Coordinated Web Controllers: A Reusable Asset
Agenda

- The Problem
  - Domain – what problem does the Coordinated Web Controller asset solve?
  - Pragmatics – how do you package and deliver an asset so that it is cost effective?
- The Solution: Coordinated Web Controller Asset
  - Making sure this is the right solution (fit analysis)
  - Applying the asset in your environment
- Coordinated Web Controllers in Action
  - A demonstration using the asset’s unit tests
Audience

- This presentation assumes that you are familiar with the following technologies
  - Web applications
  - Java, J2EE 1.2, JSP 1.1
  - Unified Modeling Language (UML)
The Problem: Domain

- Many web applications are driven by controllers (MVC, Front Controller J2EE Design Pattern).
  - Controllers manage the dialog between the user and the system, overriding page requests if appropriate.
- Many web applications are a composition of discrete sub systems or separate web applications.
- Controllers are often developed independently from each other.
- Clashes and confusion can happen when controllers are integrated in the same
  - Which controller has precedence?
  - Can multiple controllers be active at the same time?
  - What are the rules for managing individual controllers?
The Problem: Pragmatics

- How do I find solutions to my problems in general?
  - Books (GoF Design Patterns, Java Cookbook, Core J2EE Patterns, etc.)
  - Web Sites (jGuru, TheServerSide.com, etc.)
  - Components

- How do I determine if this is the right solution for my problem?
  - Try/Evaluate it
  - Ask a friend.

- What do I have to do to get this solution working in my development environment?
The Reusable Asset Specification is a standard for packaging assets. It provides mechanisms and guidance for:

- How to manage discrete artifacts
- What documentation is required
- Custom automation via scripting
The Coordinated Web Controller (CWC) asset is a MVC based design pattern/mechanism that allows multiple controllers to be integrated in the same web application. It extends the Front Controller design pattern by providing a mechanism to coordinate the activities of multiple active Front Controllers.

This asset is a combination of:

- Documentation
- Java source and configuration files
- UML models
- Executable patterns (XDE)
- Process and usage guidance
- Unit test infrastructure
The Coordinated Web Controller asset is packaged as a RAS file.

The asset has a single descriptor file, formatted as an XML document.

