Presentation Tier Design Considerations and Bad Practices

Topics in This Chapter

- Presentation Tier Design Considerations
- Presentation Tier Bad Practices

Chapter 3

Presentation Tier Design Considerations

When developers apply the presentation patterns that appear in the catalog in this book, there will be adjunct design issues to consider. These issues relate to designing with patterns at a variety of levels, and they may affect numerous aspects of a system, including security, data integrity, manageability, and scalability. We discuss these issues in this chapter.

Although many of these design issues could be captured in pattern form, we chose not to do so because they focus on issues at a lower level of abstraction than the presentation patterns in the catalog. Rather than documenting each issue as a pattern, we have chosen to document them more informally: We simply describe each issue as one that you should consider when implementing systems based on the pattern catalog.

Session Management

The term *user session* describes a conversation that spans multiple requests between a client and a server. We rely on the concept of user session in the discussion in the following sections.

Session State on Client

Saving session state on the client involves serializing and embedding the session state within the view markup HTML page that is returned to the client.

There are benefits to persisting session state on the client:

- It is relatively easy to implement.
- It works well when saving minimal amounts of state.

Additionally, this strategy virtually eliminates the problem of replicating state across servers in those situations that implement load balancing across physical machines.

There are two common strategies for saving session state on the client—HTML hidden fields and HTTP cookies—and we describe these strategies below. A third strategy entails embedding the session state directly into the URIs referenced in each page (for example, `<form action=someServlet?var1=x&var2=y method=GET>`). Although this third strategy is less common, it shares many of the limitations of the following two methods.

HTML Hidden Fields

Although it is relatively easy to implement this strategy, there are numerous drawbacks to using HTML hidden fields to save session state on the client. These drawbacks are especially apparent when saving large amounts of state. Saving large amounts of state negatively affects performance. Since all view markup now embeds or contains the state, it must traverse the network with each request and response.

Additionally, when you utilize hidden fields to save session state, the persisted state is limited to string values, so any object references must be “stringified”. It is also exposed in clear text in the generated HTML source, unless specifically encrypted.

HTTP Cookies

Similar to the hidden fields strategy, it is relatively easy to implement the HTTP cookies strategy. This strategy unfortunately shares many of the same drawbacks as well. In particular, saving large amounts of state causes performance to suffer, because all the session state must traverse the network for each request and response.

We also run into size and type limitations when saving session state on the client. There are limitations on the size of cookie headers, and this limits the amount of data that can be persisted. Moreover, as with hidden fields, when you use cookies to save session state, the persisted state is limited to stringified values.

Security Concerns of Client-Side Session State

When you save session state on the client, security issues are introduced that you must consider. If you do not want your data exposed to the client, then you need to employ some method of encryption to secure the data.

Although saving session state on the client is relatively easy to implement initially, it has numerous drawbacks that take time and thought to overcome. For projects that deal with large amounts of data, as is typical with enterprise systems, these drawbacks far outweigh the benefits.

Session State in the Presentation Tier

When session state is maintained on the server, it is retrieved using a session ID and typically persists until one of the following occurs:

- A predefined session timeout is exceeded.
- The session is manually invalidated.
- The state is removed from the session.

Note that after a server shutdown, some in-memory session management mechanisms may not be recoverable.

It is clearly preferable for applications with large amounts of session state to save their session state on the server. When state is saved on the server, you are not constrained by the size or type limitations of client-side session management. Additionally, you avoid raising the security issues associated with exposing session state to the client, and you do not have the performance impact of passing the session state across the network on each request.

You also benefit from the flexibility offered by this strategy. By persisting your session state on the server, you have the flexibility to trade off simplicity versus complexity and to address scalability and performance.

If you save session state on the server, you must decide how to make this state available to each server from which you run the application. This issue is one that requires you to deal with the replication of session state among clustered software instances across load-balanced hardware, and it is a multidimensional problem. However, numerous application servers now provide a variety of out-of-the-box solutions. There are solutions available that are above the application server level. One such solution is to maintain a "sticky" user experience, where you use traffic management software, such as that available from Resonate [Resonate], to route users to the same server to handle each request in their session. This is also referred to as *server affinity*.

Another alternative is to store session state in either the business tier or the resource tier. Enterprise JavaBeans components may be used to hold session state in the business tier, and a relational database may be used in the resource tier. For more information on the business-tier option, please refer to "Using Session Beans" on page 55.

Controlling Client Access

There are numerous reasons to restrict or control client access to certain application resources. In this section, we examine two of these scenarios.

