"This book is essential for J2EE developers and has helped reinvigorate the patterns movement."
–Martin Fowler, Chief Scientist, ThoughtWorks

core J2EE™ PATTERNS

Best Practices and Design Strategies

SECOND EDITION



Updated and Revised!

- J2EE Pattern Catalog from the Sun Java Center
- Patterns and Strategies for J2EE and Web Services
- Refactorings and Bad Practices for J2EE
- Sample Code for Patterns, Strategies, and Refactorings

Sun microsystems

DEEPAK ALUR • JOHN CRUPI • DAN MALKS

Forewords by Grady Booch and Martin Fowler

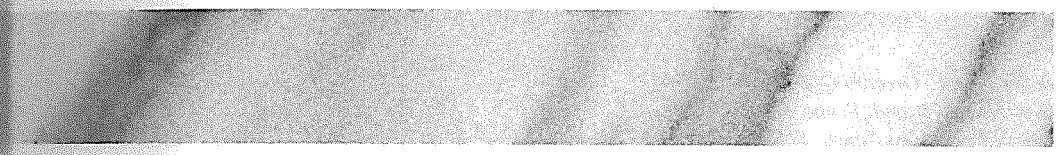
Core Design Series

Spotify USA Inc. Exhibit 1008 Page 1

core J2EE™ PATTERNS

Best Practices and Design Strategies

Second Edition



DEEPAK ALUR
JOHN CRUPI
DAN MALKS



Prentice Hall PTR, Upper Saddle River, NJ 07458
www.phptr.com

Sun Microsystems Press
A Prentice Hall Title

Spotify USA Inc. Exhibit 1008 Page 2

© 2003 Sun Microsystems, Inc.—
Printed in the United States of America.
901 San Antonio Road, Palo Alto, California
94303 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.
The products described may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS—HotJava, Java, Java Development Kit, J2EE, JPS, JavaServer Pages, Enterprise JavaBeans, EJB, JDBC, J2SE, Solaris, SPARC, SunOS, and Sunsoft are trademarks of Sun Microsystems, Inc. All other products or services mentioned in this book are the trademarks or service marks of their respective companies or organizations.

Prentice Hall PTR offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact U.S. Corporate and Government Sales, 1-800-382-3419, corpsales@pearsontechgroup.com. For sales outside of the U.S., please contact International Sales, 1-317-581-3793, international@pearsontechgroup.com.

Acquisitions Editor: *Gregory G. Doench*
Editorial Assistant: *Brandt Kenna*
Editor and Production Assistant: *Solveig Haugland*
Cover Design Director: *Jerry Votta*
Cover Designer: *Anthony Gemmellaro*
Art Director: *Gail Cocker-Bogusz*
Manufacturing Manager: *Alexis R. Heydt*
Marketing Manager: *Debby vanDijk*

Sun Microsystems Press Publisher: *Myrna Rivera*

5th Printing

ISBN 0-13-142246-4

Sun Microsystems Press
A Prentice Hall Title

Spotify USA Inc. Exhibit 1008 Page 3

Integration Tier Patterns

Topics in This Chapter

- Data Access Object
- Service Activator
- Domain Store
- Web Service Broker

Spotify USA Inc. Exhibit 1008 Page 4

Chapter 8

Integration Tier Patterns

Data Access Object

Problem

You want to encapsulate data access and manipulation in a separate layer.

Many real-world J2EE applications implement persistent objects as *Business Object* (374), which use POJOs or entity beans. An application with simpler requirements might forego using business objects and instead use *Session Façades* (341), *Application Services* (357), or other helper objects that directly access and manipulate data in the persistent storage. Such business objects and application components require access to business data in a persistent store.

Most enterprise applications typically use relational database management systems (RDBMS) as the persistent store. However, enterprise data can reside in other types of repositories, such as mainframe or legacy systems, Lightweight Directory Access Protocol (LDAP) repositories, object-oriented databases (OODB), flat files, and so on. Another example of persistent storage is where data is provided by services in external systems, such as business-to-business (B2B) integration systems, credit card bureau services, and so on.

Access mechanisms, supported APIs, and features vary among such persistent stores. Even within a single API, the underlying implementations might provide proprietary extensions in addition to standard features.

Mingling persistence logic with application logic creates a direct dependency between the application and the persistence storage implementation. Such code dependencies in components make it difficult and tedious to migrate the application from one type of data source to another. When the data source changes, the components must be modified to handle the new type of data source.

Forces

- You want to implement data access mechanisms to access and manipulate data in a persistent storage.
- You want to decouple the persistent storage implementation from the rest of your application.
- You want to provide a uniform data access API for a persistent mechanism to various types of data sources, such as RDBMS, LDAP, OODB, XML repositories, flat files, and so on.
- You want to organize data access logic and encapsulate proprietary features to facilitate maintainability and portability.

Spotify USA Inc. Exhibit 1008 Page 6

Solution

Use a *Data Access Object* to abstract and encapsulate all access to the persistent store. The *Data Access Object* manages the connection with the data source to obtain and store data.

The *Data Access Object* (also known simply as **DAO**) implements the access mechanism required to work with the data source. Regardless of what type of data source is used, the DAO always provides a uniform API to its clients. The business component that needs data access uses the simpler interface exposed by the DAO for its clients. The DAO completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this allows you to change a DAO implementation without changing the DAO client's implementation. Essentially, the DAO acts as an adapter between the component and the data source.

The DAO is implemented as a stateless object. It does not cache the results of any query execution, or any data that the client might need at a later point. This makes the DAOs lightweight objects and avoids potential threading and concurrency issues. The DAO encapsulates the details of the underlying persistence API. For example, when an application uses JDBC, the DAO encapsulates all the JDBC usage inside the data access layer and does not expose any exception, data structure, object, or interface that belongs to the java.sql.* or javax.sql.* packages to the clients outside the data access layer.

Structure

The diagram in Figure 8.1 shows the structure of the *Data Access Object* pattern.