- **Classification** – specifies elements that help classify the asset’s application and development environments.
- **Solution** – specifies the artifacts and types that make up the asset.
- **Usage** – lists activities for the use of this asset. Can be automated scripts or simply pointers to included documentation.
- **Related Assets** – references to other dependent assets.
<?xml version="1.0" encoding="UTF-8"?>
<asset asset-guid="" date="" name="RAS File" ... >
<classification>
    <descriptor-group name="Reference Application Classification">
        <context id="Application_Environment" name="Application Environment" value="Websphere 4.2"/>
        <context id="Development_Environment" name="Development Environment" value="WSSAD 4.0.3, Rational XDE"/>
        <descriptor name="Artifact Author" value="jim conallen"/>
        <descriptor name="Packager" value="jim conallen"/>
        <descriptor name="Keyword" value="web application"/>
        <descriptor name="Keyword" value="web development"/>
        <descriptor name="Keyword" value="J2EE"/>
        <descriptor name="Keyword" value="JSP"/>
        <descriptor name="Keyword" value="controller"/>
        <descriptor name="Keyword" value="coordinator"/>
        <descriptor name="Keyword" value="MVC"/>
        <descriptor name="Keyword" value="request processing"/>
        <descriptor name="Keyword" value="XDE"/>
        <descriptor name="Keyword" value="Websphere"/>
        <descriptor name="Keyword" value="Java"/>
        <descriptor name="Problem Solved" value="Coordination of discrete web controllers."/>
    </descriptor-group>
</classification>
</solution>
<descriptor name="Keyword" value="coordinator"/>
<descriptor name="Keyword" value="MVC"/>
<descriptor name="Keyword" value="request processing"/>
<descriptor name="Keyword" value="XDE"/>
<descriptor name="Keyword" value="Websphere"/>
<descriptor name="Keyword" value="Java"/>
<descriptor name="Problem Solved" value="Coordination of discrete web controllers."/>
</descriptor-group>
</classification>
<solution>
<artifact reference="CWC Code.mdx" type="XDE Model"/>
<artifact reference="Unit Test Design.mdx" type="artifact"/>
<artifact reference="Unit Test User-Experience.mdx" type="artifact"/>
<artifact reference="documentation/Web Controller/cwc.html" type="artifact"/>
<artifact reference="documentation/Web Controller/install.html" type="artifact"/>
<artifact reference="source/com/rational/cwc/Coordinator.java" type="artifact"/>
<artifact reference="source/com/rational/cwc/IPageDispatcher.java" type="artifact"/>
<artifact reference="source/com/rational/cwc/IWebController.java" type="artifact"/>
<artifact reference="source/com/rational/cwc/PageDispatcher.java" type="artifact"/>
<artifact reference="source/com/rational/cwc/RequestProcessor.java" type="artifact"/>
<artifact reference="webApplication/WEB-INF/web.xml" type="artifact"/>
<artifact reference="webApplication/WEB-INF/WebControllerConfig.xml" type="artifact"/>
<artifact reference="webApplication/WEB-INF/WebControllerConfig.xsd" type="artifact"/>
<artifact reference="webApplication/WEB-INF/lib/xercesImpl.jar" type="artifact"/>
<artifact reference="webApplication/WEB-INF/lib/xmlParserAPIs.jar" type="artifact"/>
...
</solution>
<usage>
<asset-activity>
<activity id="Install" task="Install this asset" reference="documentation/Web Controller/cwc.html"/>
<activity id="Factory config file default" reference="source/com/rational/cwc/WebControllerFactory.java"/>
<activity id="Update web.xml configuration file" reference="webApplication/WEB-INF/web.xml"/>
</asset-activity>
</usage>
Fit Analysis

- What environments has this asset been designed for? Tested against?
- Who created this asset? Is this a reliable source?
- How much of the asset is open? How much is binary and how much is source?
- How much will I have to change my current design to accommodate this pattern?
- What other assets/patterns am I using? Are there any conflicts?
  - Is this asset compatible with the “Pattern Template” J2EE pattern?
  - What impact will this have on my HTML development (UX) team?
Assets provide their own documentation which includes:

- Author/packager
- Classification keywords
- Problem summary
- Benefits
- Liabilities
- Known uses
- Usage instructions
- Related Assets
- Resource Summary
Asset Usage

The usage section guides the developer on how to “use” the asset including:

- Installation
- Configuration
- Testing
- Modifying

The decision to use this asset is expected to be made early on in the evolution of the system’s architecture. This is because the use of this asset effects how all the URL’s of the application are constructed.

The first step in using this pattern is to ensure it will work in your environment (web container, network, browser, etc.). The supplied unit test code should be sufficient to validate most of the working assumptions made in the development of this asset. This asset was developed with IBM WebSphere Studio Application Developer 4.0.3 and used Xerces 2.0.2 for XML parsing. No propriety APIs were used in the creation of this asset so it is expected that it can run on other J2EE compliant platforms. Specific directions are supplied for IBM WebSphere Studio Application Developer.

The Controlled Web Controllers Asset Installation document details how this asset can be installed into WebSphere Studio Application Developer 4.0.3 with Rational XDE Release 2. If you are not using WebSphere Studio Application Developer or Rational XDE Release 2, you will have to unpack the asset manually and import the required files manually. Refer to your IDE and web container documentation for more information on how to create and configure an appropriate web application.

