A Pattern Language for J2EE Web Component Development[1]

Chris Richardson[2]

Servlets and JSP pages are very effective technologies for building the presentation layer of an application that support both traditional web browsers and newer wireless handheld devices. They are a platform independent standard supported by multiple vendors and are considerably easier to use and more efficient than older technologies such as CGI scripts. In the Java 2 Platform Enterprise Edition (J2EE) architecture [J2EE], servlets and JSP pages are referred to as web components and handle HTTP requests from clients, invoke components in the business logic layer (typically enterprise java beans), and generate responses that are usually HTML or XML documents.

Unfortunately, the speed and ease at which developers can use servlets and JSP pages to create a web application can cause them to ignore important issues such as maintainability and testability. Undisciplined development techniques that work for small-scale applications fail badly when applied to larger applications. For instance, one commonly used approach is to construct the presentation layer (and in some cases the complete application) entirely out of JSP pages containing large amounts of embedded Java code: presentation logic and sometimes business logic. Applications written this way typically have several serious problems:

- **Poor maintainability** - the presentation layer consists of a large number of JSP pages, each of which consists of as much as hundreds of lines of Java code and hundreds of lines of content (XML/HTML) template. Even though it is written in Java it fails to leverage the benefits of object-oriented design. The result is a large, hard to understand and maintain mass of unstructured code.
- **Poor separation of artifacts according to role** - the Java code is mixed in with the content and so Java developers and web page designers can end up getting in each other's way. They need to be able to work on separate artifacts using the appropriate tools.
- **Poor testability due to dependence on the servlet API** - a JSP page is hard to test since the only way the test driver could verify that it is working correctly is to examine the content that it generates.
- **Poor testability due to dependence on the back-end** – since the JSP pages call the business logic, they cannot be tested independently of it. Web page designers end up having to navigate through a sequence of screens in order to get to the screen built by the JSP page that they need to test. They might also have to carefully ensure that the back-end system is put into the correct state in order to test a particular JSP page, which makes manual testing tedious and automated testing more complicated. Some systems, such as those that have nightly batch processes, might only allow some tests to be performed at certain times.

[1] Copyright © 2001, Chris Richardson. Permission is granted to copy for the PLoP 2001 conference. All other rights reserved.

[2] cer@acm.org or cer@bea.com

- **Hard to adapt to different types of client devices** –An application that supports mobile devices needs to be able generate content in variety of different markup languages (e.g. HTML, WML, and HDML). The lack of separation of presentation logic from content makes this hard.
- **Poor reusability –** the lack of modularity means that it is hard to reuse components in multiple applications.

Some of these issues can be partially addressed by moving some of the code into helper beans [JSP]. This makes the code easier to understand and separates presentation logic from the content, enabling web page designers to work separately from the developers. However, unit testing is still difficult since the helper beans typically use the servlet API. Also, the JSP pages still call the back-end system and cannot be tested without it running. Furthermore, using helper beans doesn't make it easier to support multiple device types or necessarily improve reusability.

The JSP Model 2 architecture [JSP], in which a servlet handles the request and then invokes a JSP page to generate the response, also addresses some of these issues: maintainability is improved; some presentation logic is separated from content; the JSP pages are independent of the business logic; it supports multiple versions by using multiple JSP pages. However, the JSP Model 2 architecture is insufficient for many applications: the servlet may still be unstructured, procedural code; testing of the servlet is still difficult; the JSP pages can still contain hard to test presentation logic; and the lack of clearly defined roles for classes makes reuse difficult.

This paper describes a pattern language for developing maintainable and testable web presentation layers. This pattern language, which builds on the JSP Model 2 architecture, grew out of the author's experience developing web-based applications and observing typical development techniques and the problems they caused. The intended audience for this pattern language are Java developers and architects who are familiar with the basics of J2EE web application development.

Figure 1 shows the architecture of a presentation layer built using these design patterns.

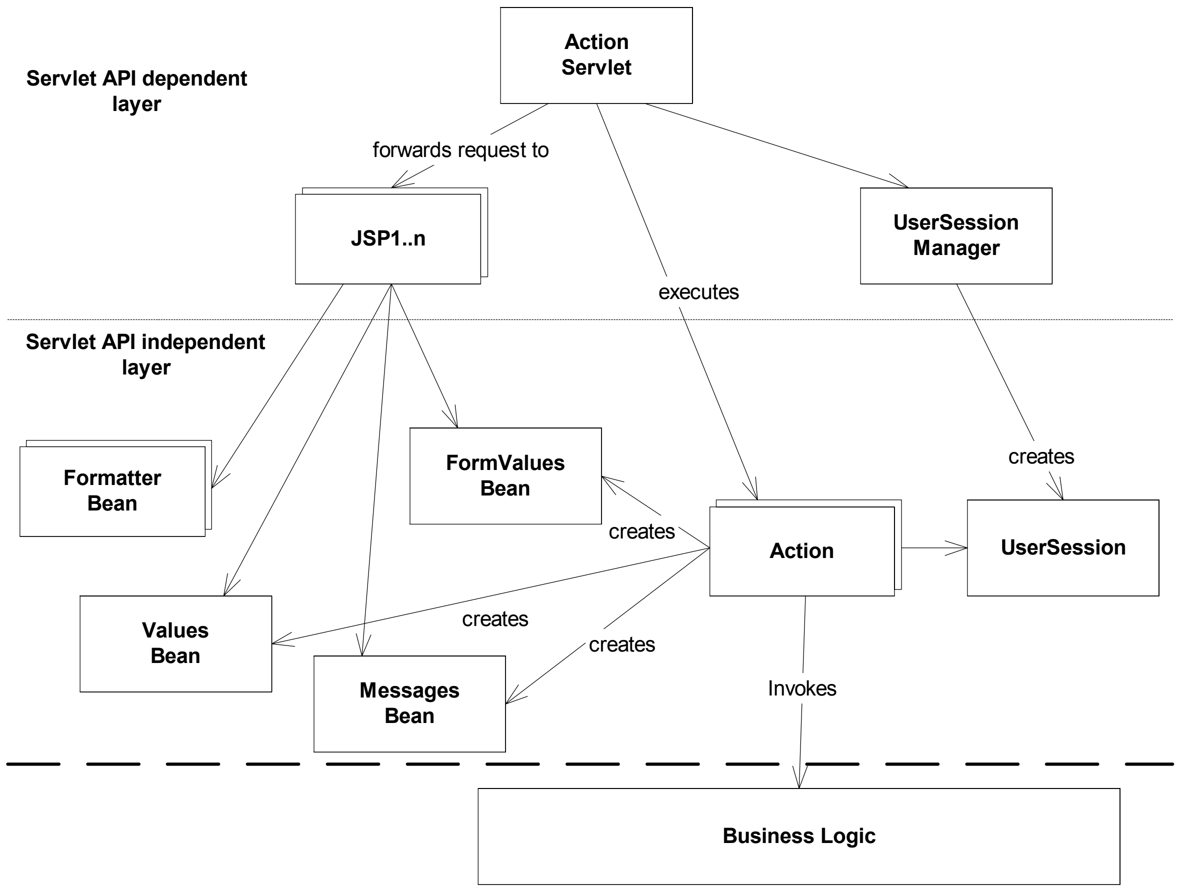


Figure 1: Presentation layer architecture

The presentation layer into two sub-layers – the top layer consists of a servlet class, the JSP pages and some other classes that use the servlet API. The bottom layer contains the majority of the code and consists of classes that don't use the servlet API. These classes can run outside of the web container and are significantly easier to develop and test.

The key aspects of this architecture are as follows:

- The presentation layer has a single ***Action servlet*** class that handles an HTTP request by first invoking an action bean, which validates the request's parameters and invokes the business logic. The servlet then forwards the request to a JSP page, which generates the response.
- A UserSessionManager class encapsulates how the user's session state is stored, i.e. within an HttpSession object and/or by using cookies.
- The ***Action servlet*** uses three beans to pass data to a JSP page: a messages bean containing the error messages to display, a values bean containing the data to display and a form values bean containing previously entered form values.
- A JSP page uses ***formatter beans*** to format values such as dates and numbers.

The following table lists the patterns and the problems they solve:

Pattern		Problem	Solution
1	Controller Servlet	How do you divide up the work of handling a request between servlets and JSP pages?	Dispatch requests to a controller servlet, which invokes the business logic and then forwards the request to a JSP page, which generates the response.
2	Three Java Beans	How do you specify the data that the controller servlet passes to the JSP page?	Define three Java beans (previous form values, error messages and display values) that the controller servlet always passes to the JSP page.
3	Encapsulate User Session Management	How do you represent user session state?	Encapsulate session state into a User Session object.
4	Action Servlet	How do you implement the controller servlet?	Implement the controller servlet as a façade that delegates to an action bean.
5	Formatter Bean	How does a JSP page format values such as numbers and dates?	Define one or more formatter beans that invoke the Java formatting classes.
6	JSP Page Test Driver	How do you enable web page designers to quickly and easily test their JSP pages?	Write a test driver (servlet) for each JSP page.

The remainder of this paper is organized as follows:

- Sections 1 through 6 describe the pattern language
- Section 7 describes how this pattern language can be used
- Section 8 is a summary

1. Controller Servlet

1.1. Aliases

- JSP page Model 2
- Front component + Presentation component

1.2. Context

You are developing the presentation layer of a J2EE application that supports web clients: desktop browsers and mobile devices. In the J2EE architecture this is accomplished by developing a presentation layer (also called a J2EE web application) that consists of static content, such as HTML pages and images, and web components (servlets and JSP pages) that generate dynamic content.

The web container uses the HTTP request's URI and a set of mappings (written by the developer) to select the web component to handle the request. The web component typically processes a request by validating the request's parameters (e.g. verify that the user filled in a form correctly), invoking the business logic and generating a response.

A web component that handles the submission of the form has to verify that the user filled in the form correctly[3] and redisplay the form with one or more error messages if they didn't. To do this, the web component verifies that each request parameter that corresponds to a required form field has a non-blank value. It will also verify that all supplied values (required and optional) are syntactically correct, e.g. number and date fields are in the correct format and that zip codes consist of five digits. Validation of a complex form can require a lot of code.

In order to invoke the business logic, the web component will convert each request parameter (which is a string) into the required data type (e.g. number, date) and create a request object containing the converted values. The web component will handle any exceptions that are thrown by the business logic.

The web component generates the response, directly, or by using other web components. It can use other web components by either forwarding the request to another web component, or including the output of one or more web components.

Servlets can be used to generate HTML/XML content. However, this is only practical for very simple web pages, since the content would be embedded within print statements inside the servlet making it hard to create and maintain. JSP pages are much better way to generate content. It is very straightforward to create and maintain JSP pages using HTML/XML editing tools.

JSP pages can use tags that make it easy to create, initialize and execute Java beans. For example, the <setProperty> tag initializes a Java bean with the parameters from the HttpServletRequest. You can define your own custom JSP tags that can encapsulate complex Java code and possibly enable web page designers to create more of the front-end.

