iBATIS Data Access Objects

Developer Guide

Version 2.0

February 18, 2006

Introduction

When developing robust Java applications, it is often a good idea to isolate the specifics of your persistence implementation behind a common API. Data Access Objects allow you to create simple components that provide access to your data without revealing the specifics of the implementation to the rest of your application. Using DAOs you can allow your application to be dynamically configured to use different persistence mechanisms. If you have a complex application with a number of different databases and persistence approaches involved, DAOs can help you create a consistent API for the rest of your application to use.

Data Access Objects (com.ibatis.dao.*)

The iBATIS Data Access Objects API can be used to help hide persistence layer implementation details from the rest of your application by allowing dynamic, pluggable DAO components to be swapped in and out easily. For example, you could have two implementations of a particular DAO, one that uses the iBATIS SQL Maps framework to persist objects to the database, and another that uses the Hibernate framework. Another example would be a DAO that provides caching services for another DAO. Depending on the situation (e.g. limited database performance vs. limited memory), either the cache DAO could be "plugged in" or the standard un-cached DAO could be used. These examples show the convenience that the DAO pattern provides, however, more important is the safety that DAO provides. The DAO pattern protects your application from possibly being tied to a particular persistence approach. In the event that your current solution becomes unsuitable (or even unavailable), you can simply create new DAO implementations to support a new solution, without having to modify any code in the other layers of your application.

Note! The DAO framework and SQLMaps Framework are completely separate and are not dependent on each other in any way. Please feel free to use either one separately, or both together.

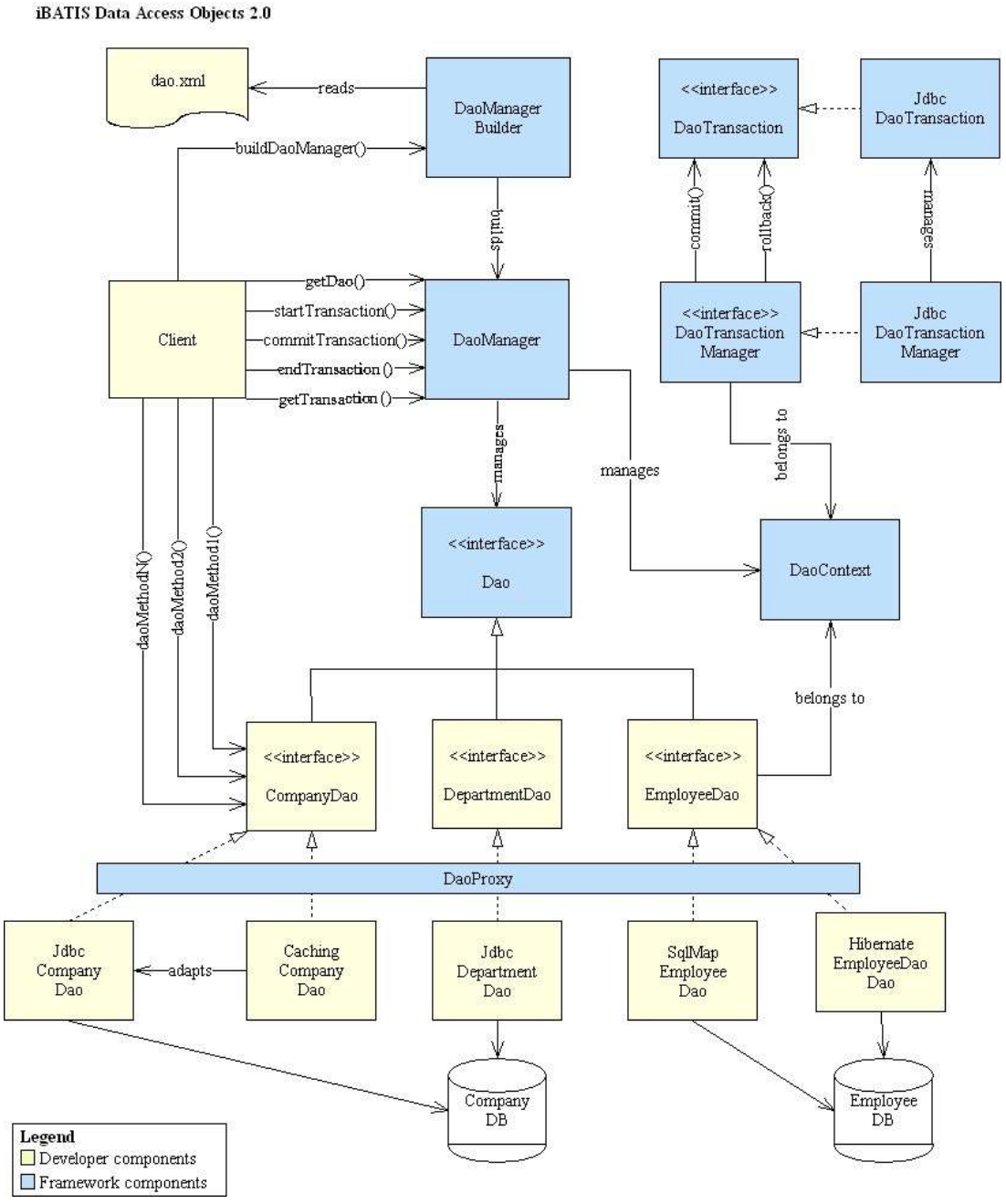
The Components of the Data Access Objects API