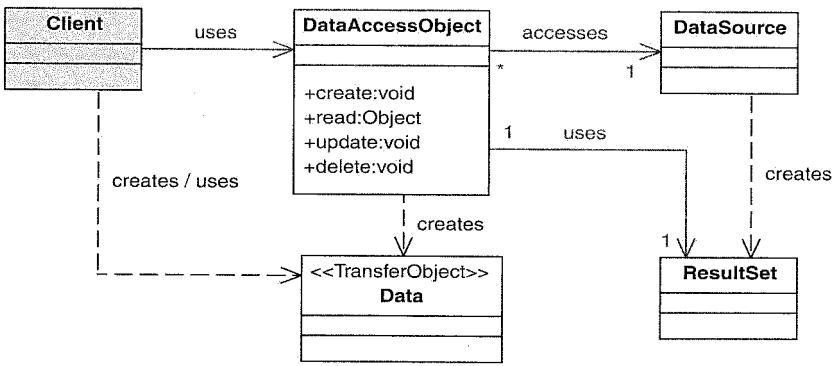


Figure 8.1 *Data Access Object* Class Diagram

Participants and Responsibilities

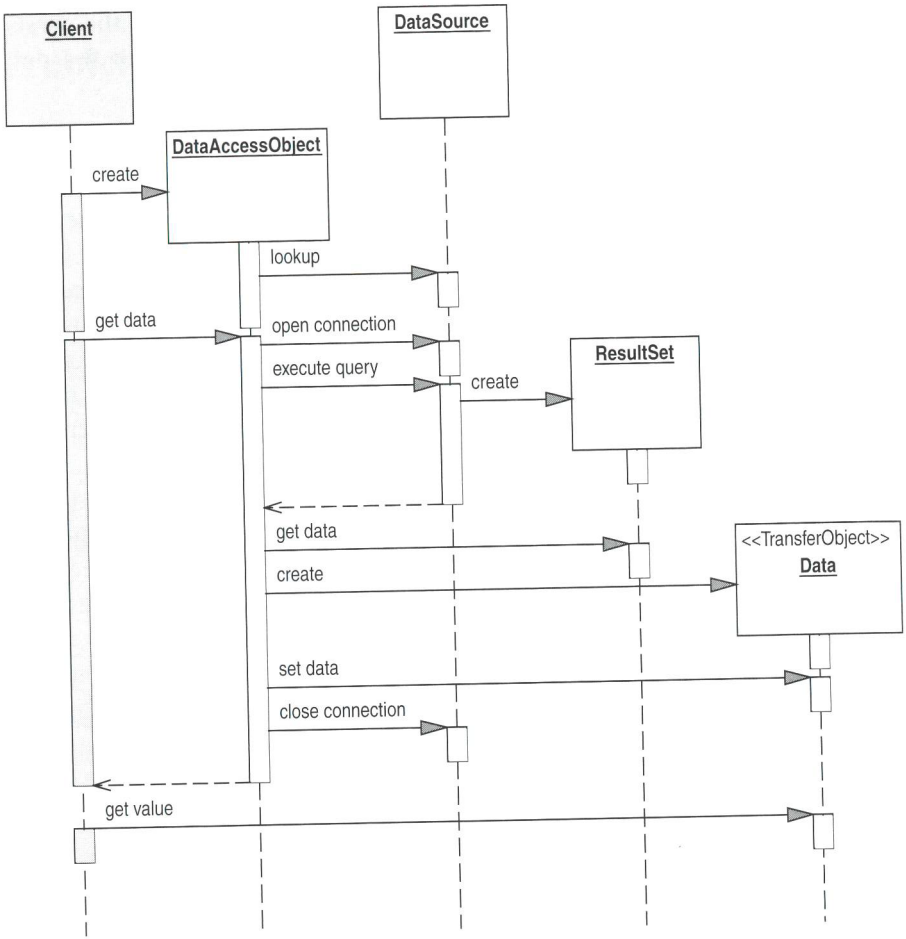


Figure 8.2 *Data Access Object* Sequence Diagram (part 1)

The diagram in Figure 8.2 shows the interaction of the various participants of the *Data Access Object* pattern for a basic operation to get data from a data source. Figure 8.3 on page 465 shows part two of the diagram.

Spotify USA Inc. Exhibit 1008 Page 8

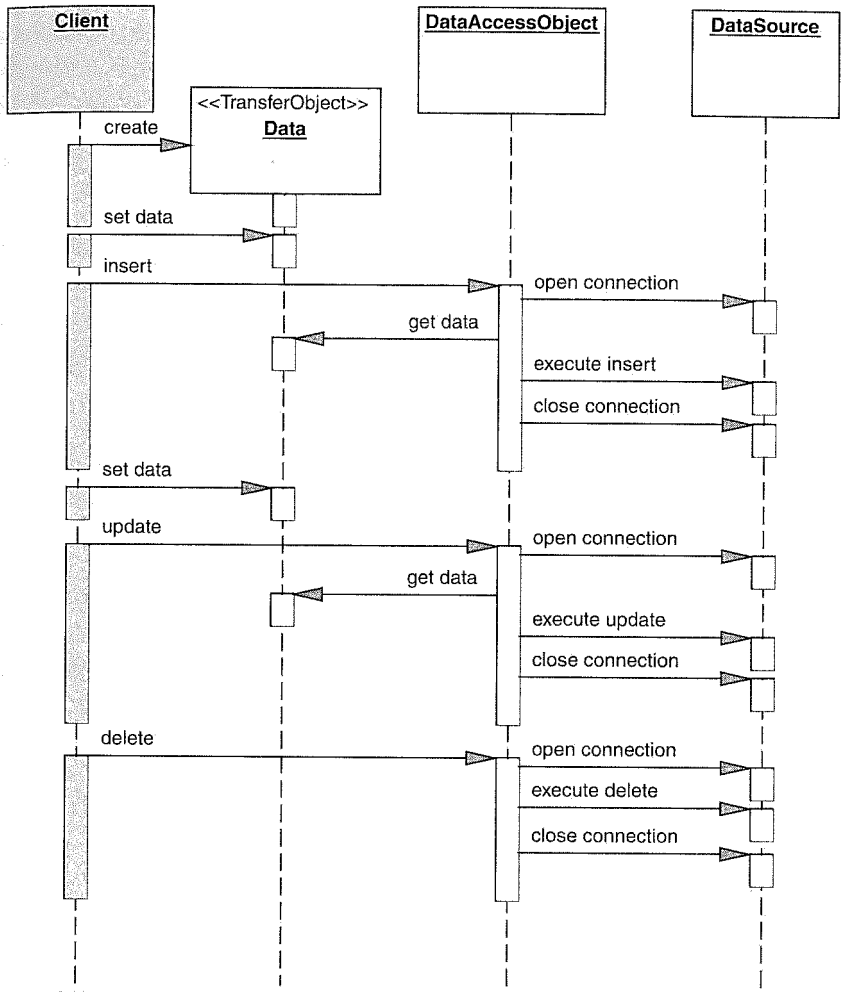


Figure 8.3 *Data Access Object* Sequence Diagram (part 2)

Client

The Client is an object that requires access to the data source to obtain and store data. The Client can be a *Business Object*, a *Session Façade*, an *Application Services* (357), a *Value List Handler* (444), a *Transfer Object Assembler* (433), or any other helper object that needs access to persistent data.

DataAccessObject

The DataAccessObject is the primary role object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the Client to enable transparent access to the data source. The DataAccessObject implements create (insert), find (load), update (store), and delete operations.

DataSource

The DataSource represents a data source implementation. A DataSource could be a database, such as an RDBMS, OODBMS, XML repository, flat file system, and so on. A DataSource can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

ResultSet

The ResultSet represents the results of a query execution. For an RDMBS DataSource, when an application is using JDBC API, this role is fulfilled by an instance of the java.sql.ResultSet.

Data

The Data represents a transfer object used as a data carrier. The DataAccessObject can use a *Transfer Object* (415) to return data to the client. The DataAccessObject could also receive the data from the client as a *Transfer Object* (415) to update the data in the data source.

Strategies

Custom Data Access Object Strategy

When you decide to write your own data access layer, you can use the *Data Access Object* and the various strategies described in this section. Note that most of these strategies are complementary, and can be combined to make your *Data Access Objects* (462) more powerful and flexible. However, some of these strategies might add too much complexity to your implementation, so use them judiciously.

In its most basic form, a *Data Access Object* provides the operations to create, delete, update, and find data in a database. A sample implementation for this is shown in Example 8.2. The *Data Access Object* typically exchanges data with its clients using *Transfer Objects* (415), as shown in Example 8.1. Furthermore, a *Data Access Object* also provides other methods to find multiple results (finder methods) to return a List or other Collection objects.

Spotify USA Inc. Exhibit 1008 Page 10

The sample code in Example 8.1 shows a *Transfer Object* (415) for customer data used by the CustomerDAO. The CustomerDAO sample implementation is shown in Example 8.2 with the methods to create, update, find, and delete a customer record from the database.

