NEC

The dbMapper Package

An Object-Relational Mapping Tool That Provides DAO Functionality For Relational Database Persistence

A White Paper

November 2002

Abstract: This document provides an overview of the features and concepts of a Java package called the dbMapper package. This package implements a Data Access Object (DAO) pattern that allows an application programmer to execute the typical create, retrieve, update, and delete (CRUD) operations on a relational database without writing SQL code. The package accomplishes this by using XML configuration files that specify the object-relational (OR) mapping of Java classes together with the Java reflection API to generate the necessary SQL statements "on the fly". This greatly reduces the effort needed to program the typical CRUD operations used by an application. This package also uses the JDBC interface so that it is portable across any database that implements JDBC. This package was developed by the software development group of the TDD division of NEC America.

Prepared by:
ONSD Software Group
14040 Park Center Road, Herndon, VA 20171
NEC America

Email: onsd@necam.com
Web: http://www.onsd.nec.com/software

1 Introduction

1.1 Background

In recent years, as the number of applications that use databases has exploded, much more attention has been devoted to the development of standardized interfaces and patterns for accessing databases from applications. Two important developments have been the specification of JDBC, and the formulation of Data Access Object (DAO) patterns.

The specification of the JDBC interface provides a standardized way for an application to communicate with and execute operations in a relational database. In fact, nearly all vendors of relational databases now support this interface. As a result, an application that is written using the JDBC interface may use any JDBC-compliant database. This is a tremendous advantage, as the application becomes portable across a large number of database products. Together with the portability of Java, this portability is extended to include multiple operating systems.

As shown in Figure 1, while the JDBC specification deals primarily with the interface to a relational database, the DAO pattern deals with the interface to the application code that deals with the business logic. The primary goal of the DAO pattern is to decouple the means of persistence from the application code as much as possible. By separating the code that deals with persistence from the business logic, the business logic does not need to be changed, even when the method of persistence changes.

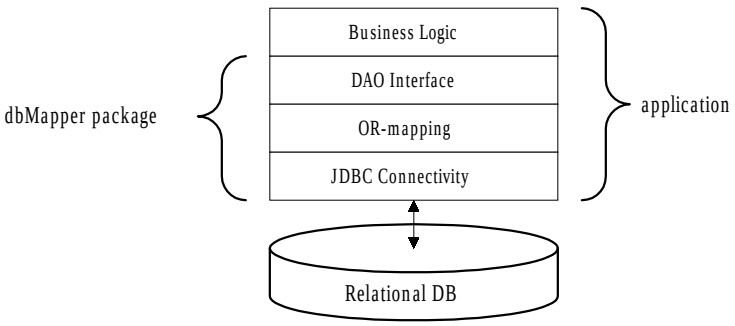

Figure 1 Application using the dbMapper package

Different versions and implementations of the DAO pattern exist. In some versions of the DAO pattern, including those first published, a DAO class is defined for each class to be persisted. This means that as new classes are added to an application, new DAO

classes must be added to the module that implements the persistence. In large systems, this can lead to the undesirable proliferation of classes. Some toolkits that implement DAO patterns automatically generate the code for the supporting DAO classes, which does reduce the workload on the application programmer. However, this does not ease the problem of class proliferation. This pattern also has implications for system maintenance, as new DAO classes must be deployed each time that the business logic classes that they support are added or modified. On the other hand, some DAO patterns eliminate the need to define individual DAO supporting classes. Refer to the article entitled "Write once, persist Anywhere"[1] for an example of such a pattern, as well as a discussion of the benefits of such a pattern.

In light of the above discussion, when implementing a database interface module for your application there are clear benefits associated with using JDBC and a DAO pattern, particularly one that minimizes class proliferation. In this white paper, we discuss the dbMapper package, which provides such an implementation. This package is offered as alternative to spending a large amount of time and effort to write database interface code. As you will see, by including the dbMapper package in your application, and by specifying the object-relational mappings of your business class in XML mapping files, your application can execute the typical create, retrieve, update, and delete (CRUD) operations for your business objects without writing a single line of SQL code, nor defining any new classes.

1.2 Features

The dbMapper package provides powerful functionality to Java applications that interact with a JDBC-capable relational database. By providing various interfaces and classes that implement a type of Data Access Object (DAO) pattern, the dbMapper package eliminates the need for an application to write any SQL statements to perform the typical create, retrieve, update, and delete (CRUD) operations on a relational database. The following list outlines some of the main features and benefits associated with the dbMapper package.

