J2EE Patterns Catalog

it has 13 patterns.
.
among only 8 patterns are important to know and I mostly used

~~Each pa~~ The UML diagram used in this document has declared all of UML notation in its diagram. it is really good TM BlueP
~~applicati~~

mostly pattern not concentrate on performance, it concentrate on de-coupling, easy exception handling, re-use, maintain, easy modification. so care about developer but not machine

Pattern Name	Description
Business Delegate [ACM01] 1	Reduce coupling between Web and Enterprise JavaBeans^{TM} tiers
Composite Entity [ACM01] 2	Model a network of related business entities
Composite View [ACM01] 3	Separately manage layout and content of multiple composed views
Data Access Object (DAO) [ACM01] 4	Abstract and encapsulate data access mechanisms
Fast Lane Reader 5	Improve read performance of tabular data
Front Controller [ACM01] 6	Centralize application request processing
Intercepting Filter [ACM01] 7	Pre- and post-process application requests
Model-View-Controller 8	Decouple data representation, application behavior, and presentation
Service Locator [ACM01] 9	Simplify client access to enterprise business services
Session Facade [ACM01] 10	Coordinate operations between multiple business objects in a workflow
Transfer Object [ACM01] 11	Transfer business data between tiers

purely it is used in EJB's Entity bean.

it is used in JSP. Tiles framework is sample of 'composite view'

it is used with DAO pattern. it is **better** than using Entity Bean when, want to get only data in read only

usually we use web. but we can use anywhere as we need

Filter and MessgeHandler sample of this pattern

mostly JNDI access

mostly use in EJB session bean

mostly use in distributed environment

Value List Handler [ACM01] 12	Efficiently iterate a virtual list is it good for lazy load of hibernate and uddi ??
View Helper [ACM01] 13	Simplify access to model state and data access logic

only one class is enough to implement BD pattern

this class has to be in client side. but iGate thy put in 'bus' layer. it is wrong

$$\begin{tikzcd}\boxed{\text{Client}} \arrow[r] & \boxed{\text{BD}} \arrow[r] & \boxed{\text{Business component}} \\ \boxed{\text{the client may be either Servlet / JSE}} \arrow[u] & & \end{tikzcd}$$

Business Delegate

16 - June -09

Brief Description

In distributed applications, lookup and exception handling for remote business components can be complex. When applications use business components directly, application code must change to reflect changes in business component APIs.

These problems can be solved by introducing an intermediate class called a ***business delegate***, which decouples business components from the code that uses them. The Business Delegate pattern manages the complexity of distributed component lookup and exception handling, and may adapt the business component interface to a simpler interface for use by views.

AIM: to free the client from
1. look up for remote business object
2. handling exception that is not related to business

.

ADDITIONAL :
1. Caching the remote object reference

Detailed Description

See the **Core J2EE™ Patterns**

Detailed Example

Sample application business delegate class AdminRequestBD handles distributed lookup and catches and adapts exceptions in the sample application order processing center (OPC).

- ***The AdminRequestBD business delegate manages distributed component lookup and handles exceptions.***

  The structure diagram in Figure 1 shows the ApplRequestProcessor servlet using AdminRequestBD to find and use distributed business components.

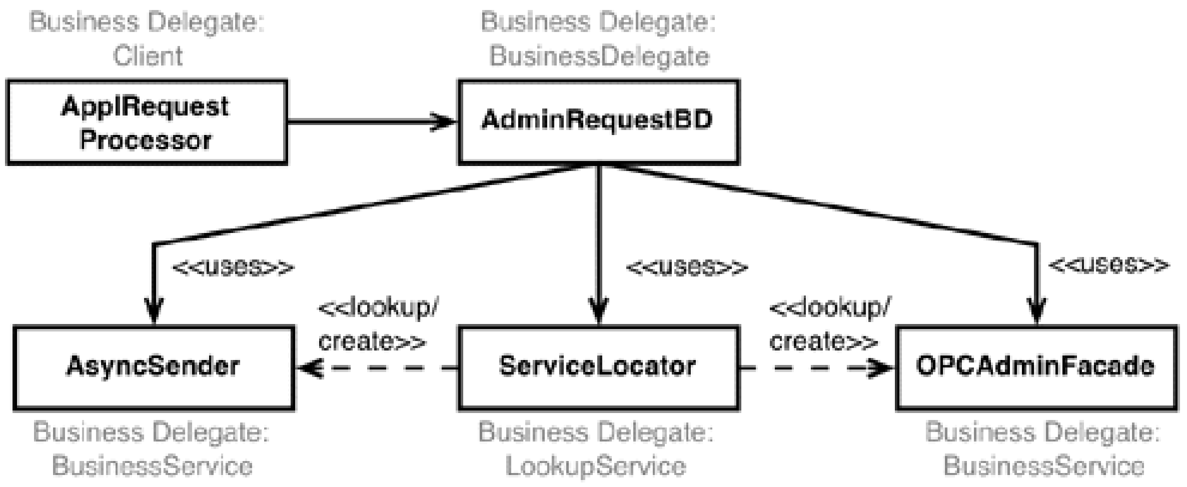

Business Delegate: Client
ApplRequest Processor
Business Delegate: BusinessDelegate
AdminRequestBD
<<uses>>
<<uses>>
<<uses>>
<<lookup/ create>>
<<lookup/ create>>
AsyncSender
ServiceLocator
OPCAdminFacade
Business Delegate: BusinessService
Business Delegate: LookupService
Business Delegate: BusinessService

Figure 1. AdminRequestBD locates and adapts other business components

The code sample below shows the constructor for AdminRequestBD. It uses class ServiceLocator to acquire the remote home interface of session facade OPCAdminFacade . (See the Service Locator and Session Facade design patterns.) It uses the remote home interface to create a OPCAdminFacade remote component interface, which it maintains in a private field. The block that locates and creates the enterprise bean reference catches exceptions related to finding the home interface and to creating the component interface. Any exception that occurs is then wrapped in an AdminBDException , which effectively hides the implementation details of the business delegate from its clients.

```
  public AdminRequestBD() throws AdminBDException {
     try {
         OPCAdminFacadeHome home = (OPCAdminFacadeHome) ServiceLocator.getInstance().getRemoteHome
(OPC_ADMIN_NAME, OPCAdminFacadeHome.class);
         opcAdminEJB = home.create();
      } catch (ServiceLocatorException sle) {
         throw new AdminBDException(sle.getMessage());
      } catch (CreateException ce) {
         throw new AdminBDException(ce.getMessage());
      } catch (RemoteException re) {
         throw new AdminBDException(re.getMessage());
      }
   }
```

BD can be used in SOA also.

- ***The OPC request processor uses AdminRequestBD for simple access to business components components.***

OPC servlet class ApplRequestProcessor receives service requests from the admin client in the form of XML messages transmitted using HTTP. One of these request is for statistics about orders that have a given status.

Method ApplRequestProcessor.getOrders receives part of an XML DOM tree representing a Web service request. It extracts the status code from the document and uses AdminRequestBD. getOrdersByStatus to retrieve a list of order information. The interface to that list is transfer object interface OrdersTO . (See the Transfer Object pattern.) The code in the request processor that retrieves the order information appears in the following code sample.

```
public class ApplRequestProcessor extends HttpServlet {
...
   String getOrders(Element root) {
      try {
         AdminRequestBD bd = new AdminRequestBD();
         NodeList nl = root.getElementsByTagName("Status");
         String status = getValue(nl.item(0));
         OrdersTO orders = bd.getOrdersByStatus(status);
...
}
```

Because it has already created the reference to an OPCAdminFacadeEJB, the AdminRequestBD object can simply forward the call to the enterprise bean's method getOrdersByStatus, as follows:

```
public class AdminRequestBD {
...
   public OrdersTO getOrdersByStatus(String status)
      throws AdminBDException {

      try {
         return opcAdminEJB.getOrdersByStatus(status);
      } catch (RemoteException re) {
         throw new AdminBDException(re.getMessage());
      } catch (OPCAdminFacadeException oafee) {
         throw new AdminBDException(oafee.getMessage());
      }
   }
...
}
```

Notice again that the method catches any exceptions that the enterprise bean may throw and re-throws an exception type that is specific to the business delegate's interface. This hides the business delegate's implementation details from the client.

18 - June -09

purely it is used in EJB's Entity bean.

Composite Entity

Also Known As

Previously known as Aggregate Entity

Brief Description

Mapping an object model to an Enterprise JavaBeans^{TM} (EJB^{TM}) object model is a common design problem in Java^{TM} 2 Platform, Enterprise Edition (J2EE^{TM}) applications. Given a network of inter-related objects, you must choose whether each object should be implemented as an entity EJB or a plain old Java object, and manage the relationships between the objects. Remote entity beans are best suited for modeling coarse-grained business entities. Modeling fine-grained entities as remote entity beans creates problems performance problems such as excessive remote communication. Choosing to use Bean Managed persistence means:

Dependent objects, which are objects whose data are meaningful only in the context of a relationship to another object, are especially prone to these problems. For example, in many applications, an Account object is meaningless outside of its relationship to its associated parent Customer object.

The ***Composite Entity*** design pattern offers a solution to modeling a networks of interrelated business entities. The composite entity's interface is coarse-grained, and it manages interactions between fine-grained objects internally. This design pattern is especially useful for efficiently managing relationships to dependent objects.

Before the EJB 2.0 specification, entity beans were always heavyweight and remote. The local, lightweight entity beans introduced with EJB 2.0 are intended for efficient modeling of fine-grained business entities and dependent objects. Efficient local access eliminates many of the problems listed above.

Detailed Description

See the **Core J2EE Patterns**

Detailed Example

It is important to note that this pattern is still viable with EJB 2.0 and Local Entity Beans. In both a local and a distributed environment there are drawbacks to using remote entity beans to model fine-grained or dependent business objects, so the composite entity pattern is applicable in both cases. Before EJB 2.0 existed, a common strategy for composite entity implementations was to use a Remote Entity Bean for the parent and use Java objects instead of remote entity beans for the dependent objects in a parent-child relationship. This also meant that the parent usually used Bean-managed persistence so it had to write the code to manage the relationships to the other objects also. But with the introduction of local entity beans in EJB 2.0, the strategy of using local entity beans to model fine-grained entities and dependent objects is recommended. An additional benefit of this strategy is that since all the objects in this strategy are Entity beans, you can efficiently use container-managed persistence and avoid writing code to manage the relationships. The following example from the Java Pet Store sample application illustrates this recommended strategy.

The following interfaces greatly simplifies implementing a composite entity, as shown by the following example.Enterprise bean local interfaces eliminate most of the performance costs associated with remote interfaces, making them practical for modeling fine-grained business entities and dependent objects.

The sample application models the application Customer class and several related objects as a composite entity. It uses local entity bean interfaces to model fine-grained objects that are dependent on the Customer object. Figure 1 below shows the Customer entity and its associated dependent objects: Profile, Account, ContactInfo, CreditCard, and Address.

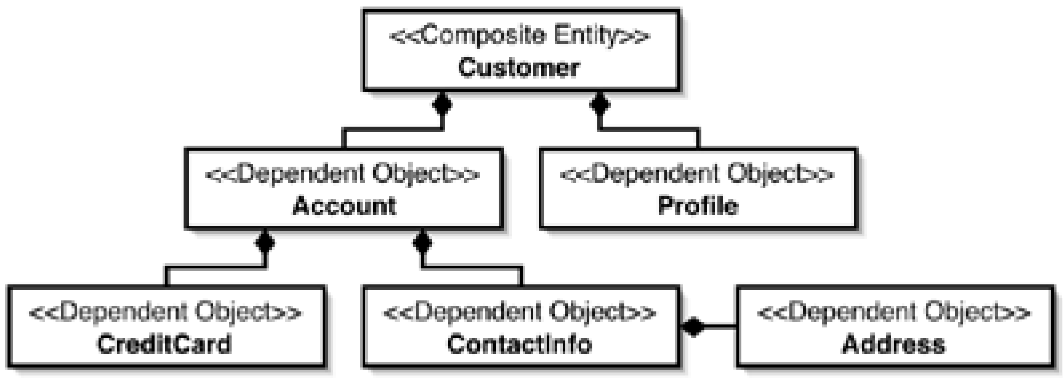


Figure 1. The Customer composite entity and its dependent objects

The diagram in Figure 1 shows the entities involved in the composite entity Customer. Several classes represent the enterprise beans used to implement these entities. The classes written by the developer appear in Table 1 below.

Table 1. Classes implementing composite entity Customer

Entity	EJB Class	Component Interface	Home Interface
Customer	CustomerEJB	CustomerLocal	CustomerLocalHome
Profile	ProfileEJB	ProfileLocal	ProfileLocalHome
Account	AccountEJB	AccountLocal	AccountLocalHome
CreditCard	CreditCardEJB	CreditCardLocal	CreditCardLocalHome
ContactInfo	ContactInfoEJB	ContactInfoLocal	ContactInfoLocalHome
Address	AddressEJB	AddressLocal	AddressLocalHome

The parent Customer composite entity provides access to the children dependent entities, as shown by the following excerpt from the CustomerLocal interface:

```
public interface CustomerLocal extends javax.ejb.EJBLocalObject {
   public String getUserId();
   public AccountLocal getAccount();
   public ProfileLocal getProfile();
}
```

Using a local entity bean instead of a plain old Java object to implement the dependent objects in a composite entity improves a design in several ways:

- *Improved performance.* Local interfaces improve performance because they eliminate the parameter and return value serialization and data transmission costs incurred by remote method calls. In addition, local interfaces are designed to allow the EJB container to optimize state representation, method calls, and relationships to other local beans.
- *Cleaner programming model.* Remote interfaces must usually be coarse to be practical. Local interfaces allow a designer to choose a granularity that is most appropriate for the application. Local interfaces also minimize the number of utility classes, such as data access objects, session facades, and transfer objects that are used to facilitate a distributed environment and remote interfaces.
- *Automatic relationship and integrity control.* Using container-managed relationships (CMR) with local interfaces simplifies implementation and reduces maintenance problems. With CMR, the container manages relationships between local entities, and enforces integrity constraints defined by the developer or deployer. The following code segment shows the definition of the relationship between the Customer and Profile entities in the sample application's EJB deployment descriptor:

```
<ejb-relation>
  ...
  <ejb-relationship-role>
    <ejb-relationship-role-name>ProfileEJB</ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <cascade-delete/>
    <relationship-role-source>
      <ejb-name>ProfileEJB</ejb-name>
    </relationship-role-source>
  </ejb-relationship-role>
</ejb-relation>
```

The application deployer or developer defines a multiplicity of One to the relationship between a Customer and a Profile. At runtime, the container ensures that exactly one Profile is associated with each Customer. In addition, the developer need not even write the code that manages the relationship, because it is automatically implemented by the container.

