Applying Design Patterns to JDBC: Building a Lightweight Object-Relational Mapping Framework

FRANK SAUER

LET’S FACE IT: JDBC is here to stay. We would all like to use Container Managed Persistence (CMP) entity beans for their simplicity, but for lack of standardization or performance, we are forced to write Bean Managed Persistence (BMP) entity beans or even session beans accessing a database directly. Although JDBC is the vehicle to do it, JDBC code can get very messy, very quickly. In this article, I present a framework that uses a number of very simple design patterns to construct a lightweight object-relational mapping framework that is easy to use and abstracts the complexity from your JDBC code. At the same time, it doesn’t limit what you can and cannot do in your mapping. You still write your own SQL statements, but the framework takes the pain out of doing so and eliminates all the repetitive code you would normally have to write to execute your statements.

If you’re reading this, you’ve probably written JDBC code in the past and know that you need to obtain a JDBC Connection, create a PreparedStatement, fill in the parameters on the PreparedStatement, and finally execute the statement. After executing the statement, you would typically examine the ResultSet and construct some object(s) based on values obtained from the database. After you’re done, you have to ensure you release any resources, such as the PreparedStatement and the Connection. Every time you need to execute a JDBC statement, you must perform all these steps. That’s a lot of repetitive code!

Removing Repetitive Code

The first step in making JDBC simpler is to factor out all...
the code related to obtaining a connection and ensuring the connection gets closed when the statement is completed. This is the task of the AbstractSqlOperation class. See Figure 1 for a UML class diagram of the entire framework. AbstractSqlOperation is based on the simplest of design patterns, namely the Template pattern.2 The public execute() method first obtains a JDBC connection by calling getJDBCConnection(), then calls the abstract protected execute(Connection) method to defer the real execution of the statement to a concrete subclass. After the subclass has performed its work, the execute() method closes the connection. The connection is closed in a finally-block to ensure the connection gets closed on successful completion of the execution, and on any exceptional conditions, such as the subclass throwing a SQLException. Note, in the environment of an Enterprise JavaBean (EJB), it is perfectly legitimate to close the JDBC connection after every statement execution, since JDBC connections are usually obtained from a JDBC connection pool. To make optimal use of the connection pool, this is a recommended practice. Closing a connection obtained from a connection pool simply places the connection back into the pool for future use. It is not really closed. Figure 2 shows a UML sequence diagram of the interaction between a client of the framework, AbstractSqlOperation, and a concrete subclass of AbstractSqlOperation (the AbstractSqlOperation code listing, available in the code section at www.javareport.com, shows the implementation of AbstractSqlOperation). Note, the code to actually acquire a JDBC Connection is encapsulated behind an interface called JDBCConnector. This allows an application developer to plug in a different connection mechanism without having to change or subclass AbstractSqlOperation. This is an example of the Strategy pattern.2 [Refer to the code listings available in the code section at www.javareport.com on JDBC Connector and its default implementation, DefaultJDBCConnector.] To execute a query, you simply create a subclass of AbstractSqlOperation, implement execute(Connection), and call execute().

Continuing the Big “Clean-Up”

So far, so good. We now no longer have to deal with JDBC connections every time we want to execute a JDBC statement. However, we still have to create the PreparedStatement, fill in the parameters, and process the resulting values in the ResultSet. How can a framework possibly eliminate the need for all this? It doesn’t know what queries to execute or what the parameters are, and how can it know what to do with the result? It turns out that encapsulating most of this in a framework is not as hard as it sounds.

First, let’s assume we always want the result of a query to be some object or list of objects, and the values we want to write to the database originate in objects. To get access to those values without having to know anything about those objects, we use a simple Java interface called Persistable (see Listing 1). Persistable is a simple interface with a getProperties() and setProperties(Map) method. Any object can be written to the database (or can be created from values read from the database) by simply implementing these two methods. The result of getProperties() is a Map object (for example, a HashMap) that contains the values for the object keyed by a name. The name is assumed to be the column name as defined in the database.

Here is an example implementation for getProperties() in a Customer class that is defined to implement both Persistable and SQLConstants (see the “SQLConstants Interface” sidebar on page 40):

```java
public class Customer implements Persistable {
    private String name;
    private int id;

    // constructors, getters, setters, etc.

    public Map<String, Object> getProperties() {
        Map<String, Object> props = new HashMap<>();
        props.put("name", name);
        props.put("id", id);
        return props;
    }
}
```

Figure 1. UML sequence diagram for AbstractSqlOperation.
Map getProperties() {
    HashMap m = new HashMap();
    m.put(CUSTOMER_ID,
          new BigDecimal(id));
    m.put(CUSTOMER_NAME, name);
    m.put(CUSTOMER_DISCOUNT,
          new BigDecimal(discount));
    return m;
}

By implementing the SQLConstants interface, the Customer class gets access to all the constants defined in it, such as all the column names. I use BigDecimal to deliver numeric values to avoid dependencies on any particular database and column definitions. I’ve found that using other Java types, such as the primitive wrappers like Integer and Float, works sometimes but not always. BigDecimal always seems to work properly. How the type of the objects returned from getProperties() is used to populate the parameters of a PreparedStatement will be discussed later in this article.

