Dynamic Data Access Object Design Pattern (CECIIS 2008)

Zdravko Roško, Mario Konecki
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
zrosko@yahoo.com, mario_konecki@yahoo.com

Abstract. *Business logic application layer accessing data from any data source (databases, web services, legacy systems, flat files, ERPs, EJBs, CORBA services, and so forth) uses the Dynamic Data Access Object which implements the Strategy[1] design pattern and hides most of the complexity away from an application programmer by encapsulating its dynamic behavior in the base data access class. By using the data source meta data, it automates most of the functionality it handles within the application. Application programmer needs only to implement specific „finder" functions, while other functions such as „create, store, remove, find, removeAll, storeAll, createAll, findAll" are implemented by the Dynamic Data Access Object base class for a specific data source type..*

Currently there are many Object Relational Mapping products such as Hibernate, iBatis, EJB CMP containers, TopLink, which are used to bridge objects and relational database. Most of the time this approach (Object Relational Mapping) makes one more unnecessary layer of the complexity. Dynamic Data Access Object helps application programmers to avoid the usage of the Object Relational Mapping when there is no benefit of using it. Dynamic Data Access Object is an implementation of „pass through" instead of Object Relational Mapping application behavior at the data access layer.

Keywords. Strategy, Data Access Object, Sovereign Value Object, EJB, Value List Handler, Result Set, Object Relational Mapping, Caching, Meta Data, Transaction, Connection, Business Object, Facade.

1 Introduction

This paper presents a pattern that help to desing the data access layer for any data source (not just relational) such as CICS, JMS/MQ, iSeries, SAP, Web Services, and so forth. Dynamic Data Access Object (DDAO) is an implementation of the *Strategy* design pattern [1] which defines a family of algorithms, encapsulate each one, and make them interchangeable through an interface.

Having many options available (EJB, Object Relational Mapping, POJO, J2EE DAO, etc.) to use while accessing a data source, including persistent storage, legacy data and any other data source, the main question for development is: what to use to bridge the business logic layer and the data from a data source ? Assuming that the data access code is not coded directly into the business logic layer (Entity Bean, Session Bean, Servlet, JSP Helper class, POJO) to avoid tight coupling to the data source, the DDAO is an option to use if one needs to avoid XML configuration, complex environment setup, unnecessary code redundancy, poor performance.

If the data source get changed during the life time of an application, there is no need to change the business logic layer code, but to introduce a new DDAO class for specific data source, which gets attached to the business entity object (BO).

Having separated DDAO and BO, it makes reuse of the DDAO possible by other parts of the business logic layer of an application.

DDAO also simplify the code development, unit test, integration test, and simultaneous access of multiple types of data sources such as relational, legacy and so forth, from one BO.

2 Example

Figure 1. shows the implementation of the interface which is used by the BO instead of the usage of a specific DDAO. An interface is implemented by each DDAO implementation for a specific BO, and represents the implementation of the *Strategy*[1] design pattern.

```
hr.adriacomsoftware.app.server.account.dao;

public interface BankAccountDao {
   publicBankAccountRs daoFindClosedAccounts (
            BankAccountVo value)
                  throws
J2EEDataAccessException;
}
```

Figure 1. Sample DDAO interface

Figure 2. shows an implementation of the DDAO for a JDBC data source. All functionality such as INSERT, DELETE, UPDATE, is implemented at the base class level (*J2EEDataAccessObjectJdbc)* while an application's DDAO implements a *finder* methods only for handling specific SQL queries.

```
hr.adriacomsoftware.app.server.account.dao.jdbc;

public final class BankAccountJdbc extends
J2EEDataAccessObjectJdbc implements
BankAccountDao {
public BankAccountJdbc() {
      setTableName("bank_account");
 }
Public BankAccountRs
daoFindClosedAccounts(BankAccountVo value) throws
J2EEDataAccessException {
      J2EEConnectionJDBC co = null;
      BankAccountRs j2eers = null;
      try {
         co = getConnection();
         Connection jco = co.getJdbcConnection();
         J2EESqlBuilder sqlstatement = new
         J2EESqlBuilder();
         sqlstatement.appendln("select * from
         bank_account where
         status = 'C' ");
         sqlstatement.append(" ORDER BY closed_date");
         PreparedStatement pstmt =
         jco.prepareStatement(sqlstatement.toString());
         pstmt.setMaxRows(0);
         ResultSet rs = pstmt.executeQuery();
         j2eers = new
BankAccountRs(getJ2EEResultSetFromJDBCResultSet(r
s));
         pstmt.close();
         return j2eers;
      } catch (Exception e) {
         J2EEDataAccessException ex = new
         J2EEDataAccessException("151");
         ex.addCauseException(e);
         throw ex;
      }
   }
}
```

