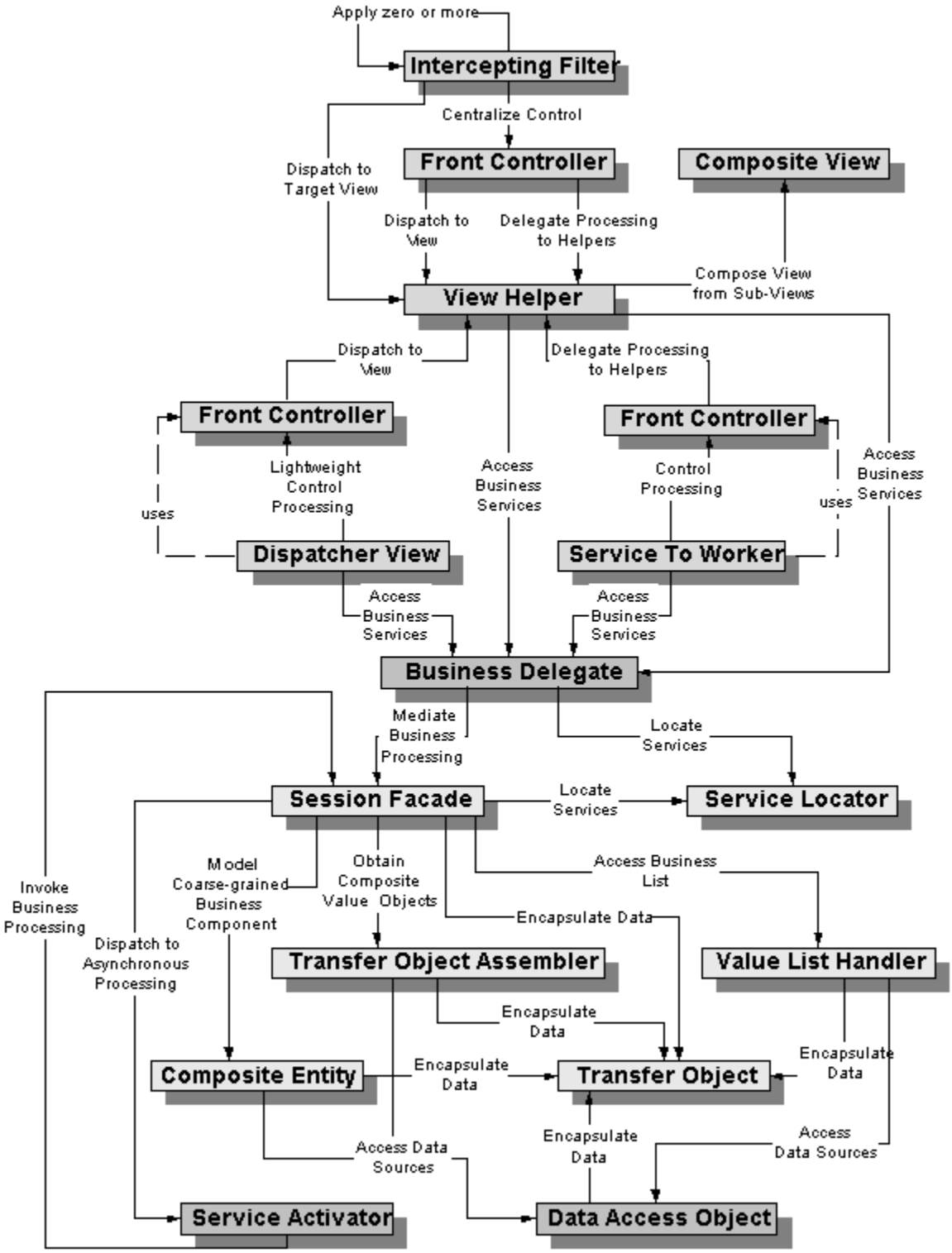
Design Patterns – Material



65, Velachery Road, Little Mount, Ch- 600 015. Ph: 44 32971945 email: info@haarisinfotech.com

Abstract Factory

Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem

If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulatation is not engineered in advance, and lots of #ifdef case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

Discussion

Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated. The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.

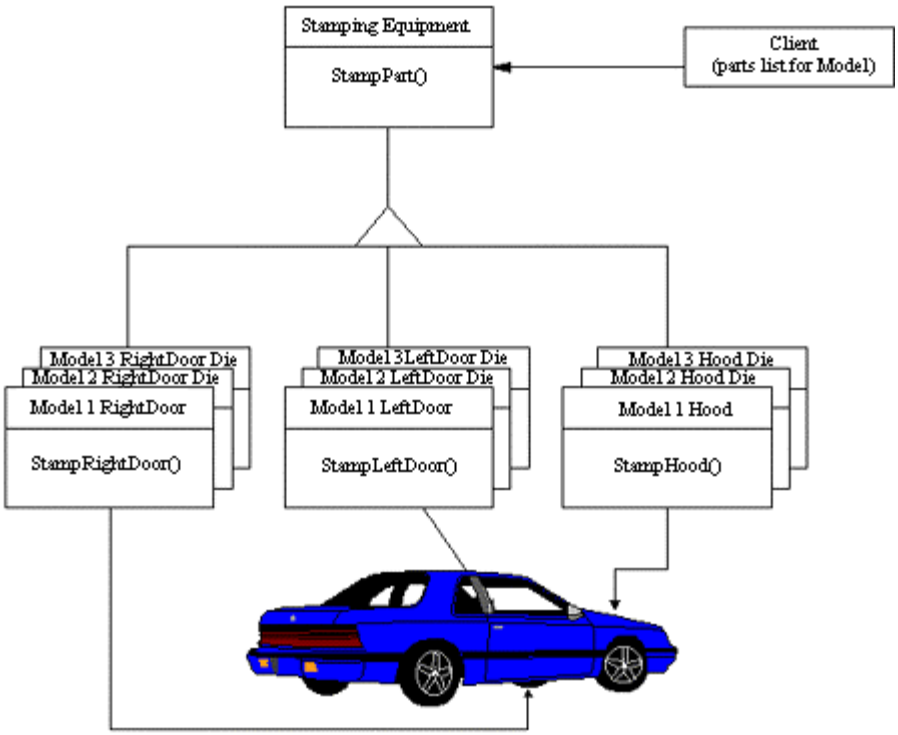
Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

Structure

This diagram is from javacoder.net.

Example

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



Rules of thumb

Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126], Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, pp81,134]

Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object. [GOF, p135]

Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype. [GOF, p95]

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GOF, p105]

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Example

```
// Abstract Factory classes are often implemented with Factory Methods,
but they
// can also be implemented using Prototype. [GOF, p95]   Abstract
Factory might
// store a set of Prototypes from which to clone and return product
objects.
// [GOF, p126]  Factory Method: creation through inheritance. Protoype:
creation
// through delegation.  Virtual constructor: defer choice of object to
create
// until run-time.

public class FactoryFmProto {

static class Expression {
   protected String str;
   public Expression( String s ) { str = s; }
   public Expression cloan()     { return null; }
   public String     toString()  { return str; }
}

static abstract class Factory {
   protected Expression prototype = null;
   public Expression makePhrase() { return prototype.cloan(); }
   public abstract Expression makeCompromise();
   public abstract Expression makeGrade();
}

static class PCFactory extends Factory {
   public PCFactory() { prototype = new PCPhrase(); }
   public Expression makeCompromise() {
      return new Expression( "\"do it your way, any way, or no way\""
); }
   public Expression makeGrade() {
      return new Expression( "\"you pass, self-esteem intact\"" ); }
}

static class NotPCFactory extends Factory {
   public NotPCFactory() { prototype = new NotPCPhrase(); }
   public Expression makeCompromise() {
      return new Expression( "\"my way, or the highway\"" ); }
   public Expression makeGrade() {
      return new Expression( "\"take test, deal with the results\"" );
}
}

public static void main( String[] args ) {
   Factory factory;
   if (args.length > 0) factory = new PCFactory();
```

```
   else                 factory = new NotPCFactory();
   for (int i=0; i < 3; i++) System.out.print( factory.makePhrase() + "
" );
   System.out.println();
   System.out.println( factory.makeCompromise() );
   System.out.println( factory.makeGrade() );
}

// D:\Java\patterns> java FactoryFmProto
// "short"  "lie"  "old"
// "my way, or the highway"
// "take test, deal with the results"
//
// D:\Java\patterns> java FactoryFmProto 1
// "vertically challenged"  "factually inaccurate"  "chronologically
gifted"
// "do it your way, any way, or no way"
// "you pass, self-esteem intact"

static class PCPhrase extends Expression {
   static String[] list = { "\"animal companion\"", "\"vertically
challenged\"",
                    "\"factually inaccurate\"", "\"chronologically
gifted\"" };
   private static int next = 0;
   public PCPhrase()         { super( list[next] );  next = (next+1) %
list.length; }
   public Expression cloan() { return new PCPhrase(); }
}

static class NotPCPhrase extends Expression {
   private static String[] list = { "\"pet\"", "\"short\"", "\"lie\"",
"\"old\"" };
   private static int next = 0;
   public NotPCPhrase()      { super( list[next] );  next = (next+1) %
list.length; }
   public Expression cloan() { return new NotPCPhrase(); }
}}
```

Core J2EE Pattern Catalog

Core J2EE Patterns - Data Access Object

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.

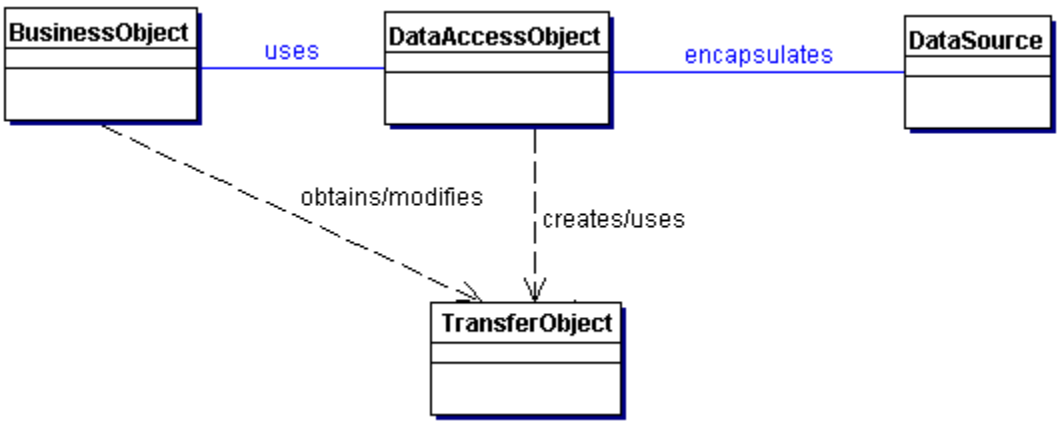


Figure 9.1 Data Access Object

Participants and Responsibilities

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

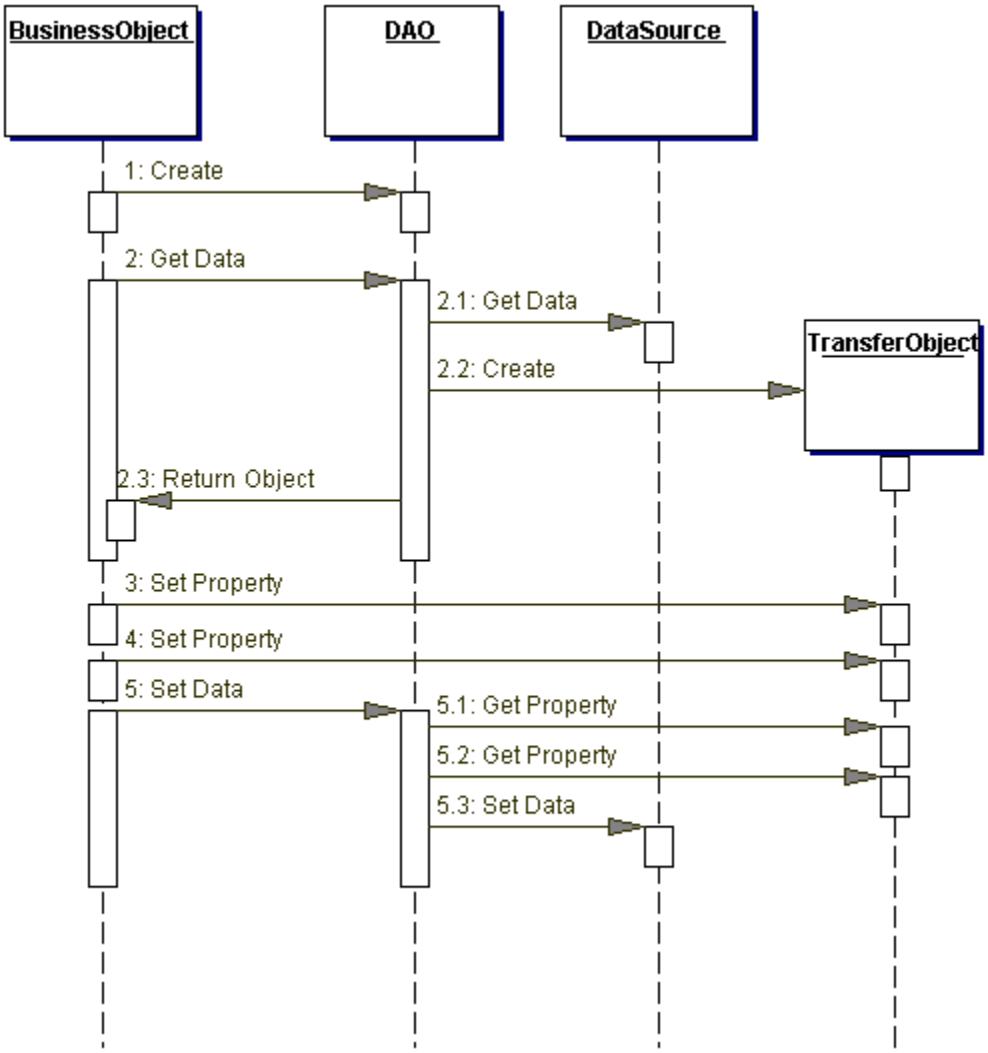


Figure 9.2 Data Access Object sequence diagram

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.

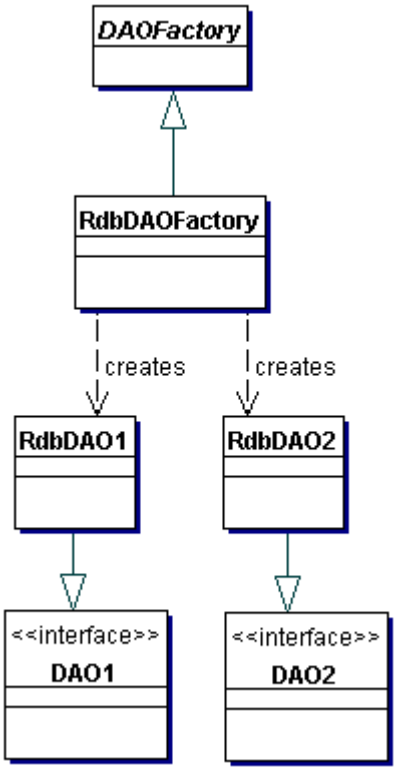


Figure 9.3 Factory for Data Access Object strategy using Factory Method

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.

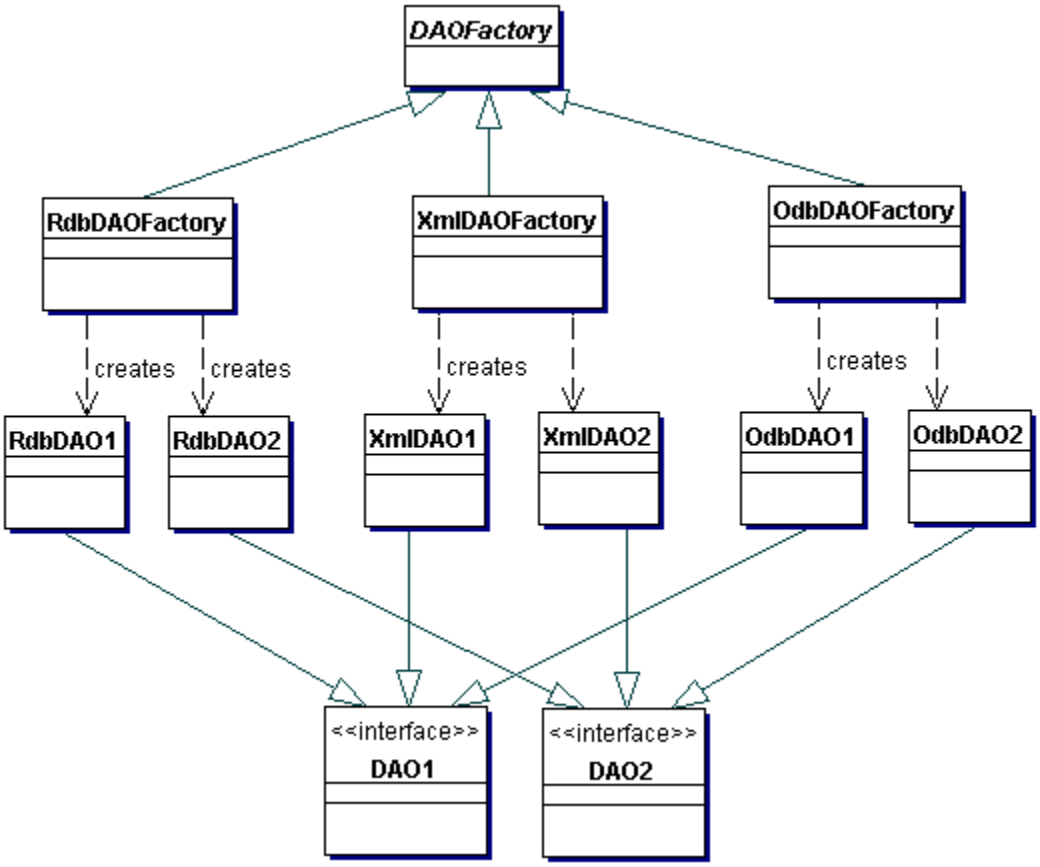


Figure 9.4 Factory for Data Access Object strategy using Abstract Factory

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

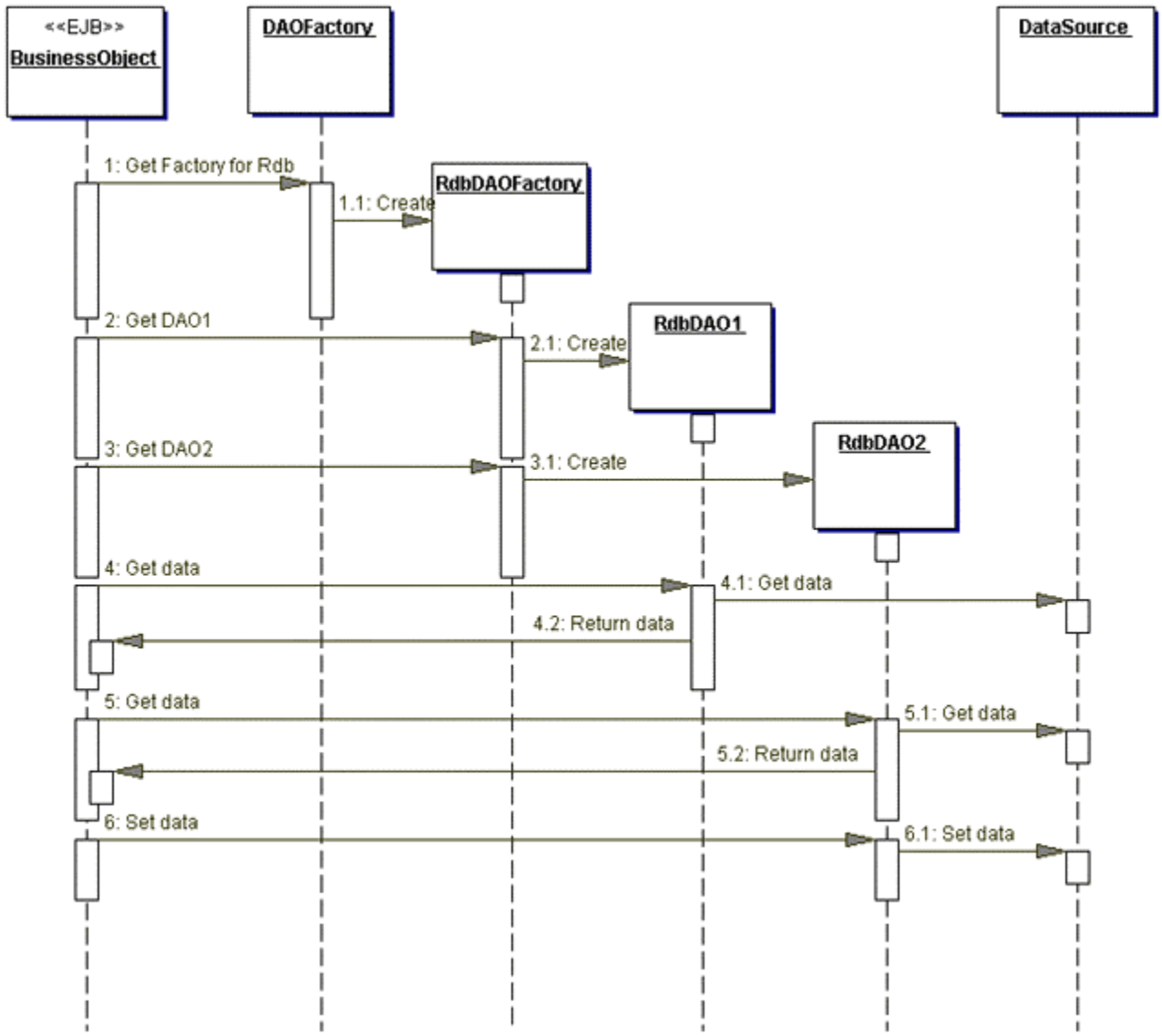


Figure 9.5 Factory for Data Access Objects using Abstract Factory sequence diagram

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.

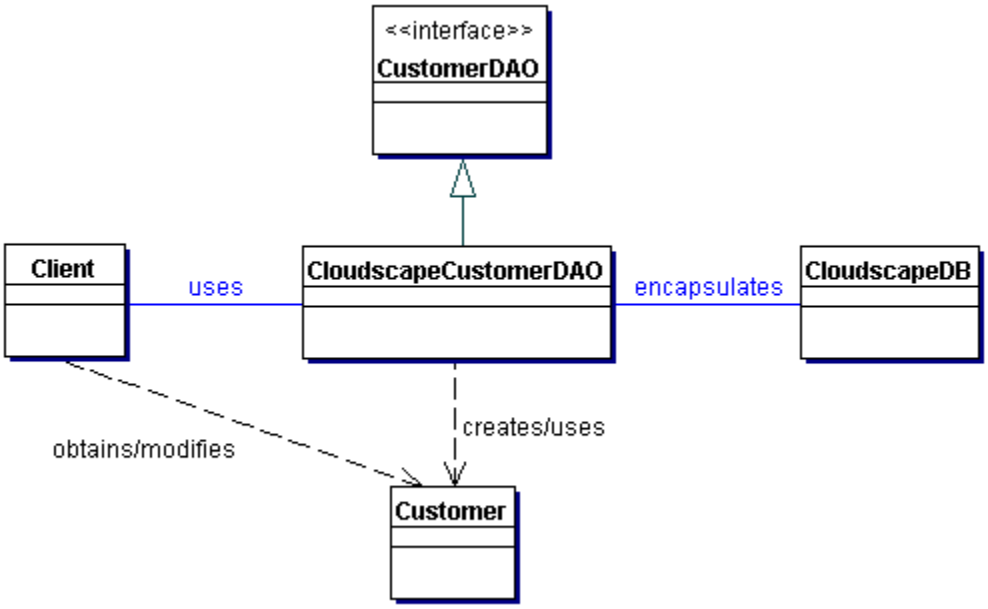


Figure 9.6 Implementing the DAO pattern

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.

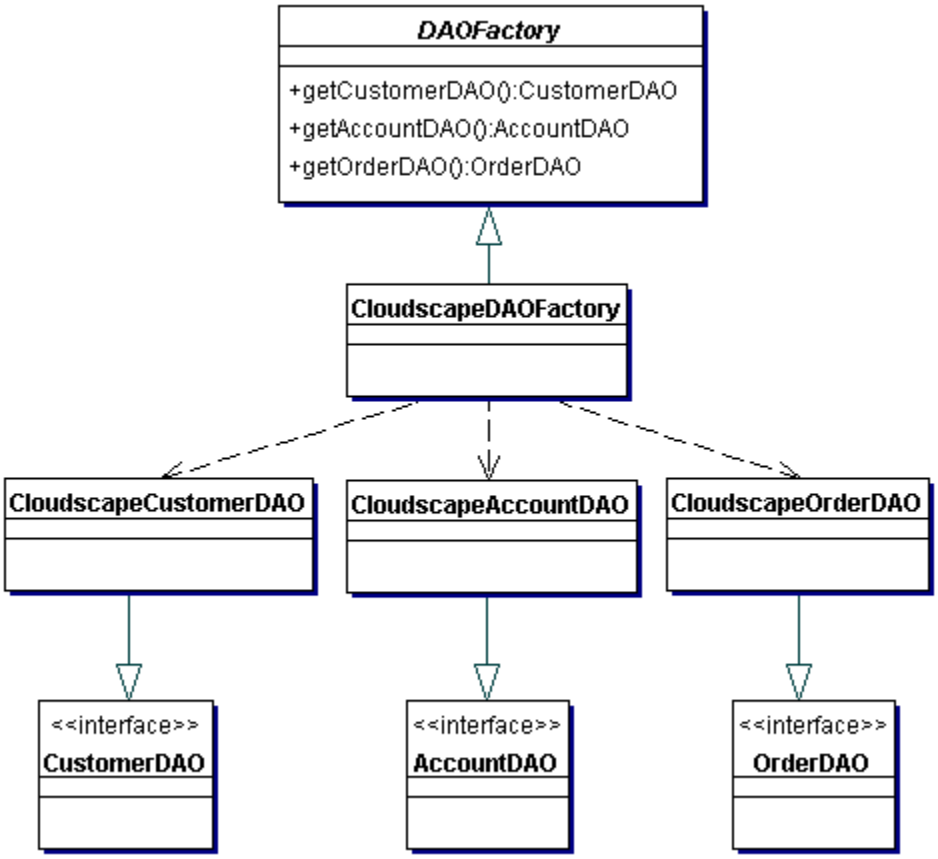


Figure 9.7 Implementing the Factory for DAO strategy using Factory Method

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.

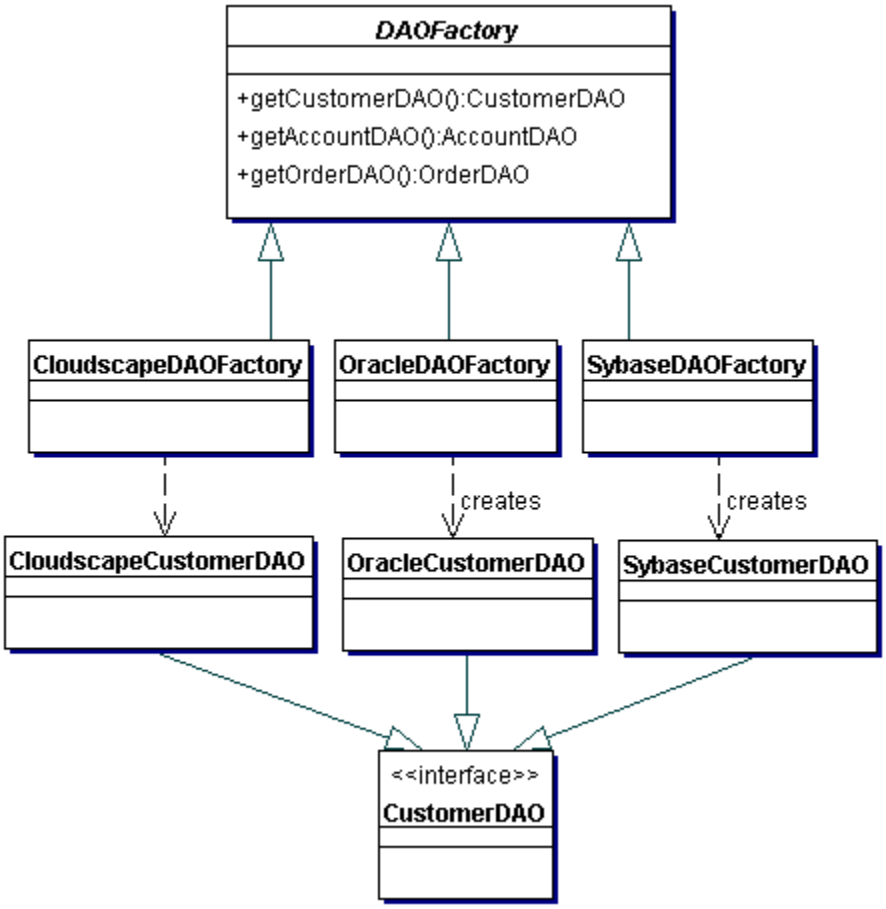


Figure 9.8 Implementing the Factory for DAO strategy using Abstract Factory

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

Core J2EE Pattern Catalog
Core J2EE Patterns - Transfer Object

Context

Application clients need to exchange data with enterprise beans.