There are a number of classes that make up the DAO API. Each has a very specific and important role. The following table lists the classes and a brief description. The next sections will provide more detail on how to use these classes.

Class/Interface (Patterns)	Description
DaoManager (Façade)	Responsible for configuration of the DAO framework (via dao.xml), instantiating Dao implementations and acts as a façade to the rest of the API.
DaoTransaction (Marker Interface)	A generic interface for marking transactions (connections). A common implementation would wrap a JDBC connection object.
DaoException (RuntimeException)	All methods and classes in the DAO API throw this exception exclusively. Dao implementations should also throw this exception exclusively. Avoid throwing any other exception type, and instead nest them within the DaoException.
Dao (Marker Interface)	A marker interface for all DAO implementations. This interface must be implemented by all DAO classes. This interface does not declare any methods to be implemented, and only acts as a marker (i.e. something for the DaoFactory to identify the class by).

JDBC Transaction Manager Implementations

SQLMAP	Manages transactions via the SQL Maps framework and its transaction management services including various DataSource and transaction manager configurations.
HIBERNATE	Provides easy integration for Hibernate and its associated transaction facilities (SessionFactory, Session, Transaction).
JDBC	Manages transactions via the JDBC API using the basic DataSource and Connection interfaces.
JTA	Manages JTA global (distributed) transaction services. Requires a managed DataSource that can be accessed via JNDI.
EXTERNAL	Allows transactions to be controlled externally.



dao.xml –The Configuration File (http://ibatis.apache.org/dtd/dao-2.dtd)