The setProperties(Map) method is used by the framework to populate new objects with values obtained from the database. Implementations read values from the map and store them in instance variables, not surprisingly, the reverse process of getProperties().

Now that we have a mechanism in place to get to all the property values of an arbitrary object (as long as it implements Persistable), let’s examine how the framework uses Persistable to automatically create a PreparedStatement and fill in its parameters.

### Listing 1

**Persistable interface.**

```java
package com.trcinc.infrastructureservices.jdbc;
import java.util.*;

/**
 * Every Persistable must be able to provide a Map
 * containing the properties of the object and a List
 * of property names that make up the Primary key.
 * Poor man’s ORM... Used by DefaultSqlOperation,
 * @author Frank Sauer
 * @version 1.0
 * @see DefaultSqlOperation
 */
public interface Persistable {
/**
 * Get a list of properties representing the persistable
 * domain object. The keys are DB column names to
 * which the values will be written.
 */
    public Map getProperties();

/**
 * Set the properties of this persistable domain object
 * from the values contained in properties.
 */
    public void setProperties(Map properties);
}
```

### DefaultSqlOperation

DefaultSqlOperation is a concrete implementation of AbstractSqlOperation that maps to and from Persistable objects. A DefaultSqlOperation is constructed with a query template and a list of Strings representing the parameters. The names used for the parameters are the same column names used by the implementations of the Persistable interface. I define this array of strings in the same way I define all the query templates, namely in the SQLConstants interface. Listing 2 is an example of an insert statement into a Customer table.

```java
public interface SQLConstants {
    final static String CUSTOMER_TABLE = "customer";
    final static String CUSTOMER_ID = "id";
    final static String CUSTOMER_NAME = "name";
    final static String CUSTOMER_DISCOUNT = "discount";
    final static String findCustomerByPK =
        "SELECT " +
        CUSTOMER_ID + ", " +
        CUSTOMER_NAME + ", " +
        CUSTOMER_DISCOUNT +
        "FROM " + CUSTOMER_TABLE +
        "WHERE " + CUSTOMER_ID + " = ?";
}
```

This seems highly inefficient with all the String concatenations, but since these are final static String definitions, the concatenations are resolved at compile time, not run time. Defining each table and column in its own variable allows for quick and easy modification of the code whenever the database schema changes. The column names are also used in the implementations of the Persistable interface methods.

### Listing 2

```
The insertCustomerColumns array defines the order of the parameters of the insertCustomer statement. DefaultSqlOperation uses the array to obtain values from its Persistable and fill in the parameters of the PreparedStatement in the correct order. Now that we have determined the order of the parameters, we must overcome one last problem before we can fully automate the creation of the PreparedStatement. We have to figure out what methods to call on the PreparedStatement to set the values for the parameters. The correct method is based on the type of the parameter. To set a string value, we have to call setString, to set a numeric value we have
Figure 1 shows an example DefaultSqlOperation subclass called UpdateCustomer. As you can see, UpdateCustomer extends DefaultSqlOperation and implements SQLConstants. The UpdateCustomer constructor takes a single Customer as its argument and uses it to set the Persistable. It can do this because Customer implements Persistable. The constructor also passes the query template and column name array (obtained from SQLConstants) to the superclass’ constructor. At this point, the UpdateCustomer statement is ready to be executed. Note, by “objectifying” every JDBC statement into an object, we’re effectively implementing the Command pattern.\(^2\)

Figure 4 shows a UML sequence diagram of the entire creation and execution process of the UpdateCustomer state-
Listing 3

Initial set of mappers initialized.