Problem

Java 2 Platform, Enterprise Edition (J2EE) applications implement server-side business components as session beans and entity beans. Some methods exposed by the business

components return data to the client. Often, the client invokes a business object's get methods multiple times until it obtains all the attribute values.

Session beans represent the business services and are not shared between users. A session bean provides coarse-grained service methods when implemented per the Session Facade pattern.

Entity beans, on the other hand, are multiuser, transactional objects representing persistent data. An entity bean exposes the values of attributes by providing an accessor method (also referred to as a *getter* or *get method*) for each attribute it wishes to expose.

Every method call made to the business service object, be it an entity bean or a session bean, is potentially remote. Thus, in an Enterprise JavaBeans (EJB) application such remote invocations use the network layer regardless of the proximity of the client to the bean, creating a network overhead. Enterprise bean method calls may permeate the network layers of the system even if the client and the EJB container holding the entity bean are both running in the same JVM, OS, or physical machine. Some vendors may implement mechanisms to reduce this overhead by using a more direct access approach and bypassing the network.

As the usage of these remote methods increases, application performance can significantly degrade. Therefore, using multiple calls to get methods that return single attribute values is inefficient for obtaining data values from an enterprise bean.

Forces

- All access to an enterprise bean is performed via remote interfaces to the bean. Every call to an enterprise bean is potentially a remote method call with network overhead.
- Typically, applications have a greater frequency of read transactions than update transactions. The client requires the data from the business tier for presentation, display, and other read-only types of processing. The client updates the data in the business tier much less frequently than it reads the data.
- The client usually requires values for more than one attribute or dependent object from an enterprise bean. Thus, the client may invoke multiple remote calls to obtain the required data.
- The number of calls made by the client to the enterprise bean impacts network performance. Chattier applications-those with increased traffic between client and server tiers-often degrade network performance.

Solution

Use a Transfer Object to encapsulate the business data. A single method call is used to send and retrieve the Transfer Object. When the client requests the enterprise bean for the business data, the enterprise bean can construct the Transfer Object, populate it with its attribute values, and pass it by value to the client.

Clients usually require more than one value from an enterprise bean. To reduce the number of remote calls and to avoid the associated overhead, it is best to use Transfer Objects to transport the data from the enterprise bean to its client.

When an enterprise bean uses a Transfer Object, the client makes a single remote method invocation to the enterprise bean to request the Transfer Object instead of numerous remote method calls to get individual attribute values. The enterprise bean then constructs a new Transfer Object instance, copies values into the object and returns it to the client. The client receives the Transfer Object and can then invoke accessor (or getter) methods

on the Transfer Object to get the individual attribute values from the Transfer Object. Or, the implementation of the Transfer Object may be such that it makes all attributes public. Because the Transfer Object is passed by value to the client, all calls to the Transfer Object instance are local calls instead of remote method invocations.

Structure

Figure 8.5 shows the class diagram that represents the Transfer Object pattern in its simplest form.

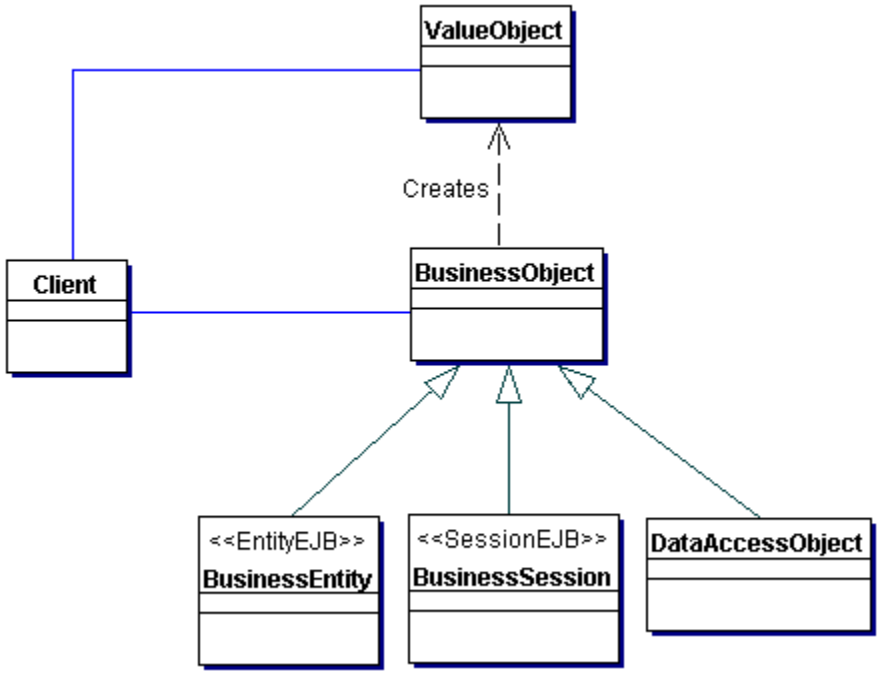


Figure 8.5 Transfer Object class diagram

As shown in this class diagram, the Transfer Object is constructed on demand by the enterprise bean and returned to the remote client. However, the Transfer Object pattern can adopt various strategies, depending on requirements. The "Strategies" section explains these approaches.

Participants and Responsibilities

Figure 8.6 contains the sequence diagram that shows the interactions for the Transfer Object pattern.

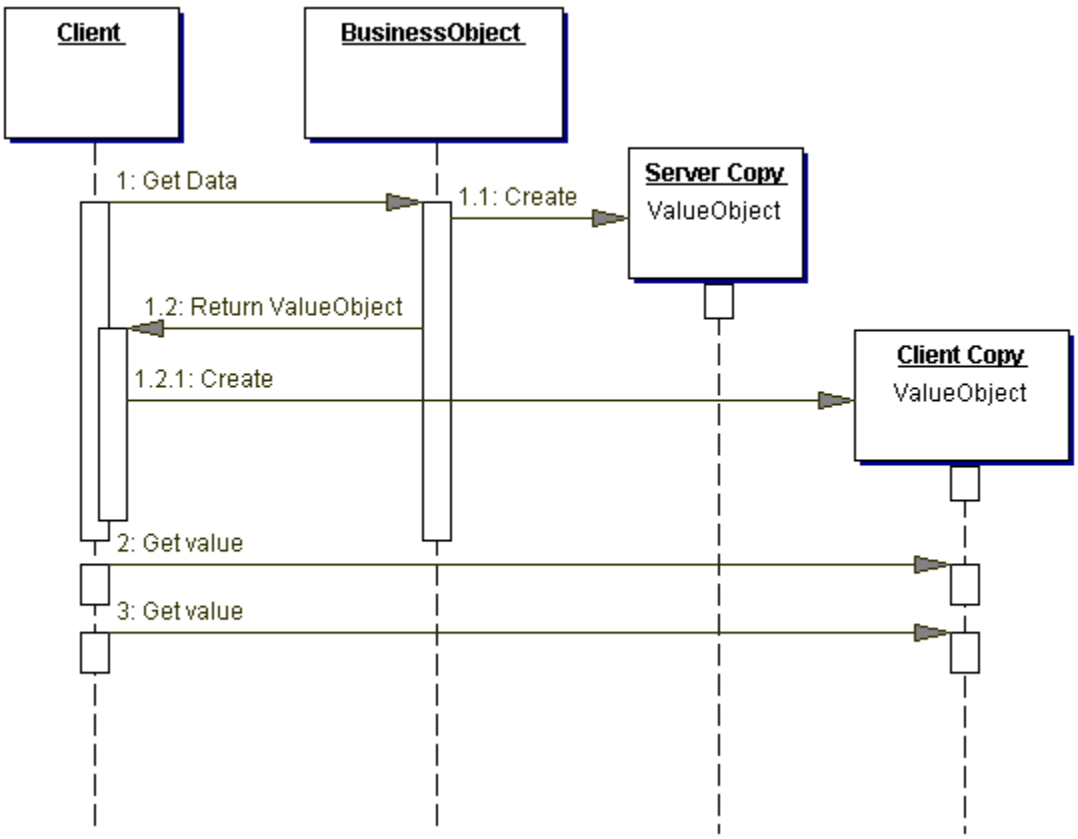


Figure 8.6 Transfer Object sequence diagram

Client

This represents the client of the enterprise bean. The client can be an end-user application, as in the case of a rich client application that has been designed to directly access the enterprise beans. The client can be Business Delegates (see "Business Delegate" on page 248) or a different BusinessObject.

BusinessObject

The BusinessObject represents a role in this pattern that can be fulfilled by a session bean, an entity bean, or a Data Access Object (DAO). The BusinessObject is responsible for creating the Transfer Object and returning it to the client upon request. The BusinessObject may also receive data from the client in the form of a Transfer Object and use that data to perform an update.

TransferObject

The TransferObject is an arbitrary serializable Java object referred to as a Transfer Object. A Transfer Object class may provide a constructor that accepts all the required attributes to create the Transfer Object. The constructor may accept all entity bean attribute values that the Transfer Object is designed to hold. Typically, the members in the Transfer Object are defined as public, thus eliminating the need for get and set methods. If some protection is necessary, then the members could be defined as protected or private, and methods are provided to get the values. By offering no methods to set the values, a Transfer Object is protected from modification after its creation. If only a few

members are allowed to be modified to facilitate updates, then methods to set the values can be provided. Thus, the Transfer Object creation varies depending on an application's requirements. It is a design choice as to whether the Transfer Object's attributes are private and accessed via getters and setters, or all the attributes are made public.

Strategies

The first two strategies discussed are applicable when the enterprise bean is implemented as a session bean or as an entity bean. These strategies are called *Updatable Transfer Objects Strategy* and *Multiple Transfer Objects Strategy*.

The following strategies are applicable only when the BusinessObject is implemented as an entity bean: *Entity Inherits Transfer Object Strategy* and *Transfer Object Factory Strategy*.

Figure 8.7 Updatable Transfer Object strategy - class diagram

The BusinessObject creates the Transfer Object. Recall that a client may need to access the BusinessObject values not only to read them but to modify these values. For the client to be able to modify the BusinessObject attribute values, the BusinessObject must provide mutator methods. Mutator methods are also referred to as *setters* or *set methods*.

Instead of providing fine-grained set methods for each attribute, which results in network overhead, the BusinessObject can expose a coarse-grained `setData()` method that accepts a Transfer Object as an argument. The Transfer Object passed to this method holds the updated values from the client. Since the Transfer Object has to be mutable, the Transfer Object class has to provide set methods for each attribute that can be modified by the client. The set methods for the Transfer Object can include field level validations and integrity checks as needed. Once the client obtains a Transfer Object from the BusinessObject, the client invokes the necessary set methods locally to change the attribute values. Such local changes do not impact the BusinessObject until the `setData()` method is invoked.

The `setData()` method serializes the client's copy of the Transfer Object and sends it to the BusinessObject. The BusinessObject receives the modified Transfer Object from the client and merges the changes into its own attributes. The merging operation may complicate the design of the BusinessObject and the Transfer Object; the "Consequences" section discusses these potential complications. One strategy to use here is to update only attributes that have changed, rather than updating all attributes. A change flag placed in the Transfer Object can be used to determine the attributes to update, rather than doing a direct comparison.

There is an impact on the design using the updatable Transfer Objects in terms of update propagation, synchronization, and version control.

Figure 8.8 shows the sequence diagram for the entire update interaction.

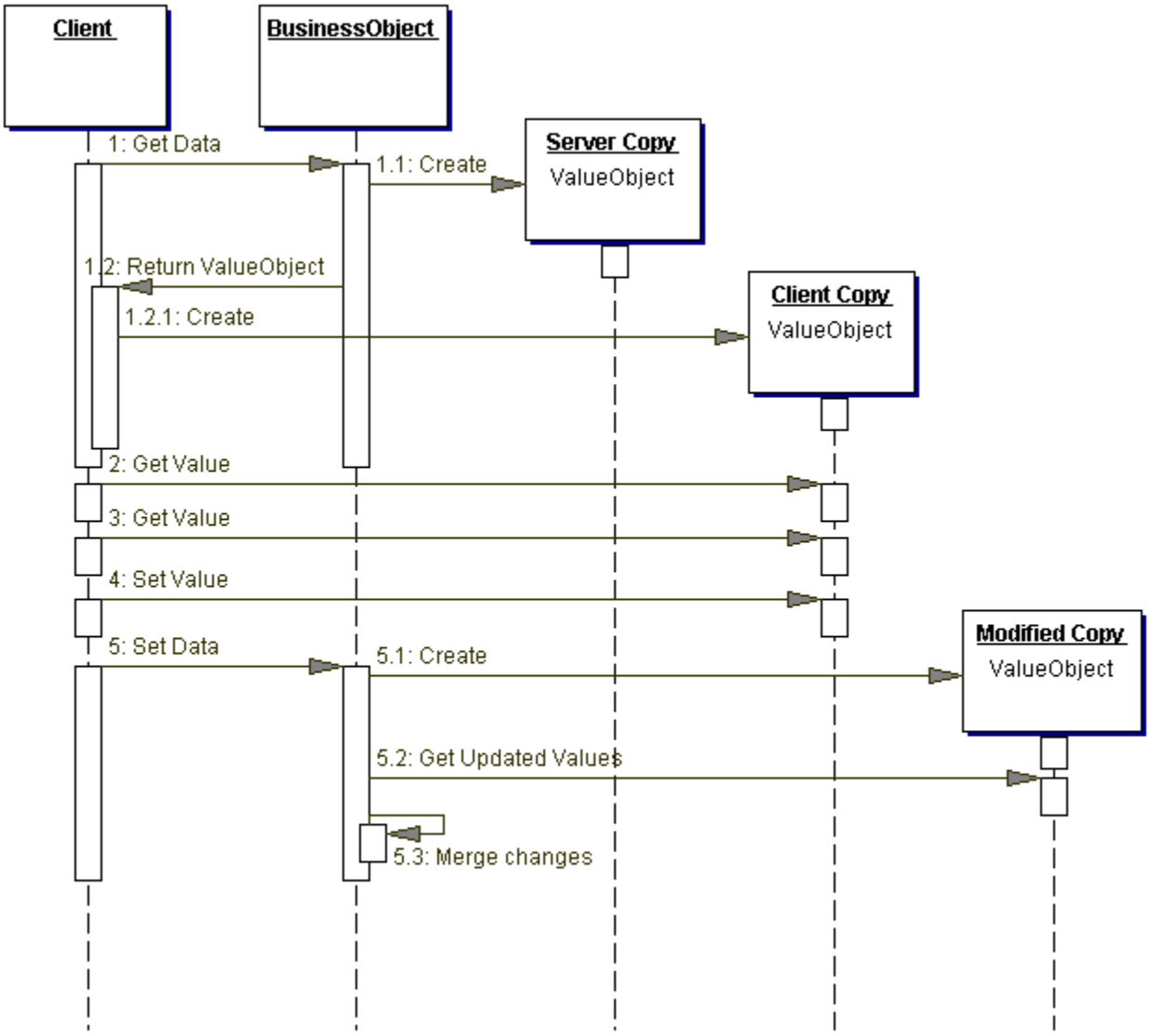


Figure 8.8 Updatable Transfer Object strategy - sequence diagram

Multiple Transfer Objects Strategy

Some application business objects can be very complex. In such cases, it is possible that a single business object produces different Transfer Objects, depending on the client request. There exists a one-to-many relationship between the business object and the many Transfer Objects it can produce. In these circumstances, this strategy may be considered.

For instance, when the business object is implemented as a session bean, typically applying the Session Facade pattern, the bean may interact with numerous other business components to provide the service. The session bean produces its Transfer Object from different sources. Similarly, when the BusinessObject is implemented as a coarse-grained entity bean, typically applying the Composite Entity pattern, the entity bean will have complex relationships with a number of dependent objects. In both these cases, it is good

practice to provide mechanisms to produce Transfer Objects that actually represent parts of the underlying coarse-grained components.

For example, in a trading application, a Composite Entity that represents a customer portfolio can be a very coarse-grained complex component that can produce Transfer Objects that provide data for parts of the portfolio, like customer information, lists of stocks held, and so on. A similar example is a customer manager session bean that provides services by interacting with a number of other BusinessObjects and components to provide its service. The customer manager bean can produce discrete small Transfer Objects, like customer address, contact list, and so on, to represent parts of its model.

For both these scenarios, it is possible to adopt and apply the *Multiple Transfer Objects Strategy* so that the business component, whether a session bean or an entity bean, can create multiple types of Transfer Objects. In this strategy, the business entity provides various methods to get different Transfer Objects. Each such method creates and returns a different type of Transfer Object. The class diagram for this strategy is shown Figure 8.9.

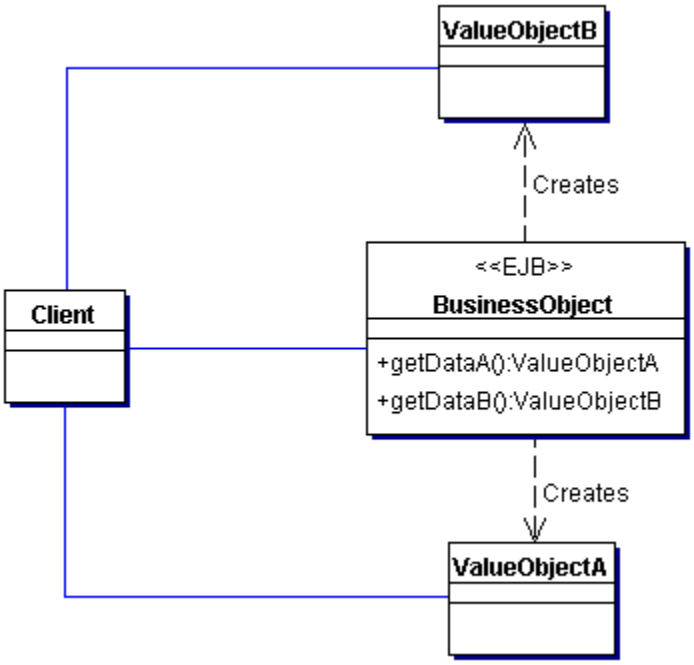


Figure 8.9 Multiple Transfer Objects strategy class diagram

When a client needs a Transfer Object of type TransferObjectA, it invokes the entity's getDataA() method requesting TransferObjectA. When it needs a Transfer Object of type TransferObjectB, it invokes the entity's getDataB() method requesting TransferObjectB, and so on. This is shown in the sequence diagram in Figure 8.10.

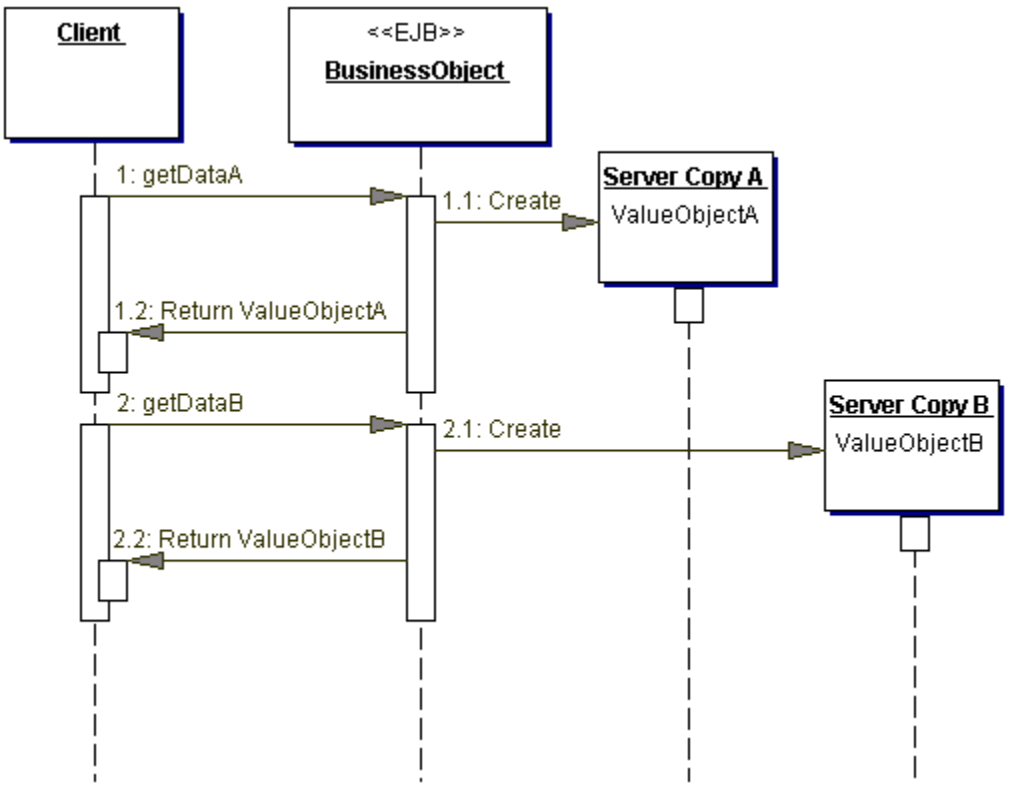


Figure 8.10 Multiple Transfer Objects strategy sequence diagram

Entity Inherits Transfer Object Strategy

When the BusinessObject is implemented as an entity bean and the clients typically need to access all the data from the entity bean, then the entity bean and the Transfer Object both have the same attributes. In this case, since there exists a one-to-one relationship between the entity bean and its Transfer Object, the entity bean may be able to use inheritance to avoid code duplication.

In this strategy, the entity bean extends (or inherits from) the Transfer Object class. The only assumption is that the entity bean and the Transfer Object share the same attribute definitions. The class diagram for this strategy is shown in Figure 8.11.

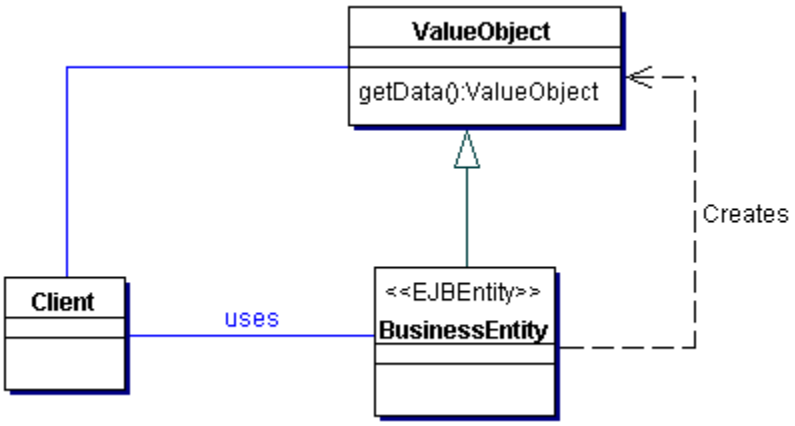


Figure 8.11 Entity Inherits Transfer Object strategy class diagram

The TransferObject implements one or more `getData()` methods as discussed in the *Multiple Transfer Objects Strategy*. When the entity inherits this Transfer Object class, the client invokes an inherited `getData()` method on the entity bean to obtain a Transfer Object.

Thus, this strategy eliminates code duplication between the entity and the Transfer Object. It also helps manage changes to the Transfer Object requirements by isolating the change to the Transfer Object class and preventing the changes from affecting the entity bean.

This strategy has a trade-off related to inheritance. If the Transfer Object is shared through inheritance, then changes to this Transfer Object class will affect all its subclasses, potentially mandating other changes to the hierarchy.

The sequence diagram in Figure 8.12 demonstrates this strategy.

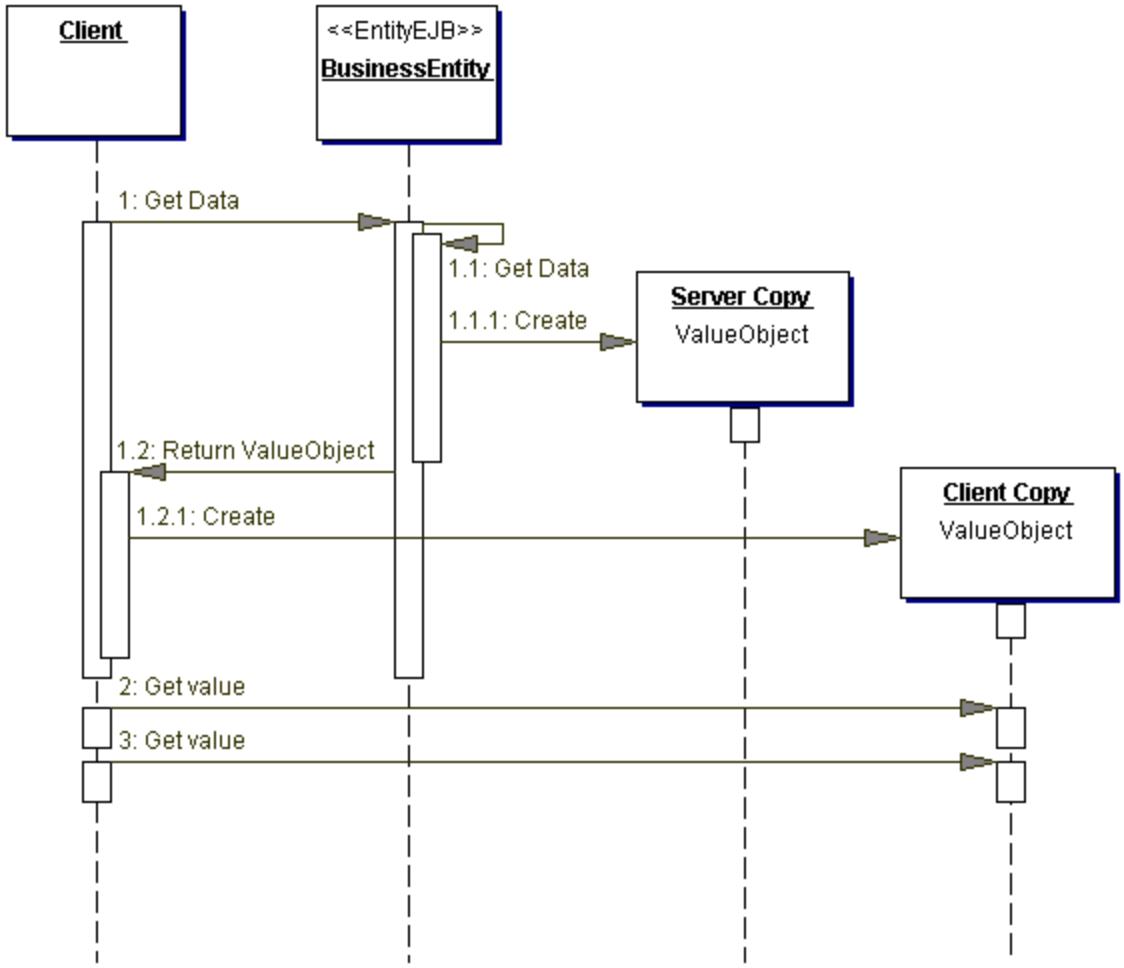


Figure 8.12 Entity Inherits Transfer Object strategy sequence diagram

The sample implementation for the Entity Inherits Transfer Object Strategy is shown in Example 8.10 (ContactTO - Transfer Object Class) and Example 8.11 (ContactEntity - Entity Bean Class).

