Using J2EE Design Patterns

Introduction

May 2002

OTN developers have implemented the Virtual Shopping Mall (VSM), a sample application that demonstrates design patterns for building J2EE applications. A software *design pattern* describes an approach to solving a recurring programming problem or performing a task. The design patterns discussed in this article can improve the performance of J2EE applications over the Internet or an intranet, and make them more flexible and easier to maintain.

Documents

- ***About the VSM Sample Application*** (HTML)
- ***Design Patterns in the VSM*** (HTML)

The following links provide background information that will help you understand this sample application.

- J2EE Design Patterns
- EJB Design Patterns
- Simplifying J2EE and EJB Development with BC4J

Download

- The VSM sample application download (JAR file, 553 KB) includes installation instructions and complete source code.
- View the source code online.
- View an online demo of the VSM in action.

Using J2EE Design Patterns

About the VSM Sample Application

The Virtual Shopping Mall (VSM) sample application enables vendors to set up online shops, customers to browse through the shops, and a system administrator to approve and reject requests for new shops and maintain lists of shop categories.

About VSM Users

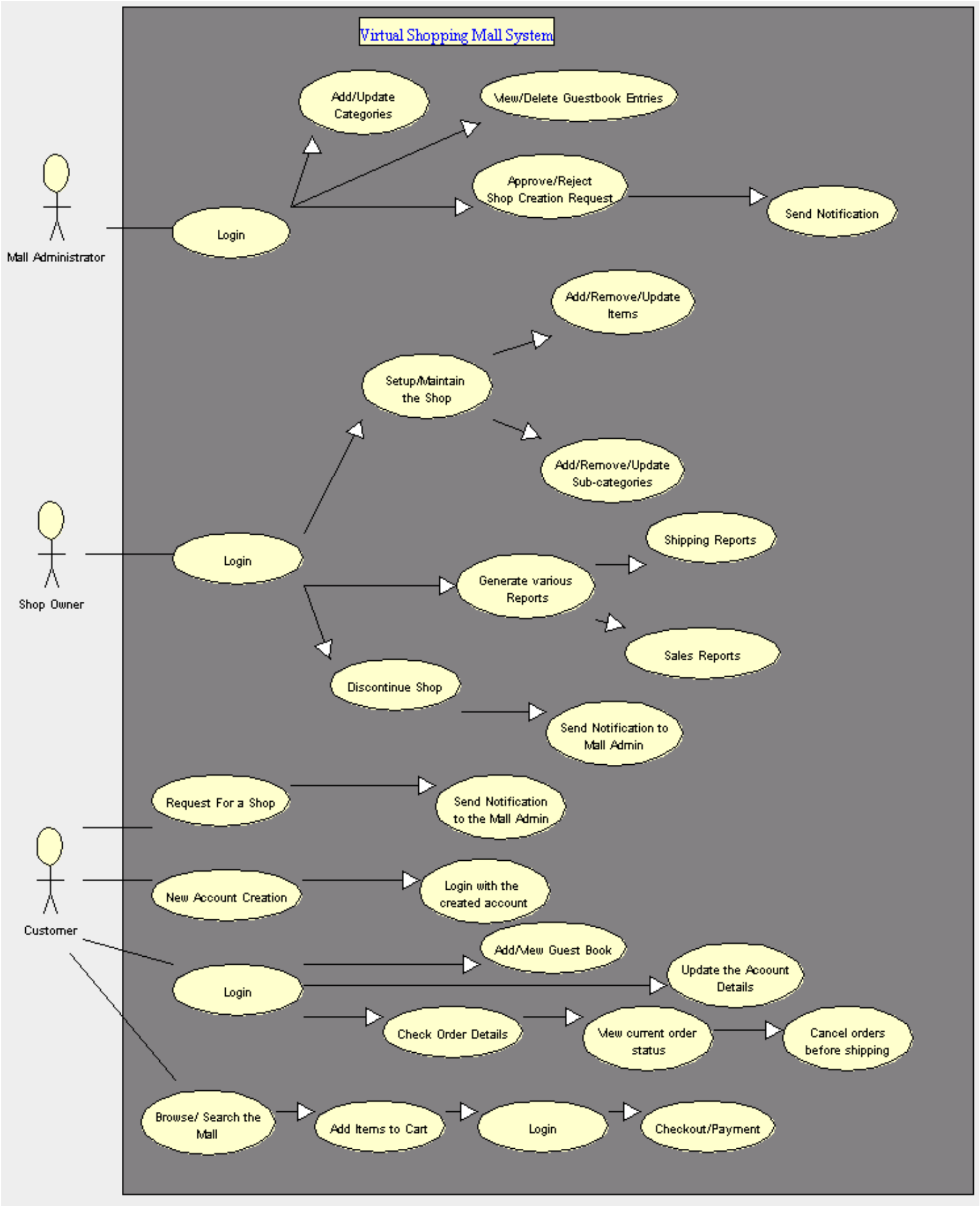
The application identifies three types of users—Mall Administrator, Shop Owner and Mall Customer—each with different privileges.

