org.jax.mgi.shr.dbutils.dao Design Document

Author: Mike Walker

Created: March 19, 2003

Last Modified: September 17, 2004 00:02

1 Purpose of Document

This document describes the classes belonging to the org.jax.mgi.shr.dbutils.dao package and provides source code examples for common usage patterns.

2 Introduction

The package name dao stands for Data Access Object which describes the primary interface to this package, the **DAO** interface. This interface represents the encapsulation of the overall mechanics of database persistence. The dao package includes various sql strategy classes for providing a number of different ways to update data within the database, including bcp, inline sql and jdbc batch. This is encapsulated within the **SQLStream** class.

This product integrates with the org.jax.mgi.shr.exception package, the org.jax.mgi.shr.types package, the org.jax.mgi.shr.config package, the org.jax.mgi.shr.dbutils package, and the org.jax.mgi.shr.dbutils.bcp package.

This product was coded and tested on Java 1.4 JDK.

3 Overview of Classes

3.1 SQLTranslatable

The **SQLTranslatable** interface provides three methods for performing database operations. They are getUpdateSQL(), getInsertSQL() and getDeleteSQL(). These methods simply return the sql strings required for updating, inserting or deleting.

3.2 BCPTranslatable

The **BCPTranslatable** interface is documented more fully in the org.jax.mgi.shr.dbutils.bcp package, but is being mentioned here because of it's use within the dao package. This interface provides two methods involved in writing lines to a bcp file. These are getBCPSupportedTables() and getBCPVector(Table). The getBCPSupportedTables() method returns a Vector of Strings which represent the list of tables that this interface will support. The getBCPVector(Table) method will provide a String which represents a line to a bcp file for the given table.

3.3 DAO

The **DAO** classes implement **SQLTranslatable** and **BCPTranslatable** interfaces and typically represent a row of data for a particular database table, but may also represent many rows from many database tables. The former would be referred to as a one-to-one mapping whereas the latter would be a one-to-many mapping. The cvs product called dbsgen automatically generates a series of **DAO** classes that map one-to-one with a given list of database tables. Currently the lib_java_dbsmgd and the lib_java_dbsrdr products use dbsgen to create a series of **DAO** classes for the MGD and RADAR database schemas. See the documentation for the dbsgen product and both the lib_java_dbsmgd and lib_java_dbsrdr products for more details.

3.4 InsertStrategy

The **InsertStrategy** is an interface which provides the insert(DAO) method. It is used by classes which insert **DAO** objects into the database. These classes will typically call the getInsertSQL() method on the given **DAO** class and route the sql appropriately for the strategy being implemented (for example, scripting, inline sql, jdbc batch, etc).

3.5 UpdateStrategy

The **UpdateStrategy** is an interface which provides the update(DAO) method. It is used by classes which update **DAO** objects in the database. These classes will typically call the getUpdateSQL() method on the given **DAO** class and route the sql appropriately for the strategy being implemented (for example, scripting, inline sql, jdbc batch, etc).

3.6 DeleteStrategy

The **DeleteStrategy** is an interface which provides the delete(DAO) method. It is used by classes which delete **DAO** objects from the database. These classes will typically call the getDeleteSQL() method on the given **DAO** class and route the sql appropriately for the strategy being implemented (for example, scripting, inline sql, jdbc batch, etc).

3.7 BCPStrategy

The **BCPStrategy** is a class which implements the **InsertStrategy** interface. This class has a **BCPManager** (see org.jax.mgi.shr.dbutils.bcp package documentation) and knows how to insert **DAO** objects into the database using bcp, using the DAO object's **BCPTranslatable** interface.

3.8 InlineStrategy

The **InlineStrategy** is a class which implements the **InsertStrategy**, **UpdateStrategy** and **DeleteStrategy** interfaces. This class has an SQLDataManager (see org.jax.mgi.shr.dbutils package documentation) and knows how to insert, update and delete the **DAO** object in the database with direct inline sql, using the **DAO** object's **SQLTranslatable** interface.

