Core J2EE Pattern Catalog

Core J2EE Patterns - Data Access Object

Print-friendly Version

Context

Access to data varies depending on the source of the data. Access to persistent storage, such as to a database, varies greatly depending on the type of storage (relational databases, object-oriented databases, flat files, and so forth) and the vendor implementation.

Problem

Many real-world Java 2 Platform, Enterprise Edition (J2EE) applications need to use persistent data at some point. For many applications, persistent storage is implemented with different mechanisms, and there are marked differences in the APIs used to access these different persistent storage mechanisms. Other applications may need to access data that resides on separate systems. For example, the data may reside in mainframe systems, Lightweight Directory Access Protocol (LDAP) repositories, and so forth. Another example is where data is provided by services through external systems such as business-to-business (B2B) integration systems, credit card bureau service, and so forth.

Typically, applications use shared distributed components such as entity beans to represent persistent data. An application is considered to employ bean-managed persistence (BMP) for its entity beans when these entity beans explicitly access the persistent storage-the entity bean includes code to directly access the persistent storage. An application with simpler requirements may forego using entity beans and instead use session beans or servlets to directly access the persistent storage to retrieve and modify the data. Or, the application could use entity beans with container-managed persistence, and thus let the container handle the transaction and persistent details.

Applications can use the JDBC API to access data residing in a relational database management system (RDBMS). The JDBC API enables standard access and manipulation of data in persistent storage, such as a relational database. The JDBC API enables J2EE applications to use SQL statements, which are the standard means for accessing RDBMS tables. However, even within an RDBMS environment, the actual syntax and format of the SQL statements may vary depending on the particular database product.

There is even greater variation with different types of persistent storage. Access mechanisms, supported APIs, and features vary between different types of persistent stores such as RDBMS, object-oriented databases, flat files, and so forth. Applications that need to access data from a legacy or disparate system (such as a mainframe, or B2B service) are often required to use APIs that may be proprietary. Such disparate data sources offer challenges to the application and can potentially create a direct dependency between application code and data access code. When business components-entity beans, session beans, and even presentation components like servlets and helper objects for JavaServer Pages (JSP) pages --need to access a data source, they can use the appropriate API to achieve connectivity and manipulate the data source. But including the connectivity and data access code within these components introduces a tight coupling between the components and the data source 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 need to be changed to handle the new type of data source.

Forces

- Components such as bean-managed entity beans, session beans, servlets, and other objects like helpers for JSP pages need to retrieve and store information from persistent stores and other data sources like legacy systems, B2B, LDAP, and so forth.
- Persistent storage APIs vary depending on the product vendor. Other data sources may have APIs that are nonstandard and/or proprietary. These APIs and their capabilities also vary depending on the type of storage-RDBMS, object-oriented database management system (OODBMS), XML documents, flat files, and so forth. There is a lack of uniform APIs to address the requirements to access such disparate systems.
- Components typically use proprietary APIs to access external and/or legacy systems to retrieve and store data.
- Portability of the components is directly affected when specific access mechanisms and APIs are included in the components.
- Components need to be transparent to the actual persistent store or data source implementation to provide easy migration to different vendor products, different storage types, and different data source types.

Solution

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

The DAO implements the access mechanism required to work with the data source. The data source could be a persistent store like an RDBMS, an external service like a B2B exchange, a repository like an LDAP database, or a business service accessed via CORBA Internet Inter-ORB Protocol (IIOP) or low-level sockets. The business component that relies on the DAO 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 pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the component and the data source.

Structure

Figure 9.1 shows the class diagram representing the relationships for the DAO pattern.

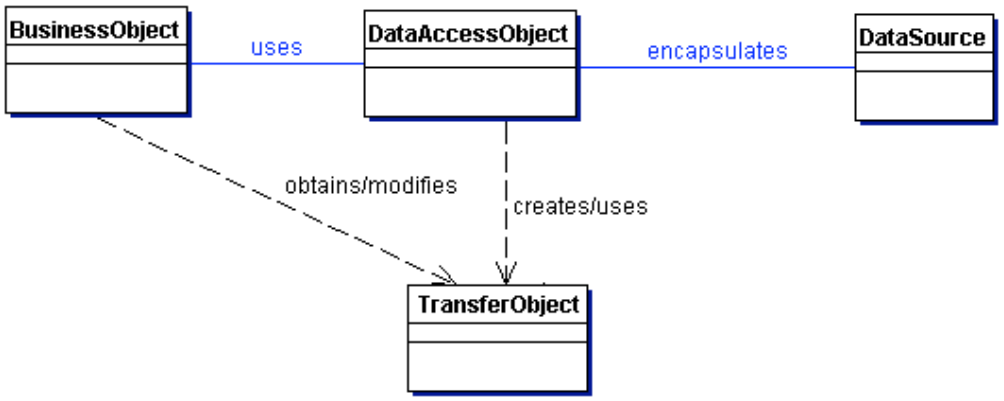

Figure 9.1 Data Access Object

Participants and Responsibilities

Figure 9.2 contains the sequence diagram that shows the interaction between the various participants in this pattern.

