

Boiler Plating Database Resource Cleanup

Part II

Paul Grenyer

In my previous article, Boiler Plating Database Resource Cleanup – Part I [Part I] I explained that cleaning up after querying a database in Java is unnecessarily verbose and complex and demonstrated how boiler plate code could be developed to reduce the amount of client code needed using the Finally For Each Release pattern [AToTP]. In this article I am going to look at an alternative boiler plate solution using the Execute Around Method (EAM) [AToTP] pattern. But first, let's take another brief look at the problem:

The Problem - Revisited

The problem is simple. Cleaning up after querying a database in Java is unnecessarily verbose and complex. Plain and simple. Here is the code needed to lookup a single string in a database:

```
try
{
    Class.forName(driver);
    Connection con = DriverManager.getConnection(connectionString,
                                                username,
                                                password );

    try
    {
        PreparedStatement ps = con.prepareStatement(
            "select url from services where name = 'Instruments'");

        try
        {
            ResultSet rs = ps.executeQuery();
            if(rs.next())
            {
                System.out.println(rs.getString("url"));
            }

            try
            {
                rs.close();
            }
            catch(SQLException e)
            {
                // Report Error
            }
        }
        finally
        {
            try
            {
                ps.close();
            }
            catch(SQLException e)
            {
                // Report Error
            }
        }
    }
}
```

```

        }
    }
}
finally
{
    try
    {
        con.close();
    }
    catch(SQLException e)
    {
        // Report Error
    }
}
}
catch(Exception e)
{
    // Report Error
}
}

```

This is a lot of code to get one string out of a database and most of it must be repeated every time a database is accessed. Most of it is error handling and resource management. For a more detailed look at this code see Part I.

Execute Around Method

The EAM pattern is described by Kevlin Henney in his article *Another Tale of Two Patterns*. EAM describes how to “encapsulate pairs of actions in the object that requires them, not code that uses the object, and pass usage code to the object as another object.”

The advantage of EAM over Finally For Each Release is that the client is able to use a resource simply by implementing an interface, using the resource passed to the subclass without worrying about how to clean up and then simply pass an instance of the subclass to another object for resource acquisition, execution and cleanup.

To Check or Not To Check

Checked Exceptions [CheckedExceptions] in Java have their advantages and disadvantages and are a source of much controversy. In my previous article I made all of my interface methods throw `Exception` (except for `ConnectionPolicy` that throws it's own custom exception) so that client code can throw almost any exception type it likes. This effectively negates the checked part of checked exceptions.

Instead of using checked exceptions for the EAM design, I am forcing client code to catch and deal with checked exceptions or translate and rethrow them as runtime exceptions, by omitting any exception specification from interface method signatures.

To aid with this I have written an `ErrorPolicy` interface:

```

public interface ErrorPolicy
{
    void handleError(Exception ex);

    void handleCleanupError(Exception ex);
}

```

and a DefaultErrorPolicy class:

```

public class DefaultErrorPolicy implements ErrorPolicy
{
    private Exception firstException = null;

    @Override
    public void handleCleanupError(Exception ex)
    {
        handleError(ex);
    }

    @Override
    public void handleError(Exception ex)
    {
        if (firstException == null)
        {
            firstException = ex;
            throw new RuntimeException(ex);
        }
    }
}

```

that translate Exception into RuntimeException where appropriate.

The ErrorPolicy interface is designed to allow exceptions thrown as a result of an error from using a database resource to be handled differently to those thrown as a result of cleaning up a database resource. For example a user may want to rethrow only use exceptions and simply log or ignore cleanup exceptions.

DefaultErrorPolicy only rethrows the first exception it is asked to handle. This guarantees that a cleanup exception, which would generally be thrown after a use exception, does not hide the use exception. If I was including the ability to log in my design I would log all exceptions handled by DefaultErrorPolicy.

