



Building a Data Persistence Tier

Leveraging Object Relational Bridge (ORB)

NetChange, LLC

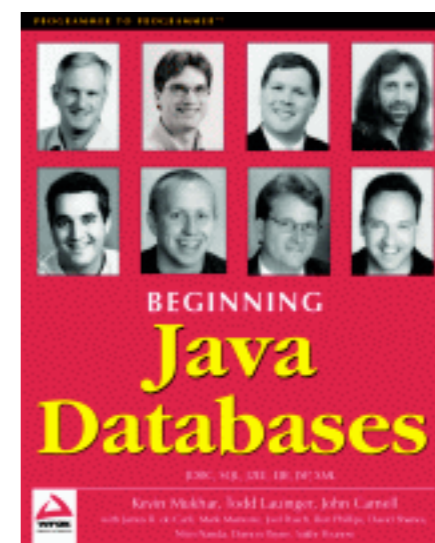
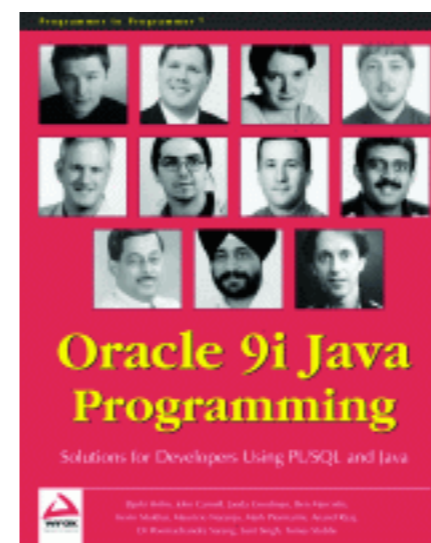
People•Ideas•Technology

About the Speaker



John Carnell is a Principal Architect for NetChange, LLC. NetChange is an IT consulting firm specializing in enterprise application design and implementation. John is also an adjunct faculty member of Waukesha County Technical College (WCTC) School of Business.

In addition to his consulting and teaching activities, John is a prolific speaker and writer. He has spoken at such national conferences as Internet Expo and the Data Warehousing Institute. John just completed a speaking tour with the Denver-based Complete Programmer Network. John has authored, co-authored and been a technical reviewer for a number of technology books and industry publications. Some of his works include:



John's current writing project is a second edition of his Struts book. This book is scheduled for release by APress in early fall.

Agenda

- ◆ The Challenges of Java Database Development
- ◆ What are Object/Relational (O/R) Mapping Tools?
- ◆ What is Object Relational Bridge?
- ◆ Installing and configuring Object Relational Bridge
- ◆ Implementing Mapping with Object Relational Bridge
- ◆ Proxies for Performance
- ◆ Using J2EE Design Patterns to build a Data Access Tier



The Challenges of Java Database Development

Relational Databases

- ◆ For many of us, the majority of our development lives are spent writing code that retrieves and manipulates data from relational databases.
- ◆ Pretty depressing
- ◆ Are we caught in the Matrix?

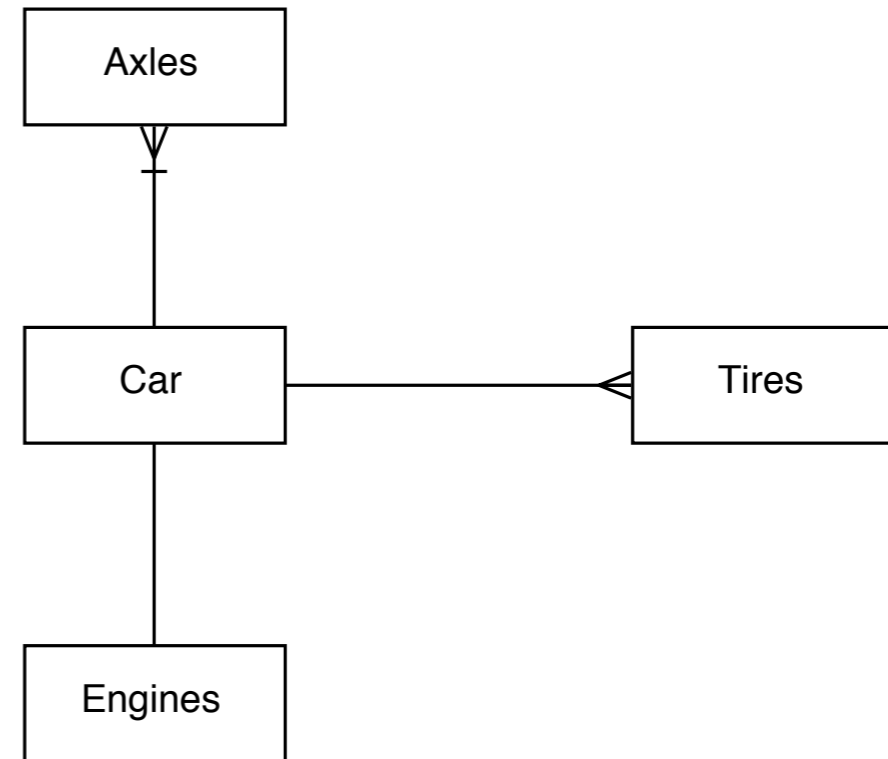
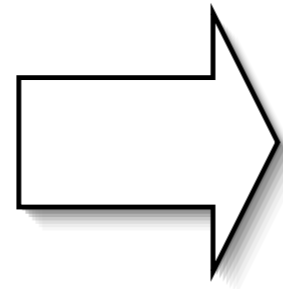
Some Truths about Java, SQL and JDBC

- ◆ Java is a language where data and action is self contained within a discrete body called an **Object**.
- ◆ SQL (**Self Query Language**) is the language of choice for retrieving and manipulating data from relational databases.
- ◆ JDBC (**Java Database Connectivity**) is the de-facto API for Java applications to issue SQL calls to a database
- ◆ JDBC and SQL are about as non-Object oriented as you can get

JDBC and SQL is powerful....

- ◆ Writing JDBC/SQL Code is Tedious to Write
- ◆ Writing JDBC/SQL code is Error-Prone
- ◆ In even small projects, you end up writing a lot of SQL Code
- ◆ Very few people are good to writing SQL code

Writing JDBC/SQL Code is Tedious



SELECT
FROM

car, tires, engines, axles

WHERE car.car_id = axles.car_id

AND car.car_id = tires.car_id

AND car.car_id = engines.car_id

Writing JDBC/SQL code is Error-Prone

```
pStatement = (OraclePreparedStatement) oraConn.prepareStatement(eventInsertSQL.toString());
eventVO.setEventID(new Integer(getNewEventID()));
```

```
pStatement.setInt(1, eventVO.getEventID().intValue());
pStatement.setString(2, eventVO.getStatusCode());
pStatement.setString(3, eventVO.getStatusMessage());
pStatement.setString(4, eventVO.getEventType());
pStatement.setString(5, eventVO.getOrderNO());
pStatement.setString(6, eventVO.getCustomer());
pStatement.setString(7, eventVO.getCreatedBy());
```

```
pStatement.execute();
```


```
//Retrieving the record I just inserted.
StringBuffer messageSQL = new StringBuffer(128);
```

```
messageSQL.append("SELECT message FROM asq_event WHERE event_id=? ");
messageSQL.append("FOR UPDATE OF message");
```

```
pStatement = (OraclePreparedStatement)oraConn.prepareStatement(messageSQL.toString());
pStatement.setInt(1, eventVO.getEventID().intValue());
rsMessage = pStatement.executeQuery()
```

```
    }
    finally{
        try{
            if (rsMessage!=null) rsMessage.close();
            if (pStatement!=null) pStatement.close();
            if (conn!=null) conn.close();
        }
        catch(SQLException e){}
    }
}
```

Forgot to close the statement
(Cursor Leak)