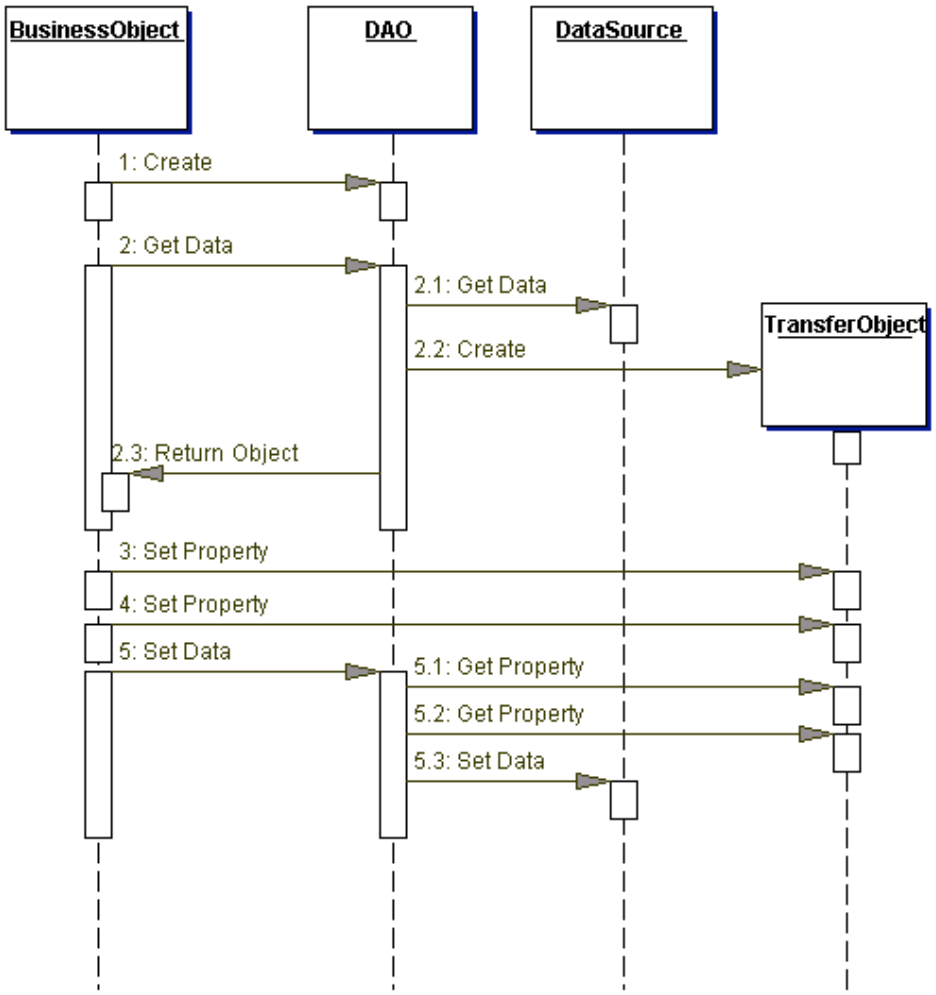

Figure 9.2 Data Access Object sequence diagram

BusinessObject

The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A

BusinessObject may be implemented as a session bean, entity bean, or some other Java object, in addition to a servlet or helper bean that accesses the data source.

DataAccessObject
The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

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

TransferObject
This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

Strategies
Automatic DAO Code Generation Strategy
Since each BusinessObject corresponds to a specific DAO, it is possible to establish relationships between the BusinessObject, DAO, and underlying implementations (such as the tables in an RDBMS). Once the relationships are established, it is possible to write a simple application-specific code-generation utility that generates the code for all DAOs required by the application. The metadata to generate the DAO can come from a developer-defined descriptor file. Alternatively, the code generator can automatically introspect the database and provide the necessary DAOs to access the database. If the requirements for DAOs are sufficiently complex, consider using third-party tools that provide object-to-relational mapping for RDBMS databases. These tools typically include GUI tools to map the business objects to the persistent storage objects and thereby define the intermediary DAOs. The tools automatically generate the code once the mapping is complete, and may provide other value-added features such as results caching, query caching, integration with application servers, integration with other third-party products (e.g., distributed caching), and so forth.

