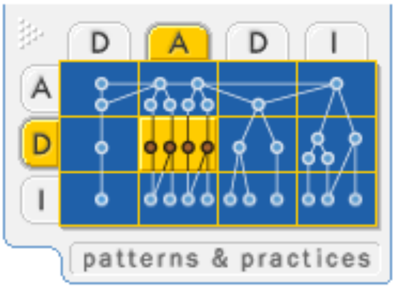
Model-View-Controller

Retired Content

Denne side er taget fra
https://msdn.microsoft.com/en-us/library/ff649643.aspx

This content is outdated and is no longer being maintained. It is provided as a courtesy for individuals who are still using these technologies. This page may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist. Please see the patterns & practices guidance for the most current information.



Bemærk - når de skriver at denne side er outdated så er det fordi
den var en del af beskrivelsen af en tidlig udgave af ASP.NET som
ikke længere bruges.
Faktisk vil jeg sige at dette er en af de bedste overordnede beskrivelser
af MVC der netop ikke snakker om specifik teknologi.

Version 1.0.1

GotDotNet community for collaboration on this pattern

Complete List of patterns & practices

Context

The purpose of many computer systems is to retrieve data from a data store and display it for the user. After the user changes the data, 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 to reduce the amount of coding and to improve application performance. However, this seemingly natural approach has some significant problems. One problem is that the user interface tends to change much more frequently than the data storage system. Another problem with coupling the data and user interface pieces is that business applications tend to incorporate business logic that goes far beyond data transmission.

Problem

How do you modularize the user interface functionality of a Web application so that you can easily modify the individual parts?

Forces

The following forces act on a system within this context and must be reconciled as you consider a solution to the problem:

- User interface logic tends to change more frequently than business logic, especially in Web-based applications. For example, new user interface pages may be added, or existing page layouts may be shuffled around. After all, one of the advantages of a Web-based thin-client application is the fact that you can change the user interface at any time without having to redistribute the application. 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. For example, when an analyst prefers 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 HTML pages 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.

- User interface activity generally consists of two parts: presentation and update. The presentation part retrieves data from a data source and formats the data for display. When the user performs an action based on the data, the update part passes control back to the business logic to update the data.

- In Web applications, a single page request combines the processing of the action associated with the link that the user selected with the rendering of the target page. In many cases, the target page may not be directly related to the action. For example, imagine a simple Web application that shows a list of items. The user returns to the main list page after either adding an item to the list or deleting an item from the list. Therefore, the application must render the same page (the list) after executing two quite different commands (adding or deleting)-all within the same HTTP request.

- 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. Therefore, reducing the amount of code that is directly tied to the user interface enhances the testability of the application.

Solution

The *Model-View-Controller (MVC)* pattern separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes [Burbeck92]:

- **Model**. The model manages the behavior and 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.

Figure 1 depicts the structural relationship between the three objects.

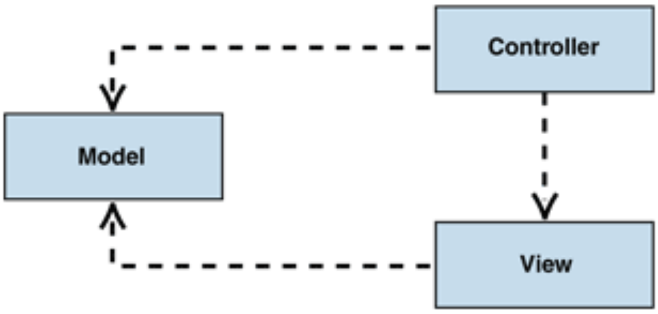


Figure 1: MVC class structure

It is important to note that both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation. The separation between view and controller is secondary in many rich-client applications, and, 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.

Model-View-Controller is a fundamental design pattern for the separation of user interface logic from business logic. Unfortunately, the popularity of the pattern has resulted in a number of faulty descriptions. In particular, the term "controller" has been used to mean different things in different contexts. Fortunately, the advent of Web applications has helped resolve some of the ambiguity because the separation between the view and the controller is so apparent.

Variations

In *Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)* [Burbeck92], Steve Burbeck describes two variations of *MVC*: a passive model and an active model.

