J2EE Data Access Objects

A brief article for developers

Prepared By Owen Taylor

The Data Access Object Pattern

In this paper, we will discuss the popular J2EE Design Pattern known as the *Data Access Object* Pattern. The article is useful for people who may not have time to read a whole book, but may be interested in learning about a snippet of knowledge as they find the time.

Motivation

In the J2EE world, there are different specifications and mechanisms for accessing persistent storage and legacy data. In an Enterprise JavaBeans application, the data source access code might be in a Session Bean or an Entity Bean. In a Web application, the code might be in a servlet or a helper class for a JSP.

If data access is coded directly into business components, the components become tightly coupled to the data source. This happens for a variety of reasons:

- Persistent storage APIs vary, depending on the vendor
- Some APIs may be completely proprietary
- With JDBC, SQL statements may vary according to the database product

Because of this, often times if the data source changes, the components must change as well. Furthermore, coding data access directly into components also limits reuse of the data access code.

To make J2EE applications flexible, it is good practice to factor data access out of business components. A way must be found to abstract business logic from detailed knowledge of where the data resides and how it should be managed.

Solution

The *Data Access Object (DAO) pattern* separates the access to a data source the business components. Business components are no longer coupled to a specific vendor implementation or API. Application developers can control how data is accessed; if a vendor or an API changes later, the DataAccessObject is the only code that needs to change.

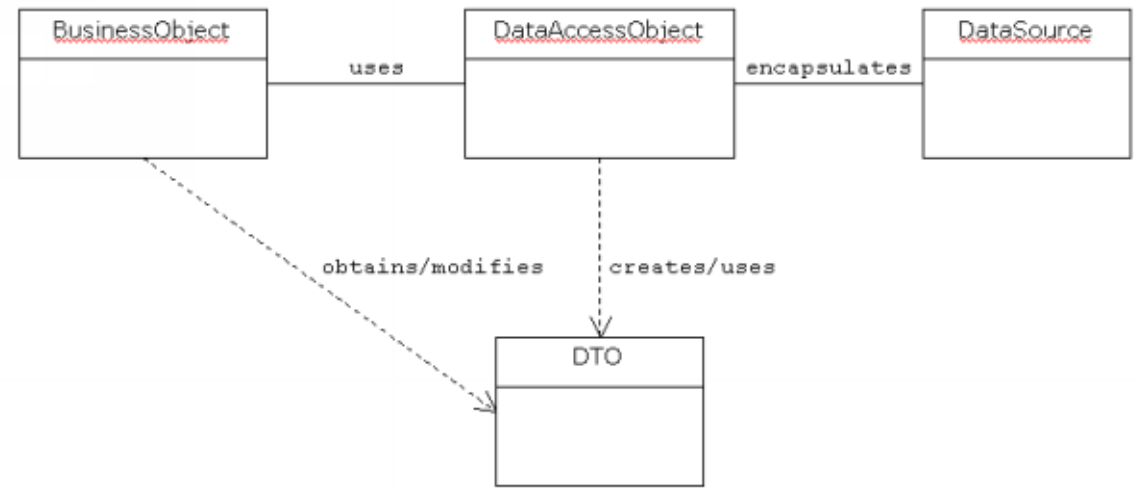
A DAO has two main functions:

- The DAO contains logic to access a data source
- The DAO manages the connection to the data source to obtain and store data