Factory for Data Access Objects Strategy
The DAO pattern can be made highly flexible by adopting the Abstract Factory [GoF] and the Factory Method [GoF] patterns (see "Related Patterns" in this chapter).

When the underlying storage is not subject to change from one implementation to another, this strategy can be implemented using the Factory Method pattern to produce a number of DAOs needed by the application. The class diagram for this case is shown in Figure 9.3.

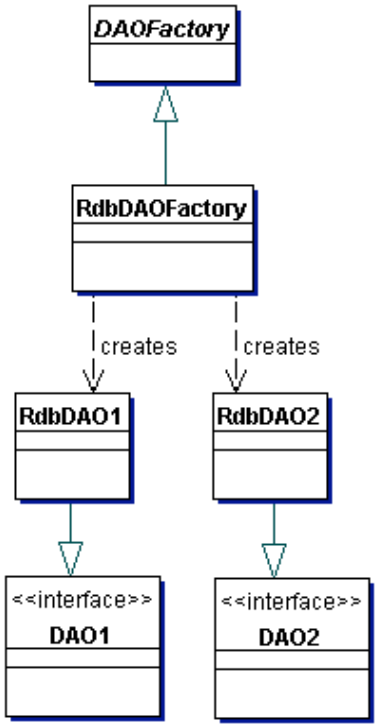

Figure 9.3 Factory for Data Access Object strategy using Factory Method

When the underlying storage is subject to change from one implementation to another, this strategy may be implemented using the Abstract Factory pattern. The Abstract Factory can in turn build on and use the Factory Method implementation, as suggested in *Design Patterns: Elements of Reusable Object-Oriented Software* [GoF]. In this case, this strategy provides an abstract DAO factory object (Abstract Factory) that can construct various types of concrete DAO factories, each factory supporting a different type of persistent storage implementation.

Once you obtain the concrete DAO factory for a specific implementation, you use it to produce DAOs supported and implemented in that implementation.

The class diagram for this strategy is shown in Figure 9.4. This class diagram shows a base DAO factory, which is an abstract class that is inherited and implemented by different concrete DAO factories to support storage implementation-specific access. The client can obtain a concrete DAO factory implementation such as RdbDAOFactory and use it to obtain concrete DAOs that work with that specific storage implementation. For example, the data client can obtain an RdbDAOFactory and use it to get specific DAOs such as RdbCustomerDAO, RdbAccountDAO, and so forth. The DAOs can extend and implement a generic base class (shown as DAO1 and DAO2) that specifically describe the DAO requirements for the business object it supports. Each concrete DAO is responsible for connecting to the data source and obtaining and manipulating data for the business object it supports.

The sample implementation for the DAO pattern and its strategies is shown in the "Sample Code" section of this chapter.