Mall Administrator The Mall Administrator is the superuser and has complete control over all the activities that can be performed. The application notifies the administrator of all shop creation requests, and the adminstrator can then approve or reject them. The administrator also manages the list of available product categories. The administrator can also view and delete entries in the guestbook.

Shop Owner Any user can submit a shop creation request through the application. When the request is approved by the Mall Administrator, the requester is notified, and from there on is given the role of Shop Owner. The Shop Owner is responsible for setting up the shop and maintaining it. The job involves managing the sub-categories of the items in the shop. Also, the shop owner can add or remove items from his shop. The Shop Owner can view different reports that give details of the sales and orders specific to his shop. The Shop Owner can also decide to close shop and remove it from the mall.

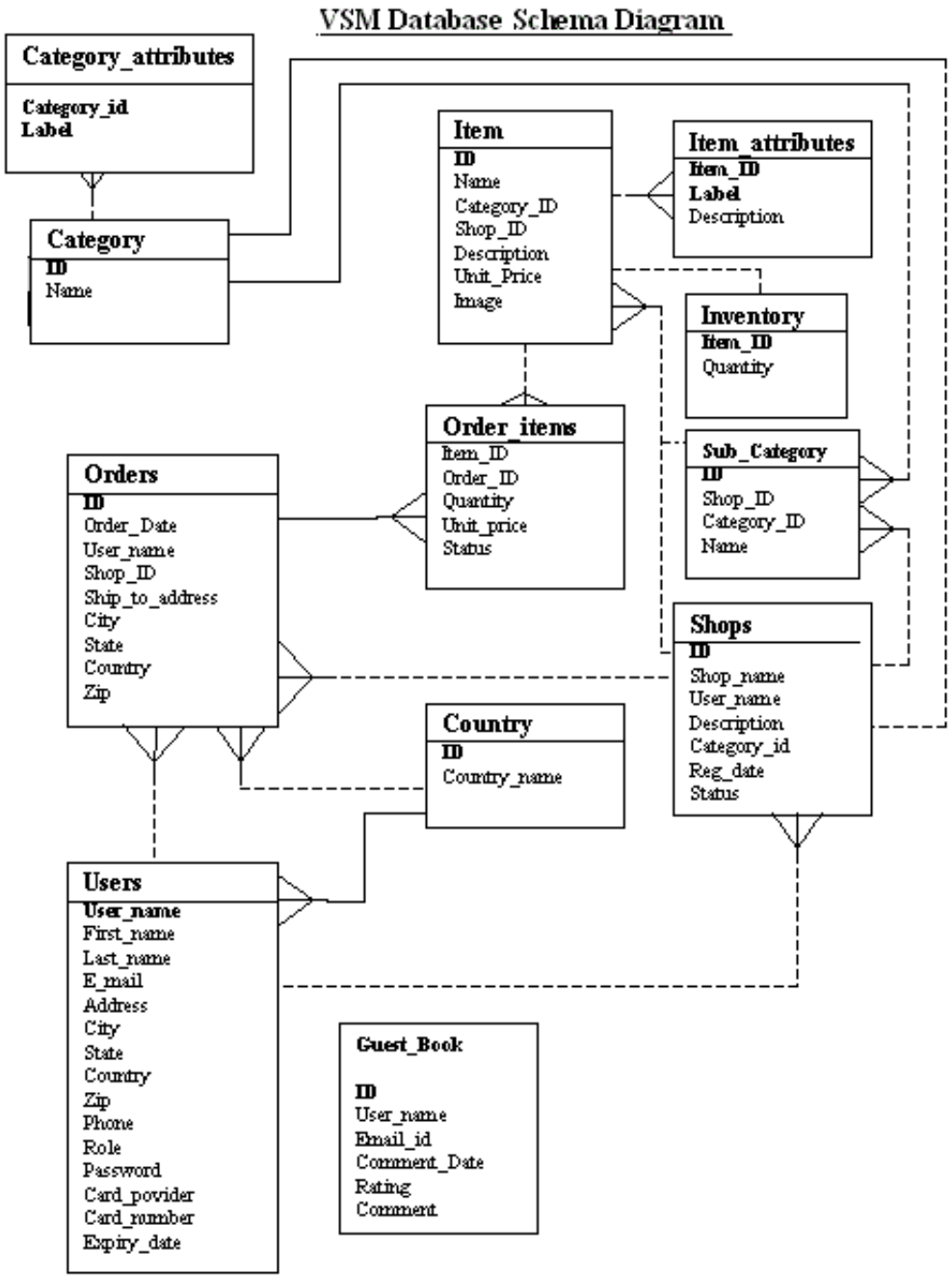
Mall Customer A Mall Customer can browse through the shops and choose products to place in a virtual shopping cart. The shopping cart details can be viewed and items can be removed from the cart. To proceed with the purchase, the customer is prompted to login. Also, the customer can modify personal profile information (such as phone number and shipping address) stored by the application. The customer can also view the status of any previous orders, and cancel any order that has not been shipped yet.

