| EAA-dev Home |

WORK-IN-PROGRESS: - this material is still under development

GUI Architectures

There have been many different ways to organize the code for a rich client system. Here I discuss a selection of those that I feel have been the most influential and introduce how they relate to the patterns.

Last significant update: 18 Jul 06

Graphical user interfaces have become a familiar part of our software landscape, both as users and as developers. Looking at it from a design perspective they represent a particular set of problems in system design - problems that have led to a number of different but similar solutions.

My interest is identifying common and useful patterns for application developers to use in rich-client development. I've seen various designs in project reviews and also various designs that have been written in a more permanent way. Inside these designs are the useful patterns, but describing them is often not easy. Take Model-View-Controller as an example. It's often referred to as a pattern, but I don't find it terribly useful to think of it as a pattern because it contains quite a few different ideas. Different people reading about MVC in different places take different ideas from it and describe these as 'MVC'. If this doesn't cause enough confusion you then get the effect of misunderstandings of MVC that develop through a system of Chinese whispers.

In this essay I want to explore a number of interesting architectures and describe my interpretation of their most interesting features. My hope is that this will provide a context for understanding the patterns that I describe.

To some extent you can see this essay as a kind of intellectual history that traces ideas in UI design through multiple architectures over the years. However I must issue a caution about this. Understanding architectures isn't easy, especially when many of them change and die. Tracing the spread of ideas is even harder, because people read different things from the same architecture. In particular I have not done an exhaustive examination of the architectures I describe. What I have done is referred to common descriptions of the designs. If those descriptions miss things out, I'm utterly ignorant of that. So don't take my descriptions as an authoritative description of the architectures I describe. Furthermore there things I've left out or simplified if I didn't think they were particularly relevant. Remember my primary interest is the underlying patterns, not in the history of these designs.

(There is something of an exception here, in that I did have access to a running Smalltalk-80 to examine MVC. Again I wouldn't describe my examination of it as exhaustive, but it did reveal things that common descriptions of it failed to - which even further makes me cautious about descriptions of other architectures that I have here. If you are familiar with one of these

architectures and you see I have something important that is incorrect and missing I'd like to know about it. I also thing that a more exhaustive survey of this territory would be a good object of academic study.)

Forms and Controls

I shall begin this exploration with an architecture that is both simple and familiar. It doesn't have a common name, so for the purposes of this essay I shall call it "Forms and Controls". It's a familiar architecture because it was the one encouraged by client-server development environments in the 90's - tools like Visual Basic, Delphi, and Powerbuilder. It continues to be commonly used, although also often vilified by design geeks like me.

To explore it, and indeed the other architectures, I'll use a common example. In New England, where I live, there is a government program that monitors the amount of ice-cream particulate in the atmosphere. If the concentration is too low, this indicates that we aren't eating enough ice-cream - which poses a serious risk to our economy and public order. (I like to use examples that are no less realistic as you usually find in books like this.)

To monitor our ice-cream health, the government has set up monitoring stations all over the New England states. Using complex atmospheric modeling the department sets a target for each monitoring station. Every so often staffers go out on an assessment where they go to various stations and note the actual ice-cream particulate concentrations. This UI allows them to select a station, and enter the date and actual value. The system then calculates and displays the variance from the target. Furthermore if the actual is more than 10% below the target, the variance is highlighted in red, if above by more than 5% it's highlighted in green.

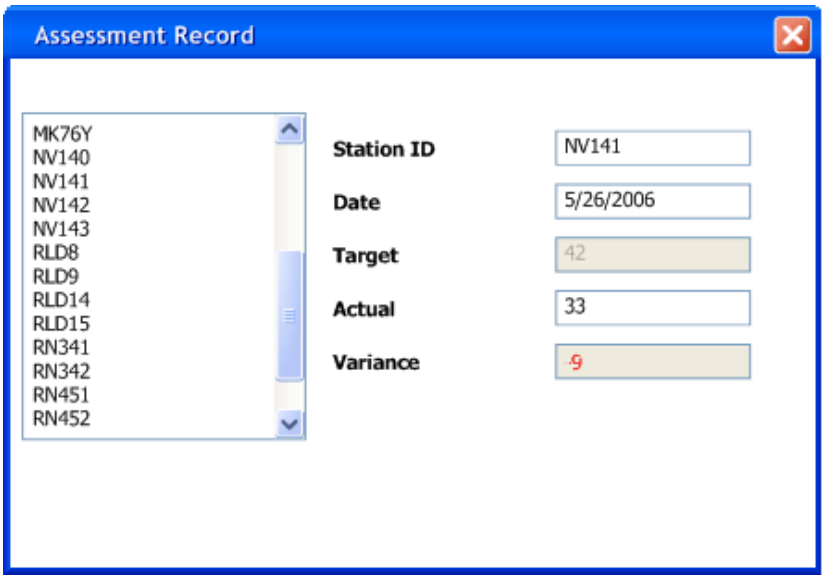

Figure 1: The UI I'll use as an example.