- *Cascading deletes.* The element <cascade-delete/> in the code sample above indicates that when a Customer instance is deleted, the container always automatically deletes the associated Profile instance, as well. Again, no code need be written or maintained to manage such deletions: the container handles it automatically.

Finally, a note about modeling. All of the entity beans that Customer uses to form a composite entity are dependent objects, meaning that they have little meaning in the application apart from the Customer object that "owns" them. In the parent-child relationship between the customer and its dependent objects means that the dependent objects such as Profile, Address, etc. can not exist if the parent Customer does not exist. In addition to dependent objects, entities accessed in a fine-grained manner may also be accesses by a composite entity, whether or not they are dependent on the composite entity. For example, adding a customer purchase history function to the customer entity might require fine-grained access to the product catalog. In such a case, catalog-related entities could be aggregated with the Customer composite entity, even though the catalog is not dependent on the Customer. Customer

18 - June -09

it is used in JSP. Tiles framework is sample of 'composite view'

Composite View

Brief Description

Many pages in an application view have the some different content and also some common content. As a user browses through the pages, the data and content among the different pages varies, but many elements such as a common header or sidebar on each view remain the same. The structure and layout of each page may be the same on all the pages of the view. And some elements or sections of a page may appear on several different pages. When such elements and groups are coded directly into application views, views are difficult to modify and are likely to contain inconsistencies. In addition, a consistent look and feel for every application view is difficult to achieve, and more difficult to maintain, when coding views directly.

A ***Composite View*** is a view built using other reusable sub-views. A single change to a sub-view is automatically reflected in every composite view that uses it. Furthermore, the composite view manages the layout of its sub-views and can provide a template, making consistent look and feel feel easier to achieve and modify across the entire application.

Detailed Description

See the **Core J2EE Patterns**

Detailed Example

There are several strategies for applying the Composite View pattern. What all strategies have in common is that sub-views are defined separately, and are individually reusable.

The layout of a composite view in the Java Pet Store sample application is controlled by a Java ServerPages^{TM} (JSP^{TM}) page called template.jsp. The contents of a specific composite view are controlled by another file, screendefinitions.xml, that binds JSP pages to named areas of the template. The following code block shows an excerpt from the Java Pet Store sample application website screen definitions file that defines the "main" screen, binding JSP pages to named areas of the template:

```
<screen name="main">
  <parameter key="banner" value="/banner.jsp" />
  <parameter key="sidebar" value="/sidebar.jsp" />
```

```
    <parameter key="body" value="/main.jsp" />
    <parameter key="footer" value="/footer.jsp" />
  </screen>
```

The example shown in Figure 1 is the screen that results from the definition above. Each page area is populated with data from a separate JSP page.

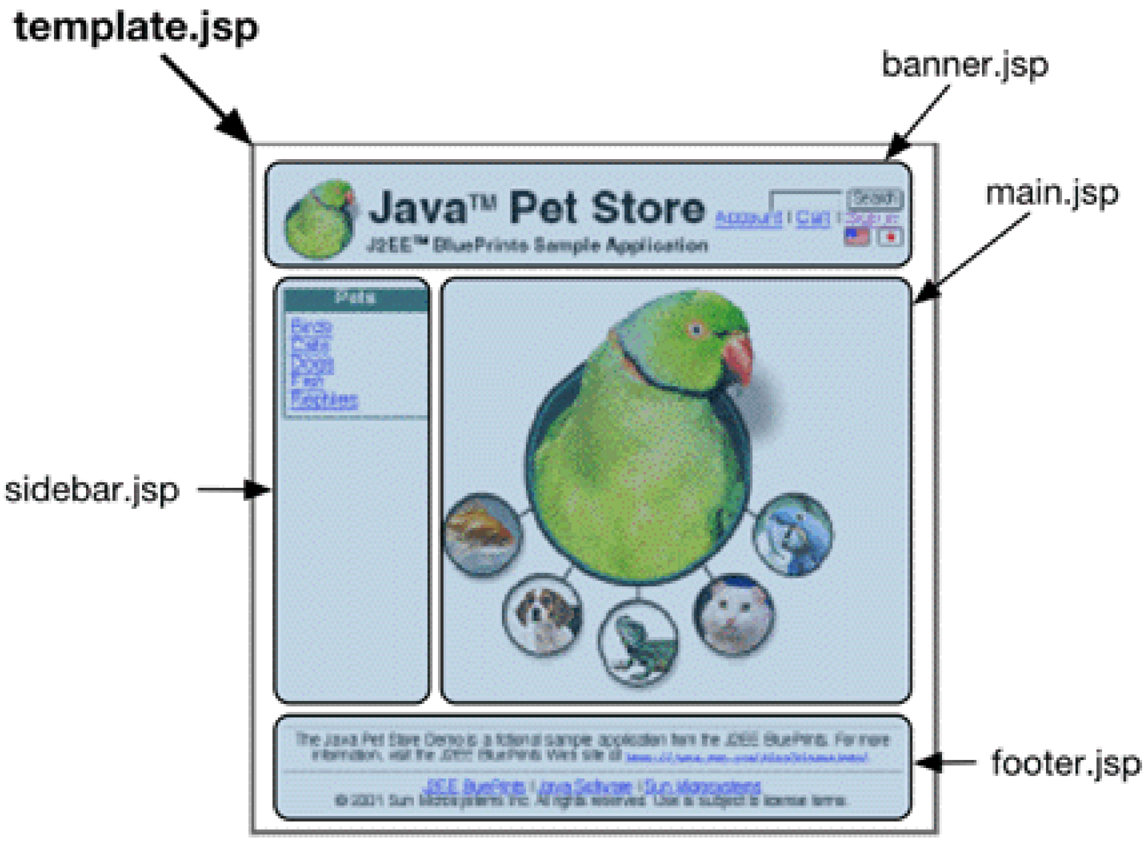


Figure 1. Multiple JSP pages form a composite view

The sample application controls composite view assembly using a centralized view manager called TemplateServlet . To create a composite view, the application developer first defines a *template file*, which controls the layout of the view. The developer uses template:insert tags at various points in the template file to include dynamically-generated content. (The insert tag is implemented by the Web Application Framework class InsertTag .)

The developer then creates a *screen definitions file*, which associates each insert tag with the URL of the sub-view to be included at that point. So the template controls the layout of the composite view, and the screen definition controls selects the group of sub-views to be included within the layout. Each screen definition selects a different set of sub-views, so the contents of each view may differ, while the layout of all views remains the same. Figure 2 below shows how the template servlet assembles a composite view.

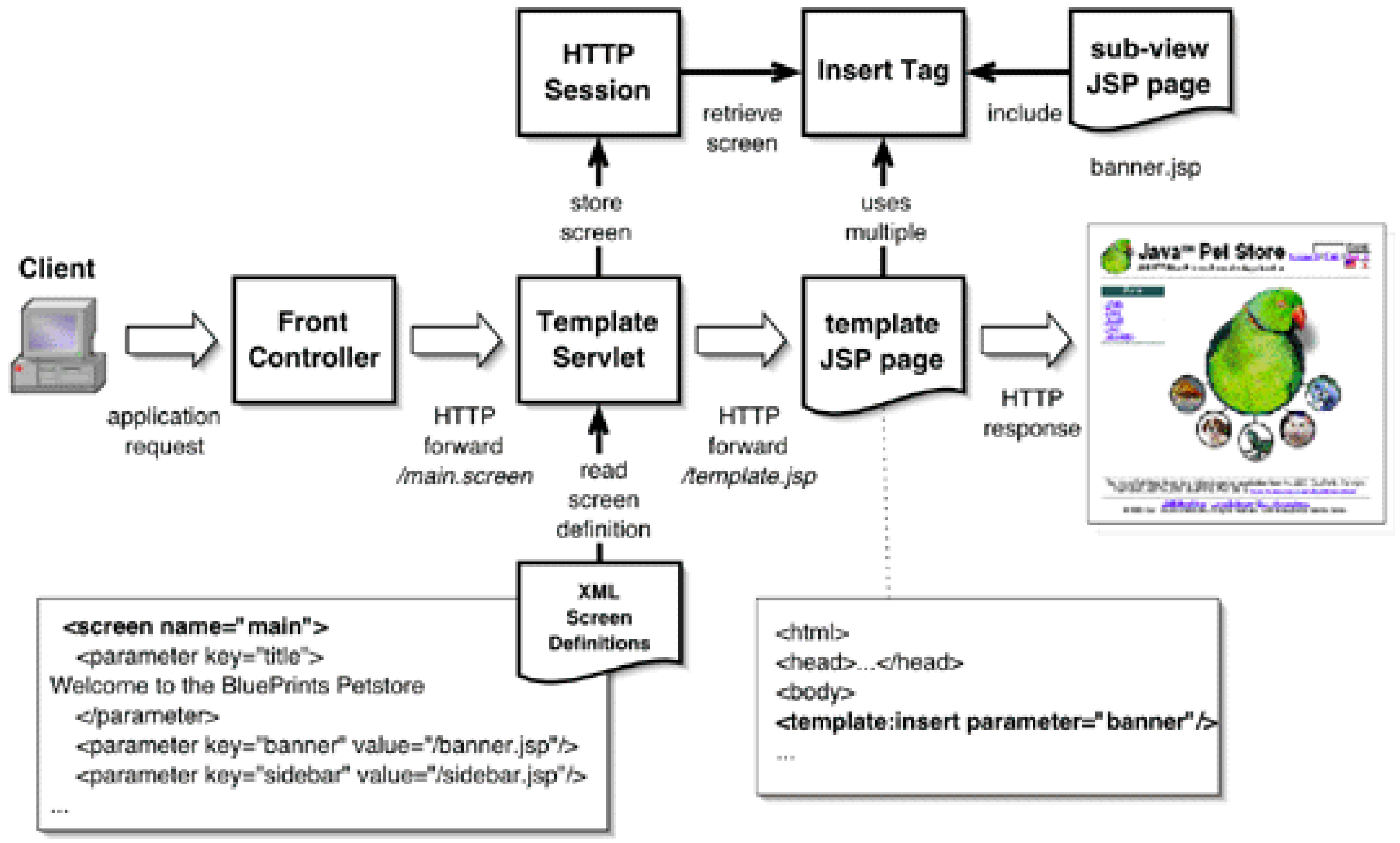


Figure 2. Block diagram of composite view assembly

After servicing a client request, the Front Controller selects the next view to display and forwards the request to it. (In the figure, this view is /main.screen.) A servlet mapping routes the request to a TemplateServlet, which reads the screen definitions defined by the application developer, finds the appropriate screen, and stores the screen in an HttpSession attribute. The TemplateServlet then forwards the request to the template defined in the screen definitions file (/template.jsp). The screen definition associates symbolic names with Web resources to be used as sub-views in the composite view. The example in the figure shows the tag <parameter key="banner" value="/banner.jsp"/>, which defines /banner.jsp as the content for the banner of the screen called main.

The template file contains the page layout, plus any text that appears on every page of the application. It also contains embedded template:insert tags, each of which indicates the name of a sub-view to be included in the response. For example, <template:insert parameter="banner"/> inserts the banner page for the current screen at that point.

When the server executes the template page, each template:insert tag uses the screen (retrieved from HttpSession) to look up the URL of the content to include. In the example shown in Figure 2, the contents of /banner.jsp are inserted at the point in the template where the template:insert tag occurs. The page resulting from the execution of the template JSP page is served to the client as the HTTP response.

A more detailed sequence diagram of this process appears in Figure 2. Each item in the list following the figure describes the corresponding numbered transition in the diagram.

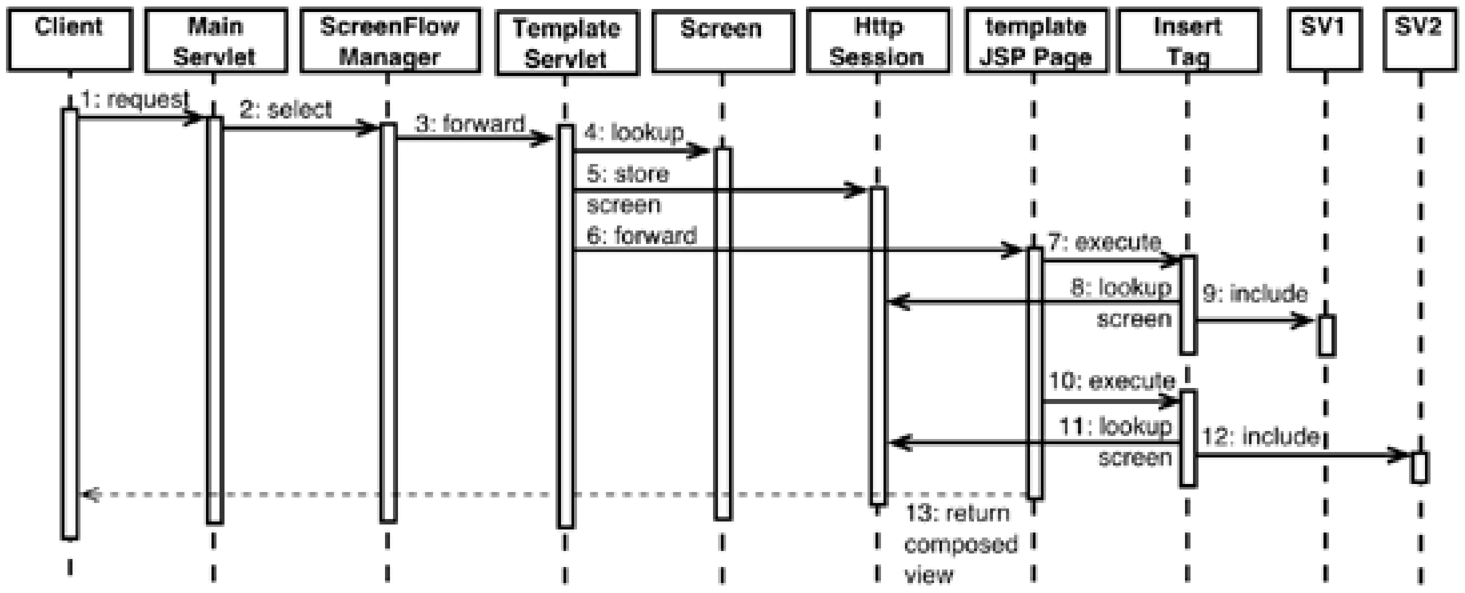


Figure 2. A sequence diagram of composite view assembly

- 1. The client requests service
- 2. After processing the request, MainServlet (the Front Controller) directs the ScreenFlowManager to select the next view
- 3. The ScreenFlowManager determines the name of the screen to display
- 4. The request is forwarded to TemplateServlet
- 5. The TemplateServlet identifies the requested Screen object and stores it in HttpSession
- 6. The request is forwarded to the template JSP page
- 7. The template JSP page executes an insert tag
- 8. The insert tag retrieves the screen from HttpSession
- 9. The insert tag inserts the content from the URL corresponding to the value of its parameter attribute
- 10 - 12. The template JSP page executes another insert tag, which inserts the

content as in the steps 7 - 9.