The error policy must always be set and *can* always be used in the same way. To provide the necessary consistency when setting the error policy I wrote the following interface:

```

public interface ErrorPolicyUser
{
    void setErrorPolicy(ErrorPolicy errorPolicy);
}

```

To provide a common, optional method of storing and accessing a reference to the error policy I wrote the following abstract class:

```

public abstract class AbstractErrorPolicyUser
    implements ErrorPolicyUser
{
    private ErrorPolicy errorPolicy = new DefaultErrorPolicy();
}

```

```

    protected ErrorPolicy getErrorPolicy()
    {
        return errorPolicy;
    }

    @Override
    public void setErrorPolicy(ErrorPolicy errorPolicy)
    {
        this.errorPolicy = errorPolicy;
    }
}

```

From Policy to Factory

The more I thought about and discussed the `ConnectionPolicy` interface from my previous article, the more I felt it was more like an Abstract Factory [GoF] than a policy. Therefore I have renamed it.

```

public interface ConnectionFactory extends ErrorPolicyUser
{
    Connection connect();

    void disconnect(Connection con);
}

```

I want `ConnectionFactory` clients to be forced to accept an error policy without relying on it being passed to subclass constructors the interface has no control over. Extending the `ErrorPolicyUser` interface also means that the error policy can be set internally by `ConnectionProvider` (discussed next).

As discussed in my previous article, most `Connection` objects are cleaned up in the same way, so having the common code encapsulated in an abstract class prevents unnecessary code duplication. An abstract class is also the ideal place for boiler plate error policy handling:

```

public abstract class AbstractConnectionFactory
    extends AbstractErrorPolicyUser implements ConnectionFactory
{
    @Override
    public void disconnect(Connection con)
    {
        if (con != null)
        {
            try
            {
                con.close();
            }
            catch (final SQLException ex)
            {
                getErrorPolicy().handleCleanupError(ex);
            }
        }
    }
}

```

`AbstractConnectionFactory` is a good example of how a class that needs to implement `ErrorPolicyUser` can extend `AbstractErrorPolicyUser` to get the implementation and access to the error policy for free.

The disconnect method is also a good example of how the error policy is used to translate a checked exception into something else, in this case a runtime exception. A little more thought needs to be put into the connection creation factories to make sure all exceptions are caught and passed to the error policy:

```
public class StringConnection extends AbstractConnectionFactory
{
    ...
    @Override
    public Connection connect()
    {
        Connection con = null;

        try
        {
            Class.forName(driver);
            con = DriverManager.getConnection( connectionString,
                                             username,
                                             password);

            try
            {
                if (database != null)
                {
                    con.setCatalog(database);
                }
            }
            catch (SQLException ex)
            {
                try
                {
                    getErrorPolicy().handleError(ex);
                }
                finally
                {
                    disconnect(con);
                }
            }
        }
        catch (ClassNotFoundException ex)
        {
            getErrorPolicy().handleError(ex);
        }
        catch (SQLException ex)
        {
            getErrorPolicy().handleError(ex);
        }

        return con;
    }
}
```

ConnectionProvider

The concept of a connection provider was suggested to me by Adrian Fagg. The idea is that a single class is responsible for acquiring a connection, providing it to another class for use and releasing it again. This is Execute Around Method!

The advantage over `DbResourceHandler` from my previous article is equal encapsulation while making the client less restricted by what they can do with the connection. It does, however, suffer

from the same disadvantage that one method is required for uses of the connection which return values and another for uses that do not.

To allow for this, two connection use interfaces are required. The `ConnectionUser` interface is for uses of the connection that do not return a value:

```
public interface ConnectionUser extends ErrorPolicyUser
{
    void use(Connection con);
}
```

The `ConnectionValue` interface is parameterised for the type returned from uses of the connection that return a value:

```
public interface ConnectionValue<T> extends ErrorPolicyUser
{
    T fetch(Connection con);
}
```

The construction of the `ConnectionProvider` class is very straight forward. The basic constructor takes a connection factory, creates a `DefaultErrorPolicy` and passes them both to another constructor that stores the references and passes the error policy to the connection factory. This means that clients of the connection provider are free to use the default error policy or provide their own.

```
public final class ConnectionProvider
{
    private final ConnectionFactory conFactory;
    private final ErrorPolicy errorPolicy;

    public ConnectionProvider(ConnectionFactory conFactory)
    {
        this(conFactory, new DefaultErrorPolicy());
    }

    public ConnectionProvider( ConnectionFactory conFactory,
                               ErrorPolicy errorPolicy)
    {
        this.conFactory = conFactory;
        this.errorPolicy = errorPolicy;
        this.conFactory.setErrorPolicy(this.errorPolicy);
    }

    ...
}
```