Java developers can write JUnit-based[JUNIT] tests for testing servlets and JSP pages using HttpUnit [HTTPUNIT] and Cactus[CACTUS]. The Cactus framework enables developers to write unit tests for servlets and other server-side classes that use the servlet API. A Cactus test case usually consists of the following steps:

1. Initialize servlet API objects, such as the request and session, to the desired state

[3] The application shouldn't rely on client-side scripting for validation: the user might have disabled Javascript on their browser; many mobile devices don't have client-side scripting.

2. Instantiate the class under test and invoke the method under test
3. Verify that the return value of the method is correct
4. Verify that the servlet API objects are in the correct state

Cactus is not intended to test the servlet's service() method (HttpUnit should be used to do that) but rather helper methods that are called by the service() method.

HttpUnit is an HTTP client framework and enables developers to write test cases for JSP pages and servlets. A test case can use HttpUnit to send an HTTP request, with the desired request parameters, headers and cookies to a servlet or JSP page. It can then verify that the servlet or JSP page generates the correct response by examining the headers, cookies and the (text) body of the response. HttpUnit includes an HTML/XML parser and so the test case can also access the DOM [DOM] representation of the response. HttpUnit is extremely useful for writing simple tests that simulate clicking on links and filling in forms. It is harder to write test cases that verify that the servlet or JSP Page has generated correctly formatted content. For example, it would be hard to write a test case that verifies that the application displays an error message next to an incorrectly filled in form field. Furthermore, HttpUnit-based tests are often fragile and easily broken by minor changes to the layout of a page, such as changing the location of a text field's error message. In general it is easier to test Java classes directly using JUnit/Cactus than it is to use HttpUnit.

The application development team includes Java developers and web page designers.

1.3. Problem

How do you divide up the work of handling requests and generating responses between servlets and JSP pages?

1.4. Forces

- Web page designers who are not Java programmers need to be able to create and maintain the content.
- Java developers and web page designers need to be able to work separately on different artifacts.
- You want to be able to write maintainable JUnit-based test cases.
- You want to be able to easily verify the appearance of dynamically generated web pages without requiring a functioning back-end.
- An application needs to be able to handle requests from and generate content for multiple client types - desktop browsers, PDAs and mobile phones - without unnecessary code duplication.

1.5. Solution

Build a web application that dispatches each request to a controller servlet, which invokes the business logic and then forwards the request to a JSP page, which generates the response. Figure 2 shows the structure of this pattern.

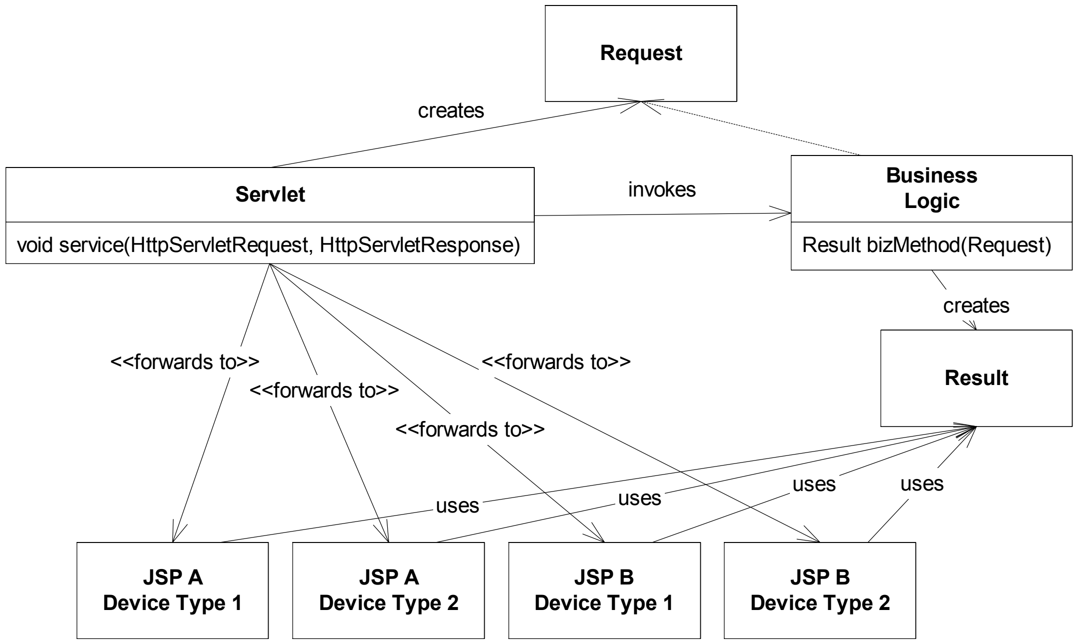


Figure 2: Controller Servlet structure

The participants in this pattern are:

- Controller Servlet
  - validates the HTTP request's parameters
  - invokes the business logic and handles exceptions and errors
  - forwards the HTTP request to the JSP page chosen to generate the response
- Business Logic
  - the interface to the application's business logic
  - defines a bizMethod() that the controller servlet calls
- Request
  - passed as an argument to the business logic method
  - contains data derived from the HttpServletRequest parameters
- Result
  - returned by the business logic method
- JSP Page
  - invoked by the controller servlet to generate the response
  - generates the content from the data in the result object

The controller servlet uses the servlet API's RequestDispatcher mechanism to forward the HTTP request to the JSP page. A servlet might forward the request to one of several different JSP pages. There might also be several different versions of each JSP page (HTML, WML etc) and the controller servlet might, for example, use the user-agent request header to select which one to use.

Figure 3 shows how the servlet handles the request in more detail.

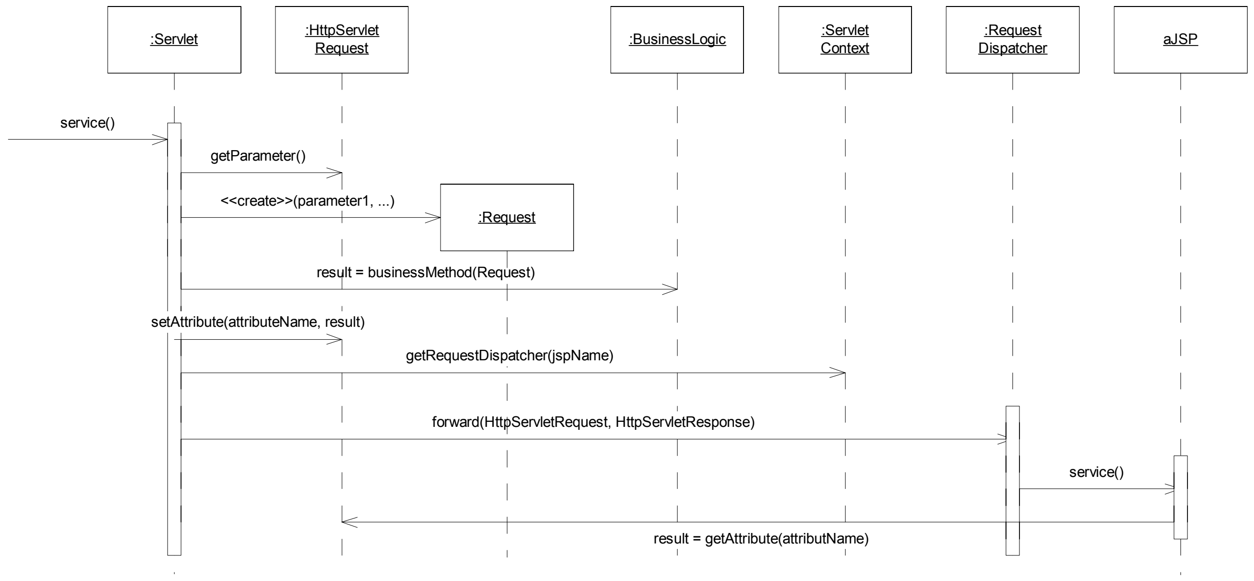

:Servlet
:HttpServlet Request
:BusinessLogic
:Servlet Context
:Request Dispatcher
aJSP
service()
getParameter()
<<create>>(parameter1, ...)
:Request
result = businessMethod(Request)
setAttribute(attributeName, result)
getRequestDispatcher(jspName)
forward(HttpServletRequest, HttpServletResponse)
service()
result = getAttribute(attributName)

Figure 3: Controller Servlet collaborations

The sequence of events is as follows:

1. The servlet gets the parameters from the HttpServletRequest, validates them and creates a request object.
2. The servlet invokes the business logic method passing the request object.
3. The servlet stores the result object returned by the business logic as an attribute of the HttpServletRequest.
4. The servlet uses the RequestDispatcher to forward the request to a JSP page.
5. The JSP page gets the result object from the HttpServletRequest and generates the response.

There are times, such as displaying a form for the first time, when a servlet will not invoke any business logic and will, instead, forward the request directly to a JSP page.

When this pattern is applied, the following decisions need to made.

How to invoke the JSP page? The controller servlet can either include the output of a JSP page or forward the request to it. A servlet can only forward the request to a single JSP page but the JSP page can set the response headers enabling it to specify, for example, the content type. A servlet can include the content of multiple JSP pages. For example, a servlet could create a web page by invoking a separate JSP page for each part (header, footer, navigation bar and main content area) of the web page. However, an included JSP page cannot set any of the response headers. Although for some applications it might make sense for the servlet to play a role in response generation, it is usually better to leave that up to JSP pages. In most cases the servlet should forward requests to JSP pages.

What is the relationship between URLs, servlets and servlet classes? A developer has a lot of flexibility when building a J2EE Web application. In the web application deployment descriptor, a servlet definition consists of a name, a servlet class name and other attributes including zero or more initialization parameters. The deployment descriptor also contains URL⇒servlet mapping rules. Consequently, there are four possible approaches:

- The web application defines a single URL that is mapped to one servlet, which is implemented by a single servlet class.
- The web application defines multiple URLs that are all mapped to a single servlet, which is implemented by a single servlet class.
- The web application defines multiple URLs that are each mapped to a different servlet. A single class implements all of the servlets.
- The web application defines multiple URLs that are each mapped to a different servlet. A different class implements each servlet.

A servlet class that handles requests for multiple URLs (all but the last approach) would typically use the URL and the request parameters to determine what to do.

Should the controller be implemented using a servlet or JSP page? You can implement the controller using a JSP page instead of a servlet. This approach has some benefits. A JSP page developer can us the tags for manipulating Java beans. You can create custom tags for use by the web page designers. However, there are drawbacks. You are unable to use inheritance to implement common behavior among pages. Since the primary purpose of JSP pages is to generate content, it is a little strange to use them in a non-presentation role. Also, as noted above, writing Java code inside a JSP page is less convenient that writing a class.

1.6. Resulting Context

Java developers and web page designers can work independently: Java developers on the servlets and web page designers on the JSP pages. The servlet contains the majority of the Java code including all of the error handling code. The servlet can inherit common behavior from a base class (the JSP specification explicitly discourages the use of inheritance). Each JSP page contains very little Java code and is testable in isolation and independently of the business logic. This pattern also enables a web application to easily support multiple client types – the servlet can select the version of the JSP page that is appropriate for the client.