- 13. Template execution is complete, and the server returns the generated response to the client.

---

17 - June -09

need one interface,
class which implements that interface.
.
DAO class can be either Plug able DAO (separate DOA class which impliment that interface for separate Data source like DB / XML / file system)
.
DAO class can be either Generic DAO (if only different JDBC) and externalizing sql to XML file.
.
DAO object created by factory pattern

Data Access Object

Also Known As

DAO

Brief Description

Code that depends on specific features of data resources ties together business logic with data access logic. This makes it difficult to replace or modify an application's data resources.

DAO pattern, hide the resource specific exception and throw as application specific exception.

The ***Data Access Object*** (or ***DAO***) pattern:

- separates a data resource's client interface from its data access mechanisms
- adapts a specific data resource's access API to a generic client interface

The DAO pattern allows data access mechanisms to change independently of the code that uses the data.

Detailed Description

See the **Core J2EE™ Patterns**

Detailed Example

The Java Pet Store sample application uses the DAO pattern both for database vendor-neutral data access, and to represent XML data sources as objects.

- ***Accessing a database with a DAO.***

  A Data Access Object class can provide access to a particular data resource without coupling the resource's API to the business logic. For example, sample application classes access catalog categories, products, and items using DAO interface CatalogDAO.

  Reimplementing CatalogDAO for a different data access mechanism (to use a Connector, for example), would have little or no impact on any classes that use CatalogDAO, because only the implementation would change. Each potential alternate implementation of CatalogDAO would access data for the items in the catalog in its own way, while presenting the same API to the class that uses it.

  The following code excerpts illustrate how the sample application uses the DAO pattern to separate business logic from data resource access mechanisms:

  - Interface CatalogDAO defines the DAO API. Notice that the methods in the interface below make no reference to a specific data access mechanism. For example, none of the methods specify an SQL query, and they throw only exceptions of type CatalogDAOSysException . Avoiding mechanism-specific information in the DAO interface, including exceptions thrown, is essential for hiding implementation details.

```
public interface CatalogDAO {
    public Category getCategory(String categoryID, Locale l)
        throws CatalogDAOSysException;
    public Page getCategories(int start, int count, Locale l)
        throws CatalogDAOSysException;
    public Product getProduct(String productID, Locale l)
        throws CatalogDAOSysException;
    public Page getProducts(String categoryID, int start, int count, Locale l)
        throws CatalogDAOSysException;
    public Item getItem(String itemID, Locale l)
        throws CatalogDAOSysException;
    public Page getItems(String productID, int start, int size, Locale l)
        throws CatalogDAOSysException;
    public Page searchItems(String query, int start, int size, Locale l)
        throws CatalogDAOSysException;
}
```

- Class CloudscapeCatalogDAO implements this interface for the Cloudscape relational database, as shown in the following code excerpt. Note that the SQL to access the Cloudscape database is hard-coded.

```
public class CloudscapeCatalogDAO implements CatalogDAO {
...
public static String GET_CATEGORY_STATEMENT
  = "select name, descn "
  + " from (category a join category_details b on a.catid=b.catid) "
  + " where locale = ? and a.catid = ?";
...
public Category getCategory(String categoryID, Locale l)
    throws CatalogDAOSysException {
      Connection c = null;
      PreparedStatement ps = null;
      ResultSet rs = null;
      Category ret = null;

      try {
        c = getDataSource().getConnection();
        ps = c.prepareStatement(GET_CATEGORY_STATEMENT,
                                ResultSet.TYPE_SCROLL_INSENSITIVE,
                                ResultSet.CONCUR_READ_ONLY);
```

```
            ps.setString(1, l.toString());
            ps.setString(2, categoryID);
            rs = ps.executeQuery();
            if (rs.first()) {
              ret = new Category(categoryID, rs.getString(1), rs.getString(2));
            }

            rs.close();
            ps.close();
            c.close();
            return ret;
          } catch (SQLException se) {
            throw new CatalogDAOSysException("SQLException: " + se.getMessage());
          }
      }
    ...
    }
```

- ***Implementation strategies.*** Designing a DAO interface and implementation is a tradeoff between simplicity and flexibility. The sample application provides examples of several strategies for implementing the Data Access Object pattern.

    - *Implement the interface directly as a class.* The simplest (but least flexible) way to implement a data access object is to write it as a class. Class ScreenDefinitionsDAO, described below and shown in Figure 1 above, is an example of a class that directly implements a DAO interface. This approach separates the data access interface from the details of how it is implemented, providing the benefits of the DAO pattern. The data access mechanism can be changed easily by writing a new class that implements the same interface, and changing client code to use the new class. Yet this approach is inflexible because it requires a code changes to modify the data access mechanism.

    - *Improve flexibility by making DAOs "pluggable".* A pluggable DAO allows an application developer or deployer to select a data access mechanism with no changes to program code. In this approach, the developer accesses a data source only in terms of an abstract DAO interface. Each DAO interface has one or more concrete classes that implement that interface for a particular type of data source. The application uses a factory object to select the DAO implementation at runtime, based on configuration information.

      For example, the sample application uses factory class CatalogDAOFactory to select the class that implements the DAO interface for the catalog. Figure 2 below presents a structure diagram of the Data Access Object design pattern using a factory to select a DAO implementation.

i guess, it is class
with static
methods. but not
abstract factory

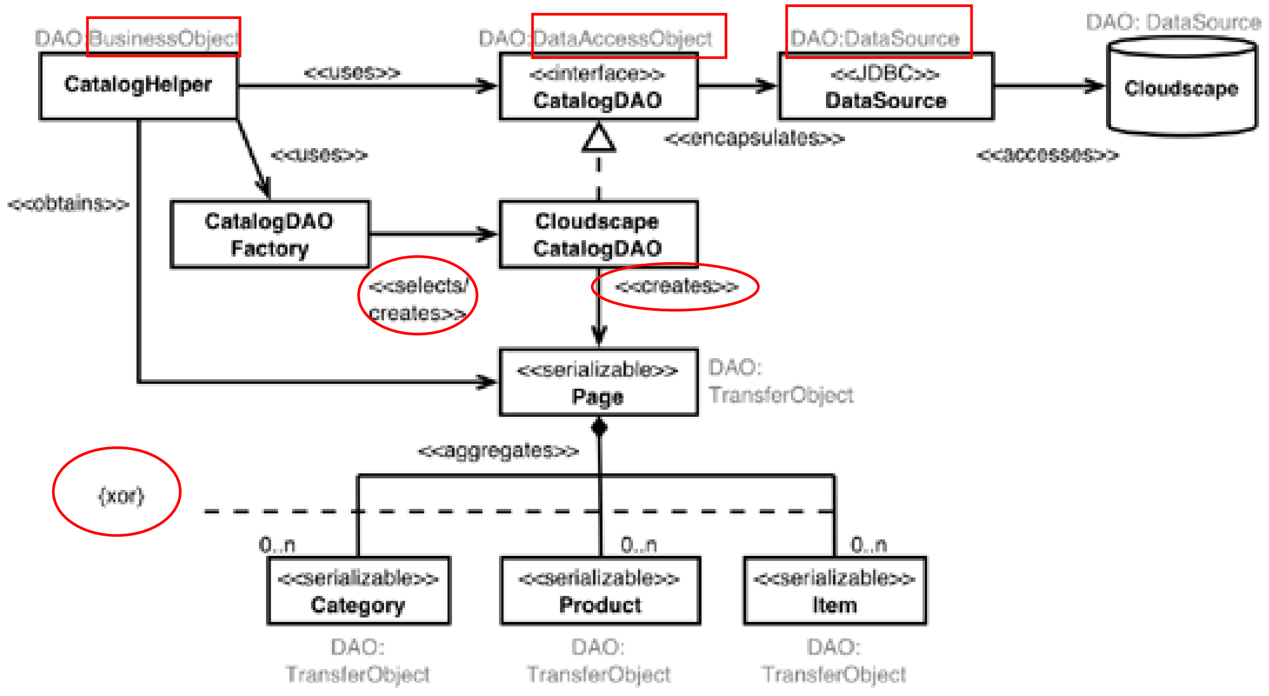


Figure 2. A pluggable DAO

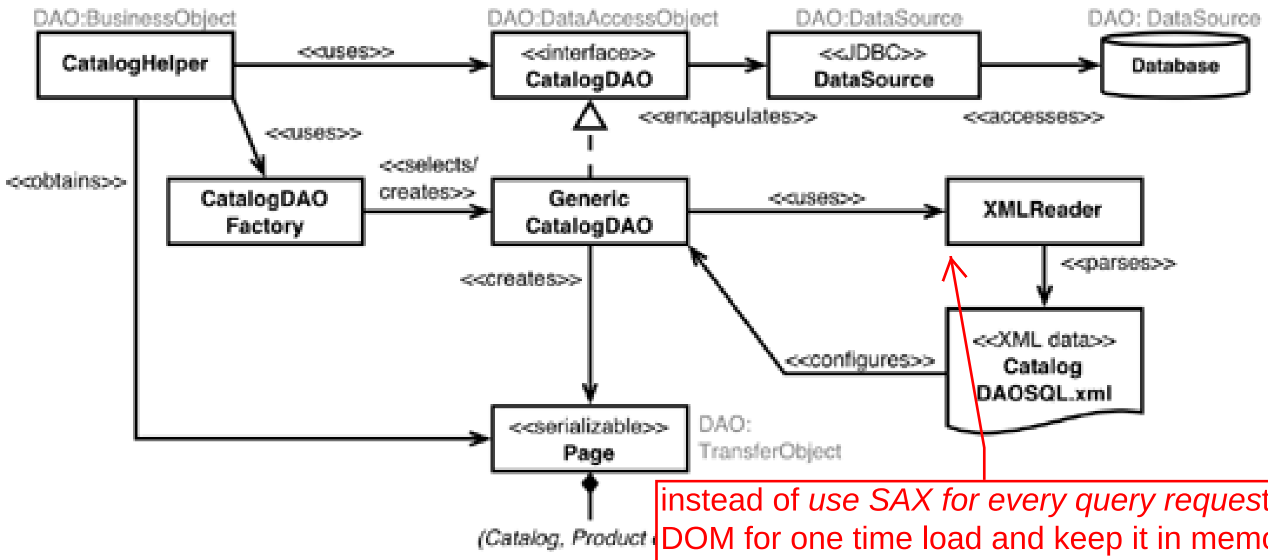
it is really super.

At runtime, the CatalogHelper uses the CatalogDAOFactory to create an object that implements CatalogDAO. The factory looks up the name of the class that implements the DAO interface in environment entry " param/CatalogDAOClass". The CatalogHelper accesses the catalog data source exclusively using the object created by the factory. In the example shown in the figure, the environment entry was set to the (fully-specified) name of class CloudscapeCatalogDAO . This class implements the catalog DAO interface in terms of JDBC TM data sources, accessing a Cloudscape relational database.

This approach is more flexible than using a hard-coded class. To add a new type of data source, an application developer would simply create a class that implements CatalogDAO in terms of the new data source type, specify the implementing class's name in the environment entry, and re-deploy. The factory would create an instance of the new DAO class, and the application would use the new data source type.

- *Reduce redundancy by externalizing SQL*. Writing a separate class for data source types that have similar APIs can create a great deal of redundant code. For example, JDBC data sources differ from one another primarily in the SQL used to access them. The only differences between the Cloudscape DAO described above and the DAO for a different SQL database are the connection string and the SQL used to access the database.

  The sample application reduces redundant code by using a "generic DAO" that externalizes the SQL for different JDBC data sources. Figure 3 below shows how the sample application uses an XML file to specify the SQL for different JDBC data sources.



instead of *use SAX for every query request*, use DOM for one time load and keep it in memory, but return the query in separate String object to avoid thread collision.

Figure 3. Externalizing DAO SQL

In the figure, the CatalogDAOFactory has selected an object of type GenericCatalogDAO as the DAO to access the catalog. An XML file called CatalogDAOSQL.xml specifies the SQL for each supported operation on each type of database. GenericCatalogDAO configures itself by selecting a group of SQL statements from the XML file that correspond to the database type named by environment entry " param/CatalogDAODatabase". The code sample below shows the definition of the XML for the getCategory operation of the CatalogDAO. Different SQL is specified for "cloudscape" and "oracle" database types.

```
<DAOConfiguration>
   <DAOStatements database="cloudscape">
      <SQLStatement method="GET_CATEGORY">
         <SQLFragment parameterNb="2">
            select name, descn
               from (category a join category_details b
                    on a.catid=b.catid)
               where locale = ? and a.catid = ?
         </SQLFragment>
      </SQLStatement>
      ...
   <DAOStatements database="oracle">
      <SQLStatement method="GET_CATEGORY">
         <SQLFragment parameterNb="2">
            select name, descn
               from category a, category_details b
               where a.catid = b.catid and locale = ? and a.catid = ?
         </SQLFragment>
      </SQLStatement>
     ...
</DAOConfiguration>
```

Method GenericCatalogDAO.getCategory chooses the SQL corresponding to the configured database type, and uses it to fetch a category from the database via JDBC. The following code excerpt shows shows the implementation of the method.

```
public Category getCategory(String categoryID, Locale locale)
  throws CatalogDAOSysException {
  Connection connection = null;
  ResultSet resultSet = null;
  PreparedStatement statement = null;
  try {
    connection = getDataSource().getConnection();
    String[] parameterValues = new String[] { locale.toString(), categoryID };
    statement = buildSQLStatement(connection, sqlStatements,
       XML_GET_CATEGORY, parameterValues);
    resultSet = statement.executeQuery();
    if (resultSet.first()) {
      return new Category(categoryID, resultSet.getString(1), resultSet.getString(2));
    }
    return null;
  } catch (SQLException exception) {
    throw new CatalogDAOSysException("SQLException: " + exception.getMessage());
  } finally {
    closeAll(connection, statement, resultSet);
  }
}
```

~~Notice that the method catches any possible SQLException and converts it to a CatalogDAOSysException, hiding the implementation detail that the DAO uses a JDBC database~~.

This strategy supports multiple JDBC databases with a single DAO class. It both decreases redundant code, and makes new database types easier to add. To support a new database type, a developer simply adds the SQL statements for that database type to the XML file, updates the environment entry to use the new type, and redeploys.

The pluggable DAO and generic DAO strategies can be used separately. If you know that a DAO class will only ever use JDBC databases (for example), the generic DAO class can be hardwired into the application, instead of selected by a factory. For maximum flexibility, the sample application uses both a factory method and a generic DAO.