3.9 ScriptStrategy

The **ScriptStrategy** is a class which implements the **InsertStrategy**, **UpdateStrategy** and **DeleteStrategy** interfaces. This class knows how to insert, update and delete **DAO** objects in the database with the isql command and isql script files using a **ScriptWriter** object. It relies on the **DAO** object supporting the **SQLTranslatable** interface to support these operations.

3.10 BatchStrategy

The **BatchStrategy** is a class which implements the **InsertStrategy**, **UpdateStrategy** and **DeleteStrategy** interfaces. This class knows how to insert, update and delete **DAO** object in the database using a **BatchProcessor** object. It relies on the **DAO** object supporting the **SQLTranslatable** interface to perform these operations.

3.11 SQLStream

The **SQLStream** class is an abstract class which has a **InsertStrategy**, **UpdateStrategy** and a **DeleteStrategy** and utilizes these strategies to insert, update and delete DAO objects in the database. It has one abstract method called close() which is intended for executing any required batch processing.

3.12 Inline_Stream

The **Inline_Stream** class extends the **SQLStream** class. It has an **InlineStrategy** and inserts, updates and deletes **DAO** objects using this strategy.

3.13 Script_Stream

The **Script_Stream** class extends the **SQLStream** class. It inserts, updates and deletes **DAO** objects in the database using a **ScriptStrategy** which writes out sql to a script file. The close() method will execute the script (see org.jax.mgi.shr.dbutils package documentation for more information about the **ScriptWriter** class).

3.14 Batch_Stream

The **Batch_Stream** class extends **SQLStream** and inserts, updates, deletes **DAO** objects in the database utilizing a **BatchStrategy** object. The close() method will execute the batch operations.

3.15 BCP_Stream

The **BCP_Stream** class extends **SQLStream** and inserts **DAO** objects in the database utilizing a **BCPStrategy** object. The delete and update methods will throw an runtime exception. The close() method will execute the bcp command.

3.16 BCP_Inline_Stream

The **BCP_Inline_Stream** class extends the **SQLStream** class. It inserts **DAO** object into the database using a **BCPStrategy,** and updates and deletes **DAO** object in the database using an

InlineStrategy. The close() method will execute the bcp file. All other inserts and deletes are processed and committed in real time.

