Design with the JSF architecture
Exploring design patterns used in the JavaServer Faces framework

Anand Prakash Joshi

In this article, author Anand Joshi explains the JavaServer™ Faces (JSF) architecture using design patterns in the JSF framework. He discusses Gang of Four design patterns employed in the JSF architecture, and how they work within the JSF framework. Anyone with a general knowledge of design patterns and JSF architecture will learn from Anand's detailed guide.

*Readers should have a good knowledge of Gang of Four design patterns and JSF technology.

Design patterns help users to abstract details at a higher level and better understand architecture. If you are familiar with Gang of Four design patterns and the JavaServer Faces (JSF) framework in general, this article will help you gain insight about the design patterns used in JSF framework, and how they work in-depth.

This article explores design patterns used in the JSF framework. Design patterns discussed in detail are Singleton, Model-View-Controller, Factory Method, State, Composite, Decorator, Strategy, Template Method, and Observer.

Design patterns and JavaServer Faces (JSF) technology

First, a brief introduction about patterns and the JSF framework.

- **Patterns.** The use patterns is a ubiquitous way to abstract a problem and its solutions. Because patterns are recognized by all developers and architects, patterns can save time and energy. In layman's terms, a pattern is a proven solution to a well-known problem. You can reuse patterns, and this reuse helps solutions become robust.

- **Java Server Faces.** JSF architecture is a framework for Web applications. It is driven by the Java Community Process (JCP), and is expected to become a standard framework for Web application developers. At present, the more than 50 frameworks for developing Web applications indicate a strong need to standardize the framework -- and JSF architecture is doing that job!

Deep dive into JSF patterns

Now let's explore a variety of design patterns in the JSF architecture. The patterns I discuss in detail include Singleton, Model-View-Controller, Factory Method, State, Composite, Decorator,
Strategy, Template Method, and Observer. I'll review the intent of each pattern, and how it functions within the JSF framework.

**Singleton pattern**

The intent of the Singleton pattern is to ensure that only one instance of a class is loaded, and that the instance provides a global point of access. While starting a Web application that has JSF support, the Web container initializes a FacesServlet instance. In this stage, FacesServlet instantiates the Application and LifeCycle instances once per Web application. These instances represent the well-known Singleton pattern, which typically requires only one instance of the type.

A Web application using JSF requires only one instance of the Application and LifeCycle classes. LifeCycle manages the entire life cycle of multiple JSF requests. It makes sense to have these objects as Singleton patterns because their states and behaviors are shared across all requests. LifeCycle maintains PhaseListeners, which are also Singleton patterns. PhaseListeners are shared by all JSF requests. You can use a Singleton pattern extensively in JSF framework to reduce the memory footprint and provide global access to objects. NavigationHandler (used to determine logical outcome of the request) and ViewHandler (used to create view) also represent Singleton patterns.

**Model-View-Controller (MVC)**

The MVC pattern's purpose is to decouple Model (or data) from the presentation of the data (View). If your application has more than one presentation, you can can replace only the view layer and reuse code for the controller and model. Similarly, if you need to change a model, the view layer remains largely unaffected. Controller handles user actions that might result in changes in the model and updates to the views. When a user requests a JSF page, the request goes to FacesServlet. FacesServlet is the front controller servlet used in JSF. Like many other Web application frameworks, JSF uses the MVC pattern to decouple the view and the model. To handle user requests centrally, the controller servlet makes changes to the model and navigates users to views.

FacesServlet is the controller element in the JSF framework which all user requests go through. FacesServlet examines user requests and calls various actions on the model using managed beans. Backing, or managed, beans are an example of the model. JSF user interface (UI) components represent the view layer. The MVC pattern helps to divide tasks among developers who have different skill sets so tasks can be carried out in parallel; that is, GUI designers can create JSF pages with rich UI components while back-end developers can create managed beans to write business-logic specific code.

**Factory Method pattern**

The purpose of the Factory Method pattern is to define an interface for creating an object but deferring instantiation of the object to subclasses. In the JSF architecture, the Factory Method pattern is used to create objects. LifeCycleFactory is a factory object that creates and returns LifeCycle instances. The getLifeCycle (String LifeCycleId) method of LifeCycleFactory is a Factory Method pattern that creates (if needed) and returns LifeCycle instances based on
LifeCycleId. Custom JSF implementation can redefine an abstract getLifeCycle method to create a custom LifeCycle instance. Default JSF implementation gives the default LifeCycle instance. Also, for every JSF request, FacesServlet gets FacesContext from FacesContextFactory. FacesContextFactory is an abstract class that exposes the getFacesContext API, and JSF implementation provides concrete implementation of the FacesContextFactory and getFacesContext API. This is another example of the Factory Method pattern where concrete FacesContextFactory creates FacesContext objects.

**State pattern**

The intent of the State pattern is to distribute state-specific logic across different classes that represent states. FacesServlet invokes execute and render methods on a LifeCycle instance. LifeCycle collaborates with different Phases in order to execute a JSF request. JSF implementation follows the State pattern here. If this pattern wasn't used, LifeCycle implementation would be cluttered with a lot of conditionals (or "if" statements). JSF implementation creates separate classes of each state (or phase) and invokes steps. A phase is an abstract class that defines the common interface for every step. In the JSF framework, six phases, or steps, are defined: RestoreViewPhase, ApplyRequestValues, ProcessValidationsPhase, UpdateModelValuesPhase, InvokeApplicationPhase, and RenderResponsePhase.