- Eliminates the need to write SQL statements to perform typical CRUD operations on a relational database.
- Works with any relational database that supports JDBC.
- Allows the application to connect to multiple databases. The object-relational mapping for a given class may be different for different databases, if desired.
- Gives the user complete control over which attributes of a class are persisted.
- Supports complex attributes (i.e. data members of a class) such as attributes that are themselves objects, as well as attributes that are arrays or collections.
- Allows the user to work with simple or composite keys.
- Is designed to work well with multi-threaded applications.

[1] "Write once, persist anywhere" by James Carman, published in Java World, March 2002, http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-dao_p.html

- Supports transactions.
- Does not require a proliferation of new classes (as some DAO implementations do).
- Allows the user to override default behavior with custom SQL when necessary.
- Employs various algorithms to provide high performance.
- Provides useful classes for managing database connections, including database connection pooling.

2 Using the dbMapper package

This section provides information related to the usage of the dbMapper package.

The dbMapper package usage can be conceptually divided into two categories: creating the necessary configuration files, and writing source code to execute the actual database operations. You will see that once the configuration files are set up, it is easy to implement code that performs database operations.

First, we discuss the configuration files, of which there are three types. Each of these configuration files is written in an XML format.

- **Data source file**. A data source file provides the information that is needed to connect to a relational database. This includes information such as the database URL, and the user name and password.
- **Object-relational mapping file**. An object-relational mapping file specifies how user-defined Java classes are mapped to a relational database. For example, a file may specify that for the class `Point`, the fields `x` and `y` map to the columns `coord_x` and `coord_y` of the `point_table` of a relational database.
- **Dbmapper configuration file**. The dbmapper configuration file defines one or more mapping contexts. A mapping context is simply the association of a data source and an object-relational mapping. A mapping context is used by the application to provide persistence for a set of Java classes to a particular database.. Note that each of these mapping contexts contains enough information to allow the dbMapper package to apply the typical CRUD operations to instances of the specified classes.

Once the configuration files have been created, the application code needed to perform the typical CRUD operations is quite simple. For example, assume that we have the appropriate configuration files setup to define a mapping context named "point_mapping". This mapping defines the mapping for a user-defined class called `Point`. We will assume that this mapping context is defined in a dbmapper configuration file named "dbmapper.xml". In this case, a `Point` object may be created, updated, and deleted from the database using only a few lines of code:

```
Point point = new Point(50,100)   // create a point with cords (50,100)
DBModule dbm = DBModule.init ("./dbmapper.xml");  // initialize module
DBInterface dbif = dbm.createDefaultMapper ("point_mapper"); // get mapping for Point
dbif.create(point);       // save point to the database
point.setX(51);           // change the value of the x coord
dbif.update(point);       // save the updated point to the database
dbif.delete(point);       // remove point from the database
```

For the sake of completeness, and to give you an idea of what type of information is entered into the configuration files, the configuration files associated with this example are provided in Appendix A.

2.1 The DBInterface interface

From the example above, you can see that `DBInterface` is a key component of the dbMapper package. This interface encapsulates all of the typical CRUD operations that an application might use, and presents them to the application in the object-oriented view of the Java language. A quick review of the methods of this interface, summarized in the following list, will provide you with a good overview of the operations provided by the dbMapper package. Comments to the right of the method names give a brief explanation of the methods. (For more complete explanations, refer to the dbMapper User's Guide.)

create methods

```
create()                // write an object to the database, simple attributes only
createTree()            // write an entire object containment tree to the database
```

delete methods

```
delete()                // delete an object from the database
deleteByAttributes()    // delete objects with certain attribute values
deleteByPrimaryKey()    // delete objects with certain key values
```

update methods

```
update()                // update an object in the database, simple attributes only
updateTree()            // update an entire object containment tree in the database
```

finder methods

```
findAll()                       // get all objects of a specific class from the database
findAllPrimaryKeys()            // get all primary keys for a specific class
findByAttributes()              // get all objects that match certain attribute values
findByPrimaryKey()              // get the object for the specified key
findByQuery()                   // get a set of objects using a user-defined SQL query

findPrimaryKeysByAttributes()   // get a set of keys for objects that match certain attribute values
findPrimaryKeysByQuery()        // get a set of keys using a user-defined SQL query
```

other (custom SQL) methods

```
executeQuery()          // execute an SQL query, and return the result set
executeUpdate()         // execute an SQL INSERT, UPDATE, or DELETE statement
```

transactional methods

```
beginTransaction()      // begin a transaction
commitTransaction()     // commit a transaction
rollbackTransaction()   // rollback (cancel) a transaction
isActiveTransaction()   // determine if the current thread executing a transaction
```