The following use-case diagram shows how each type of user interacts with the VSM application.


Virtual Shopping Mall System
Add/Update Categories
View/Delete Guestbook Entries
Approve/Reject Shop Creation Request
Send Notification
Mall Administrator
Login
Add/Remove/Update Items
Setup/Maintain the Shop
Add/Remove/Update Sub-categories
Shop Owner
Login
Shipping Reports
Generate various Reports
Sales Reports
Discontinue Shop
Send Notification to Mall Admin
Request For a Shop
Send Notification to the Mall Admin
Customer
New Account Creation
Login with the created account
Add/View Guest Book
Login
Update the Acoount Details
Check Order Details
View current order status
Cancel orders before shipping
Browse/ Search the Mall
Add Items to Cart
Login
Checkout/Payment

Database Schema

The figure below shows the database schema for the VSM application.



Conclusion

The Virtual Shopping Mall (VSM) sample application shows how design patterns can simplify enterprise development tasks, and demonstrates approaches you can use to implement common J2EE design patterns. If you're interested in other approaches, you can visit OTN to learn how Oracle's J2EE-compliant Business Components for Java (BC4J) framework provides off-the-shelf implementations of numerous design patterns you would otherwise have to code by hand. The article Simplifying J2EE and EJB Development with BC4J provides more information.

< Back to Introduction	Design Patterns in the VSM >

Questions or comments? Post a message in the OTN Sample Code discussion forum or send email to the author.

Using J2EE Design Patterns: About the Virtual Shopping Mall Sample Application
Author: Robert Hall, Oracle Corporation
Date: May 2002

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7200

Using J2EE Design Patterns

Design Patterns in the VSM

The VSM sample application demonstrates ways to implement the following design patterns.

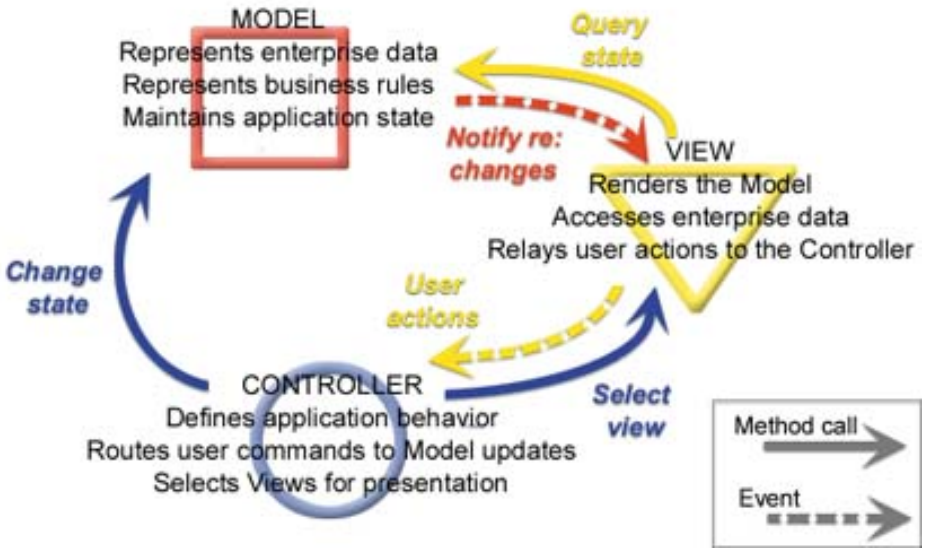
- Model-View-Controller
- Command Façade
- Session Façade
- Value Object
- Message Façade
- Service Locator

MVC (Model–View–Controller)

The MVC design pattern differs from the others discussed in this article, which describe ways to solve specific programming problems or perform specific tasks, because the MVC design pattern describes an approach to an application as a whole. The approach is multi-tiered and modular; the MVC design pattern identifies three entities, each operating in a different logical layer within the application space:

- **Model**: A back-end representation of enterprise data and business logic, the model also maintains data about the state of the application. In many cases, the model is a logical representation of a real-world process.
- **View**: The front-end presentation layer that renders the model for end-users. Views also provide user interfaces for accessing the enterprise data represented by the model.
- **Controller**: The middle tier of the MVC pattern. The controller defines application behavior, selecting views for presentation and capturing a user's interactions with views and routing them as commands to the model.