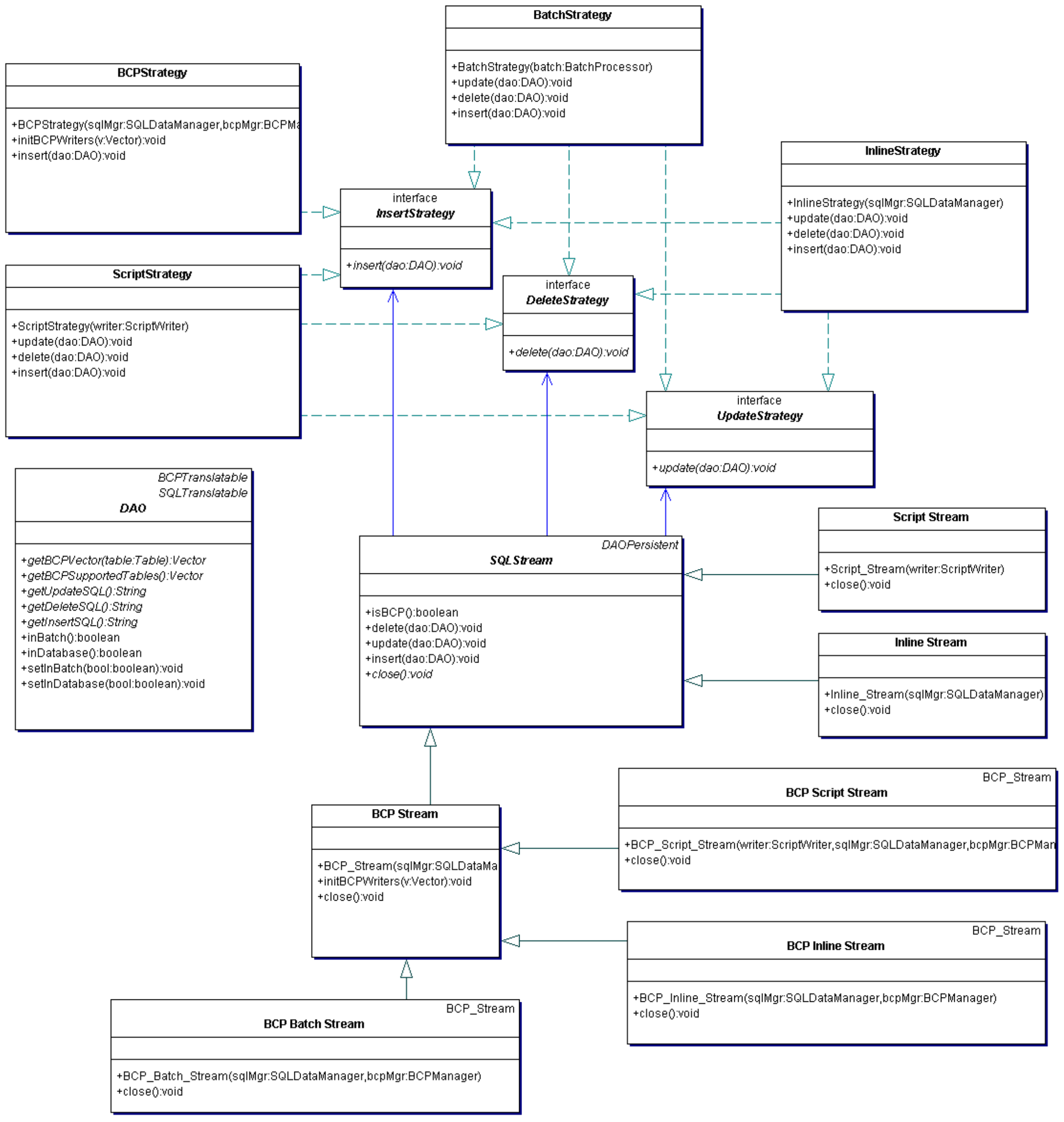
3.17 BCP_Script_Stream

The **BCP_Script_Stream** class extends the **SQLStream** class. It inserts **DAO** object into the database using a **BCPStrategy**, and updates, deletes **DAO** objects using a **ScriptStrategy**. The close() method will execute the bcp file(s) first and then subsequently execute the isql script.

3.18 BCP_Batch_Stream

The **BCP_Batch_Stream** class extends the **SQLStream** class. It inserts **DAO** object into the database using a **BCPStrategy**, and updates, deletes **DAO** object in the database using a **BatchStrategy**. The close() method will execute the bcp file(s) first and then subsequently execute the JDBC batch.