Transfer Object Factory Strategy

The Entity Inherits Transfer Object Strategy can be further extended to support multiple Transfer Objects for an entity bean by employing a Transfer Object factory to create Transfer Objects on demand using reflection. This results in an even more dynamic strategy for Transfer Object creation.

To achieve this, define a different interface for each type of Transfer Object that must be returned. The entity bean implementation of Transfer Object superclass must implement all these interfaces. Furthermore, you must create a separate implementation class for each defined interface, as shown in the class diagram for this strategy in Figure 8.13.

Once all interfaces have been defined and implemented, create a method in the TransferObjectFactory that is passed two arguments:

- The entity bean instance for which a Transfer Object must be created.
- The interface that identifies the kind of Transfer Object to create.

The TransferObjectFactory can then instantiate an object of the correct class, set its values, and return the newly created Transfer Object instance.

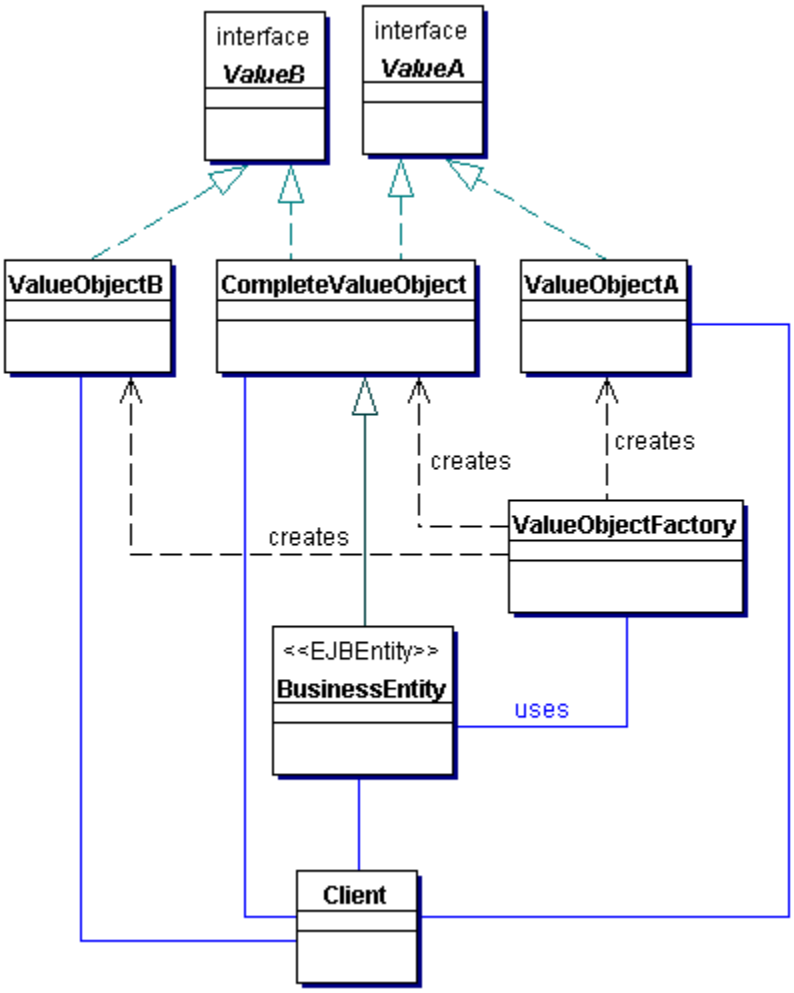


Figure 8.13 Transfer Object Factory strategy class diagram

The sequence diagram for this strategy is shown in Figure 8.14.

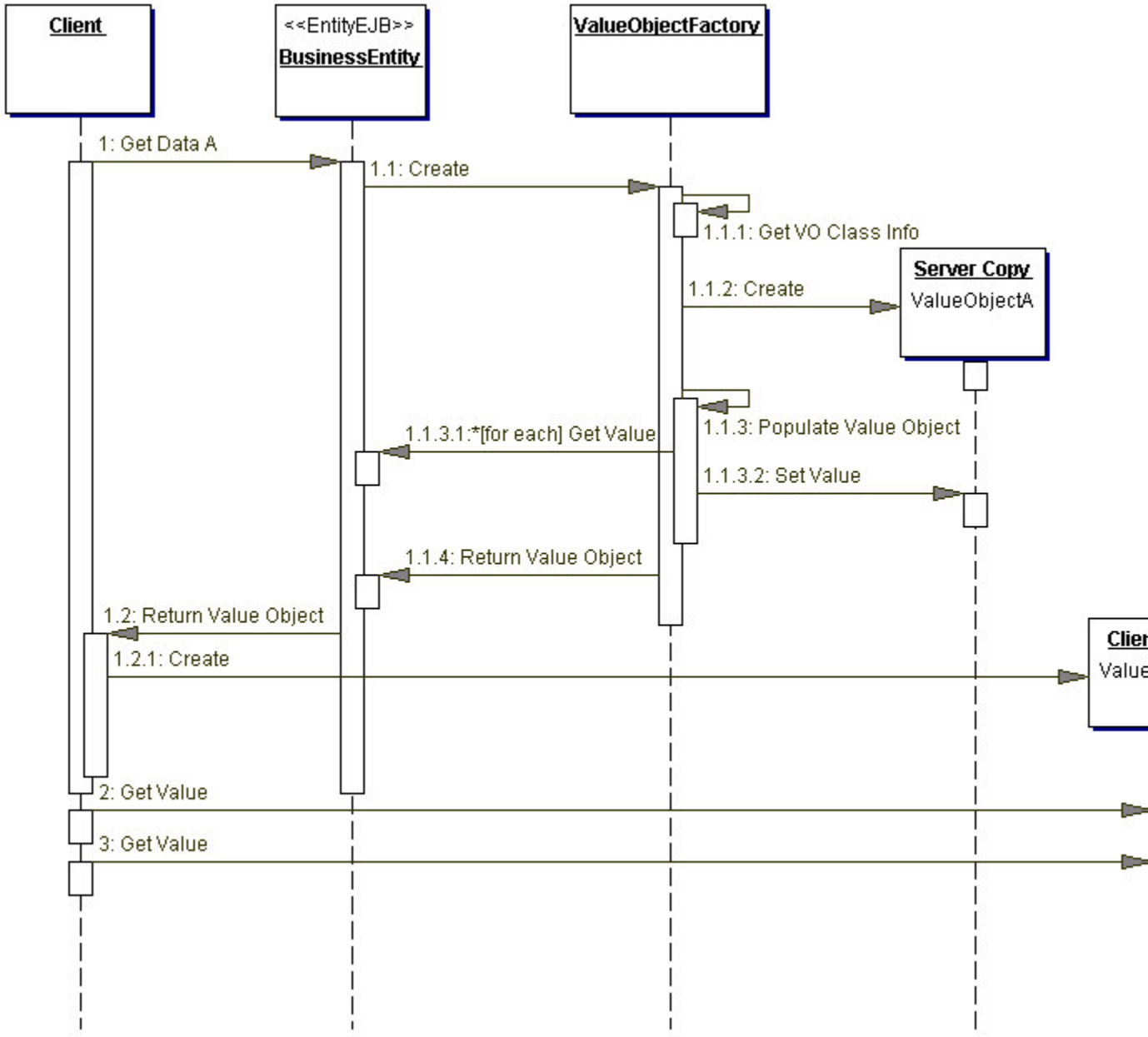


Figure 8.14 Transfer Object Factory strategy sequence diagram

The client requests the Transfer Object from the BusinessEntity. The BusinessEntity passes the required Transfer Object's class to the TransferObjectFactory, which creates a new Transfer Object of that given class. The TransferObjectFactory uses reflection to dynamically obtain the class information for the Transfer Object class and construct a new Transfer Object instance. Getting values from and setting values into the BusinessEntity by the TransferObjectFactory is accomplished by using dynamic invocation.

An example implementation for this strategy is shown in the "Sample Code" section for "Implementing Transfer Object Factory Strategy" on page 284.

The benefits of applying the Transfer Object Factory Strategy are as follows:

There is less code to write in order to create Transfer Objects. The same Transfer Object factory class can be reused by different enterprise beans. When a Transfer Object class definition changes, the Transfer Object factory automatically handles this change without any additional coding effort. This increases maintainability and is less error prone to changes in Transfer Object definitions.

The Transfer Object Factory Strategy has the following consequences:

It is based on the fact that the enterprise bean implementation extends (inherits) from the complete Transfer Object. The complete Transfer Object needs to implement all the interfaces defined for different Transfer Objects that the entity bean needs to supply. Naming conventions must be adhered to in order to make this strategy work. Since reflection is used to dynamically inspect and construct Transfer Objects, there is a slight performance loss in construction. However, when the overall communication time is considered, such loss may be negligible in comparison.

There is a trade-off associated with this strategy. Its power and flexibility must be weighed against the performance overhead associated with runtime reflection.

Consequences

- **Simplifies Entity Bean and Remote Interface**
  The entity bean provides a `getData()` method to get the Transfer Object containing the attribute values. This may eliminate having multiple get methods implemented in the bean and defined in the bean's remote interface. Similarly, if the entity bean provides a `setData()` method to update the entity bean attribute values in a single method call, it may eliminate having multiple set methods implemented in the bean and defined in the bean's remote interface.
- **Transfers More Data in Fewer Remote Calls**
  Instead of multiple client calls over the network to the BusinessObject to get attribute values, this solution provides a single method call. At the same time, this one method call returns a greater amount of data to the client than the individual accessor methods each returned. When considering this pattern, you must consider the trade-off between fewer network calls versus transmitting more data per call. Alternatively, you can provide both individual attribute accessor methods (fine-grained get and set methods) and Transfer Object methods (coarse-grained get and set methods). The developer can choose the appropriate technique depending on the requirement.
- **Reduces Network Traffic**
  A Transfer Object transfers the values from the entity bean to the client in one remote method call. The Transfer Object acts as a data carrier and reduces the number of remote network method calls required to obtain the attribute values from the entity beans. The reduced chattiness of the application results in better network performance.
- **Reduces Code Duplication**
  By using the Entity Inherits Transfer Object Strategy and the Transfer Object Factory Strategy, it is possible to reduce or eliminate the duplication of code between the entity and its Transfer Object. However, with the use of Transfer Object Factory Strategy, there could be increased complexity in implementation. There is also a runtime cost associated with this strategy due to the use of dynamic reflection. In most cases, the Entity Inherits Transfer Object Strategy may be sufficient to meet the needs.
- **May Introduce Stale Transfer Objects**
  Adopting the Updatable Transfer Objects Strategy allows the client to perform modifications on the local copy of the Transfer Object. Once the modifications are completed, the client can invoke the entity's `setData()` method and pass the modified Transfer Object to the entity. The entity receives the

modifications and merges the new (modified) values with its attributes. However, there may be a problem with stale Transfer Objects. The entity updates its values, but it is unaware of other clients that may have previously requested the same Transfer Object. These clients may be holding in their local cache Transfer Object instances that no longer reflect the current copy of the entity's data. Because the entity is not aware of these clients, it is not possible to propagate the update to the stale Transfer Objects held by other clients.

- **May Increase Complexity due to Synchronization and Version Control**
  The entity merges modified values into its own stored values when it receives a mutable Transfer Object from a client. However, the entity must handle the situation where two or more clients simultaneously request conflicting updates to the entity's values. Allowing such updates may result in data conflicts. Version control is one way of avoiding such conflict. As one of its attributes, the entity can include a version number or a last-modified time stamp. The version number or time stamp is copied over from the entity bean into the Transfer Object. An update transaction can resolve conflicts using the time stamp or version number attribute. If a client holding a stale Transfer Object tries to update the entity, the entity can detect the stale version number or time stamp in the Transfer Object and inform the client of this error condition. The client then has to obtain the latest Transfer Object and retry the update. In extreme cases this can result in client starvation-the client might never accomplish its updates.
- **Concurrent Access and Transactions**
  When two or more clients concurrently access the BusinessObject, the container applies the transaction semantics of the EJB architecture. If, for an enterprise bean, the transaction isolation level is set to `TRANSACTION_SERIALIZED` in the deployment descriptor, the container provides the maximum protection to the transaction and ensures its integrity. For example, suppose the workflow for the first transaction involves obtaining a Transfer Object, then subsequently modifying the BusinessObject attributes in the process. The second transaction, since it is isolated to serialized transactions, will obtain the Transfer Object with the correct (most recently updated) values. However, for transactions with lesser restrictions than serialized, protection is less rigid, leading to inconsistencies in the Transfer Objects obtained by competing accesses. In addition, problems related to synchronization, stale Transfer Objects, and version control will have to be dealt with.

Sample Code

Implementing the Transfer Object Pattern

Consider an example where a business object called Project is modeled and implemented as an entity bean. The Project entity bean needs to send data to its clients in a Transfer Object when the client invokes its `getProjectData()` method. The Transfer Object class for this example, ProjectTO, is shown in Example 8.3.

Example 8.3 Implementing the Transfer Object Pattern - Transfer Object Class

```
// Transfer Object to hold the details for Project
public class ProjectTO implements java.io.Serializable {
    public String projectId;
    public String projectName;
    public String managerId;
    public String customerId;
    public Date startDate;
    public Date endDate;
    public boolean started;
    public boolean completed;
    public boolean accepted;
    public Date acceptedDate;
    public String projectDescription;
    public String projectStatus;

    // Transfer Object constructors...
}
```

The sample code for the entity bean that uses this Transfer Object is shown in Example 8.4.

Example 8.4 Implementing the Transfer Object Pattern - Entity Bean Class

```
...
public class ProjectEntity implements EntityBean {
    private EntityContext context;
    public String projectId;
    public String projectName;
    public String managerId;
    public String customerId;
    public Date startDate;
    public Date endDate;
    public boolean started;
    public boolean completed;
    public boolean accepted;
    public Date acceptedDate;
    public String projectDescription;
    public String projectStatus;
    private boolean closed;

    // other attributes...
    private ArrayList commitments;
    ...

    // Method to get Transfer Object for Project data
    public ProjectTO getProjectData() {
      return createProjectTO();
    }

    // method to create a new Transfer Object and
    // copy data from entity bean into the value
    // object
    private ProjectTO createProjectTO() {
        ProjectTO proj = new ProjectTO();
        proj.projectId = projectId;
        proj.projectName = projectName;
        proj.managerId = managerId;
        proj.startDate = startDate;
        proj.endDate = endDate;
        proj.customerId = customerId;
        proj.projectDescription = projectDescription;
        proj.projectStatus = projectStatus;
        proj.started = started;
        proj.completed = completed;
        proj.accepted = accepted;
        proj.closed = closed;
        return proj;
    }
    ...
}
```

Implementing the Updatable Transfer Objects Strategy

Example 8.4 can be extended to implement Updatable Transfer Objects Strategy. In this case, the entity bean would provide a `setProjectData()` method to update the entity bean by passing a Transfer Object that contains the data to be used to perform the update. The sample code for this strategy is shown in Example 8.5.

Example 8.5 Implementing Updatable Transfer Objects Strategy

```
...
public class ProjectEntity implements EntityBean {
    private EntityContext context;
  ...
  // attributes and other methods as in Example 8.4
  ...

  // method to set entity values with a Transfer Object
  public void setProjectData(ProjectTO updatedProj) {
    mergeProjectData(updatedProj);
  }

  // method to merge values from the Transfer Object into
  // the entity bean attributes
  private void mergeProjectData(ProjectTO updatedProj) {
    // version control check may be necessary here
    // before merging changes in order to
    // prevent losing updates by other clients
    projectId = updatedProj.projectId;
    projectName = updatedProj.projectName;
    managerId = updatedProj.managerId;
    startDate = updatedProj.startDate;
    endDate = updatedProj.endDate;
    customerId = updatedProj.customerId;
    projectDescription                  =
        updatedProj.projectDescription;
    projectStatus = updatedProj.projectStatus;
    started = updatedProj.started;
    completed = updatedProj.completed;
    accepted = updatedProj.accepted;
    closed = updatedProj.closed;
  }
  ...
}
```

Implementing the Multiple Transfer Objects Strategy

Consider an example where a Resource entity bean is accessed by clients to request different Transfer Objects. The first type of Transfer Object, ResourceTO, is used to transfer data for a small set of attributes. The second type of Transfer Object, ResourceDetailsTO, is used to transfer data for a larger set of attributes. The client can use the former Transfer Object if it needs only the most basic data represented by that Transfer Object, and can use the latter if it needs more detailed information. Note that this strategy can be applied in producing two or more Transfer Objects that contain different data, and not just subset-superset as shown here.

The sample code for the two Transfer Objects for this example are shown in Example 8.6 and Example 8.7. The sample code for the entity bean that produces these Transfer Objects is shown in Example 8.8, and finally the entity bean client is shown in Example 8.9.

Example 8.6 Multiple Transfer Objects Strategy - ResourceTO

```
// ResourceTO: This class holds basic information
// about the resource
public class ResourceTO implements
  java.io.Serializable {
  public String resourceId;
  public String lastName;
  public String firstName;
  public String department;
  public String grade;
  ...
}
```

Example 8.7 Multiple Transfer Objects Strategy - ResourceDetailsTO

```
// ResourceDetailsTO This class holds detailed
// information about resource
public class ResourceDetailsTO {
  public String resourceId;
  public String lastName;
  public String firstName;
  public String department;
  public String grade;
  // other data...
  public Collection commitments;
  public Collection blockoutTimes;
  public Collection skillSets;
}
```

Example 8.8 Multiple Transfer Objects Strategy - Resource Entity Bean

```
// imports
...
public class ResourceEntity implements EntityBean {
  // entity bean attributes
  ...

  // entity bean business methods
  ...

  // Multiple Transfer Object method : Get ResourceTO
  public ResourceTO getResourceData() {

    // create new ResourceTO instance and copy
    // attribute values from entity bean into TO
    ...
    return createResourceTO();
  }
```

```
  // Multiple Transfer Object method : Get
  // ResourceDetailsTO
  public ResourceDetailsTO getResourceDetailsData() {

    // create new ResourceDetailsTO instance and copy
    // attribute values from entity bean into TO
    ...
    return createResourceDetailsTO();
  }

  // other entity bean methods
  ...
}
```

Example 8.9 Multiple Transfer Objects Strategy - Entity Bean Client

```
...
  private ResourceEntity resourceEntity;
  private static final Class homeClazz =

  corepatterns.apps.psa.ejb.ResourceEntityHome.class;
  ...
  try {
    ResourceEntityHome home =
      (ResourceEntityHome)
        ServiceLocator.getInstance().getHome(
            "Resource", homeClazz);
        resourceEntity = home.findByPrimaryKey(
                          resourceId);
  } catch(ServiceLocatorException ex) {
    // Translate Service Locator exception into
    // application exception
    throw new ResourceException(...);
  } catch(FinderException ex) {
    // Translate the entity bean finder exception into
    // application exception
    throw new ResourceException(...);
  } catch(RemoteException ex) {
    // Translate the Remote exception into
    // application exception
    throw new ResourceException(...);
  }
  ...
  // retrieve basic Resource data
  ResourceTO vo = resourceEntity.getResourceData();
  ...
  // retrieve detailed Resource data
  ResourceDetailsTO =
    resourceEntity.getResourceDetailsData();
  ...
```

Implementing the Entity Inherits Transfer Object Strategy

Consider an example where an entity bean ContactEntity inherits all its properties from a Transfer Object ContactTO. Example 8.10 shows the code sample for an example Transfer Object ContactTO that illustrates this strategy.

Example 8.10 Entity Inherits Transfer Object Strategy - Transfer Object Class

```
// This is the Transfer Object class inherited by
// the entity bean
public class ContactTO
  implements java.io.Serializable {

  // public members
  public String firstName;
  public String lastName;
  public String address;

  // default constructor
  public ContactTO() {}

  // constructor accepting all values
  public ContactTO(String firstName,
    String lastName, String address){
      init(firstName, lastName, address);
  }

  // constructor to create a new TO based
  // using an existing TO instance
  public ContactTO(ContactTO contact) {
    init (contact.firstName,
      contact.lastName, contact.address);
  }

  // method to set all the values
  public void init(String firstName, String
              lastName, String address) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.address = address;
  }

  // create a new Transfer Object
  public ContactTO getData() {
    return new ContactTO(this);
  }
}
```

The entity bean sample code relevant to this pattern strategy is shown in Example 8.11.

Example 8.11 Entity Inherits Transfer Object Strategy - Entity Bean Class

```
public class ContactEntity extends ContactTO implements
javax.ejb.EntityBean {
  ...
  // the client calls the getData method
  // on the ContactEntity bean instance.
```

```
  // getData() is inherited from the Transfer Object
  // and returns the ContactTO Transfer Object
  ...
}
```

Implementing Transfer Object Factory Strategy

Example 8.12 demonstrates the Transfer Object Factory strategy. The entity bean extends a complete Transfer Object called CustomerContactTO. The CustomerContactTO Transfer Object implements two interfaces, Customer and Contact. The CustomerTO Transfer Object implements Customer, and the ContactTO Transfer Object implements Contact.

Example 8.12 Transfer Object Factory Strategy - Transfer Objects and Interfaces

```
public interface Contact
  extends java.io.Serializable {
  public String getFirstName();
  public String getLastName();
  public String getContactAddress();
  public void setFirstName(String firstName);
  public void setLastName(String lastName);
  public void setContactAddress(String address);
}

public class ContactTO implements Contact {
  // member attributes
  public String firstName;
  public String lastName;
  public String contactAddress;

  // implement get and set methods per the
  // Contact interface here.
  ...
        }
public interface Customer
  extends java.io.Serializable {
  public String getCustomerName();
  public String getCustomerAddress();
  public void setCustomerName(String customerName);
  public void setCustomerAddress(String
      customerAddress);
}

public class CustomerTO implements Customer {
  public String customerName;
  public String customerAddress;

  // implement get and set methods per the
  // Customer interface here.
  ...
}

public class CustomerContactTO implements Customer,
  Contact {
  public String firstName;
```

```
  public String lastName;
  public String contactAddress;
  public String customerName;
  public String customerAddress;

  // implement get and set methods per the
  // Customer and Contact interfaces here.
  ...
}
```

The entity bean code sample to obtain these three different Transfer Objects is shown Example 8.13.

Example 8.13 Transfer Object Factory Strategy - Entity Bean Class

```
public class CustomerContactEntity extends
  CustomerContactTO implements javax.ejb.EntityBean {

  // implement other entity bean methods...not shown

  // define constant to hold class name
  // complete Transfer Object. This is required by
  // the TransferObjectFactory.createTransferObject(...)
  public static final String COMPLETE_TO_CLASSNAME =
      "CustomerContactTO";

  // method to return CustomerContactTO Transfer Object
  public CustomerContactTO getCustomerContact() {
    return (CustomerContactTO)
      TransferObjectFactory.createTransferObject(
        this, "CustomerContactTO",
        COMPLETE_TO_CLASSNAME);
  }

// method to return CustomerTO Transfer Object
  public CustomerTO getCustomer() {
    return (CustomerTO)
      TransferObjectFactory.createTransferObject(
        this, "CustomerTO",
        COMPLETE_TO_CLASSNAME);
  }

  // method to return ContactTO Transfer Object
  public ContactTO getContact() {
    return (ContactTO)
      TransferObjectFactory.createTransferObject(
        this, "ContactTO",
        COMPLETE_TO_CLASSNAME);
  }

  // other entity bean business methods
  ...
}
```

The TransferObjectFactory class is shown in Example 8.14.

Example 8.14 Transfer Object Factory Strategy - Factory Class

```
import java.util.HashMap;
import java.lang.*;

/**
* The factory class that creates a Transfer Object for a
* given EJB.
*/
public class TransferObjectFactory {

/**
* Use a HashMap to cache class information for
* Transfer Object classes
*/
private static HashMap classDataInfo = new HashMap();

/**
* Create a Transfer Object for the given object. The
* given object must be an EJB Implementation and have
* a superclass that acts as the class for the entity's
* Transfer Object. Only the fields defined in this
* superclass are copied in to the Transfer Object.
*/
public static java.io.Serializable
  createTransferObject(Object ejb,
    String whichTOType,
    String completeTOType) {
      try {
      // Get the class data for the complete
      // Transfer Object type
      ClassData cData = getClassData (completeTOType);

      // Get class data for the requested TO type
      ClassData voCData = getClassData (whichTOType);

      // Create the Transfer Object of the requested
      // Transfer Object type...
      java.lang.Object whichTO =
          Class.forName(whichTOType).newInstance();

      // get the TO fields for the requested TO
      // from the ClassData for the requested TO
      java.lang.reflect.Field[] voFields =
                  voCData.arrFields;

      // get all fields for the complete TO
      // from the ClassData for complete TO
      java.lang.reflect.Field[] beanFields =
                  cData.arrFields;

      // copy the common fields from the complete TO
      // to the fields of the requested TO
      for (int i = 0; i < voFields.length; i++) {
        try {
```

```
          String voFieldName = voFields[i].getName();
          for (int j=0; j < beanFields.length; j++) {
            // if the field names are same, copy value
            if ( voFieldName.equals(
                  beanFields[j].getName())) {
              // Copy value from matching field
              // from the bean instance into the new
              // Transfer Object created earlier
              voFields[i].set(whichTO,
                    beanFields[j].get(ejb));
              break;
            }
          }
        } catch (Exception e) {
          // handle exceptions that may be thrown
          // by the reflection methods...
        }
      }
    // return the requested Transfer Object
    return (java.io.Serializable)whichTO;
  } catch (Exception ex) {
    // Handle all exceptions here...
  }
  return null;
}

/**
* Return a ClassData object that contains the
* information needed to create
* a Transfer Object for the given class. This information
* is only obtained from the
* class using reflection once, after that it will be
* obtained from the classDataInfo HashMap.
*/
private static ClassData getClassData(String
  className){

  ClassData cData =
    (ClassData)classDataInfo.get(className);

  try {
    if (cData == null) {
      // Get the class of the given object and the
      // Transfer Object to be created
      java.lang.reflect.Field[] arrFields ;
      java.lang.Class ejbTOClass =
          Class.forName(className);

      // Determine the fields that must be copied
      arrFields = ejbTOClass.getDeclaredFields();

      cData = new ClassData(ejbTOClass, arrFields);
      classDataInfo.put(className, cData);
    }
  } catch (Exception e) {
    // handle exceptions here...
  }
```

```
  return cData;
  }
}

/**
* Inner Class that contains class data for the
* Transfer Object classes
*/
class ClassData {
  // Transfer Object Class
  public Class    clsTransferObject;

  // Transfer Object fields
  public java.lang.reflect.Field[] arrFields;

  // Constructor
  public ClassData(Class cls,
      java.lang.reflect.Field[] fields) {
    clsTransferObject = cls;
    arrFields = fields;
  }
}
```

Related Patterns

- **Session Facade**
  The Session Facade, which is the business interface for clients of J2EE applications, frequently uses Transfer Objects as an exchange mechanism with participating entity beans. When the facade acts as a proxy to the underlying business service, the Transfer Object obtained from the entity beans can be passed to the client.
- **Transfer Object Assembler**
  The Transfer Object Assembler is a pattern that builds composite Transfer Objects from different data sources. The data sources are usually session beans or entity beans that may be requested to provide their data to the Transfer Object Assembler as Transfer Objects. These Transfer Objects are considered to be parts of the composite object that the Transfer Object Assembler assembles.
- **Value List Handler**
  The Value List Handler is another pattern that provides lists of Transfer Objects constructed dynamically by accessing the persistent store at request time.
- **Composite Entity**
  The Transfer Object pattern addresses the need of getting data from BusinessObjects across tiers. This certainly is one aspect of design considerations for entity beans. The Composite Entity pattern discusses issues involved in designing coarse-grained entity beans. The Composite Entity pattern addresses complex requirements and discusses other factors and considerations involved in entity bean design.

Core J2EE Patterns - Composite Entity

Context

Entity beans are not intended to represent every persistent object in the object model. Entity beans are better suited for coarse-grained persistent business objects.

