MVC

Model-view-controller (**MVC**) is an architectural pattern used in software engineering. Successful use of the pattern isolates business logic from user interface considerations, resulting in an application where it is easier to modify either the visual appearance of the application or the underlying business rules without affecting the other. In MVC, the *model* represents the information (the data) of the application; the *view* corresponds to elements of the user interface such as text, checkbox items, and so forth; and the *controller* manages the communication of data and the business rules used to manipulate the data to and from the model.

History

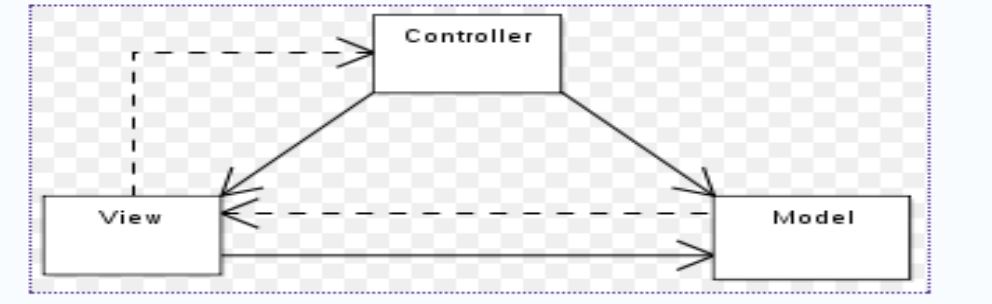
MVC was first described in 1979 by Trygve Reenskaug, then working on Smalltalk at Xerox PARC. The original implementation is described in depth in the influential paper *Applications Programming in Smalltalk-80: How to use Model-View-Controller.*

There have been several derivatives of MVC; one of the most known (due to its use by Microsoft) is the Model View Presenter pattern which appeared in the early 1990s and was designed to be an evolution of MVC. However Model-View-Controller still remains very widely used.

In November 2002 the W3C voted to make MVC structures part of their XForms architecture for all future web applications . These specifications will now be integrated directly into the XHTML 2.0 specifications. There are now over 20 vendors that support XForms frameworks with MVC integrated into the application stack.

Pattern description

Model-view-controller is both an architectural pattern and a design pattern, depending on where it is used.



As an architectural pattern

It is common to split an application into separate layers that run on different computers: presentation (UI), domain logic, and data access. In MVC the presentation layer is further separated into view and controller.

MVC is often seen in web applications, where the view is the actual HTML page, and the controller is the code that gathers dynamic data and generates the content within the HTML. Finally, the model is represented by the actual content, usually stored in a database or in XML nodes, and the business rules that transform that content based on user actions.

Though MVC comes in different flavors, control flow generally works as follows:

1. The user interacts with the user interface in some way (e.g. presses a button).
2. A controller handles the input event from the user interface, often via a registered handler or callback.
3. The controller notifies the model of the user action, possibly resulting in a change in the model's state. (e.g. controller updates user's Shopping cart).[4]
4. A view uses the model (indirectly) to generate an appropriate user interface (e.g. the view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view.
5. The user interface waits for further user interactions, which begins then a new cycle.

Some implementations such as the W3C XForms also use the concept of a dependency graph to automate the updating of views when data in the model changes.

By decoupling models and views, MVC helps to reduce the complexity in architectural design, and to increase flexibility and reuse.

As a design pattern

MVC encompasses more of the architecture of an application than is typical for a design pattern.

Model

The **domain**-specific representation of the information on which the application operates. Domain logic adds meaning to raw data (e.g., calculating whether today is the user's birthday, or the totals, taxes, and shipping charges for shopping cart items).

Many applications use a persistent storage mechanism (such as a database) to store data. MVC does not specifically mention the data access layer because it is understood to be underneath or encapsulated by the Model.

View

Renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes.

Controller

Processes and responds to events, typically user actions, and may invoke changes on the model.