As in the State pattern, LifeCycle passes a FacesContext object to the phases. Each phase or state changes the contextual information passed to it and then sets the flag in the FacesContext itself to indicate the next possible step. JSF implementation alters its behavior with each step. Each phase can be responsible for the next phase. FacesContext has two flags -- renderResponse and responseComplete -- to change the sequence of execution. After execution of each step, LifeCycle checks whether either of these flags was set by the previous phase. If a responseComplete is set, LifeCycle abandons the execution of the request altogether. If a renderResponse flag is set after some phase, JSF implementation skips remaining phases and goes directly to the render response phase. If neither flag is set, LifeCycle executes the next step in the sequence.

**Composite pattern**

The Composite pattern allows clients to deal with composite and primitive objects uniformly. A composite object is a container for primitive objects. In the first phase (Restore View phase) and the last phase (Render Response phase), the UI View is constructed using JSF UI components. UIComponentBase represents an abstract Component class in the Composite pattern. UIViewRoot is the Composite class, and UIOutput, for example, is the leaf (or primitive). The UIComponentBase class defines common methods for leaf and composite objects, such as encoding and decoding values and child management functions. Child management functions, such as getChildren, return an empty list for leaf nodes and children for composite nodes.

**Decorator pattern**

The intent of the Decorator pattern is to extend the behavior of an object dynamically without subclassing. The JSF framework has many extension points, or pluggability mechanisms.
JSF implementation can replace default PropertyResolver, VariableResolver, ActionListener, NavigationHandler, ViewHandler, orStateManager by using the Decorator pattern. Typically custom implementation receives the reference to the default implementation passed through its constructor. Custom implementation overrides only a subset of the functionality and delegates the rest of the functions to the default implementation. If you want to implement a custom ViewHandler that overrides the calculateLocale method from the default ViewHandler implementation, you can write CustomViewHandler class as in Listing 1:

Listing 1. CustomViewHandler snippet

```java
public class CustomViewHandler extends ViewHandler {
    public CustomViewHandler(ViewHandler handler) {
        super();
        oldViewHandler = handler;
    }
    private ViewHandler oldViewHandler = null;
    public void renderView (facesContext context, UIViewRoot view) {
        //delegate method to oldViewHandler
        oldViewHandler.renderView(context, view);
    }
    //custom implementation of calculateLocale
    public Locale calculateLocale(FacesContext context) {
    }
}
```

Strategy pattern

The purpose of the Strategy pattern is to encapsulate a concept that varies. The JSF framework employs the Strategy pattern for rendering UI components using the delegated implementation model. JSF technology supports two kinds of rendering models. In a direct implementation model, UI components decode the values from the incoming request themselves and encode the values to be displayed. In a delegated implementation model, decoding and encoding operations are delegated to the separate renderer associated with the component. The latter model is more flexible than direct implementation and leverages the Strategy design pattern. In the Strategy pattern, you encapsulate an algorithm that varies into a separate object so the algorithm can be varied dynamically. JSF implementation might register additional renderers with the existing renderkit instance, and, at application startup time, JSF implementation reads the configuration file and associates those renderers to the UI components.

Template Method pattern

The Template Method pattern's intention is to defer variant steps to the subclasses while the parent class defines invariant steps of the algorithm. The JSF framework offers functionality provided by the Template Method pattern through PhaseListeners. Template Method (or "hook") is implemented so Web authors can provide implementation for the optional steps between phases while main phases remain the same as defined by the JSF framework. The JSF framework provides PhaseListeners that are conceptually similar to variant steps of the Template Method pattern. The JSF framework has six predefined phases, and between each phase, a Web author can implement PhaseListeners to provide hooks similar to the Template Method hooks. In fact, this is much more extensible than the Template Method pattern. You can provide hooks after each
phase by registering a PhaseListener that has Phaseld of ANY_PHASE value. If the Phaseld is
ANY_PHASE, the JSF implementation calls the PhaseListener before and after every phase. The
implementation is slightly different in the JSF framework because one can have no PhaseListeners
at all, but in the Template Method pattern, subclasses generally redefine variant steps that are left
abstract in the parent class.

Observer pattern

The intent of the Observer pattern is to notify all dependent objects (or observers) automatically
when the state in the subject changes. The JSF framework implements the Observer
pattern in UI components. JSF technology has two kinds of built-in events: ActionEvent and
ValueChangedEvent. ActionEvent is useful in determining the activation of user interface
components such as buttons. When a user clicks on the button, the JSF implementation notifies
one or more action listeners added to the button. The button is activated or the state of the button
(subject) changes. All listeners (or observers) are notified of the changes of state in subject that
are added to the button. Similarly, when the value in the input UI component changes, the JSF
implementation notifies ValueChangeListeners.

In conclusion

The JSF framework leverages Singleton, Model-View-Controller, Factory Method, State,
Composite, Decorator, Strategy, Template Method, and Observer design patterns. It’s a robust
framework in that its architecture is based on already proven design patterns, which are utilized
very nicely in the JSF framework.
Related topics

- Visit the developerWorks Web Architecture zone and Java technology zone. They specialize in articles and tutorials covering various Web-based and Java-based solutions, respectively.
- Get free JSF downloads from the Sun Developer Network.
- Download a free trial version of IBM Rational Software Architect.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)