Patterns of Thought

Ralph Johnson - Department of Computer Science - 1304 W. Springfield Ave. - Urbana IL 61801 - johnson@cs.uiuc.edu

Patterns and Frameworks

I'm often asked "What is the difference between a pattern and a framework?" According to the dictionary definition of a pattern, there isn't much difference. The *World Book Dictionary* gives eleven noun definitions for pattern, including "a model or a guide for something to be made", which certainly matches a framework. But even a macro or a C++ template is a pattern according to that definition, and that is too broad to be useful.

The definition of a pattern that I use (which is Christopher Alexander's definition) is based on the realization that the reason that there are reoccuring solutions is that there are reoccuring problems. Thus, the problem that a pattern solves is just as much a part of the pattern as the particular arrangement of objects that makes up the solution. A pattern is therefore a *solution* to a *problem* in a *context*.. This influences the way we describe them, and patterns are usually described using a form that points out the solution, problem, and context of a pattern.

A framework like Model-View-Controller or InterViews is more than just a pattern, it is also code. It is a reusable design expressed as a set of abstract classes (i.e. code) and the way that instances of those classes interact. Since it includes the way that instances of those classes interact, it is the collaborative model or pattern of object interaction as much as it is the kinds of classes in the design. A single framework will contain many of the patterns in *Design Patterns: Elements of Reusable Object-Oriented Software.* Moreover, those patterns cannot be expressed as C++ or Smalltalk classes and then just reused by inheritance or composition. So, those patterns are both smaller and more abstract than frameworks.

The first part of this article will describe how Model-View-Controller is made up of simpler design patterns. It will reuse a couple of patterns I've talked about already and describe some new ones. The second part will revisit Model-View-Controller and try to find what someone might mean by "the MVC pattern".

Model-View-Controller

MVC is probably the earliest framework that was recognized to be a framework. It was part of Smalltalk-80, and was the inspiration for later user interface frameworks like MacApp and the Andrew Toolkit, which in turn inspired Interviews, ET++, OWL, MFC, and all the other modern user interface frameworks. It has evolved a lot since its earliest days, but still consists of three main classes; Model, View, and Controller. The Model is the application object, and represents the data that the program manipulates. The View manages a region of the display and is reponsible for making sure that the region contains a picture that is a function of the state of the model. When the model changes state, the view must redraw the part of the screen that it manages. The Controller translates user actions (like pressing a key or moving the mouse) into operations on the View or the Model. Thus, it is responsible for user input.

MVC is usually treated as a single mechanism, but it is useful to break it down into smaller patterns. These patterns are easier to understand in isolation, they are often reused in other contexts, and it is easier to precisely define their purpose. There are three main patterns; Observer (which describes the relationship between Models and Views), Composite (which

describes the relationship between Views and their subviews), and Strategy (which describes the relationship between Views and Controllers).

The Observer pattern

One of the most important steps in making reusable user interface components is to separate the application data from the user interface objects. A common way to do this is to make an object for each component of the user interface that we can see on the screen. In other words, buttons, input fields, scrollbars, and lists are all objects. But an application doesn't consist of buttons, input fields, and scrollbars. An invoice has a vender, a due date, a total amount, and a list of items purchased. The objects in the interface have to be tied to the application objects in some way, so that an invoice's vender is displayed as a label or in an input field of the user interface for an invoice. When the vender name or total amount changes in the application, the corresponding user interface objects must change their appearance.

How does a change in an application object cause a user interface object to change its apperance? There are lots of ways to accomplish this; user interface objects can register their interest in global "events", application objects can notify user interface objects directly, or they can notify them indirectly. For the Observer pattern, observers (like user interface objects) register with subjects (like application objects), and are then notified when the subjects change their state. Each subject keeps track of its observers, and each time it changes state, the subject will notify all its observers. Although there is a kind of coupling between subject and observer, it does not cause trouble in practice because it is quite abstract. A subject does not know the concrete class of its observer, though an observer often knows the class of its subject. The Subject class just supports a few operations to add an observer, to remove an observer, or to notify its observers. The Observer class only needs to support one operation; Update.

Here is a description of the Observer pattern, in "pattern form". Note that it does not mention user interfaces. The Observer pattern is much more general than user interfaces.

Observer

Context Two objects must be synchronized. Changes in one object must be reflected in the other.

Problem If changing one object requires changing another then there is a constraint between them. But we don't want every client to have to know about this constraint. Moreover, we aren't even sure how many objects need to be changed when the first object is changed. Different instances of the same class might have different numbers of dependents, or the dependents might change over time.