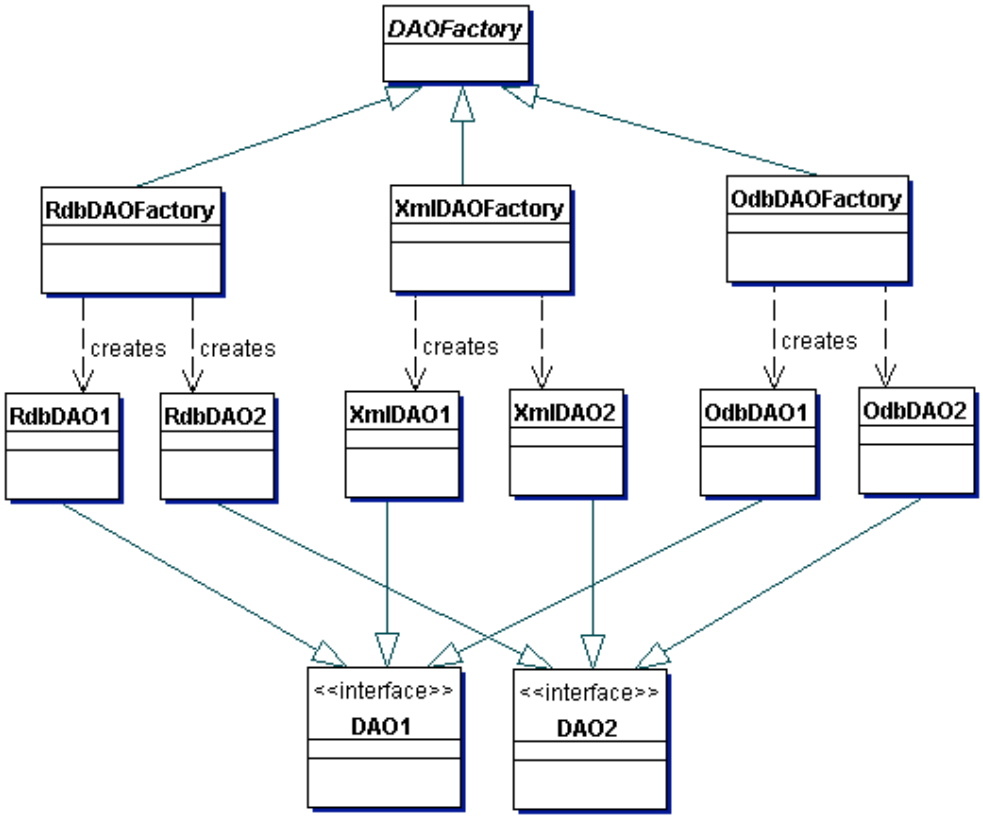

Figure 9.4 Factory for Data Access Object strategy using Abstract Factory

The sequence diagram describing the interactions for this strategy is shown in Figure 9.5.