Figure 2. Sample DDAO base class

3 Context

To make an application flexible, the creation of DDAO is done out of BO. DDAO abstracts business logic from the knowledge where the data resides, how to access the data, where to cache the data, etc.
Figure 3. shows the context where the BO is used to access the DDAO for the data manipulation purpose. Typical Java application uses the data entered by the client or data retrieved from the data source. Data is transfered from the *model* to the *business object* in the form of *value object* [2]. From the presentation layer the data is transfered by using a business logic *proxy* which encapsulates the communication to the *business logic* layer. It is not necessary that business logic layers resides on the separate CPU process context, which means that it can run within the same process as the client presentation layer. Communication from the *proxy* to the logical server component is done by the J2EETransport component (it will be the subject for a new paper form the authors). Value object implements a *command* design pattern which is used to invoke *SampleBankComponentFacde* from the *transport layer*. When BO gets called, it can use DDAO to do all data source related logic, and return the data back to the calling client.

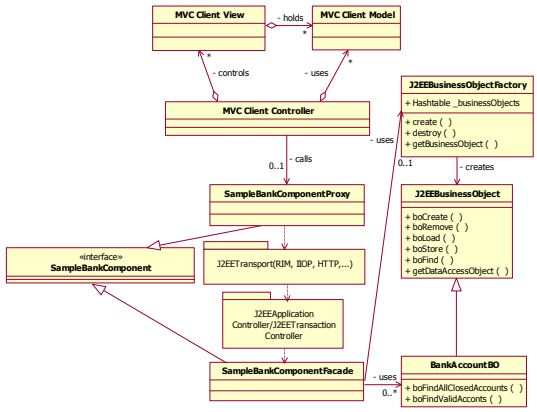

Figure 3. The context for the DDAO usage

4 Problem

Hibernate, iBatis and other available implementations of object relational mapping or Data Access Object (DAO) pattern, relay more on programmers to produce the data access code, then on the data source meta data for SQL statement generation, or other meta data sources if not JDBC data source is used. DDAO does not require complex configuration files at all except one for each data source destination to identify the host, port, user and couple of other parameters. DDAO has the default implementation for specific data sources such as CICS, JMS, iSeries, etc.

If we build the application which does require simple and fast access to the data sources and which uses a data source capabilities such as stored procedures, CICS programs, JMS programmed logic, we do not need an object relational mapping, but instead we need to have a mechanism to handle access, connection, transaction, caching, for each of the specific data source types on a unique and manageable way.

The DDAO architecture enables the transaction management for more then one transaction destination (RDMS, CICS, JMS, etc.) at the same time during a client logical unit of work. It is possible to build a transaction by using DDAO which uses CICS, JMS and JDBC data source at the same logical unit of work.

During analysis you usually define the data model of the application in third normal form. However, the entire system performs poorly if you also use third normal form as the physical table layout [6]. Using EJB bean managed persistence or other types of object relational mapping leads to a limited tuning activities needed to achieve acceptable performance.

Most of the DAO implementations require a set up of complex development environment which includes containers and other specific environment set up, depending on the DAO implementation being used.

5 Solution

The DDAO access the data source while the BO is not coupled to a specific vendor implementation or API. DDAO contains all the source code developer needs to change, in case data source vendor or API get changed. BO stays the same not depending on specific data source.

The DDAO has the following functions:

- Contains the logic to access a data source
- Uses a connection from a connection pool
- Caches the data access statements (eg. SQL)
- Manages cached data from the data source
- Converts data from the data source to a specific format such as Sovereign Value Object (SVO) [2], Service Data Object (SDO)[3], Data Transfer Object (DTO)[4], and etc.
- Validates the fields lengths and types for the fields candidates for being stored to a data source by using its meta data.
- Enables transaction management (flat transactions only) for JDBC and not JDBC data source (JMS, CICS, etc.). DDAO is defined within the transaction context which transparently handles the transaction, leaving application programmer free from the transaction handling complexity. DDAO can handle JTA, JDBC, JMS, CICS, and other transaction types during one logical unit of work.
- Simplify the application development environment set up. DDAO development does not require any kind of containers or other specific environment set up. Once the DDAO is developed, unit tested, it could be deployed to any thread safe environment such as servlet container, EJB container or as a Java application or Java Applet (not recommended).