- ***Encapsulating non-database data resources as DAO classes.***

  A data access object can represent data that is not stored in a database. The sample application uses the DAO pattern to represent XML data sources as objects. Sample application screens are defined in an XML file which is interpreted by the class ScreenDefinitionDAO. Specifying screen definitions externally makes access to the screen definitions more flexible. For example, if the application designers (or maintainers) decide to change the application to store screen descriptions in the database, instead of in an XML file, they would need only to implement a single new class (ScreenFlowCloudscapeDAO, for example). The code that uses ScreenDefinitionDAO would remain unchanged, but the data would come from the database via the new class.

The screen definitions mechanism in the sample application provides an example of a concrete Data Access Object representing an underlying, non-database resource (an XML file).

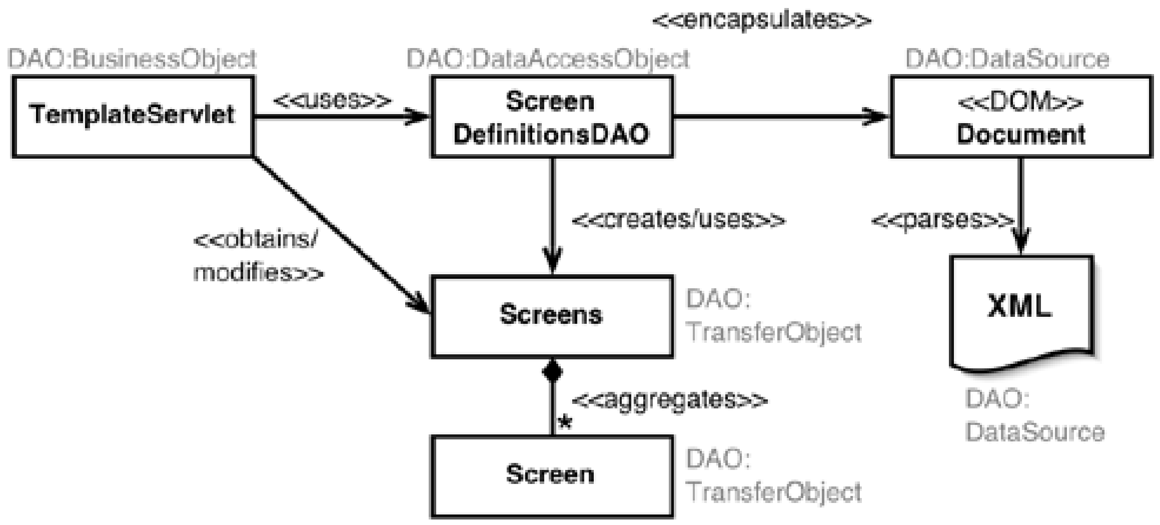


Figure 1. Data Access Object providing access to XML data source

~~Figure 1 shows a structure diagram of the ScreenDefinitionDAO managing the loading and interpretation of XML data that defines application screens.~~

- ~~TemplateServlet uses the ScreenDefinitionDAO to load screen definitions:~~

```
~~Screens screenDefinitions =~~
  ~~ScreenDefinitionDAO.loadScreenDefinitions(screenDefinitionURL);~~
```

- ~~ScreenDefinitionDAO represents screen definitions in an XML file deployed with an application. It uses XML APIs to read the screen definitions from the XML file. Only this class would need to be replaced to support storing screen definitions in some other way. The method that loads screen definitions looks like this:~~

```
 ~~public static Screens loadScreenDefinitions(URL location) {~~
    ~~Element root = loadDocument(location);~~
    ~~if (root != null) return getScreens(root);~~
    ~~else return null;~~
 }
~~...~~
~~public static Screens getScreens(Element root) {~~
 ~~// get the template~~
 ~~String defaultTemplate = getTagValue(root, DEFAULT_TEMPLATE);~~
 ~~if (defaultTemplate == null) {~~
```

```
        ~~System.err.println("*** ScreenDefinitionDAO error: " +~~
                        ~~" Default Template not Defined.");~~
        ~~return~~ null;
    }

    Screens screens = new Screens(defaultTemplate);
    getTemplates(root, screens);
    // get screens
    NodeList list = root.getElementsByTagName(SCREEN);
    for (int loop = 0; loop < list.getLength(); loop++) {
        Node node = list.item(loop);
        if ((node != null) && node instanceof Element) {
            String templateName = ((Element)node).getAttribute(TEMPLATE);
            String screenName = ((Element)node).getAttribute(NAME);
            HashMap parameters = getParameters(node);
            Screen screen = new Screen(screenName, templateName, parameters);
            if (!screens.containsScreen(screenName)) {
                screens.addScreen(screenName, screen);
            } else {
                System.err.println("*** Non Fatal errror: Screen " + screenName +
                                " defined more than once in screen definitions file");
            }
        }
    }
    return screens;
    }
   ...
```

~~The code fragment above shows how loadScreenDefinitions loads screen definitions using DOM interfaces, while hiding that fact from clients of the class. A client of this class can expect to receive a Screens object regardless of how those screens are loaded from persistent storage. Method getScreens handles all of the DOM-specific details of loading a screen from an XML file.~~

18 - June - 09

it is used with DAO pattern. it is **better** than using Entity Bean when, want to get only data in read only

Fast Lane Reader

if we need to update or delete with table go for entity bean.
.
if you want get table data as read only go for Fast Lane Reader

Brief Description

it is alternative for Entity bean for data access from table

Sometimes applications access data in a tabular fashion, such as when browsing a catalog or a list of names or when exporting data in batches for use elsewhere. This kind of data access is usually read-only. In such situations, using entity beans to represent persistent data incurs overhead and provides little benefit. Entity beans are best for coarse-grained access to individual business entities, and are not effective for read-only access to large quantities of tabular data.

The ***Fast Lane Reader*** design pattern provides a more efficient way to access tabular, read-only data. A fast lane reader component directly accesses persistent data using JDBC^{TM} components, instead of using entity beans. The result is improved performance and less coding, because the component represents data in a form that is closer to how the data are used.

Detailed Description

See **FastLaneReader-detailed.html**

Detailed Example

The sample application uses the Fast Lane Reader pattern in class CatalogHelper , together with DAO class CatalogDAO . Figure 1 below shows a structure diagram of CatalogHelper acting as a fast lane reader, along with associated classes.

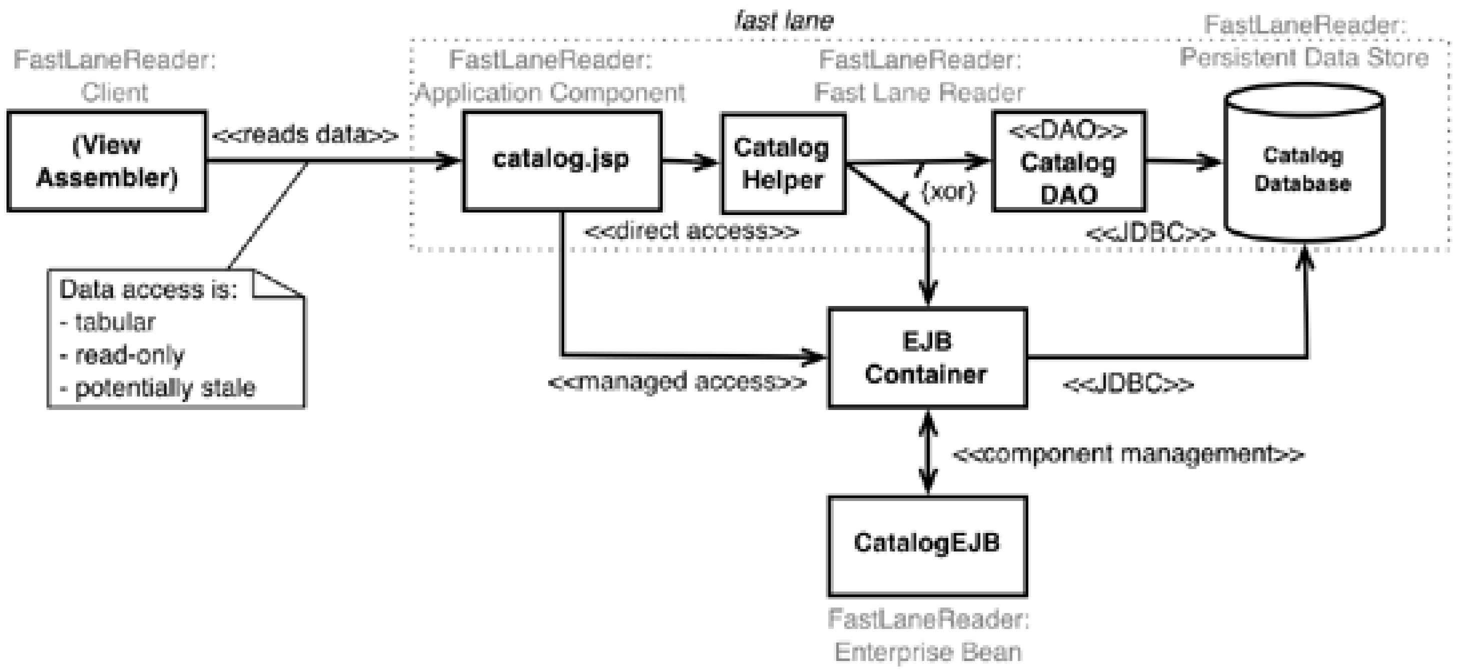


Figure 1. The CatalogHelper acting as fast lane reader

CatalogHelper has a boolean property useFastLane that indicates whether or not to use the Fast Lane Reader pattern. If useFastLane is true, the CatalogHelper implements most of its methods to the CatalogDAO of the same name; if it is false, CatalogHelper delegates the call to the corresponding method of CatalogEJB . The code sample below shows the CatalogHelper calling the appropriate method depending on whether useFastLane is true.

```
public Page getCategories()
    throws CatalogException {
    return useFastLane
       ? getCategoriesFromDAO(start, count, locale)
       : getCategoriesFromEJB(start, count, locale);
 }
```

By using the Fast Lane Reader pattern, CatalogHelper can improve performance by avoiding using enterprise beans.

The sample application provides these two options primarily for illustration purposes. In an actual production application, the architecture would either use a fast lane pattern or it would not, instead of providing a runtime choice.

17 - June -09

This pattern does not limit the number of Front Controller components to just one. In fact, a design can have multiple controller components each acting as an entry point for a related set of services. For example a design can have two front controllers; one
mapped as an entry point for servicing audio processing requests and another front controller can be mapped as an entry point for all requests regarding online content management.

Front Controller

Brief Description

Many interactive Web applications are composed of brittle collections of interdependent Web pages. Such applications can be hard to maintain and extend.

The ***Front Controller*** pattern defines a single component that is responsible for processing application requests. A front controller centralizes functions such as view selection, security, and templating, and applies them consistently across all pages or views. Consequently, when the behavior of these functions need to change, only a small part of the application needs to be changed: the controller and its helper classes.

Detailed Description

See the **Core J2EE^{TM} Patterns**

Detailed Example

- ***Centralizing request processing and view selection.***

  A Servlet is utilized as the main point of entry for web requests. The class MainServlet is the front controller for the Java Pet Store sample application website. All requests that end with **.do* are mapped to go through the MainServlet for processing. The following code excerpts form the core of the controller. A sequence diagram outlining the actions taken by the MainServlet in response to the user request appears in Figure 1 below.

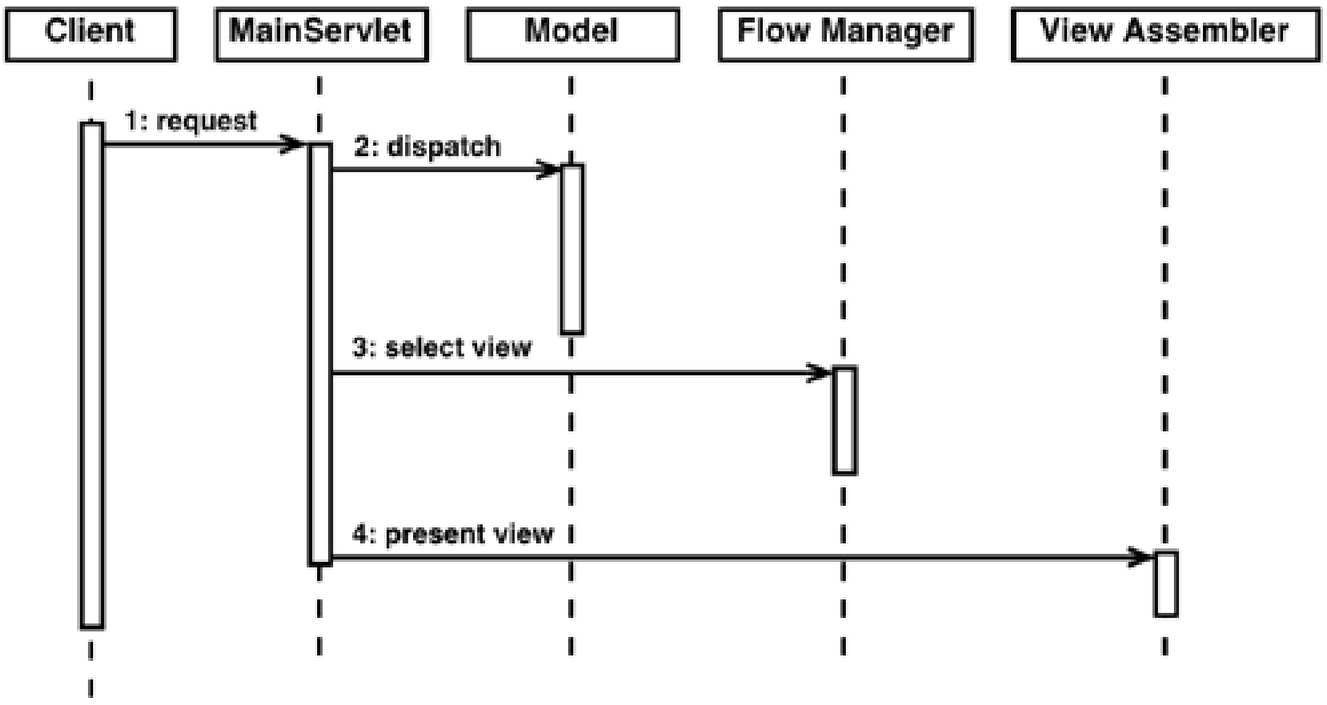


Figure 1. Sequence diagram of MainServlet in action

The MainServlet source code is straightforward:

- The methods that process HTTP POST and GET (transition number 1 in Figure 1) both use method doProcess, shown in this example. The method receives the request and response, and passes the request to the RequestProcessor, which dispatches the request to the business logic (represented by the "Model" in Figure 1 above) that handles it. The request processor executes an application function that corresponds to the request URL ("2: dispatch" in Figure 1). The map from request URLs to application functions is defined in an XML file, mappings.xml.

```
private void doProcess(HttpServletRequest request,
                       HttpServletResponse response)
  throws IOException, ServletException {
   ...
  try {
    getRequestProcessor().processRequest(request);
```

- After dispatching the request to the business logic, the controller then passes the request to the ScreenFlowManager, which chooses the next screen to display, again based on the contents of mappings.xml ("3: select view" in Figure 1). Exceptions can also be mapped to screens in mappings.xml: if business logic throws an exception, the exception is stored in the request, and the next screen is chosen based on the exception type. If no next screen is defined, a default screen is used.

```
      getScreenFlowManager().forwardToNextScreen(request, response);
   } catch (Throwable ex) {
     String className = ex.getClass().getName();
     nextScreen = getScreenFlowManager().getExceptionScreen(ex);
     // put the exception in the request
     request.setAttribute("javax.servlet.jsp.jspException", ex);
     if (nextScreen == null) {
        // send to general error screen
        ex.printStackTrace();
        throw new ServletException("MainServlet: unknown exception: " +
           className);
     }
   }
```

- Finally, the front controller forwards the request to the next screen ("4: present view" in Figure 1). The component (usually the templating service) at the URL for the next screen receives the screen name and any server-side state defined by the previous operations. This functionality is delegated to the ScreenFlowManager, which forwards to the next screen.

