Model-View-Controller
Patterns and Frameworks

MVC Context

- The purpose of many computer systems is to retrieve data from a data store and display it for the user.
  - The user may then modify the data in keeping with various business logic rules, and the system stores the updates in the data store.
- Because the key flow of information is between the data store and the user interface, you might be inclined to tie these two pieces together. However, there are problems:
  - The user interface tends to change much more frequently than the data storage system. Furthermore, multiple user interfaces might be required (HTML, WML for wireless, etc.).
  - Adding new data views often requires reimplementing or cutting and pasting business logic code, which then requires maintenance in multiple places.
  - Data access code suffers from the same problem, being cut and pasted among business logic methods.

Problem

- How do you decouple data access, business logic, and user interface functionality of an application so that you can easily modify the individual parts?

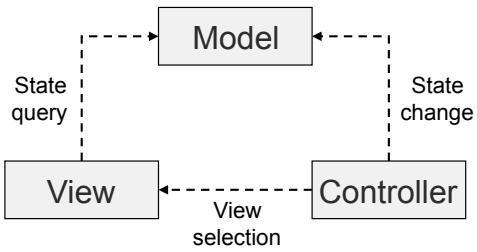
Forces

- User interface logic tends to change more frequently than business logic, especially in Web-based applications.
  - If presentation code and business logic are combined in a single object, you have to modify an object containing business logic every time you change the user interface.
  - This is likely to introduce errors and require the retesting of all business logic after every minimal user interface change.
- In some cases, the application displays the same data in different ways.
  - An analyst may prefer a spreadsheet view of data whereas management prefers a pie chart of the same data.
  - In some rich-client user interfaces, multiple views of the same data are shown at the same time.
  - If the user changes data in one view, the system must update all other views of the data automatically.
- Designing visually appealing and efficient user interfaces generally requires a different skill set than does developing complex business logic.
  - Rarely does a person have both skill sets. Therefore, it is desirable to separate the development effort of these two parts.

Forces (2)

- User interface code tends to be more device-dependent than business logic.
  - If you want to migrate the application from a browser-based application to support personal digital assistants (PDAs) or Web-enabled cell phones, you must replace much of the user interface code, whereas the business logic may be unaffected.
  - A clean separation of these two parts accelerates the migration and minimizes the risk of introducing errors into the business logic.
- Creating automated tests for user interfaces is generally more difficult and time-consuming than for business logic.
  - Reducing the amount of code that is directly tied to the user interface enhances the testability of the application.

Solution



- Separate the data access, application logic, and user interface into three distinct components.

- Model
  - The model manages the data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).
- View
  - The view manages the display of information.
- Controller
  - The controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.

Solution (2)

- Although the view and the controller depend on the model, the model is independent of the view and controller.
  - This separation allows the model to be built and tested independently of the visual presentation.
- The separation between view and controller is less important.
  - Controllers may be tightly coupled to a view. In fact, many user interface frameworks implement the roles as one object.
  - In Web applications, on the other hand, the separation between view (the browser) and controller (the server-side components handling the HTTP request) is very well defined.

MVC Variations

- Passive Model
  - The model is only modified by a single controller.
  - The controller modifies the model and then informs the view that the model has changed.
  - The model has no way to notify the controller or the view when its state is changed.
- Active Model
  - The active model is used when the model changes state without the controller's involvement.
  - This can happen when other components are changing the data and the changes must be reflected in the views.
  - Consider a stock-ticker display. You receive stock data from an external source and want to update the views (for example, a ticker band and an alert window) when the stock data changes. Because only the model detects changes to its internal state when they occur, the model must notify the views to refresh the display.

MVC with Observer Pattern

- One of the major reasons for using the MVC pattern is to make the model independent from of the views.
  - If the model has to notify the views of changes, you reintroduce the dependency you were looking to avoid.
- The *Observer* pattern provides a mechanism to alert other objects of state changes without introducing dependencies on them.
  - The individual views (and sometimes also controllers) implement the Observer interface and register with the model.
  - When the model changes, it notifies the registered observers, but the model *never requires specific information about any views*.
  - This approach is often called *publish/subscribe*.
  - If there are many views, it makes sense to define multiple subjects so that each view can subscribe only to types of changes that are relevant to the view.

MVC Benefits

- Accommodates change
  - User interface requirements tend to change more rapidly than business rules. Because the model does not depend on the views, modifying or adding new types of views to the system generally does not affect the model. As a result, the scope of change is confined to the view.
- Supports multiple views.
  - Because the view is separated from the model and there is no direct dependency from the model to the view, the user interface can display multiple views of the same data at the same time.
- Enhances testability
  - Testing components is difficult when they are highly interdependent. When an error occurs, it is hard to isolate the problem to a specific component. MVC separates the concerns of storing, displaying, and updating data into three components that can be tested individually.

MVC Liabilities

- Complexity
  - The MVC pattern introduces new levels of indirection and therefore increases the complexity of the solution slightly. It also increases the event-driven nature of the user-interface code, which can become more difficult to debug.
- Cost of frequent updates
  - Decoupling the model from the view does not mean that developers of the model can ignore the nature of the views.
  - For example, an active model that undergoes frequent changes could flood the views with update requests. If a view (such as a graphical display) is slow to render, it may fall behind update requests.
  - Therefore, it is important to keep the view in mind when coding the model. For example, the model could batch multiple updates into a single notification to the views.

PageController Pattern

- The PageController pattern describes a design strategy for the Controller component of an MVC architecture in web applications.
  - There is a separate controller (e.g. servlet) for each url in the web application.
  - The controller handles HTTP requests, updates the model (e.g. database) as appropriate, and forwards the request to the appropriate view (e.g. JSP).
  - It is important to keep view code out of the controller.
  - Common functionality is abstracted into a base class for the page controllers.

FrontController Pattern

- FrontController is another design strategy for the Controller component of an MVC architecture in web applications.
  - All requests are initially handled by a centralized controller.
  - The controller takes care of common system services such as authentication and authorization checks.
  - The controller then invokes the appropriate Command depending on the specific request.
  - The Commands perform request specific actions, and then invoke the appropriate view.