Figure 4. illustrates DDAO layer. DDAO layer manages reading, writing, updating and deleting data at the data source in both cases whether data is stored or created by the data source at the run time.

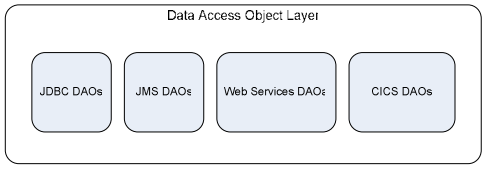

Figure 4. DDAO layer perspective

While making a decision which persistence method to use, we can consider what needs to happen at the data access object layer. Grading each persistence method according to how well it achieves the goals, we can use the following rating system[7]:

- High (best rating): Gets high marks toward achieving the stated goal
- Medium (middle rating): Moderately achieves the stated goal
- Low (lowest rating): does not achieve the stated goal very well

Figure 5. lists the goals and ratings of DDAO and several others persistence methods.

Goal	JDBC	EJB/ BMP	EJB/ CMP	O/R	DDAO
Minimize learning curve	High	Low	Low	Med	**High**
Minimize code and configuration files written and maintaeined	Low	Low	Low	Med	**Med**
Maximize ability to tune	High	Med	Low	Low	**High**
Minimize development effort	High	Low	Low	Med	**High**
Maximize code portability	Med	Med	High	High	**Med**
Minimize vendor reliance	High	Med	Med	Low	**High**
Maximize availability and fail-over	Low	High	High	Low	**Med**
Manageable via JTA	High	Low	Low	Med	**High**
Handles other then JTA transaction	Low	Low	Low	Low	**High**
Access other then JDBC data sources	Low	Low	Low	Low	**High**

Figure 5. Ratings of data persistence methods

6 Structure

Figure 6. shows the class diagram for the DDAO pattern.

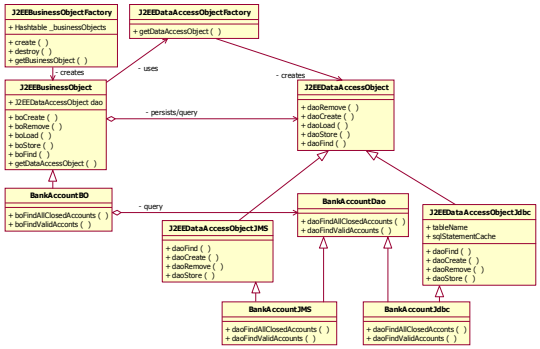

Figure 6. DDAO class diagram

DDAO includes the following classes:

- J2EEDataAccessObjectFactory is a class that is responsible for creating and returning DataAccessObject instances for Business Objects. It holds a mapping that BO query to get the Data Access Object that handles its persistence. The default mapping assumed by the naming convention does not require any programmers action at all or an entry to the mapping configuration file. Just for the specific naming of the data access objects for particular BO, an entry needs to be entered to the configuration file. The J2EEDataAccessObjectFactory uses Business Object name to choose a specific class that subclass the DataAccessObject and serves as a DDAO for the BO.
- J2EEDataAccessObject is a common abstract class for all Data Access Objects. The standard access methods daoStore, daoLoad, daoCreate, daoFind, daoRemove, daoRemoveAll, daoLoadNext, etc., are defined within the class. DDAO method naming and class structure by design are ready to be converted to an EJB Bean Managed Persistence class, in case it is required by a specific application.
- J2EEDataAccessObjectJdbc is the JDBC implementation of J2EEDataAccessObject which defines all its base class methods specific for the JDBC data source.
- J2EEDataAccessException is the Data Access layer *unchecked* exception. Unchecked exceptions may be thrown from any method without being specified as part of the method's public interface (and a handler need not be present)[5]. It is recommended that all DDAO inheriting from J2EEDataAccessObject catch the other types of the exceptions and throw the J2EEDataAccessException. Business logic layer need not to handle any other type of exception while accessing data access layer.
- J2EEBusinessObject asks the J2EEDataAccessObjectFactory class for an J2EEDataAccessObject instance to which it delegates persistence operations.

Figure 7. shows the DDAO data source access classes for caching, connection management and configuration properties access. Each DDAO inherits from the specific data source class, which is connected to its J2EEPersistenceService and has its own configuration file attached. For example, if there are two JDBC data sources and one JMS data sources to be accessed, there needs to exist three J2EEPersistenceService implementations which inherit from the J2EEPersistenceService and have just simple *getConnection* method implemented, plus a configuration file for each of them configured.