After installing the asset you may need to edit a default value in one of the classes of the asset. The WebControllerFactory class has a method called ReloadControllers. This method has hard-coded a default value for the position of the controller configuration file. Edit this default value if necessary and rebuild the application.

```java
public static void ReloadControllers()
{
  TreeMap controllers = new TreeMap();
  String configfile = System.getProperty("WebControllerConfig", "C:\\Program Files\\IBM\\Application Developer\\workspace\\Web-INF\\WebControllerConfig.xml");
  ...
}
```

After installing and configuring the asset you should run through some simple unit test scenarios to ensure that all the
Participants - 1

- **RequestProcessor**
  - The request processor is a servlet that intercepts all incoming HTTP requests and delegates its processing to the coordinator. A separate coordinator instance is cached in each user's session (i.e. use bean tag with session scope). After processing an IPageDispatcher object is returned that is used to initiate the dispatching of the response page.

- **Coordinator**
  - The Coordinator is responsible for managing a queue of Web controllers. The coordinator is called upon by the RequestProcessor to process an incoming HttpRequest. The HttpRequest is processed by the active controller and then the coordinator queries the controller for the next screen. The controller returns an instance of an object that implements the IPageDispatcher interface, which is passed back to the RequestProcessor.
  - The coordinator examines the URL to see if a different controller is specified. Only if the active controller (top of stack) doesn't require exclusive access will the controller try to find the specified controller in active queue. If it isn't found then the coordinator will create a new one from the WebControllerFactory.
Servlet Configuration

- The most important web.xml configuration option directs all incoming requests to the RequestProcessor servlet

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd" >
<web-app id="WebApp" >
  <display-name> Cwc </display-name>
  <servlet>
    <servlet-name>RequestProcessor</servlet-name>
    <display-name>RequestProcessor</display-name>
    <servlet-class>com.rational.cwc.RequestProcessor</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>RequestProcessor</servlet-name>
    <url-pattern>*.scr</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <error-page>
    <exception-type>com.rational.cwc.UnknownControllerException</exception-type>
    <location>/index.jsp</location>
  </error-page>
</web-app>
```
Participants - 2

- **IWebController**
  - An object that implements the IWebController interface is capable of processing an HttpRequest. The controller interacts with middle tier components to update the state of the business. A controller must also be able to return an IPageDispatcher object that can be used to forward the response page processing to. A web controller must also be able to return key meta information (name and description) and indicate whether it is in an exclusive state or not (i.e. should not be bumped from the active controller role). The last responsibility of a controller is to indicate when it has completed. When a controller indicates that it has completed the coordinator is free to remove it from the active controller stack.
Participants - 3

- **WebControllerFactory**
  - The WebControllerFactory is a singleton factory object that is responsible for the creation of WebControllerMeta instances. Its main responsibility is to read the XML configuration file that lists the names and Java classes of all the known controller classes. The file also specifies some meta information about the controller. The configuration file can be reloaded at runtime.

- **WebControllerMeta**
  - A WebControllerMeta instance represents meta information about an actual Web controller class. The meta class is used to create actual instances of a controller via the Class.forName(...) and createInstance() methods.
Sample Controller Configuration File

- A custom XML schema for this asset.
- Represents controller meta information.
- Controllers are defined with names and implementing classes. Exclusivity is set in this file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<WebControllerConfig:web-controller-config
 xmlns:WebControllerConfig="http://www.rational.com/cwc"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.rational.com/cwc/WebControllerConfig.xsd">
 <web-controllers>
   <web-controller name="UTController1A" class="com.rational.cwc.unittest.Controller1"
     description="Sample controller 1A"/>
   <web-controller name="UTController2A" class="com.rational.cwc.unittest.Controller2"
     description="Sample controller 2A"/>
   <web-controller name="UTController1Cx" class="com.rational.cwc.unittest.Controller1"
     exclusive="true" description="Sample controller 1C Exclusive control"/>
   <web-controller name="UTController2Cx" class="com.rational.cwc.unittest.Controller2"
     exclusive="true" description="Sample controller 2C Exclusive control"/>
   <web-controller name="Login" class="com.rational.cwc.unittest.LoginWebController"
     description="Simple login web-controller that includes a balance check"/>
 </web-controllers>
 <default-web-controller name="UTController1A"/>
</WebControllerConfig:web-controller-config>
An object that implements the IPageDispatcher interface is capable of issuing a forward request to a RequestDispatcher object. This interface provides an abstraction layer in the HTTP request processing cycle. Implementations may interpret the forward request very simply (i.e. forward directly to the supplied URL), or they may provide sophisticated implementations that involve page composition (i.e. Sun J2EE Templated Page pattern).

A simple implementation of the IPageDispatcher interface. When created the PageDispatcher object accepts a URL. This URL is used directly (without modification) with the RequestDispatcher when forwarding.
Collaborations: New Controller

: RequestProcessor : Coordinator : WebControllerFactory

/ controller : IWebController / request / new controller : IWebController

1 : processRequest ( request )
2 : getActiveController ( )
3 : GetDefaultWebControllerName ( )
4 : addController ( controllerName )
5 : completed ( )
6 : removeController ( controller )
7 : \
8 : getName ( )
9 : getExclusive ( )
10 : addController ( controllerName )
11 : getActiveController ( )
12 : \
13 : process ( req )
14 : nextScreen ( )
Collaborations: Add Controller

If not currently an active controller get a new one from the factory and make it the currently active controller.

If the controller was found in the vector of active controllers, then remove it and then add it to the end of the vector (and hence making it top of the virtual stack).

1: addController (controllerName)
2: iterate activeControllers
3: getName()
4: remove(controller)
5: add(controller)
6: GetWebController(name)
7: add(controller)
Collaborations: Controller Complete

1: processRequest(request)

2: setActiveController(controller)
3: setActiveController(completed)
4: removeActiveController(controller)
5: \remove(controller)
6: \lastElement()
7: getLastActiveController()
**RequestProcessor Servlet**

- All requests are funneled through this servlet therefore it must be efficient.