Problem

In a Java 2 Platform, Enterprise Edition (J2EE) application, clients -- such as applications, JavaServer Pages (JSP) pages, servlets, JavaBeans components -- access entity beans via their remote interfaces. Thus, every client invocation potentially routes through network stubs and skeletons, even if the client and the enterprise bean are in the same JVM, OS, or machine. When entity beans are fine-grained objects, clients tend to invoke more individual entity bean methods, resulting in high network overhead.

Entity beans represent distributed persistent business objects. Whether developing or migrating an application to the J2EE platform, object granularity is very important when deciding what to implement as an entity bean. Entity beans should represent coarse-grained business objects, such as those that provide complex behavior beyond simply getting and setting field values. These coarse-grained objects typically have dependent objects. A dependent object is an object that has no real domain meaning when not associated with its coarse-grained parent.

A recurring problem is the direct mapping of the object model to an Enterprise JavaBeans (EJB) model (specifically entity beans). This creates a relationship between the entity bean objects without consideration of coarse-grained versus fine-grained (or dependent) objects. Determining what to make coarse-grained versus fine-grained is typically difficult and can best be done via modeling relationships in Unified Modeling Language (UML) models.

There are a number of areas impacted by the fine-grained entity bean design approach:

- **Entity Relationships** - Directly mapping an object model to an EJB model does not take into account the impact of relationships between the objects. The inter-object relationships are directly transformed into inter-entity bean relationships. As a result, an entity bean might contain or hold a remote reference to another entity bean. However, maintaining remote references to distributed objects involves different techniques and semantics than maintaining references to local objects. Besides increasing the complexity of the code, it reduces flexibility, because the entity bean must change if there are any changes in its relationships.

  Also, there is no guarantee as to the validity of the entity bean references to other entity beans over time. Such references are established dynamically using the entity's home object and the primary key for that entity bean instance. This implies a high maintenance overhead of reference validity checking for each such entity-bean-to-entity-bean reference.

- **Manageability** - Implementing fine-grained objects as entity beans results in a large number of entity beans in the system. An entity bean is defined using several classes.

For each entity bean component, the developer must provide classes for the home interface, the remote interface, the bean implementation, and the primary key.

In addition, the container may generate classes to support the entity bean implementation. When the bean is created, these classes are realized as real objects in the container. In short, the container creates a number of objects to support each entity bean instance. Large numbers of entity beans result in more classes and code to maintain for the development team. It also results in a large number of objects in the container. This can negatively impact the application performance.

- **Network Performance** - Fine-grained entity beans potentially have more inter-entity bean relationships. Entity beans are distributed objects. When one entity bean invokes a method on another entity bean, the call is potentially treated as a remote call by the container, even if both entity beans are in the same container or JVM. If the number of entity-bean-to-entity-bean relationships increases, then this decreases system scalability due to heavy network overhead.
- **Database Schema Dependency** - When the entity beans are fine-grained, each entity bean instance usually represents a single row in a database. This is not a proper application of the entity bean design, since entity beans are more suitable for coarse-grained components. Fine-grained entity bean implementation typically is a direct representation of the underlying database schema in the entity bean design. When clients use these fine-grained entity beans, they are essentially operating at the row level in the database, since each entity bean is effectively a single row. Because the entity bean directly models a single database row, the clients become dependent on the database schema. When the schema changes, the entity bean definitions must change as well. Further, since the clients are operating at the same granularity, they must observe and react to this change. This schema dependency causes a loss of flexibility and increases the maintenance overhead whenever schema changes are required.
- **Object Granularity (Coarse-Grained versus Fine-Grained)** - Object granularity impacts data transfer between the enterprise bean and the client. In most applications, clients typically need a larger chunk of data than one or two rows from a table. In such a case, implementing each of these fine-grained objects as an entity bean means that the client would have to manage the relationships between all these fine-grained objects. Depending on the data requirements, the client might have to perform many lookups of a number of entity beans to obtain the required information.

Forces

- Entity beans are best implemented as coarse-grained objects due to the high overhead associated with each entity bean. Each entity bean is implemented using several objects, such as EJB home object, remote object, bean implementation, and primary key, and each is managed by the container services.
- Applications that directly map relational database schema to entity beans (where each row in a table is represented by an entity bean instance) tend to have a large number of fine-grained entity beans. It is desirable to keep the entity beans coarse-grained and reduce the number of entity beans in the application.

- Direct mapping of object model to EJB model yields fine-grained entity beans. Fine-grained entity beans usually map to the database schema. This entity-to-database row mapping causes problems related to performance, manageability, security, and transaction handling. Relationships between tables are implemented as relationships between entity beans, which means that entity beans hold references to other entity beans to implement the fine-grained relationships. It is very expensive to manage inter-entity bean relationships, because these relationships must be established dynamically, using the entity home objects and the enterprise beans' primary keys.
- Clients do not need to know the implementation of the database schema to use and support the entity beans. With fine-grained entity beans, the mapping is usually done so that each entity bean instance maps to a single row in the database. This fine-grained mapping creates a dependency between the client and the underlying database schema, since the clients deal with the fine-grained beans and they are essentially a direct representation of the underlying schema. This results in tight coupling between the database schema and entity beans. A change to the schema causes a corresponding change to the entity bean, and in addition requires a corresponding change to the clients.
- There is an increase in chattiness of applications due to intercommunication among fine-grained entity beans. Excessive inter-entity bean communication often leads to a performance bottleneck. Every method call to the entity bean is made via the network layer, even if the caller is in the same address space as the called bean (that is, both the client, or caller entity bean, and the called entity bean are in the same container). While some container vendors optimize for this scenario, the developer cannot rely on this optimization in all containers.
- Additional chattiness can be observed between the client and the entity beans because the client may have to communicate with many fine-grained entity beans to fulfill a requirement. It is desirable to reduce the communication between or among entity beans and to reduce the chattiness between the client and the entity bean layer.

Solution

Use Composite Entity to model, represent, and manage a set of interrelated persistent objects rather than representing them as individual fine-grained entity beans. A Composite Entity bean represents a graph of objects.

In order to understand this solution, let us first define what is meant by persistent objects and discuss their relationships.

A persistent object is an object that is stored in some type of data store. Multiple clients usually share persistent objects. Persistent objects can be classified into two types: coarse-grained objects and dependent objects.

A coarse-grained object is self-sufficient. It has its own life cycle and manages its relationships to other objects. Each coarse-grained object may reference or contain one or more other objects. The coarse-grained object usually manages the lifecycles of these objects. Hence, these objects are called dependent objects. A dependent object can be a simple self-contained object or may in turn contain other dependent objects.

The life cycle of a dependent object is tightly coupled to the life cycle of the coarse-grained object. A client may only indirectly access a dependent object through the coarse-grained object. That is, dependent objects are not directly exposed to clients because their parent (coarse-grained) object manages them. Dependent objects cannot exist by themselves. Instead, they always need to have their coarse-grained (or parent) object to justify their existence.

Typically, you can view the relationship between a coarse-grained object and its dependent objects as a tree. The coarse-grained object is the root of the tree (the root node). Each dependent object can be a standalone dependent object (a leaf node) that is a child of the coarse-grained object. Or, the dependent object can have parent-child relationships with other dependent objects, in which case it is considered a branch node.

A Composite Entity bean can represent a coarse-grained object and all its related dependent objects. Aggregation combines interrelated persistent objects into a single entity bean, thus drastically reducing the number of entity beans required by the application. This leads to a highly coarse-grained entity bean that can better leverage the benefits of entity beans than can fine-grained entity beans.

Without the Composite Entity approach, there is a tendency to view each coarse-grained and dependent object as a separate entity bean, leading to a large number of entity beans.

Structure

While there are many strategies in implementing the Composite Entity pattern, the first one we discuss is represented by the class diagram in Figure 8.17. Here the Composite Entity contains the coarse-grained object, and the coarse-grained object contains dependent objects.

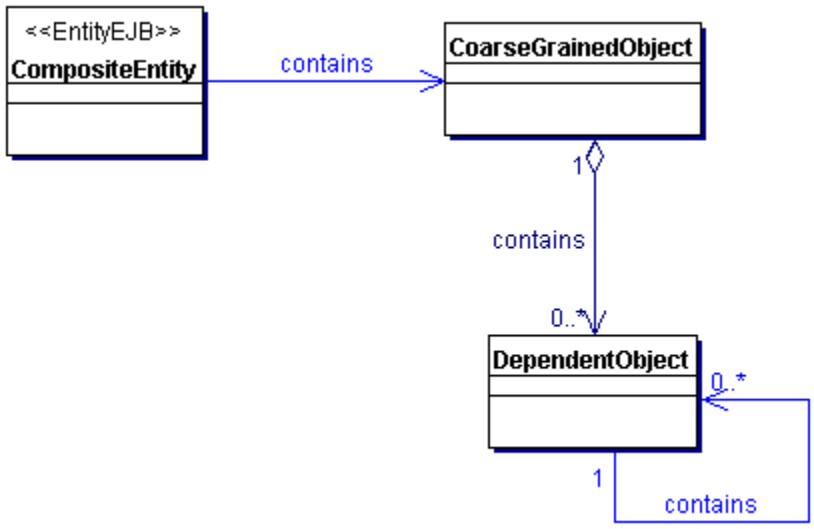


Figure 8.17 Composite Entity class diagram

The sequence diagram in Figure 8.18 shows the interactions for this pattern.

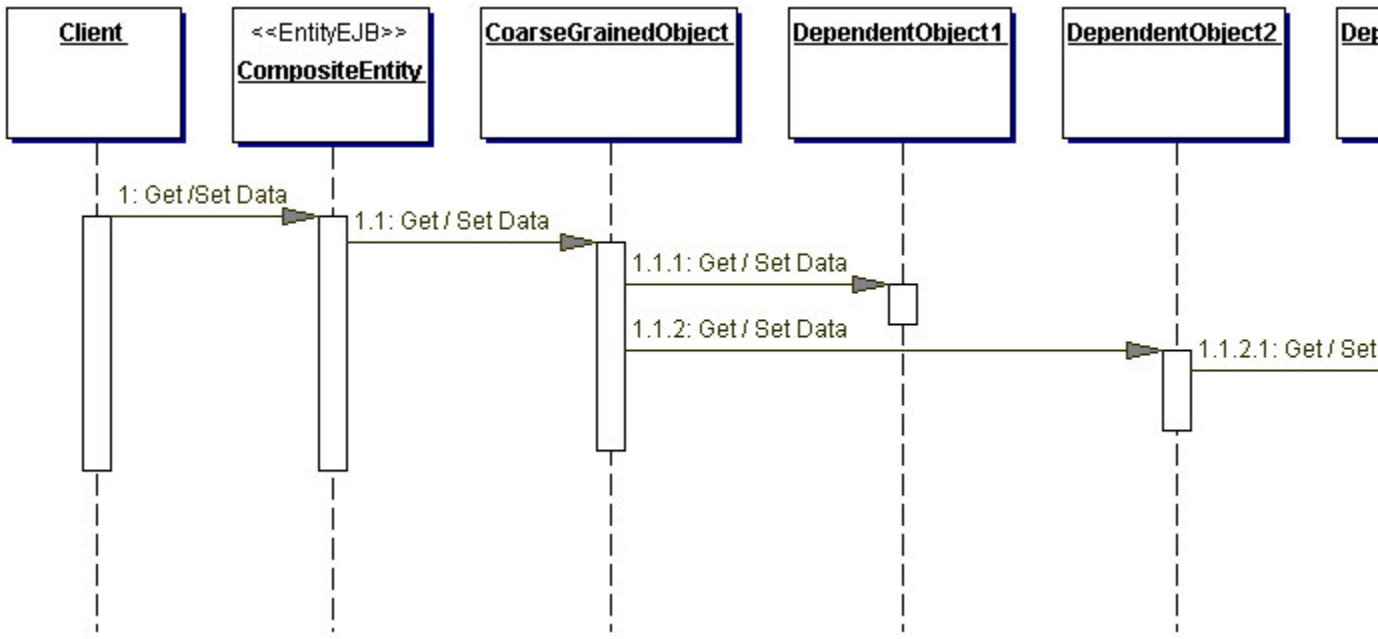


Figure 8.18 Composite Entity sequence diagram

Participants and Responsibilities

CompositeEntity

CompositeEntity is the coarse-grained entity bean. The CompositeEntity may be the coarse-grained object, or it may hold a reference to the coarse-grained object. The "Strategies" section explains the different implementation strategies for a Composite Entity.

Coarse-Grained Object

A coarse-grained object is an object that has its own life cycle and manages its own relationships to other objects. A coarse-grained object can be a Java object contained in the Composite Entity. Or, the Composite Entity itself can be the coarse-grained object that holds dependent objects. These strategies are explained in the "Strategies" section.

DependentObject1, DependentObject2, and DependentObject3

A dependent object is an object that depends on the coarse-grained object and has its life cycle managed by the coarse-grained object. A dependent object can contain other dependent objects; thus there may be a tree of objects within the Composite Entity.

Strategies

This section explains different strategies for implementing a Composite Entity. The strategies consider possible alternatives and options for persistent objects (coarse-grained and dependent) and the use of Transfer Objects.

Composite Entity Contains Coarse-Grained Object Strategy

In this strategy, the Composite Entity holds or contains the coarse-grained object. The

coarse-grained object continues to have relationships with its dependent objects. The structure section of this pattern describes this as the main strategy.

Composite Entity Implements Coarse-Grained Object Strategy

In this strategy, the Composite Entity itself is the coarse-grained object and it has the coarse-grained object's attributes and methods. The dependent objects are attributes of the Composite Entity. Since the Composite Entity is the coarse-grained object, the entity bean expresses and manages all relationships between the coarse-grained object and the dependent objects.

Figure 8.19 is the class diagram for this strategy.

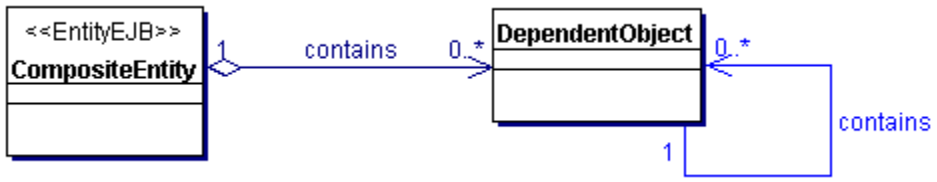


Figure 8.19 Composite Entity Implements Coarse-Grained Object class diagram

The sequence diagram for this strategy is shown in Figure 8.20.

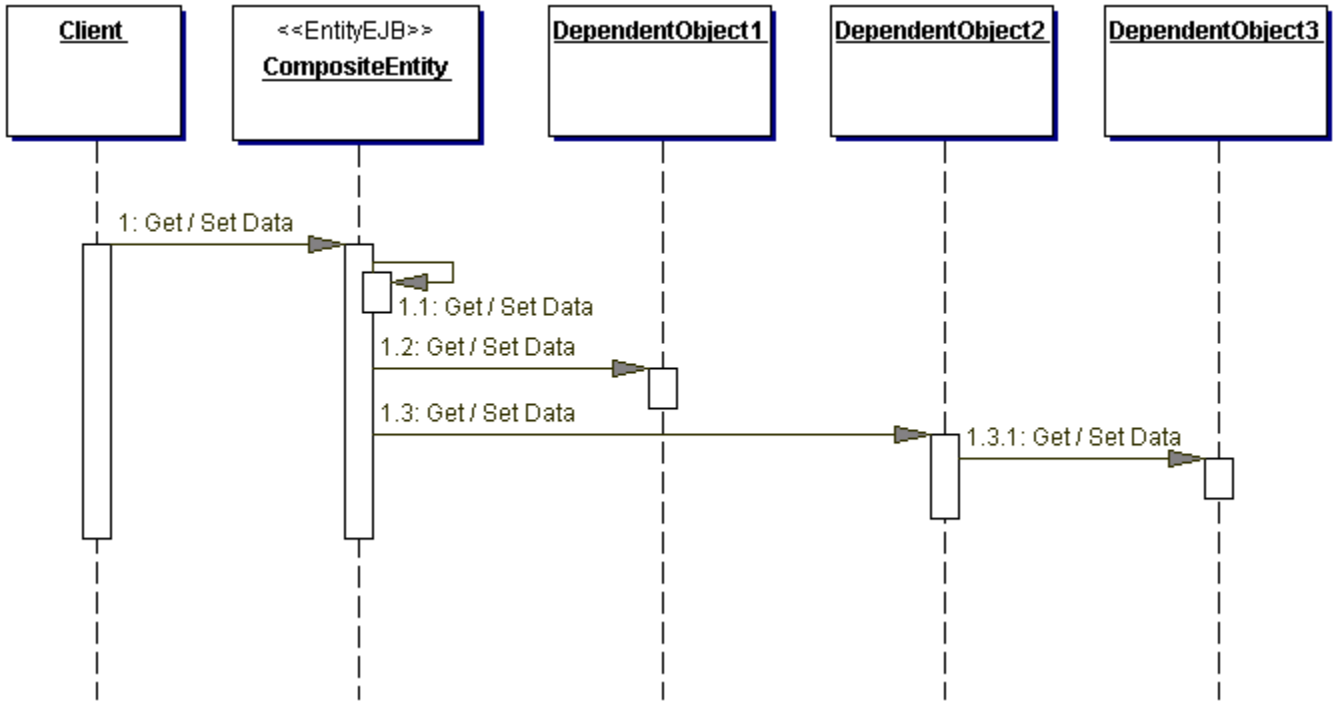


Figure 8.20 Composite Entity Implements Coarse-Grained Object sequence diagram

Lazy Loading Strategy

A Composite Entity can be composed of many levels of dependent objects in its tree of

objects. Loading all the dependent objects when the Composite Entity's `ejbLoad()` method is called by the EJB Container may take considerable time and resources. One way to optimize this is by using a lazy loading strategy for loading the dependent objects. When the `ejbLoad()` method is called, at first only load those dependent objects that are most crucial to the Composite Entity clients. Subsequently, when the clients access a dependent object that has not yet been loaded from the database, the Composite Entity can perform a load on demand. Thus, if some dependent objects are not used, they are not loaded on initialization. However, when the clients subsequently need those dependent objects, they get loaded at that time. Once a dependent object is loaded, subsequent container calls to the `ejbLoad()` method must include those dependent objects for reload to synchronize the changes with the persistent store.

Store Optimization (Dirty Marker) Strategy

A common problem with bean-managed persistence occurs when persisting the complete object graph during an `ejbStore()` operation. Since the EJB Container has no way of knowing what data has changed in the entity bean and its dependent objects, it puts the burden on the developer to determine what and how to persist the data. Some EJB containers provide a feature to identify what objects in Composite Entity's graph need to be stored due to a prior update. This may be done by having the developers implement a special method in the dependent objects, such as `isDirty()`, that is called by the container to check if the object has been updated since the previous `ejbStore()` operation.

A generic solution may be to use an interface, DirtyMarker, as shown in the class diagram in Figure 8.21. The idea is to have dependent objects implement the DirtyMarker interface to let the caller (typically the `ejbStore()` method) know if the state of the dependent object has changed. This way, the caller can choose to obtain the data for subsequent storage.

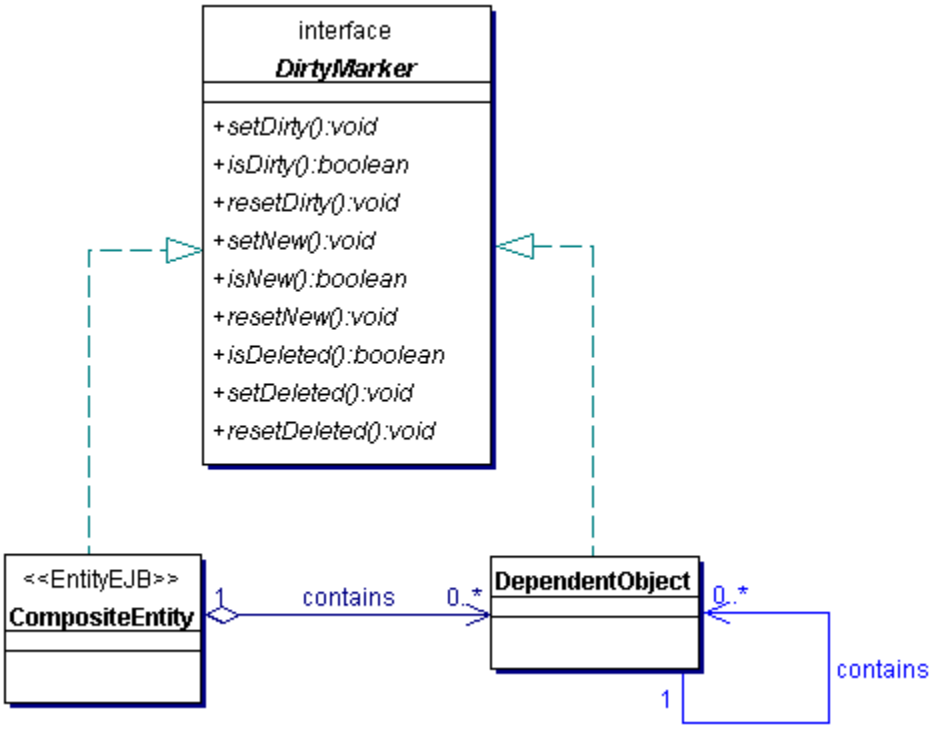


Figure 8.21 Store Optimization Strategy class diagram

Figure 8.22 contains a sequence diagram showing an example interaction for this strategy.

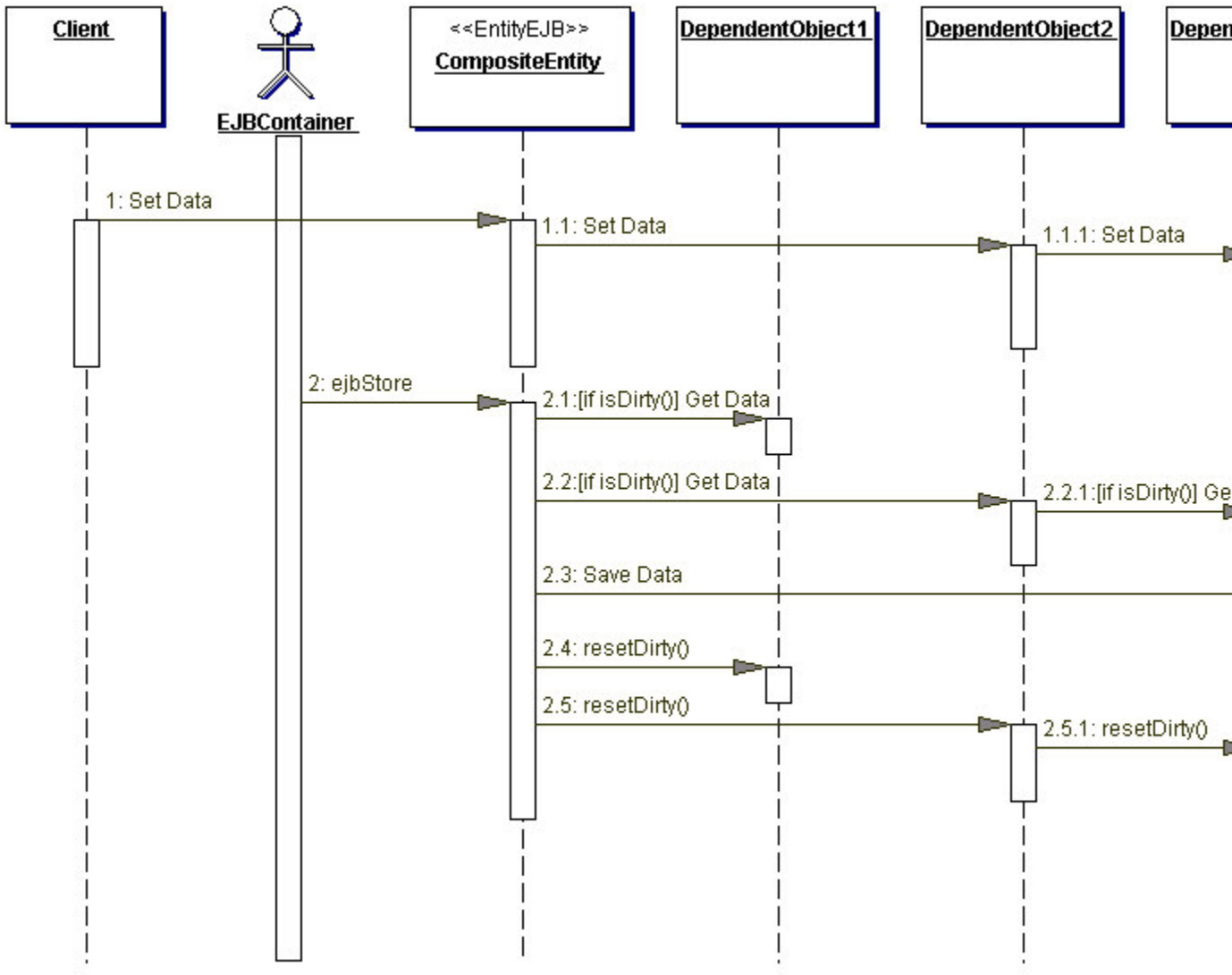


Figure 8.22 Store Optimization Strategy sequence diagram

The client performs an update to the Composite Entity, which results in a change to DependentObject3. DependentObject3 is accessed via its parent DependentObject2. The Composite Entity is the parent of DependentObject2. When this update is performed, the `setDirty()` method is invoked in the DependentObject3. Subsequently, when the container invokes the `ejbStore()` method on this Composite Entity instance, the `ejbStore()` method can check which dependent objects have gone dirty and selectively save those changes to the database. The dirty marks are reset once the store is successful.

The DirtyMarker interface can also include methods that can recognize other persistence status of the dependent object. For example, if a new dependent object is included into the Composite Entity, the `ejbStore()` method should be able to recognize what operation to use-in this case, the dependent object is not dirty, but is a new object. By extending the DirtyMarker interface to include a method called `isNewz()`, the `ejbStore()` method can invoke an insert operation instead of an update operation. Similarly, by including a

method called `isDeleted`(), the `ejbStore()` method can invoke delete operation as required.

In cases where `ejbStore()` is invoked with no intermediate updates to the Composite Entity, none of the dependent objects have been updated.

This strategy avoids the huge overhead of having to persist the entire dependent objects graph to the database whenever the `ejbStore()` method is invoked by the container.

Note

The EJB 2.0 specification addresses the Lazy Loading strategy and the Store Optimization strategy. The 2.0 specification is in final draft at the time of this writing. However, it is possible to use these strategies in pre-EJB 2.0 implementations. Please follow the EJB 2.0 developments to understand how these strategies will be finalized in the specification.

Composite Transfer Object Strategy

With a Composite Entity, a client can obtain all required information with just one remote method call. Because the Composite Entity either implements or holds the coarse-grained object and the hierarchy (or tree) of dependent objects, it can create the required Transfer Object and return it to the client by applying the Transfer Object pattern (see "Transfer Object" on page 261). The sequence diagram for this strategy is shown in Figure 8.23.