The MVC design pattern defines a clear separation of application logic, presentation details, business rules and data. As a result, multiple clients, displays, and devices can use the same application and business rules to work with the same data. The following figure shows how model, view, and controller interact.



- The *model* is implemented by EJBs and other Java classes, many of which represent real-world objects. such as shops, shopping carts, and orders. For implementation examples, see `Shops.java`, `ShopsBean.java`, and `ShopsHome.java`.
- The *views* are provided by JavaServer Pages (JSPs) rendered in a browser. Examples include `shoppingMall.jsp`, `mallAdmin.jsp`, and `cart.jsp`.
- The central *controller* is implemented in `RouterServlet.java`. When an end-user interacts with a view (for example, by submitting a form), this HTTP servlet dispatches the action to a controller object, such as `LoginController` or`CartController`, as appropriate, which in turn invoke methods on objects in the model.

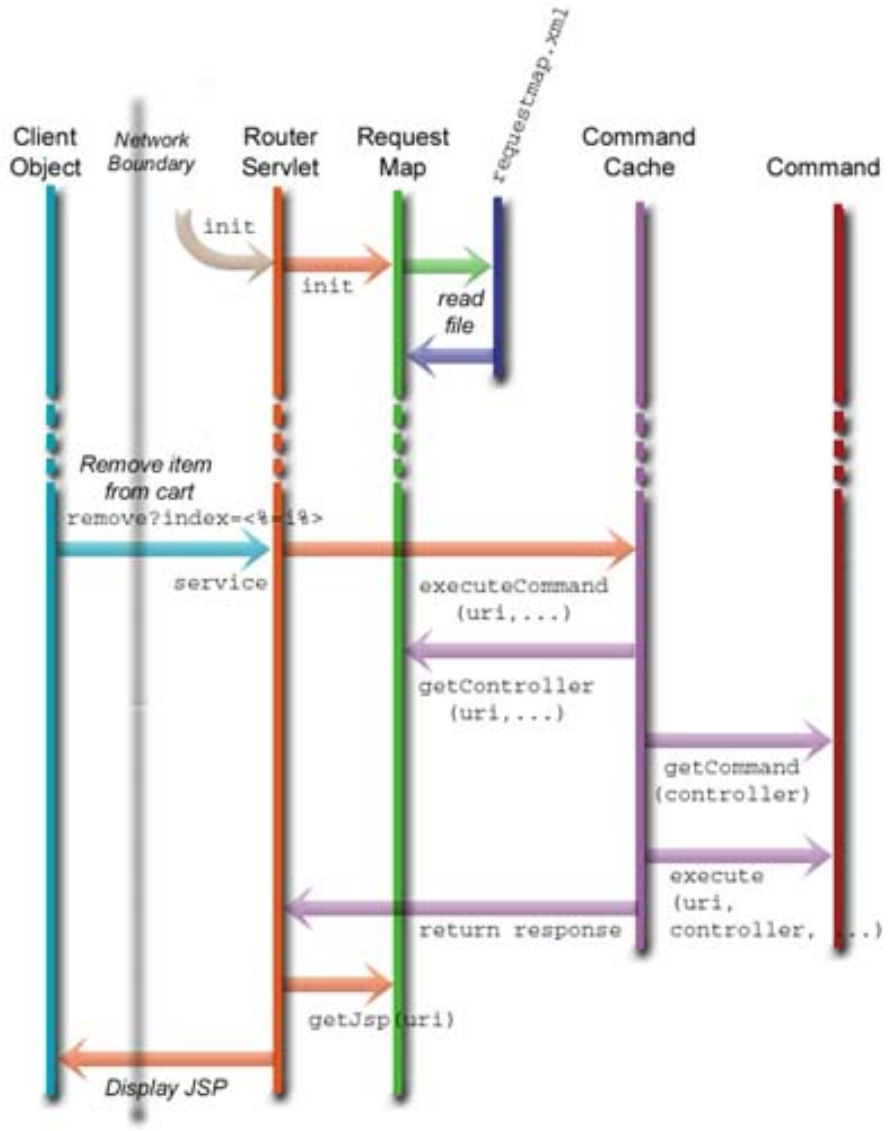
The process of dispatching and executing commands is itself captured in a design pattern, Command Façade, discussed in the next section.