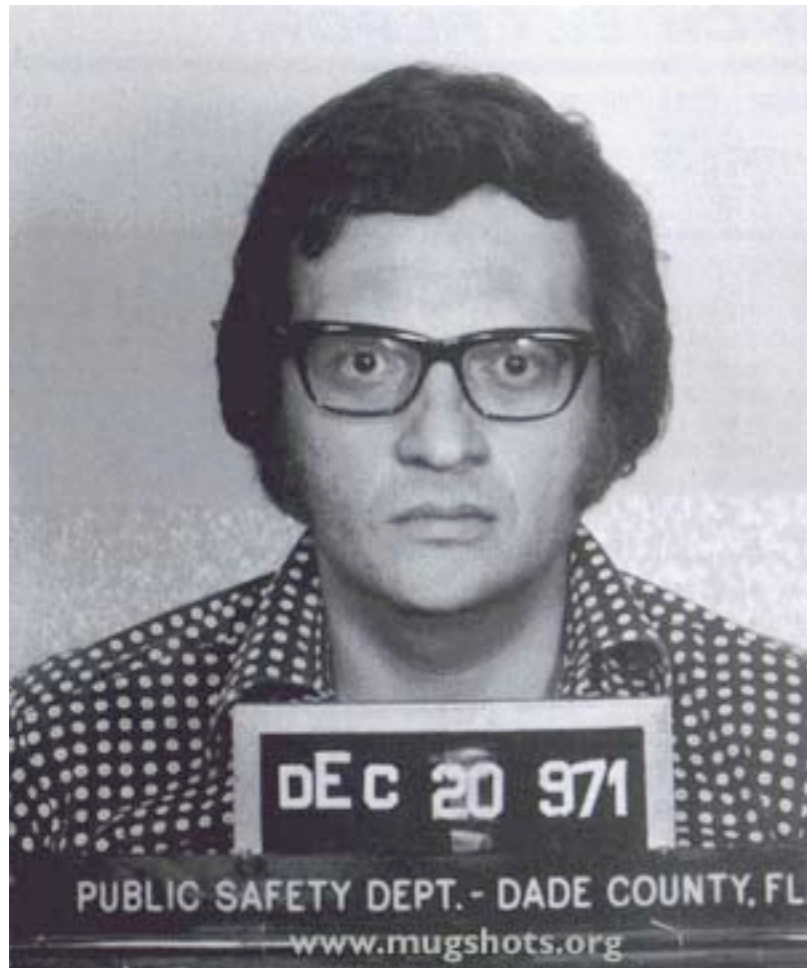


Writing SQL Code often Involves a Lot of Code



- ◆ On a previous project where we had to interact with a CRM we had over 1000+ lines of code purely dedicated to retrieving/manipulating customer information. (Never mind all of the other JDBC/SQL code we had to write)
- ◆ For most data entities you need to write a lot of code to carry out simple C.R.U.D. actions
- ◆ It becomes even more complicated when you have to start dealing with data relationships and referential integrity

Very Few People Write Good SQL Code



`SELECT * FROM customer
WHERE customer table has 75
columns and he only needs 3
columns`

Guilty of needlessly retrieving more data than he needs and hosting a bad talk show.

Very Few People Write Good SQL Code



```
SELECT customer_name,  
products_name FROM  
customer, products  
WHERE  
customer_name='OZZY'
```

Guilty of cartesian joins, cruelty to bats and destroying American culture. (Repeat Offender)

Very Few People Write Good SQL Code



```
SELECT first_name, middle_name ,  
last_name FROM Customer
```

```
WHERE SUBSTR(customer_id,1,5)  
LIKE '1234%' on a table with 12  
million customer rows
```

Guilty of not understanding how indexes work
, world domination and not paying for his
coffee.....

The Question Becomes Then

If Java is an Object-Oriented language....

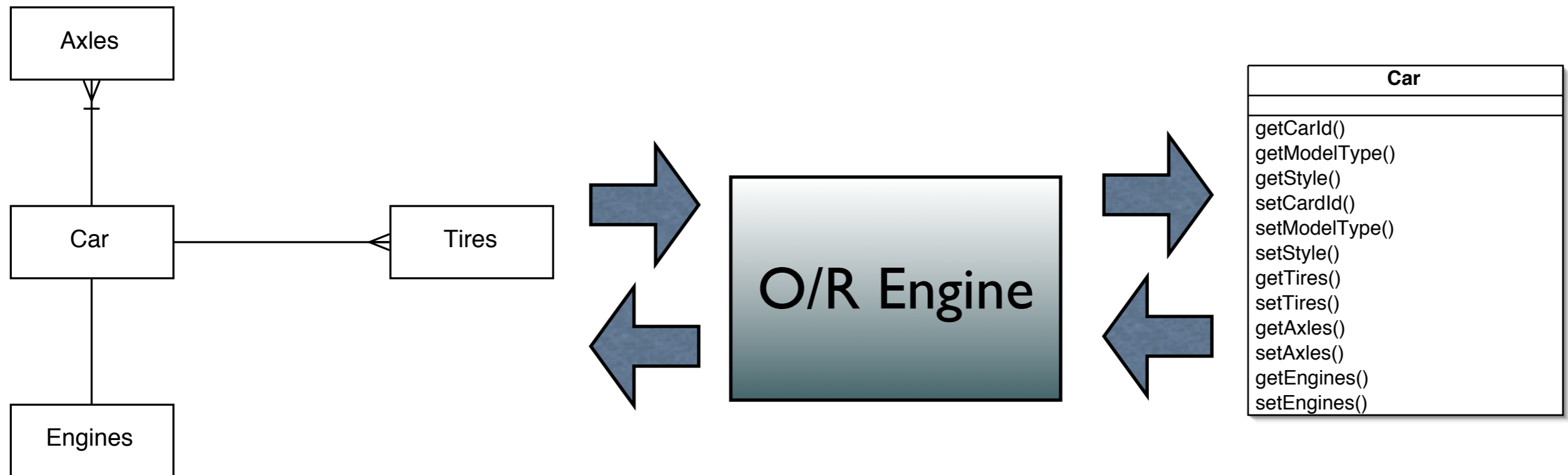
Why are we using a “row” and “table” based API for retrieving and manipulating data?

Wouldn't be nicer to retrieve object using Java Objects?



What are Object Relational Mapping Tools?

Enter Object-Relational Mapping Tools....



An O/R engines allows us to declaratively map data and data relationships from a relational database to a Java Object.

The O/R Engine generates all SQL Code to the database

The developer uses a Object Query Language (OQL) to retrieve data.

Types of Object-Relational Mapping Tools

- ◆ **Code Generators** - Takes a defined mapping and then generate Java code to carry out all database transactions.
 - ◆ Torque (<http://jakarta.apache.org/turbine/torque>)
 - ◆ Middlegen (<http://boss.bekk.no/boss/middlegen>)
- ◆ **Transparent Persistence** - Takes a defined mapping and execute database calls on the fly.
 - ◆ Object Relational Bridge
 - ◆ Hibernate

The Reduction in Work is Staggering

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("java:/MySQLDS");
conn = ds.getConnection();
conn.setAutoCommit(false);
```

```
StringBuffer insertSQL = new StringBuffer();
```

```
insertSQL.append("INSERT INTO story(      ");
insertSQL.append(" member_id      , ");
insertSQL.append(" story_title      , ");
insertSQL.append(" story_intro      , ");
insertSQL.append(" story_body      , ");
insertSQL.append(" submission_date      ");
insertSQL.append("      ");
insertSQL.append("VALUES(      ");
insertSQL.append(" ?      , ");
insertSQL.append(" ?      , ");
insertSQL.append(" ?      , ");
insertSQL.append(" ?      , ");
insertSQL.append(" CURDATE()      ) ");
```

```
ps = conn.prepareStatement(insertSQL.toString());
```

```
ps.setLong(1, memberVO.getMemberId().longValue());
ps.setString(2, postStoryForm.getStoryTitle());
ps.setString(3, postStoryForm.getStoryIntro());
ps.setString(4, postStoryForm.getStoryBody());
```

```
ps.execute();
conn.commit();
```

```
broker = ServiceLocator.getInstance().findBroker();
memberVO = (MemberVO) insertRecord;
```