```java
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    HttpSession session = request.getSession();
    Coordinator coordinator = (Coordinator) session.getAttribute("coordinator");
    if( coordinator == null ) {
        //get fresh list of current controllers from the servlet context (application)
        coordinator = new Coordinator();
        session.setAttribute("coordinator", coordinator);
    }

    IPageDispatcher nextScreen = coordinator.processRequest(request);
    // this is our last chance to change the returned page
    nextScreen.forward( request, response );
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    doGet(request, response);
}
```
public IPageDispatcher processRequest(HttpServletRequest request) {
    IWebController controller = getActiveController();
    if( controller.completed() ) {
        removeController( controller );
        controller = getActiveController();
    }

    // check to see if another controller has been requested
    String controllerName = request.getParameter("controller");
    if( controllerName != null && !controllerName.equals(controller.getName())
        && !controller.getExclusive() ) {
        addController( controllerName );
        controller = getActiveController();
    }

    // check to see if we've been asked to reset the controller first
    String reset = request.getParameter("reset");
    if( reset != null ) controller.reset();

    controller.process( request );
    IPageDispatcher next = controller.nextScreen();

    return next;
}
Lazy instantiation is used when getting the active controller.

The addController method is responsible to maintaining the integrity of the controller queue.

```java
public IWebController getActiveController() {
    // there is also where any custom logic in prioritizing
    // which active controller should go. In this example
    // the priority is what ever is on top of the stack, or
    // rather the last controller to be added.

