Model View Controller

Probably the widest quoted pattern in UI development is Model View Controller (MVC) - it's also the most misquoted. I've lost count of the times I've seen something described as MVC which turned out to be nothing like it. Frankly a lot of the reason for this is that parts of classic MVC don't really make sense for rich clients these days. But for the moment we'll take a look at its origins.

As we look at MVC it's important to remember that this was one of the first attempts to do serious UI work on any kind of scale. Graphical User Interfaces were not exactly common in the 70's. The Forms and Controls model I've just described came after MVC - I've described it first because it's simpler, not always in a good way. Again I'll discuss Smalltalk 80's MVC using the assessment example - but be aware that I am taking a few liberties with the actual details of Smalltalk 80 to do this - for start it was a monochrome system.

At the heart of MVC, and the idea that was the most influential to later frameworks, is what I call Separated Presentation. The idea behind Separated Presentation is to make a clear division between domain objects that model our perception of the real world, and presentation objects that are the GUI elements we see on the screen. Domain objects should be completely self contained and work without reference to the presentation, they should also be able to support multiple presentations, possibly simultaneously. This approach was also an important part of the Unix culture, and continues today allowing many applications to be manipulated through both a graphical and command-line interface.

In MVC, the domain element is referred to as the model. Model objects are completely ignorant of the UI. To begin discussing our assessment UI example we'll take the model as a reading, with fields for all the interesting data upon it. (As we'll see in a moment the presence of the list box makes this question of what is the model rather more complex, but we'll ignore that list box for a little bit.)

In MVC I'm assuming a Domain Model of regular objects, rather than the Record Setnotion that I had in Forms and Controls. This reflects the general assumption behind the design. Forms and Controls assumed that most people wanted to easily manipulate data from a relational database, MVC assumes we are manipulating regular Smalltalk objects.

The presentation part of MVC is made of the two remaining elements: view and controller. The controller's job is to take the user's input and figure out what to do with it.

At this point I should stress that there's not just one view and controller, you have a view-controller pair for each element of the screen, each of the controls and the screen as a whole. So the first part of reacting to the user's input is the various controllers collaborating to see who got edited. In this case that's the actuals text field so that text field controller would now handle what happens next.

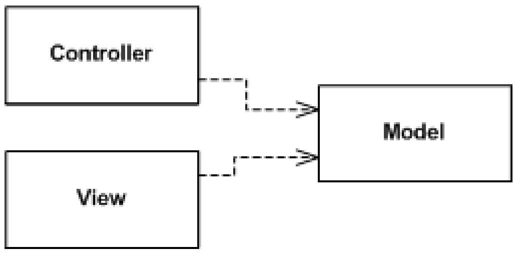

Figure 4: Essential dependencies between model, view, and controller. (I call this essential because in fact the view and controller do link to each other directly, but developers mostly don't use this fact.)

Like later environments, Smalltalk figured out that you wanted generic UI components that could be reused. In this case the component would be the view-controller pair. Both were generic classes, so needed to be plugged into the application specific behavior. There would be an assessment view that would represent the whole screen and define the layout of the lower level controls, in that sense similar to a form in Forms and Controllers. Unlike the form, however, MVC has no event handlers on the assessment controller for the lower level components.