Command Façade

The Command Façade design pattern describes a way to work with method calls as objects. Without the Command Façade design pattern, a client must find an object, hold a reference to it, then use that reference to call a method. Using the Command Façade design pattern, a client issues commands by name, and a command object dispatches each command to a method call on an underlying EJB. This encapsulation of commands presents a consistent interface to clients while hiding implementation details--developers are free to change the back-end implementation (model or controller) without fear of breaking the presentation layer (view). Also, commands are cached in a history list so that they can be reused, improving performance.

The VSM implements this design pattern in several files, including `RouterServlet.java`, `Command.java`, and `CommandCache.java`. When the main VSM controller (`RouterServlet`) receives a request from a view, its `service` method executes and invokes the Java class `RequestMap` to load the values from `requestmap.xml` into a Hashtable, thereby mapping the URI to a `Controller` class. The `RequestMap` class maintains these values in a Hashtable so that subsequent calls can be made from memory instead of reading the properties file each time.

The following figure shows two event sequences. First, it shows what happens when the `RouterServlet` is initialized: it initializes the `RequestMap` class, which in turn reads mappings from `requestmap.xml`. The second sequence shows what happens later in the application life cycle when an end-user sends a command (in this case, to remove an item from the shopping cart).



Here is a portion of `requestmap.xml`, which maps URIs to methods. This example maps the `/remove` URI to the `removeItem` method of the `CartController` class, and specifies `cart.jsp` as the page to display for the resulting view.

```
<?xml version="1.0"?>
<mappings>
  ...
    <uri-mapping>
        <uri>/remove</uri>
        <jsp-template>jsps/mallUser/cart.jsp</jsp-template>
        <controller class="oracle.otnsamples.vsm.controllers.CartController"
                    method="removeItem"/>
        <security>
            <authenticate>false</authenticate>
            <invalidate-session>false</invalidate-session>
        </security>
    </uri-mapping>
  ...
</mappings>
```

The following code comes from `Command.execute`. Given a URI submitted with a client request, this code searches the request map for the corresponding controller and method, then invokes the method on that controller. Then it returns the response to the `RouterServlet` which parses it to find out which JSP to invoke for the next client view.

```
public class Command {
...
public UserResponse execute( String uri,
                                    Hashtable request,
                               UserSession session,
                                    String controller )
                               throws InvalidURIException, BusinessException {
    // Initialize Response object
    UserResponse response = null;
    try {
      // Provides access to the required method on the Controller Class
      Method methodObject = null;
      // Create the controller object only if not created previously
      if( baseController == null ) {
        baseController = (BaseController)Class.forName( controller ).newInstance();
      }
      // Check the cache for method object
      if( uriMethodMap.containsKey( uri ) ) {
        methodObject = (Method)uriMethodMap.get( uri );
      } else {
        // An array of Class objects that identify the method's formal
        // parameter types in declared order
        Class[] paramTypes = new Class[] {
          Hashtable.class, UserSession.class
        };
        // Get the method name from the RequestMap and instantiate the method
        String methodName = RequestMap.getInstance().getMethod( uri );
        methodObject = baseController.getClass().getMethod(methodName, paramTypes);
        // Cache this method object
        uriMethodMap.put( uri, methodObject );
      }
      // Create an object array of the parameters to be passed to the method
      Object[] params =  {
        request, session
      };
      // Execute the method on the specified controller object using reflection
      response = (UserResponse)methodObject.invoke( baseController, params );
    } catch ...
    }
    // Return the response from the Controller
    return response;
  }
...
}
```

Session Façade

The Session Façade design pattern is useful in situations where client objects need to interact with a set of EJBs to perform tasks in a workflow. For example, in the VSM environment, customers browse shops and order products, shop owners track orders and maintain inventory, and adminstrators approve and reject requests for new shops and manage category lists. In an implementation where client objects interact directly with the underlying EJBs, the following problems can arise:

- When an EJB's interface changes, client objects must also be updated. This situation is analogous to the brittle class problem common in object-oriented programming.
- To carry out a workflow, client objects must make numerous remote calls to access the EJBs, leading to increased network traffic and reduced performance.