As we look at this screen we can see there is an important division as we put it together. The form is specific to our application, but it uses controls that are generic. Most GUI environments come with a hefty bunch of common controls that we can just use in our application. We can build new controls ourselves, and often it's a good idea to do so, but there is still a distinction between generic reusable controls and specific forms. Even specially written controls can be reused across multiple forms.

The form contains two main responsibilities:

- Screen layout: defining the arrangement of the controls on the screen, together with their hierarchic structure with other.
- Form Logic: behavior that cannot be easily programmed into the controls themselves.

Most GUI development environments allow the developer to define screen layout with a graphical editor that allows you to drag and drop the controls onto a space for the form. This pretty much handles the form layout. This way it's easy to setup a pleasing layout of controls on the form (although it isn't always the best way to do it - we'll come to that later.)

The controls display data - in this case about the reading. This data will pretty much always come from somewhere else, in this case let's assume a SQL database as that's the environment that most of these client server tools assume. In most situations there are three copies of the data involved:

- One copy of data lies in the database itself. This copy is the lasting record of the data, so I call it the **record state**. The record state is usually shared and visible to multiple people via various mechanisms.
- A further copy lies inside in-memory Record Sets within the application. Most client-server environments provided tools which made this easy to do. This data was only relevant for one particular session between the application and the database, so I call it **session state**. Essentially this provides a temporary local version of the data that the user works on until they save, or commit it, back to the database - at which point it merges with the record state. I won't worry about the issues around coordinating record state and session state here: I did go into various techniques in [P of EAA].
- The final copy lies inside the GUI components themselves. This, strictly, is the data they see on the screen, hence I call it the **screen state**. It is important to the UI how screen state and session state are kept synchronized.

Keeping screen state and session state synchronized is an important task. A tool that helped make this easier was Data Binding. The idea was that any change to either the control data, or the underlying record set was immediately propagated to the other. So if I alter the actual reading on the screen, the text field control effectively updates the correct column in the underlying record set.

In general data binding gets tricky because if you have to avoid cycles where a change to the control, changes the record set, which updates the control, which updates the record set.... The flow of usage helps avoid these - we load from the session state to the screen when the screen is opened, after that any changes to the screen state propagate back to the session state. It's unusual for the session state to be updated directly once the screen is up. As a result data binding might not be entirely bi-directional - just confined to initial upload and then propagating changes from the controls to the session state.

Data Binding handles much of the functionality of a client sever application pretty nicely. If I change the actual value the column is updated, even changing the selected station alters the currently selected row in the record set, which causes the other controls to refresh.

Much of this behavior is built in by the framework builders, who look at common needs and make it easy to satisfy them. In particular this is done by setting values, usually called properties, on the controls. The control binds to a particular column in a record set by having its column name set through a simple property editor.

Using data binding, with the right kind of parameterization, can take you a long way. However it can't take you all the way - there's almost always some logic that won't fit with the parameterization options. In this case calculating the variance is an example of something that doesn't fit in this built in behavior - since it's application specific it usually lies in the form.

In order for this to work the form needs to be alerted whenever the value of the actual field

changes, which requires the generic text field to call some specific behavior on the form. This is a bit more involved than taking a class library and using it through calling it as Inversion of Control is involved.

There are various ways of getting this kind of thing to work - the common one for client-server tool-kits was the notion of events. Each control had a list of events it could raise. Any external object could tell a control that it was interested in an event - in which case the control would call that external object when the event was raised. Essentially this is just a rephrasing of the Observer pattern where the form is observing the control. The framework usually provided some mechanism where the developer of the form could write code in a subroutine that would be invoked when the event occurred. Exactly how the link was made between event and routine varied between platform and is unimportant for this discussion - the point is that some mechanism existed to make it happen.

Once the routine in the form has control, it can then do whatever it needed. It can carry out the specific behavior and then modify the controls as necessary, relying on data binding to propagate any of these changes back to the session state.

This is also necessary because data binding isn't always present. There is a large market for windows controls, not all of them do data binding. If data binding isn't present then it's up to the form to carry out the synchronization. This could work by pulling data out of the record set into the widgets initially, and copying the changed data back to the record set when the save button was pressed.

Let's examine our editing of the actual value, assuming that data binding is present. The form object holds direct references to the generic controls. There'll be one for each control on the screen, but I'm just interested in the actual, variance, and target fields here.

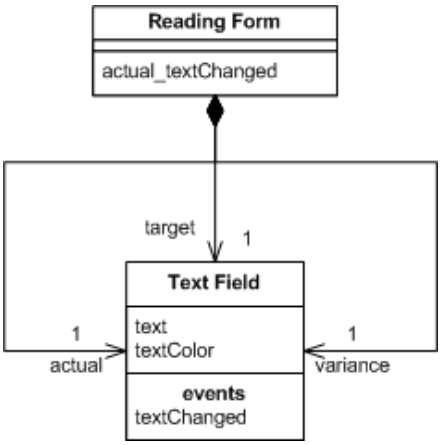

Figure 2: Class diagram for forms and controls

The text field declares an event for text changed, when the form assembles the screen during initialization it subscribes itself to that event, binding it a method on itself - here `actual_textChanged`.