```java
protected Mapper getMapper(Object key) {
    if (mappers == null) {
        mappers = new HashMap();
        registerMapper(Byte.class, new ByteMapper());
        registerMapper(Short.class, new ShortMapper());
        registerMapper(Integer.class, new IntMapper());
        registerMapper(Long.class, new LongMapper());
        registerMapper(Float.class, new FloatMapper());
        registerMapper(Double.class, new DoubleMapper());
        registerMapper(Decimal.class, new NumericMapper());
        registerMapper(BigDecimal.class, new NumericMapper());
    }
    return (Mapper)mappers.get(key);
}
```

Creating Objects From a ResultSet. Even using an AbstractSqlOperation subclass, it is still a lot of work to iterate over the values returned in a ResultSet and construct objects based on these values. What if the framework could eliminate this repetitive code as well? As you might have guessed by now that it can. We have already seen that the Persistable interface has a setProperties(Map) method that takes a Map of objects as its argument. It seems logical that the framework uses this method to populate objects with the values obtained from a ResultSet. It does, but there are a couple of issues left to resolve. The first issue is that the framework does not know what concrete objects to create. All it knows is that those objects implement Persistable, but that is not enough to actually instantiate these objects. The next issue is similar to the mapping problem we saw when filling in the parameters of the PreparedStatement. The framework has to map the values in the ResultSet back to the appropriate Java types before it can store them in the Map object given to the Persistable, but how does it know what those types are?

The first issue is resolved by not resolving it at all! As you can see in Figure 1, DefaultSqlOperation has a method named getNewPersistable(). This is a Factory Method\(^2\) that subclasses can override to deliver new instances of some Persistable implementation to the framework. Every time the framework needs to populate a new Persistable with data from a ResultSet, it will call this method. This technique defers the instance creation to the application class, where the needed knowledge about domain objects is available. DefaultSqlOperation has a DefaultPersistable inner class it will use if the subclass does not override this method. DefaultPersistable is simply a wrapper on a HashMap that implements the Persistable interface (see the DefaultSqlOperation code listing online at [www.javareport.com](http://www.javareport.com)).

The second issue brings us back to the Mapper interface discussed previously. It was used to call the correct method on a PreparedStatement to set a parameter value. The same Mapper interface also has a getValue method, which we use here to convert a value from a ResultSet to a Java type. When the PreparedStatement has finished execution, a ResultSet contains records with values that need to be obtained using getString(index), getBigDecimal (index), etc. To find out what the column names and the column types are, we use the ResultSet's meta data object,
Applying Design Patterns to JDBC

Handling Relationships

Now, let’s look at a slightly more complex example where a Customer object has a one-to-many relationship with Address objects. The Customer’s addresses are stored in a separate Address table. The Address table has a CUSTOMERNO column that links each address with a customer in the Customer table. The Customer object has a List of Address objects. How can we use the framework to insert a new address into the address table and associate it with an existing customer?

If you take a close look at the code listing for DefaultSqlOperation (available online at www.javareport.com) you will see that DefaultSqlOperation does not call its Persistable's getProperties() method directly. This is encapsulated in a protected getProperties() method on DefaultSqlOperation itself, giving subclasses an opportunity to modify the resulting Map object before passing it to the framework. This technique enables us to handle relationships as in this example. The same technique can be used to populate columns in the database that are not actually mapped to the domain objects, for example a LAST_UPDATED column. Listing 4 provides an example that inserts a new address into the address table and relates it to the customer.

Using the Database to Generate Primary Keys

Often, it is useful to let the database generate primary keys for your data rather than figure out how to create a unique key for each row in a table. Oracle has a mechanism for this, which is called Sequence. You can define a Sequence for each table and obtain the next value with a simple query. The values are automatically incremented by Oracle. The framework contains a simple subclass of AbstractSqlOperation called OracleSequence that encapsulates this use of sequences. It has a getNextValue() and getCurrentValue() method you can use to query the Sequence. It is created with the name of an OracleSequence. OracleSequence is shown in Listing 5.

Represented by a ResultSetMetaData object and obtained from the ResultSet using getMetaData(). ResultSetMetaData gives us the column names with getColumnName(index), the column type with getColumnType(index), and the total number of columns with getColumnCount(). With all this information, we are ready to construct Persistable objects from the data values in the ResultSet. Recall that the Mapper objects are registered with a Class object, as well as an integer SQL type. With the result of ResultSetMetaData's getColumnTypes[index], we can find the correct Mapper instance for that column. We will use the result of getColumnName(index) as the key in the Persistable's Map object. The UML Sequence Diagram in Figure 5 outlines this process. I assumed that all columns were of type String to simplify the diagram. Now, we can write a simple select statement as:

```
public class GetAllCustomers
    extends DefaultSqlOperation
    implements SQLConstants {

    public GetAllCustomers() {
        // no parameters in query
        super(getCustomerQuery,
             new String[]{});
    }