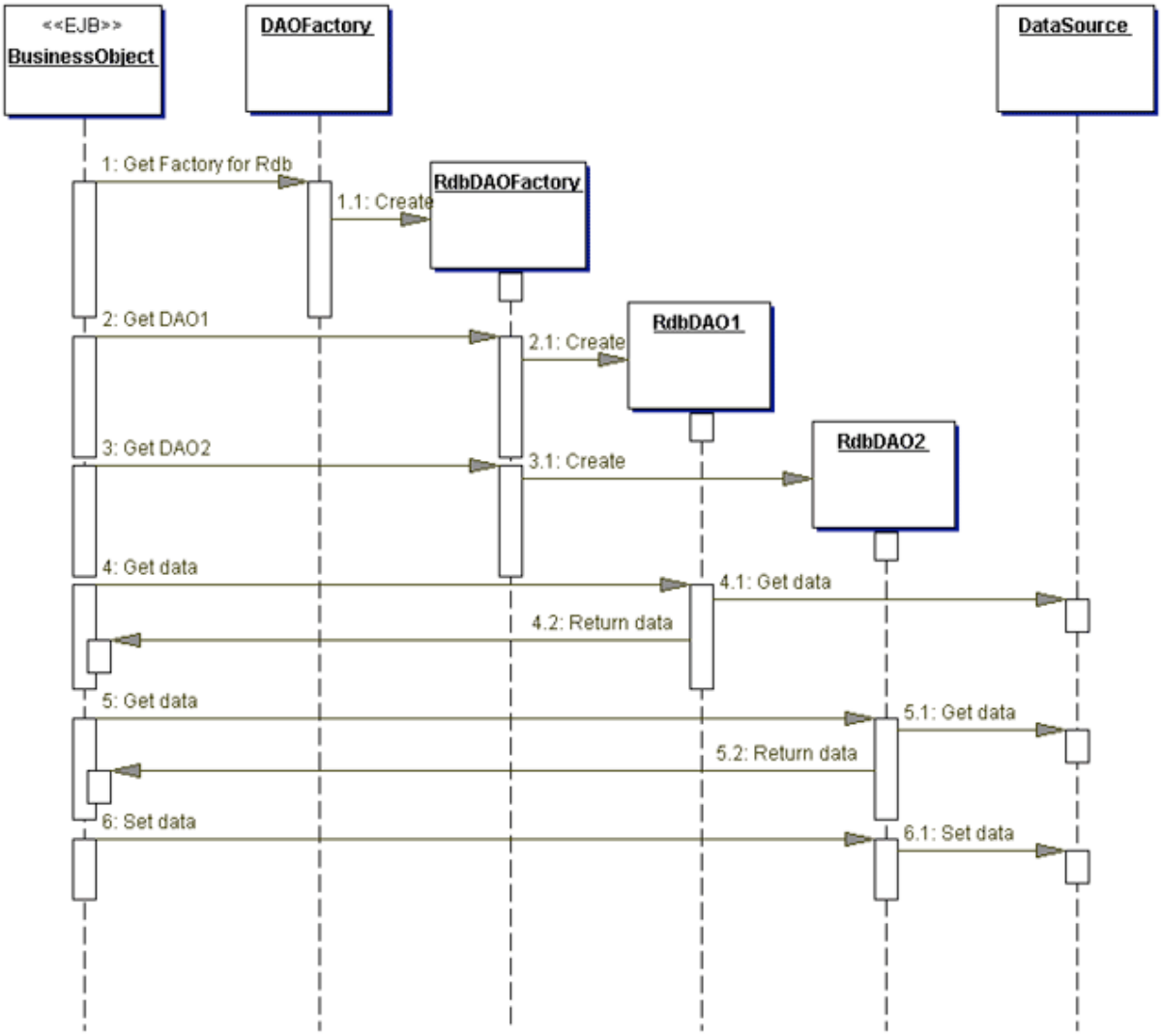

Figure 9.5 Factory for Data Access Objects using Abstract Factory sequence diagram

Consequences

- **Enables Transparency**
  Business objects can use the data source without knowing the specific details of the data source's implementation. Access is transparent because the implementation details are hidden inside the DAO.
- **Enables Easier Migration**
  A layer of DAOs makes it easier for an application to migrate to a different database implementation. The business objects have no knowledge of the underlying data implementation. Thus, the migration involves changes only to the DAO layer. Further, if employing a factory strategy, it is possible to provide a concrete factory implementation for each underlying storage implementation. In this case, migrating to a different storage implementation means providing a new factory implementation to the application.
- **Reduces Code Complexity in Business Objects** Because the DAOs manage all the data access complexities, it simplifies the code in the business objects and other data clients that use the DAOs. All implementation-related code (such as SQL statements) is contained in the DAO and not in the business object. This improves code readability and development productivity.
- **Centralizes All Data Access into a Separate Layer**
  Because all data access operations are now delegated to the DAOs, the separate data access layer can be viewed as the layer that can isolate the rest of the application from the data access implementation. This centralization makes the application easier to maintain and manage.
- **Not Useful for Container-Managed Persistence**
  Because the EJB container manages entity beans with container-managed persistence (CMP), the container automatically services all persistent storage access. Applications using container-managed entity beans do not need a DAO layer, since the application server transparently provides this functionality. However, DAOs are still useful when a combination of CMP (for entity beans) and BMP (for session beans, servlets) is required.
- **Adds Extra Layer**
  The DAOs create an additional layer of objects between the data client and the data source that need to be designed and implemented to leverage the benefits of this pattern. But the benefit realized by choosing this approach pays off for the additional effort.
- **Needs Class Hierarchy Design**
  When using a factory strategy, the hierarchy of concrete factories and the hierarchy of concrete products produced by the factories need to be designed and implemented. This additional effort needs to be considered if there is sufficient justification warranting such

flexibility. This increases the complexity of the design. However, you can choose to implement the factory strategy starting with the Factory Method pattern first, and then move towards the Abstract Factory if necessary.

Sample Code

Implementing Data Access Object pattern

An example DAO code for a persistent object that represents Customer information is shown in Example 9.4. The CloudscapeCustomerDAO creates a Customer Transfer Object when the `findCustomer()` method is invoked.

The sample code to use the DAO is shown in Example 9.6. The class diagram for this example is shown in Figure 9.6.