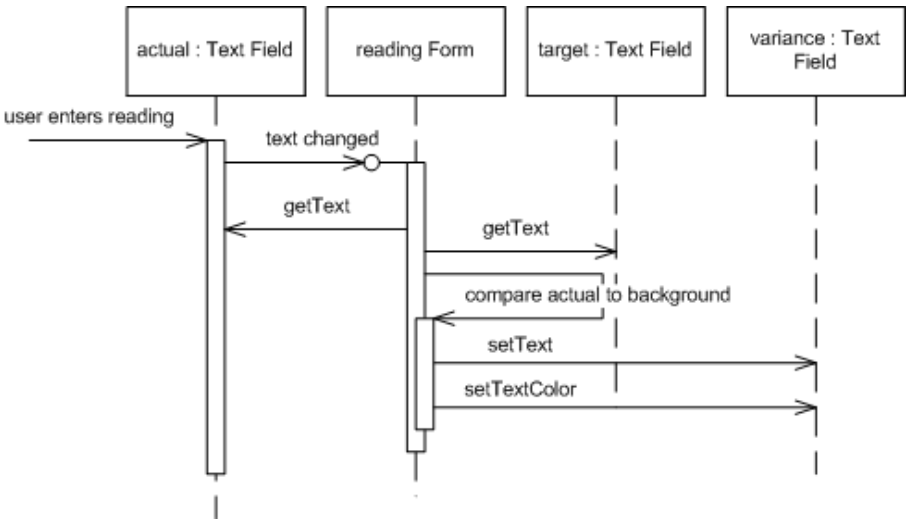

Figure 3: Sequence diagram for changing a genre with forms and controls.

When the user changes the actual value, the text field control raises its event and through the magic of framework binding the `actual_textChanged` is run. This method gets the text from the actual and target text fields, does the subtraction, and puts the value into the variance field. It also figures out what color the value should be displayed with and adjusts the text color appropriately.

We can summarize the architecture with a few soundbites:

- Developers write application specific forms that use generic controls.
- The form describes the layout of controls on it.
- The form observes the controls and has handler methods to react to interesting events raised by the controls.
- Simple data edits are handled through data binding.
- Complex changes are done in the form's event handling methods.

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 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 Set notion 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 the case that's it's the actuals text field so that text field controller would now handle what happens next.

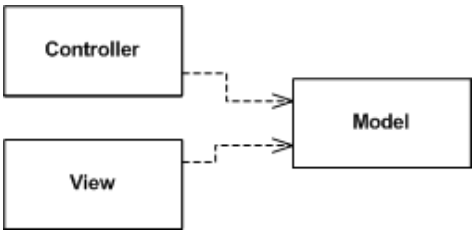

Figure 4: Essential dependencies between model, view, and controller. (I call this essential because in fact the view and controller do link to each other directly, but developers mostly don't use this fact.)

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.

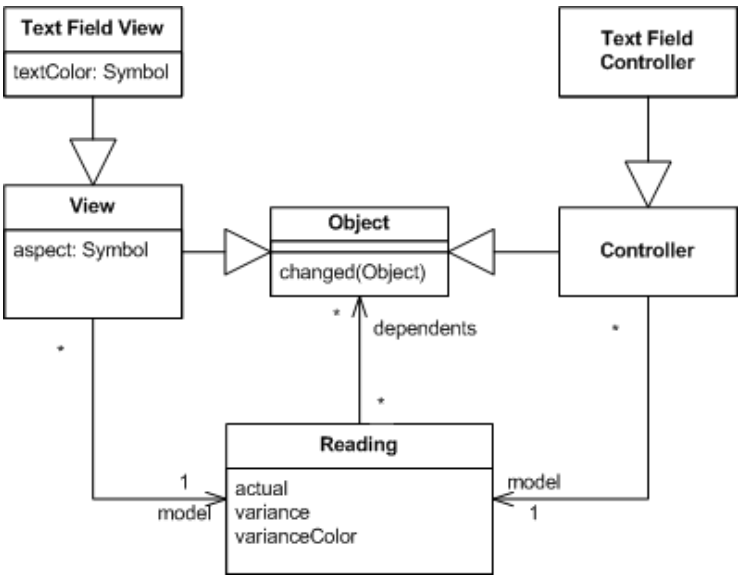

Figure 5: Classes for an MVC version of an ice-cream monitor display

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.