Example 8.1 CustomerTO.java: Transfer Object Used by CustomerDAO and Its Clients

```
1  package com.corej2eepatterns.to;
2
3  public class CustomerTO implements java.io.Serializable {
4    private String id;
5    private String name;
6    private String address;
7    . . .
8
9    public String getId(){ return id; }
10   public void setId(String id){ this.id = id; }
11   public String getName(){ return name; }
12   public void setName(String name){ this.name = name; }
13   public String getAddress(){ return address; }
14   public void setAddress(String address){
15     this.address = address;
16   }
17   // other getters and setters
18   . . .
19 }
```

Example 8.2 CustomerDAO.java: Data Access Object

```
1  package com.corej2eepatterns.dao;
2
3  // imports
4
5  public class CustomerDAO {
6    protected static final String FIELDS_INSERT =
7         "customer_name, customer_address, " +
8         "customer_contact, customer_phone, customer_email";
9
10   protected static final String FIELDS_RETURN =
11        "customer_id, " + FIELDS_INSERT;
12
13   protected static String INSERT_SQL =
14        "insert into customer ( " + FIELDS_INSERT +
```


Spotify USA Inc. Exhibit 1008 Page 11

```
15          " ) " + "values ( ?, ?, ?, ?, ?)";
16
17      protected static String SELECT_SQL = "select " +
18          FIELDS_RETURN +
19          " from customer where customer_id = ? ";
20
21      protected static String UPDATE_SQL =
22          "update customer set customer_name = ?, " +
23          "customer_address = ?, customer_contact = ?, " +
24          "customer_phone = ?, customer_email = ? " +
25          "where customer_id = ? ";
26
27      protected static String DELETE_SQL =
28          "delete from Customer where customer_id = ? ";
29
30      // the data source used to connect to the back-end database
31      private DataSource datasource;
32
33      public CustomerDAO() throws DAOException {
34        try {
35          // Shown here for clarity only. Typically, looking
36          // up a data source is done by a Service Locator
37          // and the DAO just uses the Service Locator to
38          // obtain a data source.
39          InitialContext initialContext =
40              new InitialContext();
41          datasource = (DataSource) initialContext.lookup(
42              OracleDAOFactory.DATASOURCE_DB_NAME);
43        } catch (NamingException e) {
44          throw new DAOException (
45              "Cannot locate data source at " +
46              DAOFactory.DATASOURCE_DB_NAME, e);
47        }
48      }
49
50      public String create(CustomerTO cust) throws DAOException {
51        // initialize variables
52        Connection con = getConnection();
53        String customerId = null;
54
55        PreparedStatement prepStmt = null;
56        try {
57          // create and setup statement
```

Data Access Object

Spotify USA Inc. Exhibit 1008 Page 12

```
58          prepStmt = con.prepareStatement(INSERT_SQL);
59          int i = 1;
60          prepStmt.setString(i++, cust.getName());
61          prepStmt.setString(i++, cust.getAddress());
62          . . .
63
64          // execute the statement
65          prepStmt.executeUpdate();
66
67          // obtain the newly created customer id value
68          . . .
69
70        } catch (Exception e) {
71          // handle exception
72        } finally {
73          // close connections
74        }
75
76        // return the newly created customer id value
77        return customerId;
78      }
79
80      public CustomerTO find(String customerId)
81        throws DAOException {
82        // initialize variables
83        CustomerTO cust = null;
84        Connection con = getConnection();
85        PreparedStatement prepStmt = null;
86        ResultSet rs = null;
87
88        try {
89          // setup statement and retrieve results
90          prepStmt = con.prepareStatement(SELECT_SQL);
91          prepStmt.setString(1, customerId);
92          rs = prepStmt.executeQuery();
93          if (rs.next()) {
94            //create the transfer object using data from rs
95            cust = new CustomerTO();
96            cust.setId(rs.getString(1));
97            cust.setName(rs.getString(2));
98            . . .
99          }
100       } catch (Exception e) {
```


Spotify USA Inc. Exhibit 1008 Page 13

```
101         // handle exception
102       } finally {
103         // close connections
104       }
105       return cust;
106     }
107
108     public void update(CustomerTO cust) throws DAOException {
109       Connection con = null;
110       PreparedStatement prepStmt = null;
111       try {
112         // prepare statement
113         con = getConnection();
114
115         prepStmt = con.prepareStatement(UPDATE_SQL);
116         int i = 1;
117
118         // add fields first
119         prepStmt.setString(i++, cust.getName());
120         prepStmt.setString(i++, cust.getAddress());
121         . . .
122
123         // now add where parameters
124         prepStmt.setString(i++, cust.getId());
125         int rowCount = prepStmt.executeUpdate();
126         prepStmt.close();
127         if (rowCount == 0) {
128           throw new DAOException(
129             "Update Error:Customer Id:" + cust.getId());
130         }
131       } catch (Exception e) {
132         // handle exception
133       } finally {
134         // close connections
135       }
136     }
137
138     public void delete(String customerId) throws Exception {
139       // setup variables
140       Connection con = getConnection();
141       PreparedStatement prepStmt = null;
142
143       try {
```

Data Access Object

Spotify USA Inc. Exhibit 1008 Page 14

```
144         // execute database update
145         prepStmt = con.prepareStatement(DELETE_SQL);
146         prepStmt.setString(1, customerId);
147         prepStmt.executeUpdate();
148       } catch (Exception e) {
149         // handle exception
150       } finally {
151         // close connections
152       }
153     }
154
155     // other methods for finders, etc.
156     . . .
157 }
```

Data Access Object Factory Strategies

You can make *Data Access Object* creation highly flexible by adopting the *Abstract Factory* [GoF] and the *Factory Method* [GoF] patterns.

When applications use a single type of persistent store (such as Oracle RDBMS), and there is no need to change the underlying storage from one implementation to another, implement the *DAO Factory Method* strategy to produce a variety of DAOs needed by the application.

Example 8.3 OracleDAOFactory.java: Data Access Object Strategy Using Factory Method [GoF]

```
1   package com.corej2eepatterns.dao;
2
3   // imports
4   public class OracleDAOFactory extends DAOFactory {
5
6     // package level constant used look up the
7     // DataSource name using JNDI
8     static String DATASOURCE_DB_NAME =
9         "java:comp/env/jdbc/CJPOraDB";
10
11    public CustomerDAO getCustomerDAO()
12        throws DAOException {
13      return (CustomerDAO) createDAO(CustomerDAO.class);
14    }
15
16    public EmployeeDAO getEmployeeDAO()
17        throws DAOException {
```

```
18        return (EmployeeDAO) createDAO(EmployeeDAO.class);
19      }
20
21      // create other DAO instances
22      . . .
23
24      // method to create a DAO instance. Can be optimized to
25      // cache the DAO Class instead of creating it everytime.
26      private Object createDAO(Class classObj)
27          throws DAOException {
28        // create a new DAO using classObj.newInstance() or
29        // obtain it from a cache and return the DAO instance
30      }
31  }
```

While the *DAO Factory Method* strategy is most commonly used, you can further extend the flexibility of the factory implementation by adopting the *Abstract Factory* pattern. This need for further flexibility arises if you're likely to change persistent stores frequently or if you use more than one. To accomplish this, you must encapsulate multiple types of data sources, such that each *Data Access Object* factory provides the DAO pattern implementation for one type of persistent store. While most applications typically use a single type of data source (such as Oracle RDBMS), and therefore the *DAO Factory Method* strategy might be adequate, more sophisticated and commercial frameworks can benefit from the added flexibility of the *DAO Abstract Factory* strategy.

The sample code for a *DAO Abstract Factory* strategy is shown in Example 8.4.

Example 8.4 DAOFactory.java: Data Access Object Factory Strategy Using Abstract Factory [GoF]


```
1   package com.corej2eepatterns.dao;
2
3   // imports
4
5   // Abstract class DAO Factory
6   public abstract class DAOFactory {
7
8     // List of DAO types supported by the factory
9     public static final int CLOUDSCAPE = 1;
10    public static final int ORACLE = 2;
11    public static final int SYBASE = 3;
```

Spotify USA Inc. Exhibit 1008 Page 16

```
12      . . .
13
14      // There will be a method for each DAO that can be
15      // created. The concrete factories will have to
16      // implement these methods.
17      public abstract CustomerDAO getCustomerDAO()
18          throws DAOException;
19      public abstract EmployeeDAO getEmployeeDAO()
20          throws DAOException;
21      . . .
22
23      public static DAOFactory getDAOFactory(int whichFactory) {
24        switch (whichFactory) {
25        case CLOUDSCAPE:
26          return new CloudscapeDAOFactory();
27        case ORACLE:
28          return new OracleDAOFactory();
29        case SYBASE:
30          return new SybaseDAOFactory();
31        . . .
32        default:
33          return null;
34        }
35      }
36  }
```

Transfer Object Collection Strategy

In addition to the basic functions of create, find, update, and delete, most application clients also need a number of methods that can find multiple results based on a search criteria. The *Data Access Object* can implement a number of **finder** methods that accept a search criteria and return a specified number of rows with the results. The *Data Access Object* creates an SQL statement using the transfer object criteria, and executes a query against the data source to obtain a ResultSet. The *Data Access Object* then processes the ResultSet to retrieve as many matching result rows as requested by the calling client. For each row, the *Data Access Object* creates a *Transfer Object* (415) and adds it to a collection that is returned to the client.