```
broker.beginTransaction();
broker.store(memberVO);
broker.commitTransaction();
```

26 Lines

5 Lines

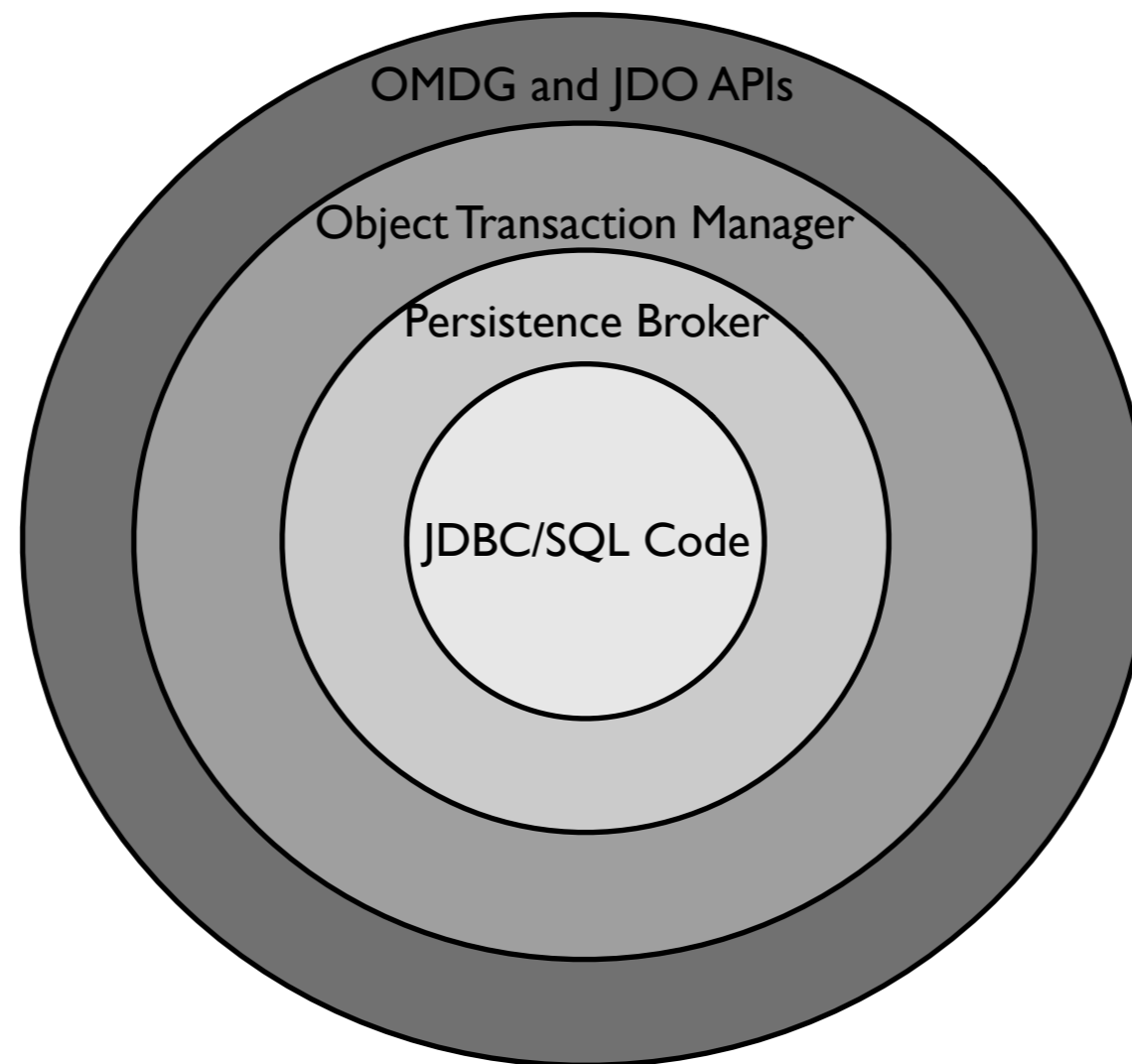
What is Object Relational Bridge?

- ◆ OJB is an Open Source O/R mapping tool from the Jakarta Apache Group.
- ◆ OJB was started by Thomas Mahler.
- ◆ It is a relative newcomer to the O/R mapping tool set
- ◆ Strengths of OJB include:
 - ◆ OJB is lightweight and extremely easy to setup.
 - ◆ Since OJB does not perform code generation, it makes it extremely easy to re-factor existing applications to use it.
 - ◆ OJB uses a micro-kernel architecture that allows it to support multiple database APIs.

Other OJB Features

- ◆ An object cache that greatly enhances performance and helps guarantee the identity of multiple objects pointing to the same row of data.
- ◆ When using the Persistence broker API, automatic persistence of children objects.
- ◆ An architecture that can support running in a single JVM or in a client-server mode that can support clusters.
- ◆ Integrates into an application server environment including participating in Container Managed transactions and access to JNDI data sources
- ◆ Multiple locking types including support for optimistic locking
- ◆ A built-in sequence manager.

The OJB Micro-kernel Architecture





Installing and Configuring Object Relational Bridge

Installing OJB

- ◆ OJB can be downloaded from the <http://jakarta.apache.org/ojb>
- ◆ At its core OJB needs the following libraries (all of which come with the distribution):
 - ◆ db-ojb-1.0.rc3-junit.jar
 - ◆ commons-beans-util.jar
 - ◆ commons-collections-2.0.jar
 - ◆ commons-logging.jar
 - ◆ commons-pool.jar
 - ◆ commons-dbcp.jar
 - ◆ commons-lang-1.0-mod.jar

OJB Configuration Files

OJB requires two basic configuration files:

- ◆ **OJB.properties** - Used to customize the run-time properties of OJB (caching, transaction management, etc....)
- ◆ **repository.xml** - Used to describe the mapping of the database tables to the database.

OJB.properties

The **OJB.properties** file is used to control the behavior of OJB run-time engine. Some of the information that can be configured in it includes:

- ◆ The server mode being used (single JVM or client-server).
- ◆ The number of persistence brokers in the pool.
- ◆ Configuration information for:
 - ◆ Locking
 - ◆ Caching
 - ◆ Logging
- ◆ Needs to be located in the CLASSPATH of the project.

OJB.properties

The repository.xml

The **repository.xml** file holds the O/R mappings that maps database tables to Java classes. Elements in this file include:

- ◆ JDBC Connection Information
 - ◆ DB2
 - ◆ Oracle
 - ◆ Microsoft Access/SQL Server
 - ◆ MySQL
 - ◆ Sybase
- ◆ <class-descriptor> elements
- ◆ <field-descriptor> elements
- ◆ <reference-descriptor> elements
- ◆ <collection-descriptor> elements

repository.xml

Setting up a Database Connection

- ◆ Database connections are defined in the **repository.xml** file
- ◆ Elements defined in the connection include:
 - ◆ A name to identify the connection with
 - ◆ The database platform being connected to.
 - ◆ The JDBC level (Version 1, 2 and 3 are supported. The default value is Version 1)
 - ◆ JDBC driver CLASS
 - ◆ Connection information (Protocol, Database alias, user name and password)

Setting up a Non-JNDI based Connection

```
<jdbc-connection-descriptor  
  jcd-alias="strutsdb"  
  platform="MySQL"  
  jdbc-level="2.0"  
  driver="org.gjt.mm.mysql.Driver"  
  protocol="jdbc"  
  dbalias="waf"  
  username="waf_user"  
  password="password"/>
```

Setting up a JNDI-based Connection

```
<jdbc-connection-descriptor  
  jcd-alias="strutsdb"  
  platform="MySQL"  
  jdbc-level="2.0"  
  jndi-datasource-name="java:/MySQLDS"/>
```



Implementing Mapping with Object Relational Bridge

Mapping Our First Table

For our first example we are going to map a class called MemberVO to a database table called member.

- ◆ Very simple data with basic data elements.
- ◆ No data relationships to worry about at this point.

repository.xml

MemberVO.java

Our First Mapping

Fully qualified Java class

Database Table

```
<class-descriptor class="com.wrox.javaedge.member.MemberVO" table="member">  
  <field-descriptor id="1" name="memberId" column="member_id" jdbc-type="BIGINT"  
    primarykey="true" autoincrement="true"/>  
  <field-descriptor id="2" name="firstName" column="first_name" jdbc-type="VARCHAR"/>  
  <field-descriptor id="3" name="lastName" column="last_name" jdbc-type="VARCHAR"/>  
  <field-descriptor id="4" name="userId" column="userid" jdbc-type="VARCHAR"/>  
  <field-descriptor id="5" name="password" column="password" jdbc-type="VARCHAR"/>  
  <field-descriptor id="6" name="email" column="email" jdbc-type="VARCHAR"/>  
</class-descriptor>
```

Java Class Property

Database Column

Data Types Supported

The **jdbc-type** attribute in the **<field-descriptor>** tags supports a number of JDBC data types including:

- ◆ INTEGER
- ◆ FLOAT
- ◆ DOUBLE
- ◆ DATE/TIMESTAMP
- ◆ VARCHAR
- ◆ BLOB/CLOBS/VARBINARY

What about JDO support?

- ◆ OJB does support JDO through the Sun reference implementation.
- ◆ To use JDO in OJB you need:
 - ◆ Download JDO from Sun (**<http://jcp.org/aboutjava/communityprocess/final/jsr012/index.html>**)
 - ◆ Make the JDO part of your Jar directory
 - ◆ Setup your repository.xml file. (No changes there)
 - ◆ Setup a *.JDO specific mapping
 - ◆ After building your class files, you need to invoke the JDO runtime enhancement.

JDO File Example

```
<!DOCTYPE jdo SYSTEM "file:///target/classes/test/org/apache/obj/jdo.dtd">
<jdo>

<package name="com.wrox.javaedge.member">

<class name="MemberVO">
<extension vendor-name="obj" key="table" value="member"/>