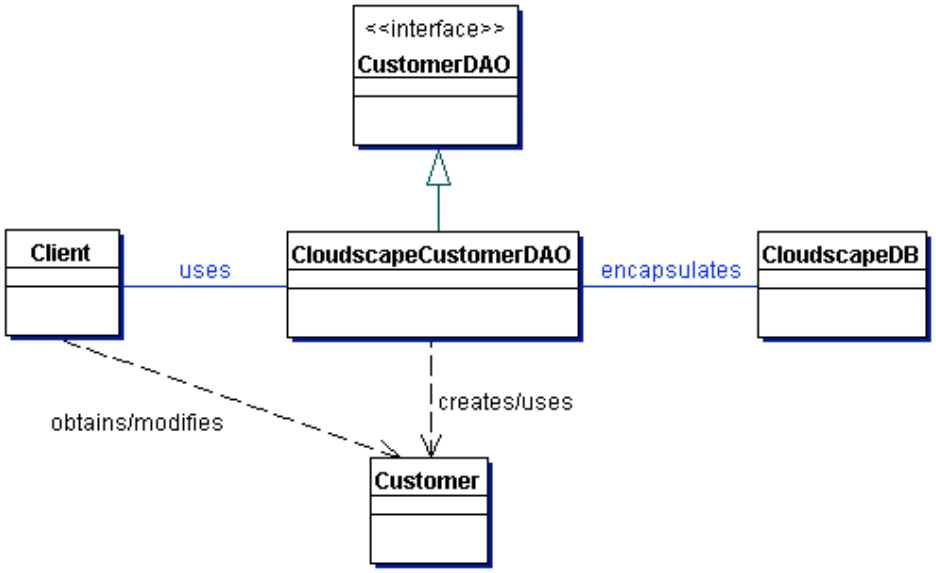

Figure 9.6 Implementing the DAO pattern

Implementing Factory for Data Access Objects Strategy

Using Factory Method Pattern

Consider an example where we are implementing this strategy in which a DAO factory produces many DAOs for a single database implementation (e.g., Oracle). The factory produces DAOs such as CustomerDAO, AccountDAO, OrderDAO, and so forth. The class diagram for this example is shown in Figure 9.7.

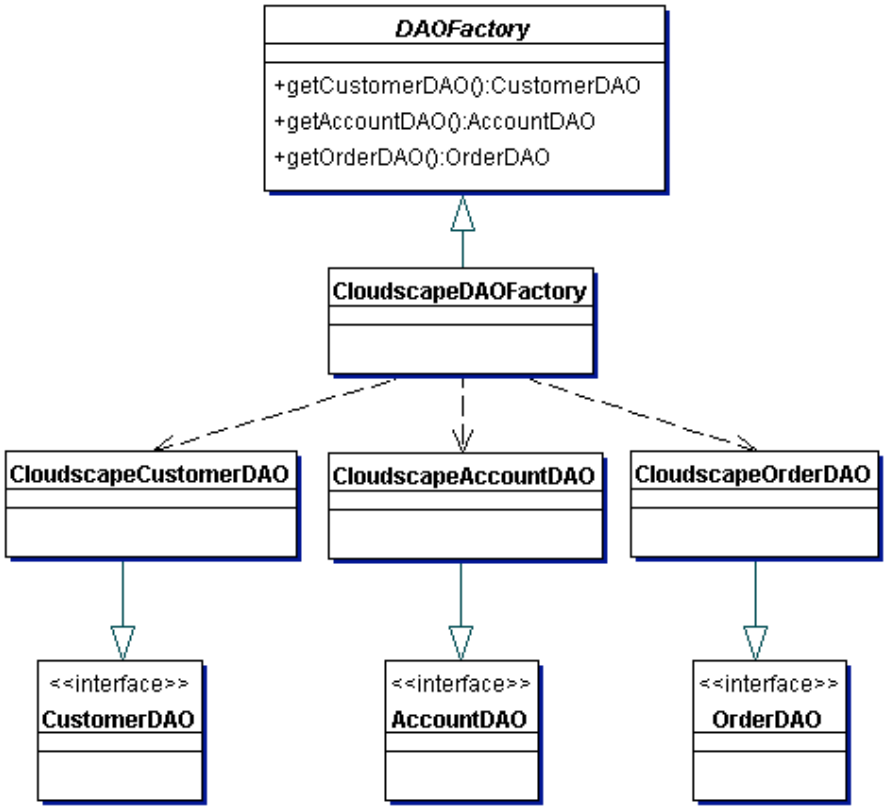

Figure 9.7 Implementing the Factory for DAO strategy using Factory Method

The example code for the DAO factory (CloudscapeDAOFactory) is listed in Example 9.2.

Using Abstract Factory Pattern

Consider an example where we are considering implementing this strategy for three different databases. In this case, the Abstract Factory pattern can be employed. The class diagram for this example is shown in Figure 9.8. The sample code in Example 9.1 shows code excerpt for the abstract DAOFactory class. This factory produces DAOs such as CustomerDAO, AccountDAO, OrderDAO, and so forth. This strategy uses the Factory Method implementation in the factories produced by the Abstract Factory.