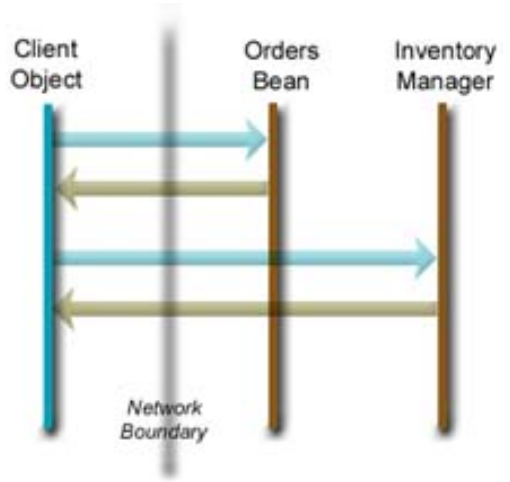
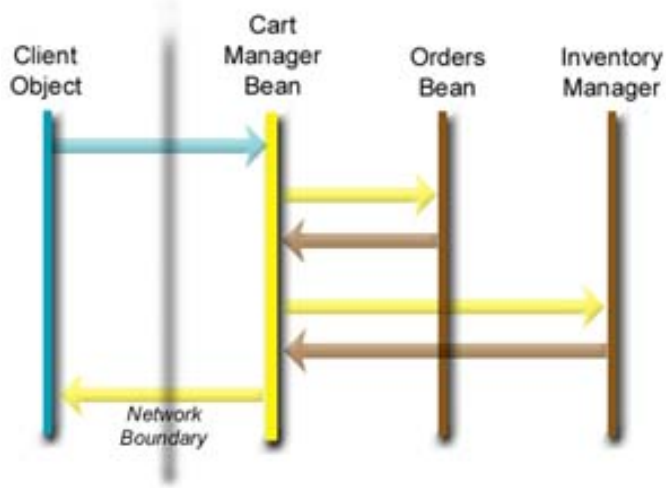
A session façade solves such problems by presenting client objects with a unified interface to the underlying EJBs. Client objects interact only with the façade, which resides on the server and invokes the appropriate EJB methods. As a result, dependencies and communication between clients and EJBs is reduced. A session façade can also simplify transaction management: for example, when a database transaction involves multiple method calls, all could be wrapped in one method of the façade and the transaction could be monitored at that level.

In the VSM, the `CartManagerBean` implements a session façade that provides an interface to the EJBs that manage the items in a customer's shopping cart. The `CartManagerBean` exposes the `checkOutCart` method to clients, encapsulating inventory management and order creation tasks performed by underlying objects `InventoryManager` and `OrdersBean` (accessed via `OrdersHome`).

```
public StringBuffer checkOutCart( ShoppingCart cart )
       throws CartException {
  try {
    // Get the order home
    OrdersHome home = (OrdersHome)ServiceLocator.getLocator().
                                                 getService( "Orders" );
    ...
    // For each item in the cart
    for( int i = cart.getItems().length - 1; i >= 0; i-- ) {
      currentItem = (CartItem)cart.getItems()[i];
      // check the inventory
      if( !InventoryManager.inventoryCheck( currentItem.getID(),
                                            currentItem.getQuantity() ) ) {
        response.append( "Failure," );
        response.append( currentItem.getID() );
        response.append( "," );
        response.append( InventoryManager.getInventory(currentItem.getID()));
        continue;
      }
      ...
        // Create the order and add it to the table
        order = home.create( new Integer( orderID ), details );
        shops.put( shopID, order );
```

```
        }
        ...
        // Add it to the order for the shop
        order.setOrderItemID( item );
        cart.removeItem( i );
      }
      if( response.length() < 1 ) {
        response.append( "Checked out your cart successfully" );
      }
      return response;
      ...
    }
    ...
  }
```

The following figures show this client-EJB interaction with and without the session façade, and how the session façade reduces network traffic.