17 - June -09

Intercepting Filter

Brief Description

Most applications have some requirements, such as security and logging, that are applicable across all application requests. To add such functionality separately to each application service would be time-consuming, error-prone, and difficult to maintain. Even implementing these services within a front controller would still require code changes to add and remove services. The sequence diagram in Figure 1 below shows how each Web resource is responsible for calling such services individually.

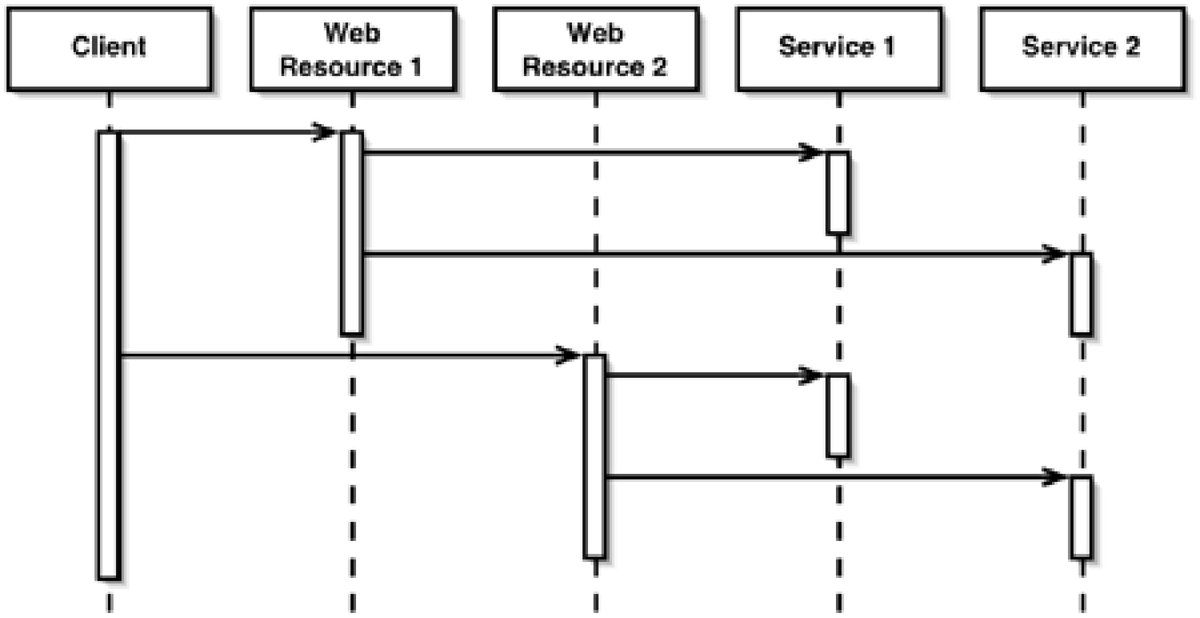


Figure 1. Before Intercepting Filter

The ***Intercepting Filter*** pattern wraps existing application resources with a filter that intercepts the reception of a request and the transmission of a response. An intercepting filter can pre-process or redirect application requests, and can post-process or replace the content of application responses. Intercepting filters can also be stacked one on top of the other to add a chain of separate, declaratively-deployable services to existing Web resources with no changes to source code. Figure 2 below shows a chain of two intercepting filters intercepting requests to two Web resources that they wrap.

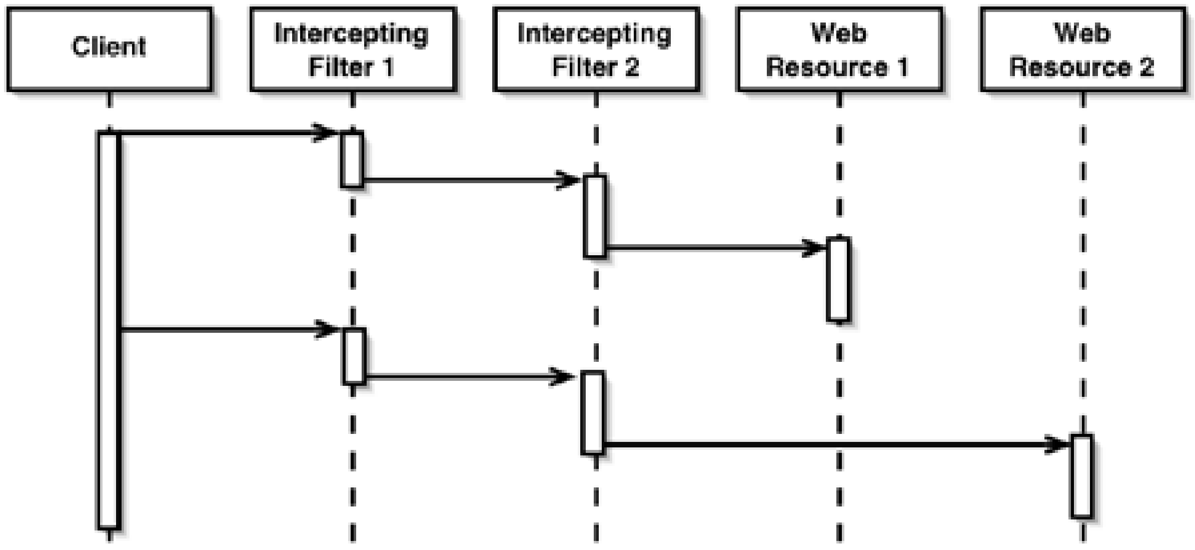


Figure 2. After Intercepting Filter

Detailed Description

See the **Core J2EE^{TM} Patterns**

Detailed Example

The Servlet Filter interface in the Java^{TM} 2 Platform, Enterprise Edition (J2EE) platform is a direct implementation of the Intercepting Filter pattern. The sample application defines two servlet filters: one that mediates customer signon, and another that enforces response encoding.

- ***The user signon filter.***

  Sample application class SignOnFilter is an intercepting filter that intercepts all requests to the pet store front controller. The SignOnFilter servlet filter detects requests for pages that require user signon, and redirects such requests to the user signon page if no user is currently signed on. An exhaustive explanation of this process appears in *User Signon and Customer Registration Module* (part of the online document *Sample Application Design and Implementation* [BT-SADI02]).

- ***The encoding filter.***

  Sample application class EncodingFilter is an intercepting filter that modifies *responses* from the pet store front controller, ensuring that the encoding of the response is always set consistently.

- The code sample below from class EncodingFilter shows how the servlet filter sets the response encoding of each response, and then passes the request to the next filter down the chain. The filter chain mechanism makes such filters composable, so multiple intercepting filters can wrap a single resource.

```
public void doFilter(ServletRequest srequest,
                     ServletResponse sresponse,
                     FilterChain chain)
  throws IOException, ServletException {

  HttpServletRequest request = (HttpServletRequest)srequest;
  request.setCharacterEncoding(targetEncoding);

  // move on to the next
  chain.doFilter(srequest,sresponse);
}
```

- The excerpt below from the pet store web application description web.xml shows how the encoding filter is configured to wrap all requests matching the pattern " /*" (relative to the application context root).

```
<!-- Encoding Filter Mapping Start-->
<filter-mapping>
  <filter-name>EncodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

---

17 - June -09

it reduce the coupling among
1. Client (presentation)
2. Model (domain objects and business logic)
3. Controller
.
and make possible to re use any of these layer
.
maintain code is easy

Model-View-Controller

Also Known As

MVC

Brief Description

Several problems can arise when applications contain a mixture of data access code, business logic code, and presentation code. Such applications are difficult to maintain, because interdependencies between all of the components cause strong ripple effects whenever a change is made anywhere. High coupling makes classes difficult or impossible to reuse because they depend on so many other classes. Adding new data views often requires reimplementing or cutting and pasting business logic code, which then requires maintenance in multiple places. Data access code suffers from the same problem, being cut and pasted among business logic methods.

The Model-View-Controller design pattern solves these problems by decoupling data access, business logic, and data presentation and user interaction.

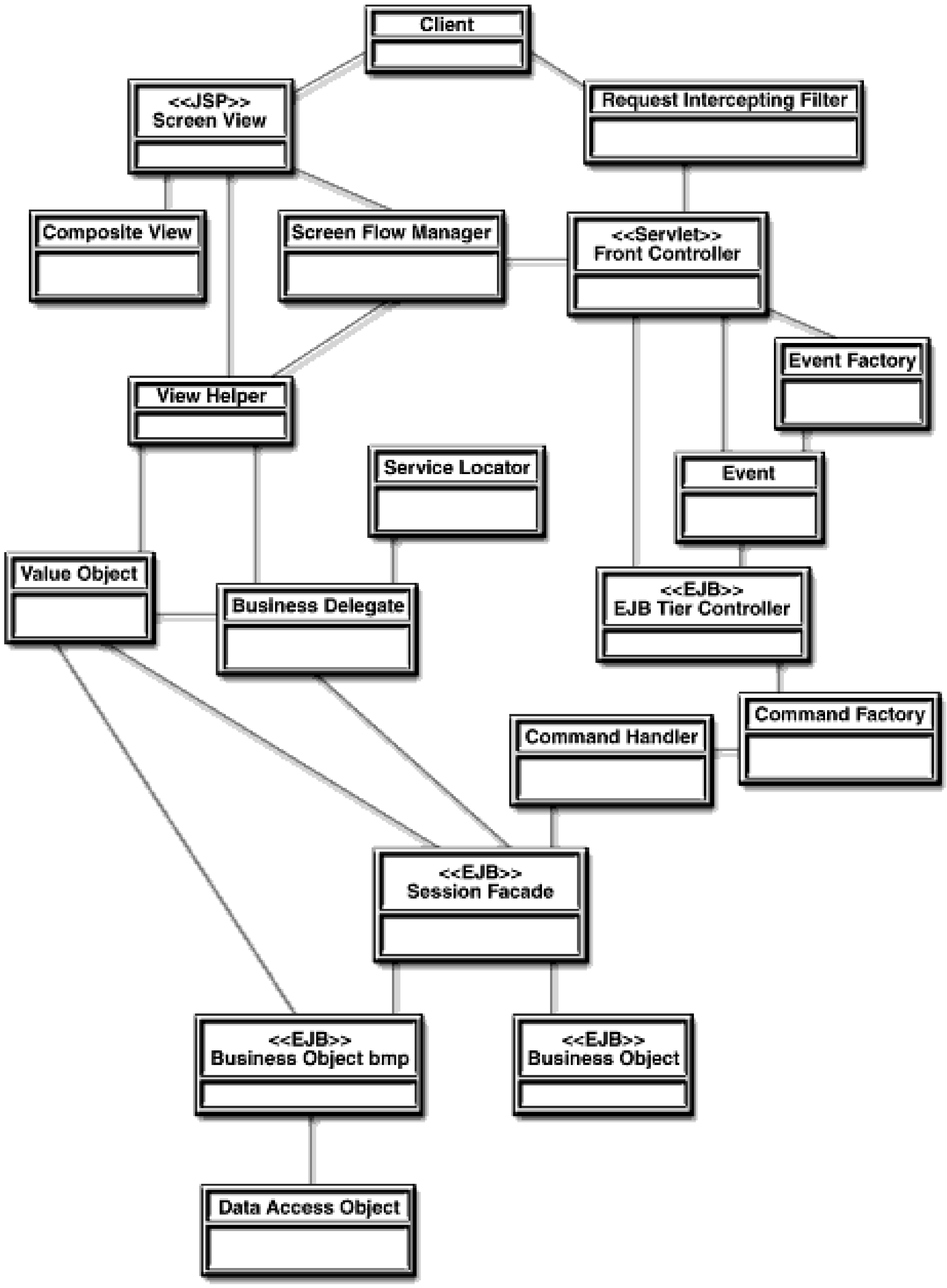
yes, the model layer can be used with any view layer like jsp / swing

Detailed Description

See Model-View-Controller explanation of this architectural pattern

Detailed Example

The architecture of the Java Pet Store sample application website applies the Model-View-Controller design pattern. Other design patterns are combined in the design of the MVC architecture.



Class Diagram Showing Sample Application Architectural Components

The architecture of the Java Pet Store website is described in more detail in the Java

BluePrints Program book [SSJ02]

~~Often, MVC functionality is captured in a framework that is reused by different applications. The sample application Web Application Framework is an extensible framework for creating MVC applications. It is an implementation of MVC to which new data sources, business logic, and data views may be added. The Web Application Framework design is described in more detail at **http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html**~~ .

17 - June -09

mostly we use this pattern for Centralized JNDI lookup for any J2EE resource and caching by using one Hashmap

Service Locator

Brief Description

Enterprise applications require a way to look up the service objects that provide access to distributed components. Java^{TM} 2 Platform, Enterprise Edition (J2EE) applications use Java Naming and Directory Interface (JNDI) to look up enterprise bean home interfaces, Java Message Service (JMS) components, data sources, connections, and connection factories. Repetitious lookup code makes code difficult to read and maintain. Furthermore, unnecessary JNDI initial context creation and service object lookups can can cause performance problems.

The ***Service Locator*** pattern centralizes distributed service object lookups, provides a centralized point of control, and may act as a cache that eliminates redundant lookups. It also encapsulates any vendor-specific features of the lookup process.

Detailed Description

See the **Core J2EE Patterns**

Detailed Example

The Java Pet Store sample application, v1.3.1 has two service locators: a Web-tier class ServiceLocator , and an Enterprise JavaBeans^{TM} (EJB) tier class, also called ServiceLocator . Both classes manage lookup and caching of enterprise bean home interfaces, JMS and database connection factories, and environment entries within their respective tiers. The only difference between them is that the Web-tier class is a singleton, and it caches the objects it looks up. The EJB-tier class is not a singleton, and does not cache.

The following code discussion uses examples from the Web-tier ServiceLocator :

- ***Clients use ServiceLocator to access services.***

  The sample application class AdminRequestBD is a business delegate that uses the Web-tier ServiceLocator to access the order processing center enterprise bean OPCAdminFacade . (See the Business Delegate design pattern for a more detailed description of AdminRequestBD.)

  Figure 1 is a structure diagram that demonstrates how AdminRequestBD uses ServiceLocator to find the remote home interface of the OPCAdminFacade enterprise bean. The ServiceLocator returns a the remote enterprise bean interface OPCAdminFacadeHome, by either retrieving it from the cache or looking it up using an internal InitialContext instance. The client then uses the OPCAdminFacade to find or create a remote component interface to an OPCAdminFacade

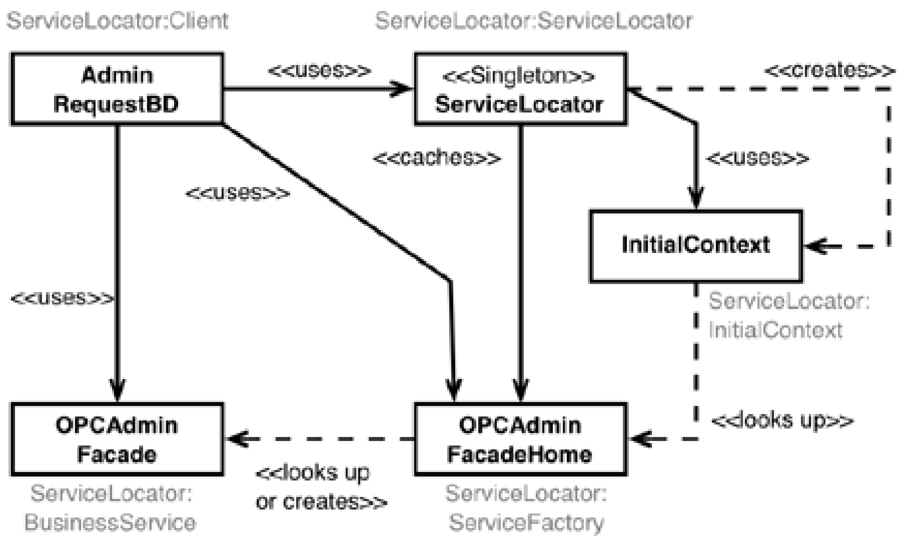


Figure 1. Structure diagram of ServiceLocator sample code

In the following code excerpt, AdminRequestBD calls the ServiceLocator static method getInstance to get the singleton instance of the service locator, then calls getRemoteHome to get the remote home interface of the OPCAdminFacade enterprise bean. Notice that the caller must typecast the remote home interface to OPCAdminFacadeHome because getRemoteHome returns type EJBHome .