The DaoManager class is responsible for configuration of the DAO Framework. The DaoManager is able to parse a special XML file with configuration information for the framework. The configuration XML file specifies the following things: 1) DAO contexts, 2) the Transaction Manager implementation for each context, 3) properties for configuration of the TransactionManager, 3) the Dao implementations for each associated DAO interface. A DAO context is a grouping of related configuration information and DAO implementations. Usually a context is associated with a single data source such as a relational database or a flat file. By configuring multiple contexts, you can easily centralize access configuration to multiple databases. The structure of the DAO configuration file (commonly called dao.xml, but not required) is as follows. Values that you will likely change for your application are highlighted.

```
<!DOCTYPE daoConfig
   PUBLIC "-//ibatis.apache.org//DTD DAO Configuration 2.0//EN"
   "http://ibatis.apache.org/dtd/dao-2.dtd">

<daoConfig>

 <properties resource="com/domain/properties/MyProperties.properties"/>

 <!-- Example JDBC DAO Configuration -->

 <context>

  <transactionManager type="JDBC">
   <property name="DataSource" value="SIMPLE"/>
   <property name="JDBC.Driver" value="${driver}"/>
   <property name="JDBC.ConnectionURL" value="${url}"/>
   <property name="JDBC.Username" value="${username}"/>
   <property name="JDBC.Password" value="${password}"/>
   <property name="JDBC.DefaultAutoCommit" value="true" />
   <property name="Pool.MaximumActiveConnections" value="10"/>
   <property name="Pool.MaximumIdleConnections" value="5"/>
   <property name="Pool.MaximumCheckoutTime" value="120000"/>
  </transactionManager>
  <dao  interface="com.domain.dao.OrderDao"
        implementation="com.domain.dao.jdbc.JdbcOrderDao"/>
  <dao  interface="com.domain.dao.LineItemDao"
        implementation="com.domain.dao.jdbc.JdbcLineItemDao"/>
  <dao  interface="com.domain.dao.CustomerDao"
        implementation="com.domain.dao.jdbc.JdbcCustomerDao"/>
 </context>

 <!-- Example SQL Maps DAO Configuration -->

 <context>
  <transactionManager type="SQLMAP">
    <property name="SqlMapConfigResource"
value="com/domain/dao/sqlmap/SqlMapConfig.xml"/>
  </transactionManager>

  <dao  interface="com.domain.dao.PersonDao"
        implementation="com.domain.dao.sqlmap.SqlMapPersonDao"/>
  <dao  interface="com.domain.dao.BusinessDao"
        implementation="com.domain.dao.sqlmap.SqlMapBusinessDao"/>
  <dao  interface="com.domain.dao.AccountDao"
        implementation="com.domain.dao.sqlmap.SqlMapAccountDao"/>
 </context>

</daoConfig>
```

In the example above, what we end up with is two DAO contexts. DAOs are automatically aware of which context they belong to and therefore which transaction manager to use. There can be any number of DAO contexts specified in a dao.xml. Generally a context will refer to a single specific data source.

In order to manage multiple configurations for different environments (DEVT, Q/A, PROD), you can also make use of the optional <properties> element as shown above. This allows you to use placeholders in the place of literal value elements. For example, if you have a properties file with the following values:

```
driver= org.postgresql.Driver
url = jdbc:postgresql://localhost:5432/test
```

Then in the dao.xml file you can use placeholders instead of explicit values. For example (as above):

```
<property name="JDBC.Driver" value="${driver}"/>
<property name="JDBC.ConnectionURL" value="${url}"/>
```

This allows for easier management of different environment configurations. Only one properties resource may be specified per dao.xml.

Within the context (continuing with the example above), the first context configuration states that we will be using a transaction manager implementation called “JDBC”. This particular implementation has certain properties that it needs for configuration, which are specified in the property elements in the body of the transaction manager element. Different transaction manager implementations will require different properties. See below for more.

Next in the context, is the specification of all DAO interfaces and their associated implementations. These mappings link a generic DAO interface to a concrete (specific) implementation. This is where we can see the “pluggable” nature of Data Access Objects. By simply replacing the implementation class for a given DAO mapping, the persistence approach taken can be completely changed (e.g. from JDBC to SQL Maps).

Transaction Manager Implementations and Configuration

The Transaction Manager implementation is the component that will manage the pool of transaction objects. A transaction manager is usually just a wrapper around a specific implementation of a connection pool such as a DataSource implementation. Similarly a transaction usually wraps a particular implementation such as a JDBC Connection instance.

There are currently five implementations of transaction managers that come with the framework: JDBC, JTA, SQLMAP, HIBERNATE and EXTERNAL. More details are provided for each below.

JDBC Transaction Manager

This implementation uses JDBC to provide connection pooling services via the DataSource API. There are 3 DataSource implementations supported: SIMPLE, DBCP and JNDI. SIMPLE is an implementation of the iBATIS SimpleDataSource, which is a standalone implementation ideal for minimal overhead and dependencies. DBCP is an implementation that uses the Jakarta DBCP DataSource. Finally, JNDI is an implementation that retrieves a DataSource reference from a JNDI directory. This is the most common and flexible configuration, as it allows you to centrally configure your application via your application server.