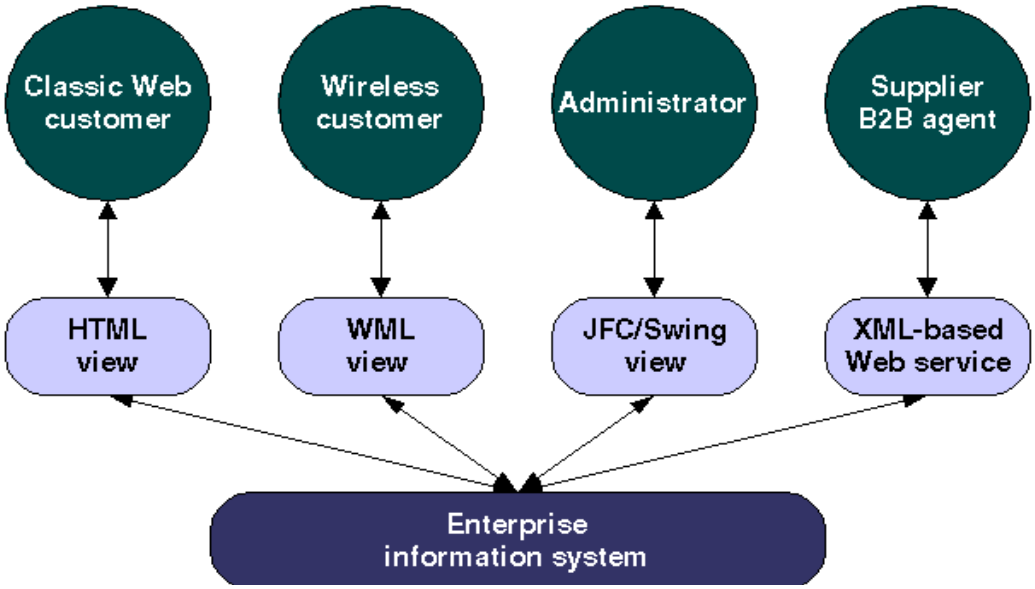
Java BluePrints
Model-View-Controller

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

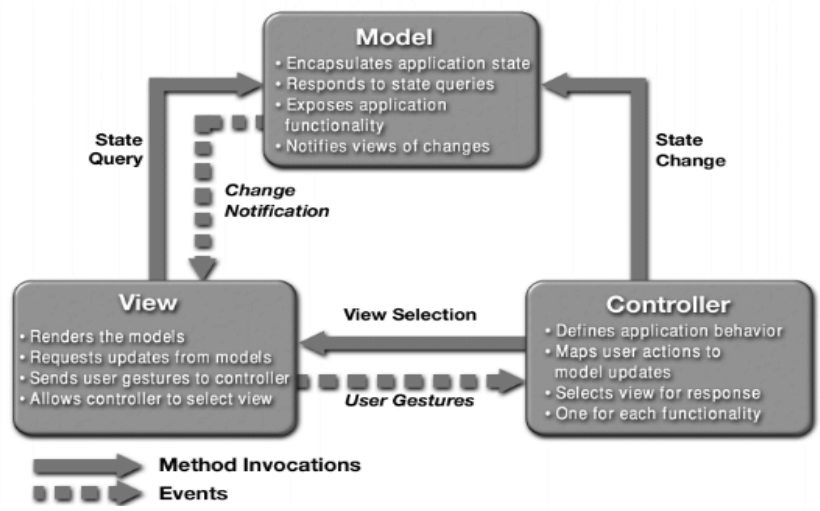
- The same enterprise data needs to be accessed when presented in different views: *e.g.* HTML, WML, JFC/Swing, XML
- The same enterprise data needs to be updated through different interactions: *e.g.* link selections on an HTML page or WML card, button clicks on a JFC/Swing GUI, SOAP messages written 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.

Bibliography

http://en.wikipedia.org/wiki/Model-view-controller

http://java.sun.com/blueprints/patterns/MVC-detailed.html