```
public class AdminRequestBD {
...
  public AdminRequestBD() throws AdminBDException {
    try {
      OPCAdminFacadeHome home =
        (OPCAdminFacadeHome) ServiceLocator.getInstance().getRemoteHome(OPC_ADMIN_NAME, OPCAdminFacadeHome.class);
      opcAdminEJB = home.create();
    } catch (ServiceLocatorException sle) {
    ...
    }
}
```

The service locator greatly simplifies the lookup of the enterprise bean home interface. The singleton and caching strategies (discussed below) also improve performance, because they avoid constructing unnecessary InitialContext and enterprise bean home interfaces.

- ***Public methods look up distributed resources.***

  The public methods of the service locator look up distributed resources by their JNDI names. There are methods that find and return enterprise bean local home interfaces, JDBC^{TM} data sources, JMS queues and topics, and JMS queue and topic connection factories. There are also convenience methods that look up and perform type conversions on environment entries.

  As an example, method getLocalHome (for finding enterprise bean local home interfaces) appears below. Each method that locates a particular type of resource returns either a cached reference to the requested resource, or uses JNDI to find the resource, placing a reference in the cache before returning it.

```
// Enterprise bean lookups
public EJBLocalHome getLocalHome(String jndiHomeName)
throws ServiceLocatorException {
  EJBLocalHome home = null;
  try {
    if (cache.containsKey(jndiHomeName)) {
        home = (EJBLocalHome) cache.get(jndiHomeName);
    } else {
        home = (EJBLocalHome) ic.lookup(jndiHomeName);
        cache.put(jndiHomeName, home);
    }
   } catch (NamingException ne) {
        throw new ServiceLocatorException(ne);
   } catch (Exception e) {
        throw new ServiceLocatorException(e);
   }
   return home;
}
```

cache is achieved
by using hashmap

Methods that return enterprise bean home interface references are only type-safe to the platform interface level; for example, getLocalHome returns a EJBLocalHome, but the client must typecast the result.

~~Method getRemoteHome is similar to getLocalHome, except that it returns an enterprise bean remote, instead of local, home interface. It also requires a reference to a class object for the specific remote home interface, because remote home lookups use method PortableRemoteObject.narrow to perform the type conversion from the object returned from the JNDI lookup to the actual home interface type. The client that calls getRemoteHome must still typecast the result to the remote home interface type, as shown in the first example above.~~

```
public EJBHome getRemoteHome(String jndiHomeName, Class className)
throws ServiceLocatorException {
  EJBHome home = null;
  try {
    if (cache.containsKey(jndiHomeName)) {
        home = (EJBHome) cache.get(jndiHomeName);
    } else {
        Object objref = ic.lookup(jndiHomeName);
        Object obj = PortableRemoteObject.narrow(objref, className);
        home = (EJBHome)obj;
        cache.put(jndiHomeName, home);
    }
   } catch (NamingException ne) {
        throw new ServiceLocatorException(ne);
   } catch (Exception e) {
        throw new ServiceLocatorException(e);
   }

   return home;
}
```

As mentioned above, the service locator returns JMS resources, JDBC data sources, and performs type conversion on values in environment entries. The table below summarizes the names and return types of these methods.

Table 1. Additional ServiceLocator methods

Method Name	Return Type	Resource Type
getQueueConnectionFactory	QueueConnectionFactory	JMS
getQueue	Queue	JMS
getTopicConnectionFactory	TopicConnectionFactory	JMS
getTopic	Topic	JMS
getDataSource	DataSource	JDBC
getUrl	URL	env-entry
getBoolean	boolean	env-entry
getString	String	env-entry

to implement ServiceLocator pattern, **only one class enough** with set of public methods for each type. one Hashmap for cahing

- ***Improving performance with the Singleton pattern and caching.***

The Singleton pattern [GHJV95] ensures that only a single instance of a class exists in an application. The meaning of the term "singleton" is not always clear in a distributed environment; in ServiceLocator it means that only one instance of the class exists per class loader.

The Singleton pattern improves performance because it eliminates unnecessary construction of ServiceLocator objects, JNDI InitialContext objects, and enables caching (see below).

The Web-tier service locator also improves performance by caching the objects it finds. The cache lookup ensures that a JNDI lookup only occurs once for each name. Subsequent lookups come from the cache, which is typically much faster than a JNDI lookup.

no need think about thread even it is web application

The code excerpt below demonstrates how the ServiceLocator improves performance with the Singleton pattern and an object cache.

```
public class ServiceLocator {

    private InitialContext ic;
    private Map cache;

    private static ServiceLocator me;

    static {
      try {
        me = new ServiceLocator();
      } catch(ServiceLocatorException se) {
        System.err.println(se);
        se.printStackTrace(System.err);
      }
    }
   private ServiceLocator() throws ServiceLocatorException  {
      try {
        ic = new InitialContext();
        cache = Collections.synchronizedMap(new HashMap());
      } catch (NamingException ne) {
            throw new ServiceLocatorException(ne);
       }
    }
```

creation of this object can be postpone till first time call by null check in public method instead of using static block

it might be good very initially put into map.

```
    static public ServiceLocator getInstance() {
      return me;
    }
```

A private class variable me contains a reference to the only instance of the ServiceLocator class. It is constructed when the class is initialized in the static initialization block shown. The constructor initializes the instance by creating the JNDI InitialContext and the HashMap that is used a cache. Note that the no-argument constructor is private: only class ServiceLocator can construct a ServiceLocator. Because only the static initialization block creates the instance, there can be only one instance per class loader.

Classes that use service locator access the singleton ServiceLocator instance by calling public method getInstance.

Each object looked up has a JNDI name which, being unique, can be used as a cache HashMap key for the object. Note also that the HashMap used as a cache is synchronized so that it may be safely accessed from multiple threads that share the singleton instance.

18 - June - 09

to implement need only one SESSION bean.

Session Facade

Brief Description

Many business processes involve complex manipulations of business classes. Business classes often participate in multiple business processes or workflows. Complex processes that involve multiple business objects can lead to tight coupling between those classes, with a resulting decrease in flexibility and design clarity. Complex relationships between low-level business components make clients difficult to write.

so, either stateless or stateful session be act as SESSION FACEDE. mostly stateless session bean used.

The ***Session Facade*** pattern defines a higher-level business component that contains and centralizes complex interactions between lower-level business components. A Session Facade is implemented as a session enterprise bean. It provides clients with a single interface for the functionality of an application or application subset. It also decouples lower-level business components from one another, making designs more flexible and comprehensible.

Fine-grained access through remote interfaces is inadvisable because it increases network traffic and latency. The "before" diagram in Figure 1 below shows a sequence diagram of a client accessing fine-grained business objects through a remote interface. The multiple fine-grained calls create a great deal of network traffic, and performance suffers because of the high latency of the remote calls.

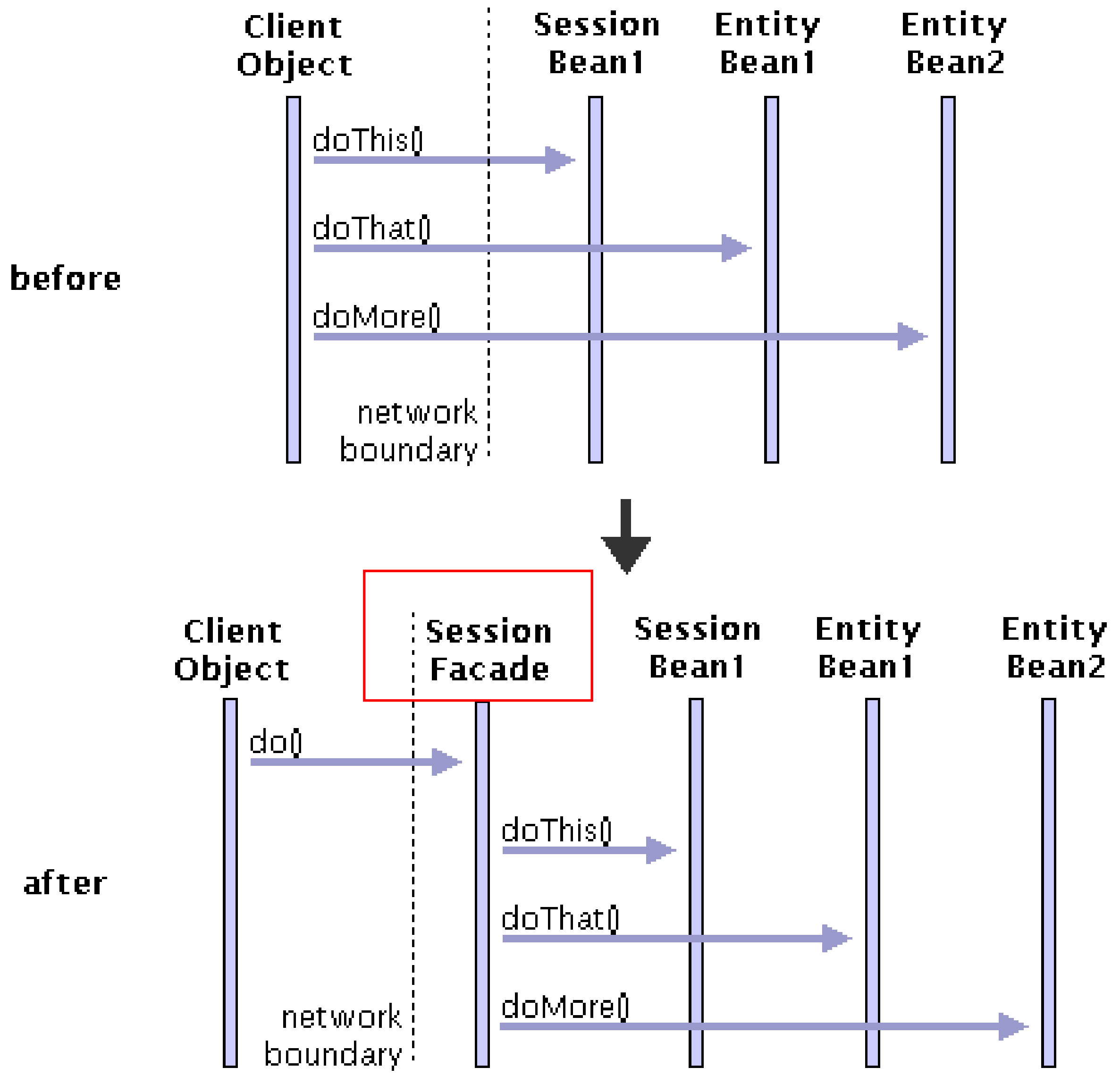


Figure 1. Sequence diagram before and after adding Session Facade

Introducing a Session Facade, as shown in the "after" diagram in Figure 1, decreases network traffic and latency, because all access to fine-grained business objects is local. The Session Facade also acts as a Mediator [GHJV95] between the business objects, decoupling their APIs from one another.

Detailed Description

See the **Core J2EE^{TM} Patterns**

Detailed Example

- ***The admin facade.***

The sample admin application of the Pet Store enterprise is responsible for management functionalities of the Pet Store order processing workflow. The admin application is a Swing client with Java^{TM} Web Start that communicates with the backend order processing center application to approve large purchase orders and view financial data. To interact with the order processing center application, the admin application uses the Session Facade pattern in the interface OPCAdminFacade , whose interface appears below.