The developer can use Cactus to write test cases for the servlet. However, these tests will be less effective if the servlet consists of a single service() method. It is important to refactor the service() method into multiple, smaller helper methods.

It introduces a new problem: the interface between each servlet and each JSP page that it invokes needs to be clearly defined.

1.7. Related Patterns

This pattern should be used in conjunction with the following other patterns:

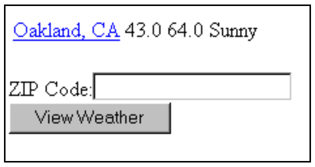
- ***Three Java Beans*** – defines the interface between a servlet and a JSP page.
- ***Action Servlet*** – describes how to implement a servlet using action beans.
- ***JSP page Test Driver*** – for testing the JSP pages.
- ***Formatter Bean*** – for simplifying JSP pages further.

1.8. Known Uses

- JSP Model 2 pattern is used widely (although not widely enough). For example, see the Sun Blueprints [SUN].
- BEA WebLogic Commerce Server Webflow mechanism [WLCS] can be considered an instance of this pattern.
- The Jakarta Struts framework [STRUTS].

1.9. Example

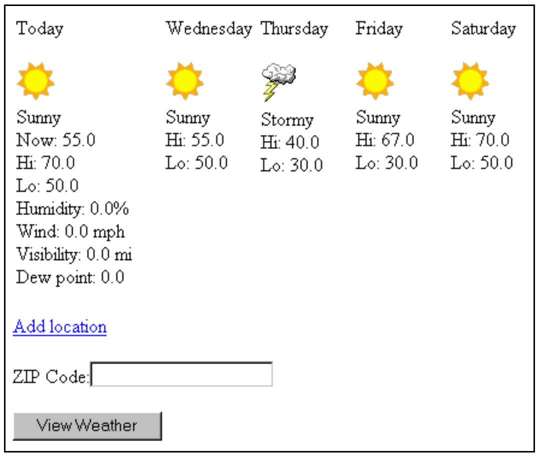
The weather application displays the current weather conditions and forecast for a selected location. Figure 4 and Figure 5 show the two screens that comprise the user interface.



Oakland, CA 43.0 64.0 Sunny

ZIP Code:

Figure 4: Weather Application Enter Location Screen



Today	Wednesday	Thursday	Friday	Saturday
Sunny	Sunny	Stormy	Sunny	Sunny
Now: 55.0	Hi: 55.0	Hi: 40.0	Hi: 67.0	Hi: 70.0
Hi: 70.0	Lo: 50.0	Lo: 30.0	Lo: 30.0	Lo: 50.0
Lo: 50.0				
Humidity: 0.0%				
Wind: 0.0 mph				
Visibility: 0.0 mi				
Dew point: 0.0				

Add location

ZIP Code:

Figure 5: Weather Application Display Weather Screen

The **EnterLocation** screen displays a summary of the weather for the user's favorite locations and prompts the user to enter either the zip code or city and state. The **DisplayWeather** screen displays the current weather conditions and the forecast for the next few days and enables the user to add the location to their list of favorites.

The web application retrieves the weather forecast from the back-end system through the WeatherService API, which is implemented by an enterprise java bean. Figure 6 shows the classes and interfaces that comprise this API.

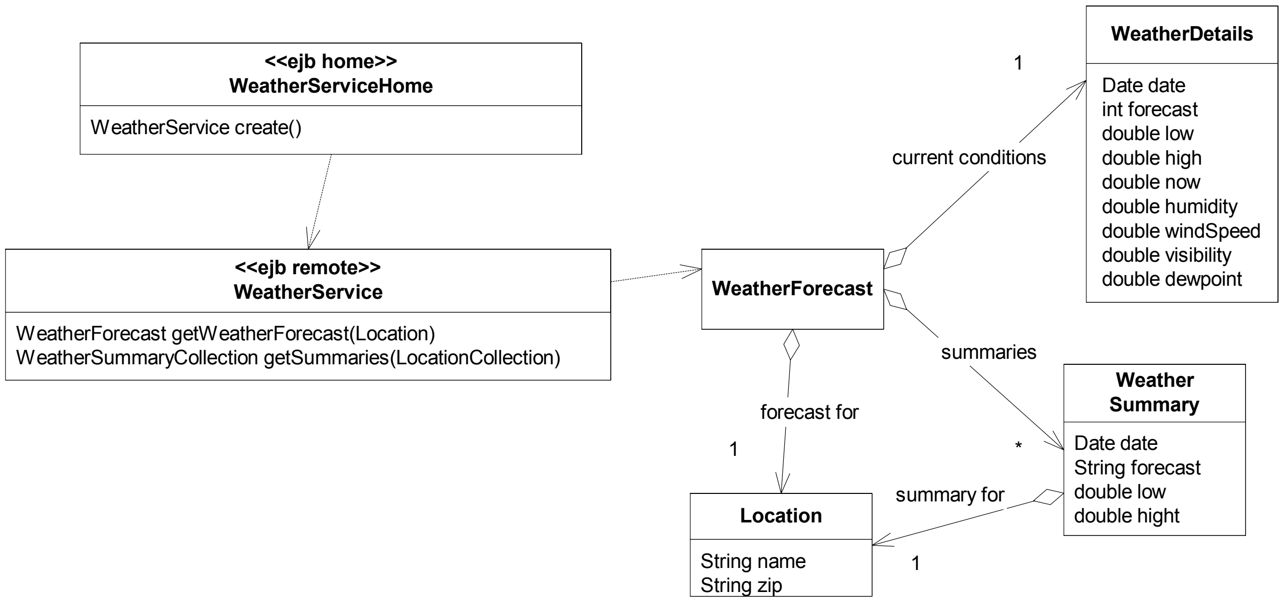


Figure 6: Weather System back-end

This API consists of the following classes:

- WeatherService – the remote interface for weather service EJB.
- WeatherServiceHome – the home interface for the EJB.
- WeatherForecast – current conditions for a location and summary forecasts for the next few days.
- WeatherDetails – detailed information for one day.
- WeatherSummary – summary information for a day.
- Location – zip code and name of the location.
- LocationCollection – collection of locations.

The weather web application has to handle three different requests:

- Display the **EnterLocation** screen – the browser sends this request when the user enters the URL for the application or clicks on a link.
- Display the weather for a location – the browser sends this request when the user either clicks on a favorite location or submits the form.
- Add Location to favorites – this request is sent by the browser when the user clicks on the *add location* link.

Figure 7 shows the servlets that handle these requests.

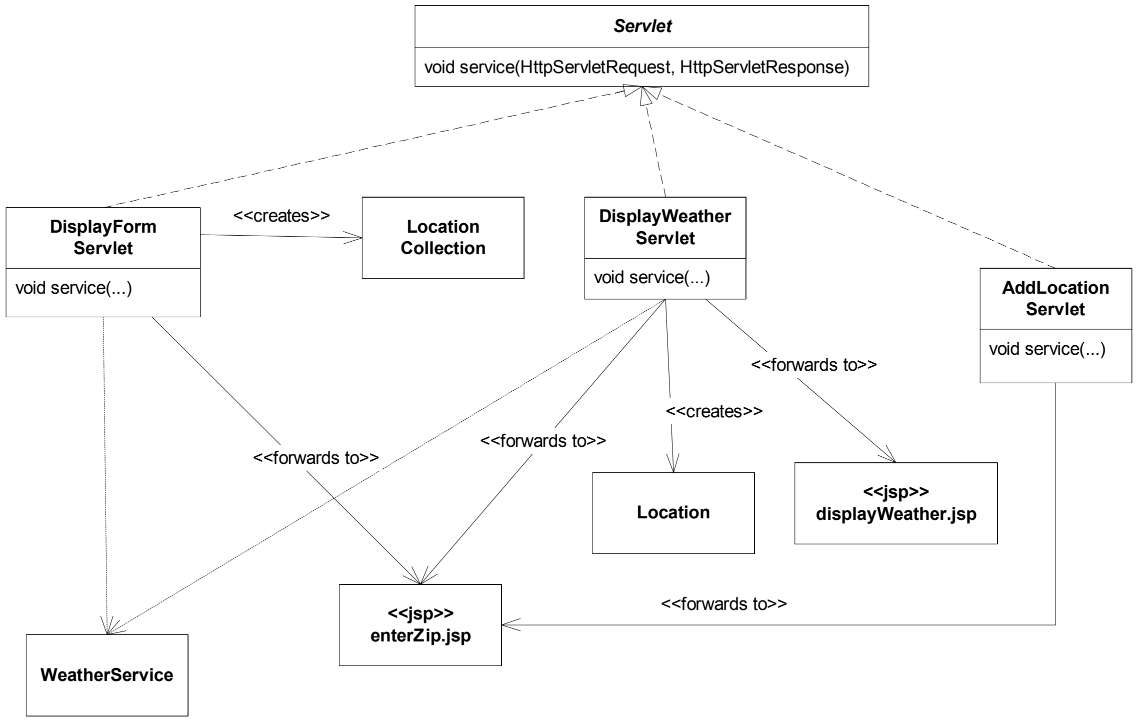


Figure 7: Weather Application Servlets

The weather web application defines three URLs: displayForm, displayWeather, addLocation. Each URL is mapped to a different servlet:

- displayForm URL ⇒ DisplayForm servlet, which calls the WeatherService EJB to retrieve the forecasts for the user's favorite locations and forwards the request to the JSP page enterZipCode.jsp to display the first screen.
- displayWeather URL ⇒ DisplayWeather servlet, which calls the WeatherService EJB to get the forecast for the selected location and forwards the request to the JSP page displayWeather.jsp to display the forecast. If the user enters a location that doesn't have a forecast then the DisplayWeather servlet redisplays the form with an error message.
- addLocation URL ⇒ AddLocation servlet, which adds the selected location to the user's list of favorites and then forwards the request to the DisplayForm servlet to redisplay the list.

The test suite for this application's presentation layer would use Cactus to test the servlets directly. Each servlet would have a corresponding test case class. The test suite would also contain some HttpUnit-based tests.

2. Three Java Beans

2.1. Context

You have applied the ***Controller Servlet*** pattern. The controller servlet is responsible for validating the request's parameters and invoking the business logic. The controller servlet then forwards the request to a JSP page passing to it the data returned by the business logic. There are two different mechanisms that the servlet could use to do this. It could store the data as attributes of the HttpServletRequest. Alternatively, it could store the data as attributes of the HttpSession but since the data is specific to a request it makes more sense to use the HttpServletRequest.

There might be multiple versions of each JSP page in order to support multiple devices.

2.2. Problem

How do you specify the data that the controller servlet passes to JSP pages that it invokes?

2.3. Forces

- A controller servlet often has to pass multiple values to a JSP page: some data to display, the previously entered form values (in order to redisplay the form when an error occurs) and zero or more error messages to display (an overall error message for the page and a separate message for each invalid form field).
- HttpServletRequest attributes are untyped (i.e. they are declared to be of type Object) and the JSP page has to downcast the values that are passed to it. There is no compile-time mechanism for ensuring that the controller servlet passes all of the values that the JSP page requires.
- There might be several different versions of a JSP page. A different person might develop each version. A JSP page developer needs to be able to easily determine the values that are passed to the JSP page without having to read through either the controller servlet source code or possibly out of date comments.