One reason to restrict or control client access is to guard a view, or portions of a view, from direct access by a client. This issue may occur, for example, when only registered or logged-in users should be allowed access to a particular view, or if access to portions of a view should be restricted to users based on role.

After describing this issue, we discuss a secondary scenario relating to controlling the flow of a user through the application. The latter discussion points out concerns relating to duplicate form submissions, since multiple submissions could result in unwanted duplicate transactions.

Guarding a View

In some cases, a resource is restricted in its entirety from being accessed by certain users. There are several strategies that accomplish this goal. One is including application logic that executes when the controller or view is processed, disallowing access. A second strategy is to configure the runtime system to allow access to certain resources only via an internal invocation from another application resource. In this case, access to these resources must be routed through another presentation-tier application resource, such as a servlet controller. Access to these restricted resources is not available via a direct browser invocation.

One common way of dealing with this issue is to use a controller as a delegation point for this type of access control. Another common variation involves embedding a guard directly within a view. We cover controller-based resource protection in "Presentation Tier Refactorings" on page 73 and in the patterns catalog, so we will focus here on view-based control strategies. We describe these strategies first, before considering the alternative strategy of controlling access through configuration.

Embedding Guard Within View

There are two common variations for embedding a guard within a view's processing logic. One variation blocks access to an entire resource, while the other blocks access to portions of that resource.

Including an All-or-Nothing Guard per View

In some cases, the logic embedded within the view processing code allows or denies access on an all-or-nothing basis. In other words, this logic prevents a particular user from accessing a particular view in its entirety. Typically, this type of guard is better encapsulated within a centralized controller, so that the logic is not sprinkled throughout the code. This strategy is reasonable to use when only a small fraction of pages need a guard. Typically, this scenario occurs when a nontechnical individual needs to rotate a small number of static pages onto a site. If the client must still be logged into the site to view these pages, then add a custom tag helper to the top of each page to complete the access check, as shown in Example 3.1.

Example 3.1 Including an All-or-Nothing Guard per View

```
<%@ taglib uri="/WEB-INF/corej2eetaglibrary.tld"
  prefix="corePatterns" %>

<corePatterns:guard/>
<HTML>
.
.
.
</HTML>
```

Including a Guard for Portions of a View

In other cases, the logic embedded within the view processing code simply denies access to portions of a view. This secondary strategy can be used in combination with the previously mentioned all-or-nothing strategy. To clarify this discussion, let's use an analogy of controlling access to a room in a building. The all-or-nothing guard tells users whether they can walk into the room or not, while the secondary guard logic tells users what they are allowed to see once they are in the room. Following are some examples of why you might want to utilize this strategy.

Portions of View Not Displayed Based on User Role

A portion of the view might not be displayed based on the user's role. For example, when viewing her organizational information, a manager has access to a subview dealing with administering review materials for her employees. An employee might only see his own organizational information, and be restricted from the portions of the user interface that allow access to any review-related information, as shown in Example 3.2.

Example 3.2 Portions of View Not Displayed Based on User Role

```
<%@ taglib uri="/WEB-INF/corej2eetaglibrary.tld"
  prefix="corePatterns" %>

<HTML>
.
.
.
<corePatterns:guard role="manager">
<b>This should be seen only by managers!</b>
<corePatterns:guard/>
.
.
.
</HTML>
```

Portions of View Not Displayed Based on System State or Error Conditions

Depending on the system environment, the display layout may be modified. For example, if a user interface for administering hardware CPUs is used with a single-CPU hardware device, portions of the display that relate solely to multiple CPU devices may not be shown.

Guarding by Configuration

To restrict the client from directly accessing particular views, you can configure the presentation engine to allow access to these resources only via other internal resources, such as a servlet controller using a RequestDispatcher. Additionally, you can leverage the security mechanisms that are built into the Web container, based on the servlet specification, version 2.2 and later. Security constraints are defined in the deployment descriptor, called `web.xml`.

The *basic* and *form-based* authentication methods, also described in the Servlet specification, rely on this security information. Rather than repeat the specification here, we refer you to the current specification for details on these methods. (See *http://java.sun.com/products/servlet/index.html*.)

So that you understand what to expect when adding declarative security constraints to your environment, we present a brief discussion of this topic and how it relates to all-or-nothing guarding by configuration. Finally, we describe one simple and generic alternative for all-or-nothing protection of a resource.