```
public interface OPCAdminFacade extends EJBObject {

   public OrdersTO getOrdersByStatus(String status)
      throws RemoteException, OPCAdminFacadeException;

   public Map getChartInfo(String request,
                  Date start,
                  Date end,
                  String requestedCategory)
      throws RemoteException, OPCAdminFacadeException;
}
```

The admin facade's implementation class is OPCAdminFacadeEJB . The admin application is a client of the OPCAdminFacade in the order processing application. A structure diagram of how the Session Facade interacts with other classes in the order processing application appears in Figure 2 below. The greyed-out class in the diagram indicates the transfer object created by the facade class; see the Transfer Object pattern for details.

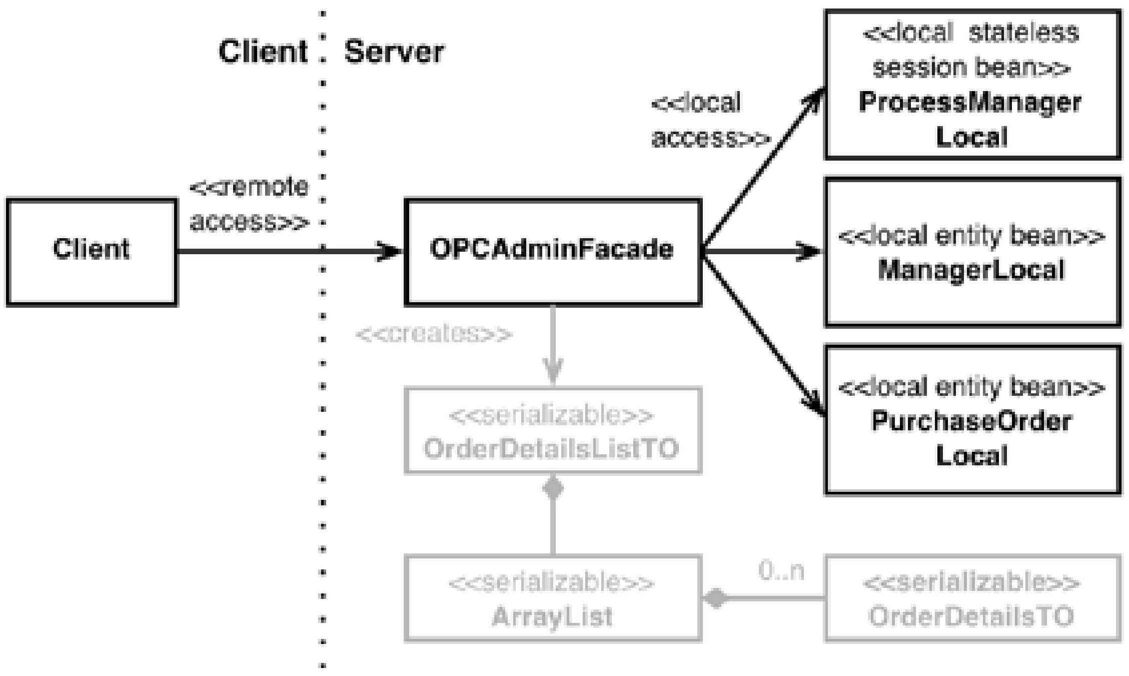

Client
Server
<<remote access>>
Client
OPCAdminFacade
<<local access>>
<<local stateless session bean>> ProcessManager Local
<<local entity bean>> ManagerLocal
<<local entity bean>> PurchaseOrder Local
<<creates>>
<<serializable>> OrderDetailsListTO
<<serializable>> ArrayList
0..n
<<serializable>> OrderDetailsTO

Figure 2. OPCAdminFacade provides an interface to complex interactions between components

The key point to notice here, is that the client only interacts with the OPCAdminFacade, and the other Enterprise Beans are hidden behind the facade. So the client has a simple interface and is not exposed to the complexity of the application logic that occurs to process the request. One of OPCAdminFacade's methods, getOrdersByStatus, builds a summary of purchase information for display by the sample application's admin client. This method uses a ProcessManagerLocal stateless session bean and ManagerLocal local entity beans and iterates over a collection of PurchaseOrderLocal entity beans of the requested status. The method creates a serializable Collection of OrderDetails transfer objects that describe PurchaseOrderLocal beans found. (See the Transfer Object pattern for details). The method then returns the collection of objects to the client.

- ***The Java Pet Store website sample application shopping facade.***
    - The Java Pet Store website application also uses the Session Facade pattern. ShoppingClientFacadeLocal is a local stateful session bean that centralizes the services that the Web site provides to shoppers. Its local interface appears below:

```
public interface ShoppingClientFacadeLocal extends EJBLocalObject {

   public ShoppingCartLocal getShoppingCart();
```

```
    public void setUserId(String userId);
    public String getUserId();
    public CustomerLocal getCustomer() throws FinderException;
    public CustomerLocal createCustomer(String userId);

}
```

so, either stateless or stateful session be act as SESSION FACEDE. mostly stateless session bean used.

The shopping facade provides a single interface to the client for all client shopping functionality: the shopping cart, storing the user's id, and finding and creating the Customer data associated with the user. The facade is a stateful session bean because it stores information (such as the user id) that is specific to an individual user in the context of a session.

18 - June - 09

Transfer Object

it is used to bring, a list of VALUE object AS IMMUTABLE either in list or attribute of one object in list.
.
it helps to reduce network traffic
.
the VALUE object in the list is not same as DOMAIN object, since domain object will have both setter and getter methods, where are value object of TO pattern must have only getter to achieve IMMUTABLE

Also Known As

Previously known as Value Object

Brief Description

it is best to use TO pattern in session facede pattern

Some entities contain a group of attributes that are always accessed together. Accessing these attributes in a fine-grained manner through a remote interface causes network traffic and high latency, and consumes server resources unnecessarily.

A ***transfer object*** is a serializable class that groups related attributes, forming a composite value. This class is used as the return type of a remote business method. Clients receive instances of this class by calling coarse-grained business methods, and then locally access the fine-grained values within the transfer object. Fetching multiple values in one server roundtrip decreases network traffic and minimizes latency and server resource usage.

this pattern used only getting RESPONSE, but not sending request.

Detailed Description

NEVER think to use TO pattern in SOA environment, since we cannot map interface or inner classes for a SOAP msg.

See the **Core J2EE™ Patterns**

Detailed Example

- ***The OrderDetails and OrdersTO transfer objects.***
  - Sample application class OPCAdminFacade (see Session Facade) has a method getChartInfo that returns an immutable, serializable collection of OrderDetails objects, each of which is a transfer object that represents data from an order. The collection returned by method getChartInfo is also a transfer object, because the values it contains are always accessed together where they are used. This hierarchical transfer object is implemented by class OrdersTO .

    Figure 1 shows the structure of the OrdersTO transfer object. Class MutableOrdersTO, which extends a serializable ArrayList, contains a collection of OrderDetails objects. But because MutableOrdersTO extends ArrayList, it is mutable. Transfer objects should be immutable so that clients do not unintentionally change their contents. Interface OrdersTO adapts the MutableOrdersTO collection, allowing read-only access to the collection while preventing modifications.

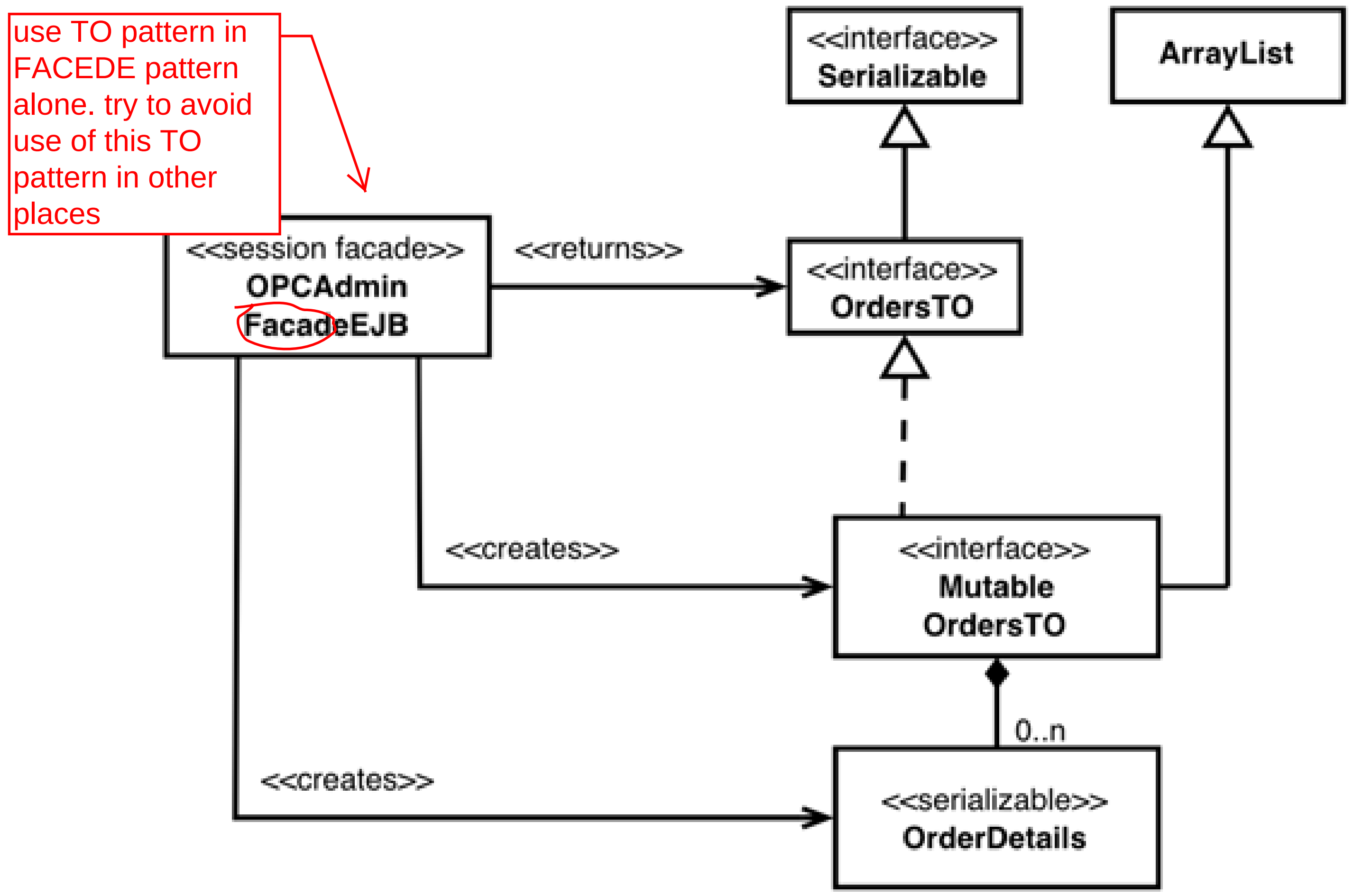

use TO pattern in FACEDE pattern alone. try to avoid use of this TO pattern in other places
<<interface>> Serializable
ArrayList
<<session facade>> OPCAdmin FacadeEJB
<<returns>>
<<interface>> OrdersTO
<<creates>>
<<interface>> Mutable OrdersTO
0..n
<<creates>>
<<serializable>> OrderDetails

Figure 1. Transfer object OrdersTO is an immutable, serializable collection

The definition of the OrdersTO interface follows.

```
public interface OrdersTO extends Serializable {

  public Iterator iterator();
  public int size();
  public boolean contains(Object o);
  public boolean containsAll(Collection c);
  public boolean equals(Object o);
  public int hashCode();
  public boolean isEmpty();
  public Object[] toArray();
  public Object[] toArray(Object[] a);

  static class MutableOrdersTO extends ArrayList implements OrdersTO {
  }
}
```

TO has declared most of ArrayList methods except setting value methods, to provide immutable

static class inside of interface definition.

this is best design pattern to expose a set of function alone without writting any copy or paste

The OrdersTO interface is clearly a collection, since it defines the Java collection methods. The collection is immutable, because it has no methods that would allow the collection contents to be changed.

An example of using OrdersTO appears in method OPCAdminFacadeEJB.getOrdersByStatus, shown below. The method constructs a OrdersTO.MutableOrdersTO object and populates it with OrderDetails instances. Note that the return type of getOrdersByStatus is OrdersTO, so callers can access only methods that read the collection contents, and not methods that would change them.

actually it is arraylist

```
public OrdersTO getOrdersByStatus(String status)
  throws OPCAdminFacadeException {

   OrdersTO.MutableOrdersTO retVal = new OrdersTO.MutableOrdersTO();
   PurchaseOrderLocal po;
   ProcessManagerLocal mgr = getProcMgr();

   try {
      PurchaseOrderLocalHome pohome = getPO();
      Collection orders = mgr.getOrdersByStatus(status);
      Iterator it = orders.iterator();
      while((it!= null) && (it.hasNext())) {
         ... // Access and format data
         retVal.add(new OrderDetails(po.getPoId(), po.getPoUserId(),
                              podate, po.getPoValue(), status));
      }
   } catch (FinderException fe) {
     ... // process exception
   }
   return(retVal);
}
```

- The details for a single order are contained in an immutable, serializable transfer object of type OrderDetails. It is a typical transfer object for a single composite value, containing private fields and public, read-only property accessors, as shown in the following code sample.

value can be passed only via constructor