2.4. Solution

Specify the data that the servlet passes to the JSP page in terms of the following Java beans:

- A **display values** bean that contains the data that the JSP page displays.
- A **previous form values** bean that contains the previously entered form field values.
- An **error messages** bean that contains error message(s).

The servlet creates one or more of these beans and stores them as attributes of the HttpServletRequest. The JSP page then retrieves the beans from HttpServletRequest and accesses their properties. The JSP can access these beans using the <jsp:usebean/> tag with a request scope. Their properties of the beans can be accessed using the <jsp:getproperty/> tag.

Figure 8 shows the structure of this pattern.

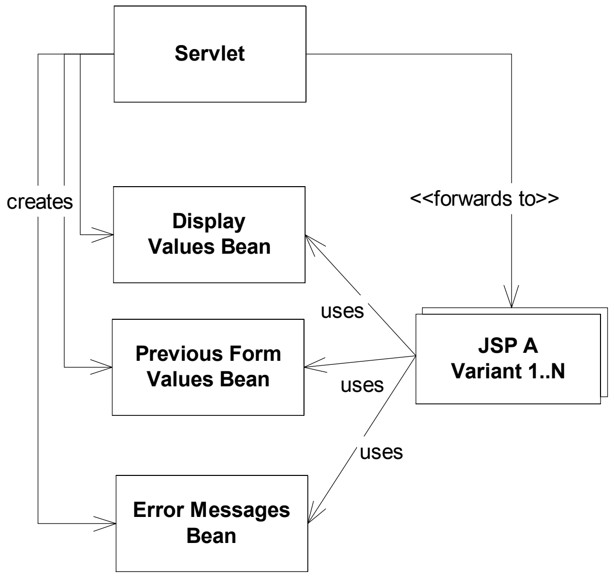


Figure 8: Structure of the Three Java Beans pattern

Figure 9 shows how the controller servlet forwards the request to a JSP page. This happens either when validation fails or after the servlet calls the business logic.

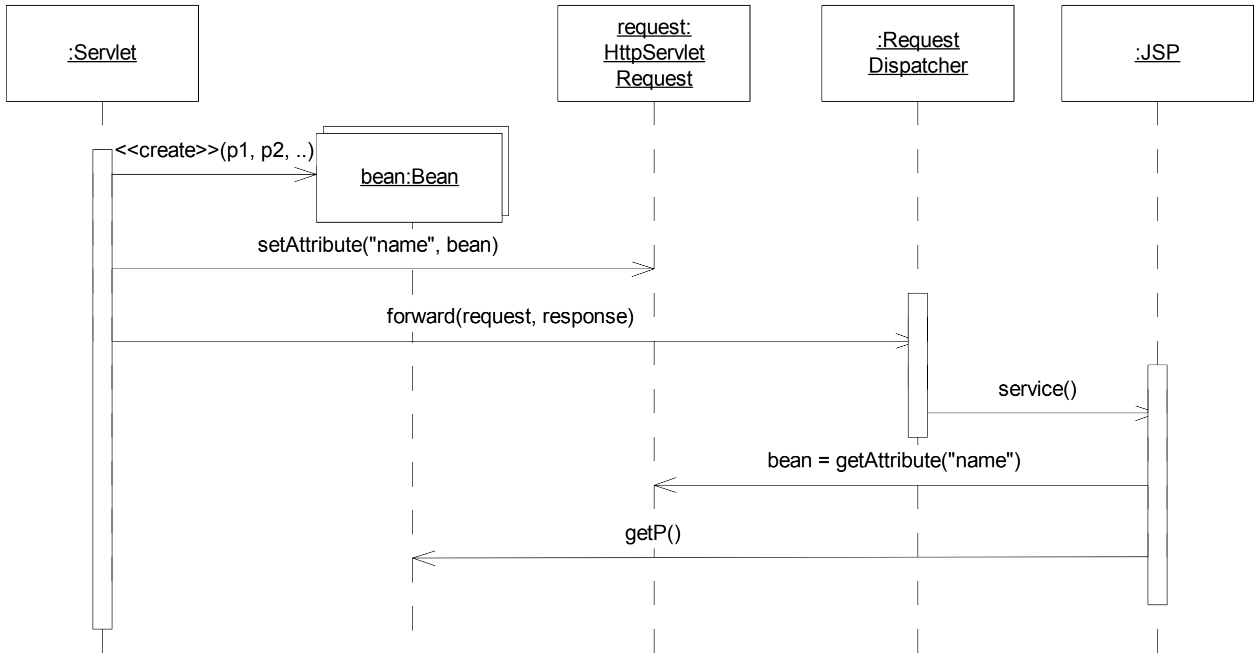

:Servlet
request: HttpServlet Request
:Request Dispatcher
:JSP
<<create>>(p1, p2, ..)
bean:Bean
setAttribute("name", bean)
forward(request, response)
service()
bean = getAttribute("name")
getP()

Figure 9: Three Java Beans Pattern collaborations

The sequence of events is as follows:

1. The servlet creates the beans, initializes their properties and stores them as attributes of the HttpServletRequest. It uses a RequestDispatcher to forward the request to the JSP page.

2. The JSP page retrieves the beans from the HttpServletRequest and uses their properties to generate the response.

One drawback of this pattern is that it requires more upfront work by the developer to define the bean classes when the benefits are only mostly apparent later in the development process.

2.5. Resulting Context

The values that each JSP page expects to be passed are clearly defined. The name and type of each value is defined. Furthermore, the beans' constructors can force the servlet to pass all of the required values. Any mismatch in the names and types of the value's beans properties will be caught at compile time. It allows for rapid changes of type and structure.

The only values that the JSP page has to downcast are the references to the three beans. Since this interface is unlikely to change during development once the JSP page and its value bean class is defined the likelihood of ClassCastException runtime error is relatively low. However, if the developer does make a mistake and passes the wrong type of bean, it will be detected the first time the JSP page is executed - thorough testing isn't required to detect this kind of major error.

The servlet developer and JSP page need to agree on the attribute names. This can be standardized across the application.

2.6. Related patterns

The ***Controller Servlet*** pattern creates the need for this pattern.

The HttpServletRequest attribute mechanism is an instance of the ***Property*** pattern [FOOTE].

2.7. Known Uses

The author has used this pattern on several different projects.

2.8. Variations

An application can combine the PreviousFormValues bean and the ErrorMessages bean into a single object. For example, BEA WebLogic Commerce Server [WLCS] stores the result of parameter validation in a ValidatedValues object, which is passed to the JSP page. It stores for each form field, the value entered by the user, the validation status (valid/invalid/missing) and an error message.

An application could pass a single bean to the JSP page. However, this bean will in practice aggregate the three different types of data: previous form values, display values and messages.

2.9. Example

The JSP page enterZip.jsp expects to be passed two values: a WeatherSummariesCollection containing the weather summaries for the user's favorite locations and an ErrorMessages object (similar to the WLCS ValidatedValues class) that contains both the error messages and the previously entered location name.

The following listing shows an excerpt from this JSP page:

```
<jsp:usebean id="values"
class="weather.weatherService.WeatherSummaryCollection"
scope="request"/>
<jsp:usebean id="messages"
class="actionFramework.actions.ErrorMessages" scope="request"/>

<%
Iterator it = messages.getMessages();
while (it.hasNext()) {
%>
<font color="red"><%= it.next() %></font>
<%
}
%>

<form method="post" action="displayWeather">
ZIP Code:<input name="zipCode" value="<%=
messages.getValue("zipCode")%>" ><font
color="red"><%=messages.getMessage("zipCode")%></font>
<br><input type="submit" value="View Weather">
</form>
…
```

The JSP page displayWeather.jsp expects to be passed a WeatherForecast object and an ErrorMessages object. The following listing shows an excerpt from this JSP page:

```
<jsp:usebean id="values" type="weather.weatherService.WeatherForecast"
scope="request"/>
<jsp:usebean id="messages"
class="actionFramework.actions.ErrorMessages" scope="request"/>
…
```

3. Encapsulate User Session Management

3.1. *Context*

You are developing a J2EE Web Application. HTTP is a stateless protocol, but web applications typically need to maintain session state for a user. The servlet API provides different ways of doing this including storing values as attributes of the HttpSession, or by using cookies. See [FOWLER] for a discussion of the pros and cons of the various techniques.

3.2. *Problem*

How do you represent a user session state?

3.3. *Forces*

- It is desirable to encapsulate the mechanism for maintaining the session state so that it can be changed without impacting the application.

- Decoupling those parts of the application that use the session state from the Servlet API will improve testability.
- HttpSession attributes are weakly typed (i.e. of type Object) and there is no compile-time type checking. Code that uses an HttpSession attribute is required to downcast it to the expected type. Furthermore, HttpSession attributes are simple name-value pairs – it is not possible for an attribute value to be computed dynamically. Similarly, when an attribute is changed it is not possible to execute any developer written code.

3.4. Solution

Define a user session manager class that encapsulates the user session management and is responsible for retrieving and saving a user session object that maintains the state. Figure 10 shows the structure of this pattern.

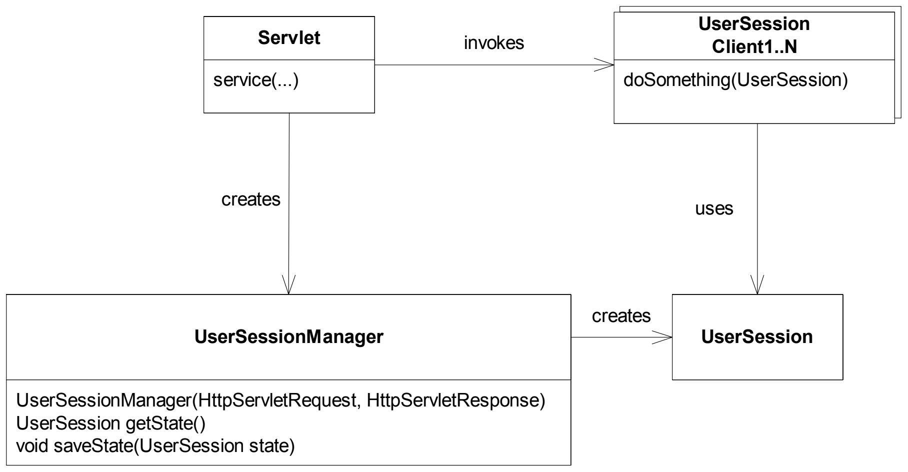


Figure 10: Structure of the Encapsulate User Session State Pattern

The responsibilities of each class are as follows:

- UserSession – provides getters and setters for accessing the session state.
- UserSessionManager – encapsulates how the session state is maintained. It might construct the session state from cookies each time or it might simply retrieve it from the HttpSession. The getState() method returns the UserSession. The saveState() method saves the UserSession if required. It might, for example, update some cookies.
- UserSessionClient – invoked by the servlet to help process the request. The UserSessionClient might, for example, validate HTTP request parameters or invoke the business logic. In order to perform its responsibilities it needs to access the session state and so invokes the getters and setters defined by the UserSession.