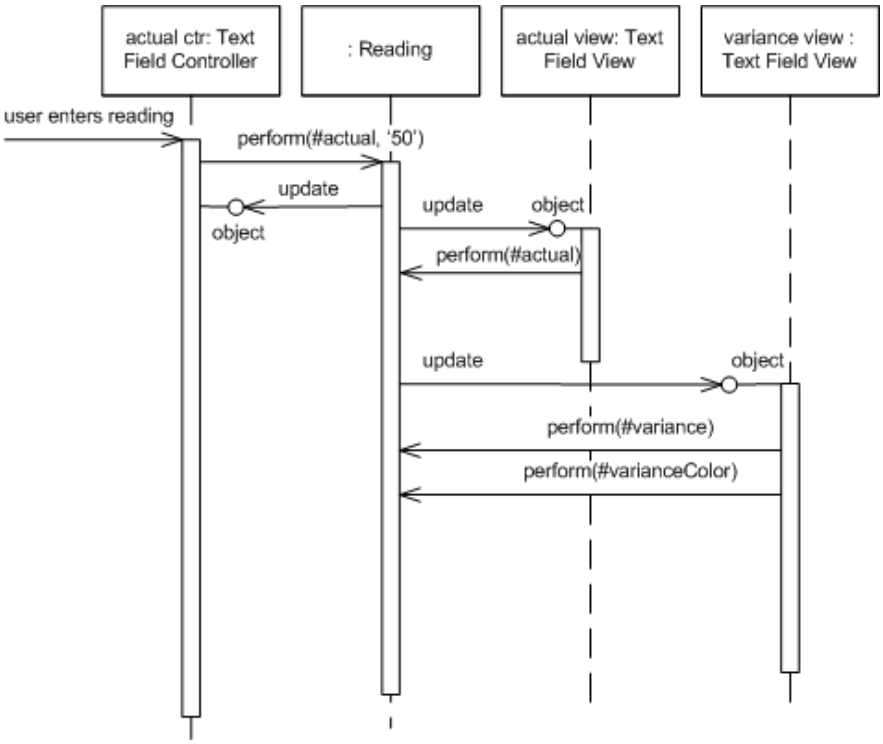

Figure 6: Changing the actual value for MVC.

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 with Observer 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 obvious depends on how well the components are designed to enable sub-classing - Smalltalk made it very easy - other environments can make it more difficult.

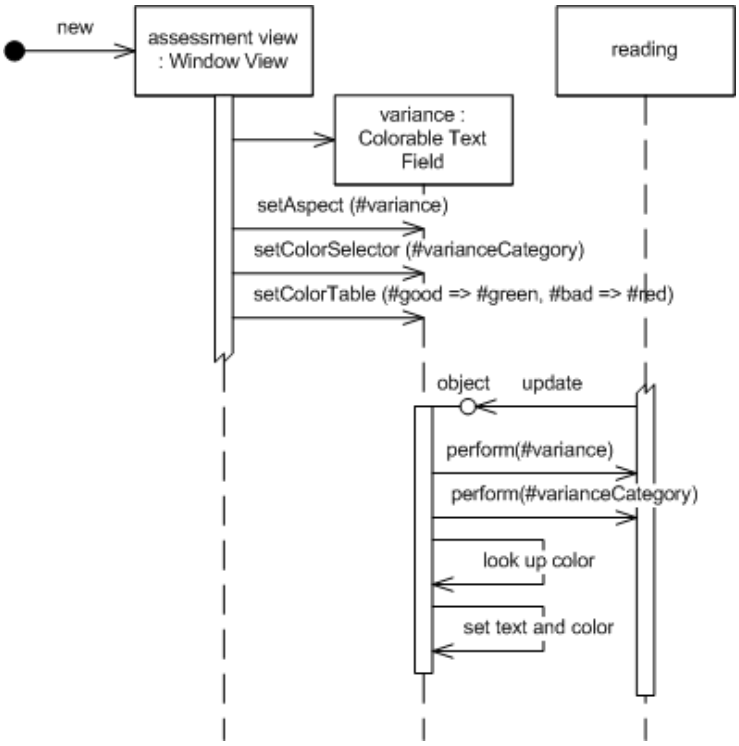

Figure 7: Using a special subclass of text field that can be configured to determine the color.

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.

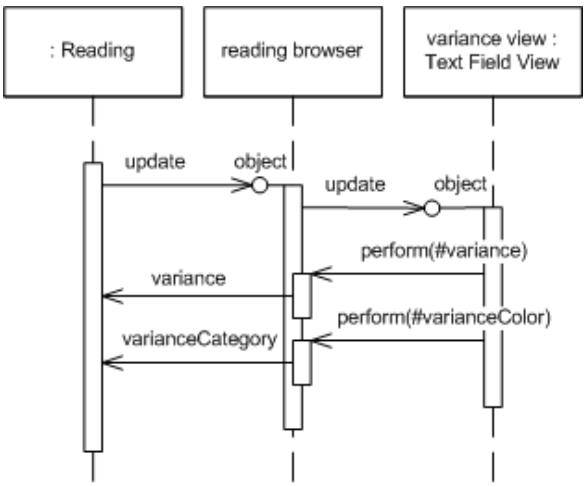

Figure 8: Using an intermediate Presentation Model to handle view logic.