The following are examples of each of the JDBC Configurations:

```
<!-- Example iBATIS SimpleDataSource JDBC Transaction Manager -->
```

```
<transactionManager type="JDBC">
  <property name="DataSource" value="SIMPLE"/>
  <property name="JDBC.Driver" value="${driver}"/>
  <property name="JDBC.ConnectionURL" value="${url}"/>
  <property name="JDBC.Username" value="${username}"/>
  <property name="JDBC.Password" value="${password}"/>
  <property name="JDBC.DefaultAutoCommit" value="true" />
  <!-- The following are optional -->
  <property name="Pool.MaximumActiveConnections" value="10"/>
  <property name="Pool.MaximumIdleConnections" value="5"/>
  <property name="Pool.MaximumCheckoutTime" value="120000"/>
  <property name="Pool.TimeToWait" value="10000"/>
  <property name="Pool.PingQuery" value="select * from dual"/>
  <property name="Pool.PingEnabled" value="false"/>
  <property name="Pool.PingConnectionsOlderThan" value="0"/>
  <property name="Pool.PingConnectionsNotUsedFor" value="0"/>
</transactionManager>

<!-- Example Jakarta DBCP JDBC Transaction Manager -->

<transactionManager type="JDBC">
  <property name="DataSource" value="DBCP"/>
  <property name="JDBC.Driver" value="${driver}"/>
  <property name="JDBC.ConnectionURL" value="${url}"/>
  <property name="JDBC.Username" value="${username}"/>
  <property name="JDBC.Password" value="${password}"/>
  <property name="JDBC.DefaultAutoCommit" value="true" />
  <!-- The following are optional -->
  <property name="Pool.MaximumActiveConnections" value="10"/>
  <property name="Pool.MaximumIdleConnections" value="5"/>
  <property name="Pool.MaximumWait" value="60000"/>
  <!-- Use of the validation query can be problematic.  If you have difficulty, try without it.  -->
  <property name="Pool.ValidationQuery" value="select 1 from ACCOUNT"/>
  <property name="Pool.LogAbandoned" value="false"/>
  <property name="Pool.RemoveAbandoned" value="false"/>
  <property name="Pool.RemoveAbandonedTimeout" value="50000"/>
</transactionManager>

<!-- Example JNDI DataSource JDBC Transaction Manager -->

<transactionManager type="JDBC">
  <property name="DataSource" value="JNDI"/>
  <property name="DBJndiContext" value="java:comp/env/jdbc/MyDataSource"/>
</transactionManager>
```

JTA Transaction Manager

The JTA Transaction Manager manages transactions using the Java Transaction Architecture (JTA) API. This implementation always requires that the DataSource implementation is retrieved via JNDI and that a UserTransaction instance is also accessible via JNDI. Hence there is some additional setup on the server side, but it makes the configuration of the DAO framework quite simple. Here is an example configuration:

```
<transactionManager type="JTA">

  <property name="DBJndiContext" value="java:comp/env/jdbc/MyDataSource"/>
  <property name="UserTransaction" value="java:comp/env/UserTransaction"/>

</transactionManager>
```

SQLMAP Transaction Manager

The SQLMAP transaction manager implementation wraps the SQL Maps transaction management services for simple use via the DAO framework. All you need to specify is the SQL Maps configuration file. Here is an example configuration:

```
  <transactionManager type="SQLMAP">
    <property name="SqlMapConfigResource"
value="com/domain/dao/sqlmap/SqlMapConfig.xml"/>
  </transactionManager>
```