Resource Guards via Standard Security Constraints

Applications may be configured with a security constraint, and this declarative security may be used programmatically to control access based on user roles. Resources can be made available to certain roles of users and disallowed to others. Moreover, as described in “Embedding Guard Within View” on page 39, portions of a view can be restricted based on these user roles as well. If there are certain resources that should be disallowed in their entirety for all direct browser requests, as in the all-or-nothing scenario described in the previous section, then those resources can be constrained to a security role that is not assigned to any users. Resources configured in this manner remain inaccessible to all direct browser requests, as long as the security role remains unassigned. See Example 3.3 for an excerpt of a `web.xml` configuration file that defines a security role to restrict direct browser access.

The role name is “sensitive” and the restricted resources are named `sensitive1.jsp`, `sensitive2.jsp`, and `sensitive3.jsp`. Unless a user or group is assigned the “sensitive” role, then clients will not be able to directly access these Java Server Pages (JSPs). At the same time, since internally dispatched requests are not restricted by these security constraints, a request that is handled initially by a servlet controller and then forwarded to one of these three resources will indeed receive access to these JSPs.

Finally, note that there is some inconsistency in the implementation of this aspect of the Servlet specification version 2.2 across vendor products. Servers supporting Servlet 2.3 should all be consistent on this issue.

Example 3.3 Unassigned Security Role Provides All-or-Nothing Control

```
<security-constraint>
      <web-resource-collection>
      <web-resource-name>SensitiveResources
  </web-resource-name>
      <description>A Collection of Sensitive Resources
  </description>
        <url-pattern>/trade/jsp/internalaccess/
  sensitive1.jsp</url-pattern>
    <url-pattern>/trade/jsp/internalaccess/
  sensitive2.jsp</url-pattern>
    <url-pattern>/trade/jsp/internalaccess/
  sensitive3.jsp</url-pattern>
        <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>sensitive</role-name>
    </auth-constraint>
  </security-constraint>
```

Resource Guards via Simple and Generic Configuration

There is a simple and generic way to restrict a client from directly accessing a certain resource, such as a JSP. This method requires no configuration file modifications, such as those shown in Example 3.3. This method simply involves placing the resource under the `/WEB-INF/` directory of the Web application. For example, to block direct browser access to a view called `info.jsp` in the `securityissues` Web application, we could place the JSP source file in the following subdirectory: `/securityissues/WEB-INF/internalaccessonly/info.jsp`.

Direct public access is disallowed to the `/WEB-INF/` directory, its subdirectories, and consequently to `info.jsp`. On the other hand, a controller servlet can still forward to this resource, if desired. This is an all-or-nothing method of control, since resources configured in this manner are disallowed in their entirety to direct browser access.

For an example, please refer to “Hide Resource From a Client” on page 100.

Duplicate Form Submissions

Users working in a browser client environment may use the Back button and inadvertently resubmit the same form they had previously submitted, possibly invoking a duplicate transaction. Similarly, a user might click the Stop button on the browser before receiving a confirmation page, and subsequently resubmit the same form. In most cases, we want to trap and disallow these duplicate submissions, and using a controlling servlet provides a control point for addressing this problem.

Synchronizer (or Déjà vu) Token

This strategy addresses the problem of duplicate form submissions. A synchronizer token is set in a user's session and included with each form returned to the client. When that form is submitted, the synchronizer token in the form is compared to the synchronizer token in the session. The tokens should match the first time the form is submitted. If the tokens do not match, then the form submission may be disallowed and an error returned to the user. Token mismatch may occur when the user submits a form, then clicks the Back button in the browser and attempts to resubmit the same form.

On the other hand, if the two token values match, then we are confident that the flow of control is exactly as expected. At this point, the token value in the session is modified to a new value and the form submission is accepted.

You may also use this strategy to control direct browser access to certain pages, as described in the sections on resource guards. For example, assume a user bookmarks page A of an application, where page A should only be accessed from page B and C. When the user selects page A via the bookmark, the page is accessed out of order and the synchronizer token will be in an unsynchronized state, or it may not exist at all. Either way, the access can be disallowed if desired.

Please refer to "Introduce Synchronizer Token in the "Presentation Tier Refactorings section for an example of this strategy.

Validation

It is often desirable to perform validation both on the client and on the server. Although client validation processing is typically less sophisticated than server validation, it provides high-level checks, such as whether a form field is empty. Server-side validation is often much more comprehensive. While both types of processing are appropriate in an application, it is not recommended to include only client-side validation. One major reason not to rely solely on client-side validation is that client-side scripting languages are user-configurable and thus may be disabled at any time.