The passive model is employed when one controller manipulates the model exclusively. The controller modifies the model and then informs the view that the model has changed and should be refreshed (see Figure 2). The model in this scenario is completely independent of the view and the controller, which means that there is no means for the model to report changes in its state. The HTTP protocol is an example of this. There is no simple way in the browser to get asynchronous updates from the server. The browser displays the view and responds to user input, but it does not detect changes in the data on the server. Only when the user explicitly requests a refresh is the server interrogated for changes.

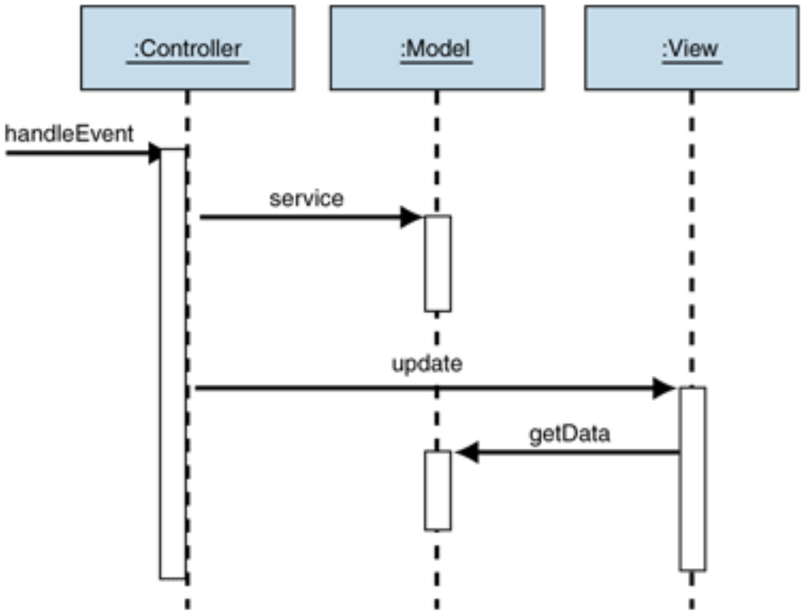


Figure 2: Behavior of the passive model

The active model is used when the model changes state without the controller's involvement. This can happen when other sources 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.

However, one of the motivations of using the *MVC* pattern is to make the model independent from of the views. If the model had to notify the views of changes, you would reintroduce the dependency you were looking to avoid. Fortunately, the *Observer* pattern [Gamma95] provides a mechanism to alert other objects of state changes without introducing dependencies on them. The individual views implement the *Observer*interface and register with the model. The model tracks the list of all observers that subscribe to changes. When a model changes, the model iterates through all registered observers and notifies them of the change. This approach is often called "publish-subscribe." The model never requires specific information about any views. In fact, in scenarios where the controller needs to be informed of model changes (for example, to enable or disable menu options), all the controller has to do is implement the *Observer* interface and subscribe to the model changes. In situations where there are many views, it makes sense to define multiple subjects, each of which describes a specific type of model change. Each view can then subscribe only to types of changes that are relevant to the view.

Figure 3 shows the structure of the active MVC using *Observer* and how the observer isolates the model from referencing views directly.

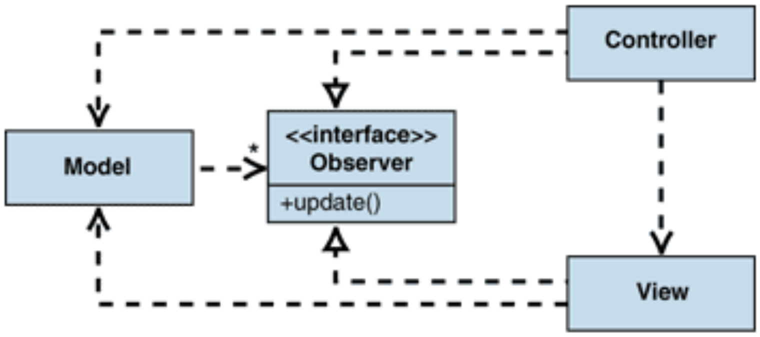


Figure 3: Using Observer to decouple the model from the view in the active model

Figure 4 illustrates how the *Observer* notifies the views when the model changes. Unfortunately, there is no good way to demonstrate the separation of model and view in a Unified Modeling Language (UML) sequence diagram, because the diagram represents instances of objects rather than classes and interfaces.