HIBERNATE Transaction Manager

Similarly, the HIBERNATE transaction manager implementation wraps the Hibernate transaction management services for simple use via the DAO framework. Basically the properties specified in the configuration are the same as those that would normally be specified in a hibernate.properties file. In addition, the persistent classes (that you would normally add to the Hibernate Configuration) are added using properties that start with "class.". Here's an example configuration:

```
  <transactionManager type="HIBERNATE">
    <property name="hibernate.dialect" value="net.sf.hibernate.dialect.PostgreSQLDialect"/>
    <property name="hibernate.connection.driver_class" value="${driver}"/>
    <property name="hibernate.connection.url" value="${url}"/>
    <property name="hibernate.connection.username" value="${username}"/>
    <property name="hibernate.connection.password" value="${password}"/>
    <property name="class.1" value="com.domain.Person"/>
    <property name="class.2" value="com.domain.Business"/>
    <property name="class.3" value="com.domain.Account"/>
  </transactionManager>
```

EXTERNAL Transaction Manager

The EXTERNAL transaction manager implementation allows for transactions to be externally controlled by the DAO framework. This implementation basically has no behavior and therefore requires no properties. You may use an EXTERNAL transaction manager if you're dealing with a flat file. Here's and example configuration:

```
  <transactionManager type="EXTERNAL"/>
```

DAO Implementation Templates

You may be wondering how this transaction manager configuration works. Well, for each of the above transaction manager implementations, there is a DAO Template to match it. The templates provide easy access to the artifacts of each implementation. For example, the JTA and JDBC templates provide simple access to the JDBC Connection object. Similarly the SQLMAP template provides access to the SqlMapExecutor instance, while the HIBERNATE template gives access to the Session object.

Using the DAO templates is completely optional, but 99% of the time, there won't be a good reason to stray from them.

More information about these templates is provided in the following section that deals with programming the DAO framework.

DaoManager -Programming

The iBATIS Data Access Objects framework has a number of goals. First, it attempts to hide the details of your persistence layer. This includes hiding all interface, implementation and exception details of your persistence solution. For example: if you're using raw JDBC, the DAO framework will hide classes like DataSource, Connection and SQLException. Similarly, if you're using the Hibernate ORM (Object Relational Mapper), the DAO framework will hide classes like Configuration, SessionFactory, Session and HibernateException. All of these implementation details will be hidden behind a consistent DAO interface layer. Even the number of data sources that you're using can be hidden from the view of the application.

The second goal is to simplify the persistence programming model, while at the same time keeping it more consistent. Different persistence solutions have different programming semantics and behavior. The DAO framework attempts to hide this as much as possible, allowing the service and domain layer of your application to be written in a consistent fashion.

The DaoManager class is responsible for configuration of the DAO framework (via dao.xml described above). In addition, the DaoManager acts as a central façade to the rest of the DAO API. Particularly it provides methods that allow you to access the transactions and DAO instances.

Reading the Configuration File

The dao.xml file is read by the static buildDaoManager() method of the DaoManagerBuilder class. The buildDaoManager() method takes a single Reader instance as a parameter, which can be a simple FileReader that points to a dao.xml file. For example:

```
Reader reader = new FileReader(C:/myapp/dao.xml);
DaoManager daoManager = DaoManagerBuilder.buildDaoManager(reader);
```

More likely though, in a Web application environment (or otherwise), it is typical to load such configuration files from the classpath. This allows the application to be moved around without having to modify properties to accommodate for the new location (i.e. achieves location transparency). This is simple to do using the com.ibatis.common.resources.Resources class that is provided with the iBATIS Database Layer. For example:

```
Reader reader = Resources.getResourceAsReader("com/domain/config/dao.xml");
DaoManager daoManager = DaoManagerBuilder.buildDaoManager(reader);
```

Contexts and the DaoManager

The DaoManager instance that is built from a dao.xml file is aware of all of the contexts contained within the configuration file. The context basically bundles DAO implementations together with a transaction manager. The DaoManager knows which DAOs and transaction managers belong to which contexts. When you request a DAO instance from the DaoManager, the proper transaction manager will be provided with it. Therefore, there is no need to ever access the context or transaction manager directly. Your DAO knows how it works. Similarly, depending on which DAOs you work with, transactions will be appropriately started and/or committed appropriately. A transaction will only be started for a context when a method is called on one of the DAOs that belong to the context. This is described in more detail below.