<field name="memberId">
  <extension vendor-name="obj" key="column" value="member_id"/>
</field>
<field name="firstName">
  <extension vendor-name="obj" key="column" value="first_name"/>
</field>
<field name="lastName">
  <extension vendor-name="obj" key="column" value="last_name"/>
</field>
.
.
.
</class>
</package>
</jdo>
```

Telling OJB How To Access and Set Properties

You can configure OJB to read/write data from a Java class in one of two ways:

- ◆ Using Java reflection to directly set a property within the class.
- ◆ Use JavaBean `get()/set()` methods to access data
- ◆ Using Java reflection allows you tighter control over what properties you are going to read and write to.
- ◆ Using JavaBean `get()/set()` methods allow you to abstract away how data is actually handled by a class.

Telling OJB How To Access and Set Properties

To configure how OJB is going to handle reading and writing data to and from a Java class you must set the **PersistentFieldClass** attribute in the OJB.properties to be one of two values:

- ◆ org.apache.ojb.broker.metadata.PersistentFieldDefaultImpl - Sets the attribute via Java reflection.
- ◆ org.apache.ojb.broker.metadata.PersistentFieldPropertyImpl - Sets an attribute via get()/set() methods.

Fun with CLOBs

OJB can handle Blob and Clob data types. However, for the end-developer isn't it better to hide the complexities of dealing with Blobs.

- ◆ Configuring OJB to use `get()/set()` methods allow us to perform data conversion on retrieved elements.
- ◆ Developers only deal with Strings and not the actually `byte[]` array.

StoryVO.java

repository.xml

Retrieving a Single Record

OJB provides two different methods for retrieving data. They are

- ◆ Query By Example - Retrieves all objects whose attributes match values set on an instance of an Object.
- ◆ Object Query Language (OQL) - Similar in function to SQL, but used to retrieve Objects instead of database rows.

First Lets Get a Connection

```
public PersistenceBroker findBroker() throws ServiceLocatorException{
    PersistenceBroker broker = null;
    try{
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
    }
    catch(PBFactoryException e) {
        e.printStackTrace();
        throw new ServiceLocatorException("PBFactoryException error " +
            "occurred while parsing the repository.xml file in " +
            "ServiceLocator constructor",e);
    }

    return broker;
}
```


Performing a Query By Example - Retrieving a Single Row

```
try {  
    broker = .getInstance().findBroker();  
    storyVO = new StoryVO();  
    storyVO.setStoryId(new Long(primaryKey));  
  
    Query query = new QueryByCriteria(storyVO);  
    storyVO = (StoryVO) broker.getObjectByQuery(query);  
} catch (ServiceLocatorException e) {  
    log.error("PersistenceBrokerException thrown in StoryDAO.findByPK():  
" + e.toString());  
    throw new DataAccessException("Error in StoryDAO.findByPK(): " +  
        e.toString(),e);  
}  
finally {  
    if (broker != null) broker.close();  
}
```

Defining the query criteria

Call to retrieve the data.
Returns a single object.

If No Object is Found a Null Value is Retrieved!

Retrieving A Single Object with JDO

```
//Plug-in Point to get to OJB
OjbStorePMF factory= new OjbStoryPMF();

//Used to manage the transaction
PersistenceManager manager = factory.getPersistenceManager();

//Setting up my query
MemberVO memberVO = new MemberVO();
MemberVO.setMemberID(id);

PersistenceBroker broker = PersistenceBrokerFactory.defaultPersistenceBroker();
Identity oid = new Identity(memberVO, broker);

//Executing the query
memberVO = (MemberVO) manager.getObjectById(oid, false);
```

Performing a Query By Example - Retrieving Multiple Rows

Defining the query criteria

```
try {  
    broker = ServiceLocator.getInstance().findBroker();  
    CustomerVO = new CustomerVO();  
    customerVO.setLastName("carnell");  
  
    Query query = new QueryByCriteria(customerVO);  
    Collection = (Collection) broker.getCollectionByQuery(query);  
} catch (ServiceLocatorException e) {....}  
finally {  
    if (broker != null) broker.close();  
}
```

Call to retrieve the data.
Returns a single object.

Inserting and Updating Data

Setting up the object to be inserted

```
StoryVO storyVO = new StoryVO();  
storyVO.setStoryTitle("Test Story");  
storyVO.setStoryIntro("This is the Story Intro");  
storyVO.setStoryBody("This is the Story Body");  
storyVO.setStoryMemberVO(memberVO);
```

```
broker = ServiceLocator.getInstance().findBroker();  
broker.beginTransaction();  
broker.store(storyVO);  
broker.commitTransaction();
```

Telling OJB to store the object.

Deleting Data

```
broker = ServiceLocator.getInstance().findBroker();
```

```
//Begin the transaction.
```

```
broker.beginTransaction();
```

```
broker.delete(storyVO);
```

```
broker.commitTransaction();
```



OBJ deletes the record

How Primary Keys are Generated

OJB does have a built in primary-key generation mechanism. To use it you must:

- ◆ Uncomment the database profile in the OJB **build.properties** file.
- ◆ Edit the OJB distribution/profile/*database type* file to point to the database where OJB specific tables will be generated for primary key usage.
- ◆ Run the **prepare-testdb** ant target to build the OJB primary key tables.

Modifying the build.properties

```
# With the 'profile' property you can choose the RDBMS
# platform OJB is using
# implemented profiles:
#profile=hsqldb
#profile=mssqldb
profile=mysql
#profile=db2
#profile=oracle
#profile=msaccess
#profile=postgresql
#profile=informix
#profile=sybase
#profile=sapdb
```



Uncommenting the mysql
profile

Modifying mysql.profile

dbmsName = MySql

jdbcLevel = 2.0

urlProtocol = jdbc

urlSubprotocol = mysql

urlDbalias = //localhost:3306/javaedge

databaseDriver = org.gjt.mm.mysql.Driver

databaseUser = waf

databasePassword = waf_user

databaseHost = localhost

Running the Ant Target

- ◆ To build the tables run **ant prepare-db**. You should see the following tables generated in your target database:
 - ◆ OJB_HL_SEQ
 - ◆ OJB_LOCKENTRY
 - ◆ OJB_NRM
 - ◆ OJB_DLIST
 - ◆ OJB_DLIST_ENTRIES
 - ◆ OJB_DSET
 - ◆ OJB_DSET_ENTRIES
 - ◆ OJB_DMAP
 - ◆ OJB_DMAP_ENTRIES

Mapping One-to-One Relationships

```
<class-descriptor class="com.wrox.javaedge.story.StoryCommentVO" table="story_comment">
```

```
  <field-descriptor id="1" name="commentId" column="comment_id" jdbc-type="BIGINT"
    primarykey="true" autoincrement="true"/>
```

```
  <field-descriptor id="2" name="storyId" column="story_id" jdbc-type="BIGINT"/>
```

```
  <field-descriptor id="3" name="memberId" column="member_id" jdbc-type="BIGINT"/>
```


.....

```
  <reference-descriptor name="commentAuthor"
    class-ref="com.wrox.javaedge.member.MemberVO"
    auto-retrieve="true">
```

```
    <foreignkey field-id-ref="3"/>
```

```
  </reference-descriptor>
```

```
</class-descriptor>
```



Creates a one-to-one relationship between a the author of the comment and the comment

Mapping One-to-Many Relationships

```
<class-descriptor class="com.wrox.javaedge.story.StoryVO" table="story">
```

```
  <field-descriptor id="1" name="storyId" column="story_id" jdbc-type="BIGINT"  
    primarykey="true" autoincrement="true"/>
```

```
  <field-descriptor id="2" name="memberId" column="member_id" jdbc-type="BIGINT"/>
```

```
  <field-descriptor id="3" name="storyTitle" column="story_title" jdbc-type="VARCHAR"/>
```

```
  <field-descriptor id="4" name="storyIntro" column="story_intro" jdbc-type="VARCHAR"/>
```

```
  <field-descriptor id="5" name="storyBody" column="story_body" jdbc-type="LONGVARBINARY"/>
```

```
  <field-descriptor id="6" name="submissionDate" column="submission_date" jdbc-type="DATE"/>
```

```
  <collection-descriptor name="comments" element-class-ref="com.wrox.javaedge.story.StoryCommentVO"  
    auto-retrieve="true" auto-update="true" auto-delete="true">
```

```
    <inverse-foreignkey field-id-ref="2"/>
```

```
  </collection-descriptor>
```

```
  <reference-descriptor name="storyAuthor" class-ref="com.wrox.javaedge.member.MemberVO" auto-  
retrieve="true">
```