This strategy is suitable when the client needs to obtain a small set of results as transfer objects. If the result set obtained from the query execution is large, then this strategy can be greatly resource intensive because it must create a transfer object instance for data in each row. If you need larger results sets, consider

using strategies such as *CachedRowSet* strategy, *Read Only RowSet* strategy, or *RowSet Wrapper List* strategy.

A sample implementation of the *Transfer Object Collection* strategy is shown in Figure 8.5.

Example 8.5 CustomerDAO.java: Transfer Object Collection Strategy

```
1   package com.corej2eepatterns.dao;
2
3   // imports
4
5   public class CustomerDAO {
6   . . .
7
8     // Create a list of Transfer Objects and return it
9     public List findCustomers(CustomerTO criteria)
10         throws DAOException {
11
12      Connection con = getConnection();
13      ResultSet rs = null;
14      ArrayList custList = new ArrayList();
15      String searchSQLString = getSearchSQLString(criteria);
16
17      try {
18        con = getConnection();
19        java.sql.Statement stmt =
20             con.createStatement(. . . );.
21        rs = stmt.executeQuery(searchSQLString);
22        while(rs.next()) {
23          //create the transfer object using data from rs
24          cust = new CustomerTO();
25          cust.setId(rs.getString(1));
26          cust.setName(rs.getString(2));
27          . . .
28
29          // add the TO to the list
30          custList.add(cust);
31        }
32      } catch (Exception e) {
33        // handle exception
34      } finally {
35        // close connections
36      }
```

Data Access Object

Spotify USA Inc. Exhibit 1008 Page 18

```
37        return custList;
38     }
39
40     . . .
41
42  }
```

Cached RowSet Strategy

When performing an operation, such as a search, that returns multiple elements, the *Data Access Object* certainly has the option of creating a collection of **transfer objects** and returning it to the client. If the search operation returns a large list, this strategy can become quite expensive with all those transfer object creations. To avoid that performance overhead, you can leverage the JDBC RowSet interface by using a **disconnected, cached** row set implementation. Row sets can be used for read and updates, and you can choose from a variety of publicly available implementations. Sun's implementation is *CachedRowSet*, *JdbcRowSet* and *WebRowSet* [SUNRS]; other vendors such as Oracle provide their own *RowSet* implementations [ORARS]. You can use one of these public implementations if it meets your needs.

Exercise caution when using RowSets, connected or disconnected, with implementations that support the update functionality. If all you need is a results list that you use for read-only purposes (for instance, to display the results in a web page), a better alternative is to use the *Read Only RowSet* strategy or *RowSet Wrapper List* strategy.

Note on RowSet implementations: The JDBC team is working on providing a set of five standard RowSet implementations in an upcoming release, which will deprecate the sun.jdbc.rowset implementations [SUNRS]:

- JdbcRowSet - A connected RowSet providing JavaBean semantics.
- CachedRowSet - A disconnected RowSet, providing JavaBean semantics and robust synchronization mechanisms.
- WebRowSet - A disconnected RowSet provided synchronization mechanism that interact with XML data sources.
- FilteredRowSet - A disconnected RowSet that permits a filtered inbound and outbound view on the RowSet data.
- JoinRowSet - A disconnected RowSet that allows SQL JOIN relationships to be established between multiple CachedRowSets.

Spotify USA Inc. Exhibit 1008 Page 19

A sample DAO that uses a Cached RowSet implementation (sun.jdbc.rowset.CachedRowSet) is shown in Example 8.6.

Example 8.6 CustomerDAO.java: Cached RowSet Strategy

```
1   package com.corej2eepatterns.dao;
2
3   // imports
4
5   public class CustomerDAO {
6     . . .
7
8     // Create the CachedRowSet using the
9     // ResultSet from query execution
10    public RowSet findCustomersRS(CustomerTO criteria)
11        throws DAOException {
12
13      Connection con = getConnection();
14      javax.sql.RowSet rowSet = null;
15      String searchSQLString = getSearchSQLString(criteria);
16      try {
17        con = getConnection();
18        java.sql.Statement stmt =
19            con.createStatement(. . . );.
20        java.sql.ResultSet rs =
21            stmt.executeQuery(searchSQLString);
22        rowSet = new CachedRowSet();
23        rowSet.populate(rs);
24      } catch (SQLException anException) {
25        // handle exception…
26      } finally {
27        con.close();
28      }
29      return rowSet;
30    }
31
32    . . .
33
34  }
```


Read Only RowSet Strategy

While the *Cached RowSet* strategy is useful for creating any kind of RowSet, most J2EE applications need only the read-only functionality of a RowSet. Instead of

Spotify USA Inc. Exhibit 1008 Page 20

using an updateable RowSet implementation, you can implement your own RowSet that caters only to read-only functionality. This strategy provides an example of such a RowSet implementation called *Read Only RowSet*. The class diagram in Figure 8.4 shows the structure of the *Read Only RowSet* strategy.

The Client asks the DataAccessObject to perform a search. The DataAccessObject typically fulfills the role of the Client itself, as explained in the *Rowset Wrapper List* strategy.

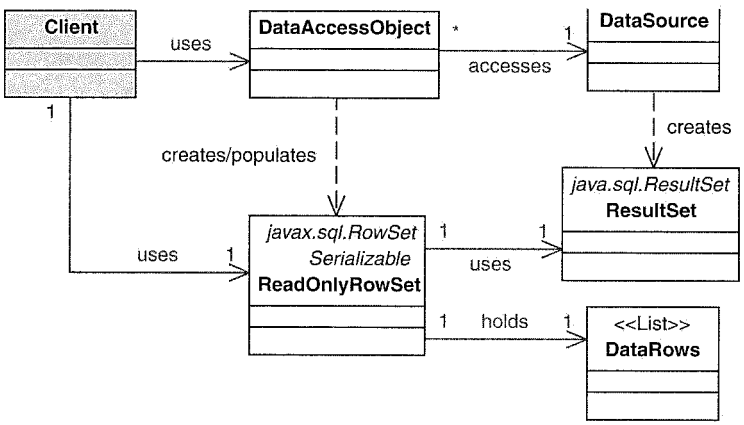


Figure 8.4 *Read Only Rowset* Strategy Class Diagram

The sequence diagram in Figure 8.5 on page 478 shows how the *Read Only RowSet* strategy works.