`ConnectionProvider` has two other methods. One that provides a connection to a `ConnectionUser` and the other which provides a connection to a `ConnectionValue` and returns the fetched value:

```
public final class ConnectionProvider
{
    ...

    public void provideTo( ConnectionUser user )
    {
        user.setErrorPolicy(errorPolicy);
        final Connection con = conFactory.connect();
    }
}
```

```

        try
        {
            user.use(con);
        }
        finally
        {
            conFactory.disconnect(con);
        }
    }

    public <T> T provideTo( ConnectionValue<T> fetcher )
    {
        fetcher.setErrorPolicy(errorPolicy);
        final Connection con = conFactory.connect();

        T result = null;

        try
        {
            result = fetcher.fetch(con);
        }
        finally
        {
            conFactory.disconnect(con);
        }

        return result;
    }
}

```

Both methods pass the error policy to the user of the connection, create the connection, pass it to the user and cleanup the connection. They rely on the connection factory to deal with any errors. The example below shows how the `ConnectionProvider` can be used:

```

class User extends AbstractErrorPolicyUser implements ConnectionUser
{
    @Override
    public void use(Connection con)
    {
        // Use the connection
    }
}

final ConnectionProvider cp =
    new ConnectionProvider(new StringConnection(...));

cp.provideTo( new User() );

```

The `User` class extends the `AbstractErrorPolicyUser` to get the common error policy storage functionality and implements `ConnectionUser` so that it can be handled by `ConnectionProvider`. There is a single override where the connection is used.

Having to extend `AbstractErrorPolicyUser` *and* implement `ConnectionUser` is not ideal. My original design had an `AbstractConnectionUser` class that extended `AbstractErrorPolicyUser` and implemented `ConnectionUser` so that clients only had to extend a single class. This meant having a similar abstract class for every user and value variant, which did not seem worth it when, as we will see later, the user and value variants are encapsulated in another class unless the client wants something custom.

StatementProvider

The StatementProvider class:

```
public final class StatementProvider
{
    private final Connection con;
    private final ErrorPolicy errorPolicy;

    public StatementProvider(Connection con, ErrorPolicy errorPolicy)
    {
        this.con = con;
        this.errorPolicy = errorPolicy;
    }

    public void provideTo( StatementUser user )
    {
        user.setErrorPolicy(errorPolicy);
        try
        {
            final Statement stmt = con.createStatement();

            try
            {
                user.use(stmt);
            }
            finally
            {
                try
                {
                    stmt.close();
                }
                catch(SQLException ex)
                {
                    errorPolicy.handleCleanupError(ex);
                }
            }
        }
        catch(SQLException ex)
        {
            errorPolicy.handleError(ex);
        }
    }

    public <T> T provideTo( StatementValue<T> fetcher )
    {
        fetcher.setErrorPolicy(errorPolicy);

        T result = null;

        try
        {
            final Statement stmt = con.createStatement();

            try
            {
                result = fetcher.use(stmt);
            }
            finally
            {
                try
                {
                    stmt.close();
                }
            }
        }
    }
}
```

```

        }
        catch (SQLException ex)
        {
            errorPolicy.handleCleanupError(ex);
        }
    }
    catch (SQLException ex)
    {
        errorPolicy.handleError(ex);
    }

    return result;
}
}

```

is a natural progression from the `ConnectionProvider` and uses EAM in the same way to provide a `Statement` to a client without the client needing to worry about acquisition or cleanup. The construction is simple and takes only a `Connection`, from which to create the statement, and an error policy. Again, it has two `provideTo` methods. One parametrised method that passes the `Statement` to a `StatementValue`:

```

public interface StatementValue<T> extends ErrorPolicyUser
{
    T use(Statement stmt);
}

```

and returns a value. The other method passes the `Statement` to a `StatementUser`:

```

public interface StatementUser extends ErrorPolicyUser
{
    void use(Statement stmt);
}

```

Both methods pass the error policy to the user of the statement, create the statement, pass it to the user, clean it up again and are responsible for error handling.