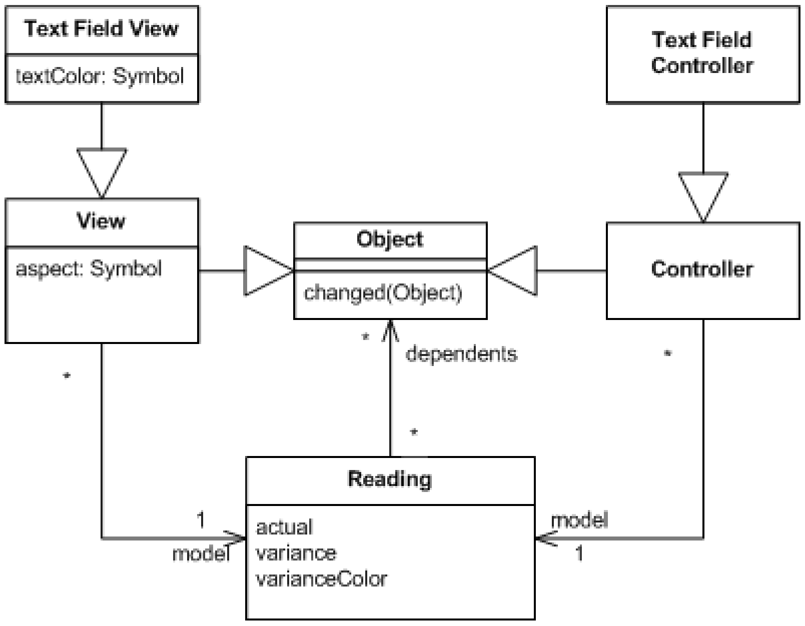

Figure 5: Classes for an MVC version of an ice-cream monitor display

The configuration of the text field comes from giving it a link to its model, the reading, and telling it what what method to invoke when the text changes. This is set to '#actual:' when the screen is initialized (a leading '#' indicates a symbol, or interned string, in Smalltalk). The text field controller then makes a reflective invocation of that method on the reading to make the change. Essentially this is the same mechanism as occurs for Data Binding, the control is linked to the underlying object (row) and told which method (column) it manipulates.

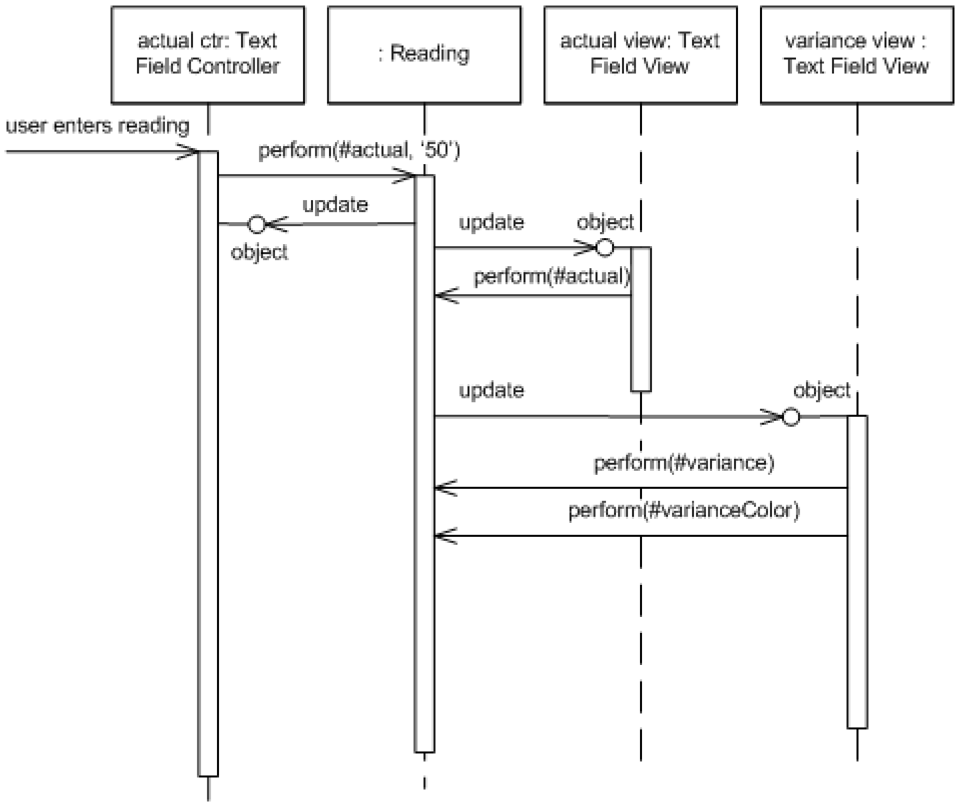

Figure 6: Changing the actual value for MVC.

So there is no overall object observing low level widgets, instead the low level widgets observe the model, which itself handles many of the decision that would be made by the form. In this case, when it comes to figuring out the variance, the reading object itself is the natural place to do that.

Observers do occur in MVC, indeed it's one of the ideas credited to MVC. In this case all the views and controllers observe the model. When the model changes, the views react. In this case the actual text field view is notified that the reading object has changed, and invokes the method defined as the aspect for that text field - in this case #actual - and sets its value to the result. (It does something similar for the color, but this raises its own specters that I'll get to in a moment.)