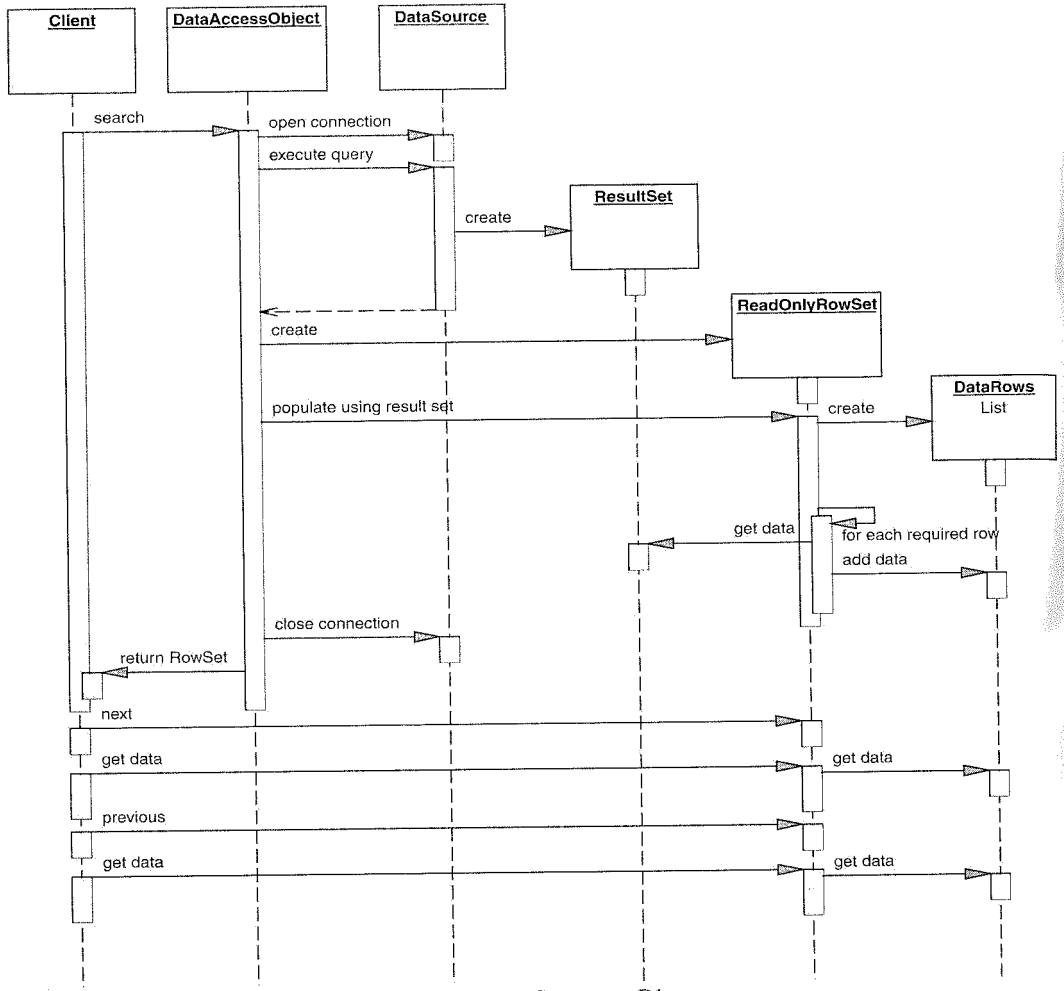


Figure 8.5 *Read Only RowSet* Strategy Sequence Diagram

Data Access Object

The Client uses a DataAccessObject to execute a search query. The DataAccessObject executes the query and obtains a ResultSet, which it uses to create an instance of ReadOnlyRowSet. The ReadOnlyRowSet extracts values from the given ResultSet and stores them in its own array shown as DataRows.

As the DataRows list is being populated, the ReadOnlyRowSet extracts only as many rows from the ResultSet as the Client requested. Once the rows are extracted and the DataRows list is populated with the requested number of rows, the connection is closed and the ResultSet is no longer needed. Hence, the ReadOnlyRowSet described in this strategy is a **disconnected cached RowSet** implementation.

Spotify USA Inc. Exhibit 1008 Page 22

The following code example shows the implementation of the *Read Only RowSet* strategy. The sample code in Example 8.7, a *Data Access Object*, creates a ReadOnlyRowSet instance obtaining a ResultSet instance after a query execution. The client of the CustomerDAO *Data Access Object* invokes the findCustomersRORS() method and passes a CustomerTO transfer object as the search criteria, indicating the maximum number of rows to return using the howManyRows argument.

Example 8.7 CustomerDAO.java: Creating the ReadOnlyRowSet

```
1  package com.corej2eepatterns.dao;
2
3  // imports
4
5  public class CustomerDAO {
6    . . .
7
8    // Create the ReadOnlyRowSet using the
9    // ResultSet from query execution
10   public RowSet findCustomersRORS(CustomerTO criteria,
11        int startAtRow, int howManyRows)
12        throws DAOException {
13
14     Connection con = getConnection();
15     javax.sql.RowSet rowSet = null;
16     String searchSQLString = getSearchSQLString(criteria);
17
18     try {
19       con = getConnection();
20       java.sql.Statement stmt = con.createStatement(. . .);
21       java.sql.ResultSet rs =
22            stmt.executeQuery(searchSQLString);
23       rowSet = new ReadOnlyRowSet();
24       rowSet.populate(rs, startAtRow, howManyRows);
25     } catch (SQLException anException) {
26       // handle exception...
27     } finally {
28       con.close();
29     }
30     return rowSet;
31   }
32
33   . . .
34 }
```


Spotify USA Inc. Exhibit 1008 Page 23

Example 8.8 ReadOnlyRowSet.java: Populating the RowSet

```
1   package com.corej2eepatterns.dao.rowset;
2
3   // imports
4
5   public class ReadOnlyRowSet implements RowSet, Serializable {
6
7     . . .
8
9     private Object[] dataRows;
10
11    . . .
12
13    /** this is a read only row set */
14    public boolean isReadOnly() {
15      return true;
16    }
17
18    public void setReadOnly(boolean flag) throws SQLException {
19      throw new SQLException(
20           "ReadOnlyRowSet: Method not supported");
21    }
22
23    // Populates the rowset without the first startRow rows
24    // of the ResultSet and with a maximum number
25    // of rows specified by howManyRows
26    public void populate(ResultSet resultSet,
27         int startRow, int howManyRows)
28         throws SQLException {
29
30      // miscellaneous code not shown for brevity…
31
32      // Create a list to hold the row values
33      List dataRows = . . . ;
34
35      // determine the number of columns from the mete data
36      int numberOfColumns =
37           resultSet.getMetaData().getColumnCount();
38
39      // Discard initial rows if beginAtRow was specified
40      setStartPosition(startAtRow, resultSet);
41
```


Spotify USA Inc. Exhibit 1008 Page 24

```
42      // if number of rows is unspecified,
43      // get all rows from resultset
44      if (howManyRows <= 0) {
45        howManyRows = Integer.MAX_VALUE;
46      }
47      int processedRows = 0;
48      while ((resultSet.next()) &&
49           (processedRows++ < howManyRows)) {
50        Object[] values = new Object[numberOfColumns];
51
52        // Read values for current row and save
53        // them in the values array
54        for (int i=0; i<numberOfColumns; i++) {
55          Object columnValue =
56               this.getColumnValue(resultSet, i);
57          values[i] = columnValue;
58        }
59
60        // Add the array of values to the linked list
61        dataRows.add(values);
62      }
63
64    } // end of row set constructor
65
66    . . .
67
68    // sets the result set to start at the given row number
69    private void setStartPosition(
70         int startAtRow, ResultSet resultSet)
71         throws SQLException {
72      if (startAtRow > 0) {
73        if (resultSet.getType() !=
74             ResultSet.TYPE_FORWARD_ONLY) {
75          // Move the cursor using JDBC 2.0 API
76          if (!resultSet.absolute(startAtRow)) {
77            resultSet.last();
78          }
79        } else {
80          // If the result set does not support JDBC 2.0
81          // skip the first beginAtRow rows
82          for (int i=0; i< startAtRow; i++) {
83            if (!resultSet.next()) {
84              resultSet.last();
```

```
85                  break;
86              }
87          }
88      }
89    }
90  }
91
92  // Reads a column value for the current row and
93  // create an appropriate java object to hold it.
94  // Return null if error reading value or for SQL null.
95  private Object getColumnValue(
96       ResultSet resultSet, int columnIndex) {
97
98    . . .
99
100 }
101
102 // implement the RowSet and ResultInterface methods
103 . . .
104
105 }
```

RowSet Wrapper List Strategy

While the *Transfer Object Collection* strategy is one way to implement your finder methods, it might prove expensive if the query returns a large set of results and you end up creating a large collection of transfer objects, which are used sparingly by the client. Clients typically search and use the first few result items and discard the rest. When an application is executing a query that returns a large set of results, the *RowSet Wrapper List* strategy might be more efficient though it requires additional work and adds more complexity.