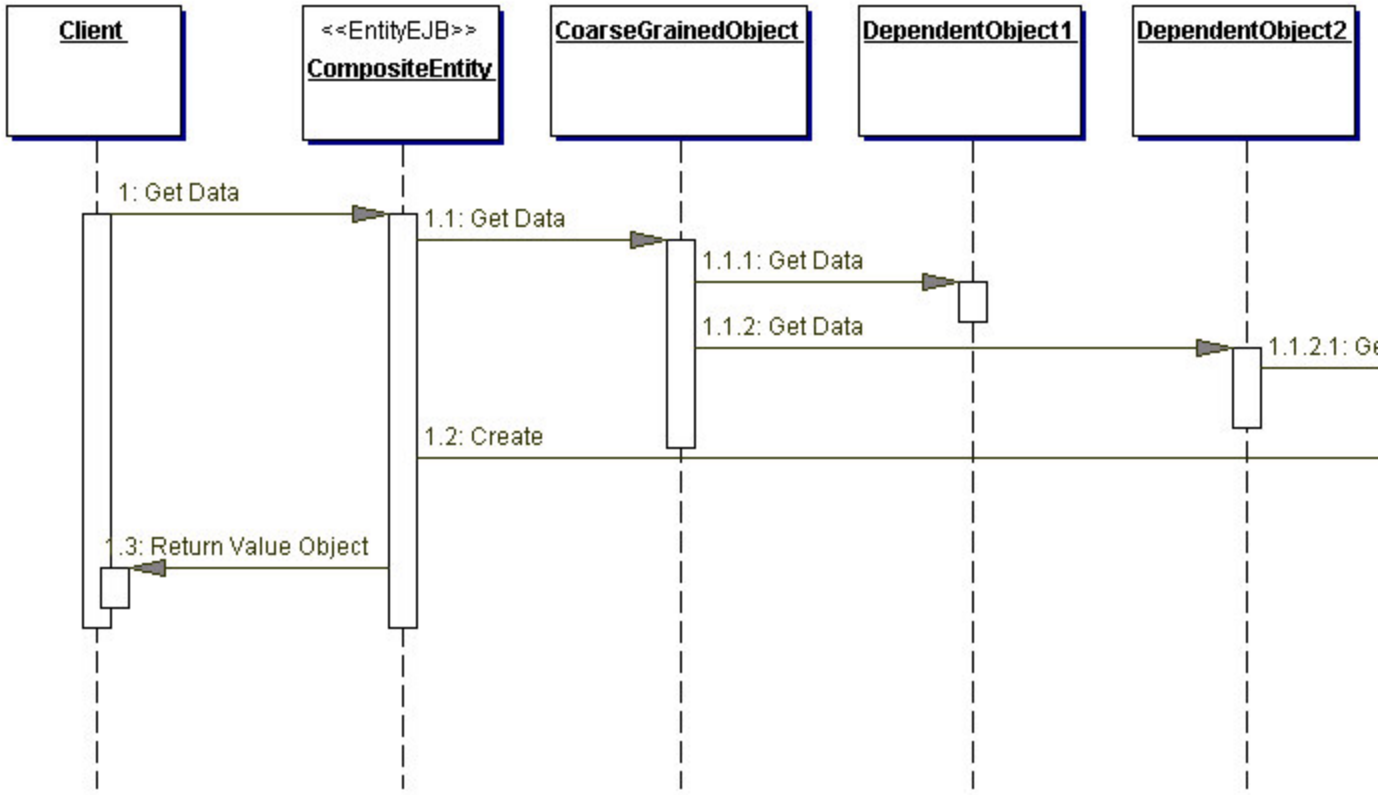


Figure 8.23 Composite Transfer Object Strategy sequence diagram

The Transfer Object can be a simple object or a composite object that has subobjects (a graph), depending on the data requested by the client. The Transfer Object is serializable and it is passed by value to the client. The Transfer Object functions only as a data transfer object; it has no responsibility with respect to security, transaction, and business logic. The Transfer Object packages all information into one object, obtaining the information with one remote call rather than multiple remote calls. Once the client receives the Transfer Object, all further calls from the client to the Transfer Object are local to the client.

This discussion points to how the entity can package all its data into a composite Transfer Object and return it to the client. However, this strategy also allows the entity bean to return only the required data to the client. If the client needs data only from a subset of dependent objects, then the composite Transfer Object returned can contain data derived from only those required parts and not from all the dependent objects. This would be an application of the Multiple Transfer Objects Strategy from the Transfer Object pattern (see "Transfer Object" on page 261).

Consequences

- **Eliminates Inter-Entity Relationships**
  Using the Composite Entity pattern, the dependent objects are composed into a single entity bean, eliminating all inter-entity-bean relationships. This pattern provides a central place to manage both relationships and object hierarchy.
- **Improves Manageability by Reducing Entity Beans**
  As discussed, implementing persistent objects as fine-grained entity beans results in a large number of classes that need to be developed and maintained. Using a Composite Entity reduces the number of EJB classes and code, and makes maintenance easier. It improves the manageability of the application by having fewer coarse-grained components instead of many more fine-grained components.
- **Improves Network Performance**
  Aggregation of the dependent objects improves overall performance. Aggregation eliminates all fine-grained communications between dependent objects across the network. If each dependent object were designed as a fine-grained entity bean, a huge network overhead would result due to inter-entity bean communications.
- **Reduces Database Schema Dependency**
  When the Composite Entity pattern is used, it results in coarse-grained entity bean implementations. The database schema is hidden from the clients, since the mapping of the entity bean to the schema is internal to the coarse-grained entity bean. Changes to the database schema may require changes to the Composite Entity beans. However, the clients are not affected since the Composite Entity beans do not expose the schema to the external world.
- **Increases Object Granularity**
  With a Composite Entity, the client typically looks up a single entity bean instead of a large number of fine-grained entity beans. The client requests the Composite Entity for data. The Composite Entity can create a composite Transfer Object that contains

all the data from the entity bean and return the Transfer Object to the client in a single remote method call. This reduces the chattiness between the client and the business tier.

- **Facilitates Composite Transfer Object Creation**
  By using this strategy, chattiness of the communication between the client and the entity bean is reduced, since the Composite Entity bean can return a composite Transfer Object by providing a mechanism to send serialized Transfer Objects from the Composite Entity bean. Although a Transfer Object returns all data in one remote call, the amount of data returned with this one call is much larger than the amount of data returned by separate remote calls to obtain individual entity bean properties. This trade-off works well when the goal is to avoid repeated remote calls and multiple lookups.
- **Overhead of Multi-level Dependent Object Graphs**
  If the dependent objects graph managed by the Composite Entity has many levels, then the overhead of loading and storing the dependent objects increases. This can be reduced by using the optimization strategies for load and store, but then there may be an overhead associated with checking the dirty objects to store and loading the required objects.

Sample Code

Consider a Professional Service Automation application (PSA) where a Resource business object is implemented using the Composite Entity pattern. The Resource represents the employee resource that is assigned to projects. Each Resource object can have different dependent objects as follows:

- BlockOutTime-This dependent object represents the time period the Resource is unavailable for reasons such as training, vacation, timeoffs, etc. Since each resource can have multiple blocked out times, the Resource-to-BlockOutTime relationship is a one-to-many relationship.
- SkillSet-This dependent object represents the Skill that a Resource possesses. Since each resource can have multiple skills, the Resource-to-SkillSet relationship is a one-to-many relationship.

Implementing the Composite Entity Pattern

The pattern for the Resource business object is implemented as a Composite Entity (ResourceEntity), as shown in Example 8.18. The one-to-many relationship with its dependent objects (BlockOutTime and SkillSet objects) are implemented using collections.

Example 8.18 Entity Implements Coarse-Grained Object

```
package corepatterns.apps.psa.ejb;

import corepatterns.apps.psa.core.*;
import corepatterns.apps.psa.dao.*;
import java.sql.*;
import javax.sql.*;
```

```
import java.util.*;
import javax.ejb.*;
import javax.naming.*;

public class ResourceEntity implements EntityBean {
  public String employeeId;
  public String lastName;
  public String firstName;
  public String departmentId;
  public String practiceGroup;
  public String title;
  public String grade;
  public String email;
  public String phone;
  public String cell;
  public String pager;
  public String managerId;

  // Collection of BlockOutTime Dependent objects
  public Collection blockoutTimes;

  // Collection of SkillSet Dependent objects
  public Collection skillSets;

  ...

  private EntityContext context;
// Entity Bean methods implementation
public String ejbCreate(ResourceTO resource) throws
  CreateException {
    try {
      this.employeeId = resource.employeeId;
      setResourceData(resource);
      getResourceDAO().create(resource);
    } catch(Exception ex) {
      throw new EJBException("Reason:" + ...);
    }
    return this.employeeId;
}

public String ejbFindByPrimaryKey(String primaryKey)
  throws FinderException {
    boolean result;
    try {
      ResourceDAO resourceDAO = getResourceDAO();
      result =
        resourceDAO.selectByPrimaryKey(primaryKey);
    } catch(Exception ex) {
      throw new EJBException("Reason:" + ...);
    }
    if(result) {
      return primaryKey;
    }
    else {
      throw new ObjectNotFoundException(...);
    }
  }
```

```
public void ejbRemove() {
  try {
    // Remove dependent objects
    if(this.skillSets != null) {

      SkillSetDAO skillSetDAO = getSkillSetDAO();
      skillSetDAO.setResourceID(employeeId);
      skillSetDAO.deleteAll();
      skillSets = null;
    }
    if(this.blockoutTime != null) {
      BlockOutTimeDAO blockouttimeDAO =
          getBlockOutTimeDAO();
      blockouttimeDAO.setResourceID(employeeId);
      blockouttimeDAO.deleteAll();
      blockOutTimes = null;
    }

    // Remove the resource from the persistent store
    ResourceDAO resourceDAO = new
      ResourceDAO(employeeId);
    resourceDAO.delete();
  } catch(ResourceException ex) {
    throw new EJBException("Reason:"+...);
  } catch(BlockOutTimeException ex) {
    throw new EJBException("Reason:"+...);
  } catch(Exception exception) {
    ...
  }
}
public void setEntityContext(EntityContext context)
{
  this.context = context;
}

public void unsetEntityContext() {
  context = null;
}

public void ejbActivate() {
  employeeId = (String)context.getPrimaryKey();
}

public void ejbPassivate() {
  employeeId = null;
}

public void ejbLoad() {
  try {
    // load the resource info from
    ResourceDAO resourceDAO = getResourceDAO();
    setResourceData((ResourceTO)
      resourceDAO.load(employeeId));

    // Load other dependent objects, if necessary
    ...
```

```
    } catch(Exception ex) {
      throw new EJBException("Reason:" + ...);
    }
  }

  public void ejbStore() {
    try {
      // Store resource information
      getResourceDAO().update(getResourceData());

      // Store dependent objects as needed
      ...
    } catch(SkillSetException ex) {
      throw new EJBException("Reason:" + ...);
    } catch(BlockOutTimeException ex) {
      throw new EJBException("Reason:" + ...);
    }
    ...
  }
  public void ejbPostCreate(ResourceTO resource) {
  }

  // Method to Get Resource Transfer Object
  public ResourceTO getResourceTO() {
    // create a new Resource Transfer Object
    ResourceTO resourceTO = new
        ResourceTO(employeeId);

    // copy all values
    resourceTO.lastName = lastName;
    resourceTO.firstName = firstName;
    resourceTO.departmentId = departmentId;
    ...
    return resourceTO;
  }

  public void setResourceData(ResourceTO resourceTO) {
    // copy values from Transfer Object into entity bean
    employeeId = resourceTO.employeeId;
    lastName = resourceTO.lastName;
    ...
  }

  // Method to get dependent Transfer Objects
  public Collection getSkillSetsData() {
    // If skillSets is not loaded, load it first.
    // See Lazy Load strategy implementation.

    return skillSets;
  }
  ...

  // other get and set methods as needed
  ...

  // Entity bean business methods
  public void addBlockOutTimes(Collection moreBOTs)
```

```
  throws BlockOutTimeException {
    // Note: moreBOTs is a collection of
    // BlockOutTimeTO objects
    try {
      Iterator moreIter = moreBOTs.iterator();
      while(moreIter.hasNext()) {
        BlockOutTimeTO botTO = (BlockOutTimeTO)
                          moreIter.next();
        if (! (blockOutTimeExists(botTO))) {
          // add BlockOutTimeTO to collection
          botTO.setNew();
          blockOutTime.add(botTO);
        } else {
          // BlockOutTimeTO already exists, cannot add
          throw new BlockOutTimeException(...);
        }
      }
    } catch(Exception exception) {
      throw new EJBException(...);
    }
  }

  public void addSkillSet(Collection moreSkills)
  throws SkillSetException {
    // similar to addBlockOutTime() implementation
    ...
  }

  ...

  public void updateBlockOutTime(Collection updBOTs)
  throws BlockOutTimeException {
    try {
      Iterator botIter = blockOutTimes.iterator();
      Iterator updIter = updBOTs.iterator();
      while (updIter.hasNext()) {
        BlockOutTimeTO botTO = (BlockOutTimeTO)
          updIter.next();
        while (botIter.hasNext()) {
          BlockOutTimeTO existingBOT =
            (BlockOutTimeTO) botIter.next();
          // compare key values to locate BlockOutTime
          if (existingBOT.equals(botTO)) {
            // Found BlockOutTime in collection
            // replace old BlockOutTimeTO with new one
            botTO.setDirty(); //modified old dependent
            botTO.resetNew(); //not a new dependent
            existingBOT = botTO;
          }
        }
      }
    } catch (Exception exc) {
      throw new EJBException(...);
    }
  }

  public void updateSkillSet(Collection updSkills)
```

```
  throws CommitmentException {
    // similar to updateBlockOutTime...
    ...
  }

  ...

}
```

Implementing the Lazy Loading Strategy

When the Composite Entity is first loaded in the `ejbLoad()` method by the container, let us assume that only the resource data is to be loaded. This includes the attributes listed in the ResourceEntity bean, excluding the dependent object collections. The dependent objects can then be loaded only if the client invokes a business method that needs these dependent objects to be loaded. Subsequently, the `ejbLoad()` needs to keep track of the dependent objects loaded in this manner and include them for reloading.

The relevant methods from the ResourceEntity class are shown in Example 8.19.

Example 8.19 Implementing Lazy Loading Strategy

```
...
public Collection getSkillSetsData() {
throws SkillSetException {
  checkSkillSetLoad();
  return skillSets;
}

private void checkSkillSetLoad()
throws SkillSetException {
  try {
    // Lazy Load strategy...Load on demand
    if (skillSets == null)
      skillSets =
        getSkillSetDAO(resourceId).loadAll();
  } catch(Exception exception) {
    // No skills, throw an exception
    throw new SkillSetException(...);
  }
}

...

public void ejbLoad() {
  try {
    // load the resource info from
    ResourceDAO resourceDAO = new
      ResourceDAO(employeeId);
    setResourceData((ResourceTO)resourceDAO.load());

    // If the lazy loaded objects are already
    // loaded, they need to be reloaded.
    // If there are not loaded, do not load them
    // here...lazy load will load them later.
    if (skillSets != null) {
```

```
      reloadSkillSets();
    }
    if (blockOutTimes != null) {
      reloadBlockOutTimes();
    }
    ...
    throw new EJBException("Reason:"+...);
  }
}


...
```

Implementing the Store Optimization (Dirty Marker) Strategy

To use the Store Optimization strategy, the dependent objects need to have implemented the DirtyMarker interface, as shown in Example 8.20. The `ejbStore()` method to optimize using this strategy is listed in Example 8.21.

Example 8.20 SkillSet Dependent Object Implements DirtyMarker Interface

```
public class SkillSetTO implements DirtyMarker,
  java.io.Serializable {
  private String skillName;
  private String expertiseLevel;
  private String info;
  ...

  // dirty flag
  private boolean dirty = false;

  // new flag
  private boolean isnew = true;

  // deleted flag
  private boolean deleted = false;

  public SkillSetTO(...) {
    // initialization
    ...
    // is new TO
    setNew();
  }

  // get, set and other methods for SkillSet
  // all set methods and modifier methods
  // must call setDirty()
  public setSkillName(String newSkillName) {
    skillName = newSkillName;
    setDirty();
  }
  ...

  // DirtyMarker methods
  // used for modified Transfer Objects only
  public void setDirty() {
```

```
    dirty = true;
  }
  public void resetDirty() {
    dirty = false;
  }
  public boolean isDirty() {
    return dirty;
  }

  // used for new Transfer Objects only
  public void setNew() {
    isnew = true;
  }
  public void resetNew() {
    isnew = false;
  }
  public boolean isNew() {
    return isnew;
  }

  // used for deleted objects only
  public void setDeleted() {
    deleted = true;
  }
  public boolean isDeleted() {
    return deleted;
  }
  public void resetDeleted() {
    deleted = false;
  }

}
```

Example 8.21 Implementing Store Optimization

```
...

  public void ejbStore() {
    try {
      // Load the mandatory data
      getResourceDAO().update(getResourceData());

      // Store optimization for dependent objects
      // check dirty and store
      // Check and store commitments
      if (skillSets != null) {
        // Get the DAO to use to store
        SkillSetDAO skillSetDAO = getSkillSetDAO();
        Iterator skillIter = skillSet.iterator();
        while(skillIter.hasNext()) {
          SkillSetTO skill =
            (SkillSetTO) skillIter.next();
          if (skill.isNew()) {
            // This is a new dependent, insert it
            skillSetDAO.insert(skill);
```

```
            skill.resetNew();
            skill.resetDirty();
          }
          else if (skill.isDeleted()) {
            // delete Skill
            skillSetDAO.delete(skill);
            // Remove from dependents list
            skillSets.  remove(skill);
          }
          else if (skill.isDirty()) {
            // Store Skill, it has been modified
            skillSetDAO.update(skill);
            // Saved, reset dirty.
            skill.resetDirty();
            skill.resetNew();
          }
        }
      }

      // Similarly, implement store optimization
      // for other dependent objects such as

      // BlockOutTime, ...
      ...
    } catch(SkillSetException ex) {
      throw new EJBException("Reason:"+...);
    } catch(BlockOutTimeException ex) {
      throw new EJBException("Reason:"+...);
    } catch(CommitmentException ex) {
      throw new EJBException("Reason:"+...);
    }
  }

  ...
```

Implementing the Composite Transfer Object Strategy

Now consider the requirement where the client needs to obtain all the data from the ResourceEntity, and not just one part. This can be done using the Composite Transfer Object Strategy, as shown in Example 8.22.

Example 8.22 Implementing the Composite Transfer Object

```
public class ResourceCompositeTO {
  private ResourceTO resourceData;
  private Collection skillSets;
  private Collection blockOutTimes;

  // Transfer Object constructors
  ...

  // get and set methods
  ...
}
```

The ResourceEntity provides a `getResourceDetailsData()` method to return the ResourceCompositeTO composite Transfer Object, as shown in Example 8.23.

Example 8.23 Creating the Composite Transfer Object

```
...
public ResourceCompositeTO getResourceDetailsData() {
  ResourceCompositeTO compositeTO =
    new ResourceCompositeTO (getResourceData(),
        getSkillsData(), getBlockOutTimesData());
  return compositeTO;
}
...
```

Related Patterns

- **Transfer Object**
  The Composite Entity pattern uses the Transfer Object pattern for creating the Transfer Object and returning it to the client. The Transfer Object pattern is used to serialize the coarse-grained and dependent objects tree, or part of the tree, as required.
- **Session Facade**
  If dependent objects tend to be entity beans rather than the arbitrary Java objects, try to use the Session Facade pattern to manage the inter-entity-bean relationships.
- **Transfer Object Assembler**
  When it comes to obtaining a composite Transfer Object from the Composite Entity (see the "Facilitates Composite Transfer Object Creation" under the "Consequences" section), this pattern is similar to the Transfer Object Assembler pattern. However, in this case, the data sources for all the Transfer Objects in the composite are parts of the Composite Entity itself, whereas for the Transfer Object Assembler, the data sources can be different entity beans, session beans, DAOs, Java objects, and so on.

Entity Bean as a Dependent Object: Issues and Recommendations

Typically, we design dependent objects as Java objects that have a direct relationship with the parent coarse-grained object. However, there may be situations when a dependent object may appear as an entity bean itself. This can happen:

1. If the dependent object appears to be depending on two different parent objects (as is the case with association classes).
2. If the dependent object already exists as an entity bean in the same application or is imported from a different application.

In these cases, the lifestyle of the dependent object may not appear to be directly related to and managed by a single parent coarse-grained object. So, what do you do when a dependent object is an entity bean? When you see a dependent object that is not totally dependent on its parent object? Or when you cannot identify its sole parent object?

Let's consider each case in a little more detail.

Case 1: The Dependent Object Depends on Two Parent Objects

Let us explore this with the following example. A Commitment represents an association between a Resource and a Project.

Figure 8.24 shows an example class diagram with relationships between Project, Resource and Commitment.

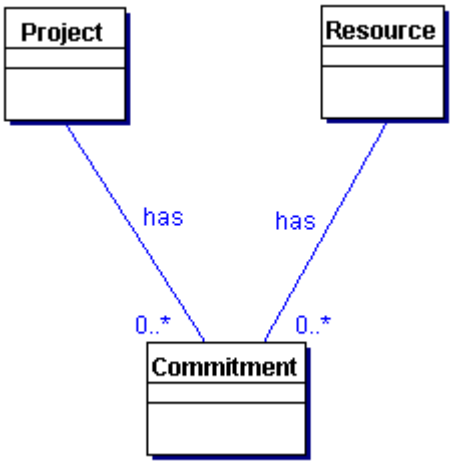


Figure 8.24 Example: Dependent object with two parent objects

Commitment is a dependent object. Both Projects and Resources are coarse-grained objects. Each Project has a one-to-many relationship with Commitment objects. Each Resource also has a one-to-many relationship with Commitment objects. So, is Commitment a dependent object of Project or of Resource? The answer lies in analyzing the interactions for the use cases that involve these three objects. If you make the Commitment a dependent of the Project, then when the Resource accesses its list Commitment objects, it has to do so through the Project object. On the other hand, if the Commitment is a dependent of a Resource, when the Project accesses its list of Commitment objects, it has to do so via the Resource. Both these choices will introduce entity-bean-to-entity-bean relationships in the design.

But, what if the Commitment is made an entity bean instead of a dependent object? Then the relationships between the Project and its list of Commitment objects, and between a Resource and its list of Commitment objects, will be entity-to-entity bean relationships. This just worsens the problem in that now there are two entity-bean-to-entity-bean relationships.

Entity-bean-to-entity-bean relationships are not recommended due to the overhead associated with managing and sustaining such a relationship.

Case 2: The Dependent Object Already Exists as an Entity Bean

In this case, it may seem that one way to model this relationship is to store the primary key of the dependent object in the coarse-grained object. When the coarse-grained object needs to access the dependent object, it results in an entity-bean- to-entity-bean invocation. The class diagram for this example is shown in Figure 8.25.

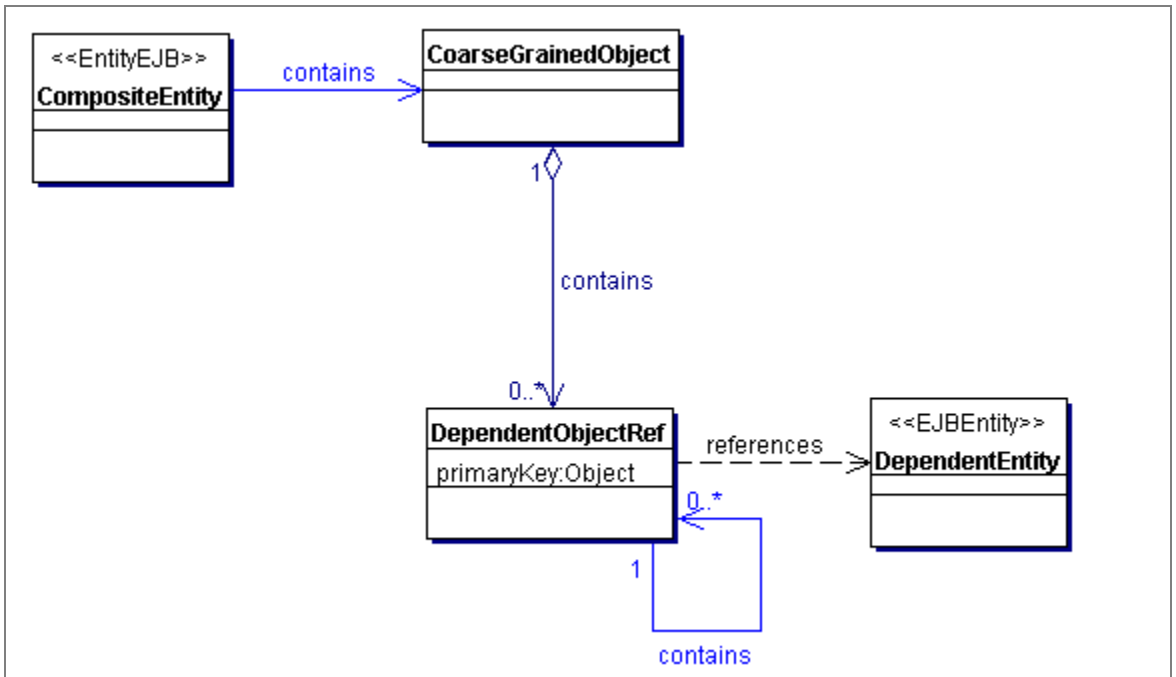


Figure 8.25 Dependent Object is an Entity Bean class diagram

The sequence diagram for this scenario is shown in Figure 8.26. The Composite Entity uses the dependent object references to look up the required dependent entity beans. The dependent object in this case is a proxy to the dependent entity bean, as shown.

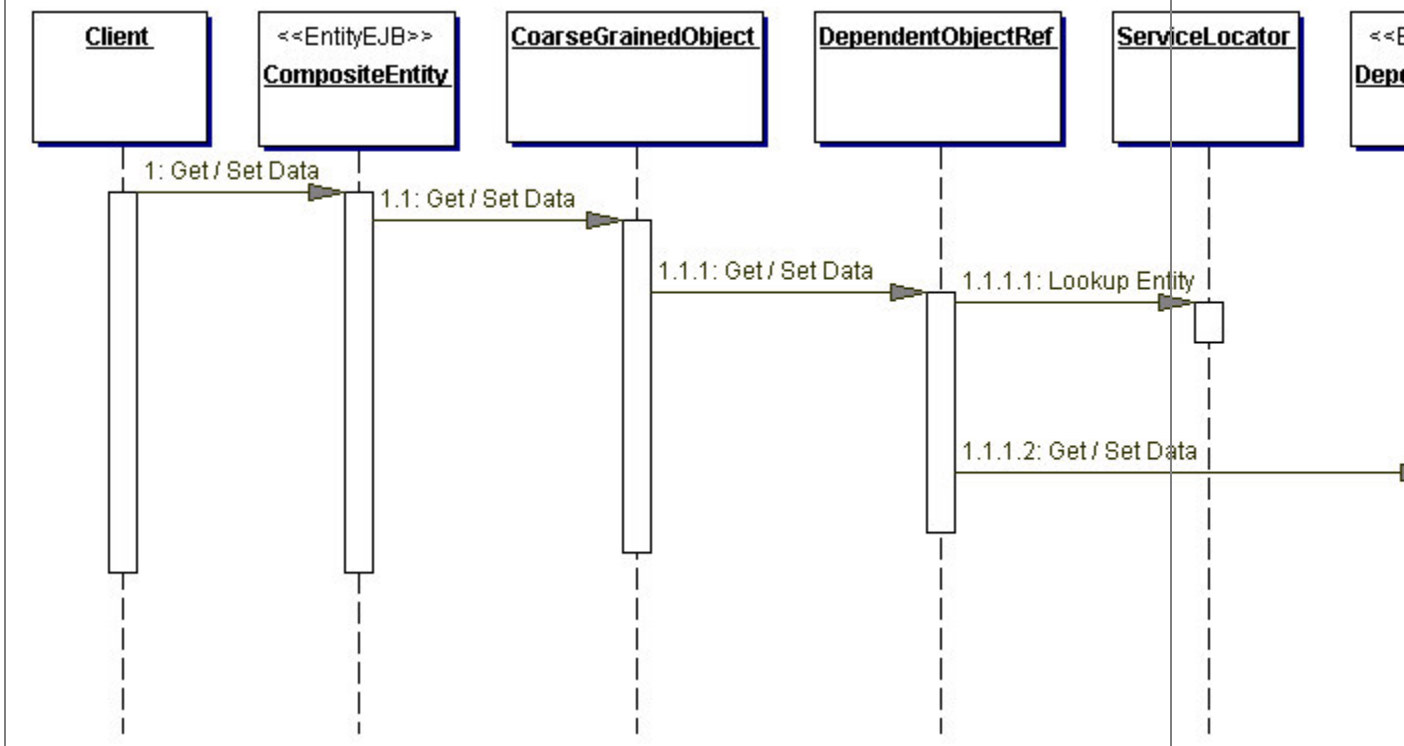


Figure 8.26 Dependent Object is an Entity Bean sequence diagram

While this may address the requirement of using a dependent entity bean from a parent entity bean, it is not an elegant solution. Instead, to avoid the complexity of designing and managing inter-entity relationships, consider using a session bean to help manage the relationships among entity beans. In our experience, we have found that the Session Facade pattern helps us to avoid this problem and provides a better way of managing entity-bean-to-entity-bean relationships.