    if (activeController == null) {
        // default to a known one
        String controllerName = WebControllerFactory.GetDefaultWebControllerName();
        addController(controllerName);
    }

    return activeController;
}
```
private void addController( String controllerName ) {
    IWebController controller = null;
    int index = -1;
    boolean found = false;
    int count = activeControllers.size();
    for (Enumeration e = activeControllers.elements(); e.hasMoreElements();) {
        IWebController ctrl = (IWebController) e.nextElement();
        index++;
        if( ctrl.getName().equals( controllerName ) ) {
            controller = ctrl;
            found = true;
            break;
        }
    }

    if( found ) {
        if( index < count-1 ) { // put it at the top of the list.
            activeControllers.remove(controller);
            activeControllers.add(controller);
            activeController = controller;
        }
    } else {
        controller = (IWebController) WebControllerFactory.GetWebController( controllerName );
        if( controller != null ) {
            activeControllers.add( controller );
            activeController = controller;
        }
    }
}
**Coordinator::removeController(...)**

- When a controller is remove the next one in the queue moves to the front.
- If the active controller is set to null the getter will instantiate a new default controller.

```java
private void removeController(IWebController controller) {
    if (controller != null) {
        activeControllers.remove(controller);
        if (activeController.equals(controller)) {
            if (activeControllers.isEmpty()) {
                activeController = null;
            } else {
                activeController = (IWebController) activeControllers.lastElement();
            }
        } else {
            activeController = (IWebController) activeControllers.lastElement();
        }
    }
}
```
Working with JSPs

- JSPs should focus on building the UI.
- JSPs are **not** responsible for processing user input. This is the responsibility of the controller.
- JSPs can access the controller and coordinator through the HttpSession object.
- All hyperlinks in web pages should be formatted to point to the RequestProcessor servlet (i.e. have a *.scr extension).
- Links can request a new controller with the “controller” parameter
  - `<a href="profile.scr?controller=account">Update Profile</a>`
<%@ page import="com.rational.cwc.*" %>
<%@ page import="com.rational.cwc.unittest.*" %>
<jsp:useBean class="com.rational.cwc.Coordinator" id="coordinator" scope="session" />
<%IWebController currentController = coordinator.getActiveController();
String ctrlName = currentController.getName();
if (!ctrlName.equals("Login")) {%
    throw new UnknownControllerException("Unknown controller...", ctrlName);
}%
LoginWebController controller = (LoginWebController) currentController;

String message = controller.getMessage();
%
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head><title>Login Page</title></head>
<body>
<h2>Login Screen</h2>

<p>Message: <%= message %></p>
<form name="login" action="login.scr" method="post">
    <p>Username: <input name="username" type="text" size=20></p>
    <p>Password: <input name="password" type="password" size=20></p>
    <p><input name="submitLogin" type="submit" value="Login"></p>
</form>
...
</body>
</html>
The Controller

Controllers in this mechanism are all state based. Each controller is expected to maintain its own state machine.

A controller must:

- Accept an HttpServletRequest object to process.
- Return an IPageDispatcher object that is used to trigger the building of the output page (usually forwards to a JSP or servlet).

Controllers must gracefully handle the following user events:

- Refresh button
- Back/Forward button
- Bookmarks
- Concurrent browser windows.
Example: Login Controller

- A controller to manage the login of a web user.
- If the user has an outstanding balance request payment before allowing them to get to user status screen.

```java
public class LoginWebController {
    # currentState : int = ST_INIT
    # requestedURL : String = null
    # message : String = null
    # balance : double = 0.0
    # cardHolder : String = null
    # cardType : String = null
    # cardNumber : String = null
    # cardExpiration : String = null
    # username : String = null
    # description : String = null
    # ST_INIT : int = 0
    # ST_LOGIN : int = 1
    # ST_PAYMENT_REQUIRED : int = 2
    # ST_ACCEPT_PAYMENT : int = 3
    # ST_USER_STATUS : int = 4
    # ST_LOGOUT : int = 5
    # password : String = null
    # name : String = null