The execution of statements that do not return a value is very straight forward, so I wrote `StatementUser` for this purpose:

```

public class Execute extends AbstractErrorPolicyUser
    implements StatementUser
{
    private final String sql;

    public Execute(String sql)
    {
        this.sql = sql;
    }

    @Override
    public void use(Statement stmt)
    {
        try
        {
            stmt.execute(sql);
        }
        catch (SQLException ex)
        {

```

```

        getErrorPolicy().handleError(ex);
    }
}

```

ResultSetProvider

Executing statements that return one or more values is less straight forward and requires a ResultSetProvider:

```

public final class ResultSetProvider
{
    private final String sql;
    private final Statement stmt;
    private final ErrorPolicy errorPolicy;

    public ResultSetProvider(String sql,
                             Statement stmt,
                             ErrorPolicy errorPolicy)
    {
        this.sql = sql;
        this.stmt = stmt;
        this.errorPolicy = errorPolicy;
    }

    public <T> T provideTo(ResultSetFunction<T> fetcher)
    {
        fetcher.setErrorPolicy(errorPolicy);

        T result = null;

        try
        {
            final ResultSet rs = stmt.executeQuery(sql);

            try
            {
                result = fetcher.read(rs);
            }
            finally
            {
                try
                {
                    rs.close();
                }
                catch (Exception ex)
                {
                    errorPolicy.handleCleanupError(ex);
                }
            }
        }
        catch (SQLException ex)
        {
            errorPolicy.handleError(ex);
        }

        return result;
    }
}

```

Again, this is another natural progression from ConnectionProvider. The

ResultSetProvider takes the SQL query to execute to create the result set, a Statement from which to create the record set and an error policy. There is only a single paramatised provideTo method as a value is always returned. The ResultSet is provided to a ResultSetFunction:

```
public interface ResultSetFunction<T> extends ErrorPolicyUser
{
    T read(ResultSet rs);
}
```

that is parameterised on return type. The method passes the error policy to the user, creates the RecordSet from the Statement and SQL query, passes the ResultSet to the ResultSetFunction, cleans up, handles any errors and returns the value.

With the ResultSetProvider, executing a query that returns a value is almost as simple as executing one that does not and can benefit from similar boilerplate:

```
public class ExecuteQuery<T> extends AbstractErrorPolicyUser
    implements StatementValue<T>
{
    private final String sql;
    private final ResultSetFunction<T> rsUser;

    public ExecuteQuery(String sql, ResultSetFunction<T> rsUser)
    {
        this.rsUser = rsUser;
        this.sql = sql;
    }

    @Override
    public T use(Statement stmt)
    {
        return new ResultSetProvider( sql,
                                     stmt,
                                     getErrorPolicy())
            .provideTo(rsUser);
    }
}
```

The ExecuteQuery class is paramatised on the return type from the query. It takes a SQL query and ResultSetFunction via it's constructor. When an instance is passed to a StatementProvider provideTo method it uses a ResultSetProvider and the ResultSetFunction to execute and return the results of the query.

Query

All of this boilerplate can be wrapped in a single class that provides two static methods, one to execute methods that return a value and another for ones that do not:

```
public final class Query
{
    ...

    public static void execute(ConnectionProvider conProvider,
                              String sql) throws Exception
    {
        conProvider.provideTo(new User(sql));
    }
}
```

```

    }

    public static <T> T execute(ConnectionProvider conProvider,
                               String sql,
                               ResultSetFunction<T> rsUser)
                               throws Exception
    {
        return conProvider.provideTo(new Value<T>(sql, rsUser));
    }

    private Query()
    {}
}

```

Both methods take a ConnectionProvider and a SQL query. The method which returns a value also takes a ResultSetFunction to process the result set into the return value. Both methods pass an instance of a nested class to the connection provider. The nested class that handles queries that do not return a value is called User:

```

private static class User extends AbstractErrorPolicyUser
    implements ConnectionUser
{
    private final String sql;

    public User(String sql)
    {
        this.sql = sql;
    }

    @Override
    public void use(Connection con)
    {
        new StatementProvider( con, getErrorPolicy())
            .provideTo( new Execute(sql) );
    }
}

```

It takes the SQL query via its constructor and, when passed to a ConnectionProvider, uses the StatementProvider and Execute classes to execute the query.

The nested class that handles queries that return a value is called Value:

```

private static class Value<T> extends AbstractErrorPolicyUser
    implements ConnectionValue<T>
{
    private final String sql;
    private final ResultSetFunction<T> rsUser;

    public Value(String sql, ResultSetFunction<T> rsUser)
    {
        this.rsUser = rsUser;
        this.sql = sql;
    }

    @Override
    public T fetch(Connection con)
    {
        return new StatementProvider( con, getErrorPolicy())
            .provideTo( new ExecuteQuery<T>(sql, rsUser) );
    }
}

```

```
}
```

It takes the SQL query and a `ResultSetFunction` via its constructor and, when passed to a `ConnectionProvider`, uses the `StatementProvider` and `ExecuteQuery` classes and the `ResultSetFunction` interface to execute the query and return the result.