4 Class Diagram



5 Data Access Layer

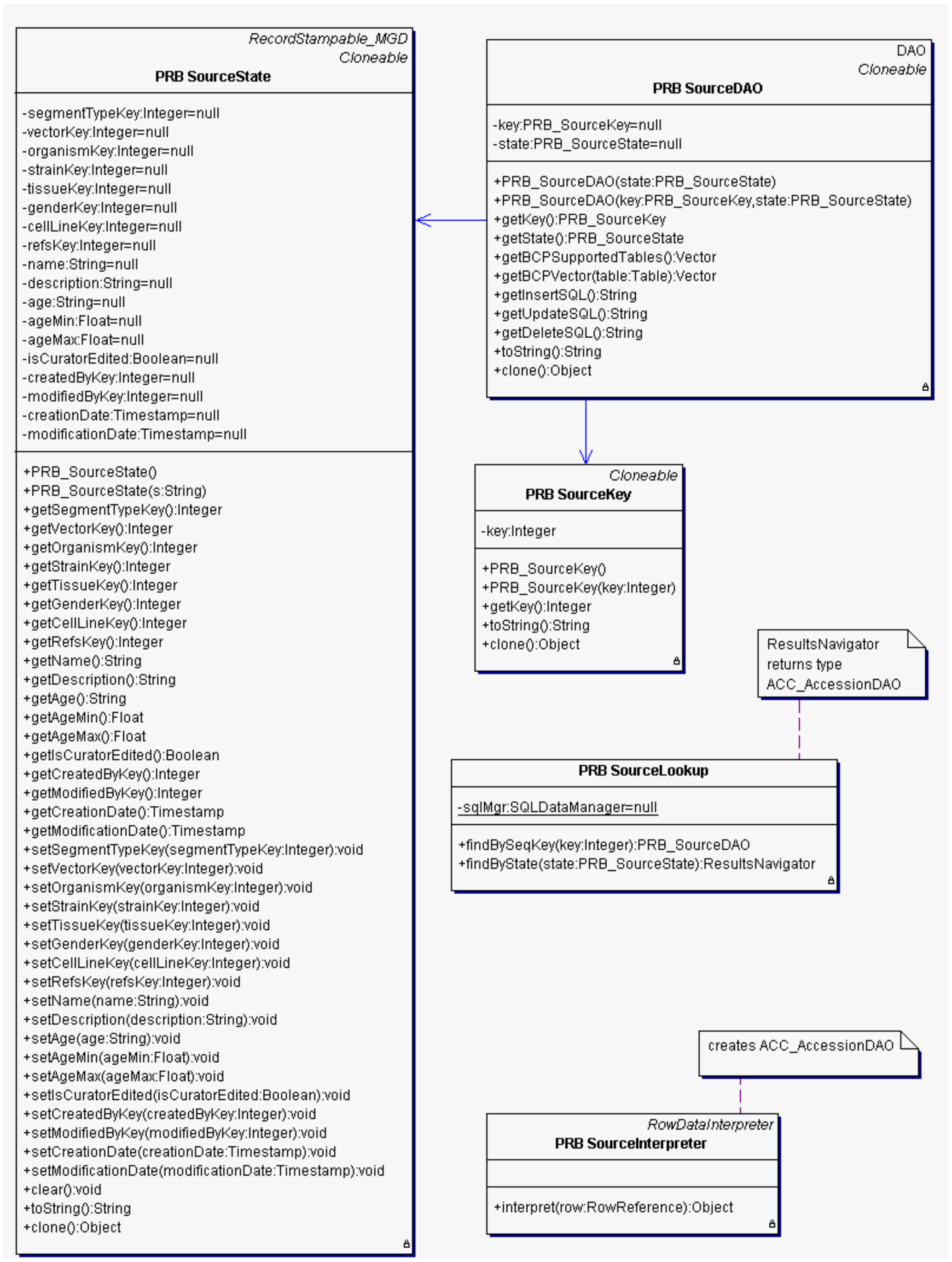
When using an Object Oriented Programming language in conjunction with a relational database, an Object Relational (OR) mapping layer around the database becomes an important reusable component central to any application frameworks. In addition to providing OR mappings, the OR layer typically provides a database persistence mechanism that is transparent to your application objects. There are many “out-of-the-box” solutions available such as the open source tool called Castor, along with commercial products such as CocoBase. And currently, there are two Java standards, Enterprise Java Beans (EJB) and Java Data Objects (JDO), which are implemented in many third party development tools, such as the open source tool, JBoss, and the commercial tool Weblogic. Eventually, the entire data access layer may become a more trivial component to application or frameworks development, but today it is far from that. There are many choices to make in regards to system design and software tools and there exist a big need for “ramp-up” time dedicated to research and prototyping when making these choices.

The **DAO** and **SQLStream** classes have become a simple in-house solution, albeit an approach which lacks the full features usually found in third party products. For example, most products provide the ability to describe object mappings either with XML or some other means, this in-house approach does not. The persistence layer of the **DAO** classes is required to be written by hand. A shortcut to creating DAO objects by hand was developed that involves automatic code generation of **DAO** classes with one-to-one mappings to the database tables. This is accomplished with the **dbsgen** product. All the generated classes provide transparent persistence to their corresponding database table. They implement the **DAO** interface and can be inserted, deleted and updated by using implementations of the **SQLStream** interface, such as **Inline_Stream**, **Script_Stream**, **Batch_Stream**, **BCP_Script_Stream**, **BCP_Batch_Stream**, and **BCP_Inline_Stream**. This product was developed in-house and documentation is available. Currently it is only possible to create a one-to-one mapping using source code generation, but these generated **DAO** classes can simplify the development of more complex mappings written by hand.