Data Access Object

You do not want to expose the RowSet strategies to external clients of the *Data Access Object*, regardless of what RowSet implementation you choose to use. It is better to encapsulate all usage of any objects that originate in the java.sql and javax.sql packages inside the Data Access Object. Such encapsulation ensures that the client is not dependent on low-level JDBC interfaces, such as ResultSet or RowSet interfaces. The *RowSet Wrapper List* strategy internally uses the *Read Only RowSet* strategy to obtain the query results and wraps the RowSet instance to expose the results in an object-oriented way. The wrapping is done by a custom List implementation that provides the traversal and caching functions to manipulate the RowSet in an object-oriented manner by exposing the data in the rowset as objects.

Spotify USA Inc. Exhibit 1008 Page 26

The advantage of the *RowSet Wrapper List* strategy is that you create and return *Transfer Objects* (415) only if necessary; that is, if and when the client needs it. On the other hand, for the *Transfer Object Collection* strategy, the *Data Access Object* must create all the *Transfer Objects* (415) before it returns a Collection to the client.

The class diagram in Figure 8.6 shows the interactions of the *RowSet Wrapper List* strategy.

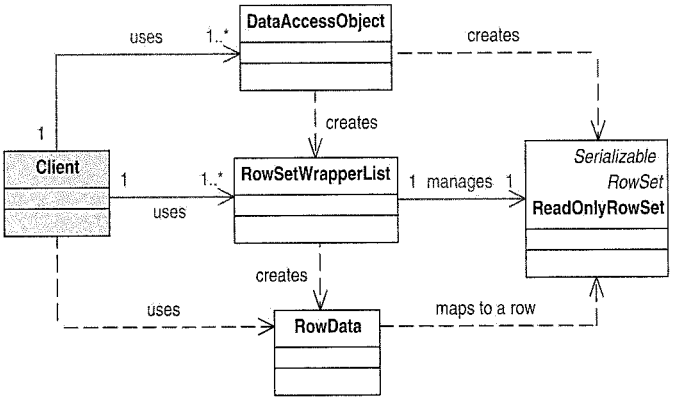


Figure 8.6 *Rowset Wrapper List* Strategy Class Diagram

The sequence diagram in Figure 8.7 shows the interactions of the *RowSet Wrapper List* strategy.

The DataAccessObject, having obtained a ReadOnlyRowSet using the *Read Only RowSet* strategy, proceeds to create a RowSetWrapperList by passing the ReadOnly-RowSet instance. The DataAccessObject returns the RowSetWrapperList to the Client, which uses that list to traverse the query results.

The RowSetWrapperList implements the List interface to provide the traversal function for the data in the ReadOnlyRowSet. The RowSetWrapperList encapsulates all data access layer functionality away from the Client, which does not have to know anything about RowSet, ResultSet, or any other classes in the java.sql or javax.sql packages. The RowSetWrapperList returns a *Transfer Object* (415) when the client wants to access the previous or next row in the results.

Spotify USA Inc. Exhibit 1008 Page 27

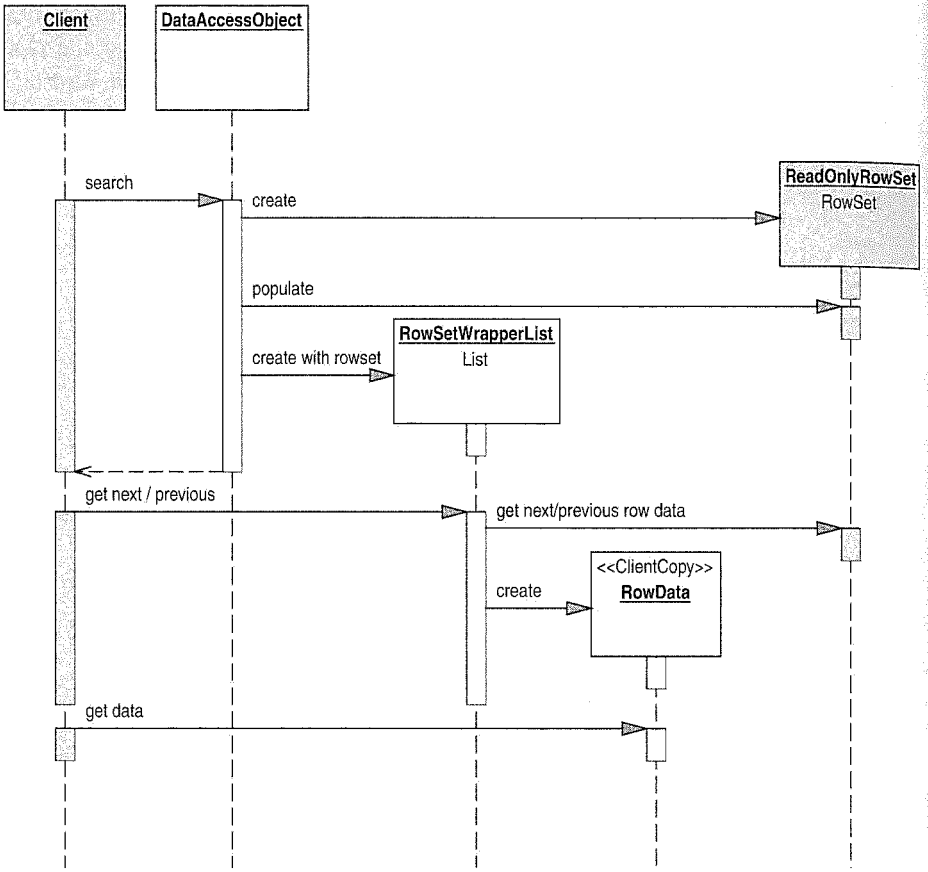


Figure 8.7 *RowSet Wrapper List* Strategy Sequence Diagram

Data Access Object

DESIGN NOTE

To further optimize this, instead of creating and returning a *Transfer Object* (415) every time the client requests the previous or next row, you can implement a cursor that is created only once. The RowSetWrapperList ensures that the cursor points to the correct row before the cursor is returned to the client. For the sake of simplicity and brevity, we have opted to use the *Transfer Object* (415) instead of a cursor.

The following code examples show the implementation of the *RowSet Wrapper List* strategy. The sample code in Example 8.9 shows the CustomerDAO creat-

ing a RowSetWrapperList instance using the ReadOnlyRowSet obtained in the previous example.

A custom list iterator, DataRowListIterator, is implemented as an inner class of RowSetWrapperList, as shown in Example 8.11.

Finally, a simple DAO client CustomerDAOClient that uses CustomerDAO and the RowSetWrapperList is shown in Example 8.13.

Example 8.9 CustomerDAO.java: Creating the RowSetWrapperList

```
1   package com.corej2eepatterns.dao;
2
3   // imports
4
5   public class CustomerDAO {
6     . . .
7
8     public List findCustomersRL(
9          CustomerTO cust, int startAtRow, int howManyRows)
10         throws Exception{
11       // create the search SQL string
12       String searchSQLString = getSearchSQLString(cust);
13
14       // execute the search
15       return executeSearch(searchSQLString,
16            StartAtRow, howManyRows);
17    }
18
19    private List executeSearch(
20         String searchSQLString, int startAtRow,
21         int howManyRows)
22         throws Exception {
23
24       RowSetWrapperList results = null;
25       try {
26         RowSet rowSet = getRORowSet(searchSQLString,
27              int startAtRow,
28              int howManyRows);
29
30         results = new RowSetWrapperList (rowSet);
31       } catch (Exception ex) {
32         throw new Exception(ex);
33       }
34
```

```
35        return results;
36      }
37
38      private String getSearchSQLString(CustomerTO cust) {
39        // convenience method to create and return
40        // a "SELECT" SQL statement as a String, incorporating
41        // non-null values in the supplied CustomerTO transfer
42        // object into the WHERE clause
43      }
44
45      . . .
46
47    }
```

