

Figure 1. UML for the JDBC framework.

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):

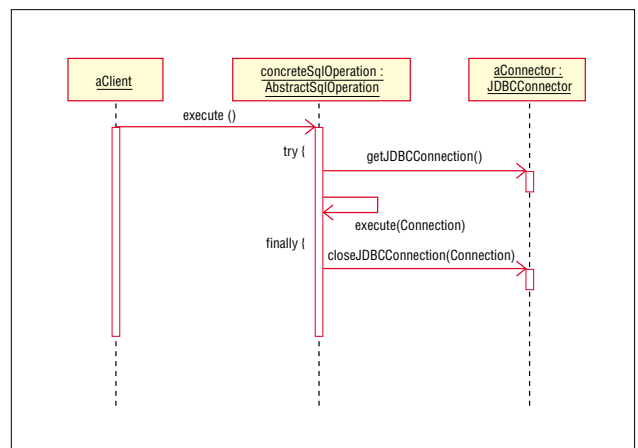


Figure 2. UML sequence diagram for AbstractSqlOperation.

```

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

LISTING 1

Persistable interface.

```

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);
}

```

SQLConstants Interface.

Technically not part of the framework, but an interesting technique to encapsulate all the actual SQL code in a single place, is the use of an interface with final static String definitions. I usually call this interface `SQLConstants` and concrete subclasses of the framework, which are part of an application—not the framework—implement this interface. For example:

```

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.

`Persistable` to automatically create a `PreparedStatement` and fill in its parameters.

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.

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

LISTING 2

Excerpt from **SQLConstants** with the definition for an insert **Customer** statement.

```
public final static String CUSTOMER_TABLE
    = "CUSTOMER";
public final static String CUSTOMER_NO
    = "CUSTOMERNO";
public final static String CUSTOMER_NAME
    = "CUSTOMERNAME";
public final static
    String CUSTOMER_SINCE_DATE
    = "CUSTOMERSINCEDATE";
public final static String CUSTOMER_ACTIVE
    = "ACTIVE";
public final static String CUSTOMER_LOGON
    = "LOGONNAME";
public final static String CUSTOMER_PASSWORD
    = "PASSWORD";
public final static
    String CUSTOMER_LASTUPDATED
    = "LASTUPDATEDDATE";
public final static String CUSTOMER_DISCOUNT
    = "DISCOUNT";
public final static String insertCustomer =
    "insert into " + CUSTOMER_TABLE +
    "(" + CUSTOMER_NO + "," +
    CUSTOMER_LOGON + "," +
    CUSTOMER_PASSWORD + "," +
    CUSTOMER_NAME + "," +
    CUSTOMER_SINCE_DATE + "," +
    CUSTOMER_ACTIVE + "," +
    CUSTOMER_DISCOUNT + "," +
    CUSTOMER_LASTUPDATED +
    ") values (?,?,?,?,?,?,?)";
public final static String[]
    insertCustomerColumns = new String[] {
    CUSTOMER_NO,
    CUSTOMER_LOGON,
    CUSTOMER_PASSWORD,
    CUSTOMER_NAME,
    CUSTOMER_SINCE_DATE,
    CUSTOMER_ACTIVE,
    CUSTOMER_DISCOUNT,
    CUSTOMER_LASTUPDATED
};
```

to call `setBigDecimal`, etc. Because the values are all objects (they have to be because they are contained in a Map), we can examine the type and use of the Strategy pattern² to set the value on the `PreparedStatement`. Let's look at this process in more detail.

The Mapper Interface. In Figure 1, you can see that `DefaultSqlOperation` has a static `HashMap` containing implementations of the `Mapper` interface. The `Mapper` interface has two methods: `setValue` and `getValue`. `setValue` is used to set a value on a `PreparedStatement`, and `getValue` is used later to retrieve a value from a `ResultSet`. The `setValue` method has the following code:

```
void setValue(PreparedStatement ps, int index,
    Object value)
```

When it is time to fill in a parameter on a `PreparedStatement`, `DefaultSqlOperation` retrieves a `Mapper` implementation from the `HashMap` using the value's class as the key:

```
Mapper m = (Mapper)mappers.get(
    value.getClass())
```

This will return a correct `Mapper` implementation for the type of the value. `DefaultSqlOperation` defines a number of inner classes that all implement the `Mapper` interface. The `mappers HashMap` is initialized with instances of each of these classes keyed on the appropriate value type (a `Class` object). For example, if the class of value is `BigDecimal`, the above statement will return a `DefaultSqlOperation.NumericMapper`. The implementation of `setValue` in `NumericMapper` calls the `setBigDecimal` method on the `PreparedStatement`:

```
public void setValue(PreparedStatement ps,
    int index,
    Object anObject)
    throws SQLException
{
    ps.setBigDecimal(index,
        (java.math.BigDecimal)anObject);
}
```

Other implementations of `Mapper` will call a different set method on the `PreparedStatement`. `DefaultSqlOperation` has a static method called `registerMapper` that can be used to register additional customized mapper objects not initially stored in the `mappers HashMap`. The initial set of mappers is lazily initialized (see Listing 3).