6 Code Generated DAO objects

The **dbsgen** product generates DAO objects for the MGD and RADAR databases and is integrated into the build scripts for the lib_java_dbsmgd and the lib_java_dbsrdr products. An example implementation would be the **PRB_ProbeDAO** class which provides access to the PRB_Probe table. The classes generated include the **PRB_ProbeDAO** class, the **PRB_ProbeState** class, and the **PRB_ProbeKey** class. The **PRB_ProbeKey** class was generated because the PRB_Probe table has a sequential key used at its primary index. This number is represented by the **PRB_ProbeKey** class. Not all tables have this key. So, for example the **ACCAccessionReferenceDAO** would not contain a **ACCAccessionReferenceKey** class. The other generated classes include the **PRB_ProbeInterpreter** and **PRB_ProbeLookup** class. Figure 1 shows the class diagram for these classes.

FIGURE 1. Example DAO Implementation Class Diagram


RecordStampable_MGD
Cloneable
PRB SourceState
-segmentTypeKey:Integer=null
-vectorKey:Integer=null
-organismKey:Integer=null
-strainKey:Integer=null
-tissueKey:Integer=null
-genderKey:Integer=null
-cellLineKey:Integer=null
-refsKey:Integer=null
-name:String=null
-description:String=null
-age:String=null
-ageMin:Float=null
-ageMax:Float=null
-isCuratorEdited:Boolean=null
-createdByKey:Integer=null
-modifiedByKey:Integer=null
-creationDate:Timestamp=null
-modificationDate:Timestamp=null
+PRB_SourceState()
+PRB_SourceState(s:String)
+getSegmentTypeKey():Integer
+getVectorKey():Integer
+getOrganismKey():Integer
+getStrainKey():Integer
+getTissueKey():Integer
+getGenderKey():Integer
+getCellLineKey():Integer
+getRefsKey():Integer
+getName():String
+getDescription():String
+getAge():String
+getAgeMin():Float
+getAgeMax():Float
+getIsCuratorEdited():Boolean
+getCreatedByKey():Integer
+getModifiedByKey():Integer
+getCreationDate():Timestamp
+getModificationDate():Timestamp
+setSegmentTypeKey(segmentTypeKey:Integer):void
+setVectorKey(vectorKey:Integer):void
+setOrganismKey(organismKey:Integer):void
+setStrainKey(strainKey:Integer):void
+setTissueKey(tissueKey:Integer):void
+setGenderKey(genderKey:Integer):void
+setCellLineKey(cellLineKey:Integer):void
+setRefsKey(refsKey:Integer):void
+setName(name:String):void
+setDescription(description:String):void
+setAge(age:String):void
+setAgeMin(ageMin:Float):void
+setAgeMax(ageMax:Float):void
+setIsCuratorEdited(isCuratorEdited:Boolean):void
+setCreatedByKey(createdByKey:Integer):void
+setModifiedByKey(modifiedByKey:Integer):void
+setCreationDate(creationDate:Timestamp):void
+setModificationDate(modificationDate:Timestamp):void
+clear():void
+toString():String
+clone():Object
DAO
Cloneable
PRB SourceDAO
-key:PRB_SourceKey=null
-state:PRB_SourceState=null
+PRB_SourceDAO(state:PRB_SourceState)
+PRB_SourceDAO(key:PRB_SourceKey,state:PRB_SourceState)
+getKey():PRB_SourceKey
+getState():PRB_SourceState
+getBCPSupportedTables():Vector
+getBCPVector(table:Table):Vector
+getInsertSQL():String
+getUpdateSQL():String
+getDeleteSQL():String
+toString():String
+clone():Object
Cloneable
PRB SourceKey
-key:Integer
+PRB_SourceKey()
+PRB_SourceKey(key:Integer)
+getKey():Integer
+toString():String
+clone():Object
ResultsNavigator returns type ACC_AccessionDAO
PRB SourceLookup
-sqlMgr:SQLDataManager=null
+findBySeqKey(key:Integer):PRB_SourceDAO
+findByState(state:PRB_SourceState):ResultsNavigator
creates ACC_AccessionDAO
RowDataInterpreter
PRB SourceInterpreter
+interpret(row:RowReference):Object