AbstractResultSetFunction

The `ResultSetFunction` interface needs a little further explanation as it is the only interface most clients will need to implement albeit then only for queries that return a value.

```
public interface ResultSetFunction<T> extends ErrorPolicyUser
{
    T read(ResultSet rs);
}
```

The interface extends `ErrorPolicyUser`, which means clients can make use of `AbstractErrorPolicyUser` to get the common implementation. As I will demonstrate later, it will often be useful to extend `ResultSetFunction` using an anonymous class. Anonymous classes cannot inherit from more than one class or interface, therefore I have written an `AbstractResultSetFunction` class that does nothing other than extend `AbstractErrorPolicyUser` and implement `ResultSetFunction`:

```
public abstract class AbstractResultSetFunction<T>
    extends AbstractErrorPolicyUser
    implements ResultSetFunction<T>
{
}
```

Now all clients have to do is extend `AbstractResultSetFunction` and implement the `read` method.

The `read` method takes a `ResultSet` and returns a value. It is responsible for extracting the results from the result set and processing them into something that can be returned. As `read` does not have an exception specification it is also responsible for error handling. For example, retrieving a single string from a database table:

```
new AbstractResultSetFunction<String>()
{
    @Override
    public String read(ResultSet rs)
    {
        String result = null;

        try
        {
            if (rs.next())
            {
                result = rs.getString("url");
            }
        }
        catch (SQLException ex)
        {
            getErrorPolicy().handleError(ex);
        }

        return result;
    }
}
```

```
    }
}
```

or retrieving multiple strings from a database table:

```
new AbstractResultSetFunction<List<String>> ()
{
    @Override
    public List<String> read(ResultSet rs)
    {
        List<String> result = new ArrayList<String>();

        try
        {
            while (rs.next())
            {
                result.add(rs.getString("url"));
            }
        }
        catch (SQLException ex)
        {
            getErrorPolicy().handleError(ex);
        }

        return result;
    }
}
```

Putting It All Together

To use the boilerplate to query a database, a client must first create a connection factory:

```
final ConnectionFactory conFactory =
    new StringConnection(DRIVER, CONNECTION_STRING)
        .setUser(USERNAME, PASSWORD)
        .setDatabase(DATABASE);
```

Then create a connection provider and pass it the connection factory and, optionally, a custom error policy:

```
final ConnectionProvider cp = new ConnectionProvider(conFactory);
```

Once created the connection factory and connection provider can be used for any number of queries and therefore only need to be created once. Executing a query that does not return any results can then be done with a single statement:

```
Query.execute( cp,
    "insert into services ([name],[url])" +
    "VALUES('Log','http://prodserv01/axis/services/Log')");
```

Queries that return results only take a little more, almost all of which is the anonymous class that processes the results from the result set:

```
final String s =
    Query.execute( cp,
        "select url from services where name = 'Instruments'",
        new AbstractResultSetFunction<String> ()
        {
            @Override
```

```
public String read(ResultSet rs)
{
    String result = null;

    try
    {
        if (rs.next())
        {
            result = rs.getString("url");
        }
    }
    catch (SQLException ex)
    {
        getErrorPolicy().handleError(ex);
    }

    return result;
}
});
```

Conclusion

Boiler plating database resource cleanup with Execute Around Method offers a high level of safety and encapsulation while not compromising on control of the database resources in client code. Reducing the amount of code that has to be written each time also reduces the possibility of mistakes and resource leaks.

Acknowledgements

Thank you to Adrian Fagg for guidance into Execute Around Method, not to mention quite a bit of help with the design.

References

[Part I] Boiler Plating Database Resource Cleanup – Part I: http://www.marauder-consulting.co.uk/Boiler_Plating_Database_Resource_Cleanup_-_Part_I.pdf

[AToTP] Another Tail of Two Patterns:
<http://www.two-sdg.demon.co.uk/curbralan/papers/AnotherTaleOfTwoPatterns.pdf>

[CheckedExceptions] Checked Exceptions:
http://en.wikipedia.org/wiki/Exception_handling#Checked_exceptions

[GoF] Gang of Four: Design Patterns : Elements of reusable object-oriented software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison Wesley ISBN-13: 978-0201633610