Each `Mapper` is registered both with a `Class` object and an integer key representing the SQL type. The second form is used later when we translate results back to objects. Now that we have a means to translate a value's type to the correct method call on a `PreparedStatement`, it is time to look at the entire process of creating a `PreparedStatement`, setting its parameters from the values in a `Persistable`, and executing the statement. The next section will discuss the translation of the `ResultSet` into an `Object` or list of `Objects`.

Figure 3 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.

```
protected Mapper getMapper(Object key) {
    if (mappers == null) {
        mappers = new HashMap();
        registerMapper(Byte.class,
            Types.TINYINT, new ByteMapper());
        DateMapper dm = new DateMapper();
        registerMapper(java.util.Date.class,
            Types.DATE, dm);
        // always convert sql Dates to util
        // Dates from ResultSets
        registerMapper(java.sql.Date.class,
            0, dm);
        registerMapper(Double.class,
            Types.DOUBLE,
            new DoubleMapper());
        registerMapper(Float.class, Types.FLOAT,
            new FloatMapper());
        registerMapper(Integer.class,
            Types.INTEGER,
            new IntMapper());
        registerMapper(BigDecimal.class,
            Types.NUMERIC,
            new NumericMapper());
        registerMapper(Long.class, Types.BIGINT,
            new LongMapper());
        registerMapper(Short.class,
            Types.SMALLINT,
            new ShortMapper());
        StringMapper strm = new StringMapper();
        registerMapper(String.class,
            Types.VARCHAR, strm);
        registerMapper(null, Types.LONGVARCHAR,
            strm);
        registerMapper(Time.class, Types.TIME,
            new TimeMapper());
        registerMapper(Timestamp.class,
            Types.TIMESTAMP,
            new TimestampMapper());
    }
    return (Mapper)mappers.get(key);
}
```

ment up to the point of actual statement execution. You can see that after the creation of the PreparedStatement, the DefaultSqlOperation's implementation of execute (Connection) first obtains the property map from its Persistable, then iterates over the column names; for each column it gets the property from the properties map, gets the appropriate Mapper implementation, and then uses it to set the parameter value on the PreparedStatement. Finally, it executes the PreparedStatement (see the DefaultSqlOperation code listing online at www.javareport.com).

The process outlined is identical for insert, update, and delete statements, but what about select statements that return a ResultSet? How does the framework create concrete objects based on the values returned in a ResultSet?

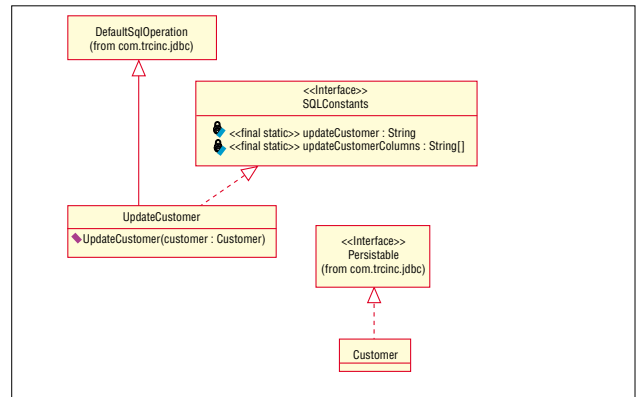


Figure 3. UML for an example DefaultSqlOperation.

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 Method² 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).