Getting a Data Access Object

Once you build a DaoManager instance, you can use the associated interface to retrieve the implementation (as specified in the dao.xml in the dao-factory section). Getting a DataAccess object is simply a matter of calling the getDao() method of a DaoManager instance. For example:

```
ProductDao productDao = (ProductDao) daoManager.getDao (ProductDao.class);
```

Working with Transactions

The DaoManager provides methods for working with transactions. These methods allow you to demarcate transactions and avoid having to pass transaction objects (like JDBC Connection) around to all of your DAOs. Here's an example of DAO transaction management in action:

```
ProductDao productDao = (ProductDao) daoManager.getDao (ProductDao.class);

try {
   daoManager.startTransaction();
   Product product = productDao.getProduct (5);
   product.setDescription ("New description.");
   productDao.updateProduct(product);
   daoManager.commitTransaction();
} finally {
   daoManager.endTransaction();
}
```

Calling startTransaction() lets the DaoManager know that you are interested in managing transactions programmatically. No transactions are physically started until a method is called on a Dao instance, and even then, transactions will only be started for contexts that require them. It is very important that you guarantee a call endTransaction() if you've called startTransaction(), which is why it is within the try-finally block. The call to endTransaction() will rollback any changes you've made, unless (of course) you've made the call to commitTransaction() first.

"Autocommit" -Like Behavior

In addition to programmatically demarcating transactions, you can allow the DaoManager to automatically start and end a transaction for you. This is similar to the setAutoCommit(true) behavior of the JDBC Connection class. The difference is that this not really autocommit, which means that within your DAO method, you can have multiple updates occurring and they'll all be committed as one single transaction. You don't need to do anything special to use the autocommit behavior, just don't call startTransaction(). Here's an example:

```
   Product product = productDao.getProduct (5);   // Transaction 1
   product.setDescription ("New description.");
   productDao.updateProduct(product);             // Transaction 2
   product = productDao.getProduct (5);           // Transaction 3
```

If the updateProduct() method contained more than a single update, those updates within the method definition itself would both be part of Transaction 2, so this type of "autocommit-like" semantic is much more powerful and simpler too! Notice that in the above example there is no exception handling. It is all taken care of internally to ensure that, in the event of an exception, transactions are rolled back (if necessary) and resources are released.

The next section shows you how to write a Dao like ProductDao from the example.

Implementing the Dao Interface (i.e. Creating Your Data Access Objects)

The Dao interface is very simple and very flexible because it does not declare any methods. It is intended to act as a marker interface only (as per the Marker Interface pattern –Grand98). In other words, by extending the Dao interface, all that is really achieved for the class (that implements your Dao interface) is the ability to be instantiated by DaoFactory and managed by DaoManager. There are no limitations to the methods that you use in your Dao interfaces. Although, it is recommended (for consistency) that Dao implementations only throw exceptions of type DaoException. The DaoException is a RuntimeException, so it won't interfere with your method signature, and it's nestable, so you won't lose the original exception. This is part of helping to hide the implementation details of your persistence solution.

An example of a good Dao interface is:

```
public interface ProductDao extends Dao {

        // Updates
        public int updateProduct (Product product);  // DAO Framework code may throw DaoException
        public int insertProduct (Product product);
        public int deleteProduct (int productId);

        // Queries
        public Product getProduct (int productId);
        public List getProductListByProductDescription (String description);

}
```

Templates and Implementations

As mentioned earlier in this document, there are a number of DAO Template abstract classes to help simplify the implementation of your DAOs. Each of the templates matches up with a specific Transaction Manager. Each of the templates provides a convenience method to retrieve the appropriate artifact that you typically need to interact with the persistence framework. The following table summarizes the templates:

Template Class	Transaction Manager	Convenience Method
JdbcDaoTemplate	JDBC	Connection getConnection()
JtaDaoTemplate	JTA	Connection getConnection()
SqlMapDaoTemplate	SQLMAP	SqlMapExecutor getSqlMapExecutor()
HibernateDaoTemplate	HIBERNATE	Session getSession()