So, we recommend avoiding entity-bean-to-entity-bean relationships as a best practice and to factor out such relationships into a session bean, using the Session Facade pattern (see "Session Facade" on page 291).

Core J2EE Patterns - Value List Handler

Context

The client requires a list of items from the service for presentation. The number of items in the list is unknown and can be quite large in many instances.

Problem

Most Java 2 Platform, Enterprise Edition (J2EE) applications have a search and query requirement to search and list certain data. In some cases, such a search and query operation could yield results that can be quite large. It is impractical to return the full result set when the client's requirements are to traverse the results, rather than process the complete set. Typically, a client uses the results of a query for read-only purposes, such as displaying the result list. Often, the client views only the first few matching records, and then may discard the remaining records and attempt a new query. The search activity often does not involve an immediate transaction on the matching objects. The practice of getting a list of values represented in entity beans by calling an `ejbFind()` method, which returns a collection of remote objects, and then calling each entity bean to get the value, is very network expensive and is considered a bad practice.

There are consequences associated with using Enterprise JavaBeans (EJB) finder methods that result in large results sets. Every container implementation has a certain amount of finder method overhead for creating a collection of EJBObject references. Finder method behavior performance varies, depending on a vendor's container implementation. According to the EJB specification, a container may invoke `ejbActivate()` methods on entities found by a finder method. At a minimum, a finder method returns the primary keys of the matching entities, which the container returns to the client as a collection of EJBObject references. This behavior applies for all container implementations. Some container implementations may introduce additional finder method overhead by associating the entity bean instances to these EJBObject instances to give the client access to those entity beans. However, this is a poor use of resources if the client is not interested in accessing the bean or invoking its methods. This overhead can significantly impede application performance if the application includes queries that produce many matching results.

Forces

- The application client needs an efficient query facility to avoid having to call the entity bean's `ejbFind()` method and invoking each remote object returned.
- A server-tier caching mechanism is needed to serve clients that cannot receive and process the entire results set.
- A query that is repeatedly executed on reasonably static data can be optimized to provide faster results. This depends on the application and on the implementation of this pattern.
- EJB finder methods are not suitable for browsing entire tables in the database or for searching large result sets from a table.
- Finder methods may have considerable overhead when used to find large numbers of result objects. The container may create a large number of infrastructure objects to

facilitate the finders.

- EJB finder methods are not suitable for caching results. The client may not be able to handle the entire result set in a single call. If so, the client may need server-side caching and navigation functions to traverse the result set.
- EJB finder methods have predetermined query constructs and offer minimum flexibility. The EJB specification 2.0 allows a query language, EJB QL, for container-managed entity beans. EJB QL makes it easier to write portable finders and offers greater flexibility for querying.
- Client wants to scroll forward and backward within a result set.

Solution

Use a Value List Handler to control the search, cache the results, and provide the results to the client in a result set whose size and traversal meets the client's requirements.

This pattern creates a ValueListHandler to control query execution functionality and results caching. The ValueListHandler directly accesses a DAO that can execute the required query. The ValueListHandler stores the results obtained from the DAO as a collection of Transfer Objects. The client requests the ValueListHandler to provide the query results as needed. The ValueListHandler implements an Iterator pattern [GoF] to provide the solution.

Structure

The class diagram in Figure 8.29 illustrates the Value List Handler pattern.

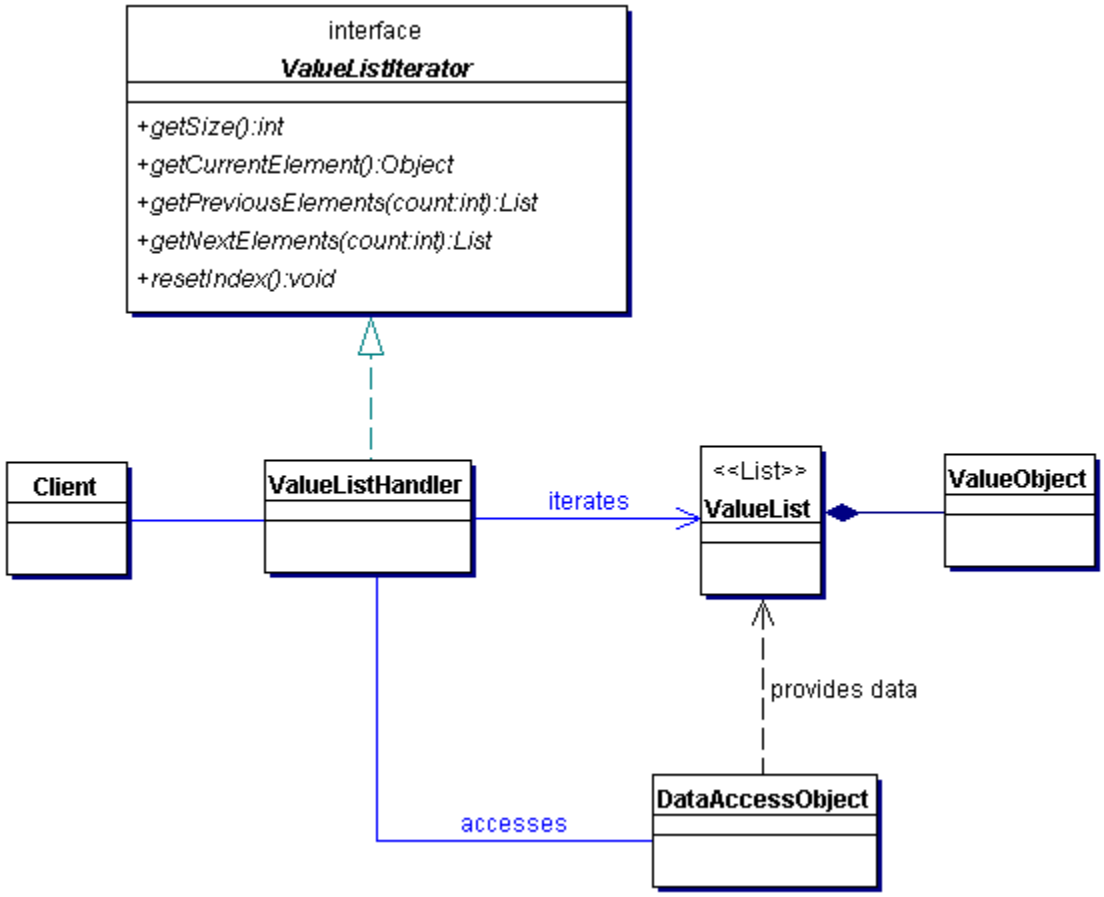


Figure 8.29 Value List Handler Class Diagram

Participants and Collaborations

The sequence diagram in Figure 8.30 shows the interactions for the Value List Handler.

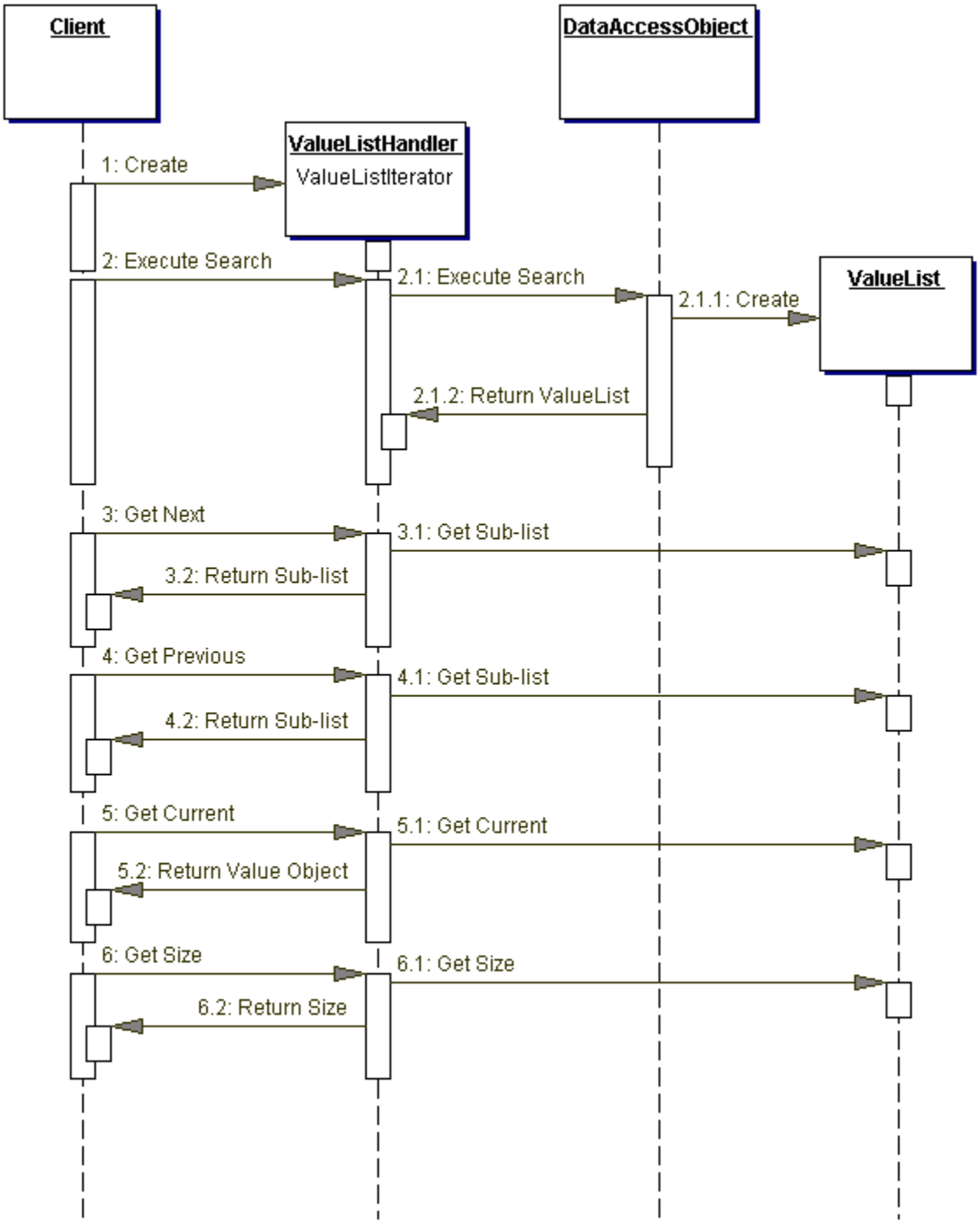


Figure 8.30 Value List Handler Sequence Diagram

ValueListIterator

This interface may provide iteration facility with the following example methods:

- `getSize()` obtains the size of the result set.
- `getCurrentElement()` obtains the current Transfer Object from the list.

- `getPreviousElements(int howMany)` obtains a collection of Transfer Objects that are in the list prior to the current element.
- `getNextElements(int howMany)` obtains a collection of Transfer Objects that are in the list after the current element.
- `resetIndex()` resets the index to the start of the list.

Depending on the need, other convenience methods can be included to be part of the ValueListIterator interface.

ValueListHandler

This is a list handler object that implements the ValueListIterator interface. The ValueListHandler executes the required query when requested by the client. The ValueListHandler obtains the query results, which it manages in a privately held collection represented by the ValueList object. The ValueListHandler creates and manipulates the ValueList collection. When the client requests the results, the ValueListHandler obtains the Transfer Objects from the cached ValueList, creates a new collection of Transfer Objects, serializes the collection, and sends it back to the client. The ValueListHandler also tracks the current index and size of the list.

DataAccessObject

The ValueListHandler can make use of a DataAccessObject to keep separate the implementation of the database access. The DataAccessObject provides a simple API to access the database (or any other persistent store), execute the query, and retrieve the results.

ValueList

The ValueList is a collection (a list) that holds the results of the query. The results are stored as Transfer Objects. If the query fails to return any matching results, then this list is empty. The ValueListHandler session bean caches ValueList to avoid repeated, unnecessary execution of the query.

TransferObject

The TransferObject represents an object view of the individual record from the query's results. It is an immutable serializable object that provides a placeholder for the data attributes of each record.

Strategies

Java Object Strategy

The ValueListHandler can be implemented as an arbitrary Java object. In this case, the ValueListHandler can be used by any client that needs the listing functionality. For applications that do not use enterprise beans, this strategy is useful. For example, simpler applications may be built using servlets, JavaServer Pages (JSP) pages, Business Delegates, and DAOs. In this scenario, the Business Delegates can use a ValueListHandler implemented as a Java object to obtain list of values.

Stateful Session Bean Strategy

When an application uses enterprise beans in the business tier, it may be preferable to

implement a session bean that uses the ValueListHandler. In this case, the session bean simply fronts an instance of a ValueListHandler. Thus, the session bean may be implemented as a stateful session bean to hold on to the list handler as its state, and thus may simply act as a facade (see "Session Facade" on page 291) or as a proxy.

Consequences

- **Provides Alternative to EJB Finders for Large Queries**
  Typically, an EJB finder method is a resource-intensive and an expensive way of obtaining a list of items, since it involves a number of EJBObject references. The Value List Handler implements a session bean that uses a DAO to perform the query and to create a collection of Transfer Objects that match the query criteria. Because Transfer Objects have relatively low overhead compared to EJBObject references and their associated infrastructure, this pattern provides benefits when application clients require queries resulting in large result sets.
- **Caches Query Results on Server Side**
  The result set obtained from a query execution needs to be cached when a client must display the results in small subsets rather than in one large list. However, not all browser-based clients can perform such caching. When they cannot, the server must provide this functionality. The Value List Handler pattern provides a caching facility in the Value List Handler session bean to hold the result set obtained from a query execution. The result set is a collection of Transfer Objects that can be serialized if required.

  When the client requests a collection, or a subset of a collection, the handler bean returns the requested results as a serialized collection of Transfer Objects. The client receives the collection and now has a local copy of the requested information, which the client can display or process. When the client needs an additional subset of the results, it requests the handler to return another serialized collection containing the required results. The client can process the query results in smaller, manageable chunks. The handler bean also provides the client with navigation facilities (previous and next) so that the results may be traversed forward and backward as necessary.

- **Provides Better Querying Flexibility**
  Adding a new query may require creating a new finder method or modifying an existing method, especially when using bean-managed entity beans. (With bean-managed entity beans, the developer implements the finder methods in the bean implementation.) With a container-managed entity bean, the deployer specifies the entity bean finder methods in the bean's deployment descriptor. Changes to a query for a container-managed bean require changes to the finder method specification in the deployment descriptor. Therefore, finder methods are ill-suited to handle query requirements that change dynamically. You can implement a Value List Handler to be more flexible than EJB finder methods by providing ad hoc query facilities, constructing runtime query arguments using template methods, and so forth. In other words, a Value List Handler developer can implement intelligent searching and caching algorithms without being limited by the finder methods.
- **Improves Network Performance**

Network performance may improve because only requested data, rather than all data, is shipped (serialized) to the client on an as-needed basis. If the client displays the first few results and then abandons the query, the network bandwidth is not wasted, since the data is cached on the server side and never sent to the client. However, if the client processes the entire result set, it makes multiple remote calls to the server for the result set. When the client knows in advance that it needs the entire result set, the handler bean can provide a method that sends the client the entire result set in one method call, and the pattern's caching feature is not used.

- **Allows Deferring Entity Bean Transactions**
  Caching results on the server side and minimizing finder overhead may improve transaction management. When the client is ready to further process an entity bean, it accesses the bean within a transaction context defined by the use case. For example, a query to display a list of books uses a Value List Handler to obtain the list. When the user wants to view a book in detail, it involves the book's entity bean in a transaction.

Sample Code

Implementing the Value List Handler as a Java Object

Consider an example where a list of Project business objects are to be retrieved and displayed. The Value List Handler pattern can be applied in this case. The sample code for this implementation is listed in Example 8.29 as ProjectListHandler, which is responsible to provide the list of Projects. This class extends the `ValueListHandler` base class, which provides the generic iteration functionality for all Value List Handler implementations in this application. The `ValueListHandler` sample code is listed in Example 8.30. The `ValueListHandler` implements the generic iterator interface `ValueListIterator`, which is shown in Example 8.32. The relevant code sample from the data access object ProjectDAO, used by `ValueListHandler` to execute the query and obtain matching results, is shown in Example 8.31.

Example 8.29 Implementing Value List Handler Pattern

```
package corepatterns.apps.psa.handlers;

import java.util.*;
import corepatterns.apps.psa.dao.*;
import corepatterns.apps.psa.util.*;
import corepatterns.apps.psa.core.*;

public class ProjectListHandler
extends ValueListHandler {

  private ProjectDAO dao = null;
  // use ProjectTO as a template to determine
  // search criteria
  private ProjectTO projectCriteria = null;

  // Client creates a ProjectTO instance, sets the
  // values to use for search criteria and passes
  // the ProjectTO instance as projectCriteria
  // to the constructor and to setCriteria() method
  public ProjectListHandler(ProjectTO projectCriteria)
```

```
  throws ProjectException, ListHandlerException {
    try {
      this.projectCriteria = projectCriteria;
      this.dao = PSADAOFactory.getProjectDAO();
      executeSearch();
    } catch (Exception e) {
      // Handle exception, throw ListHandlerException
    }
  }

  public void setCriteria(ProjectTO projectCriteria) {
    this.projectCriteria = projectCriteria;
  }

  // executes search. Client can invoke this
  // provided that the search criteria has been
  // properly set. Used to perform search to refresh
  // the list with the latest data.
  public void executeSearch()
  throws ListHandlerException {
    try {
      if (projectCriteria == null) {
        throw new ListHandlerException(
          "Project Criteria required...");
      }
      List resultsList =
        dao.executeSelect(projectCriteria);
      setList(resultsList);
    } catch (Exception e) {
      // Handle exception, throw ListHandlerException
    }
  }
}
```

The Value List Handler is a generic iterator class that provides the iteration functionality.

Example 8.30 Implementing Generic ValueListHandler Class

```
package corepatterns.apps.psa.util;

import java.util.*;

public class ValueListHandler
implements ValueListIterator {

  protected List list;
  protected ListIterator listIterator;

  public ValueListHandler() {
  }

  protected void setList(List list)
  throws IteratorException {
    this.list = list;
    if(list != null)
      listIterator =  list.listIterator();
```

```
    else
      throw new IteratorException("List empty");
  }

  public Collection getList(){
    return list;
  }

  public int getSize() throws IteratorException{
    int size = 0;

    if (list != null)
      size = list.size();
    else
      throw new IteratorException(...); //No Data

    return size;
  }

  public Object getCurrentElement()
  throws IteratorException {

    Object obj = null;
    // Will not advance iterator
    if (list != null)
    {
      int currIndex = listIterator.nextIndex();
      obj = list.get(currIndex);
    }
    else
      throw new IteratorException(...);
    return obj;

  }

  public List getPreviousElements(int count)
  throws IteratorException {
    int i = 0;
    Object object = null;
    LinkedList list = new LinkedList();
    if (listIterator != null) {
      while (listIterator.hasPrevious() && (i < count)){
        object = listIterator.previous();
        list.add(object);
        i++;
      }
    }// end if
    else
      throw new IteratorException(...); // No data

    return list;
  }

  public List getNextElements(int count)
  throws IteratorException {
    int i = 0;
    Object object = null;
```

```
    LinkedList list = new LinkedList();
    if(listIterator != null){
      while(  listIterator.hasNext() && (i < count) ){
        object = listIterator.next();
        list.add(object);
        i++;
      }
    } /  / end if
    else
      throw new IteratorException(...); // No data

    return list;
  }

  public void resetIndex() throws IteratorException{
    if(listIterator != null){
      listIterator = list.ListIterator();
    }
    else
      throw new IteratorException(...); // No data
  }
  ...
}
```

Example 8.31 ProjectDAO Class

```
package corepatterns.apps.psa.dao;

public class ProjectDAO {
  final private String tableName = "PROJECT";

  // select statement uses fields
  final private String fields = "project_id, name," +
      "project_manager_id, start_date, end_date, " +
      " started, completed, accepted, acceptedDate," +
      " customer_id, description, status";

  // the methods relevant to the ValueListHandler
  // are shown here.
  // See Data Access Object pattern for other details.
  ...
  private List executeSelect(ProjectTO projCriteria)
  throws SQLException {

    Statement stmt= null;
    List list = null;
    Connection con = getConnection();
    StringBuffer selectStatement =   new StringBuffer();
    selectStatement.append("SELECT "+ fields +
          " FROM " + tableName + "where 1=1");

    // append additional conditions to where clause
    // depending on the values specified in
    // projCriteria
    if (projCriteria.projectId != null) {
```

```
      selectStatement.append (" AND PROJECT_ID = '" +
        projCriteria.projectId + "'");
    }
    // check and add other fields to where clause
    ...

    try {
      stmt = con.prepareStatement(selectStatement);
      stmt.setString(1, resourceID);
      ResultSet rs = stmt.executeQuery();
      list = prepareResult(rs);
      stmt.close();
    }
    finally {
      con.close();
    }
    return list;
  }

  private List prepareResult(ResultSet rs)
  throws SQLException {
    ArrayList list = new ArrayList();
    while(rs.next()) {
      int i = 1;
      ProjectTO proj = new
        ProjectTO(rs.getString(i++));
      proj.projectName = rs.getString(i++);
      proj.managerId = rs.getString(i++);
      proj.startDate = rs.getDate(i++);
      proj.endDate = rs.getDate(i++);
      proj.started = rs.getBoolean(i++);
      proj.completed = rs.getBoolean(i++);
      proj.accepted = rs.getBoolean(i++);
      proj.acceptedDate = rs.getDate(i++);
      proj.customerId = rs.getString(i++);
      proj.projectDescription = rs.getString(i++);
      proj.projectStatus = rs.getString(i++);
      list.add(proj);

    }
    return list;
  }
  ...
}
```

Example 8.32 ValueListIterator Class

```
package corepatterns.apps.psa.util;

import java.util.List;

public interface ValueListIterator {

  public int getSize()
    throws IteratorException;
```

```
  public Object getCurrentElement()
    throws IteratorException;

  public List getPreviousElements(int count)
    throws IteratorException;

  public List getNextElements(int count)
    throws IteratorException;

  public void resetIndex()
    throws IteratorException;

  // other common methods as required
  ...
}
```

Related Patterns

- **Iterator [GoF]**
  This Value List Handler pattern is based on Iterator pattern, described in the GoF book, *Design Patterns: Elements of Reusable Object-Oriented Software*.
- **Session Facade**
  Since the Value List Handler is a session bean, it may appear as a specialized Session Facade. However, in isolation, it is a specialized session bean rather than a specialized Session Facade. A Session Facade has other motivations and characteristics (explained in the Session Facade pattern), and it is much coarser grained.

Iterator

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Problem

Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

Discussion

"An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you need to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you'll require. You might also need to have more than one traversal pending on the same list." [GOF, p257] And, providing a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration) might be valuable.

The Iterator pattern lets you do all this. The key idea is to take the responsibility for

access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal protocol.

The Iterator abstraction is fundamental to an emerging technology called "generic programming". This strategy seeks to explicitly separate the notion of "algorithm" from that of "data structure". The motivation is to: promote component-based development, boost productivity, and reduce configuration management.

As an example, if you wanted to support four data structures (array, binary tree, linked list, and hash table) and three algorithms (sort, find, and merge), a traditional approach would require four times three permutations to develop and maintain. Whereas, a generic programming approach would only require four plus three configuration items.

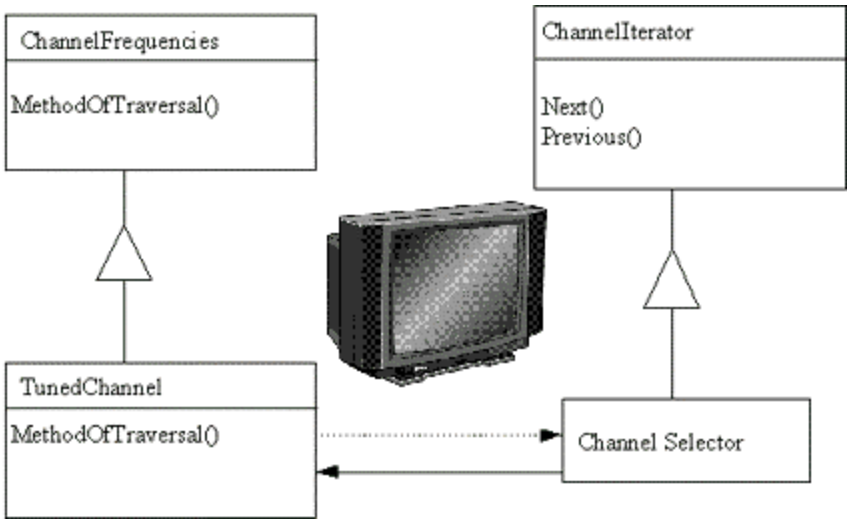
Structure

This diagram is from javacoder.net.

Example

The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. Files are aggregate objects. In office settings where access to files is made through administrative or secretarial staff, the Iterator pattern is demonstrated with the secretary acting as the Iterator. Several television comedy skits have been developed around the premise of an executive trying to understand the secretary's filing system. To the executive, the filing system is confusing and illogical, but the secretary is able to access files quickly and efficiently. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]

On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed. Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.



Rules of thumb

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Iterator can traverse a Composite. Visitor can apply an operation over a Composite. [GOF, p173]

Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass. [GOF, p271]

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GOF, p271]

Example

```
// Purpose.  Iterator design pattern

// Take traversal-of-a-collection functionality out of the collection
and
// promote it to "full object status".  This simplifies the collection,
allows
// many traversals to be active simultaneously, and decouples
collection algo-
// rithms from collection data structures.

// 1. Design an internal "iterator" class for the "collection" class
// 2. Add a createIterator() member to the collection class
// 3. Clients ask the collection object to create an iterator object
// 4. Clients use the first(), isDone(), next(), and currentItem()
protocol

import java.util.*;

class IntSet {
   private Hashtable ht = new Hashtable();

   // 1. Design an internal "iterator" class for the "collection" class
   public static class Iterator {
```

```
      private IntSet      set;
      private Enumeration e;
      private Integer     current;
      public Iterator( IntSet in ) { set = in; }
      public void    first()       { e = set.ht.keys();  next(); }
      public boolean isDone()      { return current == null; }
      public int     currentItem() { return current.intValue(); }
      public void    next()        {
         try { current = (Integer) e.nextElement(); }
         catch (NoSuchElementException e) { current = null; }
   }  }

   public void      add( int in )     { ht.put( new Integer( in ),
"null" ); }
   public boolean   isMember( int i ) { return ht.containsKey(new
Integer(i)); }
   public Hashtable getHashtable()    { return ht; }
   // 2. Add a createIterator() member to the collection class
   public Iterator  createIterator()  { return new Iterator( this ); }
}

class IteratorDemo {
   public static void main( String[] args ) {
      IntSet set = new IntSet();
      for (int i=2; i < 10; i += 2) set.add( i );
      for (int i=1; i < 9; i++)
         System.out.print( i + "-" + set.isMember( i ) + "  " );

      // 3. Clients ask the collection object to create many iterator
objects
      IntSet.Iterator it1 = set.createIterator();
      IntSet.Iterator it2 = set.createIterator();

      // 4. Clients use the first(), isDone(), next(), currentItem()
protocol
      System.out.print( "\nIterator:    " );
      for ( it1.first(), it2.first();  ! it1.isDone();  it1.next(),
it2.next() )
         System.out.print( it1.currentItem() + " " + it2.currentItem()
+ "  " );

      // Java uses a different collection traversal "idiom" called
Enumeration
      System.out.print( "\nEnumeration: " );
      for (Enumeration e = set.getHashtable().keys();
e.hasMoreElements(); )
         System.out.print( e.nextElement() + "  " );
      System.out.println();
}  }

// 1-false  2-true  3-false  4-true  5-false  6-true  7-false  8-true
// Iterator:    8 8  6 6  4 4  2 2
// Enumeration: 8  6  4  2
```

Core J2EE Patterns - Transfer Object Assembler

Context

In a Java 2 Platform, Enterprise Edition (J2EE) application, the server-side business components are implemented using session beans, entity beans, DAOs, and so forth. Application clients frequently need to access data that is composed from multiple objects.