```
public class OrderDetails implements java.io.Serializable {

   private String orderId;
   private String userId;
```

```
    private String orderDate;
    private float orderValue;
    private String orderStatus;

    public OrderDetails(String oid, String uid, String date, float value,
                String stat) {
      orderId = oid;
      userId = uid;
      orderDate = date;
      orderValue = value;
      orderStatus = stat;
    }

    public String getOrderId() {
      return(orderId);
    }

    // ...
  }
```

18 - June - 09

it is primarily used when pagination for list of values as a read only.
.
to implement only one class enough
that class keep track of current page, no. of record for a page, etc.
.
this pattern work with DAO and/or Fast lane reader.
.
it helps for performance improvement

Value List Handler

Brief Description

Web applications frequently allow users to browse large virtual lists, such as query result sets, which can not practically be transmitted to the client. Access to the list is usually read-only and bi-directional, and the list is often discarded after the first few elements are examined. Using entity beans to represent elements of such lists provides little benefit and drains system resources. In particular, finder methods do not provide caching, scrolling, and random access to result sets, and have limited selection capabilities.

The ***Value List Handler*** design pattern provides a more efficient way to iterate a large, read-only list across tiers. A value list handler provides a client with an iterator for a virtual list that resides in another application tier. The iterator typically accesses a local ordered collection of Transfer Objects, representing a subrange of the large list. A Data Access Object usually manages access to the list data, and may provide caching.

Detailed Description

See the **Core J2EE^{TM} Patterns**

Detailed Example

The sample application catalog is an ordered collection of categories. Each category contains an ordered collection of products, and each product contains an ordered list of items. The catalog can also be searched for items that meet specific criteria.

Because it would be impractical to download the entire catalog each time the user selected a category, product, or item, or performed a search, the sample application manages catalog searching and browsing with the Value List Handler design pattern. The client requests information a page at a time by indicating where in the collection to begin and how many items to return. The client uses the value list handler as an iterator over the resulting collection and displays the result list in the user interface.

Figure 1 below shows a structure diagram of the sample application value list handler CatalogHelper.

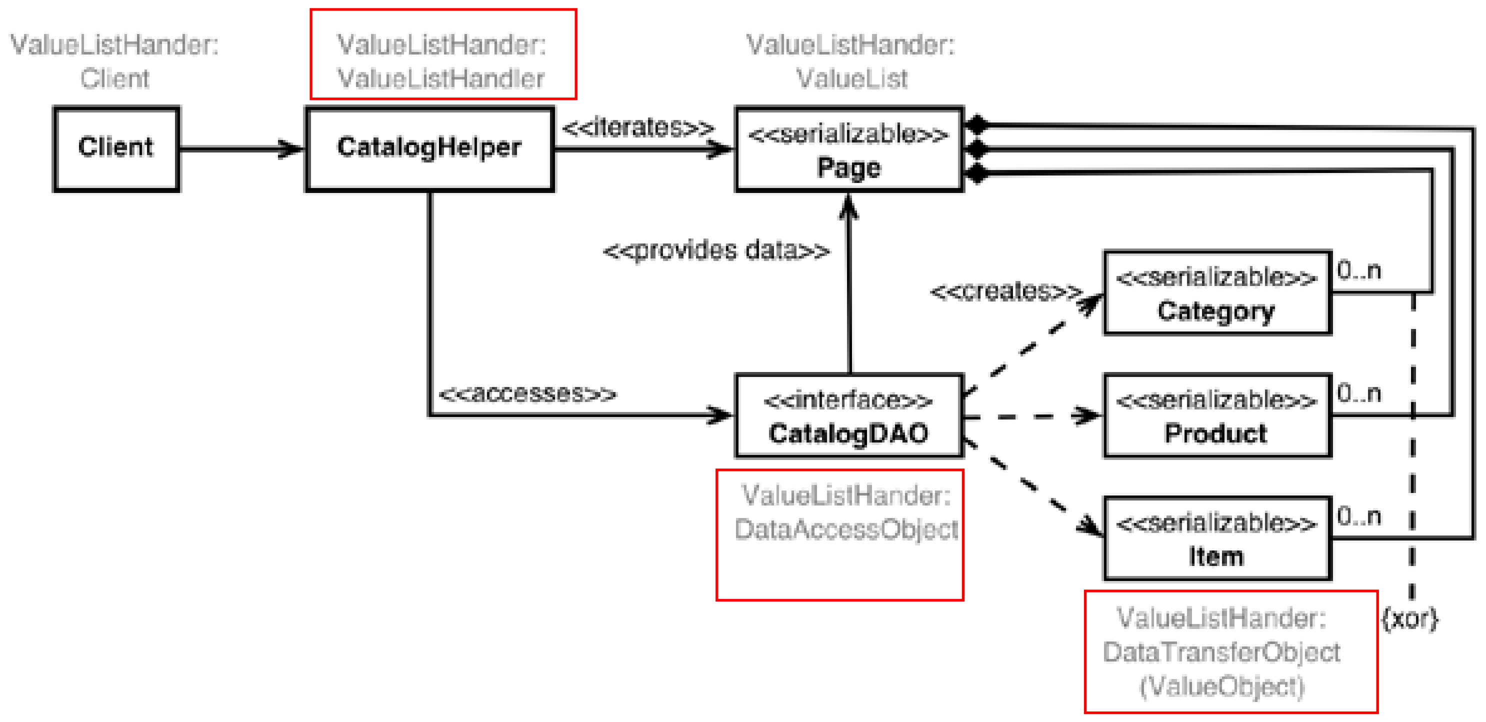


Figure 1. Sample application CatalogHelper structure diagram

The CatalogHelper acts and as a value list handler for the client. The Client shown in Figure 1 is a JavaServer Pages^{TM} (JSP^{TM}) page, accessed through a Web browser, that accesses an instance of CatalogClientHelper using a useBean tag, as shown in the following excerpt from category.jsp:

```
<jsp:useBean
  id="catalog"
  class="com.sun.j2ee.blueprints.catalog.client.CatalogHelper"
  scope="session"
/>
```

A CatalogHelper instance, maintained in an HTTP session attribute, keeps track of:

1. the current query being executed (if the page represents a search)
2. the start position the list of elements being requested
3. the number of list items to retrieve at a time
4. the locale to be used for displaying the items

The CatalogHelper delegates responsibility for data access to an object that implements interface CatalogDAO (see the Data Access Object design pattern for details). The CatalogDAO retrieves a range of categories, products, items, or query results from the catalog, and returns them as a Page object. A Page is a transfer object that is a collection of other transfer objects. It represents the list of requested elements from the catalog (categories, products, or items). The CatalogHelper returns a Page of catalog entries (built by the CatalogDAO) to the client for display.

Figure 2 is a collaboration diagram of how the CatalogHelper uses a CatalogDAO to access a subrange of query results. Following the figure is a description of the method call sequence.

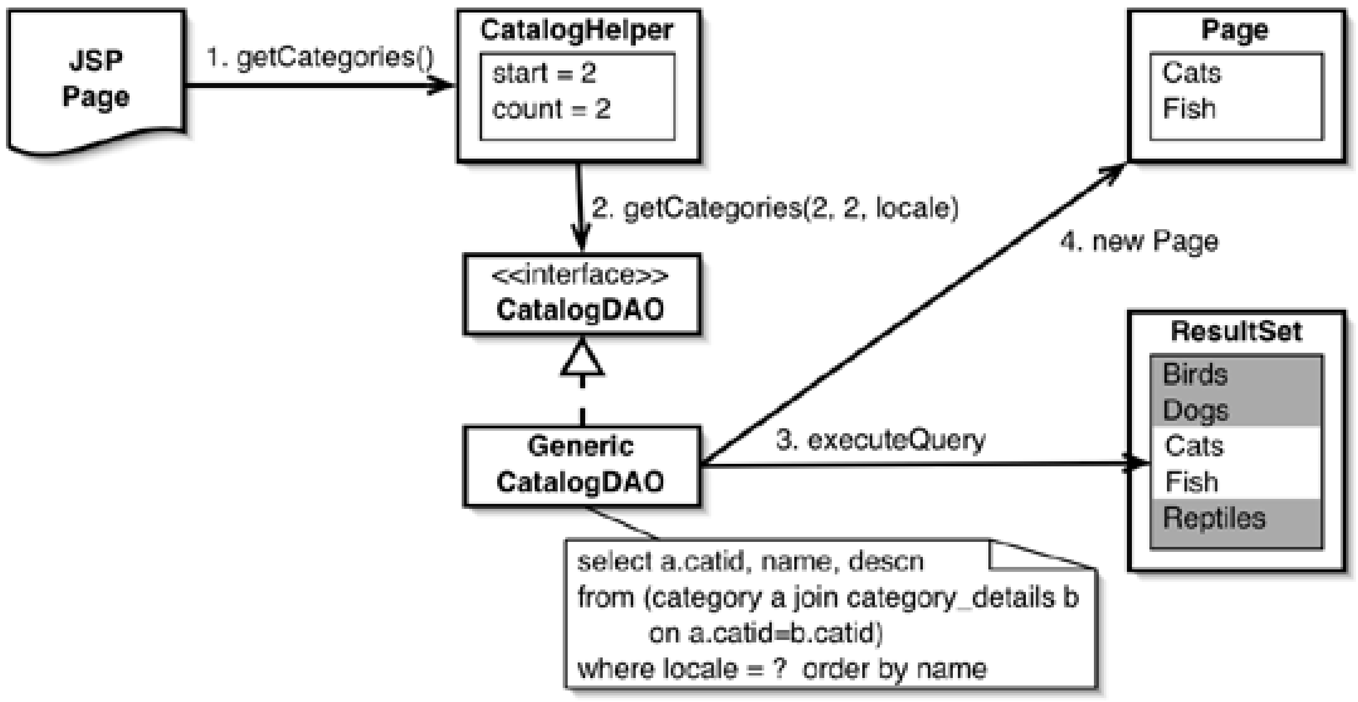


Figure 2. Collaboration diagram for value list handler CatalogHelper

- 1. A tag in the JSP page requests categories from the CatalogHelper. The tag is from the JavaServer Pages^TM Standard Tag Library (JSTL)
- 2. The CatalogHelper, which is keeping track of the current list position, size, and locale, requests a list of categories from CatalogDAO.getCategories.
- 3. The CatalogDAO, which in this scenario is implemented by class GenericCatalogDAO, executes the query shown, producing a ResultSet.
- 4. The GenericCatalogDAO then constructs a new Page object that contains the requested subrange of the larger list in the ResultSet. The Page is returned to the CatalogHelper, where it is used for display by the JSP page.

This implementation illustrates several interesting points about the Value List Handler pattern:

- ***Using a Data Access Object maximizes flexibility.*** The Data Access Object pattern used by the Value List Handler is an abstract API that encapsulates and hides the data access mechanism. At runtime, the sample application uses a factory object to select and load the class that implements interface CatalogDAO. This allows the data access mechanism to change--and new mechanisms to be added -- with no changes to client code. See the **Data Access Object** pattern description for further details.

- ***The CatalogHelper may or may not use the Fast Lane Reader design pattern.*** The CatalogHelper class has a property, useFastLane, that indicates whether to access the catalog directly in the EIS tier, instead of accessing it through the Enterprise JavaBeans^{TM} (EJB^{TM}) tier. When useFastLane is true, the CatalogHelper access the catalog using an object that implements interface CatalogDAO. When useFastLane is false, it accesses the catalog using a CatalogLocal enterprise bean. See the **Fast Lane Reader** design pattern for details.

- ***A value list handler may provide an internal or external iterator.*** The CatalogHelper provides an external iterator interface because it keeps track of the catalog page size and position. The iterator could also have been internal, providing only nextPage and prevPage methods while tracking position and size internally. (See *Design Patterns* for a description of internal and external iterators [GHJV95]). The choice of an internal or external iterator interface is a design decision that depends on application requirements.

- ***Caching can improve performance.*** Application performance can often be improved by caching value lists. The sample application could have cached catalog lists, but this data is not cached in the current implementation. The CatalogHelper could improve performance by maintaining a cache of previously-requested lists. However, the Java Pet Store does cache page views using a custom tag CacheTag.

- ***The Value List Handler pattern is appropriate for both remote and local architectures.*** While the Value List Handler pattern was originally created to maximize performance in remote architectures, it is appropriate for local architectures as well. Modeling pages of information as sublists of a larger virtual list is a natural design concept, and it leads to a clean implementation. Where there are many repeated lookups of the same information, value lists are natural candidates for cache objects. Finally, the Value List Handler pattern provides more flexible query functionality than does using an EJB finder method. All of these benefits apply to both local and remote architectures.

Enterprise BluePrints

Java BluePrints > Enterprise > Patte

Guidelines, Patterns, and code for end-to-end Java

Java BluePrints > Enterprise >

View Helper

Brief Description

Business classes change frequently, and application views change ev
logic complicates maintenance and reuse.

A View Helper is a class that does data retrieval for the view. It adapts
Helper pattern decouples business and application classes from one a
promotes reuse, because each business or presentation component h
presentation logic, and let the View Helper handle the processing and

Detailed Description

See the **`Core J2EE Patterns`**

Detailed Example

The Java BluePrints Program recommendation for presentation on bro
Two common strategies for implementing View Helpers is the JavaBea
offer technologies that are useful for implementing View Helper: the u
Helper called `CatalogHelper` as a view helper for several view
to adapt the `CatalogHelper` methods to the needs of JSP pages

- Using the `CatalogHelper` View Helper from a JSP page.

  The class `CatalogHelper` handles and hides the complexi
  simplified interface to its clients. The `CatalogHelper` is an

  - The sequence diagram in Figure 1 shows how the View He
    a JSP page class, generated from the JSP page source co
    the `CatalogHelper` in two ways: directly, using the u

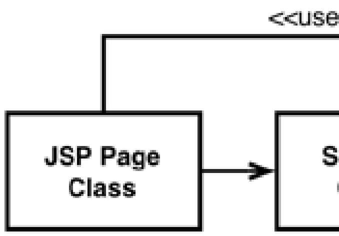


Figure 1. Structure diagra

- Sample application view component `category.jsp` defines
  as follows:

```
<jsp:useBean
  id="catalog"
  class="com.sun.j2ee.blueprints.catalog.cli
  scope="session"
/>
```

- The tag above defines an `HttpSession` attribute `catalo`
  called `pageResults`, which is the result of a call to `Catal`

```
<c:choose>
 <c:when test="${param.count != null}">
  <c:set value="${param.start}" target="${ca
  <c:set value="${param.count}" target="${ca
 </c:when>
 <c:otherwise>
  <c:set value="0" target="${catalog}" prope
  <c:set value="2" target="${catalog}" prope
 </c:otherwise>
</c:choose>
<c:set value="en_US" target="${catalog}" pro
<c:set value="${param.category_id}" target="
<c:set value="${catalog.products}" var="page
```

- The block of tags above initializes method call arguments for the attribute `catalog.products` in the last line). It then invok variable called `pageResults`. The `CatalogHelper` ma data from the data source, and encapsulating those data as a `Pa` the code sample below.

```
Page getProductsFromDAO(String categoryId, in
        throws CatalogClientException {
        try {
            if (dao == null)
                dao = CatalogDAOFactory.getDA
            return dao.getProducts(categoryId
                                        locale);
        }
        catch (CatalogDAOSysException se) {
            System.out.println("Exception rea
            throw new CatalogClientException(
        }
    }
```

- The DAO code above gets the products directly using a JDBC qu

```
public Page getProducts(String categoryID, in
                            int count, Locale l)
  throws CatalogDAOSysException {

    Connection c = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    Page ret = null;

    try {
        c = getDataSource().getConnection();

        // Select
        ps = c.prepareStatement("select a.pro
                                + "from (proc
                                + "product_de
                                + "a.producti
                                + "where loca
                                + "and a.cati
                                + "order by n
                                ResultSet.TYP
                                ResultSet.CON
        ps.setString(1, l.toString());
        ps.setString(2, categoryID);
        rs = ps.executeQuery();
        // ... and so on ...
```

- Finally, again in `products.jsp`, the following block of tags
  and kept in session scope). The JSTL tags (in XML namespace
  the resulting page:

```
<c:forEach var="item" items="${pageResults.l
<tr>
 <td class="petstore_listing">
    <c:url value="/product.screen" var="prod
     <c:param name="product_id" value="${ite
    </c:url>
    <a href='<c:out value="${productURL}"/>'
    <c:out value="${item.name}"/>
   </a>
   <br>
   <c:out value="${item.description}"/>
  </td>
 </tr>
</c:forEach>
```

The key point to understand in this example is that all of the code that
in scriptlets in the JSP page.

Also, note that the View Helper pattern may be combined with other p
Service Locator to help manage the data access. The Java Pet Store
with other patterns as part of its overall design.