Constraints To make objects as reusable as possible, we do not want to hard-code constraints into them. Objects should be responsible only for their own state, not for the state of other objects. On the other hand, if there is a constraint between the states of two objects, then that constraint must be recorded somewhere. In some way, changes to the first object must be translated into changes to the second.

Therefore:

Solution Have an object involved in a constraint keep a list of dependents. Each time it changes, it notifies all its dependents. When an object is notified that something it depends on has changed, it takes appropriate action. In general, an object can have many dependents, and a change to the object will require only some of its dependents to

change. But each dependent will be able to determine whether the change to the object was significant and can ignore those that are not.

In practice, the Observer pattern leads to loosely linked objects that communicate indirectly. It is easy to write new observers or to add new observers to an existing design, because the observers just have to support the Update operation, and a new observer just has to register itself with the subject it is observing. However, it can be hard to design Subject classes correctly, because they must be written to notify their observers whenever there is a significant change. The biggest problem, however, is that the abstract coupling makes it hard to imagine the effects of a change to a design. For example, if you give a subject a new operation that makes several changes to its state, you might end up notifying its observers several times, which might be inefficient or result in annoying redraws of the screen. Although the loose coupling makes most changes easy, when a change leads to unpredicted behavior, it can be hard to figure out why. Although the Observer pattern causes problems as well as solving them, its advantages are strong enough that it is used in most user interface frameworks.

There are many variations on the Observer pattern. In the Smalltalk-80 dependence mechanism, class Object supports both the Observer and the Subject protocol. Thus, any object can be an observer or a subject, and sometimes an object is both. Class Model is a subclass of Object that provides a more efficient implementation of the Subject interface. Class View is the Observer. In C++, where it is unlikely that you will have a universal Object class, you will probably have a separate Subject and Observer class. Another design choice is what the arguments of the Notify operation will be. In Smalltalk-80, where the operation is called "update:", this argument is traditionally a symbol, which denotes the aspect of the state of the observer that changed. This does not have much expressive power, so often there is a new class hierarchy for describing update events.

MVC eliminates some of the potential pitfalls of the Observer pattern. In particular, you don't have to worry about registering or unregistering observers when you use MVC, because a view will automatically register itself as a dependent of its model.

The Composite pattern

I've said a lot about the Composite pattern in earlier articles, so I won't say much about it here. In short, the Composite pattern lets you group a set of objects together and treat them as a single object. The main use of Composite in MVC is to let you group a set of views together and treat them as a single view. In the original MVC, every view has a set of subviews, so any view could be used as a composite. In practice, however, most subclasses of View never had any subviews, and never would. Thus, the original design was a mistake, and it was changed so that instead of View having a set of subviews, there was a class CompositeView that had a set of subviews. Now, you can make a set of radio buttons by creating a CompositeView and giving it some Buttons as subviews. The Buttons all have the same model, but each has a different value that they will give to their model if they are pushed. When the state of the model changes, they compare the new value with their value to see whether they are pressed.

MVC uses some other patterns that are closely related to Composite. It uses Decorator (called Wrapper in Smalltalk) to add borders or scrollbars to Views or to change the way they are clipped or positioned. It uses Adaptor to customize the interface between Views and Models, thereby making Views more reusable. It uses Abstract Factory to create views, thereby making it easier to change the look and feel of an application. But all of these patterns are enhanced by the fact that there is a universal View interface and that a set

of views can be treated as a single view, so in some sense they all follow from the Composite pattern.

The Strategy pattern

Many user interface frameworks do not separate views from controllers, but instead have a single "user interface object". A common criticism of MVC is that views and controllers are tightly coupled, and you cannot mix views and controllers arbitrarily. A text controller only works with a text view, and a list controller only works with list view. It is misleading to provide two separate class hierarchies when there is only one way to combine them.

The best way to think about controllers is that they are components of views. A view delegates to its controller responsibility for handling user input. The original MVC hid this fact because a controller would communicate with other controllers (the controllers of its subclass's subviews, to be exact). But the current version of MVC routes all operations to the view, which then delegates them to its controller. Thus, it is easier to see that the controller is just a component of the view.

Separating controllers from views sometimes lets you mix and match them. For example, you can give any view a NoController as a controller, and it will then ignore all user input. It also divides the responsibility of a view into two pieces along a natural boundary, making each easier to understand. The view keeps track of screen real estate, and knows how to translate data elements to screen coordinates and vice versa, while the controller keeps track of the mouse and keyboard and translates user events into operations on the view and model.