    + completed ( )
    + getExclusive ( )
    + nextScreen ( )
    + reset ( )
    + process ( )
    + getName ( )
    + getDescription ( )
    + getMessage ( )
    + getBalance ( )
    + getCardHolder ( )
    + getCardType ( )
    + getCardNumber ( )
    + getCardExpiration ( )
    - eventLogin ( )
    - eventPay ( )
    - eventPayNow ( )
    - eventLogout ( )
    + setName ( )
    + setDescription ( )
    + setExclusive ( )
    + getUsername ( )
    + getPassword ( )
}
```

```java
public interface IWebController {
    + nextScreen ( )
    + completed ( )
    + process ( )
    + reset ( )
    + getExclusive ( )
    + setExclusive ( )
    + getName ( )
    + getDescription ( )
    + setName ( )
    + setDescription ( )
}
```
The user navigates to the login screen.

The user supplies a username and password. Then logs in.

If the username/password is invalid the login screen is returned with a notification message. The user can re-enter a valid combination.

With a valid username/password combination (and no outstanding balance for the user's account, the system navigates to the user status page.

The user logs out of the system. The system returns a final logout screen, confirming a successful logout.
The user navigates to the login screen.

The user supplies a username and password. Then logs in.

The user has an outstanding balance. The system notifies the user and offers an opportunity to make the payment now, or to logout.

The user opts to pay now.

The user enters new payment information.

If the information is invalid, the system returns the payment screen and displays a message indicating the information was not accepted. The user enters new information.

The user enters valid payment information. The system returns with the user status screen.
Login Controller: JSPs

Coordinator

+ getActiveControllers()
+ getActiveController()
+ addController()
+ removeController()
+ numberActiveControllers()
+ processRequest()

- coordinator

- coordinator

«ServerPage»

login.jsp

«JSPUseBean»

«Build»

«ClientPage»

login.jsp_Client1

«ServerPage»

balance.jsp

«JSPUseBean»

«ClientPage»

balance.jsp_Client1

«Build»

«ServerPage»

payment.jsp

«JSPUseBean»

«ClientPage»

payment.jsp_Client1

«Build»

«ServerPage»

status.jsp

«JSPUseBean»

«ClientPage»

status.jsp_Client1

«Build»

«ClientPage»

logout.html

«Build»
Login Controller: Statemachine / UX mapping

- «ServerPage» login.jsp
- «ServerPage» payment.jsp
- «ServerPage» balance.jsp
- «ServerPage» status.jsp
- «ClientPage» logout.html

States and transitions:
- Login
  - [balance >= 0]
  - [balance < 0]
  - [invalid]
- Payment required
  - [valid]
  - [invalid]
- Accept payment
  - User status
  - Logout
- Logout
  - Logout
Login Controller: Implementation

- Each state in the statemachine defines an internal enumeration.

```java
protected static final int ST_INIT = 0;
protected static final int ST_LOGIN = 1;
protected static final int ST_PAYMENT_REQUIRED = 2;
protected static final int ST_ACCEPT_PAYMENT = 3;
protected static final int ST_USER_STATUS = 4;
protected static final int ST_LOGOUT = 5;
```
Login Controller: Implementation

- Of course each controller will define member attributes appropriate for its unique role.

```java
protected int currentState = ST_INIT;
protected String requestedURL = null;
protected String message = null;
protected double balance = 0.0;
protected String cardHolder = null;
protected String cardType = null;
protected String cardNumber = null;
protected String cardExpiration = null;
protected String username = null;
protected String password = null;
protected String name = "";
protected String description = "";
```
The controller provides getter, setters and simple implementations for important operations.