```
    <foreignkey field-id-ref="2"/>
```

```
  </reference-descriptor>
```

```
</class-descriptor>
```

Mapping Many-to-Many Relationships

```
<class-descriptor class="com.wrox.javaedge.story.StoryVO" table="story">
  <field-descriptor id="1" name="storyId" column="story_id" jdbc-type="BIGINT" primarykey="true"
autoincrement="true"/>
  <field-descriptor id="2" name="memberId" column="member_id" jdbc-type="BIGINT"/>
  <field-descriptor id="3" name="storyTitle" column="story_title" jdbc-type="VARCHAR"/>
  <field-descriptor id="4" name="storyIntro" column="story_intro" jdbc-type="VARCHAR"/>
  <field-descriptor id="5" name="storyBody" column="story_body" jdbc-type="LONGVARBINARY"/>
  <field-descriptor id="6" name="submissionDate" column="submission_date" jdbc-type="DATE"/>

  <collection-descriptor name="comments"
    element-class-ref="com.wrox.javaedge.story.StoryCommentVO"
    auto-retrieve="true"
    auto-update="true" auto-delete="true"
    indirection_table="STORY_STORY_COMMENTS">
    <fk-pointing-to-this-class column="STORY_ID"/>
    <fk-pointing-to-this-class column="COMMENT_ID"/>
  </collection-descriptor>

  <reference-descriptor name="storyAuthor" class-ref="com.wrox.javaedge.member.MemberVO"
auto-retrieve="true">
  <foreignkey field-id-ref="2"/>
</reference-descriptor>
</class-descriptor>
```

Building More Sophisticated Queries

OJB allows the developer to retrieve data using more sophisticated queries.

- ◆ Queries are built using a Criteria object
- ◆ Criteria objects can be “AND” and “OR”ed to create more complex queries

More Complex Queries in Action

```
Criteria criteriaEquals = new Criteria();
```

```
Criteria criteriaLike = new Criteria();
```

```
criteriaEquals.addEqualTo("last_name", "Smith");
```

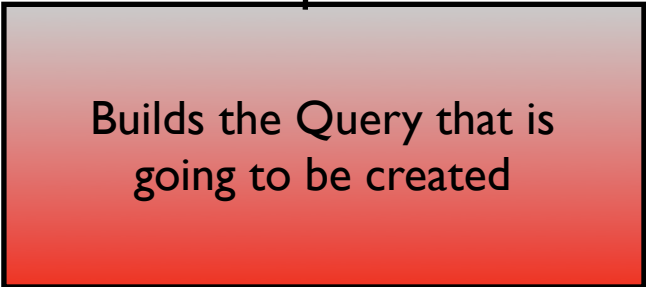
```
criteriaLike.addIsLike("first_name", "J%");
```

```
criteriaEquals.addAndCriteria(criteriaLike);
```

```
criteriaEquals.addOrderByAscending("first_name");
```

```
Query query = new QueryByCriteria(MemberVO.class,  
                                criteriaEquals);
```

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



Builds the Query that is
going to be created

All of This Translates Into...

**WHERE last_name='Smith' and first_name
LIKE 'J%' ORDER BY first_name
ASCENDING**

**To see the SQL code generated by OJB, you need to set the
org.apache.broker.accesslayer.sql.SqlGeneratoDefaultImpl.LogLevel to
DEBUG. All the SQL code generated by OJB will be written out
System.out.**

Some of the Methods for Building Queries

- ◆ addEqualTo()
- ◆ addGreaterThan()
- ◆ addGreaterOrEqualThan()
- ◆ addLessThan()
- ◆ addLessOrEqualThan()
- ◆ addIsNull()
- ◆ addNotNull()
- ◆ addIsLike()
- ◆ addSQL()

This is only a small subset of the functions for building queries



Proxies for Performance

So What's Wrong with this Picture?

```
<class-descriptor class="com.wrox.javaedge.story.StoryVO" table="story">
```

```
  <field-descriptor id="1" name="storyId" column="story_id" jdbc-type="BIGINT" primarykey="true"
    autoincrement="true"/>
```

```
  <field-descriptor id="2" name="memberId" column="member_id" jdbc-type="BIGINT"/>
```

```
  <field-descriptor id="3" name="storyTitle" column="story_title" jdbc-type="VARCHAR"/>
```

```
  <field-descriptor id="4" name="storyIntro" column="story_intro" jdbc-type="VARCHAR"/>
```

```
  <field-descriptor id="5" name="storyBody" column="story_body" jdbc-type="LONGVARBINARY"/>
```

```
  <field-descriptor id="6" name="submissionDate" column="submission_date" jdbc-type="DATE"/>
```

```
  <collection-descriptor name="comments" element-class-ref="com.wrox.javaedge.story.StoryCommentVO"
    auto-retrieve="true" auto-update="true" auto-delete="true">
```

```
    <inverse-foreignkey field-id-ref="2"/>
```

```
  </collection-descriptor>
```

```
  <reference-descriptor name="storyAuthor" class-ref="com.wrox.javaedge.member.MemberVO"
    auto-retrieve="true">
```

```
    <foreignkey field-id-ref="2"/>
```

```
  </reference-descriptor>
```

```
</class-descriptor>
```

Real-World Hyperlinks

Posted By: Anonymous on 2003-07-10

RunAmuk writes "Wired is reporting about being able to "Point and click your mobile phone at a poster in London movie theaters this July and you'll be able to directly access the movie's Web page." While there are many practical uses for this technology,

[Full Story](#)

Linux on the Desktop

Posted By: Anonymous on 2003-07-10

webmaven writes "Mitch Kapor's Open Source Application Foundation just released a 34 page report on the Desktop Linux market, written by Bart Decrem, who has discussed desktop Linux previously. The OSAF is working on Chandler, which the press have general

[Full Story](#)

Developers: "Quick 'n Dirty" vs. "Correct and Proper"?

Posted By: Anonymous on 2003-07-10

A not-so Anonymous Coward enters this query: "I keep finding myself on projects where a quick and dirty solution will bring in money for the company, and a correct (ie, properly documented, well engineered, process followed, etc) solution will get us left

[Full Story](#)

Repel Bugs With Your Cell Phone

Posted By: Anonymous on 2003-07-10

telstar writes "Starting Monday, SK Telecom Co. in South Korea will begin offering a ringtone designed to repel mosquitoes for the one-time price of \$2.50. The ringtone, inaudible to humans, has a range of three feet, and functions just like any other ri

[Full Story](#)

```
PersistenceBroker broker = null;  
Collection results = null;
```

```
Criteria criteria = new Criteria();  
criteria.addOrderByDescending("storyId");
```

```
Query query = QueryFactory.newQuery(StoryVO.class, criteria);
```

```
query.setStartAtIndex(1);  
query.setEndAtIndex(MAXIMUM_TOPSTORIES - 1);
```

The way the Story mapping is setup, every StoryVO automatically retrieves a Collection of all of StoryCommentVO's associated with it. In the main home page we only care about the Stories, not the Story comments.

Lets Do the Math

- 10 Stories with 10 Comments
- 10 users all hit the main home page concurrently.
- $10 \times 10 \times 10 = 1000$ Approximately Objects

Yikes!

- ◆ Wow if you are not careful you can end up with a ton of mappings.
- ◆ Fortunately, OJB supports the concepts of Proxies.
- ◆ Proxies allow you to retrieve data without having to retrieve all of the data for an object or all of the children objects. This is the concept “lazy instantiation.”
- ◆ Two types of Proxies:
 - ◆ Single Class Proxies - Used to only load the OID for an object.
 - ◆ Collection Proxies - Will not load children mappings until the Children objects are accessed.

Writing a Proxy Class

- ◆ To use Proxies class you need to implement the following items:
 - ◆ An Interface class that has all of the get()/set() methods for a class.
 - ◆ A concrete implementation of the class being mapped in OJB.
 - ◆ A Proxy class that extends the OJB class VirtualProxy.
 - ◆ An proxy attribute defined for the OJB mapping.

IPerson.java

Person.java

PersonProxy.java

Dynamic Proxies

- ◆ Writing all of these proxies classes are a pain.
- ◆ Fortunately OJB allows you to use Dynamic Proxies and eliminate the need to write a proxy class.
- ◆ However, even with Dynamic Proxies, you still need to write an interface class.

Collection Proxies

- ◆ Collection proxies are used to proxy relationships between objects.
- ◆ Collection proxies do not retrieve data until the first time a object is requested from the collection.
- ◆ Do not use collection proxies in combination with regular proxies, as you will end up far more calls to a database then needed.
- ◆ You can implement your own Collection proxies by modifying the `CollectionProxyClass` attribute in the `OJB.properties`.

StoryCommentVO.java

IStoryComment.java



Moving Beyond the Technology: Developing a Data Access Strategy

Beyond OJB

- ◆ OJB is a tool, not a solution.
- ◆ Tools like OJB can significantly speed up your development efforts.
- ◆ However, most organizations should architect their data access tier to be technology independent.
- ◆ In the next section, we are going to look at how to use some common J2EE Design Patterns to build a flexible data access solution.

The Truth about the Data Access Tier

Most development teams do not have a coherent strategy for building their data access tier

- ◆ The main reason is developers tend to only think about modeling the business tier of the application. They define their data access tier by:
 - ◆ The particular data access technology being used (JDBC, Entity EJBs, etc..)
 - ◆ The database they use to hold their data (Oracle, Sybase, SQL Server)

The Problem with This Approach

- ◆ Technologies change at a rapid rate.
- ◆ Different technologies offer different competitive advantages
- ◆ Not paying attention to this and coupling your applications to particular technology or vendor can leave you unable to be responsive to new business requirements.

This Can Lead To Data Madness

Symptoms of Data Madness include:

- ◆ The creation of tight dependencies between applications and the structures of the underlying data stores.
- ◆ The presence of a 2.5 tier architecture.
- ◆ The inability to easily port an application to another database platform because of vendor-specific database extensions.
- ◆ The inability to easily change data access technologies.

A Bad, But Typical Piece of Code