Example 8.10 RowSetWrapperList.java

```
1   package com.corej2eepatterns.to.list;
2
3   // imports
4
5   // Read only implementation of the List interface.
6   // It supports iterators and absolute positioning.
7
8   public class RowSetWrapperList
9   implements List, Serializable {
10
11    // variable to hold a RowSet instance
12    private RowSet rowSet;
13
14
15    public RowSetWrapperList(RowSet rowSet) {
16      this.rowSet = rowSet;
17      . . .
18    }
19
20    // return the current row as a transfer object
21    public Object get(int index) {
22      try {
23        rowSet.absolute(index);
24      } catch (SQLException anException) {
25        // handle exception
26      }
27      // create a new transfer object and return
```

Data Access Object

Spotify USA Inc. Exhibit 1008 Page 30

```
28        return
29             TORowMapper.createCustomerTO(this);
30      }
31      . . .
32
33      // Returns a Sub List of the current list.
34      public List subList(int fromIndex, int toIndex) {
35        // Create a new RowSet with the required rows
36        ReadOnlyRowSet roRowSet = new ReadOnlyRowSet();
37        roRowSet.populate(this.rowSet, fromIndex, toIndex);
38
39        // Create a new RowSetWrapperList instance and
40        // return it
41        return
42             new RowSetWrapperList(roRowSet);
43      }
44
45      // Returns an iterator over the elements in this list in
46      // proper sequence. It is possible to define multiple
47      // independent iterators for the same RowSetWrapperList
48      // object.
49
50      public Iterator iterator() {
51        try {
52          rowSet.beforeFirst();
53        } catch (SQLException anException) {
54          System.out.println(
55               "Error moving RowSet before first row." +
56               anException);
57        }
58
59        return this.listIterator();
60      }
61
62      // Create a List Iterator that can iterate over the
63      // rowset
64      public ListIterator listIterator() {
65        // ListResultIterator is implemented as an inner class
66        return new DataRowListIterator();
67      }
68
69      // implement the List interface methods
70
```

```
71     . . .
72
73  }
```

Example 8.11 RowSetWrapperList.java: Inner class DataRowListIterator

```
1   package com.corej2eepatterns.to.lists;
2
3   // imports
4
5   public class RowSetWrapperList
6   implements List, Serializable {
7
8     . . .
9
10    private class DataRowListIterator implements ListIterator {
11      int currentRow=0;
12      // sets the rowset cursor to next and returns
13      // the Transfer object
14      public Object next() {
15      // get transfer object for next row of RowSetWrapperList
16      currentRow++;
17      return this.get(currentRow);
18    }
19
20    public Object previous() {
21      // get transfer object for previous row of
22      // RowSetWrapperList
23      currentRow--;
24      return this.get(currentRow);
25
26    }
27
28    // Implement the List Iterator interface
29    public boolean hasNext() {
30      // Check the cursor position in the rowSet using
31      // isLast, isAfterLast, isEmpty methods and
32      // return true or false accordingly
33    }
34
35    public boolean hasPrevious() {
36      // Check the cursor position in the rowSet using
37      // isFirst, isBeforeFirst, isEmpty methods and
```


Spotify USA Inc. Exhibit 1008 Page 32

```
38        // return true or false accordingly
39      }
40
41      // implement other ListIterator methods
42
43      public int nextIndex() {
44        . . .
45      }
46
47      public int previousIndex() {
48        . . .
49      }
50
51      // optional methods not implemented throw
52      // UnsupportedException
53      public void set(Object o) {
54        throw new UnsupportedOperationException();
55      }
56
57      public void add(Object o) {
58        throw new UnsupportedOperationException();
59      }
60      . . .
61  }
```

Example 8.12 TORowMapper.java: A Sample Implementation

```
1   package com.corej2eepatterns.util;
2
3   // imports
4   public class TORowMapper {
5
6     // create Customer TO
7     public CustomerTO createCustomerTO(RowSet rowSet) {
8       CustomerTO to = new CustomerTO();
9       to.setId(getString(rowSet), 0);
10      to.setName(getString(rowSet), 1);
11      . . .
12    }
13
14    // create other TOs
15    . . .
16
```

```
17    // implement primitive methods used by create methods
18    protected boolean wasNull(RowSet rowSet) {
19      try {
20        return rowSet.wasNull();
21      } catch (SQLException e) {
22        throw new RuntimeException(e.getMessage());
23      }
24    }
25
26    protected String getString(RowSet rowSet, int columnIndex) {
27      try {
28        return rowSet.getString(columnIndex);
29      } catch (SQLException e) {
30        throw new RuntimeException(e.getMessage());
31      }
32    }
33
34    protected boolean getBoolean(
35        RowSet rowSet, int columnIndex) {
36      try {
37        return rowSet.getBoolean(columnIndex);
38      } catch (SQLException e) {
39        throw new RuntimeException(e.getMessage());
40      }
41    }
42
43    protected java.util.Date getDate(
44        RowSet rowSet, int columnIndex) {
45      try {
46        return rowSet.getDate(columnIndex);
47      } catch (SQLException e) {
48        throw new RuntimeException(e.getMessage());
49      }
50    }
51
52    // Other primitive getXXX methods for all required
53    // data types
54    . . .
55
56  }
```

Data Access Object

Example 8.13 A Sample DAO Client: Using the RowSetWrapperList

```
1  package com.corej2eepatterns.rowset;
2
3  // imports
4
5  public class CustomerDAOClient {
6
7    . . .
8
9    public void search() {
10     try {
11       CustomerDAO dao = new CustomerDAO();
12       CustomerTO criteria = new CustomerTO();
13
14       criteria.setZip("94539");
15
16       // search for all customers with Zip code 94539
17       // results to contain a maximum of 1000 matching rows
18       List searchResults =
19            dao.findCustomersRL(criteria, 0, 999);
20
21       int resultSize = searchResults.size();
22       for (int rowNum=0; rowNum < resultSize; rowNum++) {
23         CustomerTO customerTO =
24              (CustomerTO)searchResults.get(rowNum);
25         System.out.println("Customer Row #" +
26              rowNum + " has ");
27         System.out.println("Customer Id = " +
28              customerTO.getId());
29         System.out.println("Name = " +
30              customerTO.getName());
31         . . .
32       }
33
34       . . .
35
36       // get a ListIterator
37       ListIterator iterator = searchResults.listIterator();
38
39       // use the iterator to traverse to next or
40       // previous row
41       . . .
```


Spotify USA Inc. Exhibit 1008 Page 35

```
42
43        } catch (Exception e) {
44          // handle exception
45        }
46      }
47      . . .
48  }
```

DESIGN NOTE: DESIGNING A DATA ACCESS OBJECT

As noted in various places in the *Data Access Object* section, we use *Transfer Object* (415) to pass data between a *Data Access Object* and its clients. The question arises, what object do we return to the client when executing a query?

Though the *Data Access Object* creates a ResultSet instance when the query is executed, it's best to keep the ResultSet encapsulated within the *Data Access Object* at all times. A ResultSet instance is associated with a database connection. Returning a ResultSet to a client outside the data access layer would violate encapsulation and introduce connection management problems. Another problem is that the ResultSet is a low-level interface and is tied to the relational structure of the data in an RDBMS. It is important to recall that the *Data Access Object* caters to other types of persistent stores as well, and therefore it's all the more important to avoid exposing implementation-specific data structures, such as ResultSet instances outside the data access layer. Specifically, the problems arising out of exposing such data structures to the clients are as follows:

- The clients must import and deal with interfaces and exceptions from the java.sql package.
- To extract data from the result set, a client must specify database columns or aliases. Therefore, the client is aware of details about the persistence implementation, such as relational tables and columns.
- The ResultSet carries with it any associated live database connections.
- RowSets provide a slightly higher level of abstraction than a result set and also provide additional methods that make it more convenient to use. While a RowSets instances appear to be a better way to return results from a *Data Access Object*, this approach also has other problems. The clients will have to import and deal with interfaces and exceptions from the java.sql and javax.sql packages.
- To extract data from the row set, the clients must specify database columns or aliases, making the clients aware of details about the persistence implementation, such as relational tables and columns.
- When using a connected RowSet, the database connection is still alive when the row set is returned to the clients of the *Data Access Object*.
- The RowSet is close to a ResultSet; it implements the ResultSet interface.