You'll notice that the text field controller didn't set the value in the view itself, it updated the model and then just let the observer mechanism take care of the updates. This is quite different to the forms and controls approach where the form updates the control and relies on data binding to update the underlying record-set. These two styles I describe as patterns: Flow Synchronization and Observer Synchronization. These two patterns describe alternative ways of handling the triggering of synchronization between screen state and session state. Forms and Controls do it through the flow of the application manipulating the various controls that need to be updated directly. MVC does it by making updates on the model and then relying of the observer relationship to update the views that are observing that model.

Flow Synchronization is even more apparent when data binding isn't present. If the application needs to do synchronization itself, then it was typically done at important point in the application flow - such as when opening a screen or hitting the save button.

One of the consequences of Observer Synchronization is that the controller is very ignorant of what other widgets need to change when the user manipulates a particular widget. While the form needs to keep tabs on things and make sure the overall screen state is consistent on a change, which can get pretty involved with complex screens, the controller in Observer Synchronization can ignore all this.

This useful ignorance becomes particularly handy if there are multiple screens open viewing the same model objects. The classic MVC example was a spreadsheet like screen of data with a couple of different graphs of that data in separate windows. The spreadsheet window didn't need to be aware of what other windows were open, it just changed the model and Observer Synchronization took care of the rest. With Flow Synchronization it would need some way of knowing which other windows were open so it tell them to refresh.

While Observer Synchronization is nice it does have a downside. The problem withObserver Synchronization is the core problem of the observer pattern itself - you can't tell what is happening by reading the code. I was reminded of this very forcefully when trying to figure out how some Smalltalk 80 screens worked. I could get so far by reading the code, but once the observer mechanism kicked in the only way I could see what was going on was via a debugger and trace statements. Observer behavior is hard to understand and debug because it's implicit behavior.

While the different approaches to synchronization are particularly noticeable from looking at the sequence diagram, the most important, and most influential, difference is MVC's use of Separated Presentation. Calculating the variance between actual and target is domain behavior, it is nothing to do with the UI. As a result following Separated Presentation says we should place this in the domain layer of the system - which is exactly what the reading object represents. When we look at the reading object, the variance feature makes complete sense without any notion of the user interface.

At this point, however, we can begin to look at some complications. There's two areas where I've skipped over some awkward points that get in the way of MVC theory. The first problem area is to deal with setting the color of the variance. This shouldn't really fit into a domain object, as the color by which we display a value isn't part of the domain. The first step in dealing with this is to realize that part of the logic is domain logic. What we are doing here is making a qualitative statement about the variance, which we could term as good (over by more than 5%), bad (under by more than 10%), and normal (the rest). Making that assessment is certainly domain language, mapping that to colors and altering the variance field is view logic. The problem lies in where we put this view logic - it's not part of our standard text field.

This kind of problem was faced by early smalltalkers and they came up with some solutions. The solution I've shown above is the dirty one - compromise some of the purity of the domain in order to make things work. I'll admit to the occasional impure act - but I try not to make a habit of it.

We could do pretty much what Forms and Controls does - have the assessment screen view observe the variance field view, when the variance field changes the assessment screen could react and set the variance field's text color. Problems here include yet more use of the observer mechanism - which gets exponentially more complicated the more you use it - and extra coupling between the various views.

A way I would prefer is to build a new type of the UI control. Essentially what we need is a UI control that asks the domain for a qualitative value, compares it to some internal table of values and colors, and sets the font color accordingly. Both the table and message to ask the domain object would be set by the assessment view as it's assembling itself, just as it sets the aspect for the field to monitor. This approach could work very well if I can easily subclass text field to just add the extra behavior. This obviously depends on how well the components are designed to enable sub-classing - Smalltalk made it very easy - other environments can make it more difficult.