The following is an example implementation using the SqlMapDaoTemplate:

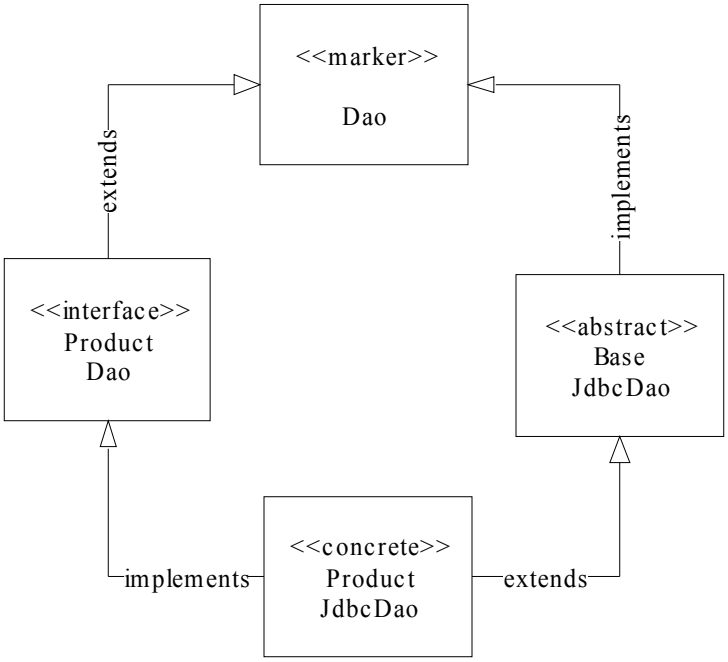
```
public class SqlMapProductDao extends SqlMapDaoTemplate implements ProductDao {

        public SqlMapProductDao (DaoManager daoManager) {
           super (daoManager);
        }

        /* Insert method */
        public int updateProduct (Product product) {
          try {
           getSqlMapExecutor().insert ("insertProduct", product);
          } catch (SQLException e) {
            throw new DaoException ("Error inserting product.  Cause: " + e, e);
          }
        }

        // … assume remaining methods to save space
}
```

Template Constructors and DaoManager Injection

Notice that in the above example, the SqlMapProductDao has only a single constructor that takes the DaoManager as a parameter. During instantiation, the DaoManagerBuilder will pass (inject) the parent DaoManager to this constructor automatically. Dao implementations are free to keep a reference to the DaoManager, as is often required to work with transactions and gain access to other DAOs. In the case of the templates, each takes the DaoManager as a parameter to the constructor, which forces all subclasses to do the same. When working with the templates, providing this constructor is a requirement. You need not supply a parameterless constructor or any other constructor. Your DAOs should only be instantiated by the DaoManagerBuilder anyway.

DAO Design Considerations

When implementing your Dao classes for your Dao interfaces, it is recommended to use a design that includes the Dao interface, an abstract (base) class and a concrete class. The advantage to having the base class is that it can contain common methods that simplify the usage of your persistence approach (e.g. wrapping up exception handling, transaction acquisition etc.). Here's an example:



Notice that both the ProductDao and BaseJdbcDao implement and extend the Dao marker interface respectively. Only one is required, but you may have different design considerations to help you go one way or the other (or both). A good approach is to allow BaseJdbcDao to implement the Dao marker and leave the ProductDao without it. Then the concrete ProductJdbcDao implements both by implementing the ProductDao and extending the BaseJdbcDao. This may allow your Product interface API to remain free of the Dao marker.

When working with Templates, you theoretically already have a similar design. However, it's recommended that you still create a base class of your own. In the above example, that would mean that BaseJdbcDao would extend JdbcDaoTemplate (which already implements the Dao marker interface).

That's all folks.

For a complete example of using the DAO framework, please visit ibatis.apache.org and download the JPetstore 5 demo application. Have fun!