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,

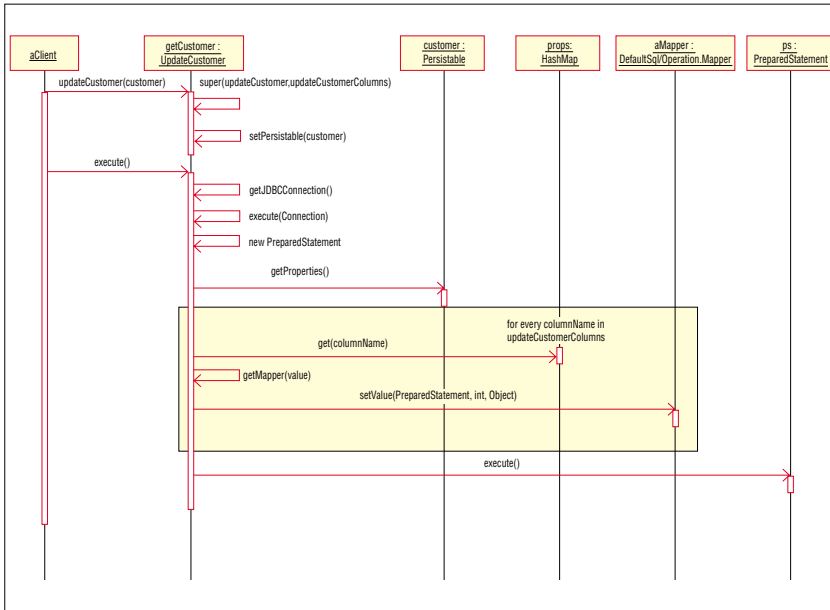


Figure 4. UML sequence diagram for creation and execution of DefaultSqlOperation.

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 getColumnName(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!

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 sub-

classes 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."³ 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.

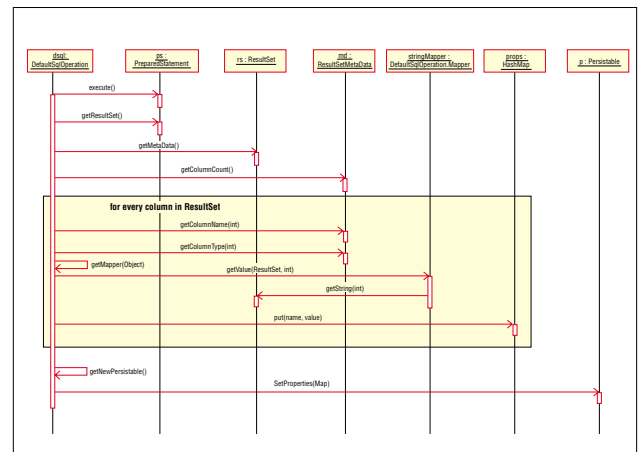


Figure 5. Processing a ResultSet.

LISTING 4

Associating an Address with a Customer.

```
class InsertDefaultAddress
    extends DefaultSQLOperation
    implements SQLConstants {

    private Customer c = null;

    InsertDefaultAddress (Customer c) {
        super(insertDefaultAddress,
            insertDefaultAddressColumns);
        this.c = c;
        // map columns from Address object
        setPersistable(c.getDefaultAddress());
    }

    protected Map getProperties() {
        Map m = super.getProperties();
        m.put(ADDRESS_LASTUPDATED,
            new java.util.Date());
        // make the link to the customer
        m.put(ADDRESS_CUSTOMER,
            new BigDecimal(c.getId()));
        return m;
    }
}
```

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.² 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 refactoring¹ 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 pro-

LISTING 5

`OracleSequence`.

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

/**
 * Allows access to Oracle Sequences
 * @author Frank Sauer
 * @version 1.0
 */
public class OracleSequence extends AbstractSQLOperation {

    private String next = null;
    private String curr = null;
    private boolean doNext = false;

    /** Creates new OracleSequence */
    public OracleSequence(String name) {
        next = "select "+name+" .nextval from dual";
        curr = "select "+name+" .currval from dual";
    }

    protected Object execute(Connection c) throws SQLException {
        ps = c.prepareStatement((doNext)?next:curr);
        ps.execute();
        ResultSet r = ps.getResultSet();
        if ((r != null) && (r.next())) return r.getBigDecimal(1);
        return null;
    }

    public BigDecimal getCurrentValue() {
        doNext = false;
        try {
            return (BigDecimal)execute();
        } catch (SQLException x) {
            return null;
        }
    }

    public BigDecimal getNextValue() {
        doNext = true;
        try {
            return (BigDecimal)execute();
        } catch (SQLException x) {
            return null;
        }
    }
}
```

jects 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.

```

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.

```

public CustomerPK ejbCreate(Customer value)
    throws RemoteException,
        CreateException {
    try {
        value.println(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

1. Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
2. Gamma, E., et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
3. Koch, G. and Kevin Loney. *Oracle 8: The Complete Reference*. Oracle Press, 1997.

URL

JDBC API documentation

www.java.sun.com/j2se/1.3/docs/guide/jdbc/index.html

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.