An application's presentation layer will typically only have one UserSessionManager class. If the UserSessionManager is stateless it can be implemented as a singleton. Otherwise, if the UserSessionManager maintains state for the duration of the request the servlet will create one each time.

Figure 11 shows how the controller servlet uses the UserSession and UserSessionManager classes to handle an HTTP request.

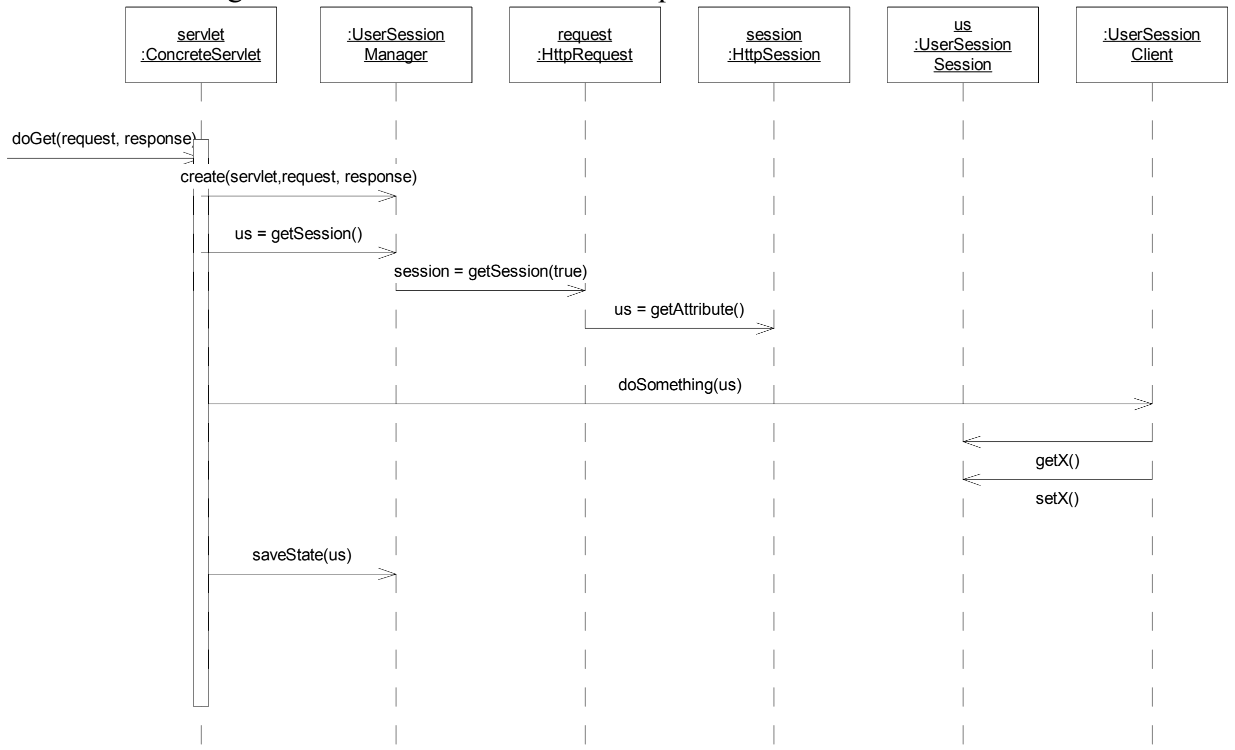


Figure 11: UserSession Pattern Collaborations

The sequence of events is as follows:

1. The servlet creates a UserSessionManager and gets the UserSession object.
2. The servlet invokes the UserSessionClient and passes to it the UserSession.
3. The servlet tells the UserSessionManager to save the UserSession.
4. The servlet invokes a JSP page to generate the response.

One drawback of this pattern is that the UserSession object contains state information for unrelated parts of an application and so might end up having a wide interface.

3.5. Resulting Context

Session state management is performed in a single class and the rest of the classes are unaware of the specific mechanism used. User session state is well defined and maintained in a single object. The UserSessionManager is the only class that has to downcast an HttpSession attribute. The rest of the code that accesses the session state is checked at compile time. Methods that manipulate the application state use the UserSession object rather than the servlet API and are easier to test.

3.6. Related Patterns

The HttpSession attribute mechanism is an instance of the ***Property*** pattern [FOOTE].

Because this pattern decouples the classes that use the session state from the servlet API, it enables the ***Action Servlet*** pattern to be used.

3.7. Known Uses

The author has used this pattern on several different projects.

3.8. Example

In the weather application, the session state consists of a list of the user's favorite locations and is stored in the HttpSession.

The following listing shows the WeatherSession class:

```
public class WeatherSession implements ApplicationSession {

    private ArrayList locations = new ArrayList();

    public Collection getLocations() {
        return locations;
    }

    public void addLocation(String location) {
        if (!locations.contains(location))
            locations.add(location);
    }
}
```

The following listing shows the WeatherSessionManager class:

```
public class WeatherSessionManager {

    private HttpServletRequest request;
    private HttpServletResponse response;
    private static final String APP_SESSION_NAME =
"ApplicationSession";

    public ApplicationSession getSession() {
        HttpSession session = request.getSession(true);
        ApplicationSession appSession =
(ApplicationSession)session.getAttribute(APP_SESSION_NAME);
        if (appSession == null) {
            appSession = makeApplicationSession();
            session.setAttribute(APP_SESSION_NAME, appSession);
        }
        return appSession;
    }

      public WeatherSessionManager(HttpServletRequest request,
HttpServletResponse response) {
        this.request = request;
        this.response = response;
    }

      protected ApplicationSession makeApplicationSession() {
            return new WeatherSession();
      }
}
```

3.9. Variations

The UserSession need not store the session state directly. It could, for example, be an interface with a concrete implementation class whose accessor methods simply call getAttribute() and setAttribute() on the HttpSession.

4. Action Servlet

4.1. Context

You have applied the ***Controller Servlet***, ***Encapsulate User Session Management*** and the ***Three Java Beans*** patterns. The controller servlet is responsible for validating the request's parameters, invoking the business logic, creating the three Java beans that JSP page required and forwarding the request to the JSP page. This can require hundreds of lines of Java code.

A servlet, like all classes that use the Servlet API, can only run within the Web container. The Cactus framework can be used to write test cases for all of the servlet's methods except the servlet's service() method. This means that the service() must call helper methods to validate the parameters and invoke the business logic. The service() method can only be tested using HttpUnit-based test cases. However, verifying that the service() method is behaving correctly might be hard since the HttpUnit-based test case would have to examine the content generated by the JSP page invoked by the servlet in order to do this.

Cactus test cases run inside the application server's web container. The edit-compile-run cycle for code that must run inside an application server can be longer because each time a change is made the code must be redeployed on the server. This can require either restarting the server, which can easily take over a minute or, if possible, utilizing a vendor-specific hot deploy mechanism, which, even though it is much quicker than a server restart, might not be instantaneous.

4.2. Problem

How do you implement the controller servlet?

4.3. Forces

- You want to be able to write test code using JUnit-based test cases.
- You want to avoid writing methods that are hard to test.
- You want to be able to reuse components in multiple applications.
- You want the presentation layer to be easy to understand and maintain.
- You want to be able to quickly and easily make changes and test them.

4.4. Solution

Implement a servlet as a façade that creates, initializes and executes an Action bean, which is a command-like object [GOF] that doesn't use the servlet API. Figure 12 shows the structure of this pattern.

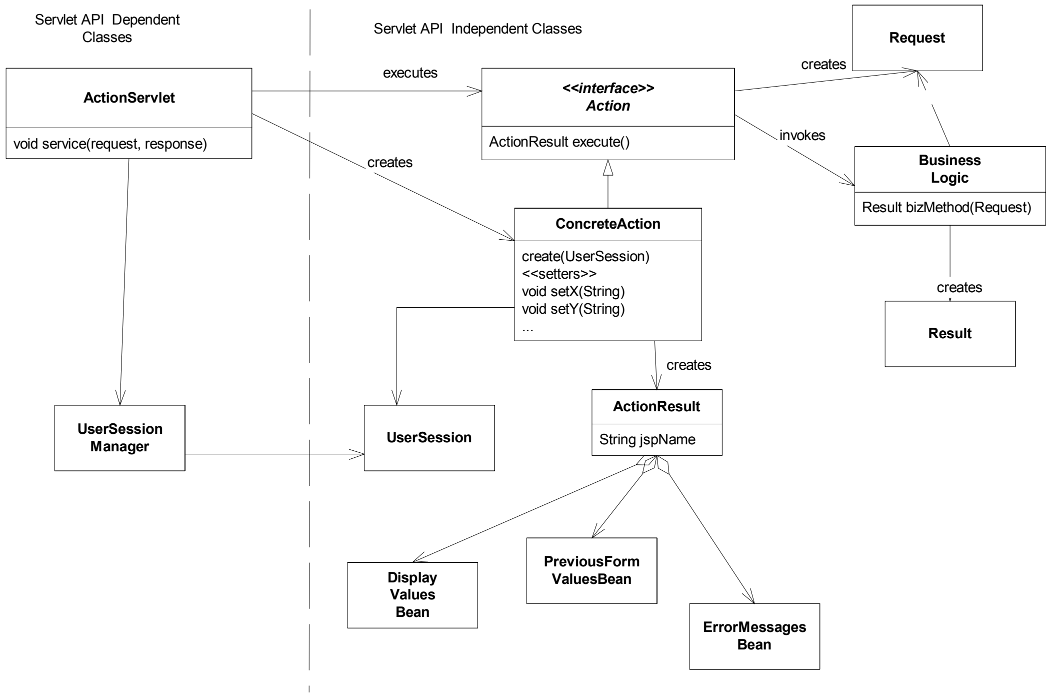


Figure 12: Structure of the Action Servlet pattern

The participants in the pattern are:

- ActionServlet
  - obtains the UserSession from the UserSessionManager
  - creates an Action and initializes its properties with the matching parameters from the request
  - executes the action
  - forwards the request to the specified JSP page
- Action
  - validates its properties (which correspond to the request's parameters)
  - invokes business logic and handles any exceptions that are thrown
  - returns an ActionResult
  - uses and updates the UserSession
- ActionResult
  - specifies which JSP page to display and the values to pass to it
- UserSession
  - maintains the user's session state
- UserSessionManager
  - manages the UserSession
- DisplayValuesBean, PreviousFormValuesBean, ErrorMessagesBean
  - these are beans from the ***Three Java Beans*** pattern
- Business Logic
  - the interface to the application's business logic
  - defines a bizMethod() that the controller servlet calls
- Request

    - passed as an argument to the business logic method
    - contains data derived from the HttpServletRequest parameters
- Result
    - returned by the business logic method

Figure 13 shows how a servlet that is implemented using this pattern handles an HTTP request.