The Model-View-Controller pattern

The patterns I've described don't competely include Model-View-Controller. For example, they don't tell you that it is not all that common to subclass View and Controller, that in practice you can usually just combine instances of existing classes to make a user interface. Some of the things you need to know about MVC come from the design of individual classes rather than from the framework as a whole. So, what is the MVC pattern? Does it just consist of a set of simpler patterns, or is it something different?

In *Patterns Generate Architecture*, Kent Beck and I described the MVC pattern as:

1) Model-View-Controller

> Context A system is going to have a graphical user-interface.
>
> Problem Graphical user-interfaces can be hard to build. Users demand programs that are easy to use, easy to learn, and powerful, and a good user interface is necessary to achieve these goals. How should the responsibilities of implementing a user interface be divided between objects?
>
> Constraints Modern graphical user interfaces have a small number of recurring visual elements. Because user-interfaces need to be consistent, we depend on a few interaction techniques, such as menus, buttons, scroll-bars, and lists. The effort that someone puts into learning how to use one program, or one part of one program, should apply to other programs and other parts of the same program.
>
> In implementing a user interface, we must strike a balance between a design that uses many objects but is difficult to learn and one that uses few objects and sacrifices flexibility. One important axis of flexibility is the information that is displayed. A

second axis of flexibility, independent of display, is interpreting user gestures and mapping them into state changes. A third degree of freedom is the ability to put multiple user interfaces on the same information.

Therefore:

Solution Divide your system into three objects: a Model, View and Controller. The Model is responsible for maintaining state and surfacing the behavior necessary to support the user interface. The View is responsible for displaying an up-to-date version of the Model. The Controller is responsible for mapping user gestures to changes of state in the model.

Although this is a much higher level pattern than the ones described earlier, since it doesn't explain how objects interact or how responsibility is divided between them, it is still a design pattern. It focuses on the kind of objects you should have, and could be described by an object diagram. It is possible to think of MVC not as a design pattern, but as a development strategy. In fact, I think this is a more accurate description of why people use MVC.

2) Model-View-Controller

Context You are a Smalltalk-80 application programmer.

Problem You need to build a graphical user interface, but none of the existing components provide what you need.

Constraints If none of the existing components provide what you need, it might be easier to build them from scratch then to try to modify MVC to support your needs. On the other hand, your user interface should fit in with that of the rest of the system. Your system needs to be maintainable, reusable, and reliable. You aren't in the user interface framework business, but in the application business, so any time you spend writing your own user interface components is time you don't spend on your real business.

Therefore:

Solution Use MVC. Don't create a new user interface framework unless you are willing to support it and to rewrite it for each new version of Smalltalk.

In my opinion, the main reason that people use MVC is because it is the standard user interface framework of the environment in which they are programming. People who are not using Smalltalk don't use MVC, they use whatever user interface framework they can get their hands on. People do not pick MVC because its design is perfect, they pick it because it is good enough, it is available, and they know it. Thus, the *real* pattern that people are following is the second one. Or maybe I am just cynical!

Frameworks are a kind of pattern. But they are not patterns on the same level as the design patterns. They are composed of many design patterns, they are represented by code, and they are much more complex. In fact, it can take a whole set of patterns just to describe how to use a single framework, as I showed in *Documenting Frameworks using Patterns.*

Given a choice between solving a problem using a framework and solving it using some other kind of pattern, I usually go with a framework. Because frameworks are code, you can reuse them by calling or including the code in your program. Patterns that are not code have to be understood and then followed. I'd rather reuse a text editor than follow patterns for building a text editor and build it from scratch. But there are many problems that can't be solved by building a framework. One of them is obviously how to use the framework. Even when you can build a framework for something, frameworks are expensive to build.

Thus, there are many times that you have to reuse knowledge instead of reusing code. Both ways of reusing design are important.

References

Kent Beck and Ralph Johnson, "*Patterns Generate Architectures*", ECOOP '94 Proceedings, Springer-Verlog Lecture Notes in Computer Science #821: 139-149, July 1994.

Erich Gamma, Richard Helm, Ralph Johnson,and John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture* Addison-Wesley, 1995.

Ralph E. Johnson, "Documenting Frameworks with Patterns" *OOPSLA '92 Proceedings*, SIGPLAN Notices, 27(10): 63-76,Vancouver BC, October 1992.