Detailed discussion of validation strategies is outside the scope of this book. At the same time, we want to mention these issues as ones to consider while designing your systems, and hope you will refer to the existing literature in order to investigate further.

Validation on Client

Input validation is performed on the client. Typically, this involves embedding scripting code, such as JavaScript, within the client view. As stated, client-side validation is a fine complement for server-side validation, but should not be used alone.

Validation on Server

Input validation is performed on the server. There are several typical strategies for doing server validation. These strategies are form-centric validation and validation based on abstract types.

Form-Centric Validation

The form-centric validation strategy forces an application to include lots of methods that validate various pieces of state for each form submitted. Typically, these methods overlap with respect to the logic they include, such that reuse and modularity suffer. Since there is a validation method that is specific to each Web form that is posted, there is no central code to handle required fields or numeric-only fields. In this case, although there may be a field on multiple different forms that is considered a required field, each is handled separately and redundantly in numerous places in the application. This strategy is relatively easy to implement and is effective, but it leads to duplication of code as an application grows.

To provide a more flexible, reusable, and maintainable solution, the model data may be considered at a different level of abstraction. This approach is considered in the following alternative strategy, “Validation Based on Abstract Types. An example of form-centric validation is shown in the listing in Example 3.4.

Example 3.4 Form-Centric Validation

```
/**If the first name or last name fields were left
  blank, then an error will be returned to client.
  With this strategy, these checks for the existence
  of a required field are duplicated. If this valida-
  tion logic were abstracted into a separate compo-
  nent, it could be reused across forms (see
  Validation Based on Abstract Types strategy)**/
public Vector validate()
{
Vector errorCollection = new Vector();
    if ((firstname == null) ||
   (firstname.trim.length() < 1))
     errorCollection.addElement("firstname required");
    if ((lastname == null) || (lastname.trim.length()
  < 1))
     errorCollection.addElement("lastname required");
return errorCollection;
}
```

Validation Based on Abstract Types

This strategy could be utilized on either the client or server, but is preferred on the server in a browser-based or thin-client environment.

The typing and constraints information is abstracted out of the model state and into a generic framework. This separates the validation of the model from the application logic in which the model is being used, thus reducing their coupling.

Model validation is performed by comparing the metadata and constraints to the model state. The metadata and constraints about the model are typically accessible from some sort of simple data store, such as a properties file. A benefit of this approach is that the system becomes more generic, because it factors the state typing and constraint information out of the application logic.

An example is to have a component or subsystem that encapsulates validation logic, such as deciding whether a string is empty, whether a certain number is within a valid range, whether a string is formatted in a particular way, and so on. When various disparate application components want to validate different aspects of a model, each component does not write its own validation code. Rather, the centralized validation mechanism is used. The centralized validation mechanism will typically be configured either programmatically, through some sort of factory, or declaratively, using configuration files.

Thus, the validation mechanism is more generic, focusing on the model state and its requirements, independent of the other parts of the application. A drawback to using this strategy is the potential reduction in efficiency and performance. Also, more generic solutions, although often powerful, are sometimes less easily understood and maintained.

An example scenario follows. An XML-based configuration file describes a variety of validations, such as “required field,” “all-numeric field,” and so on. Additionally, handler classes can be designated for each of these validations. Finally, a mapping links HTML form values to a specific type of validation. The code for validating a particular form field simply becomes something similar to the code snippet shown in Example 3.5.

Example 3.5 Validation Based on Abstract Types

```
//firstNameString="Dan"
//formFieldName="form1.firstname"
Validator.getInstance().validate(firstNameString,
  formFieldName);
```

Helper Properties—Integrity and Consistency

JavaBean helper classes are typically used to hold intermediate state when it is passed in with a client request. JSP runtime engines provide a mechanism for automatically copying parameter values from a servlet request object into properties of these JavaBean helpers. The JSP syntax is as follows:

```
<jsp:setProperty name="helper" property="*"/>
```

This tells the JSP engine to copy all *matching* parameter values into the corresponding properties in a JavaBean called “helper,” shown in Example 3.6:

Example 3.6 Helper Properties - A Simple JavaBean Helper

```
public class Helper
{
  private String first;
  private String last;

  public String getFirst()
  {
    return first;
  }

  public void setFirst(String aString)
  {
    first=aString;
  }

  public String getLast()
  {
    return last;
  }


  public void setLast(String aString)
  {
    last=aString;
  }

}
```

How is a match determined, though? If a request parameter exists with the same name and same type as the helper bean property, then it is considered a match. Practically, then, each parameter is compared to each bean property name and the type of the bean property setter method.