    Persistable getNewPersistable() {
        // Customer implements Persistable
        return new Customer();
    }
}
```

The result of execute() will be a list containing Customer objects. As you can see, we now only have to write the actual SQL statements in the SQLConstants interface and very simple subclasses of DefaultSqlOperation to execute them. No more boring, repetitive, and error-prone code to execute your JDBC statements!

Figure 4. UML sequence diagram for creation and execution of DefaultSqlOperation.

Figure 5. Processing a ResultSet.
Combining Statements to Execute On a Single Connection

It is often critical to execute multiple JDBC statements within the context of a single transaction. In an EJB using container-managed transactions, we don’t always want to execute each query on a new connection, even if it is obtained from a connection pool. For this purpose, the framework contains a class called CompoundSqlOperation. This class, together with its superclass AbstractSqlOperation, forms a Composite pattern. In other words, CompoundSqlOperation aggregates a number of AbstractSqlOperation objects and the execute(Connection) implementation of CompoundSqlOperation simply executes each contained AbstractSqlOperation on its connection. Listing 6 shows the implementation of CompoundSqlOperation. Listing 7 uses this technique in the context of a simple BMP entity bean.

The customer information is stored in three different tables in the database, and information has to be inserted in each of these. The rows have to be associated through a foreign key. Listing 7 also shows the use of OracleSequence to generate a new primary key for the new customer.

Conclusion

As I have hopefully demonstrated in this article, JDBC code can be dramatically cleaned up using some simple design patterns. It is possible to write clean JDBC code by applying some simple refactoring of repetitive code and the appropriate design patterns. The framework presented here uses the Command, Composite, Strategy, and Template patterns to eliminate the writing of JDBC code.

I have used this framework on various EJB-based projects requiring BMP, and in an internal training course for new Java developers. Future work on this framework could include an extension that reads the SQL statements from an external file (XML file, for example) instead of hard-coding the SQL in the SQLConstants interface. That way, SQL can be modified without requiring a recompilation of your code. Parsing an XML file and constructing the framework objects to execute the statements should be a relatively straightforward exercise.
LISTING 6

**CompoundSqlOperation.**

```java
package com.trcinc.infrastructureservices.jdbc;
import java.sql.*;
import java.util.*;

/** CompoundSqlOperation allows for multiple AbstractSqlOperations
 * to be executed on the same JDBC connection. It has three
 * convenience constructors taking a number of
 * AbstractSqlOperations to be combined and has an add method to
 * add more.
 * @author Frank Sauer
 * @version 1.0
 */
public class CompoundSqlOperation extends AbstractSqlOperation {
    private List operations = new ArrayList();

    /**
     * Creates new CompoundSqlOperation containing the single
     * AbstractSqlOperation op1
     */
    public CompoundSqlOperation(AbstractSqlOperation op1) {
        add(op1);
    }

    /**
     * Creates new CompoundSqlOperation containing the two
     * AbstractSqlOperations op1 and op2
     */
    public CompoundSqlOperation(AbstractSqlOperation op1,
                                 AbstractSqlOperation op2) {
        add(op1);
        add(op2);
    }

    /**
     * Creates new CompoundSqlOperation containing the three
     * AbstractSqlOperations op1, op2 and op3
     */
    public CompoundSqlOperation(AbstractSqlOperation op1,
                                 AbstractSqlOperation op2,
                                 AbstractSqlOperation op3) {
        add(op1);
        add(op2);
        add(op3);
    }

    /**
     * Add the AbstractSqlOperation op to my list of operations.
     */
    public void add(AbstractSqlOperation op) {
        operations.add(op);
    }

    /**
     * Execute all operations contained in my operations list on the
     * same Connection. This method returns null. Compounding
     * SQL operations only makes sense for update, insert and
     * remove operations, which have no result.
     */
    protected Object execute(Connection c) throws SQLException {
        if (debugSQL()) System.err.println("Start of
        CompoundSqlOperation");
        int index = 1;
        for (Iterator i = operations.iterator();i.hasNext();index++) {
            AbstractSqlOperation op = (AbstractSqlOperation)i.next();
            op.execute(c);
        }
        if (debugSQL()) System.err.println("End of
        CompoundSqlOperation");
        return null;
    }
}
```

LISTING 7

**Use of OracleSequence and CompoundSqlOperation to create an EJB.**

```java
public CustomerPK ejbCreate(Customer value)
    throws RemoteException,
    CreateException {
try {
    value.printOn(System.err);
    // let Oracle decide what the next
    // primary key will be...
    BigDecimal nextId = (new OracleSequence(
        SQLConstants.CUSTOMER_SEQ)
    ).getNextValue();
    value.setId(nextId.intValue());
    InsertCustomer sql1 =
        new InsertCustomer(value);
    InsertCustomerContact sql2 =
        new InsertCustomerContact(value);
    CompoundSqlOperation cso =
        new CompoundSqlOperation(sql1,sql2);
    // write the address if it's not null
    if (value.getDefaultAddress() != null) {
        cso.add(
            new InsertDefaultAddress(value));
    }
    cso.execute();
    setValue(value);
    return new CustomerPK(value.getId());
} catch (SQLException x) {
    throw new CreateException(
        "SQLException trying to create customer " +
        value);
}
```

REFERENCES

URL

**JDBC API documentation**

Frank Sauer is a technologist for Technical Resource Connection Inc., a
wholly owned subsidiary of Perot Systems, Tampa, FL. He can be contacted at
frank.sauer@trcinc.com.