Model-View-Controller (MVC) Architecture

Author: John Deacon

Synopsis: Although the MVC architecture (or pattern or idiom) has been around for a long time, and although it is important and widely used, much of the information regarding the idiom is available only as folklore rather than from textbooks. This note gives a short guide to MVC.

August 1995, revised August 2000, April 2005 and May 2009

(Official source of this document: http://www.jdl.co.uk/briefings/index.html#mvc)

1. Introduction

Software, of course, has to interact with something in order to be useful. Sometimes it interacts with other machines; very often it's with people. And so, of course, there are interfaces. Indeed, more effort often goes into an interface than goes into the remainder of the application.

It's reasonable to propose that any given application is likely to change its interface as time goes by, or indeed have several interfaces at any one point in time. Yet the underlying application might well be fairly constant. A banking application that used to sit behind character-based menu systems or command-line interfaces is likely to be the exact same application that today is probably sitting behind a graphical user interface (GUI). Building any particular interface into an application would be to the detriment of both the application, making it less flexible and harder to migrate; and the interface, making it harder to use for other applications. It makes sense then, to keep the essence of an application separate from any and all of its interfaces.

There have been development practices that actually discouraged such a separation. One of the principle objections to *rapid application prototyping* as an approach, for example, was its tempting of developers with, "Go on! We'll help you make a slick GUI and you can then just tack your application code on the back of the buttons." Never a good idea. Apart from not separating the application from the presentation, why should the considerations that influence the design of a GUI be the same considerations as influence the design of the application?

Long ago, in the 70's, Smalltalk defined an architecture to cope with this, called the Model-View-Controller architecture[1]. Since that time, the MVC design idiom has become commonplace, especially in object-oriented systems. As Smalltalk is the origin of the idiom, and as most of the literature is to be found in and around Smalltalk writings, I will use objects and Smalltalk for examples.

It's interesting to note that a new version of ASP.NET will be arriving soon (2009) with an MVC framework, to "stop program logic and presentation logic being mixed up together".

---

1. One cannot credit Smalltalk with inventing MVC. This credit should go to a Prof. Trygve Reenskaug.

2. Model-View-Controller

We will call the unchanging essence of the application/domain, the **model** (in the singular). In object-oriented terms, this will consist of the set of classes which model and support the underlying problem, and which therefore will tend to be stable and as long-lived as the problem itself.

How much should the model (classes) know about the connection to the outside world? Nothing, *absolutely* nothing.

3. Model-View-Controller

For a given situation, in a given version there will be one or more interfaces with the model, which we'll call the **views** (plural).

In object-oriented terms, these will consist of sets of classes which give us "windows" (very often actual windows) onto the model, e.g.

- The GUI/widget (graphical user interface) view,
- The CLI (command line interface) view,
- The API (application program interface) view.

Or:

- The novice view,
- The expert view.

Although views are very often graphical, they don't have to be.

What will the views know about the model? They have to know of its existence. They have to know something of its nature. A *bookingDate* entry field, for example, might display, and perhaps change, an instance variable of some model class somewhere.

4. Model-View-Controller

A controller is an object that lets you manipulate a view. Over-simplifying a bit, the controller handles the input whilst the view handles the output. Controllers have the most knowledge of platforms and operating systems. Views are fairly independent of whether their event come from *Microsoft Windows*, *X Windows* or whatever.

And, just as the views know their model but the model doesn't know its views, the controllers knows their views but the view doesn't know its controller.

Controllers were Smalltalk specific. They are not of general interest and are not covered in any greater depth here. In Java's Swing architecture, for example, the view and the controller are combined (this is often done in other architectures). In Swing the combined view/controller is called the delegate.

5. “Model” confusion

Smalltalk, then, can be credited with inventing and promoting the MVC architecture. But it could also be accused of confusing things. A better acronym for the architecture would be:

$\mathbf{M_dM_aVC}$

5.1 The domain model

What analysts and designers would think of as the “model” is the M_d part—the **domain model**.

The domain model will consist of the objects which represent and support the essence of the problem—*MagneticField*, *Client*, *Invoice*, *Booking*, …

These are the classes that today’s software engineering modelling and implementation would focus on first. Indeed it is usually considered crucial that the core structure of the solution matches an appropriate and useful structuring of the problem. The domain classes will truly know nothing about the mechanisms that interface them to the outside world.

What Smalltalk programmers (with “classic”, or “blue book”[2] MVC at least) sometimes mean by “model”, however, is the M_a part—the **application model**.

5.2 The application model

It should be clear, then, what the domain model classes’ objects will be doing—supporting and modelling the problem, the whole problem and nothing but the problem. What should the application model be doing?

The application model is the object that knows that views exist and that those views need some way of obtaining information and notification.

Many of the writings from the early days tend to start with an *application* model, and to treat the application model as though it were all that you would need. It would contain all of the model logic. So whilst some separation occurred, the “model” would know quite a lot about interfacing in general. It would have lists of dependents for update purposes; it would typically inherit a heap of mechanisms to facilitate connectivity with the views.