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.

VisualWorks Application Model

As I've discussed above, Smalltalk 80's MVC was very influential and had some excellent features, but also some faults. As Smalltalk developed in the 80's and 90's this led to some significant variations on the classic MVC model. Indeed one could almost say that MVC disappeared, if you consider the view/controller separation to be an essential part of MVC - which the name does imply.

The things that clearly worked from MVC were Separated Presentation and Observer Synchronization. So these stayed as Smalltalk developed - indeed for many people they were the key element of MVC.

Smalltalk also fragmented in these years. The basic ideas of Smalltalk, including the (minimal) language definition remained the same, but we saw multiple Smalltalks develop with different libraries. From a UI perspective this became important as several libraries started using native widgets, the controls used by the Forms and Controls style.

Smalltalk was originally developed by Xerox Parc labs and they span off a separate company, ParcPlace, to market and develop Smalltalk. ParcPlace Smalltalk was called VisualWorks and made a point of being a cross-platform system. Long before Java you could take a Smalltalk program written in Windows and run it right away on Solaris. As a result VisualWorks didn't use native widgets and kept the GUI completely within Smalltalk.

In my discussion of MVC I finished with some problems of MVC - particularly how to deal with view logic and view state. VisualWorks refined its framework to deal with this by coming up with a construct called the Application Model - a construct that moves towards Presentation Model. The idea of using something like a Presentation Model wasn't new to VisualWorks - the original Smalltalk 80 code browser was very similar, but the VisualWorks Application Model baked it fully into the framework.

A key element of this kind of smalltalk was the idea of turning properties into objects. In our usual notion of objects with properties we think of a Person object having properties for name and address. These properties may be fields, but could be something else. There is usually a standard convention for accessing the properties: in Java we would see `temp = aPerson.getName()` and `aPerson.setName("martin")`, in C# it would `temp = aPerson.name` and `aPerson.name = "martin"`.

A **Property Object** changes this by having the property return an object that wraps the actual value. So in VisualWorks when we ask for a name we get back a wrapping object. We then get the actual value by asking the wrapping object for its value. So accessing a person's name would use `temp = aPerson name value` and `aPerson name value: 'martin'`

Property objects make the mapping between widgets and model a little easier. We just have to tell the widget what message to send to get the corresponding property, and the widget knows to access the proper value using `value` and `value:`. VisualWorks's property objects also allow you to set up observers with the message onChangeSend: aMessage to: anObserver.

You won't actually find a class called property object in Visual Works. Instead there were a number of classes that followed the value/value:/onChangeSend: protocol. The simplest is the ValueHolder - which just contains its value. More relevant to this discussion is the AspectAdaptor. The AspectAdaptor allowed a property object to wrap a property of another object completely. This way you could define a property object on a PersonUI class that wrapped a property on a Person object by code like

```
adaptor := AspectAdaptor subject: person
adaptor forAspect: #name
adaptor onChangeSend: #redisplay to: self
```

So let's see how the application model fits into our running example.

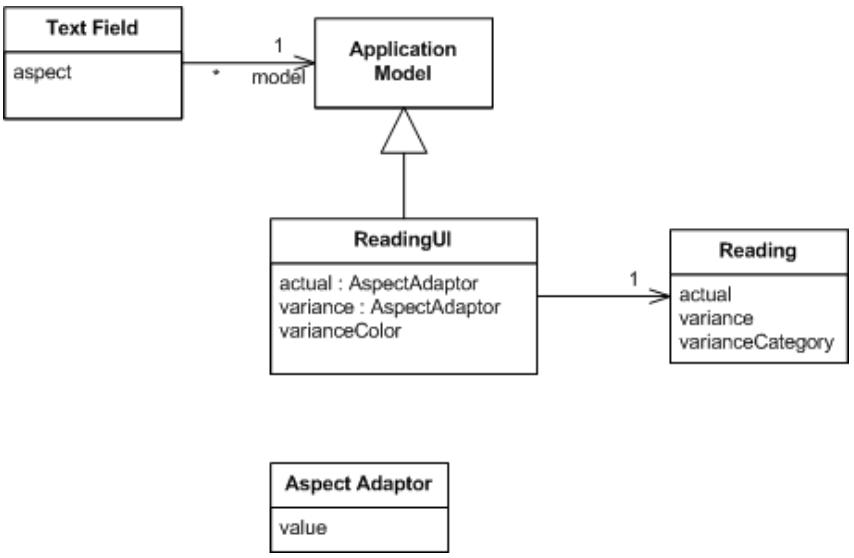

Figure 9: Class diagram for visual works application model on the running example

The main difference between using an application model and classic MVC is that we now have an intermediate class between the domain model class (Reader) and the widget - this is the application model class. The widgets don't access the domain objects directly - their model is the application model. Widgets are still broken down into views and controllers, but unless you're building new widgets that distinction isn't important.