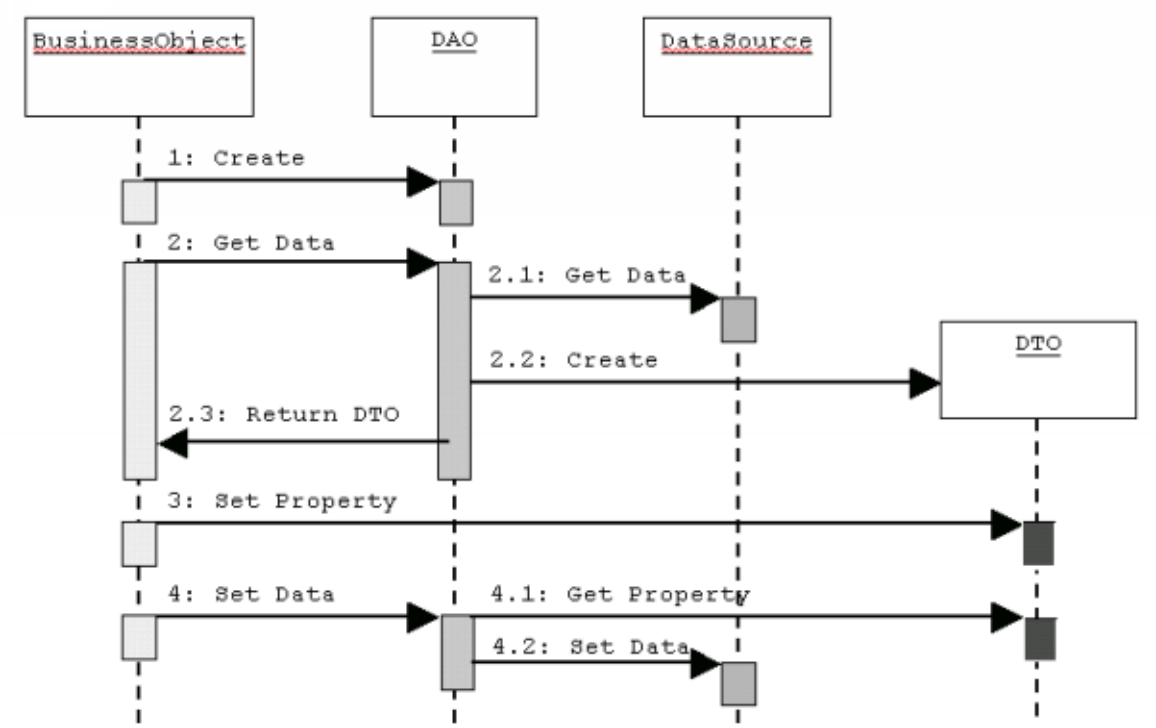
A DAO is used by a business component. The DAO access the underlying data source, and returns results back to the business component. The data that is sent back and forth between a business component and a DAO are encapsulated inside of another special object that we will call a *Data Transfer Object (DTO)*, also commonly known as a *value object (VO)*.

Structure

The following class diagram shows the relationships for the Data Access Object Pattern. The DataAccessObject (DAO) encapsulates access to the DataSource and acts as a factory and manager for Data Transfer Objects.



The following sequence diagram shows the interaction among the participants in the Data Access Object pattern. The BusinessObject creates the DataAccessObject and uses it to get data. The DataAccessObject encapsulates access to a DataSource. The Data Access Object also manages creation of and updates to data in the DataSource. A Data Access Object consists of getter and setter methods to manipulate the data.



As you can see, the DAO is responsible for creating Data Transfer Objects (DTOs), which are convenient means to pass data between BusinessObjects and DAOs. Once the BusinessObject completes processing of the DTO, it can return the DTO to the DAO. The DAO can write the changes out to the DataSource to complete the transaction.

Sample Code

The following code example will illustrate an implementation of the Data Access Object pattern. For brevity, the code example will not include data source access or JDBC examples. The object model we will use is an inventory tracking system.

The Data Transfer Object (DTO) code

The first class is an *Item* which is a Data Transfer Object that the DAO can return or manipulate to make changes to the data source.

```
//Item.java

package com.test;

public class Item {

      private String name;
      private int count;
      private float cost;

      public String getItemName() { … }
      public void setItemName(String name) { … }
      public int getItemCount() { … }
      public void setItemName(int count) { … }
      public float getItemCost() { … }
      public void setItemCost(float cost) { … }
}
```

The exception class code

We will also create an exception class to encapsulate any kind of data access difficulty.

```
//DataAccessException.java

package com.test;

public class DataAccessException extends Exception {
   Throwable whyException;

   public DataAccessException(String message){
     super(message);
   }

   public DataAccessException(String message, Throwable t){
     super(message);
      this.whyException = t;
   }

   public void printStackTrace(){
      if(! (whyException==null)){
        System.err.println("DATA ACCESS ISSUE: ");
        whyException.printStackTrace();
      }
   }
}
```