We, on the other hand, will be keeping the model logic in the *domain* model classes. And we will be putting whatever mechanisms the MVC idiom requires into an *application* model class (or classes). A better name for the application model would be *application coordinator*—indeed that is what it is called in some dialects of Smalltalk. Incidentally, if you want to look at Smalltalk in order to better understand MVC, the dialect which remained truest to the ideals of MVC was Smalltalk-80, and that meant the *ParcPlace* products *ObjectWorks* and *Visual Works*. (Corporate stupidity killed off a viable commercial Smalltalk; today you would have to take a look among the excellent free and open source Smalltalks. I, however, haven’t kept up with them all, and wouldn’t be able to advise which is best for what. For more on the story take a look at this Wiki entry and don’t take the title of the entry too seriously.)

---

2. The blue book is the original Goldberg and Robson book: “Smalltalk: the language and its implementation”.

Application model class would typically be implemented by inheriting from a base or library class (called

ViewManager in Windows ST/V,

ApplicationCoordinator in Windows ST/V 32,

Application in Mac ST/V and

Model in VisualWorks).

6. View to model communication

The views know of the model and will interact with the model.

- If a button is clicked an action message might be sent to a model object in order to get something done.
- If a new value is typed into an entry field an update message might be sent to a model object in order to give it its new value.
- If a value is needed for a display an enquiry message might be sent to a model object in order to get a value.

These messages will be sent when events occur.

It is then the job of the method in the application model responding to the message to obtain values, set values or make things happen. In naïve MVC the work would actually happen there. The more up-to-date and righteous way (M_dM_aVC) is to have the method in the application model send the appropriate messages to the domain model object(s).

7. Model (and controller) to view communication

We've said that the model does not know about its views. But surely the model needs to communicate with the views. What if some aspect of the model changes; an aspect that is displayed in a view; especially an aspect that is being displayed in more that one view? Won't the view have to be sent a message giving it the new value?

How can the model communicate with the view when it doesn't and shouldn't know which, if any, views exist? Indeed how can the controllers communicate with the views when sometimes an input event is interesting (left click on a button) and sometimes no view will be interested (double right click).

The controllers and the model communicate with the view via **events**. Events provide nicely decoupled mechanisms that allow communication, with minimal dependencies.

Using graphical components—a *list box,* an *entry field* or a *radio button*—as examples, these view components will receive event notifications such as *needs contents,* or *clicked.* The events will often come from the controller. Views register handlers for the events they wish to handle. In Smalltalk, during the setting up of a view, a series of *when: event send: message to: recipient* expressions will be used. The recipient of the message will be the application model. So when the *clicked* event occurs, a button might send the *recalibrate* message to the application model that owns the view of which it is part.

The answer, then, to the original question of this section is that the model can also trigger events. Let's be clear again which model class we are talking about—the application model. The domain model objects have no responsibilities other than receiving the messages defined by their interfaces.

Indeed, the most important role that the application model plays, is notifying any and all dependent views whenever a viewed aspect of the model changes (an application model's "distinguishing trait", as Smalltalk 80 put it).

The application model knows that it might have views, and it knows that those views will be dependent on it. So whenever an aspect of the model changes, the application model triggers an event. It doesn't know who, if anyone, is interested but any dependents listening for that event (the views and their components) can then take appropriate action. The action they will take will be to send messages to model objects in order to obtain the latest values of the aspects they are viewing.

8. Application model to domain model communication

How it the domain model connected to the application model? Even Smalltalk dialects vary quite widely.

The simplest scheme is to put the domain model object(s) into the application model as instance variables and to give the application model knowledge of what accessor and update methods of the domain model classes it should use.

We expect to reuse views though. A text pane should be useful to thousands of models. We can also expect that views will be built by code generators. Writing view code by hand is error-prone, tedious and unnecessary; screen painters can do it for us. How can re-usable views or generated views know what the accessor and update methods of the domain model classes are? They can't; some convention or translation is needed.

Visual Works, for example, defines an extension to M_dM_aVC that uses adaptors. The view components (widgets) send *value* and *value:* messages to their respective adaptors and the adaptors know (they've been told) what messages to send on to their domain model objects.

9. About the author

John Deacon is a lecturer and writer. *Object-Oriented Analysis and Design: A Pragmatic Approach* was published by Addison-Wesley in 2005 (0-321-26317-0), and is a fresh look at the practice of analysis and design in the light of what we have learned about the nature of systems and object technology over the last twenty years. It proposes, for example, that many of us have been spending too much time for too little return in object-oriented analysis, and that our approaches to object-oriented design have been inside-out.

John's course offerings include:

- Object-Oriented Analysis and Design (with UML 1.x or UML 2.0)
- Hands-On Design and Programming with C++
- Advanced C++ : Traps and Pitfalls

10. References

Smalltalk: the Language and its Implementation, Goldberg & Robson. This is the "blue book", the original bible from the creators of Smalltalk. It is now out of print, and of its two-volume replacement, only the first volume is easily available—*Smalltalk-80: The Language*, Goldberg & Robson, Addison-Wesley 1989—and this does not cover MVC.

A cookbook approach to using MVC, Krasner and Pope, JOOP 1(3):26–49. A description of classic MVC, rather than the M_dM_aVC described here.

Remembrance of things past: Layered architectures for Smalltalk applications, Kyle Brown, The Smalltalk Report 4(9):4–7. The MVC acronym isn't actually used but a nice description of the domain/application separation.

Making MVC code more reusable, Bobby Woolf, The Smalltalk Report 4(4):15–18. Discusses how adaptors can be used in generated views.