Although the application programmer only needs to use the DBInterface itself, note that the DefaultMapper class itself is the actual workhorse of the dbMapper package. The DefaultMapper class, which is obtained by the createDefaultMapper() method, uses all of the information provided in the XML configuration files to provide the functionality represented by DBInterface. Or in other words, all of the object-relational mapping, etc., is handled by the DefaultMapper class and other internal support classes. This detail is transparent to the application programmer who uses the dbMapper package.

3 Conclusion

The dbMapper package allows an application to persist Java objects in relational databases. The design of this package is such that the application programmer does not need to deal with SQL programming or database connection programming. The only input needed by the application programmer is a set of configuration files that define the data source and object-relational mappings to be used.

The design of the dbMapper package employs an efficient DAO pattern that does not require additional DAO classes to support new user-defined business classes. Instead, the application programmer merely creates or edits existing XML files to describe the object-relational mappings to be used for the new classes. The dbMapper package uses these mappings together with the Java reflection API to internally generate SQL statements as needed to support a generic interface that is used by the application programmer for all database operations.

The purpose of this document has been to provide some general information about the dbMapper package to help the reader decide if this package might be of use to his/her own projects. To gain a more detailed understanding of the dbMapper package, the "dbMapper User's Guide" is recommended. For this and other information related to the dbMapper package, please contact NEC America at onsd@necam.com, or visit our website at www.onsd.nec.com/software.

Appendix A. Example configuration files

This appendix contains XML configuration files that support the simple example described in *Section 2, Using the dbMapper package*. While these example files are not representative of the full power and flexibility of the dbMapper package, they are provided to give the reader a rough idea of the type of information that must be specified. Refer to the "dbMapper User's Guide" for a full explanation of the format and content of these files.

This example is based on the `Point` class, which is defined as follows:

```
class Point {
        int x;
        int y;    }
```

Listing for the **dbmapper.xml** configuration file, which defines the `point_mapper` mapping context:

```
<?xml version="1.0"?>
<!DOCTYPE root SYSTEM "./dbmapper.dtd">
<root>

<!—Specify the data sources to be used. -->
<data_sources>
   <data_source id="default_ds">
      <basic_data_source
         connection_info_file="./db_connection.xml"
      />
   </data_source>
</data_sources>

<!—Specify the object-relational mappings to be used. -->
<or_dbmappers>
   <or_dbmapper id="point_mapper" data_source_id="default_ds">
      <or_mapping_files>
         <or_mapping_file> point_or_mapping.xml </or_mapping_file>
      </or_mapping_files>
   </or_dbmapper>
</or_dbmappers>
</root>
```

Listing for the **point_or_mapping.xml** file, which defines the object-relational mapping for the `Point` class:

```
<?xml version="1.0"?>
<!DOCTYPE mappings SYSTEM "./or_mapping.dtd">
<!-- Set DTD validation file -->

<mappings>

<!-- Bind the Point class to point_table -->
<mapping class = "com.nec.tdd.tools.dbMapper.samples.Point"
       table="point_table" >
      <!-- bind java int x to sql column coord_x:int (indicate x as key
           attribute -->
      <field name="x" is_key="true" >
                <basic_type column="col_x">int</basic_type>
      </field>
      <!-- bind java int y to sql column coord_y:int (indicate y as key
           attribute also -->
      <field name="y" is_key="true" >
                <basic_type column="col_y">int</basic_type>
      </field>
</mapping>

</mappings>
```

Listing for the data source file, **db_connection.xml**:

```
<?xml version="1.0"?>
<!DOCTYPE connection_info SYSTEM "./db_connection.dtd">

<connection_info name="oracle" engine="oracle">
     <driver> oracle.jdbc.driver.OracleDriver </driver>
     <url> jdbc:oracle:thin:@143.102.32.169:1521:db </url>
     <user_name> joe </user_name>
     <password> secret </password>
</connection_info>
```