The Data Access Object (DAO) interface code

The following interface will be implemented by all concrete implementations of the Inventory DAO. The implementations are responsible for managing resources (connections) and reading and writing data. The interface ties all implementations together with a single type. This is convenient if we decide to use a factory to create or access Inventory DAOs.

```
//InventoryDAO.java

package com.test;

public interface InventoryDAO {

      public Collection getAllItems()throws DataAccessException;

      public void addItem(Item i) throws DataAccessException;

      public void removeItem(Item i) throws DataAccessException;
}//end of InventoryDAO interface
```

The Data Access Object (DAO) implementation code

Our DAO implementation, *JDBCInventoryDAO*, is a simple implementation of the InventoryDAO interface.

```
//JDBCInventoryDAO.java

package com.test;

import java.sql.*;
import java.util.*;

public class JDBCInventoryDAO implements InventoryDAO {
      public JDBCInventoryDAO() {
        // Initialize access to the database (not shown)
      }
      public Collection getAllItems() throws DataAccessException{
        //JDBC Access code not shown
      }
      public void addItem(Item I) throws DataAccessException{
        //JDBC Insert code not shown
      }
      public void removeItem(Item I) throws DataAccessException{
        //JDBC Delete code not shown
      }
}
```

The client code

Because we are implementing a solution within the context of J2EE, our client is shown as an EJB Session Bean.

This simple client illustrates how to use a DAO. Notice that the client is completely abstracted from any data source implementation details. The client does not have to import any data source specific packages and simply creates a DAO to access items.

```
//InventoryClient.java

package com.test;
import java.util.*;
import java.ejb.*;

public class InventoryClientBean implements SessionBean {

     /*  Code required to be a SessionBean not shown. . .*/

  public void WorkWithInventory() {
    try {
      InventoryDAO dao = new JDBCInventoryDAO();
      Collection items = dao.getAllItems();
      // do something with the items
    }

    catch (DataAccessException dae) {
           //handle the exception
    }
  }
}//end of InventoryClientBean
```

Summary

By defining an interface InventoryDAO we have defined a specific context within which all of the DAO implementations must be created. The more specific our context, the easier it is to build and use appropriate implementations, such as:

- JDO
- JDBC
- EJB CMP entity beans
- EJB BMP entity beans

It is always up to the design team on a particular project to determine the breadth of the context for both encapsulating business responsibilities and creational responsibilities. In anticipation of future changes, we encapsulated the responsibilities within our **InventoryDAO** interface. We are free to implement these responsibilities in any new Objects as we choose. (As long as they implement **InventoryDAO**).

This pattern illustrates one of our goals as designers, which should be to follow the *Open-closed principle*:

Design applications so that they can be extended without modification to the existing code. They will therefore be:

- ***Open to new features.***
- ***Closed to changes in the code.***

In summary, the values of this design pattern are as follows:

- Enables persistence transparency
- Enables easier migration
- Reduces code complexity in Business Objects
- Centralizes all data access into a separate layer

There are a few considerations one should take into account, however:

- This pattern is less important when CMP is used, since CMP encapsulates data access by definition, unless a legacy system demands it.
- Using DAOs adds an extra layer to your application
- It requires a somewhat complex class hierarchy design

Related Patterns

- Data Transfer Object (or Value Object)
- Bridge
- Adapter
- Factory
- Abstract Factory

Closing Thoughts

This article is one in a series of articles available for download on TheServerSide.com that The Middleware Company is making available to Java developers. The Middleware Company believes that learning patterns such as the ones discussed in this white paper can mean the difference between failure and success when developing J2EE applications. Patterns can save time and money and avoid great frustration when used properly. And it can also help you…

- Increase your value as a Java developer in today's economy
- Become a better Enterprise Java programmer
- Succeed on that upcoming project

Teaching you to apply them in the correct context, understand when they can be combined to form compound patterns, and recognize anti-patterns is also the goal of a new advanced *J2EE Patterns* course that is available immediately for registration worldwide.

The J2EE Patterns course is:

- An intensive 5-day course, purely on the subject of J2EE Patterns
- A hardcore, advanced training course for the seasoned J2EE programmer
- Requires knowledge of J2EE programming, and is chock-full of hands-on lab exercises where students gain experience programming with J2EE design patterns

You can learn more about it here:
http://www.middleware-company.com/training/j2eepatterns.shtml