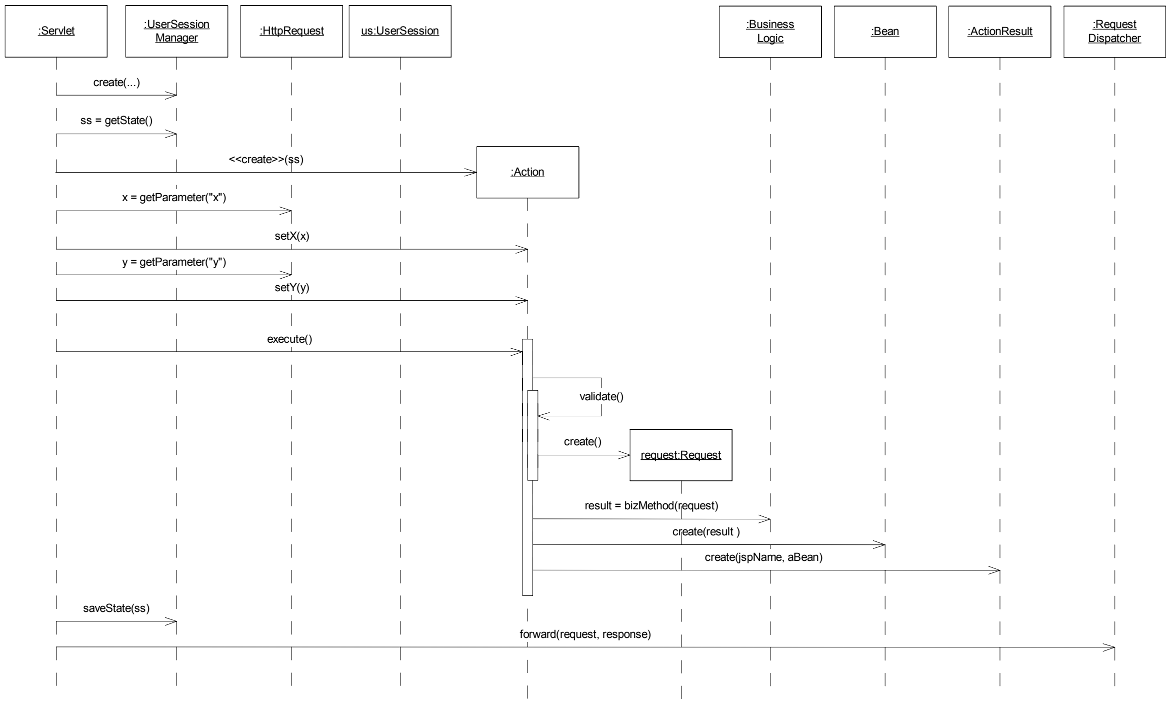


Figure 13: Action Servlet collaborations

The sequence of events is as follows:

1. The servlet creates a UserSessionManager and gets the UserSession.
2. It creates an Action passing the UserSession to Action's constructor.
3. It initializes each of the Action's properties with the corresponding parameter from the HttpServletRequest.
4. The servlet invokes execute() on the Action.
5. The Action validates its properties and invokes the business logic.
6. The Action returns an ActionResult object specifying which JSP page to use and the beans to pass to the JSP page.
7. The servlet passes the possibly updated UserSession back to the UserSessionManager.
8. The servlet then forwards the request to the specified JSP page passing the values returned by the Action.

This pattern does have some drawbacks. Sometimes the code that handles a request needs to access the servlet API classes and interfaces in order, for example, to access request's headers, cookies or perhaps servlet context init-parameters. Fortunately, this is relatively rare and actions can co-exist with regular servlets.

4.5. Resulting Context

The action classes don't use the servlet API, which means they can be executed outside of the web container. This makes them considerably easier to develop and unit test. Each action class has a well-defined interface and a small set of responsibilities, which makes it easier to understand and maintain. Although the business logic invoked by the Action might have side-affects, the incremental functionality directly implemented by the Action can be considered to be a function that computes an ActionResult, which improves testability. Test cases can verify the behavior of the Action by examining the ActionResut.

The Action Servlet is generic and can be reused across multiple applications.

4.6. Related Patterns

This pattern is enabled by the ***Controller Servlet***, ***Encapsulate User Session*** and the ***Three Java Beans*** patterns.

An Action could be considered to be a ***Command*** [GOF].

4.7. Known Uses

An example of this pattern is the WebWork framework [WEBWORK]. Developers write action classes that implement the WebWork's Action interface. A Dispatcher servlet creates an Action, initializes its properties with the corresponding request parameters and then invokes its execute() method, which returns the name of the JSP page to forward the request to.

A good example of a variation of this pattern is the Struts framework [STRUTS]. This framework has an ActionServlet class that uses Action classes to handle requests. This approach makes the design more modular and easier to understand since each Action class is relatively small and has a well-defined purpose. However, testing is still difficult since Struts Action classes use the HttpServletRequest and HttpServletResponse objects.

The author has used this pattern on several different projects.

4.8. Example

Figure 14 shows the action classes and their test cases for the weather application.

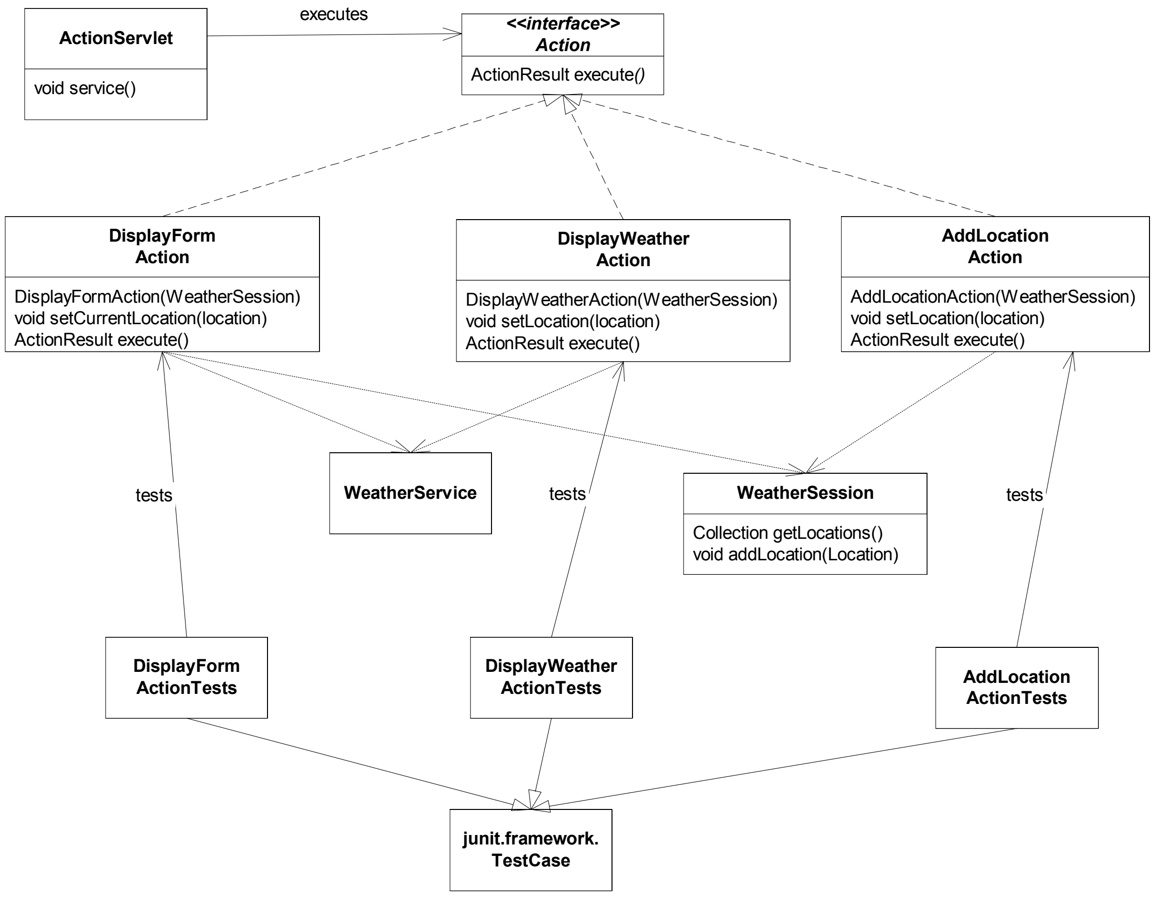

ActionServlet
void service()
executes
<<interface>>
Action
ActionResult execute()
DisplayForm Action
DisplayFormAction(WeatherSession)
void setCurrentLocation(location)
ActionResult execute()
DisplayWeather Action
DisplayWeatherAction(WeatherSession)
void setLocation(location)
ActionResult execute()
AddLocation Action
AddLocationAction(WeatherSession)
void setLocation(location)
ActionResult execute()
tests
WeatherService
tests
WeatherSession
Collection getLocations()
void addLocation(Location)
tests
DisplayForm ActionTests
DisplayWeather ActionTests
AddLocation ActionTests
junit.framework. TestCase

Figure 14: Weather Action beans

The responsibilities of each class are as follows:

- ActionServlet – creates and executes the Action specified by an init parameters.
- DisplayFormAction - handles requests for the displayForm URL. It calls the WeatherService to get weather summaries for the user's favorite locations. It returns an ActionResult that specifies the JSP page enterZip.jsp.
- DisplayWeatherAction - handles requests for the displayWeather URL. It calls the WeatherService to get the detailed weather forecast for the chosen location. It returns an ActionResult that specifies the JSP page displayWeather.jsp.
- AddLocationAction - adds the location to the list of user's favorites. It returns an ActionResult that specifies the JSP page enterZip.jsp.
- DisplayFormActionTests – JUnit TestCase class for DisplayFormAction.
- DisplayWeatherActionTests – test case class for DisplayWeatherAction.
- DisplayAddLocationActionTests – test case class for AddLocationAction.

The following listing shows the servlet definitions from the deployment descriptor:

```
    <servlet>
        <servlet-name>DisplayForm</servlet-name>
        <servlet-class>actionFramework.servlets.ActionServlet</servlet-
class>
      <init-param>
            <param-name>actionClass</param-name>
            <param-value>weather.ui.actions.DisplayFormAction</param-
value>
      </init-param>
    </servlet>

    <servlet>
        <servlet-name>DisplayWeather</servlet-name>
        <servlet-class>actionFramework.servlets.ActionServlet</servlet-
class>
      <init-param>
            <param-name>actionClass</param-name>
            <param-
value>weather.ui.actions.DisplayWeatherAction</param-value>
      </init-param>
    </servlet>

    <servlet>
        <servlet-name>AddLocation</servlet-name>
        <servlet-class>actionFramework.servlets.ActionServlet</servlet-
class>
      <init-param>
            <param-name>actionClass</param-name>
            <param-value>weather.ui.actions.AddLocationAction</param-
value>
      </init-param>
    </servlet>
```

The three servlet definitions all use the same Action class.

Each Action class has a test class that implements the JUnit-based test cases. Each test case consists of the following steps:

1. Create an Action object
2. Set its parameters
3. Invoke its execute() method
4. Verify that the ActionResult is correct

Some test cases would execute the action with valid parameters and others would use invalid parameters.