The **DAO** class names are derived by prefixing the class type name with the database table name represented by the **DAO** class. The table below describes these classes in more detail and uses an 'X' in place of the table name.

Table 1: DAO family of classes

XDAO	The **DAO** class which implements **SQLTranslatable** and **BCPTranslatable**. Objects of this type can be passed to the insert, update and delete methods of **SQLStreams**.
XState	The java bean class which represents a row of data from the represented database table. There are corresponding setter and getter methods for these attributes.
XKey	The key class for a table which contains a incremental numeric key. This data is separated from the State class data since it has no biological function and is only used to uniquely identify records within the database tables. This class can automatically define the next key value for the represented table. This class is not generated for tables which do not have this type of primary key defined.
XInterpreter	This class implements the **RowDataInterpreter** interface and allows **DAO** objects to be created from result set data.
XLookup	This class provides two methods for accessing **DAO** objects from the database. The findByKey(Integer) is a lookup by integer key. The findByState(XState) method allows looking up by attribute values. This method is called by creating a new **XState** object, set the attributes you want to search on and then pass this object to the lookup method. A **ResultsNavigator** is returned which allows navigating over the X objects.

7 XInterpreterExample

The **XInterpreter** class mentioned above implements the **RowDataInterpreter** interface from the org.jax.mgi.shr.dbutil package. The interpret(RowReference) method accesses values from a **RowReference** that pertain to the X table. Each column value is accessed by the column name and the interpreter object can access values from any result set generated from queries where column aliasing is not used. If you design a query to access a complex assortment of data from many tables, then result sets from these queries can be passed to the interpret(RowReference) methods of various XInterpreters to obtain the individual **DAO** objects represented within the query results. For example, if there was an object you were designing to represent a bibliography reference, then you may design a BibRef object and write a lookup method that can create new BibRef objects by querying the database:

```
select acc_accession.*, bib_refs.* from acc_accession, bib_refs
where...
```

This query will generate a **ResultsNavigator** when passed to the executeQuery method of the **SQLDataManager** and the **ResultsNavigator** class iterates over **RowReference** objects for each row of data. These RowReference objects can be passed to the interpret methods as is shown in example 2.

EXAMPLE 2. using DAO Interpreters

```
ACC_AccessionDAO acc =
        ACC_AccessionInterpreter.interpret(row);
BIB_Refs ref =
        BIB_RefsInterpreter.interpret(row);
BibRefs bibRefs = new BibRefs(acc, ref);
```

8 XLookup Example

The XLookup class from the table above, is used to obtain **DAO** objects for a given table. The findByKey method obtains a single **DAO** object represented in the database by the given incremental primary key. If the table does not have an incremental primary key, then the findByKey method is not created when using automated code generation through the dbsgen product. If you knew the key to the record you were interested in then you could obtain a **DAO** object for the record as is shown in example 3 which uses the **ACC_AccessionLookup** class.

EXAMPLE 3. using findByKey methods

```
ACC_AccessionDAO acc =
     ACC_AccessionLookup.findByKey(keyValue);
```

The findByState method allows looking up by attributes and this method potentially returns more than one **DAO** object. The return type is a **ResultsNavigator** and you can iterate over all the returned **DAO** objects by using this object. Example 4 shows a query for all accessions records for a given accession id.

EXAMPLE 4.

```
ACC_AccessionState state = new ACC_AccessionState();
state.setAccid(accid);
ResultsNavigator nav = ACC_AccessionLookup.findByState(state);
while (nav.next())
  ACC_AccessionDAO acc = (ACC_Accession)nav.getCurrent():
```