When you assemble the UI you do so in a UI painter, while in that painter you set the aspect for each widget. The aspect corresponds to a method on the application model that returns a property object.

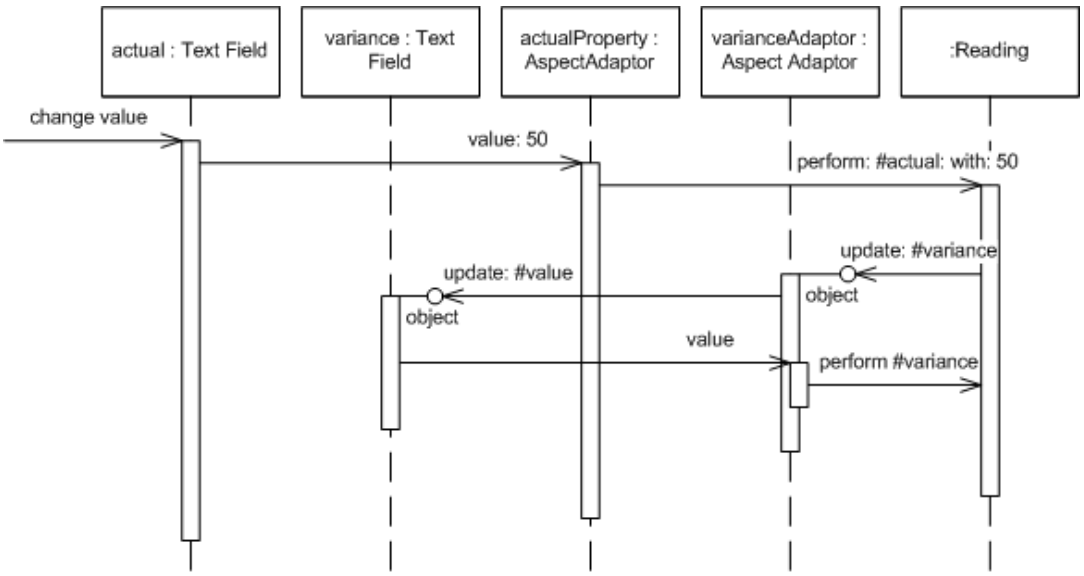

Figure 10: Sequence diagram showing how updating the actual value updates the variance text.

Figure 10 shows how the basic update sequence works. When I change a value in text field, that field then updates the value in the property object inside the application model. That update follows through to the underlying domain object, updating its actual value.

At this point the observer relationships kick in. We need to set things up so that updating the actual value causes the reading to indicate that it has changed. We do this by putting a call in the modifier for actual to indicate that the reading object has changed - in particular that the

variance aspect has changed. When setting up the aspect adaptor for variance it's easy to tell it to observe the reader, so it picks up the update message which it then forwards to its text field. The text field then initiates getting a new value, again through the aspect adaptor.

Using the application model and property objects like this helps us wire up the updates without having to write much code. It also supports fine-grained synchronization (which I don't think is a good thing).

Application models allow us to separate behavior and state that's particular to the UI from real domain logic. So one of the problems I mentioned earlier, holding the currently selected item in a list, can be solved by using a particular kind of aspect adaptor that wraps the domain model's list and also stores the currently selected item.

The limitation of all this, however, is that for more complex behavior you need to construct special widgets and property objects. As an example the provided set of objects don't provide a way to link the text color of the variance to the degree of variance. Separating the application and domain models does allow us to separate the decision making in the right way, but then to use widgets observing aspect adapters we need to make some new classes. Often this was seen as too much work, so we could make this kind of thing easier by allowing the application model to access the widgets directly, as in Figure 11.

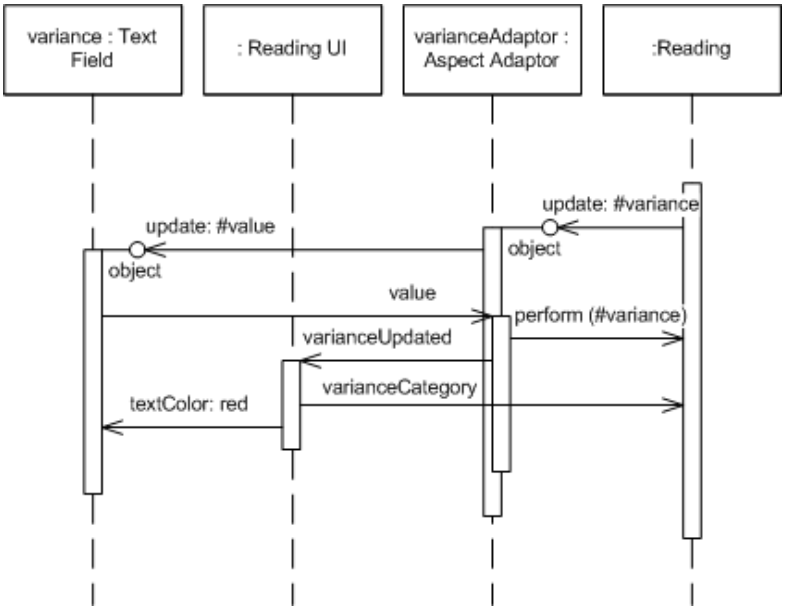

Figure 11: Application Model updates colors by manipulating widgets directly.