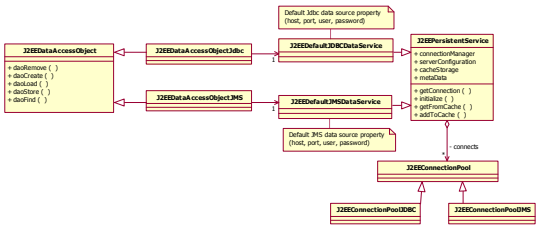

Figure 7. DDAO data source access classes

7 Dynamics

Figure 8. shows the message flow between the objects at the data access layer. BO is a pass-through entity object which is being instantiated only once and used by all clients on a thread-safe way during the time the business logic layer is up and running. BO does not keep the state even though this option exists for the specific needs (not explained here). BO can keep the state by using a thread specific storage design pattern [8]. BO asks J2EEDataAccessObjectFactory to create the Data Access Object and use it to read and write the data to the data source. Methods such as daoStore, daoCreate, daoRemove, daoLoad and daoFind are implemented by the base J2EEDataAccessObjectJdbc class, while the rest of query methods are implemented by the concrete (BankAccoutJdbc) classs. BankAccountBO does not access BankAccountJdbc class directly but using its BankAccountDao interface, which makes the DDAO replaceable by other data access class in case the data source get changed.

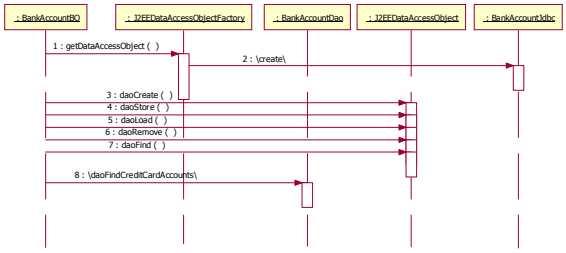

Figure 8. DDAO sequence diagram

8 Implementation

Figure 9. show a part of the source code from the **J2EEDataAccessObjectJdbc** class. Methods *daoCreate* and *daoCreateMany* are inherited by application specific DDAO. *daoCreate* gets the JDBC connection from the connection pool for it's target data source, do prepares data specific for an environment or locale (*prepareSqlCreate*), prepares (just once per connection session) or retrieves from the local cache the INSERT statement, checks the fields length (*validateFieldLength*) using the meta data column size information and maps the client data from *J2EEValueObject*[2] to the table columns before executing INSERT SQL statement.

```
public class J2EEDataAccessObjectJdbc extends
J2EEDataAccessObject{

public J2EEValueObject daoCreate(J2EEValueObject value)
throws J2EEDataAccessException {
    J2EEValueObject res = new J2EEValueObject();
    J2EEConnectionJDBC co = null;
    try {
            prepareData(value, getTableName());
            co = getConnection();
            Connection jco = co.getJdbcConnection();
            String sql = prepareSqlCreate(value, getTableName());
            PreparedStatement pstmt = jco.prepareStatement(sql);
            J2EEResultSet columnsInfo =
            co.getColumnsInfo(getTableName());
            Vector columns = columnsInfo.getRows();
            validateFieldLength(columnsInfo, value);
            Hashtable hashKeys =
            co.getPrimaryKeysInHashtable(getTableName());
            int i = 0;
            for (Enumeration e = columns.elements();
                        e.hasMoreElements();){
                        J2EEValueObject model =
                        (J2EEValueObject)    e.nextElement();
                        Hashtable row = model.getProperties();
                        if(co.isIdentityColumn(row))
                                    continue; //identity column
                        String tableColumn =
                        co.getColumnName(row);
            ……..
```

Figure 9. DDAO base class *daoCreate* method

Figure 10. shows the source code for creating many objects of the same table to the JDBC data source.

```
public int daoCreateMany(J2EEResultSet valueList) throws
J2EEDataAccessException {
   J2EEValueObject vo;
   int rowCounter=0;
   Enumeration E = valueList.getRows().elements();
   while(E.hasMoreElements()){
            vo = (J2EEValueObject)E.nextElement();
            daoCreate(vo);
            rowCounter++;
   }
   return rowCounter;
}
```

Figure 10. DDAO base class *daoCreateMany* method

9 Applicability

DDAO could be used when

- Acces CICS, JMS, Web Services, CORBA service and other data sources. DDAO defines a common class for each of these data source types, which inherits from J2EEDataAccessObject and handles connection and transaction logic for the specific data source. By inheriting the same base class as other DDAO classes, it is possible to switch the data sources if they get changed and it frees the application programmer from the concerns about connection pool and transaction logic.
- Building the application for the multiple deployment options (pure Java application, applet or an application server component within a Servlet od EJB container).
- Using Eclipse or other IDE tools to develop data access logic without the need to have a container configured, but pure JDK enviroment.
- Building a complex enterprise size application to access legacy, RDBMS, and other data sources for implementing a single logical unit of work combined from many differnet data source types.