Although this mechanism is simple, it can produce some confusing and unwanted side effects. First of all, it is important to note what happens when a request parameter has an empty value. Many developers assume that a request parameter with an empty string value should, if matched to a bean property, cause that bean property to take on the value of an empty string, or null. The spec-compliant behavior is actually to make no changes to the matching bean property in this case, though. Furthermore, since JavaBean helper instances are typically reused across requests, such confusion can lead to data values being inconsistent and incorrect. Figure 3.1 shows the sort of problem that this might cause.

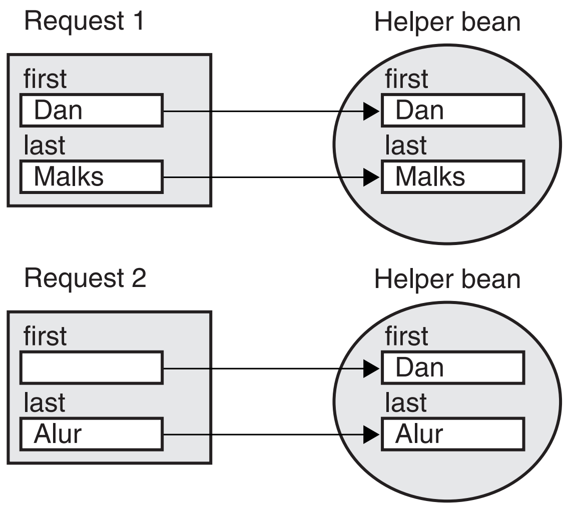

Figure 3.1 Helper properties.

Request 1 includes values for the parameter named “first” and the one named “last,” and each of the corresponding bean properties is set. Request 2 includes a value only for the “last” parameter, causing only that one property to be set in the bean. The value for the “first” parameter is unchanged. It is not reset to an empty string, or null, simply because there is no value in the request parameter. As you can see in Figure 3.1, this may lead to inconsistencies if the bean values are not reset manually between requests.

Another related issue to consider when designing your application is the behavior of HTML form interfaces when controls of the form are not selected. For example, if a form has multiple checkboxes, it is not unreasonable to expect that *un*checking every checkbox would result in clearing out these values on the server. In the case of the request object created based on this interface, however, there would simply not be a parameter included in this request object for any of the checkbox values. Thus, no parameter values relating to these checkboxes are sent to the server (see *http://www.w3.org* for full HTML specification).

Since there is no parameter passed to the server, the matching bean property will remain unchanged when using the `<jsp:setProperty>` action, as described. So, in this case, unless the developer manually modifies these values, there is the potential for inconsistent and incorrect data values to exist in the application. As stated, a simple design solution to this problem is to reset all state in the JavaBean between requests.

Presentation Tier Bad Practices

Bad practices are less than optimal solutions that conflict with many of the patterns' recommendations. When we documented the patterns and best practices, we naturally discarded those practices that were less than optimal.

In this part of the book, we highlight what we consider to be bad practices in the presentation tier.

In each section, we briefly describe the bad practice and provide numerous references to design issues, refactorings, and patterns that provide further information and preferable alternatives. We do not provide an in-depth discussion of each bad practice, but rather present a brief synopsis as a starting point for further investigation.

The "Problem Summary" section provides a quick description of a less than optimal situation, while the "Solution Reference" section includes references to:

- *Patterns* that provide information on context and trade-offs;
- *Design considerations* that provide related details;
- *Refactorings* that describe the journey from the less than optimal situation (bad practice) to a more optimal one, a best practice, or pattern.

Consider this part of the book as a roadmap, using the references to locate further detail and description in other parts of the book.

Control Code in Multiple Views

Problem Summary

Custom tag helpers may be included at the top of a JSP View to perform access control and other types of checks. If a large number of views include similar helper references, maintaining this code becomes difficult, since changes must be made in multiple places.

Solution Reference

Consolidate control code, introducing a controller and associated Command helpers.

Refactoring
- See "Introduce a Controller" on page 74.

Refactoring
- See "Localize Disparate Logic" on page 83.

Pattern
- See "Front Controller – "Command and Controller Strategy" on page 179.

When there is a need to include similar control code in multiple places, such as when only a portion of a JSP View is to be restricted from a particular user, delegate the work to a reusable helper class.