```java
public boolean completed(){
    return (currentState == ST_LOGOUT);
}

public boolean getExclusive(){
    return false;
}

public void setExclusive(boolean exclusive){
    // ignore this since this will never be exclusive
}

public String getBalance(){
    // return the balance as a locally formatted currency string
    return NumberFormat.getCurrencyInstance().format(balance);
}
```
public void process(HttpServletRequest req){
  message = "";
  String event = req.getServletPath();
  switch( currentState ){
    case ST_INIT: // no processing required, only possible next state is login
      currentState = ST_LOGIN;
      break;
    case ST_LOGIN: // only accepted event is login
      eventLogin(req);
      break;
    case ST_PAYMENT_REQUIRED: // user can either pay now, or logout
      if( event.equals("/paynow.scr") ) {
        eventPayNow(req);
      } else if( event.equals("/logout.scr") ) {
        eventLogout(req);
      } break;
    case ST_ACCEPT_PAYMENT:
      if ( event.equals( "/logout.scr" ) ) { // user is aborting
        currentState = ST_LOGOUT;
      } else if( event.equals("/enterpayment.scr") ) {
        eventPay(req);
      } break;
    case ST_USER_STATUS:
      if( event.equals("/logout.scr") ) {
        eventLogout(req);
      } break;
    case ST_LOGOUT: // this is a final state - nothing to do
      break;
  }
}
private void eventLogin(HttpServletRequest req) {
    // For authorization this implementation just checks for non-empty fields
    // The user balance is based entered username - just to keep this unit test
    // simple. Normally there would be calls to session beans, etc.
    username = req.getParameter("username"); // set controller state
    password = req.getParameter("password");
    if (username == null || password == null ||
        username.equals("") || password.equals("")) {
        message = "Please enter a value for both fields.";
        currentState = ST_LOGIN;
    } else {
        //if username begins with a capital letter then has balance
        char ch = username.charAt(0);
        if (ch >= 'a') {
            balance = 0;
            currentState = ST_USER_STATUS;
        } else {
            balance = 120.00;
            currentState = ST_PAYMENT_REQUIRED;
        }
    }
}

public IPageDispatcher nextScreen(){
    IPageDispatcher screen;
    // this is where the mapping between internal states of the controller
    // are mapped to screens. In this example the mapping is 1:1 - very
    // simple. So simple in fact that I could have implemented this with
    // a HashMap, however using a switch statement helps to emphasize that 1:1
    // mappings are not necessarily the only ones possible.
    switch( currentState ){
        case ST_LOGIN:
            screen = new PageDispatcher("unit-test/login.jsp");
            break;
        case ST_PAYMENT_REQUIRED:
            screen = new PageDispatcher("unit-test/balance.jsp");
            break;
        case ST_ACCEPT_PAYMENT:
            screen = new PageDispatcher("unit-test/payment.jsp");
            break;
        case ST_USER_STATUS:
            screen = new PageDispatcher("unit-test/status.jsp");
            break;
        case ST_LOGOUT:
            // not all pages need be jsp
            screen = new PageDispatcher("unit-test/logout.html");
            break;
    }
    return screen;
}
We’ve reviewed this asset’s:
- Documentation
- Models (Design, UX)
- Code
- Examples/Unit Test

The decision can be made whether or not to use this asset. If it can we need to understand how it will be incorporated into our particular project.
Using the Asset

- Incorporate the asset’s core classes
  - All java code sans unit test specific classes.
- Update our code or the asset code to use the same support components for XML parsing (Xerces 2.0.2).
- Document how to use pattern automation in the development process.
  - The CWC asset comes with an XDE pattern asset that provides automation for the creation of IWebController classes. This automation initializes the controller’s interface and creates case statements in key operations based on its modeled statemachine.
- Document how to specify and code JSPs to use controllers for all access to middle tier components.
- Ensure all HTML hyperlinks and form submissions are encoded to go through the RequestProcessor servlet.
Demonstration

- Running unit tests for
  - Login Controller
  - Simple Controller1 and Controller2
    - Examine controller stack behavior
    - Examine controllers with exclusive control

- Using Web Controller pattern asset to automate the creation of an IWebController class.
Summary

- Assets can provide sophisticated solutions to common problems.
- Standardization of asset packaging will enable greater use of assets.
- The CWC asset expands on the J2EE Front Controller Pattern and provides a mechanism to cleanly manage simultaneous web controllers.