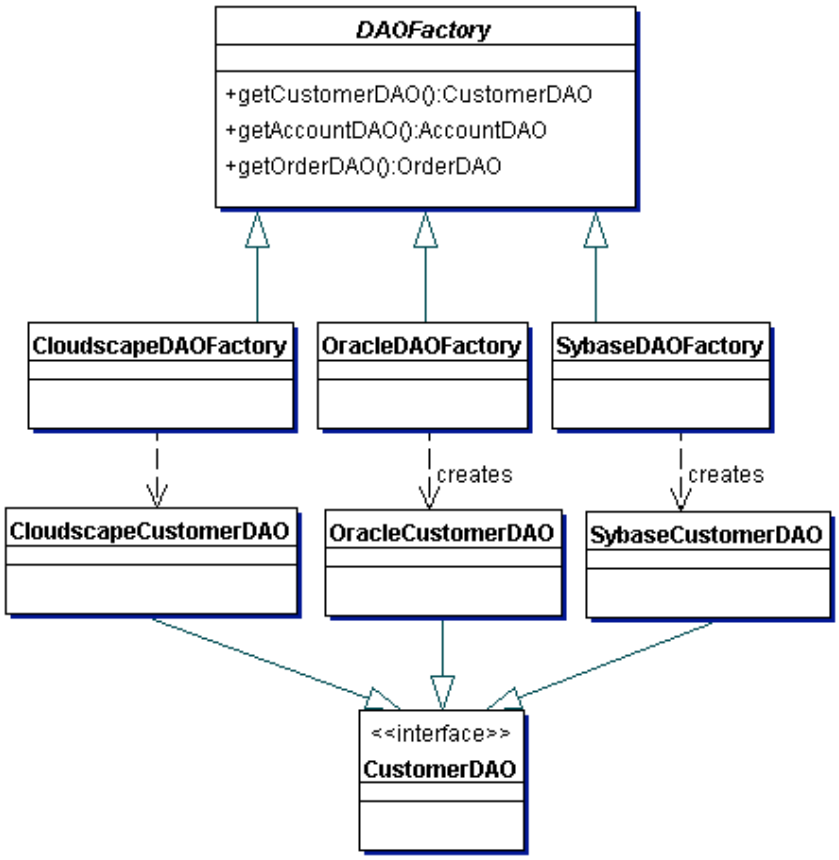

Figure 9.8 Implementing the Factory for DAO strategy using Abstract Factory Example 9.1 Abstract DAOFactory Class

```
// Abstract class DAO Factory
public abstract class DAOFactory {

  // List of DAO types supported by the factory
  public static final int CLOUDSCAPE = 1;
  public static final int ORACLE = 2;
  public static final int SYBASE = 3;
  ...

  // There will be a method for each DAO that can be
  // created. The concrete factories will have to
  // implement these methods.
  public abstract CustomerDAO getCustomerDAO();
  public abstract AccountDAO getAccountDAO();
  public abstract OrderDAO getOrderDAO();
  ...

  public static DAOFactory getDAOFactory(
      int whichFactory) {

    switch (whichFactory) {
      case CLOUDSCAPE:
          return new CloudscapeDAOFactory();
      case ORACLE    :
          return new OracleDAOFactory();
      case SYBASE    :
          return new SybaseDAOFactory();
      ...
      default           :
          return null;
    }
  }
}
```

The sample code for CloudscapeDAOFactory is shown in Example 9.2. The implementation for OracleDAOFactory and SybaseDAOFactory are similar except for specifics of each implementation, such as JDBC driver, database URL, and differences in SQL syntax, if any.

Example 9.2 Concrete DAOFactory Implementation for Cloudscape

```
// Cloudscape concrete DAO Factory implementation
import java.sql.*;

public class CloudscapeDAOFactory extends DAOFactory {
  public static final String DRIVER=
    "COM.cloudscape.core.RmiJdbcDriver";
  public static final String DBURL=
    "jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";

  // method to create Cloudscape connections
  public static Connection createConnection() {
    // Use DRIVER and DBURL to create a connection
    // Recommend connection pool implementation/usage
  }
  public CustomerDAO getCustomerDAO() {
    // CloudscapeCustomerDAO implements CustomerDAO
    return new CloudscapeCustomerDAO();
  }
  public AccountDAO getAccountDAO() {
    // CloudscapeAccountDAO implements AccountDAO
    return new CloudscapeAccountDAO();
  }
  public OrderDAO getOrderDAO() {
    // CloudscapeOrderDAO implements OrderDAO
    return new CloudscapeOrderDAO();
  }
  ...
}
```

The CustomerDAO interface shown in Example 9.3 defines the DAO methods for Customer persistent object that are implemented by all concrete DAO implementations, such as CloudscapeCustomerDAO, OracleCustomerDAO, and SybaseCustomerDAO. Similar, but not listed here, are AccountDAO and OrderDAO interfaces that define the DAO methods for Account and Order business objects respectively.