```
public ActionForward perform(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response){
```

```
    PostStoryForm postStoryForm = (PostStoryForm) form;
    HttpSession session = request.getSession();
```

```
    .....
```

```
    try{
        Context ctx = new InitialContext();
        DataSource ds = (DataSource) ctx.lookup("java:/OracleDS");
        conn = ds.getConnection();
        conn.setAutoCommit(false);
```

```
        StringBuffer insertSQL = new StringBuffer();
```

```
        insertSQL.append("INSERT INTO story      ");
        insertSQL.append("VALUES(                ");
        insertSQL.append("  story_id_seq.nextval, ");
        insertSQL.append(" ?                , ");
        insertSQL.append(" ?                , ");
        insertSQL.append(" ?                , ");
        insertSQL.append(" ?                , ");
        insertSQL.append(" SYSDATE         ) ");
```

```
        ps = conn.prepareStatement(insertSQL.toString());
```

```
        ps.setLong(1, memberVO.getMemberId().longValue());
        ps.setString(2, postStoryForm.getStoryTitle());
        ps.setString(3, postStoryForm.getStoryIntro());
        ps.setString(4, postStoryForm.getStoryBody());
```

```
        ps.execute();
        conn.commit();
```

```
    }
```

The application has been needlessly exposed to data access implementation details

The application now has a dependency on the Oracle database.

Details about the underlying data structure are exposed to the application.

Watch out for SQL droppings in the business tier.

The Problem with This Approach

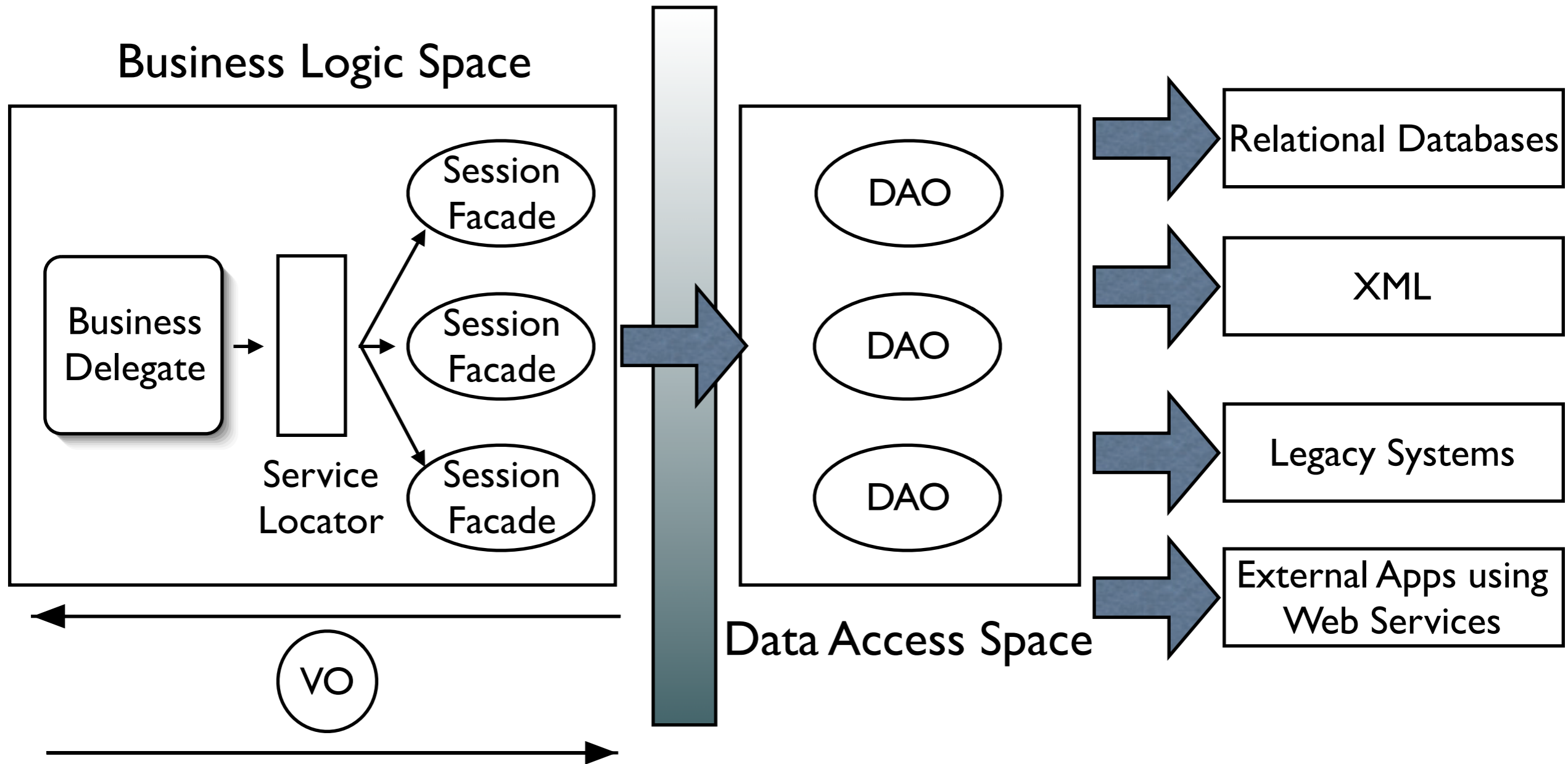
- ◆ Good architecture often starts with what appear to be small design decisions.
- ◆ We often times do not feel the pain of bad architecture until after we try to extend or maintain the application.
- ◆ Paying attention to these details at the data access tier often results in big re-use payoffs later on
- ◆ Remember, for many developers the majority of their careers are spent moving data from point A-B.

What Should Be The Goal?

Ideally a data access tier should be designed so that business services can consume data without giving any idea how or from where the data is being retrieved.

- ◆ Allow a clean separation of data persistence logic or business logic.
- ◆ Decouple the application(s) from any knowledge of the database platform in which the data resides.