Problem

Application clients typically require the data for the model or parts of the model to present to the user or to use for an intermediate processing step before providing some service. The application model is an abstraction of the business data and business logic implemented on the server side as business components. A model may be expressed as a collection of objects put together in a structured manner (tree or graph). In a J2EE application, the model is a distributed collection of objects such as session beans, entity beans, or DAOs and other objects. For a client to obtain the data for the model, such as to display to the user or to perform some processing, it must access individually each distributed object that defines the model. This approach has several drawbacks:

- Because the client must access each distributed component individually, there is a tight coupling between the client and the distributed components of the model over the network
- The client accesses the distributed components via the network layer, and this can lead to performance degradation if the model is complex with numerous distributed components. Network and client performance degradation occur when a number of distributed business components implement the application model and the client directly interacts with these components to obtain model data from that component. Each such access results in a remote method call that introduces network overhead and increases the chattiness between the client and the business tier.
- The client must reconstruct the model after obtaining the model's parts from the distributed components. The client therefore needs to have the necessary business logic to construct the model. If the model construction is complex and numerous objects are involved in its definition, then there may be an additional performance overhead on the client due to the construction process. In addition, the client must contain the business logic to manage the relationships between the components, which results in a more complex, larger client. When the client constructs the application model, the construction happens on the client side. Complex model construction can result in a significant performance overhead on the client side for clients with limited resources.
- Because the client is tightly coupled to the model, changes to the model require changes to the client. Furthermore, if there are different types of clients, it is more difficult to manage the changes across all client types. When there is tight coupling between the client and model implementation, which occurs when the client has direct knowledge of the model and manages the business component relationships, then changes to the model necessitate changes to the client. There is the further problem of code duplication for model access, which occurs when an application has many types of clients. This duplication makes client (code) management difficult

when the model changes.

Forces

- Separation of business logic is required between the client and the server-side components.
- Because the model consists of distributed components, access to each component is associated with a network overhead. It is desirable to minimize the number of remote method calls over the network.
- The client typically needs only to obtain the model to present it to the user. If the client must interact with multiple components to construct the model on the fly, the chattiness between the client and the application increases. Such chattiness may reduce the network performance.
- Even if the client wants to perform an update, it usually updates only certain parts of the model and not the entire model.
- Clients do not need to be aware of the intricacies and dependencies in the model implementation. It is desirable to have loose coupling between the clients and the business components that implement the application model.
- Clients do not otherwise need to have the additional business logic required to construct the model from various business components.

Solution

Use a Transfer Object Assembler to build the required model or submodel. The Transfer Object Assembler uses Transfer Objects to retrieve data from various business objects and other objects that define the model or part of the model.

The Transfer Object Assember constructs a composite Transfer Object that represents data from different business components. The Transfer Object caries the data for the model to the client in a single method call. Since the model data can be complex, it is recommended that this Transfer Object be immutable. That is, the client obtains such Transfer Objects with the sole purpose of using them for presentation and processing in a read-only manner. Clients are not allowed to make changes to the Transfer Objects.

When the client needs the model data, and if the model is represented by a single coarse-grained component (such as a Composite Entity), then the process of obtaining the model data is simple. The client simply requests the coarse-grained component for its composite Transfer Object. However, most real-world applications have a model composed of a combination of many coarse-grained and fine-grained components. In this case, the client must interact with numerous such business components to obtain all the data necessary to represent the model. The immediate drawbacks of this approach can be seen in that the clients become tightly coupled to the model implementation (model elements) and that the clients tend to make numerous remote method invocations to obtain the data from each individual component.

In some cases, a single coarse-grained component provides the model or parts of the model as a single Transfer Object (simple or composite). However, when multiple components represent the model, a single Transfer Object (simple or composite) may not

represent the entire model. To represent the model, it is necessary to obtain Transfer Objects from various components and assemble them into a new composite Transfer Object. The server, not the client, should perform such "on-the-fly" construction of the model.

Structure

Figure 8.27 shows the class diagram representing the relationships for the Transfer Object Assembler pattern.

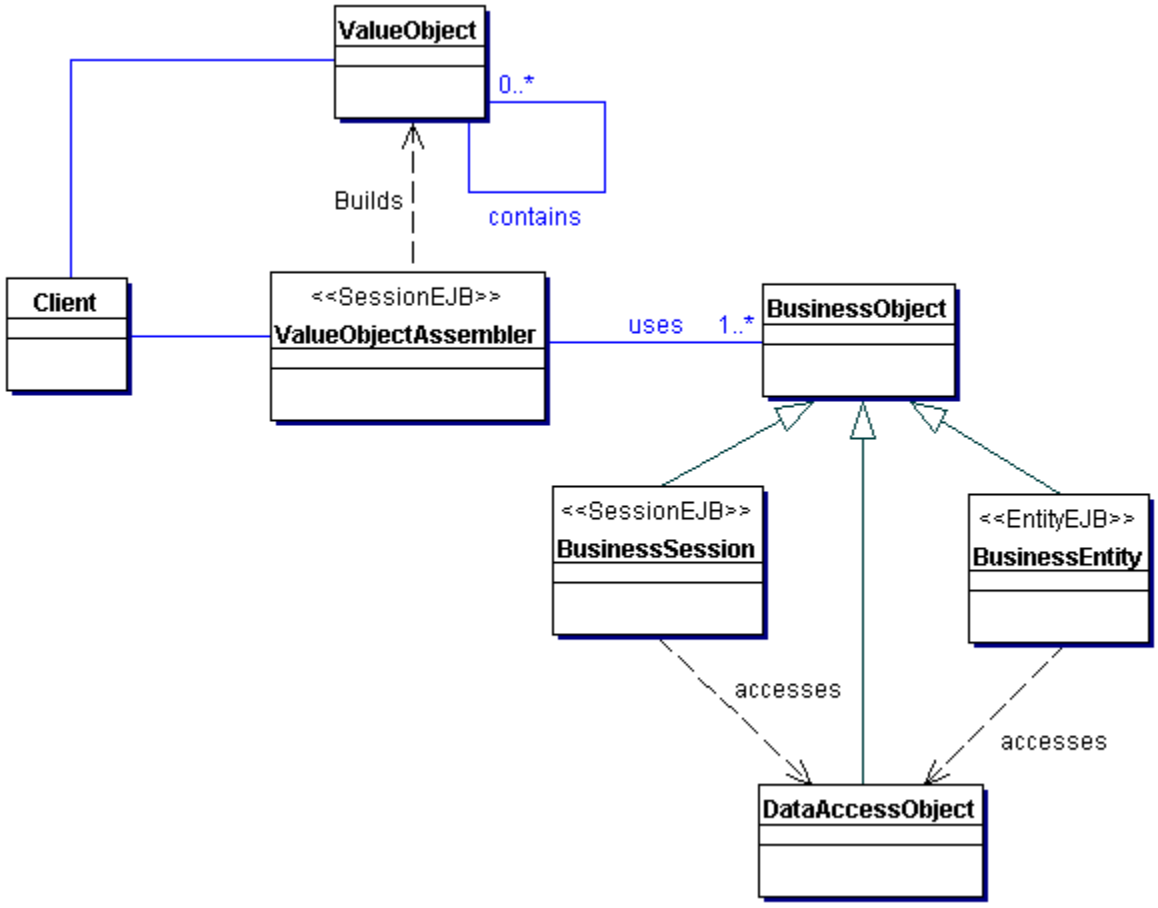


Figure 8.27 Transfer Object Assembler class diagram

Participants and Responsibilities

The sequence diagram in Figure 8.28 shows the interaction between the various participants in the Transfer Object Assembler pattern.

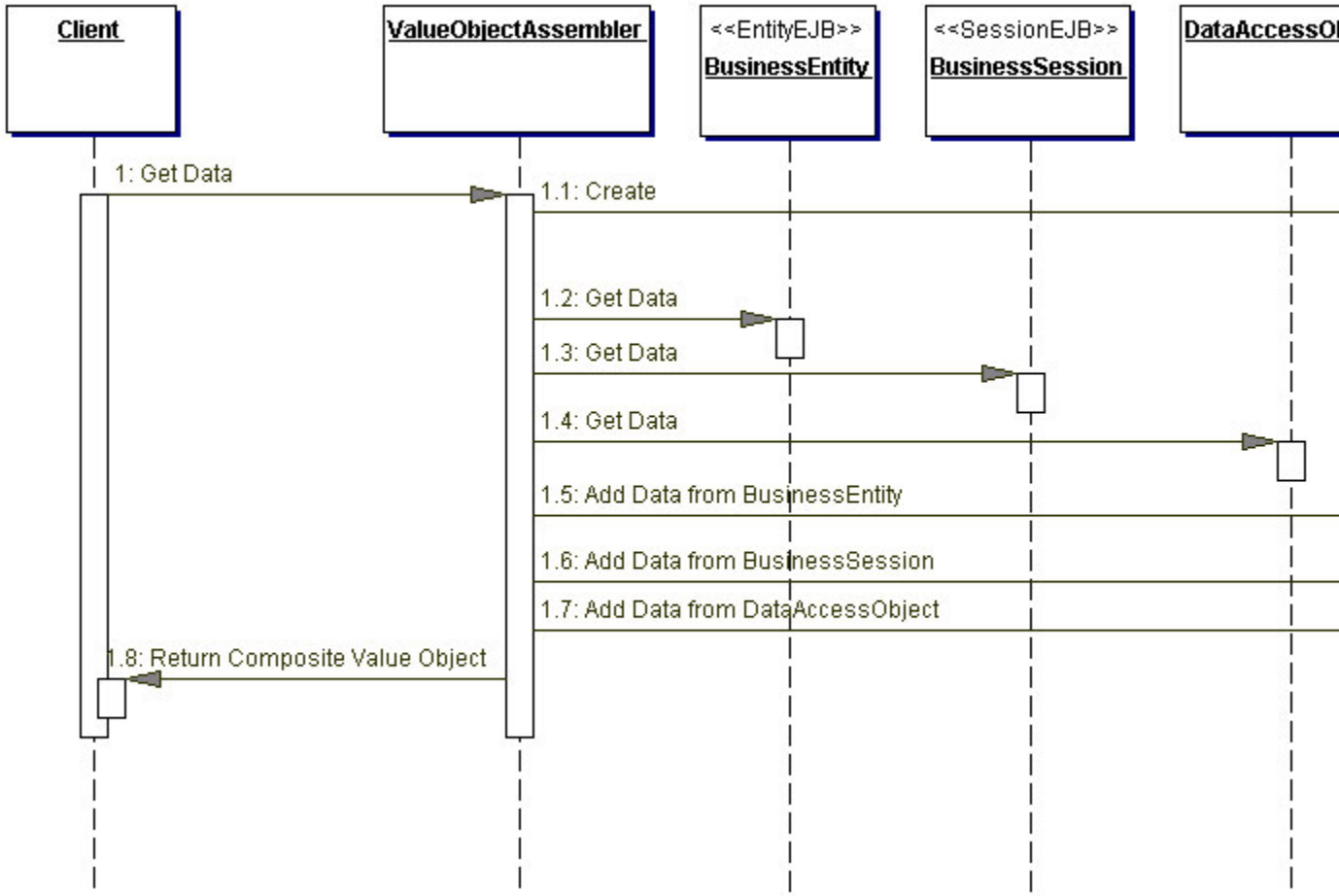


Figure 8.28 Transfer Object Assembler sequence diagram

TransferObjectAssembler

The TransferObjectAssembler is the main class of this pattern. The TransferObjectAssembler constructs a new Transfer Object based on the requirements of the application when the client requests a composite Transfer Object. The TransferObjectAssembler then locates the required BusinessObject instances to retrieve data to build the composite Transfer Object. BusinessObjects are business-tier components such as entity beans and session beans, DAOs, and so forth.

Client

If the TransferObjectAssembler is implemented as an arbitrary Java object, then the client is typically a Session Facade that provides the controller layer to the business tier. If the TransferObjectAssembler is implemented as a session bean, then the client can be a Session Facade or a Business Delegate.

BusinessObject

The BusinessObject participates in the construction of the new Transfer Object by providing the required data to the TransferObjectAssembler. Therefore, the BusinessObject is a role that can be fulfilled by a session bean, an entity bean, a DAO, or

a regular Java object.

TransferObject

The TransferObject is a composite Transfer Object that is constructed by the TransferObjectAssembler and returned to the client. This represents the complex data from various components that define the application model.

BusinessObject

BusinessObject is a role that can be fulfilled by a session bean, entity bean, or DAO. When the assembler needs to obtain data directly from the persistent storage to build the Transfer Object, it can use a DAO. This is shown as the DataAccessObject object in the diagrams.

Strategies

This section explains different strategies for implementing a Transfer Object Assembler pattern.

Java Object Strategy

The TransferObjectAssembler can be an arbitrary Java object and need not be an enterprise bean. In such implementations, a session bean usually fronts the TransferObjectAssembler. This session bean is typically a Session Facade that performs its other duties related to providing business services. The TransferObjectAssembler runs in the business tier, regardless of the implementation strategies. The motivation for this is to prevent the remote invocations from the TransferObjectAssembler to the source objects from crossing the tier.

Session Bean Strategy

This strategy implements the TransferObjectAssembler as a session bean (as shown in the class diagram). If a session bean implementation is preferred to provide the TransferObjectAssembler as a business service, it is typically implemented as a stateless session bean. The business components that make up the application model are constantly involved in transactions with various clients. As a result, when a TransferObjectAssembler constructs a new composite Transfer Object from various business components, it produces a snapshot of the model at the time of construction. The model could change immediately thereafter if another client changes one or more business components, effectively changing the business application model.

Therefore, implementing TransferObjectAssembler as a stateful session bean provides no benefits over implementing it as a stateless session bean, as preserving the state of the composite model data value when the underlying model is changing is futile. If the underlying model changes, it causes the Transfer Object held by the assembler to become stale. The TransferObjectAssembler, when next asked for the Transfer Object, either returns a stale state or reconstructs the Transfer Object to obtain the most recent snapshot. Therefore, it is recommended that the assembler be a stateless session bean to leverage the benefits of stateless over stateful session beans.

However, if the underlying model rarely changes, then the assembler may be a stateful

session bean and retain the newly constructed Transfer Object. In this case, the TransferObjectAssembler must include mechanisms to recognize changes to the underlying model and to reconstruct the model for the next client request.

Business Object Strategy

The BusinessObject role in this pattern can be supported by different types of objects, as explained below.

- The BusinessObject can be a session bean. The Transfer Object Assembler may use a Service Locator (see "Service Locator" on page 368) to locate the required session bean. The Transfer Object Assembler requests this session bean to provide the data to construct the composite Transfer Object.
- The BusinessObject can be an entity bean. The Transfer Object Assembler may use a Service Locator to locate the required entity bean. The Transfer Object Assembler requests this entity bean to provide the data to construct the composite Transfer Object.
- The BusinessObject can be a DAO. The Transfer Object Assembler requests this DAO to provide the data to construct the composite Transfer Object.
- The BusinessObject can be an arbitrary Java object. The Transfer Object Assembler requests this Java object to provide the data to construct the composite Transfer Object.
- The BusinessObject can be another Transfer Object Assembler. The first Transfer Object Assembler requests the second Transfer Object Assembler to provide the data to construct the composite Transfer Object.

Consequences

- **Separates Business Logic**
  When the client includes logic to manage the interactions with distributed components, it becomes difficult to clearly separate business logic from the client tier. The Transfer Object Assembler contains the business logic to maintain the object relationships and to construct the composite Transfer Object representing the model. The client needs no knowledge of how to construct the model or the different components that provide data to assemble the model.
- **Reduces Coupling Between Clients and the Application Model**
  The Transfer Object Assembler hides the complexity of the construction of model data from the clients and establishes a loose coupling between clients and the model. With loose coupling, if the model changes, then the Transfer Object Assembler requires a corresponding change. However, the client is not dependent on the model construction and interrelationships between model business components, so model changes do not directly affect the client. In general, loose coupling is preferred to tight coupling.
- **Improves Network Performance**
  The Transfer Object Assembler drastically reduces the network overhead of remote method calls and chattiness. The client can request the data for the application model from the Transfer Object Assembler in a single remote method call. The assembler

constructs and returns the composite Transfer Object for the model. However, the composite Transfer Object may contain a large amount of data. Thus, while use of the Transfer Object Assembler reduces the number of network calls, there is an increase in the amount of data transported in a single call. This trade-off should be considered in applying this pattern.

- **Improves Client Performance**
  The server-side Transfer Object Assembler constructs the model as a composite Transfer Object without using any client resources. The client spends no time assembling the model.
- **Improves Transaction Performance**
  Typically, updates are isolated to a very small part of the model and can be performed by fine-grained transactions. These transactions focus on isolated parts of the model instead of locking up the coarse-grained object (model). After the client obtains the model and displays or processes it locally, the user (or the client) may need to update or otherwise modify the model. The client can interact directly with a Session Facade to accomplish this at a suitable granularity level. The Transfer Object Assembler is not involved in the transaction to update or modify the model. There is better performance control because transactional work with the model happens at the appropriate level of granularity.
- **May Introduce Stale Transfer Objects**
  The Transfer Object Assembler constructs Transfer Objects on demand. These Transfer Objects are snapshots of the current state of the model, represented by various business components. Once the client obtains a Transfer Object from the assembler, that Transfer Object is entirely local to the client. Since the Transfer Objects are not network-aware, other changes made to the business components used to construct the Transfer Object are not reflected in the Transfer Objects. Therefore, after the Transfer Object is obtained, it can quickly become stale if there are transactions on the business components.

Sample Code

Implementing the Transfer Object Assembler

Consider a Project Management application where a number of business-tier components define the complex model. Suppose a client wants to obtain the model data composed of data from various business objects, such as:

- Project Information from the Project component
- Project Manager information from the ProjectManager component
- List of Project Tasks from the Project component
- Resource Information from the Resource component

A composite Transfer Object to contain this data can be defined as shown in Example 8.24. A Transfer Object Assembler pattern can be implemented to assemble this composite Transfer Object. The Transfer Object Assembler sample code is listed in Example 8.28.

Example 8.24 Composite Transfer Object Class

```
public class ProjectDetailsData {
  public ProjectTO projectData;
  public ProjectManagerTO projectManagerData;
  public Collection listOfTasks;
  ...
}
```

The list of tasks in the ProjectDetailsData is a collection of TaskResourceTO objects. The TaskResourceTO is a combination of TaskTO and ResourceTO. These classes are shown in Example 8.25, Example 8.26, and Example 8.27.

Example 8.25 TaskResourceTO Class

```
public class TaskResourceTO {
  public String projectId;
  public String taskId;
  public String name;
  public String description;
  public Date startDate;
  public Date endDate;
  public ResourceTO assignedResource;
  ...

  public TaskResourceTO(String projectId,
    String taskId, String name, String description,
    Date startDate, Date endDate, ResourceTO
    assignedResource) {
        this.projectId = projectId;
        this.taskId = taskId;
        ...
        this.assignedResource = assignedResource;
    }
    ...
}
```

Example 8.26 TaskTO Class

```
public class TaskTO {
  public String projectId;
  public String taskId;
  public String name;
  public String description;
  public Date startDate;
  public Date endDate;
  public assignedResourceId;

  public TaskTO(String projectId, String taskId,
      String name, String description, Date startDate,
      Date endDate, String assignedResourceId) {
        this.projectId = projectId;
```

```
        this.taskId = taskId;
        ...
        this.assignedResource = assignedResource;
    }
    ...
}
```

Example 8.27 ResourceTO Class

```
public class ResourceTO {
  public String resourceId;
  public String resourceName;
  public String resourceEmail;
  ...

  public ResourceTO (String resourceId, String
    resourceName, String resourceEmail, ...) {
      this.resourceId = resourceId;
      this.resourceName = resourceName;
      this.resourceEmail = resourceEmail;
      ...
  }
}
```

The ProjectDetailsAssembler class that assembles the ProjectDetailsData object is listed in Example 8.28.

Example 8.28 Implementing the Transfer Object Assembler

```
public class ProjectDetailsAssembler
  implements javax.ejb.SessionBean {

  ...

  public ProjectDetailsData getData(String projectId){

    // Construct the composite Transfer Object
    ProjectDetailsData pData = new
                      ProjectDetailsData();

    //get the project details;
    ProjectHome projectHome =
        ServiceLocator.getInstance().getHome(
          "Project", ProjectEntityHome.class);
    ProjectEntity project =
      projectHome.findByPrimaryKey(projectId);
    ProjectTO projTO = project.getData();

    // Add Project Info to ProjectDetailsData
    pData.projectData = projTO;

    //get the project manager details;
    ProjectManagerHome projectManagerHome =
```

```
      ServiceLocator.getInstance().getHome(
        "ProjectManager", ProjectEntityHome.class);

    ProjectManagerEntity projectManager =
      projectManagerHome.findByPrimaryKey(
        projTO.managerId);

    ProjectManagerTO projMgrTO =
      projectManager.getData();

    // Add ProjectManager info to ProjectDetailsData
    pData.projectManagerData = projMgrTO;

    // Get list of TaskTOs from the Project
    Collection projTaskList = project.getTasksList();

    // construct a list of TaskResourceTOs
    ArrayList listOfTasks = new ArrayList();

    Iterator taskIter = projTaskList.iterator();
    while (taskIter.hasNext()) {
      TaskTO task = (TaskTO) taskIter.next();

      //get the Resource details;
      ResourceHome resourceHome =
      ServiceLocator.getInstance().getHome(
        "Resource",         ResourceEntityHome.class);

      ResourceEntity resource =
        resourceHome.findByPrimaryKey(
          task.assignedResourceId);

      ResourceTO resTO = resource.getResourceData();

      // construct a new TaskResourceTO using Task
      // and Resource data
      TaskResourceTO trTO = new TaskResourceTO(
              task.projectId, task.taskId,
              task.name, task.description,
              task.startDate, task.endDate,
              resTO);

      // add TaskResourceTO to the list
      listOfTasks.add(trTO);
    }

    // add list of tasks to ProjectDetailsData
    pData.listOfTasks = listOfTasks;

    // add any other data to the Transfer Object
    ...

    // return the composite Transfer Object
    return pData;

  }
```

```
    . . .
}
```

Related Patterns

- **Transfer Object**
  The Transfer Object Assembler uses the Transfer Object pattern in order to create and transport Transfer Objects to the client. The Transfer Objects created carry the data representing the application model from the business tier to the clients requesting the data.
- **Composite Entity**
  The Composite Entity pattern promotes a coarse-grained entity bean design, where entities can produce composite Transfer Objects similar to the one produced by the Transfer Object Assembler. However, the Transfer Object Assembler is more applicable when the composite Transfer Object constructed is derived from a number of components (session beans, entity beans, DAOs, and so forth), whereas the Composite Entity pattern constructs the Transfer Object from its own data (that is, a single entity bean).
- **Session Facade**
  The Transfer Object Assembler is typically implemented as a stateless session bean. As such, it could be viewed as a limited special application of the Session Facade pattern. More importantly, Transfer Object Assembler constructs composite Transfer Objects that are immutable. Therefore, the client receiving this composite Transfer Object can only use the data for its presentation and processing purposes. The client cannot update the Transfer Object. If the client needs to update the business objects that derive the composite Transfer Object, it may have to access the Session Facade (session bean) that provides that business service.
- **Data Access Object**
  A possible strategy for the Transfer Object Assembler involves obtaining data for the composite Transfer Object from the persistent store without enterprise bean involvement. The Data Access Object pattern can be applied, thus leveraging its benefits to provide persistent storage access to the Transfer Object Assembler.
- **Service Locator**
  The Transfer Object Assembler needs to locate and use various business objects. The Service Locator pattern can be used in conjunction with the Transfer Object Assembler pattern whenever a business object or a service needs to be located.

Core J2EE Pattern Catalog
Core J2EE Patterns - Service Locator

Context

Service lookup and creation involves complex interfaces and network operations.

Problem

Service (JMS) components, which provide business services and persistence capabilities. To interact with these components, clients must either locate the service component (referred to as a lookup operation) or create a new component. For instance, an EJB client must locate the enterprise bean's home object, which the client then uses either to find an object or to create or remove one or more enterprise beans. Similarly, a JMS client must first locate the JMS Connection Factory to obtain a JMS Connection or a JMS Session.

All Java 2 Platform, Enterprise Edition (J2EE) application clients use the JNDI common facility to look up and create EJB and JMS components. The JNDI API enables clients to obtain an initial context object that holds the component name to object bindings. The client begins by obtaining the initial context for a bean's home object. The initial context remains valid while the client session is valid. The client provides the JNDI registered name for the required object to obtain a reference to an administered object. In the context of an EJB application, a typical administered object is an enterprise bean's home object. For JMS applications, the administered object can be a JMS Connection Factory (for a Topic or a Queue) or a JMS Destination (a Topic or a Queue).

So, locating a JNDI-administered service object is common to all clients that need to access that service object. That being the case, it is easy to see that many types of clients repeatedly use the JNDI service, and the JNDI code appears multiple times across these clients. This results in an unnecessary duplication of code in the clients that need to look up services.

Also, creating a JNDI initial context object and performing a lookup on an EJB home object utilizes significant resources. If multiple clients repeatedly require the same bean home object, such duplicate effort can negatively impact application performance.

Let us examine the lookup and creation process for various J2EE components.

1. The lookup and creation of enterprise beans relies upon the following:
   - A correct setup of the JNDI environment so that it connects to the naming and directory service used by the application. Setup entails providing the location of the naming service and the necessary authentication credentials to access that service.
   - The JNDI service can then provide the client with an initial context that acts as a placeholder for the component name-to-object bindings. The client requests this initial context to look up the EJBHome object for the required enterprise bean by providing the JNDI name for that EJBHome object.
   - Find the EJBHome object using the initial context's lookup mechanism.
   - After obtaining the EJBHome object, create, remove, or find the enterprise bean, using the EJBHome object's create, move, and find (for entity beans only).
2. The lookup and creation of JMS components (Topic, Queue, QueueConnection, QueueSession, TopicConnection, TopicSession, and so forth) involves the following steps. Note that in these steps, Topic refers to the publish/subscribe messaging model and Queue refers to the point-to-point messaging model.
   - Set up the JNDI environment to the naming service used by the application. Setup entails providing the location of the naming service and the necessary authentication credentials to access that service.
   - Obtain the initial context for the JMS service provider from the JNDI naming service.
   - Use the initial context to obtain a Topic or a Queue by supplying the JNDI name for the topic or the queue. Topic and Queue are JMSDestination objects.
   - Use the initial context to obtain a TopicConnectionFactory or a QueueConnectionFactory by supplying the JNDI name for the topic or queue connection factory.
   - Use the TopicConnectionFactory to obtain a TopicConnection or QueueConnectionFactory to obtain a QueueConnection.
   - Use the TopicConnection to obtain a TopicSession or a QueueConnection to obtain a QueueSession.
   - Use the TopicSession to obtain a TopicSubscriber or a TopicPublisher for the required

Queue.

The process to look up and create components involves a vendor-supplied context factory implementation. This introduces vendor dependency in the application clients that need to use the JNDI lookup facility to locate the enterprise beans and JMS components, such as topics, queues, and connection factory objects.

Forces

- EJB clients need to use the JNDI API to look up EJBHome objects by using the enterprise bean's registered JNDI name.
- JMS clients need to use JNDI API to look up JMS components by using the JNDI names registered for JMS components, such as connection factories, queues, and topics.
- The context factory to use for the initial JNDI context creation is provided by the service provider vendor and is therefore vendor- dependent. The context factory is also dependent on the type of object being looked up. The context for JMS is different from the context for EJB, with different providers.
- Lookup and creation of service components could be complex and may be used repeatedly in multiple clients in the application.
- Initial context creation and service object lookups, if frequently required, can be resource-intensive and may impact application performance. This is especially true if the clients and the services are located in different tiers.
- EJB clients may need to reestablish connection to a previously accessed enterprise bean instance, having only its Handle object.

Solution

Use a Service Locator object to abstract all JNDI usage and to hide the complexities of initial context creation, EJB home object lookup, and EJB object re-creation. Multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control, and improve performance by providing a caching facility.