In spite of the issues highlighted here, you might think it appropriate to return a RowSet instance from the *Data Access Object*, depending on how you intend to use the

Spotify USA Inc. Exhibit 1008 Page 36

RowSet. Therefore, using *Transfer Object* (415) to pass data between the *Data Access Object* and its clients solves the problem of coupling the clients to the JDBC API.

Furthermore, for finder methods, using either the simpler *Transfer Object Collection* strategy or the more complex *RowSet Wrapper List* strategy is a better option for returning results to external clients of the *Data Access Objects*. These strategies use row set implementations under the covers. More importantly, these strategies provide an object-oriented access to results data and avoid dealing with data structures such as, ResultSet or RowSet from the JDBC API.

Consequences

- ***Enables transparency***
  Clients can leverage the encapsulation of data sources within the *Data Access Objects* to gain transparency to the location and implementation of the persistent storage mechanisms.
- ***Provides object-oriented view and encapsulates database schemas***
  The clients use transfer objects or data cursor objects (*RowSet Wrapper List* strategy) to exchange data with the *Data Access Objects*. Instead of depending on low-level details of database schema implementations, such as ResultSets and RowSets, where the clients must be aware of table structures, column names, and so on, the clients handle data in an object-oriented manner using the transfer objects and data cursors.
- ***Enables easier migration***
  A layer of DAOs makes it easier for an application to migrate to a different database implementation. The clients have no knowledge of the underlying data store implementation. Thus, the migration involves changes only to the DAO layer.

Spotify USA Inc. Exhibit 1008 Page 37

- ***Reduces code complexity in clients***
  Since the DAOs encapsulate all the code necessary to interact with the persistent storage, the clients can use the simpler API exposed by the data access layer. This reduces the complexity of the data access client code and improves the maintainability and development productivity.
- ***Organizes all data access code into a separate layer***
  Data access objects organize the implementation of the data access code in a separate layer. Such a layer isolates the rest of the application from the persistent store and external data sources. Because all data access operations are now delegated to the DAOs, the separate data access layer isolates the rest of the application from the data access implementation. This centralization makes the application easier to maintain and manage.
- ***Adds extra layer***
  The DAOs create an additional layer of objects between the data client and the data source that needs to be designed and implemented, to leverage the benefits of this pattern. While this layer might seem to be extra development and run-time overhead, it is typically necessary in order to decouple the data access implementation from the other parts of the application.
- ***Needs class hierarchy design***
  When you use a factory strategy, the hierarchy of concrete factories and the hierarchy of concrete products (DAOs) produced by the factories need to be designed and implemented. Consider this additional effort if you think you'll need this extra flexibility, because it increases the complexity of the design. Use the *DAO Factory Method Strategy* if that meets your needs, and use *DAO Abstract Factory Strategy* only if absolutely required.
- ***Introduces complexity to enable object-oriented design***
  While the *RowSet Wrapper List* strategy encapsulates the data access layer dependencies and JDBC APIs, and exposes an object-oriented view of the results data, it introduces considerable complexity in your implementation. You need to decide whether its benefits outweigh the drawbacks of using the JDBC RowSet API in the *Cached RowSet* strategy, or the performance drawback of using the *Transfer Object Collection* strategy.

Spotify USA Inc. Exhibit 1008 Page 38

Related Patterns

- ***Transfer Object (415)***
  Data access objects in their most basic form use transfer objects to transport data to and from their clients. *Transfer Object* (415) are used in other strategies of *Data Access Objects*. The *RowSet Wrapper List* strategy returns a list as a transfer object.
- ***Factory Method [GoF] and Abstract Factory [GoF]***
  The *Data Access Object Factory* strategies use the *Factory Method* pattern to implement the concrete factories and its products (DAOs). For added flexibility, the *Abstract Factory* pattern is used as described in the *Data Access Object Factory* strategy.
- ***Broker [POSA1]***
  The DAO pattern is related to the *Broker* pattern, which describes approaches for decoupling clients and servers in distributed systems. The DAO pattern more specifically applies this pattern to decoupling the resource tier from clients in another tier, such as the business or presentation tier.
- ***Transfer Object Assembler (433)***
  The *Transfer Object Assembler* (433) uses the *Data Access Object* to obtain data to build the composite transfer object it needs to assemble and send as the model data to the client.
- ***Value List Handler (444)***
  The value list handler needs a list of results to act upon. Such a list is often obtained using a *Data Access Object*, which can return a set of results. While the value list handler has an option of obtaining a RowSet from the *Data Access Object*, it might be better to obtain and use a *RowSet Wrapper List* instead.

Spotify USA Inc. Exhibit 1008 Page 39



"The Java landscape is littered with libraries, tools, and specifications. What's been lacking is the expertise to fuse them into solutions to real-world problems. These patterns are the intellectual mortar for J2EE software construction."

–John Vlissides, co-author of **Design Patterns**, *the "Gang of Four" book*

"The authors of ***Core J2EE Patterns*** have harvested a really useful set of patterns. They show how to apply these patterns and how to refactor your system to take advantage of them. It's just like having a team of experts sitting at your side."

–Grady Booch, Chief Scientist, Rational Software Corporation

"The authors do a great job describing useful patterns for application architectures. The section on refactoring is worth the price of the entire book!"

–Craig McClanahan, Struts Lead Architect and Specification Lead for JavaServer Faces

"***Core J2EE Patterns*** is the gospel that should accompany every J2EE application server…Built upon the in-the-trenches expertise of its veteran architect authors, this volume unites the platform's many technologies and APIs in a way that application architects can use, and provides insightful answers to the whys, whens, and hows of the J2EE platform."

–Sean Neville, JRun Enterprise Architect, Macromedia

Developers often confuse learning the technology with learning to design with the technology. In this book, senior architects from the Sun Java Center share their cumulative design experience on Java 2 Platform, Enterprise Edition (J2EE) technology.

The primary focus of the book is on patterns, best practices, design strategies, and proven solutions using the key J2EE technologies including JavaServer Pages™ (JSP™), Servlets, Enterprise JavaBeans™ (EJB™), and Java™ Message Service (JMS) APIs. The J2EE Pattern Catalog with 21 patterns and numerous strategies is presented to document and promote best practices for these technologies.

Core J2EE Patterns, Second Edition offers the following:

- J2EE Pattern Catalog with 21 patterns–fully revised and newly documented patterns providing proven solutions for enterprise applications
- Design strategies for the presentation tier, business tier, and integration tier
- Coverage of servlets, JSP, EJB, JMS, and Web Services
- J2EE technology bad practices
- Refactorings to improve existing designs using patterns
- Fully illustrated with UML diagrams
- Extensive sample code for patterns, strategies, and refactorings

About the Authors

DEEPAK ALUR is an Enterprise Java Architect with the Sun Java Center with over 14 years of experience. He remains focused on architecture, design, and implementation of large-scale Enterprise applications using Object-Oriented technologies, patterns, and Java and J2EE technologies.

JOHN CRUPI is a Distinguished Engineer and the Chief Java Architect of the Sun Java Center. He has over 17 years of experience in distributed object computing and remains focused on creating reusable, scalable J2EE architectures and bringing patterns to the next level.

DAN MALKS is a Principal Engineer with the Sun Java Center. He has over 16 years of experience and focuses on Object-Oriented technologies and their application within Enterprise and Web Services-based projects. His publications include industry periodicals and books on Java and J2EE technologies and patterns.

$49.99 U.S./$75.99 Canada

PRENTICE HALL
Professional Technical Reference
Upper Saddle River, NJ 07458
www.phptr.com



