How to run the Test Suite

If the platform depended settings are done, the test suite can be started with the ant target:

```
ant junit
```

If compiling of the sources should be skipped use

```
ant junit-no-compile
```

If you did not manage to set up the target database with the `ant prepare-testdb` you can use

```
ant junit-no-compile-no-prepare
```

to run the testsuite without generation of the test database (and without compiling).

After running the regression tests you should see a console output as follows:

```
junit-no-compile-no-prepare:
[junit] Running org.apache.ojb.broker.AllTests
[junit] Tests run: 382, Failures: 0, Errors: 0, Time elapsed: 50,843 sec

[junit] Running org.apache.ojb.odmg.AllTests
[junit] Tests run: 193, Failures: 0, Errors: 0, Time elapsed: 16,243 sec

[junit] Running org.apache.ojb.soda.AllTests
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 8,392 sec

[junit] Running org.apache.ojb.otm.AllTests
[junit] Tests run: 79, Failures: 0, Errors: 0, Time elapsed: 21,871 sec
junit-no-compile:
junit:
BUILD SUCCESSFUL
Total time: 3 minutes 58 seconds
```

We aim at shipping that releases have no failures and errors in the regression tests! If the Junit tests report errors or failures something does not work properly! There may be several reasons:

- You made a mistake in configuration (OJB was shipped with settings pass all tests). See platform, OJB.properties, repository file, .
- Your database doesn't support specific features used by the tests
- Evil hex
- Bug in OJB

JUnit writes a *log-file* for each tested API. You can find the logs under `[db-ojb]/target/test`. The log files named like `tests-XXX.txt`. The test logs show in detail what's going wrong.

In such a case please check again if you followed all the above steps. If you still have problems you might post a request to the OJB user mailinglist.

4.7.2.3. What about known issues?

All major known issues are listed in the release-notes file.
The tests reproduce open bugs will be skipped on released OJB versions. It is possible to enable these tests to see all failing test cases of the shipped version by changing a flag in `[db-ojb]/build.properties` file:

```
###
# If 'true', junit tests marked as known issue in the junit-test
# source code (see OJBTestCase class for more detailed info) will be
# skipped. Default value is 'true'. For development 'false' is recommended,
# because this will show unsolved problems.
OJB.skip.issues=true
```

4.7.2.4. Donate own tests for OJB Test Suite

Details about donate own test to OJB you can find here.

4.7.3. OJB - Write Tests

4.7.3.1. Introduction

As described in *test suite* section OJB emphasise quality assurance and provide a huge test suite. But it is impossible to cover all parts of OJB with tests and OJB will never be perfect (of course it's nearly perfect ;-)), thus if you miss a test or found an bug don't hesitate, write your own test and send it to the lists or attach it in the bug report.

4.7.3.2. How to write a new Test

Before start writing your own test case please pay attention of these rules.

The Test Class

All test classes have to inherit from `org.apache.ojb.junit.OJBTestCase` and have to provide a static main method to start the Junit test:

```
public class MyTest extends OJBTestCase
{
    public static void main(String[] args)
    {
        String[] arr = {MyTest.class.getName()};
        junit.textui.TestRunner.main(arr);
    }

    public void testMyFirstOne()
    {
        ....
    {
```

In package `org.apache.ojb.junit` can be found some test classes for specifc circumstances:

- `org.apache.ojb.junit.PBTestCase` - Provide a public `org.apache.ojb.broker.PersistenceBroker` field.
- `org.apache.ojb.junit.ODMGTestCase` - Provide public `org.odmg.Implementation` and `org.odmg.Database` fields.

- `org.apache.ojb.junit.JUnitExtensions` - Provide base classes for write multithreaded test classes. More info see javadoc comment of this class.

A test case for the PB-API may look like:

```
public class ReferenceRuntimeSettingTest extends PBTestCase
{
    public static void main(String[] args)
    {
        String[] arr = {ReferenceRuntimeSettingTest.class.getName()};
        junit.textui.TestRunner.main(arr);
    }

    public void testChangeReferenceSetting()
    {
        ClassDescriptor cld = broker.getClassDescriptor(MainObject.class);
        // and so on
        ....
    }
```

The PersistenceBroker cleanup is done by *PBTestCase*.

Persistent Objects used by Test

We recommend to introduce separate persistent objects for each TestCase class. In test suite two concepts are used:

- Include your persistent objects as *public static classes* in your test class.
- Separate your test class in an independent package and include the test case and all persistent object classes in this new package.

Test Class Metadata

Currently all test specific object metadata (class-descriptor used for tests) are shared among several xml files. The naming convention is `repository_junit_XXX.xml`. Thus metadata for new tests should be included in one of the existing junit repository (sub) files or writen in an new separate one and included in repository main file.

```
<!DOCTYPE descriptor-repository PUBLIC
       "-//Apache Software Foundation//DTD OJB Repository//EN"
       "repository.dtd"
[
<!ENTITY database SYSTEM "repository_database.xml">
<!ENTITY internal SYSTEM "repository_internal.xml">
<!ENTITY user SYSTEM "repository_user.xml">

<!-- here the junit include files begin  -->
<!ENTITY junit SYSTEM "repository_junit.xml">
<!ENTITY junit_odmg SYSTEM "repository_junit_odmg.xml">
<!ENTITY junit_otm SYSTEM "repository_junit_otm.xml">
<!ENTITY junit_ref SYSTEM "repository_junit_reference.xml">
<!ENTITY junit_meta_seq SYSTEM "repository_junit_meta_seq.xml">
<!ENTITY junit_model SYSTEM "repository_junit_model.xml">
<!ENTITY junit_cloneable SYSTEM "repository_junit_cloneable.xml">

<!ENTITY junit_myfirsttest SYSTEM "repository_junit_myfirsttest.xml">
]>
<descriptor-repository version="1.0" isolation-level="read-uncommitted"
        proxy-prefetching-limit="50">

    <!-- include all used database connections -->
    &database;

    <!-- include ojb internal mappings here -->
    &internal;

    <!-- include user defined mappings here -->
    &user;

    <!-- include mappings for JUnit tests -->
    <!-- This could be removed (with <!ENTITY entry),
         if junit test suite was not used
    -->
    &junit;
    &junit_odmg;
    &junit_otm;
    &junit_ref;
    &junit_meta_seq;
    &junit_model;
    &junit_cloneable;

    &junit_myfirsttest;
```

5. All