Java BluePrints

Model-View-Controller

Context

Application presents content to users in numerous pages containing various data. Also, the engineering team responsible for designing, implementing, and maintaining the application is composed of individuals with different skill sets.

Problem

Now, more than ever, enterprise applications need to support multiple types of users with multiple types of interfaces. For example, an online store may require an HTML front for Web customers, a WML front for wireless customers, a Java™ (JFC) / Swing interface for administrators, and an XML-based Web service for suppliers.

When developing an application to support a single type of client, it is sometimes beneficial to interweave data access and business rules logic with interface-specific logic for presentation and control. Such an approach, however, is inadequate when applied to enterprise systems that need to support multiple types of clients. Different applications need to be developed, one to support each type of client interface. Non-interface-specific code is duplicated in each application, resulting in duplicate efforts in implementation (often of the copy-and-paste variety), as well as testing and maintenance. The task of determining what to duplicate is expensive in itself, since interface-specific and non-interface-specific code are intertwined. The duplication efforts are inevitably imperfect. Slowly, but surely, applications that are supposed to provide the same core functionality evolve into different systems.
Forces

- The same enterprise data needs to be accessed when presented in different views: e.g. HTML, WML, JFC/Swing, XML
- The same enterprise data needs to be updated through different interactions: e.g. link selections on an HTML page or WML card, button clicks on a JFC/Swing GUI, SOAP messages written in XML
- Supporting multiple types of views and interactions should not impact the components that provide the core functionality of the enterprise application

Solution

By applying the Model-View-Controller (MVC) architecture to a Java™ 2 Platform, Enterprise Edition (J2EE™) application, you separate core business model functionality from the presentation and control logic that uses this functionality. Such separation allows multiple views to share the same enterprise data model, which makes supporting multiple clients easier to implement, test, and maintain.

Structure

The following diagram represents the Model-View-Controller pattern:

Participants & Responsibilities

The MVC architecture has its roots in Smalltalk, where it was originally applied to map the traditional input, processing, and output tasks to the graphical user interaction model. However, it is straightforward to map these concepts into the domain of multi-tier enterprise applications.
• **Model** - The model represents enterprise data and the business rules that govern access to and updates of this data. Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.

• **View** - The view renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented. It is the view's responsibility to maintain consistency in its presentation when the model changes. This can be achieved by using a push model, where the view registers itself with the model for change notifications, or a pull model, where the view is responsible for calling the model when it needs to retrieve the most current data.

• **Controller** - The controller translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET and POST HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

**Strategies**

• **Web-based clients such as browsers.** JavaServer Pages™ (JSP™) pages to render the view, Servlet as the controller, and Enterprise JavaBeans™ (EJB™) components as the model. The Java Pet Store sample application illustrates this strategy.

• **Centralized controller.** Instead of having multiple servlets as controllers, a main Servlet is used to make control more manageable. The Front Controller pattern describes this strategy in more detail.

• **Wireless clients such as cell phones.** The Smart Ticket sample application illustrates this strategy.

**Consequences**

• **Re-use of Model components.** The separation of model and view allows multiple views to use the same enterprise model. Consequently, an enterprise application's model components are easier to implement, test, and maintain, since all access to the model goes through these components.

• **Easier support for new types of clients.** To support a new type of client, you simply write a view and some controller logic and wire them into the existing enterprise application.

• **Increased design complexity.** This pattern introduces some extra classes due to the separation of model, view, and controller.

**Related Patterns**

[BRMSS96]: Presentation-Abstraction-Control defines a structure with a hierarchy or cooperating agents, each with specific duties of the application functionality. It divides the functionality responsibilities into three components: presentation, abstraction, and control.

© 2000-2002 Sun Microsystems, Inc. All Rights Reserved.