- ◆ Abstract away the physical details of how data is stored within the database and the relationships that exist between entities within the database.
- ◆ Simplify the application development process by hiding details associated with getting a DB connection, issuing commands etc....

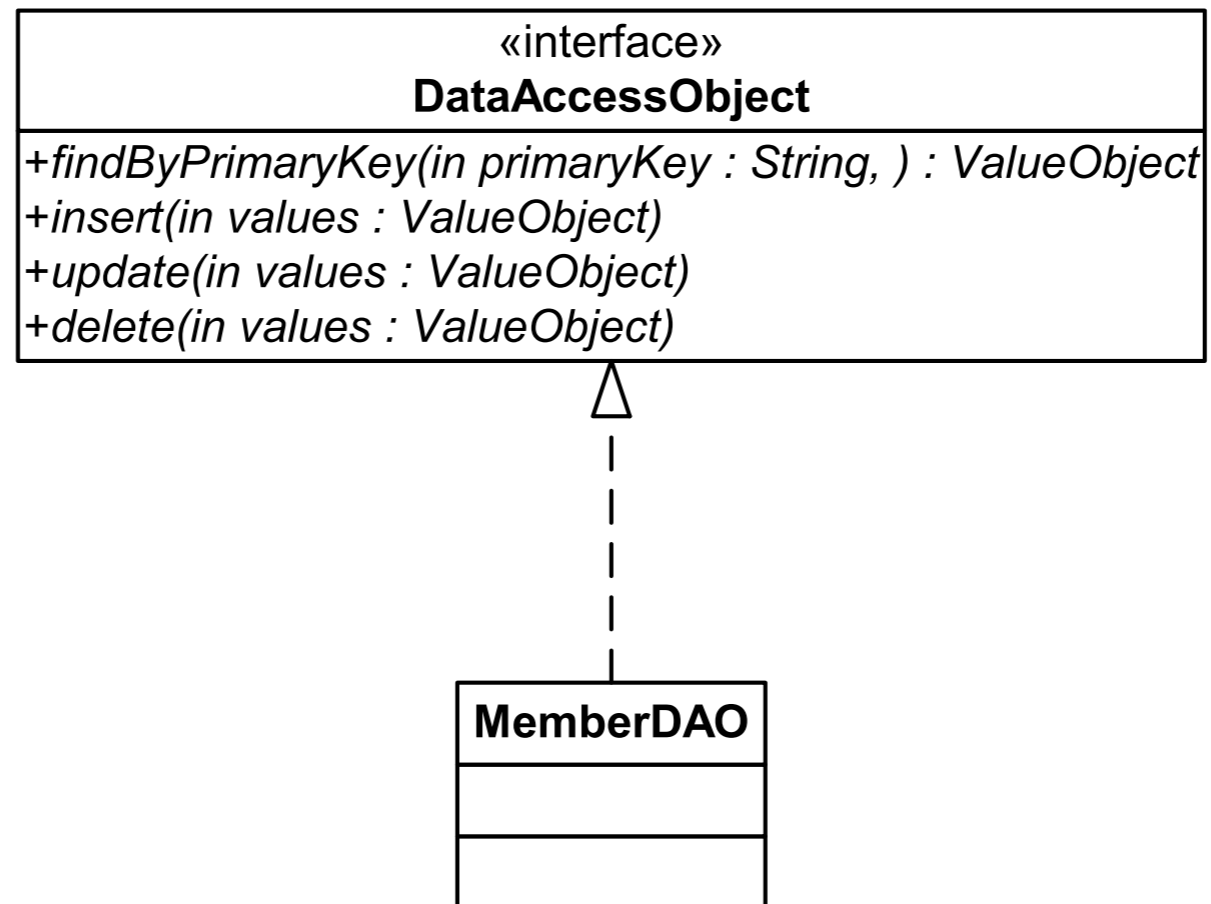
Ideally We Want



What is a Data Access Object?

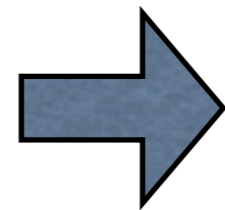
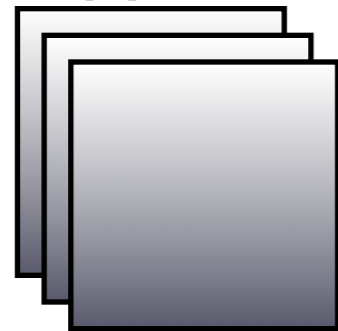
- ◆ Encapsulates all physical details of the data source being accessed.
- ◆ Abstracts away how data is being retrieved and manipulated.
- ◆ Centralizes all CRUD logic behind a logical Java interface.
- ◆ Allows for the optimization of data requests through caching.

The Data Access Object

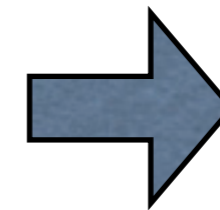
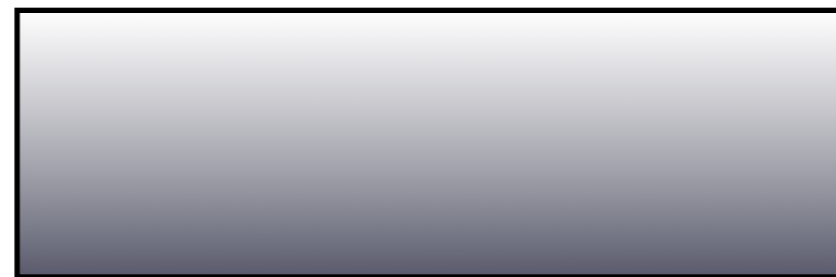


The DAO Factory

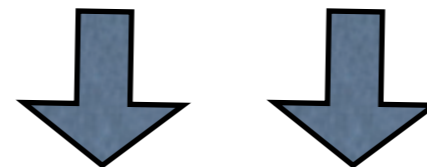
I. Applications



2. DAOFactory



3. DAO Implementations

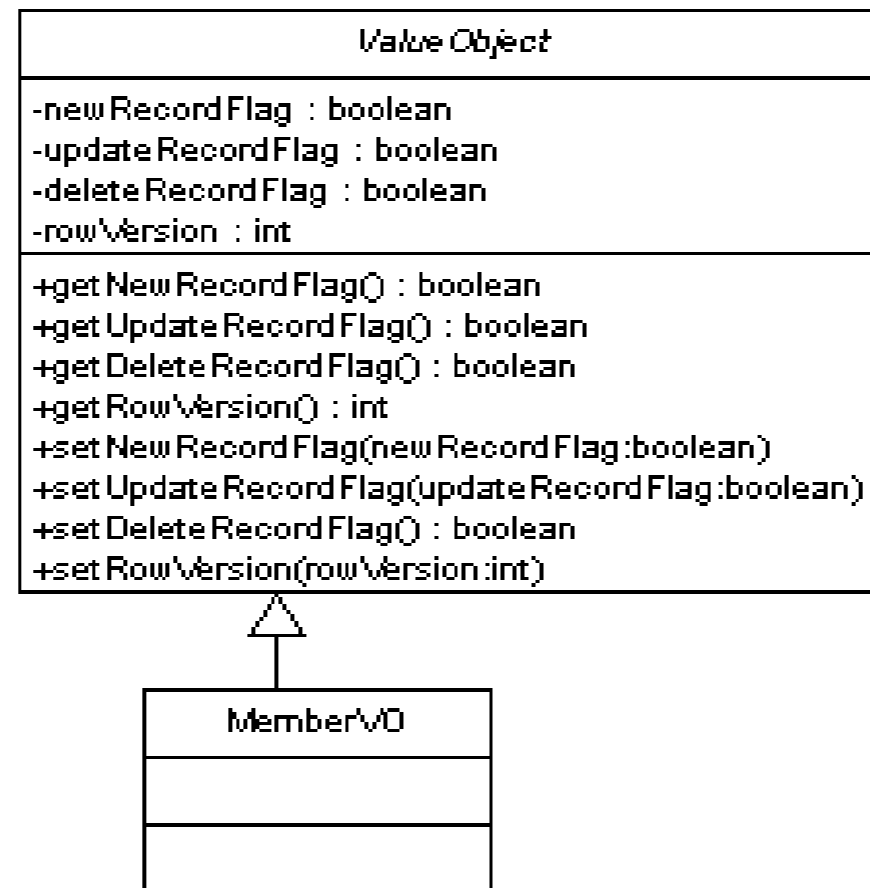


1. Application needs a DAO to retrieve and manipulate data. It creates a DAOFactory and calls getDAO().

2. The DAOFactory will take the user's request and looks up the fully-qualified class name (dao.properties).

3. Upon retrieving the class name, the DAO will instantiate an instance of the DAO.

Value Object/Data Transfer Object Pattern



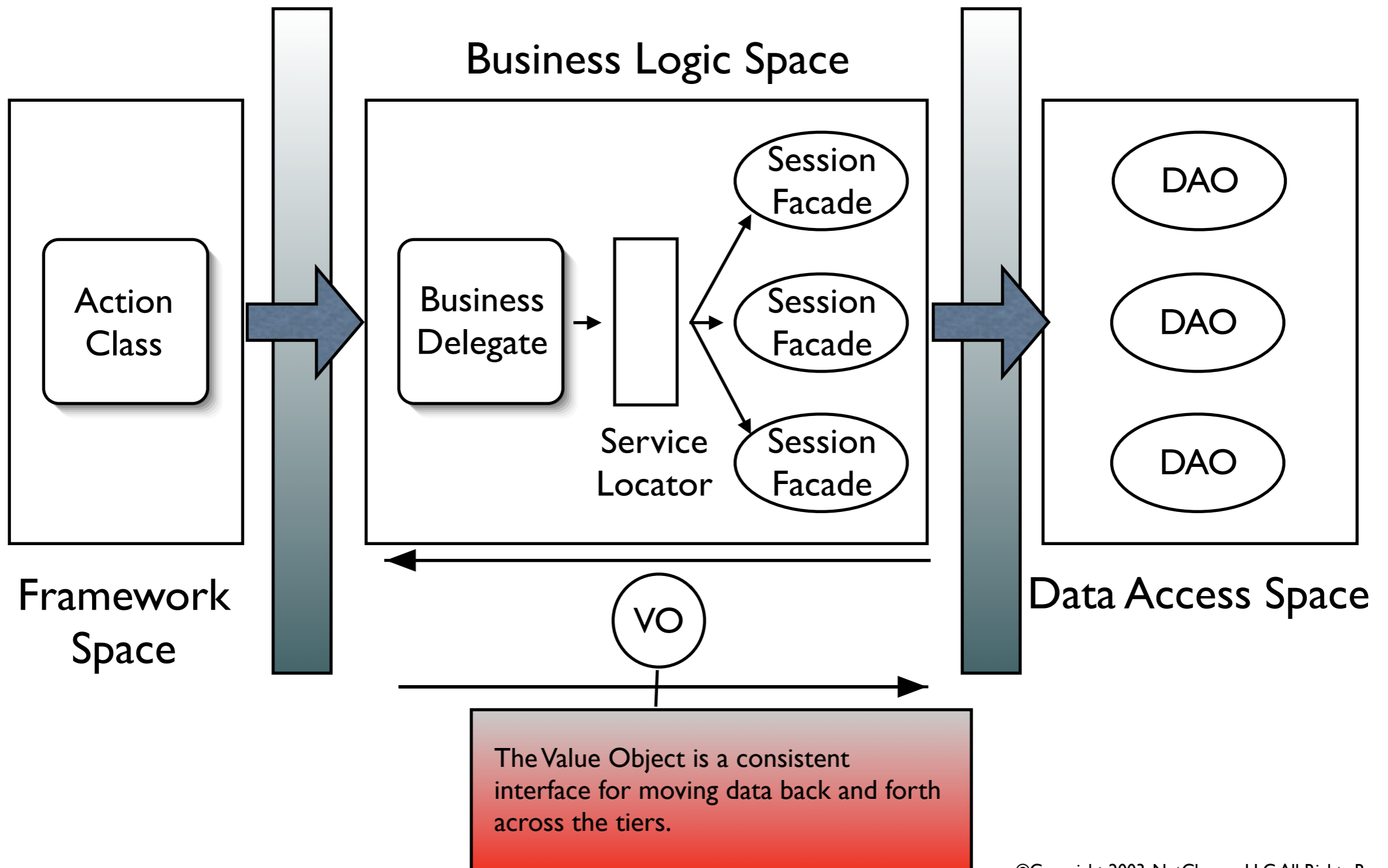
Some History On the Value Object

- ◆ The Value Object pattern was originally implemented to deal with the performance problems inherent in the Entity EJB 1.1 specification.
- ◆ The introduction of Local interfaces for the Entity EJB has mitigated the need for this pattern.
- ◆ Some would argue that the Value Object Pattern is a step back from OO-design because it is separating data and behavior.

The Value Object Pattern Still has Value

- ◆ A Value Object Pattern wrappers a row of data retrieved from a data source in a Plain Old Java (POJ) class.
- ◆ This lets the user deal with a logical view of the data instead of a physical view.
 - ◆ Value objects hide the data types of the data being retrieved.
 - ◆ Value objects hide the relationships that exist between tables in the database.
- ◆ Value objects are the glue that ties the tiers together. They are used to pass data back and forth between all tiers in the application. This is where they have **Value** (OK bad Pun)

The Value Object In Action



Understand How Data is Going to be Used