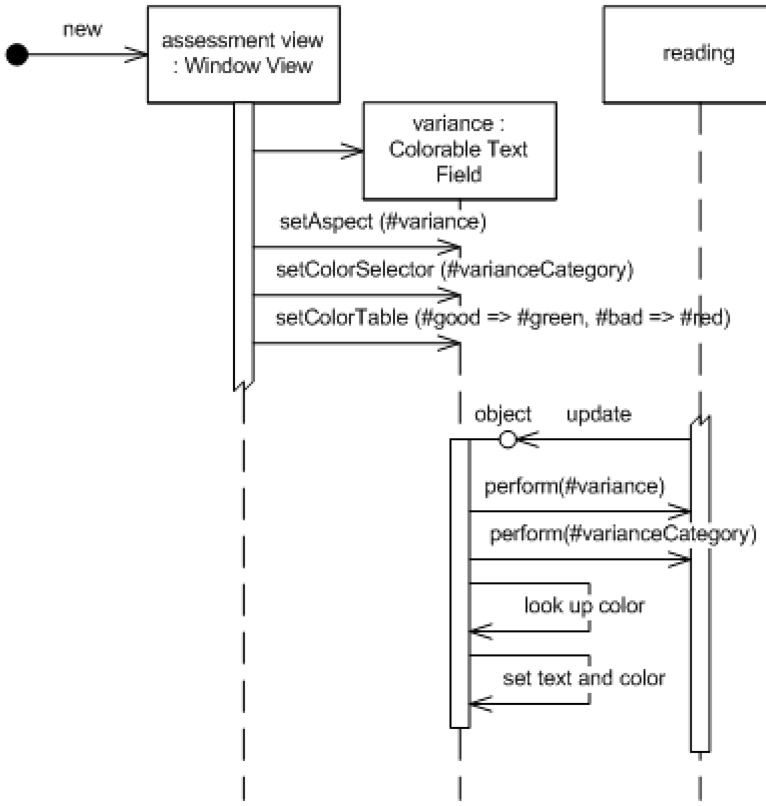
new
assessment view : Window View
reading
variance : Colorable Text Field
setAspect (#variance)
setColorSelector (#varianceCategory)
setColorTable (#good => #green, #bad => #red)
object
update
perform(#variance)
perform(#varianceCategory)
look up color
set text and color

Figure 7: Using a special subclass of text field that can be configured to determine the color.

The final route is to make a new kind of model object, one that's oriented around around the screen, but is still independent of the widgets. It would be the model for the screen. Methods that were the same as those on the reading object would just be delegated to the reading, but it would add methods that supported behavior relevant only to the UI, such as the text color.

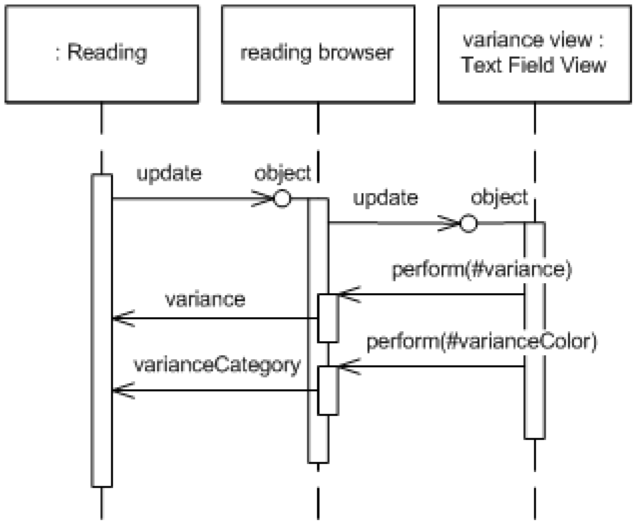

Figure 8: Using an intermediate Presentation Model to handle view logic.

This last option works well for a number of cases and, as we'll see, became a common route for Smalltalkers to follow - I call this a Presentation Model because it's a model that is really designed for and thus part of the presentation layer.

The Presentation Model works well also for another presentation logic problem - presentation state. The basic MVC notion assumes that all the state of the view can be derived from the state of the model. In this case how do we figure out which station is selected in the list box? The Presentation Model solves this for us by giving us a place to put this kind of state. A similar problem occurs if we have save buttons that are only enabled if data has changed - again that's state about our interaction with the model, not the model itself.

So now I think it's time for some soundbites on MVC.

- Make a strong separation between presentation (view & controller) and domain (model) - Separated Presentation.
- Divide GUI widgets into a controller (for reacting to user stimulus) and view (for displaying the state of the model). Controller and view should (mostly) not communicate directly but through the model.
- Have views (and controllers) observe the model to allow multiple widgets to update without needed to communicate directly - Observer Synchronization.