Without Session Façade	**With Session Façade (CartManagerBean)**
	

Value Object

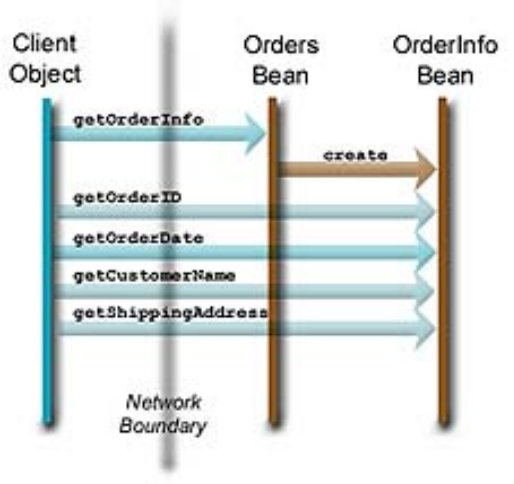
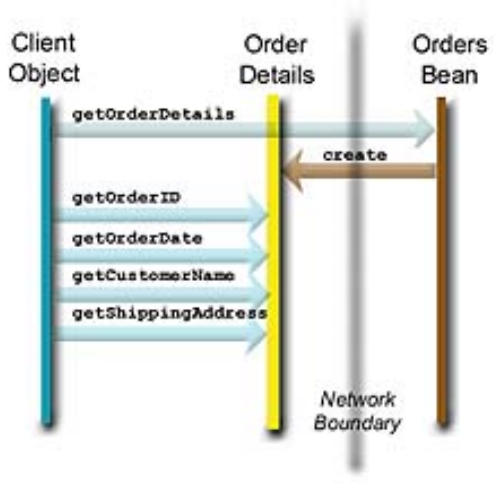
The Value Object design pattern (also known as Data Transfer Object) describes a container for a set of related data. It is often (but not necessarily) used with the Session Façade pattern to reduce network traffic and the number of method calls required to get an entity's attribute values. For example, when a customer uses the VSM to buy a product, the application generates an order comprising several attributes including order ID, order date, customer name, and shipping address. To retrieve the details of an order, an application that does not implement the Value Object pattern would have to make a remote `get` method call for each attribute (example: `Orders.getOrderID`), adding to network traffic and increasing EJB container resource usage.

In contrast, the VSM implements the Value Object design pattern in several places, creating a container object and sending it across the network to the client, which can then access the data via local method calls. For example, the following code from

`CartManagerBean.checkOutCart` uses an `OrderDetails` object to store data for a given order.

```
public StringBuffer checkOutCart( ShoppingCart cart )
       throws CartException {
  ...
        // Create order details
        details = new OrderDetails( orderID,
                                    new Date(),
                                    cart.getUserName(),
                                    shopID.intValue(),
                                    shippingAddress.getAddress(),
                                    shippingAddress.getCity(),
                                    shippingAddress.getState(),
                                    shippingAddress.getCountry(),
                                    shippingAddress.getZip(),
                                    shippingAddress.getPhone() );
        // Create the order and add it to the table
        order = home.create( new Integer( orderID ), details );
        shops.put( shopID, order );
  ...
}
```

The figures below show interactions with and without the Value Object design pattern, and how the pattern reduces network traffic.

Without Value Object	**With Value Object (OrderDetails)**
	

The VSM application implements the Value Object design pattern in these files:

- `UserDetails.java`
- `ItemDetails.java`

- `ItemAttributes.java`
- `OrderDetails.java`
- `ShopDetails.java`

Message Façade

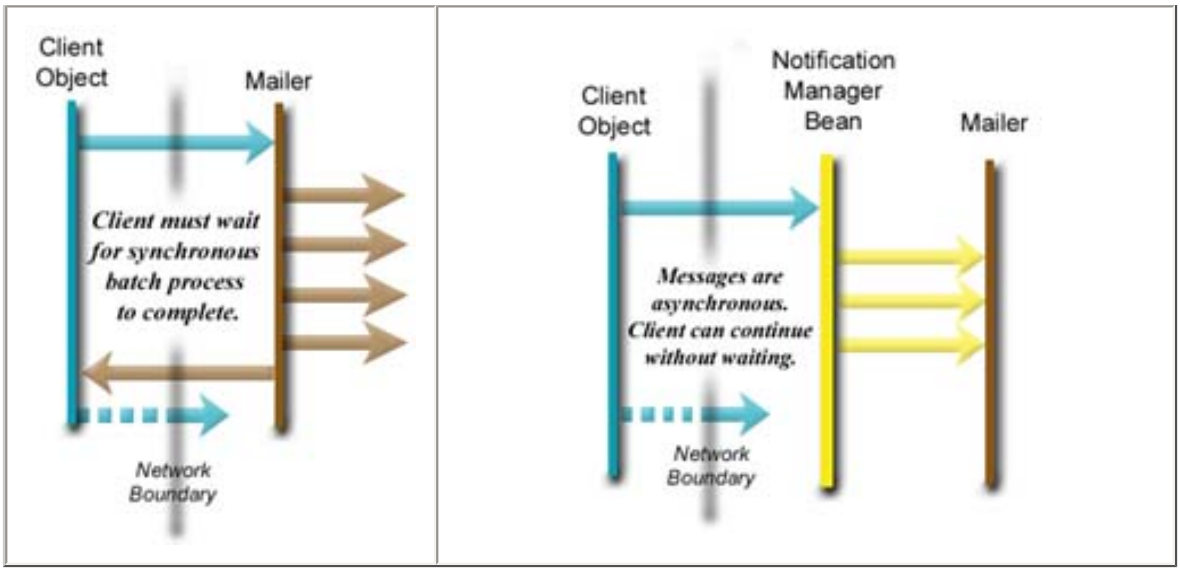
The Message Façade design pattern is similar to the Session Façade. A server-side object provides client objects with an interface to the methods of one or more EJBs. However, unlike the Session Facade, where messages are synchronous, the Message Facade uses asynchronous messaging. This design pattern is useful in situations where a client message triggers a process such as a batch update or a mass emailing, and the client doesn't want or need to wait for a return value before continuing its flow of execution.