Pattern
- See "View Helper" on page 186

Design
- See "Guarding a View" on page 38.

Exposing Presentation-Tier Data Structures to Business Tier

Problem Summary

Presentation-tier data structures, such as HttpServletRequest, should be confined to the presentation tier. Sharing these details with the business tier, or any other tier, increases coupling between these tiers, dramatically reducing the reusability of the available services. If the method signature in the business service accepts a parameter of type HttpServletRequest, then any other clients to this service (even those outside of the Web space) must wrap their request state in an HttpServletRequest object. Additionally, in this case the business-tier services need to understand how to interact with these presentation tier-specific data structures, increasing the complexity of the business-tier code and increasing the coupling between the tiers.

Solution Reference

Instead of sharing data structures specific to the presentation tier with the business tier, copy the relevant state into more generic data structures and share those. Alternatively, extract and share the relevant state from the presentation tier-specific data structure as individual parameters.

Refactoring
- See "Hide Presentation Tier-Specific Details From the Business Tier" on page 91.

ch03.fm Page 51 Friday, June 1, 2001 1:34 PM

Exposing Presentation-Tier Data Structures to Domain Objects

Problem Summary

Sharing request handling data structures, such as HttpServletRequest, with domain objects needlessly increases the coupling between these two distinct aspects of the application. Domain objects should be reusable components, and if their implementation relies on protocol or tier-specific details, their potential for reuse is reduced. Furthermore, maintaining and debugging tightly coupled applications is more difficult.

Solution Reference

Instead of passing an HttpServletRequest object as a parameter, copy the state from the request object into a more generic data structure and share this object with the domain object. Alternatively, extract the relevant state from the HttpServletRequest object and provide each piece of state as an individual parameter to the domain object.

Refactoring
- See “Hide Presentation Tier-Specific Details From the Business Tier” on page 91.

Allowing Duplicate Form Submissions

Problem Summary

One of the limitations of the browser-client environment is the lack of control an application has over client navigation. A user might submit an order form that results in a transaction that debits a credit card account and initiates shipment of a product to a residence. If after receiving the confirmation page, the user clicks the Back button, then the same form could be resubmitted.

Solution Reference

To address these issues, monitor and control the request flow.

Refactoring
- See “Introduce Synchronizer Token” on page 77.

Refactoring
- See “Controlling Client Access” on page 38.

Design
- See “Synchronizer (or Déjà vu) Token” on page 43.

Exposing Sensitive Resources to Direct Client Access

Problem Summary

Security is one of the most important issues in enterprise environments. If there is no need for a client to have direct access to certain information, then this information must be protected. If specific configuration files, property files, JSPs, and class files are not secured appropriately, then clients may inadvertently or maliciously retrieve sensitive information.

Solution Reference

Protect sensitive resources, disallowing direct client access

- **Refactoring** See "Hide Resource From a Client" on page 100.
- **Refactoring** See "Controlling Client Access" on page 38.

Assuming <jsp:setProperty> Will Reset Bean Properties

Problem Summary

While the expected behavior of the `<jsp:setProperty>` standard tag is to copy request parameter values into JavaBean helper properties of the same name, its behavior when dealing with parameters that have empty values is often confusing. For example, a parameter with an empty value is ignored, although many developers incorrectly assume that the matching JavaBean property will be assigned a null or empty string value.

Solution Reference

Take into account the less than intuitive nature of how properties are set when using the `<jsp:setProperty>` tag, and initialize bean properties before use.

- **Design** See "Helper Properties—Integrity and Consistency" on page 46.

Creating Fat Controllers

Problem Summary

Control code that is duplicated in multiple JSP views should, in many cases, be refactored into a controller. If too much code is added to a controller, though, it becomes too heavyweight and cumbersome to maintain, test, and debug. For example, unit testing a servlet controller, particularly a “fat controller,” is more complicated than unit testing individual helper classes that are independent of the HTTP protocol.

Solution Reference

A controller is typically the initial contact point for handling a request, but it should also be a delegation point, working in coordination with other control classes. Command objects are used to encapsulate control code to which the controller delegates. It is much easier to unit test these JavaBean command objects, independent of the servlet engine, than it is to test less modular code.

Refactoring
- See “Introduce a Controller” on page 74.

Pattern
- See “Front Controller–“Command and Controller Strategy” on page 179.

Refactoring
- See “Localize Disparate Logic” on page 83.

Pattern
- See “View Helper” on page 186.