5. Formatter Bean

5.1. *Context*

You are writing a JSP page that is displaying the properties of a single object or the properties of a hierarchy of objects. Some of the properties are dates, numbers and money. A web application might have users in multiple countries, with different conventions for formatting these types of values.

5.2. *Problem*

How does the JSP page format values such as numbers and dates?

5.3. Forces

- A JSP page should format the date, money and number properties of an object using Java formatting classes such as SimpleDateFormat, and NumberFormat.
- The JSP page must use the appropriate locale and time zone.
- You need to be able to test the formatting code, yet Java code inside a JSP page is hard to test.
- Java developers and web page designers need to be able to work separately on different artifacts.
- You need to format values consistently throughout an application.

5.4. Solution

Define one or more formatter beans that use the Java formatting classes to format the numeric, date and other similar properties of an object. A formatter bean has read-only properties corresponding to the properties of the object it is formatting. The value of each formatter property is a formatted string value. Figure 15 shows the participants of this pattern.

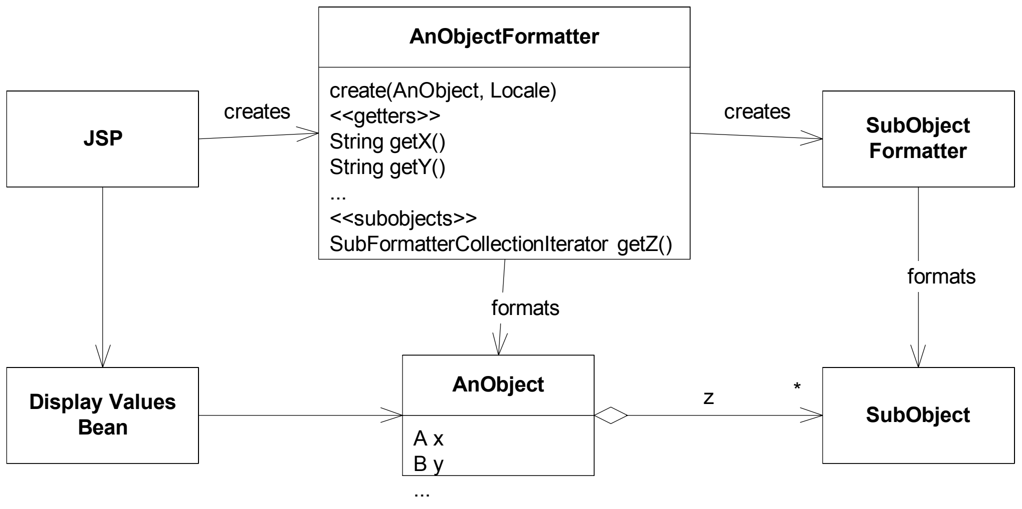


Figure 15: Formatting Java Bean participants

The JSP page will get an object from the **display values** bean and create the appropriate formatter class passing to it the object and perhaps the locale and time zone. The JSP page will then call the bean's getters and insert the values into the generated content. If the JSP page needs to display multiple sub-objects, the formatter will provide a method that returns an iterator through a collection of sub-object formatters. Figure 16 shows how the classes collaborate.

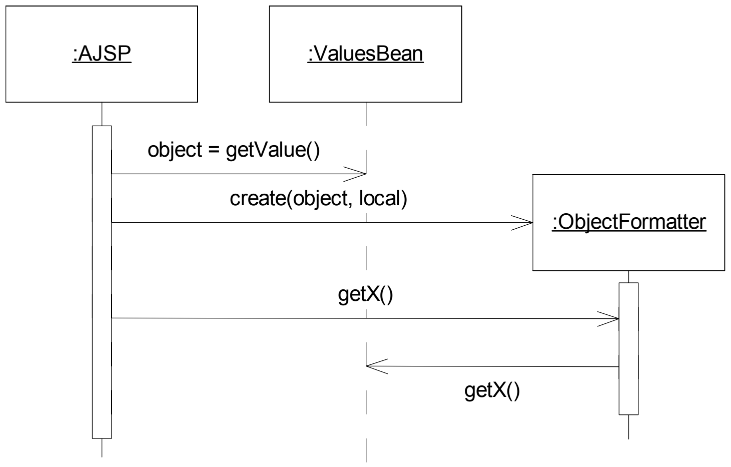


Figure 16: Formatting Bean Collaborations

A formatter bean might not use the Java formatting classes directly. Instead, it might call static formatting methods on a global formatting utility class. Using a global formatting class can help ensure consistency across an application.

One drawback of this pattern is that it requires more work by the developer to define the additional formatter beans and sometimes the benefits are not immediately apparent.

5.5. Resulting Context

This pattern reduces the amount of Java code inside a JSP page. The formatting classes are potentially reusable by multiple JSP pages. They can be easily tested using JUnit.

5.6. Related Patterns

This pattern is a specialization of the JSP Helper Bean pattern [JSPHELPER].

5.7. Known Uses

The author has used this pattern on several different projects.

5.8. Variations

An application could combine the values bean and the formatter into a single class. Although this simplifies the implementation, it might not always be possible since different versions of a JSP page might want to format the same value differently. For example, the HTML version might display a time and date as a "Wednesday July 5th at 10:15am" where as the WML version might display the date and time on separate lines.

5.9. Example

The JSP page displayWeather.jsp uses the formatting classes shown in Figure 17.

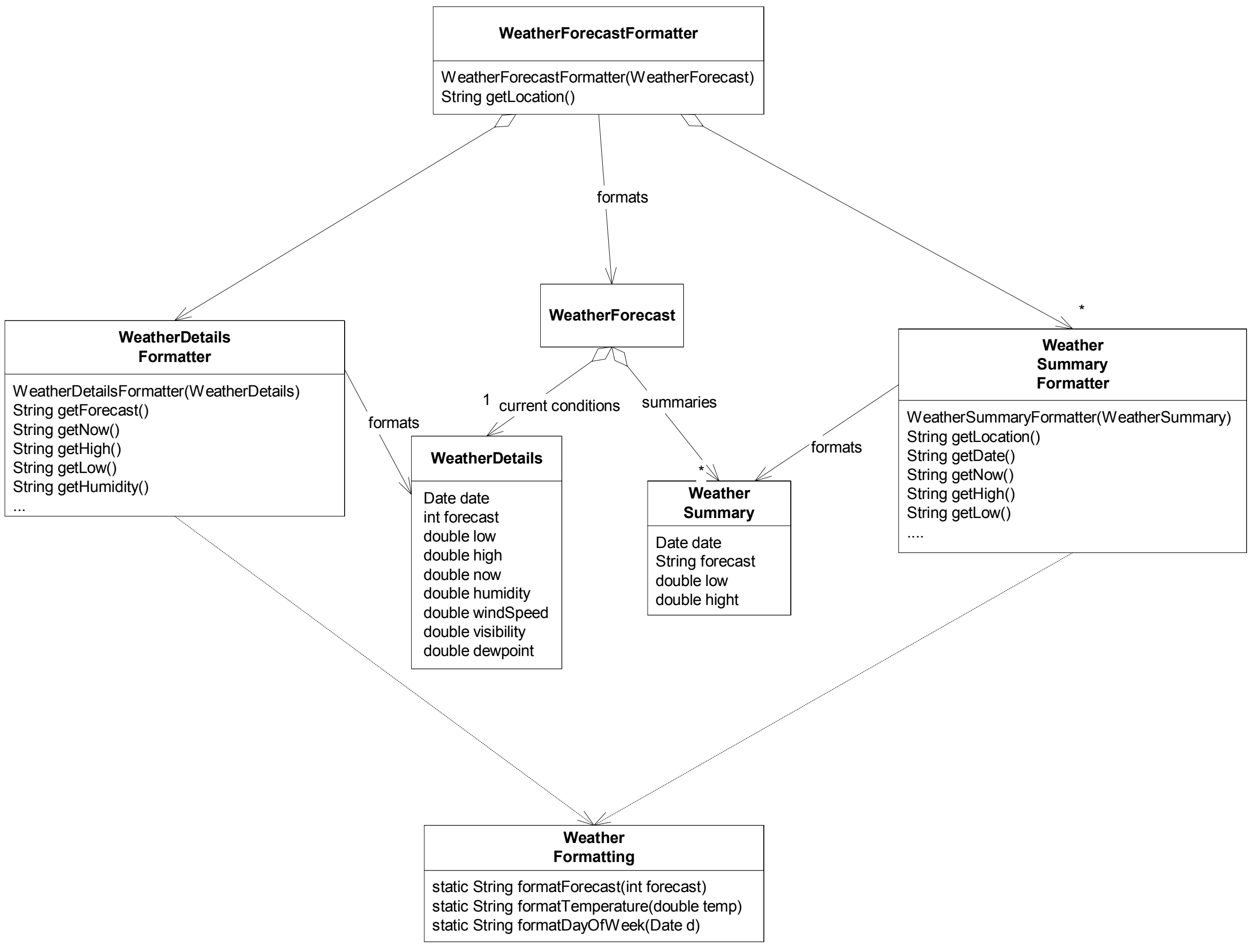


Figure 17: Weather formatting classes

The WeatherForecastFormatter, WeatherDetailsFormatter and WeatherSummaryFormatter classes format the WeatherForecast, WeatherDetails and WeatherSummary classes. The class WeatherFormatting is a utility class that defines some general purpose formatting methods.

The JSP page enterZip.jsp uses the WeatherSummaryFormatter.

6. JSP Page Test Driver

6.1. Context

You are developing a web application and you have applied the ***Controller Servlet*** and the ***Three Java Beans*** patterns. A JSP page is only responsible for content generation. A ***Controller Servlet*** passes to the JSP page three Java beans that contain the data that the JSP page needs. Web page designers who are not Java programmers are responsible for creating and maintaining the HTML/XML content in the JSP pages.

6.2. Problem

How do you enable web page designers to quickly and easily verify that their JSP pages generate the correct content?

6.3. Forces

- Content pages often undergo lots of minor, last minute changes as the web page designers tweak them to make them work correctly with multiple browsers, monitor sizes and operating systems.
- It is essential that the web page designers are able to easily and repeatedly test their work.
- JSP pages (like all other code) need to be tested against a wide range of input values in order to ensure adequate coverage.
- Manually navigating to the screen built by the JSP page you want to test is tedious.
- Setting up the exact test conditions can be difficult.
- It is desirable to be able to test JSP pages without relying on a working back-end system.

6.4. Solution

Write a test driver servlet that creates the beans that the JSP page expects to be passed, initializes them with test data and then invokes the JSP page. Figure 18 shows the structure of this pattern.