- ◆ Watch how much data you pull back from your DAOs.
- ◆ A DAO can abstract any data source.
- ◆ A DAO can build Value Objects that have to one-to-many and many-to-many relationships.
- ◆ Watch the level of DAO granularity.
- ◆ Value Objects represent a view of the data.
- ◆ It is possible for the same DAO to return multiple types of Value Objects all representing a specific view of the data.
- ◆ Be careful with how “deep” you make your value objects.
- ◆ Leverage proxies at the appropriate time.

Additional Data Access Strategies

Even with the J2EE Data Access Patterns there are still several things that need to be built into a persistence framework:

- ◆ Abstracting small details
- ◆ Exception Handling
- ◆ SQL Code Management
- ◆ Primary Key Generation

The Service Locator (SL)



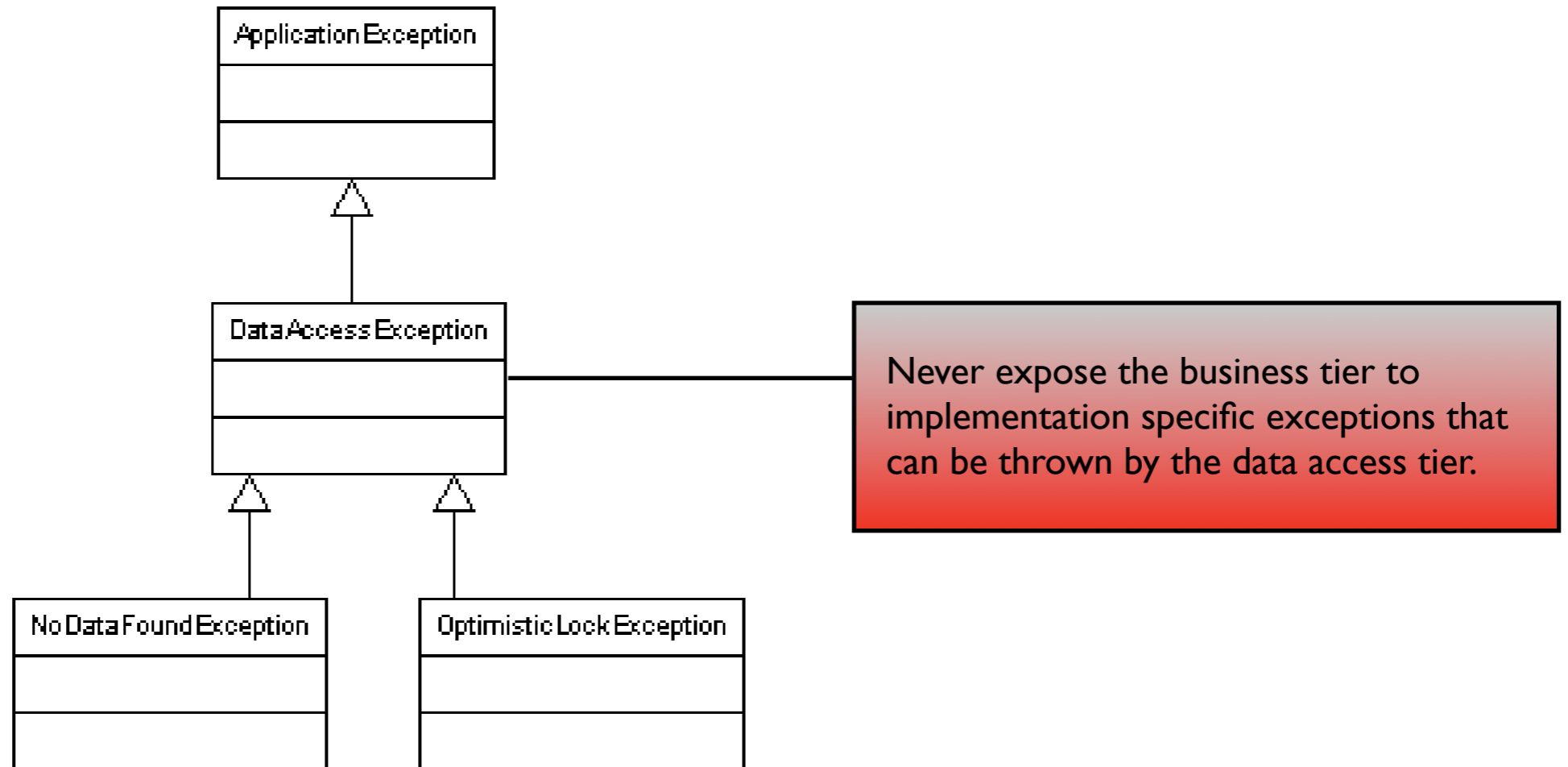
The Service Locator hides how a resource is instantiated and returned.

This patterns gives the architect more flexibility in implementing optimizations.

SQL Code Management

- ◆ String and StringBuffer objects containing SQL code can be the most frequently created objects in an application.
- ◆ This rampant object creation can represent a significant amount of overhead.
- ◆ However, we can implement a mechanism that externalizes the SQL statement from the application's code. In addition, we can cache these SQL statements to improve efficiency.

Exception Handling



Primary Key Generation

- ◆ OJB provides us with a mechanism for generating primary keys. However what happens if you have to integrate to an existing CRM.
- ◆ The management of primary keys is often times very database dependent.
- ◆ Ideally we do not want to expose our applications to underlying database extensions.
- ◆ We are going to examine different options for primary key generation:
 - ◆ Database Triggers
 - ◆ Database “Sequence” Objects
 - ◆ PrimaryKeyManager Class

Some Final Thoughts

- ◆ A data access tier is often times one of the most neglected pieces of an application architecture.
- ◆ However, for most organizations a data access tier can be one of the more re-used pieces of code.
- ◆ Use the J2EE Data Access patterns shown in this presentation to abstract away implementation details.
- ◆ Most importantly, understand how your data is used by your applications. There is no one size fits all solution.

Any Questions?

Please feel free to send me an email if you have any questions:

john.carnell@netchange.us