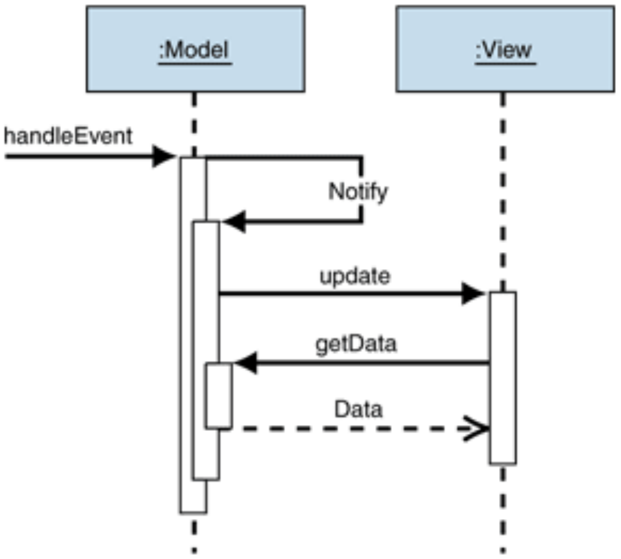


Figure 4: Behavior of the active model

Example

See *Implementing Model-View-Controller in ASP.NET.*

Testing Considerations

Testability is greatly enhanced when you employ employing *Model-View-Controller*. Testing components becomes difficult when they are highly interdependent, especially with user interface components. These types of components often require a complex setup just to test a simple function. Worse, when an error occurs, it is hard to isolate the problem to a specific component. This is the reason why separation of concerns is such an important architectural driver. *MVC* separates the concern of storing, displaying, and updating data into three components that can be tested individually.

Apart from the problems posed by interdependencies, user interface frameworks are inherently difficult to test. Testing user interfaces either requires tedious (and error-prone) manual testing or testing scripts that simulate user actions. These scripts tend to be time-consuming to develop and brittle. *MVC* does not eliminate the need for user interface testing, but separating the model from the presentation logic allows the model to be tested independent of the presentation and reduces the number of user interface test cases.

Resulting Context

Architecting the presentation layer around the *MVC* pattern results in the following benefits and liabilities:

Benefits

- **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. For example, multiple pages in a Web application may use the same model objects. Another example is a Web application that allows the user to change the appearance of the pages. These pages display the same data from the shared model, but show it in a different way.

- **Accommodates change**. User interface requirements tend to change more rapidly than business rules. Users may prefer different colors, fonts, screen layouts, and levels of support for new devices such as cell phones or PDAs. Because the model does not depend on the views, 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. This pattern lays the foundation for further specializations of this pattern such as *Page Controller*and *Front Controller*.

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, if the model undergoes frequent changes, it could flood the views with update requests. Some views, such as graphical displays, may take some time to render. As a result, the view 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 view.

Variants

The *Document-View* variant recognizes all three roles of *Model-View-Controller* but merges the controller into the view. The document corresponds to the model role in *MVC*. This variant is present in many existing GUI platforms. An excellent example of *Document-View* is the Microsoft Foundation Class Library (MFC) in the Microsoft Visual C++ environment. The tradeoff of using this variant is that the view and the controller are more tightly coupled.

Related Patterns

For more information, see the following related patterns:

- *Observer*. This pattern is often mentioned in conjunction with *MVC* due to the need to keep the views and the associated model synchronized.

- *Page Controller* and *Front Controller* describe implementation strategies for the controller portion of the *MVC* pattern.

Acknowledgments

Model-View-Controller began as a framework developed by Trygve Reenskaug for the Smalltalk platform in the late 1970s [Fowler03]. The version you have just read references the following works:

[Burbeck92] Burbeck, Steve. "Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)."*University of Illinois in Urbana-Champaign (UIUC) Smalltalk Archive.* Available at: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html.

[Fowler03] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[Gamma95] Gamma, Helm, Johnson, and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.