The VSM uses this design pattern where the application has to send reports about shops requested or discontinued, password requests, etc. The VSM uses a Message Driven Bean (MDB) as a message façade, as shown in the following code from `NotificationManagerBean.java`. This bean implements the asynchronous messaging functionality defined in the interfaces `javax.jms.MessageListener` and `javax.ejb.MessageDrivenBean`, creating a system of message queues, senders, and receivers.

```
public class NotificationManagerBean implements MessageDrivenBean,
                                                 MessageListener {
...
  /**
   * Consumes messages and sends mails
   * @param <b>msg</b> the message from the client
   */
  public void onMessage( Message msg ) {
    // Get the object message
    ObjectMessage message = (ObjectMessage)msg;
    try {
      Mailer.sendMail( (MailContent)message.getObject() );
    } catch( JMSException ex ) {
      ...
    }
  }
  ...
}
```

The figures below show interactions with and without the Message Façade design pattern, and how the Message Façade enables the client object to send a message and continue its flow of execution without waiting for the Mailer bean to finish processing and return a value.

Without Message Façade	**With Message Façade (NotificationManagerBean)**



Service Locator

The Service Locator design pattern gives clients a simple interface to the potentially complex details of finding and creating application objects and services. Without a Service Locator, clients must use the JNDI (Java Naming and Directory Interface™) API to perform resource-intensive lookup and creation tasks. A Service Locator simplifies client-side code: it abstracts and encapsulates such dependencies and network details, and moves logic to the server tier. In addition, the Service Locator caches the initial context objects and references to the factory objects (e.g., EJBHome interfaces, JMS connection factories) shares them with other clients to improve overall application performance.

In the VSM, `ServiceLocator.java` implements this design pattern. This singleton class is the central place for looking up objects in the JNDI tree, as shown in the following code from `ServiceLocator.getService`, which finds an object matching a supplied JNDI name.

```
public class ServiceLocator {
...
  /**
   * Method to return an object in the default JNDI context, with
   * the supplied JNDI name.
   * @param <b>jndiName</b> The JNDI name
   * @returns <b>Object</b> The object in the JNDI tree for this name.
   * @throws <b>UtilityException</b> Exception this method can throw
   */
  public Object getService( String jndiName )
                throws UtilityException {
    try {
      // If the service is not in the cache,
      if( !homeCache.containsKey( jndiName ) ) {
        // Get the object for the supplied jndi name and put it in the cache
        homeCache.put( jndiName, defaultContext.lookup( jndiName ) );
      }
    } catch( NamingException ex ) {
      throw new UtilityException( "Exception thrown from getService " +
                                  "method of ServiceLocator class of given "+
```

```
                                        "status : " + ex.getMessage() );
    } catch( SecurityException ex ) {
      throw new UtilityException( "Exception thrown from from getService " +
                                  "method of ServiceLocator class of given "+
                                        "status : " + ex.getMessage() );
    }
    // Return object from cache
    return homeCache.get( jndiName );
  }
...
}
```

Conclusion

The Virtual Shopping Mall (VSM) sample application shows how design patterns can simplify entrerprise development tasks, and demonstrates approaches you can use to implement common J2EE design patterns. If you're interested in other approaches, you can visit OTN to learn how Oracle's J2EE-compliant Business Components for Java (BC4J) framework provides off-the-shelf implementations of numerous design patterns you would otherwise have to code by hand. The article Simplifying J2EE and EJB Development with BC4J provides more information.

< Back to Introduction	About the VSM Sample Application >

Questions or comments? Post a message in the OTN Sample Code discussion forum or send email to the author.

Using J2EE Design Patterns: Design Patterns in the VSM
Author: Robert Hall, Oracle Corporation
Date: May 2002

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7200