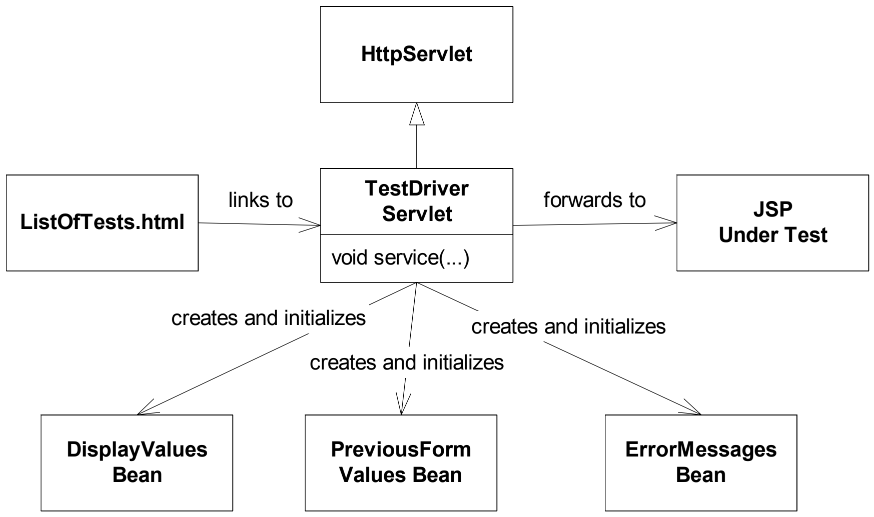


Figure 18: JSP Unit test structure

The HTML page ListOfTests.html displays the list of available tests as hyperlinks. A web page designer can easily test their changes by clicking on each of the links. Clicking on a link executes the test case by invoking the TestDriverServlet. The link has a testCase parameter, which tells the TestDriverServlet which test data to use to create the beans that it passes to the JSP page. After creating the beans, the servlet forwards the request to the JSP page.

6.5. Resulting Context

It is extremely easy for a web page designer to execute all of the test cases for a JSP page. Although, the primary purpose of the test is to enable the web page designers to view their JSP pages, it can sometimes be useful to also execute these tests as part of an automated test suite using HttpUnit.

A functioning back-end system is not required since the TestDriverServlet directly creates the beans the JSP page needs.

The TestDriverServlet can be implemented using the ***Action Servlet*** pattern.

Creating and maintaining the potentially complex test data for each test case is an issue. An application could create the objects in a number of ways: directly using Java code; read them from an XML file or load them from a database.

6.6. Related Patterns

This pattern is enabled by the ***Controller Servlet*** pattern.

6.7. Known Uses

The author has used this pattern on several different projects.

6.8. Example

The JSP page enterZip.jsp requires the following test cases:

- No favorite locations
- One favorite location
- Two favorite locations
- Invalid zip entered

The HTML page EnterZipCodeTests.html displays a list of these test cases. Each test case is a link to the servlet TestDisplayFormJSP, which in turn is implemented using the Action bean TestDisplayFormJSPAction. The following listing shows an excerpt from the web application's web.xml file:

```
<servlet>
      <servlet-name>TestDisplayFormJSP</servlet-name>
      <servlet-class>actionFramework.servlets.ActionServlet</servlet-
class>
      <init-param>
            <param-name>actionClass</param-name>
            <param-
value>weather.ui.actions.TestDisplayFormJSPAction</param-value>
      </init-param>
</servlet>
…
<servlet-mapping>
        <servlet-name>TestDisplayFormJSP</servlet-name>
        <url-pattern>testDisplayFormJSP</url-pattern>
    </servlet-mapping>
<servlet>
```

The following listing shows the source for TestDisplayFormJSPAction:

```
public class TestDisplayFormJSPAction implements Action {

    private WeatherSession session;
      private String testCase;

    public TestDisplayFormJSPAction(WeatherSession session) {
        this.session = session;
    }

      public void setTestCase(String testCase) {
            this.testCase = testCase;
      }

    protected ActionResult execute() {
        ActionResult ar = new ActionResult("enterZipCode");
            WeatherSummaryCollection summaries = new
WeatherSummaryCollection();
            String location = "Oakland, CA";
            Date now = new Date();
            if (testCase.equals("none")) {
            } else if (testCase.equals("one")) {
            summaries.add(new WeatherSummary(new Location(location,
null), now, WeatherConstants.SUNNY, 43, 64));
            } else if (testCase.equals("two")) {
            summaries.add(new WeatherSummary(new Location(location,
null), now, WeatherConstants.SUNNY, 43, 64));
            summaries.add(new WeatherSummary(new Location(location,
null), now, WeatherConstants.SUNNY, 43, 64));
            } else if (testCase.equals("error")) {
                  ActionMessages messages = new ActionMessages();
                  Validator validator = new Validator(messages);
                  messages.invalidField("zipCode", "123456", "Invalid
Location");
                  messages.addMessage("An error occurred");
                  ar.setMessages(messages);
            }
            ar.setValues(summaries);
            return ar;
      }
}
```

The action returns an ActionResult that specifies the JSP page enterZip.jsp. The action's testCase property is initialized from the request's testCase parameter and determines which test data to use.

7. Using this Pattern Language

Developers can easily apply this pattern language when developing brand new applications or implementing brand new functionality in an existing application.

Developers can also use these patterns to refactor existing JSP pages. For example, the ***Formatting Bean*** pattern can be applied to move Java code out of a JSP page. The ***Controller Servlet*** pattern can be applied to split a JSP page into a controller JSP page and one or more presentation JSP pages.

The pattern language can also be used to guide the major restructuring of an entire JSP-based application in order, for example, to support mobile devices. The patterns can be applied in the following order:

1. ***Controller Servlet*** - divide each JSP page into multiple JSP pages: a controller and one or more presentation JSP pages.
2. ***Three Java Beans*** - define the interface between the controller JSP page and the presentation JSP pages.
3. ***Encapsulate User Session Management*** – create a session state manager that encapsulates the session management policy.
4. ***JSP Page Test Driver*** – write unit tests for the presenter JSP pages.
5. ***Action Servlet*** - extract the action classes from the controller JSP pages. Initially, the JSP pages could execute the action classes. However, at some point in time the JSP page controllers can be replaced by a single controller servlet. This requires the URLs that reference the JSP pages directly to be replaced with URLs that mapped to the controller servlet.

8. Summary

Developers can use JSP pages and servlets to quickly build a presentation layer that supports web clients. However, they often ignore important design principles and end up building applications that are hard to maintain and test. For example, a commonly used approach is to build the entire presentation layer using JSP pages that invoke the business logic directly. This means that web page designers and Java developers are unable to work independently. Effective testing is difficult since the only way to test a JSP page is to send an HTTP request and examine the content it generates. Furthermore, because applications written this way don't cleanly separate content and application logic, it is difficult to extend them to support mobile devices, which is becoming an increasingly important requirement.

This paper presented a pattern language consisting of six patterns for building maintainable and testable presentation layers for applications that support both desktop and mobile clients. These patterns address testability, maintainability, reusability and adaptability in the following ways:

- **Testability** - The ***Controller Servlet*** pattern improves testability by making the servlet solely responsible for calling the back-end. Web page designers can test JSP pages without a working back-end. The ***Encapsulate User Session Management***, ***Action Servlet*** and ***Formatter Bean*** patterns further enhance testability by minimizing the number of classes that use the servlet API. ***JSP Page Test Driver*** outlines an effective testing strategy for JSP pages.
- **Maintainability** – Maintainability is significantly improved in a number of ways. The ***Formatter Bean***, and ***Controller Servlet*** patterns minimize the amount of code in the JSP pages. The ***Three Java Beans*** pattern specifies a well-defined interface for each JSP page.
- **Reusability** – The ***Action Servlet*** pattern defines a Controller servlet and Action classes that are reusable by multiple applications.

- **Adaptability** – The ***Controller Servlet*** pattern makes it straightforward to support multiple types of client device and multiple languages simply by using multiple sets of JSP pages.

Some patterns enable other patterns. Figure 19 shows the dependencies between them.

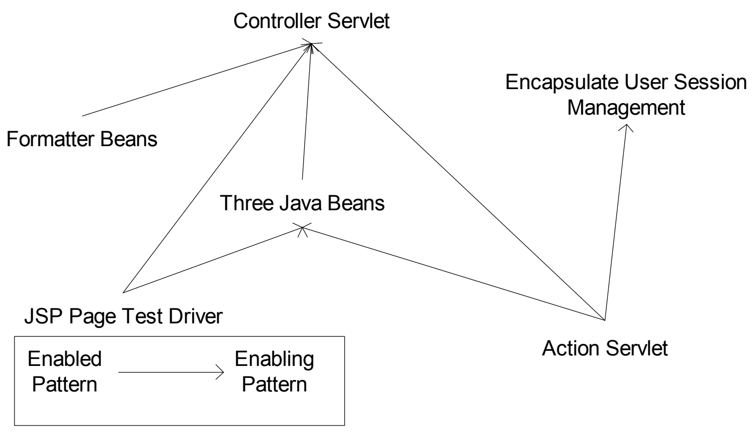


Figure 19: Dependencies between patterns

In this diagram, a pattern points to the patterns that enable it.

Applying the pattern language replaces each large monolithic JSP page with several components including a simpler JSP page that is only responsible for content generation, a controller servlet and several helper classes:

- Action beans – command-like classes that perform validation and invoke business logic
- UserSession – encapsulates the session state
- UserSessionManager – responsible for retrieving and saving the UserSession
- Values, messages and old form values beans – used by the controller servlet to pass data to the JSP page
- Formatter beans – used by JSP pages to format values, such as dates and numbers

Each helper class has a well-defined interface and implements a small well-defined piece of functionality. Most of the helper classes don't use the Servlet API classes and interfaces, which means that developing test cases for them is straightforward. Furthermore, this componentization of the presentation layer of the web-based application makes it possible to modify the behavior of the application without modifying any existing code. New behavior can be plugged in by changing a configuration file.

The controller servlet and most of the helper classes are reusable by multiple applications.

Developers can easily use these patterns when writing brand new applications. They can also apply them to improve the design of an existing application.

9. References

- [CACTUS] – Cactus test framework, http://jakarta.apache.org/commons/cactus/index.html

- [DOM] – Document Object Model, www.w3.org/DOM
- [GOF] – Design Patterns – Elements of Reusable Object-Oriented Software, Gamma et al
- [FOOTE] – Meta data and Active Models, Brian Foote and Joseph Yoder, http://www.laputan.org
- [FOWLER] – Martin Fowler, Information Systems Architecture, http://www.martinfowler.com/isa/index.html
- [HTTPUNIT], HttpUnit API, http://httpunit.sourceforge.net
- [J2EE] – J2EE Specification, http://www.javasoft.com/j2ee
- [JUNIT] – JUNIT testing framework, http://www.junit.org
- [JSP] – JavaServer Pages Specification, http://www.javasoft.com/jsp
- [STRUTS] – Jakarta Struts framework, http://jakarta.apache.org/struts/index.html
- [SUN] Developing Enterprise Applications, http://www.javasoft.com/j2ee/blueprints
- [WEBWORK] – WebWork framework, http://sourceforge.net/projects/webwork
- [WLCS] – WebLogic Commerce Server, http://e-docs.bea.com/

10. Acknowledgements

Thanks to Ashok Anand, Bill Burdick, Dragos A. Manolescu, Rich MacDonald, Georgia McNamara, Fritz Passow, Laura Richardson, Scott Shaw and Bjorn Wennerstrom for their constructive comments on drafts of this paper.