Example 9.3 Base DAO Interface for Customer

```
// Interface that all CustomerDAOs must support
public interface CustomerDAO {
  public int insertCustomer(...);
  public boolean deleteCustomer(...);
  public Customer findCustomer(...);
  public boolean updateCustomer(...);
  public RowSet selectCustomersRS(...);
  public Collection selectCustomersTO(...);
  ...
}
```

The CloudscapeCustomerDAO implements the CustomerDAO as shown in Example 9.4. The implementation of other DAOs, such as CloudscapeAccountDAO, CloudscapeOrderDAO, OracleCustomerDAO, OracleAccountDAO, and so forth, are similar.

Example 9.4 Cloudscape DAO Implementation for Customer

```
// CloudscapeCustomerDAO implementation of the
// CustomerDAO interface. This class can contain all
// Cloudscape specific code and SQL statements.
// The client is thus shielded from knowing
```

```
  // these implementation details.

  import java.sql.*;

  public class CloudscapeCustomerDAO implements
      CustomerDAO {

    public CloudscapeCustomerDAO() {
      // initialization
    }

    // The following methods can use
    // CloudscapeDAOFactory.createConnection()
    // to get a connection as required

    public int insertCustomer(...) {
      // Implement insert customer here.
      // Return newly created customer number
      // or a -1 on error
    }

    public boolean deleteCustomer(...) {
      // Implement delete customer here
      // Return true on success, false on failure
    }

    public Customer findCustomer(...) {
      // Implement find a customer here using supplied
      // argument values as search criteria
      // Return a Transfer Object if found,
      // return null on error or if not found
    }

    public boolean updateCustomer(...) {
      // implement update record here using data
      // from the customerData Transfer Object
      // Return true on success, false on failure or
      // error
    }

    public RowSet selectCustomersRS(...) {
      // implement search customers here using the
      // supplied criteria.
      // Return a RowSet.
    }

    public Collection selectCustomersTO(...) {
      // implement search customers here using the
      // supplied criteria.
      // Alternatively, implement to return a Collection
      // of Transfer Objects.
    }
    ...
  }
```

The Customer Transfer Object class is shown in Example 9.5. This is used by the DAOs to send and receive data from the clients. The usage of Transfer Objects is discussed in detail in the Transfer Object pattern.

Example 9.5 Customer Transfer Object

```
  public class Customer implements java.io.Serializable {
    // member variables
    int CustomerNumber;
    String name;
    String streetAddress;
    String city;
```

```
    ...

    // getter and setter methods...
    ...
  }
```

Example 9.6 shows the usage of the DAO factory and the DAO. If the implementation changes from Cloudscape to another product, the only required change is the `getDAOFactory()` method call to the DAO factory to obtain a different factory.

Example 9.6 Using a DAO and DAO Factory - Client Code

```
...
// create the required DAO Factory
DAOFactory cloudscapeFactory =
  DAOFactory.getDAOFactory(DAOFactory.DAOCLOUDSCAPE);

// Create a DAO
CustomerDAO custDAO =
  cloudscapeFactory.getCustomerDAO();

// create a new customer
int newCustNo = custDAO.insertCustomer(...);

// Find a customer object. Get the Transfer Object.
Customer cust = custDAO.findCustomer(...);

// modify the values in the Transfer Object.
cust.setAddress(...);
cust.setEmail(...);
// update the customer object using the DAO
custDAO.updateCustomer(cust);

// delete a customer object
custDAO.deleteCustomer(...);
// select all customers in the same city
Customer criteria=new Customer();
criteria.setCity("New York");
Collection customersList =
  custDAO.selectCustomersTO(criteria);
// returns customersList - collection of Customer
// Transfer Objects. iterate through this collection to
// get values.

...
```

Related Patterns

- **Transfer Object**
  A DAO uses Transfer Objects to transport data to and from its clients.
- **Factory Method [GoF] and Abstract Factory [GoF]**
  The *Factory for Data Access Objects Strategy* uses the Factory Method pattern to implement the concrete factories and its products (DAOs). For added flexibility, the Abstract Factory pattern may be employed as discussed in the strategies.
- **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 decouple the resource tier from clients in another tier, such as the business or presentation tier