This pattern reduces the client complexity that results from the client's dependency on and need to perform lookup and creation processes, which are resource-intensive. To eliminate these problems, this pattern provides a mechanism to abstract all dependencies and network details into the Service Locator.

Structure

Figure 8.31 shows the class diagram representing the relationships for the Service Locator pattern.

Figure 8.31 Service Locator class diagram

Participants and Responsibilities

Figure 8.32 contains the sequence diagram that shows the interaction between the various participants of the Service Locator pattern.

Figure 8.32 Service Locator Sequence diagram

Client

This is the client of the Service Locator. The client is an object that typically requires access to business objects such as a Business Delegate (see "Business Delegate" on page 248).

Service Locator

The Service Locator abstracts the API lookup (naming) services, vendor dependencies, lookup complexities, and business object creation, and provides a simple interface to clients. This reduces the client's complexity. In addition, the same client or other clients can reuse the Service Locator.

InitialContext

The InitialContext object is the start point in the lookup and creation process. Service providers provide the context object, which varies depending on the type of business object provided by the Service Locator's lookup and creation service. A Service Locator that provides services for multiple types of business objects (such as enterprise beans, JMS components, and so forth) utilizes multiple types of context objects, each obtained from a different provider (e.g., context provider for an EJB application server may be different from the context provider for JMS service).

ServiceFactory

The ServiceFactory object represents an object that provides life cycle management for the BusinessService objects. The ServiceFactory object for enterprise beans is an EJBHome object. The ServiceFactory for JMS components can be a JMS ConnectionFactory object, such as a TopicConnectionFactory (for publish/subscribe messaging model) or a QueueConnectionFactory (for point-to-point messaging model).

BusinessService

The BusinessService is a role that is fulfilled by the service the client is seeking to access. The BusinessService object is created or looked up or removed by the ServiceFactory. The BusinessService object in the context of an EJB application is an enterprise bean. The BusinessService object in the context of a JMS application can be a TopicConnection or a QueueConnection. The TopicConnection and QueueConnection can then be used to produce a JMSSession object, such as TopicSession or a QueueSession respectively.

Strategies

EJB Service Locator Strategy

The Service Locator for enterprise bean components uses EJBHome object, shown as BusinessHome in the role of the ServiceFactory. Once the EJBHome object is obtained, it can be cached in the ServiceLocator for future use to avoid another JNDI lookup when the client needs the home object again. Depending on the implementation, the home object can be returned to the client, which can then use it to look up, create, and remove enterprise beans. Otherwise, the ServiceLocator can retain (cache) the home object and gain the additional responsibility of proxying all client calls to the home object. The class diagram for the EJB Service Locator strategy is shown in Figure 8.33.

Figure 8.33 EJB Service Locator Strategy class diagram

The interaction between the participants in a Service Locator for an enterprise bean is shown in Figure 8.34.

Figure 8.34 EJB Service Locator Strategy sequence diagram

JMS Queue Service Locator Strategy

This strategy is applicable to point-to-point messaging requirements. The Service Locator for JMS components uses QueueConnectionFactory objects in the role of the ServiceFactory. The QueueConnectionFactory is looked up using its JNDI name. The QueueConnectionFactory can be cached by the ServiceLocator for future use. This avoids repeated JNDI calls to look it up when the client needs it

again. The ServiceLocator may otherwise hand over the QueueConnectionFactory to the client. The Client can then use it to create a QueueConnection. A QueueConnection is necessary in order to obtain a QueueSession or to create a Message, a QueueSender (to send messages to the queue), or a QueueReceiver (to receive messages from a queue). The class diagram for the JMS Queue Service Locator strategy is shown in Figure 8.35. In this diagram, the Queue is a JMS Destination object registered as a JNDI-administered object representing the queue. The Queue object can be directly obtained from the context by looking it up using its JNDI name.

Figure 8.35 JMS Queue Service Locator strategy class diagram

The interaction between the participants in a Service Locator for point-to-point messaging using JMS Queues is shown in Figure 8.36.

Figure 8.36 JMS Queue Service Locator Strategy sequence diagram

JMS Topic Service Locator Strategy

This strategy is applicable to publish/subscribe messaging requirements. The Service Locator for JMS components uses TopicConnectionFactory objects in the role of the ServiceFactory. The TopicConnectionFactory is looked up using its JNDI name. The TopicConnectionFactory can be cached by the ServiceLocator for future use. This avoids repeated JNDI calls to look it up when the client needs it again. The ServiceLocator may otherwise hand over the TopicConnectionFactory to the client. The Client can then use it to create a TopicConnection. A TopicConnection is necessary in order to obtain a TopicSession or to create a Message, a TopicPublisher (to publish messages to a topic), or a TopicSubscriber (to subscribe to a topic). The class diagram for the JMS Topic Service Locator strategy is shown in Figure 8.37. In this diagram, the Topic is a JMS Destination object registered as a JNDI-administered object representing the topic. The Topic object can be directly obtained from the context by looking it up using its JNDI name.

Figure 8.37 JMS Topic Service Locator strategy

The interaction between the participants in a Service Locator for publish/subscribe messaging using JMS Topics is shown in Figure 8.38.

Figure 8.38 JMS Topic Service Locator Strategy sequence diagram

Combined EJB and JMS Service Locator Strategy

These strategies for EJB and JMS can be used to provide separate Service Locator implementations, since the clients for EJB and JMS may more likely be mutually exclusive. However, if there is a need to combine these strategies, it is possible to do so to provide the Service Locator for all objects-enterprise beans and JMS components.

Type Checked Service Locator Strategy

The diagrams in Figures 8.37 and 8.38 provide lookup facilities by passing in the service lookup name. For an enterprise bean lookup, the Service Locator needs a class as a parameter to the PortableRemoteObject.narrow() method. The Service Locator can provide a getHome() method, which accepts as arguments the JNDI service name and the EJBHome class object for the enterprise bean. Using this method of passing in JNDI service names and EJBHome class objects can lead to client errors. Another approach is to statically define the services in the ServiceLocator, and instead of passing in string names, the client passes in a constant. Example 8.34 illustrates such a strategy.

This strategy has trade-offs. It reduces the flexibility of lookup, which is in the Services Property Locator strategy, but add the type checking of passing in a constant to the ServiceLocator.getHome() method.

Service Locator Properties Strategy

This strategy helps to address the trade-offs of the type checking strategy. This strategy suggests the use of property files and/or deployment descriptors to specify the JNDI names and the EJBHome class name. For presentation-tier clients, such properties can be specified in the presentation-tier deployment descriptors or property files. When the presentation tier accesses the business tier, it typically uses the Business Delegate pattern.

The Business Delegate interacts with the Service Locator to locate business components. If the presentation tier loads the properties on initialization and can provide a service to hand out the JNDI names and the EJB class names for the required enterprise bean, then the Business Delegate could request this service to obtain them. Once the Business Delegate has the JNDI name and the EJBHome Class name, it can request the Service Locator for the EJBHome by passing these properties as arguments.

The Service Locator can in turn use Class.forName(EJBHome ClassName) to obtain the EJBHome Class object and go about its business of looking up the EJBHome and using the Portable RemoteObject.narrow() method to cast the object, as shown by the getHome() method in the ServiceLocator sample code in Example 8.33. The only thing that changes is where the JNDI name and the Class objects are coming from. Thus, this strategy avoids hardcoded JNDI names in the code and provides for flexibility of deployment. However, due to the lack of type checking, there is scope for avoiding errors and mismatches in specifying the JNDI names in different deployment descriptors.

Consequences

- **Abstracts Complexity**
  The Service Locator pattern encapsulates the complexity of this lookup and creation process (described in the problem) and keeps it hidden from the client. The client does not need to deal with the lookup of component factory objects (EJBHome, QueueConnectionFactory, and TopicConnectionFactory, among others) because the ServiceLocator is delegated that responsibility.
- **Provides Uniform Service Access to Clients**
  The Service Locator pattern abstracts all the complexities, as explained previously. In doing so, it provides a very useful and precise interface that all clients can use. The pattern interface ensures that all types of clients in the application uniformly access business objects, in terms of lookup and creation. This uniformity reduces development and maintenance overhead.
- **Facilitates Adding New Business Components**
  Because clients of enterprise beans are not aware of the EJBHome objects, it's possible to add new EJBHome objects for enterprise beans developed and deployed at a later time without impacting the clients. JMS clients are not directly aware of the JMS connection factories, so new connection factories can be added without impacting the clients.
- **Improves Network Performance**
  The clients are not involved in JNDI lookup and factory/home object creation. Because the Service Locator performs this work, it can aggregate the network calls required to look up and create business objects.
- **Improves Client Performance by Caching**
  The Service Locator can cache the initial context objects and references to the factory objects (EJBHome, JMS connection factories) to eliminate unnecessary JNDI activity that occurs when obtaining the initial context and the other objects. This improves the application performance.

Sample Code

Implementing Service Locator Pattern

A sample implementation of the Service Locator pattern is shown in Example 8.33. An example for implementing the Type Checked Service Locator strategy is listed in Example 8.34.

Example 8.33 Implementing Service Locator

```
package corepatterns.apps.psa.util;
import java.util.*;
```

```
import javax.naming.*;
import java.rmi.RemoteException;
import javax.ejb.*;
import javax.rmi.PortableRemoteObject;
import java.io.*;

public class ServiceLocator {
  private static ServiceLocator me;
  InitialContext context = null;

  private ServiceLocator()
  throws ServiceLocatorException {
    try {
      context = new InitialContext();
    } catch(NamingException ne) {
      throw new ServiceLocatorException(...);
    }
  }

  // Returns the instance of ServiceLocator class
  public static ServiceLocator getInstance()
  throws ServiceLocatorException {
    if (me == null) {
      me = new ServiceLocator();
    }
    return me;
  }

  // Converts the serialized string into EJBHandle
  // then to EJBObject.
  public EJBObject getService(String id)
  throws ServiceLocatorException {
    if (id == null) {
      throw new ServiceLocatorException(...);
    }
    try {
      byte[] bytes = new String(id).getBytes();
      InputStream io = new
        ByteArrayInputStream(bytes);
      ObjectInputStream os = new
        ObjectInputStream(io);
      javax.ejb.Handle handle =
        (javax.ejb.Handle)os.readObject();
      return handle.getEJBObject();
    } catch(Exception ex) {
      throw new ServiceLocatorException(...);
    }
  }

  // Returns the String that represents the given
  // EJBObject's handle in serialized format.
  protected String getId(EJBObject session)
  throws ServiceLocatorException {
    try {
      javax.ejb.Handle handle = session.getHandle();
      ByteArrayOutputStream fo = new
        ByteArrayOutputStream();
      ObjectOutputStream so = new
```

```
        ObjectOutputStream(fo);
      so.writeObject(handle);
      so.flush();
      so.close();
      return new String(fo.toByteArray());
    } catch(RemoteException ex) {
      throw new ServiceLocatorException(...);
    } catch(IOException ex) {
      throw new ServiceLocatorException(...);
    }
    return null;
  }

  // Returns the EJBHome object for requested service
  // name. Throws ServiceLocatorException If Any Error
  // occurs in lookup
  public EJBHome getHome(String name, Class clazz)
  throws ServiceLocatorException {
    try {
      Object objref = context.lookup(name);
      EJBHome home = (EJBHome)
        PortableRemoteObject.narrow(objref, clazz);
      return home;
    } catch(NamingException ex) {
      throw new ServiceLocatorException(...);
    }
  }
}
```

Implementing Type Checked Service Locator Strategy

Example 8.34 Implementing Type Checked Service Locator Strategy

```
package corepatterns.apps.psa.util;
// imports
...
public class ServiceLocator {
  // singleton's private instance
  private static ServiceLocator me;

  static {
    me = new ServiceLocator();
  }

  private ServiceLocator() {}

  // returns the Service Locator instance
  static public ServiceLocator getInstance() {
    return me;
  }


  // Services Constants Inner Class - service objects
  public class Services {
    final public static int PROJECT  = 0;
    final public static int RESOURCE = 1;
```

```
}

// Project EJB related constants
final static Class  PROJECT_CLASS = ProjectHome.class;
final static String PROJECT_NAME  = "Project";

// Resource EJB related constants

final static Class  RESOURCE_CLASS = ResourceHome.class;
final static String RESOURCE_NAME  = "Resource";

// Returns the Class for the required service
static private Class getServiceClass(int service){
  switch( service ) {
    case Services.PROJECT:
      return PROJECT_CLASS;
    case Services.RESOURCE:
    return RESOURCE_CLASS;
  }
  return null;
}

// returns the JNDI name for the required service
static private String getServiceName(int service){
  switch( service ) {
    case Services.PROJECT:
      return PROJECT_NAME;
    case Services.RESOURCE:
      return RESOURCE_NAME;
  }
  return null;
}

/* gets the EJBHome for the given service using the
** JNDI name and the Class for the EJBHome
*/
public EJBHome getHome( int s )
  throws ServiceLocatorException {
  EJBHome home = null;
  try {
      Context initial  = new InitialContext();

    // Look up using the service name from
    // defined constant
    Object objref =
      initial.lookup(getServiceName(s));

    // Narrow using the EJBHome Class from
    // defined constant
    Object obj = PortableRemoteObject.narrow(
            objref, getServiceClass(s));
    home = (EJBHome)obj;
  }
  catch( NamingException ex ) {
      throw new ServiceLocatorException(...);
  }
  catch( Exception ex ) {
      throw new ServiceLocatorException(...);
```

```
    }
    return home;
  }
}
```

The client code to use the Service Locator for this strategy may look like the code in Example 8.35.

Example 8.35 Client Code for Using the Service Locator

```
public class ServiceLocatorTester {
  public static void main( String[] args ) {
    ServiceLocator serviceLocator =
      ServiceLocator.getInstance();
    try {
      ProjectHome projectHome = (ProjectHome)
        serviceLocator.getHome(
          ServiceLocator.Services.PROJECT );
    }
    catch( ServiceException ex ) {
      // client handles exception
      System.out.println( ex.getMessage( ));
    }
  }
}
```

This strategy is about applying type checking to client lookup. It encapsulates the static service values inside the ServiceLocator and creates an inner class Services, which declares the service constants (PROJECT and RESOURCE). The Tester client gets an instance to the ServiceLocator singleton and calls getHome(), passing in the PROJECT. ServiceLocator in turn gets the JNDI entry name and the Home class and returns the EJBHome.

Related Patterns

- **Business Delegate**
  The Business Delegate pattern uses Service Locator to gain access to the business service objects such as EJB objects, JMS topics, and JMS queues. This separates the complexity of service location from the Business Delegate, leading to loose coupling and increased manageability.
- **Session Facade**
  The Session Facade pattern uses Service Locator to gain access to the enterprise beans that are involved in a workflow. The Session Facade could directly use the Service Locator or delegate the work to a Business Delegate (See "Business Delegate" on page 248.).
- **Transfer Object Assembler**
  The Transfer Object Assembler pattern uses Service Locator to gain access to the various enterprise beans it needs to access to build its composite Transfer Object. The Transfer Object Assembler could directly use the Service Locator or delegate the work to a Business Delegate (See "Business Delegate" on page 248.).

Core J2EE Pattern Catalog
Core J2EE Patterns - Business Delegate

Context

A multi-tiered, distributed system requires remote method invocations to send and receive data across tiers. Clients are exposed to complexity of dealing with distributed components.

Problem

Presentation-tier components interact directly with business services. This direct interaction exposes the underlying implementation details of the business service application program interface (API) to the presentation tier. As a result, the presentation-tier compon are vulnerable to changes in the implementation of the business services: When the implementation of the business services chan the exposed implementation code in the presentation tier must change too.

Additionally, there may be a detrimental impact on network performance because presentation-tier components that use the busine service API make too many invocations over the network. This happens when presentation-tier components use the underlying AP directly, with no client-side caching mechanism or aggregating service.

Lastly, exposing the service APIs directly to the client forces the client to deal with the networking issues associated with the distrib nature of Enterprise JavaBeans (EJB) technology.

Forces

- Presentation-tier clients need access to business services.
- Different clients, such as devices, Web clients, and thick clients, need access to business service.
- Business services APIs may change as business requirements evolve.
- It is desirable to minimize coupling between presentation-tier clients and the business service, thus hiding the underlying implementation details of the service, such as lookup and access.
- Clients may need to implement caching mechanisms for business service information.
- It is desirable to reduce network traffic between client and business services.

Solution

Use a Business Delegate to reduce coupling between presentation-tier clients and business services. The Business Deleg hides the underlying implementation details of the business service, such as lookup and access details of the EJB architecture.

The Business Delegate acts as a client-side business abstraction; it provides an abstraction for, and thus hides, the implementation the business services. Using a Business Delegate reduces the coupling between presentation-tier clients and the system's busines services. Depending on the implementation strategy, the Business Delegate may shield clients from possible volatility in the implementation of the business service API. Potentially, this reduces the number of changes that must be made to the presentation client code when the business service API or its underlying implementation changes.

However, interface methods in the Business Delegate may still require modification if the underlying business service API changes Admittedly, though, it is more likely that changes will be made to the business service rather than to the Business Delegate.

Often, developers are skeptical when a design goal such as abstracting the business layer causes additional upfront work in return future gains. However, using this pattern or its strategies results in only a small amount of additional upfront work and provides considerable benefits. The main benefit is hiding the details of the underlying service. For example, the client can become transpar to naming and lookup services. The Business Delegate also handles the exceptions from the business services, such as java.rmi.Remote exceptions, Java Messages Service (JMS) exceptions and so on. The Business Delegate may intercept such serv level exceptions and generate application level exceptions instead. Application level exceptions are easier to handle by the clients, may be user friendly. The Business Delegate may also transparently perform any retry or recovery operations necessary in the eve a service failure without exposing the client to the problem until it is determined that the problem is not resolvable. These gains pre a compelling reason to use the pattern.

Another benefit is that the delegate may cache results and references to remote business services. Caching can significantly impro performance, because it limits unnecessary and potentially costly round trips over the network.

implementation details of the business service lookup code. The Lookup Service may be written as part of the Delegate, but we recommend that it be implemented as a separate component, as outlined in the Service Locator pattern (See "Service Locator" on p 368.)

When the Business Delegate is used with a Session Facade, typically there is a one-to-one relationship between the two. This one one relationship exists because logic that might have been encapsulated in a Business Delegate relating to its interaction with mult business services (creating a one-to-many relationship) will often be factored back into a Session Facade.

Finally, it should be noted that this pattern could be used to reduce coupling between other tiers, not simply the presentation and th business tiers.

Structure

Figure 8.1 shows the class diagram representing the Business Delegate pattern. The client requests the BusinessDelegate to prov access to the underlying business service. The BusinessDelegate uses a LookupService to locate the required BusinessService component.

Figure 8.1 BusinessDelegate class diagram

Participants and Responsibilities

Figure 8.2 and Figure 8.3 show sequence diagrams that illustrate typical interactions for the Business Delegate pattern.

Figure 8.2 BusinessDelegate sequence diagram

The BusinessDelegate uses a LookupService for locating the business service. The business service is used to invoke the busines
methods on behalf of the client. The Get ID method shows that the BusinessDelegate can obtain a String version of the handle (su
as EJBHandle object) for the business service and return it to the client as a String. The client can use the String version of the har
at a later time to reconnect to the business service it was using when it obtained the handle. This technique will avoid new lookups
since the handle is capable of reconnecting to its business service instance. It should be noted that handle objects are implemente
the container provider and may not be portable across containers from different vendors.

The sequence diagram in Figure 8.3 shows obtaining a BusinessService (such as a session or an entity bean) using its handle.

Figure 8.3 BusinessDelegate with ID sequence diagram

BusinessDelegate

The BusinessDelegate's role is to provide control and protection for the business service. The BusinessDelegate can expose two types of constructors to clients. One type of request instantiates the BusinessDelegate without an ID, while the other instantiates it with an ID, where ID is a String version of the reference to a remote object, such as EJBHome or EJBObject.

When initialized without an ID, the BusinessDelegate requests the service from the Lookup Service, typically implemented as a Service Locator (see "Service Locator" on page 368), which returns the Service Factory, such as EJBHome. The BusinessDelegate requests that the Service Factory locate, create, or remove a BusinessService, such as an enterprise bean.

When initialized with an ID string, the BusinessDelegate uses the ID string to reconnect to the BusinessService. Thus, the BusinessDelegate shields the client from the underlying implementation details of BusinessService naming and lookup. Furthermore, the presentation-tier client never directly makes a remote invocation on a BusinessSession; instead, the client uses the BusinessDelegate.

LookupService

The BusinessDelegate uses the LookupService to locate the BusinessService. The LookupService encapsulates the implementation details of BusinessService lookup.

BusinessService
The BusinessService is a business-tier component, such as an enterprise bean or a JMS component, that provides the required service to the client.
Strategies
Delegate Proxy Strategy
The Business Delegate exposes an interface that provides clients access to the underlying methods of the business service API. In this strategy, a Business Delegate provides proxy function to pass the client methods to the session bean it is encapsulating. The Business Delegate may additionally cache any necessary data, including the remote references to the session bean's home or remote objects to improve performance by reducing the number of lookups. The Business Delegate may also convert such references to String versions (IDs) and vice versa, using the services of a Service Locator.
The example implementation for this strategy is discussed in the "Sample Code" section of this pattern.
Delegate Adapter Strategy
The Business Delegate proves to be a nice fit in a B2B environment when communicating with Java 2 Platform, Enterprise Edition (J2EE) based services. Disparate systems may use an XML as the integration language. Integrating one system to another typically requires an Adapter [GoF] to meld the two disparate systems. Figure 8.4 gives an example.

Figure 8.4 Using the Business Delegate pattern with an Adapter strategy
Consequences
Reduces Coupling, Improves Manageability
The Business Delegate reduces coupling between the presentation tier and the business tier by hiding all business-tier implementation details. It is easier to manage changes because they are centralized in one place, the Business Delegate.
Translates Business Service Exceptions
The Business Delegate is responsible for translating any network or infrastructure-related exceptions into

business exceptions, shielding clients from knowledge of the underlying implementation specifics.

Implements Failure Recovery and Thread Synchronization

The Business Delegate on encountering a business service failure, may implement automatic recovery features without exposing the problem to the client. If the recovery succeeds, the client need not know about the failure. If the recovery attempt does not succeed, then the Business Delegate needs to inform the client of the failure. Additionally, the business delegate methods may be synchronized, if necessary.

Exposes Simpler, Uniform Interface to Business Tier

The Business Delegate, to better serve its clients, may provide a variant of the interface provided by the underlying enterprise beans.

Impacts Performance

The Business Delegate may provide caching services (and better performance) to the presentation tier for common service requests.

Introduces Additional Layer

The Business Delegate may be seen as adding an unnecessary layer between the client and the service, thus introducing added complexity and decreasing flexibility. Some developers may feel that it is an extra effort to develop Business Delegates with implementations that use the Delegate Proxy strategy. At the same time, the benefits of the pattern typically outweigh such drawbacks.

Hides Remoteness

While location transparency is one of the benefits of this pattern, a different problem may arise due to the developer treating a remote service as if it was a local one. This may happen if the client developer does not understand that the Business Delegate is a client side proxy to a remote service. Typically, a method invocations on the Business Delegate results in a remote method invocation under the wraps. Ignoring this, the developer may tend to make numerous method invocations to perform a single task, thus increasing the network traffic.

Sample Code

Implementing the Business Delegate Pattern

Consider a Professional Services Application (PSA), where a Web-tier client needs to access a session bean that implements the Session Facade pattern. The Business Delegate pattern can be applied to design a Delegate class ResourceDelegate, which encapsulates the complexity of dealing with the session bean ResourceSession. The ResourceDelegate implementation for this example is shown in Example 8.1, and the corresponding remote interface for the Session Facade bean ResourceSession is shown in Example 8.2.

Example 8.1 Implementing Business Delegate Pattern - ResourceDelegate

```
// imports
...

public class ResourceDelegate {

  // Remote reference for Session Facade
  private ResourceSession session;

  // Class for Session Facade's Home object
  private static final Class homeClazz =
  corepatterns.apps.psa.ejb.ResourceSessionHome.class;

  // Default Constructor. Looks up home and connects
  // to session by creating a new one
  public ResourceDelegate() throws ResourceException {
    try {
      ResourceSessionHome home = (ResourceSessionHome)
        ServiceLocator.getInstance().getHome(
          "Resource", homeClazz);
      session = home.create();
    } catch(ServiceLocatorException ex) {
      // Translate Service Locator exception into
      // application exception
      throw new ResourceException(...);
    } catch(CreateException ex) {
```

```
    // Translate the Session create exception into
    // application exception
    throw new ResourceException(...);
  } catch(RemoteException ex) {
    // Translate the Remote exception into
    // application exception
    throw new ResourceException(...);
  }
}

// Constructor that accepts an ID (Handle id) and
// reconnects to the prior session bean instead
// of creating a new one
public BusinessDelegate(String id)
  throws ResourceException {
  super();
  reconnect(id);
}

// Returns a String ID the client can use at a
// later time to reconnect to the session bean
public String getID() {
  try {
    return ServiceLocator.getId(session);
  } catch (Exception e) {
    // Throw an application exception
  }
}

// method to reconnect using String ID
public void reconnect(String id)
  throws ResourceException {
  try {
    session = (ResourceSession)
            ServiceLocator.getService(id);
  } catch (RemoteException ex) {
    // Translate the Remote exception into
    // application exception
    throw new ResourceException(...);
  }
}

// The following are the business methods
// proxied to the Session Facade. If any service
// exception is encountered, these methods convert
// them into application exceptions such as
// ResourceException, SkillSetException, and so
// forth.

public ResourceTO setCurrentResource(
  String resourceId)
  throws ResourceException {
  try {
    return session.setCurrentResource(resourceId);
```

```
    } catch (RemoteException ex) {
      // Translate the service exception into
      // application exception
      throw new ResourceException(...);
    }
  }

  public ResourceTO getResourceDetails()
    throws ResourceException {

    try {
      return session.getResourceDetails();
    } catch(RemoteException ex) {
      // Translate the service exception into
      // application exception
      throw new ResourceException(...);
    }
  }

  public void setResourceDetails(ResourceTO vo)
    throws ResourceException {
    try {
      session.setResourceDetails(vo);
    } catch(RemoteException ex) {
      throw new ResourceException(...);
    }
  }

  public void addNewResource(ResourceTO vo)
    throws ResourceException {
    try {
      session.addResource(vo);
    } catch(RemoteException ex) {
      throw new ResourceException(...);
    }
  }

  // all other proxy method to session bean
  ...
}
```

Example 8.2 Remote Interface for ResourceSession

```
// imports
...
public interface ResourceSession extends EJBObject {

  public ResourceTO setCurrentResource(
    String resourceId) throws
    RemoteException, ResourceException;

  public ResourceTO getResourceDetails()
```

```
    throws RemoteException, ResourceException;
  public void setResourceDetails(ResourceTO resource)
    throws RemoteException, ResourceException;

  public void addResource(ResourceTO resource)
     throws RemoteException, ResourceException;

  public void removeResource()
     throws RemoteException, ResourceException;

  // methods for managing blockout time by the
  // resource
  public void addBlockoutTime(Collection blockoutTime)
     throws RemoteException, BlockoutTimeException;

  public void updateBlockoutTime(
    Collection blockoutTime)
     throws RemoteException, BlockoutTimeException;

  public void removeBlockoutTime(
    Collection blockoutTime)
     throws RemoteException, BlockoutTimeException;

  public void removeAllBlockoutTime()
     throws RemoteException, BlockoutTimeException;

  // methods for resource skillsets time by the
  //resource
  public void addSkillSets(Collection skillSet)
     throws RemoteException, SkillSetException;

  public void updateSkillSets(Collection skillSet)
     throws RemoteException, SkillSetException;

  public void removeSkillSet(Collection skillSet)
     throws RemoteException, SkillSetException;

   ...
}
```

Related Patterns

Service Locator

The Service Locator pattern may be used to create the Business Delegate's Service Locator, hiding the implementation details of any business service lookup and access code.

Proxy [GoF]

A Business Delegate may act as a proxy, providing a stand-in for objects in the business tier.

Adapter [GoF]

A Business Delegate may use the Adapter pattern to provide coupling for disparate systems.

Broker [POSA1]

A Business Delegate performs the role of a broker to decouple the business tier objects from the clients in other tiers.