Directly updating the widgets like this is not part of Presentation Model, which is why the visual works application model isn't truly a Presentation Model. This need to manipulate the widgets directly was seen by many as a bit of dirty work-around and helped develop the Model-View-Presenter approach.

So now the soundbites on Application Model

- Followed MVC in using Separated Presentation and Observer Synchronization.
- Introduced an intermediate application model as a home for presentation logic and state - a partial development of Presentation Model.
- Widgets do not observe domain objects directly, instead they observe the application model.
- Made extensive use of Property Objects to help connect the various layers and to support the fine grained synchronization using observers.
- It wasn't the default behavior for the application model to manipulate widgets, but it was

commonly done for complicated cases.

Model-View-Presenter (MVP)

MVP is an architecture that first appeared in IBM and more visibly at Taligent the 1990's. It's most commonly referred via the Potel paper. The idea was further popularized and described by the developers of Dolphin Smalltalk. As we'll see the two descriptions don't entirely mesh but the basic idea underneath it has become popular.

To approach MVP I find it helpful to think about a significant mismatch between two strands of UI thinking. On the one hand is the Forms and Controller architecture which was mainstream approach to UI design, on the other is MVC and its derivatives. The Forms and Controls model provides a design that is easy to understand and makes a good separation between reusable widgets and application specific code. What it lacks, and MVC has so strongly, is Separated Presentation and indeed the context of programming using a Domain Model. I see MVP as a step towards uniting these streams, trying to take the best from each.

The first element of Potel is to treat the view as structure of widgets, widgets that correspond to the controls of the Forms and Controls model and remove any view/controller separation. The view of MVP is a structure of these widgets. It doesn't contain any behavior that describes how the widgets react to user interaction.

The active reaction to user acts lives in a separate presenter object. The fundamental handlers for user gestures still exist in the widgets, but these handlers merely pass control to the presenter.

The presenter then decides how to react to the event. Potel discusses this interaction primarily in terms of actions on the model, which it does by a system of commands and selections. A useful thing to highlight here is the approach of packaging all the edits to the model in a command - this provides a good foundation for providing undo/redo behavior.

As the Presenter updates the model, the view is updated through the same Observer Synchronization approach that MVC uses.

The Dolphin description is similar. Again the main similarity is the presence of the presenter. In the Dolphin description there isn't the structure of the presenter acting on the model through commands and selections. There is also explicit discussion of the presenter manipulating the view directly. Potel doesn't talk about whether presenters should do this or not, but for Dolphin this ability was essential to overcoming the kind of flaw in Application Model that made it awkward for me to color the text in the variation field.

One of the variations in thinking about MVP is the degree to which the presenter controls the widgets in the view. On one hand there is the case where all view logic is left in the view and the presenter doesn't get involved in deciding how to render the model. This style is the one implied by Potel. The direction behind Bower and McGlashan was what I'm calling Supervising Controller, where the view handles a good deal of the view logic that can be describe declaratively and the presenter then comes in to handle more complex cases.

You can also move all the way to having the presenter do all the manipulation of the widgets. This style, which I call Passive View isn't part of the original descriptions of MVP but got developed as people explored testability issues. I'm going to talk about that style later, but that style is one of the flavors of MVP.

Before I contrast MVP with what I've discussed before I should mention that both MVP papers here do this too - but not quite with the same interpretation I have. Potel implies that MVC

controllers were overall coordinators - which isn't how I see them. Dolphin talks a lot about issues in MVC, but by MVC they mean the VisualWorks Application Model design rather than classic MVC that I've described (I don't blame them for that - trying to get information on classic MVC isn't easy now let alone then.)

So now it's time for some contrasts:

- Forms and Controls: MVP has a model and the presenter is expected to manipulate this model with Observer Synchronization then updating the view. Although direct access to the widgets is allowed, this should be in addition to using the model not the first choice.
- MVC: MVP uses a Supervising Controller to manipulate the model. Widgets hand off user gestures to the Supervising Controller. Widgets aren't separated into views and controllers. You can think of presenters as being like controllers but without the initial handling of the user gesture. However it's also important to note that presenters are typically at the form level, rather than the widget level - this is perhaps an even bigger difference.
- Application Model: Views hand off events to the presenter as they do to the application model. However the view may update itself directly from the domain model, the presenter doesn't act as a Presentation Model. Furthermore the presenter is welcome to directly access widgets for behaviors that don't fit into the Observer Synchronization.