10 Variants

DDAO method assumes one or more DDAO per business entity object. If one BO needs to access more then one data source destination, one needs to create as many DDAO and configure J2EEDataAccessObjectFactory to handle this situation. Multiple DDAO are kept within a BO as a list of DDAO and accessed accordingly when needed by the BO. When CICS programs are called from DDAO, COBOL copy books are used to produce meta data which are used by the DDAO to handle calls to a CICS programs. Similarly when accessing iSeries programs from a DDAO, C include header files are used to generate meta data used by iSeries DDAO to access C programs.

11 Consequences

DDAO is very lightweight persistence solution which ease the development of data-driven applications by abstracting the low-level details involved in data source communication (loading driver, managing connections, managing transaction, etc.) as well as providing higher-level capabilities (data type conversion management, support for static and dynamic queries or program calls, mapping attributes to columns or other data source types, etc.). DDAO includes a code generation tool which helps to generate specific DDAO by using a data source meta data (JDBC meta data, PDML, COBOL copy books, etc.).

13 Related patterns

Figure 11. shows the message flow between the objects at the data access layer.

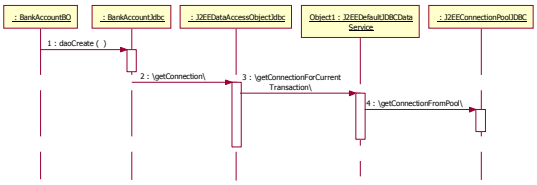

Figure 11. DDAO data source access sequence diagram

Transaction management, while using DDAO, is done up front at the start of the business logic layer, by using current thread identification as a technique for separating client transaction contexts. There is no need to transfer a connection or transaction objects from business logic layer down to the data access layer by sending it as an parameter. All connections used by the dynamic data access objects are kept within the transaction object for current client invocation. At the end of the logical unit of work, the transaction which could include JDBC, JMS, CICS, etc., is closed by issuing a commit or rollback command ouside of DDAO implementation.

It is possible to use the transaction commands directly inside the DDAO implementation, but it rarely makes the sense. CICS, JMS and other data source connection and transaction handling, is a subject of a new paper which will be published by the authors of this paper. DDAO supports JTA, EJB, or its own flat transaction service. DDAO's own transaction service does not support chained or nested transaction types. If using DDAO transaction service, it is up to the user to decide if some of the logical unit of work needs to issue a transaction commit or rollback command while inside an existing transaction. In that case, the existing transaction needs to be closed before a new transaction is opened. DDAO uses Value List Handler [4] to return subsets of the result to the client as requested.

References

[1] Eric Gamma: **Design Patterns, Elements of Reusable Object-Oriented Software,** page 315**,** Addison-Wesley Publishing Company, Reading, USA, 1995.

[2] Zdravko Roško: **Sovereign Value Object,** Faculty of Organization and Informatics University of Zagreb, IIS 2007.

[3] BEA Systems, Inc., International Business Machines Corp, Oracle, Primeton Technologies Ltd, Rogue Wave Software, SAP AG., Software AG., Sun Microsystems, Sybase Inc., Xcalia, Zend Technologies, SDO 2.1.0, http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf?version=1, November 2006.

[4] Deepak Alur, John Crupi, Dan Malks: **Core J2EE Pattern, Best Practices and Design Strategies**, page 286, Pearson Education, 2001.

[5] Kathy Sierra, Bert Bates: **Java™ 2 Sun Certified Programmer & Developmer for Java 2**, page 250, McGraw-Hill/Osborne, New York, USA, 2003.

[6] Rebert C. Martin, Dirk Riehle, Frank Buschmann: **Pattern Languages of Program Design 3**, page 315, Addison Wesley Longman, Inc., 1998.

[7] Derek C. Ashmore: **The J2EE Architect's Handbook,** pages 51,53,54 , DVT Press, Lombard, 2004.

[8] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: **Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects**, page 475, John Wiley & Sons, Ltd., West Sussex, 2000.