There are obvious similarities between MVP presenters and MVC controllers, and presenters are a loose form of MVC controller. As a result a lot of designs will follow the MVP style but use 'controller' as a synonym for presenter. There's a reasonable argument for using controller generally when we are talking about handling user input.

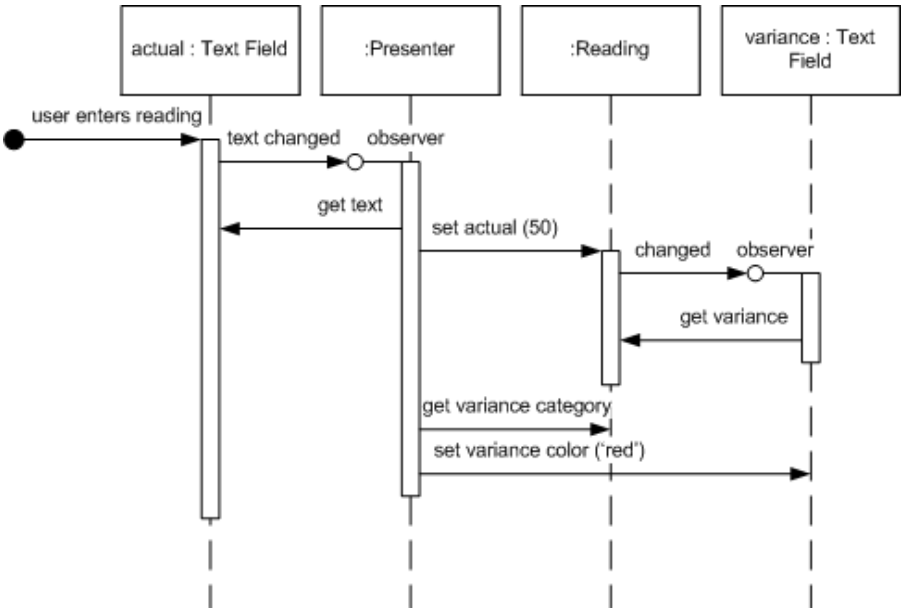

Figure 12: Sequence diagram of the actual reading update in MVP.

Let's look at an MVP (Supervising Controller) version of the ice-cream monitor (Figure 12). It starts much the same as the Forms and Controls version - the actual text field raises an event when its text is changed, the presenter listens to this event and gets the new value of the field. At this point the presenter updates the reading domain object, which the variance field observes and updates its text with. The last part is the setting of the color for the variance field, which is done by the presenter. It gets the category from the reading and then updates the color of the variance field.

Here are the MVP soundbites:

- User gestures are handed off by the widgets to a Supervising Controller.
- The presenter coordinates changes in a domain model.
- Different variants of MVP handle view updates differently. These vary from using Observer Synchronization to having the presenter doing all the updates with a lot of ground in-between.

Humble View

In the past few years there's been a strong fashion for writing self-testing code. Despite being the last person to ask about fashion sense, this is a movement that I'm thoroughly immersed in. Many of my colleagues are big fans of xUnit frameworks, automated regression tests, Test-Driven Development, Continuous Integration and similar buzzwords.

When people talk about self-testing code user-interfaces quickly raise their head as a problem. Many people find that testing GUIs to be somewhere between tough and impossible. This is largely because UIs are tightly coupled into the overall UI environment and difficult to tease apart and test in pieces.

Sometimes this test difficulty is over-stated. You can often get surprisingly far by creating widgets and manipulating them in test code. But there are occasions where this is impossible, you miss important interactions, there are threading issues, and the tests are too slow to run.

As a result there's been a steady movement to design UIs in such a way that minimizes the behavior in objects that are awkward to test. Michael Feathers crisply summed up this approach in The Humble Dialog Box. Gerard Meszaros generalized this notion to idea of a **Humble Object** - any object that is difficult to test should have minimal behavior. That way if we are unable to include it in our test suites we minimize the chances of an undetected failure.

The Humble Dialog Box paper uses a presenter, but in a much deeper way than the original MVP. Not just does the presenter decide how to react to user events, it also handles the population of data in the UI widgets themselves. As a result the widgets no longer have, nor need, visibility to the model; they form a Passive View, manipulated by the presenter.

This isn't the only way to make the UI humble. Another approach is to use Presentation Model, although then you do need a bit more behavior in the widgets, enough for the widgets to know how to map themselves to the Presentation Model.

The key to both approaches is that by testing the presenter or by testing the presentation model, you test most of the risk of the UI without having to touch the hard-to-test widgets.

With Presentation Model you do this by having all the actual decision making made by the Presentation Model. All user events and display logic is routed to the Presentation Model, so that all the widgets have to do is map themselves to properties of the Presentation Model. You can then test most of the behavior of the Presentation Model without any widgets being present - the only remaining risk lies in the widget mapping. Provided that this is simple you can live with not testing it. In this case the screen isn't quite as humble as with the Passive View approach, but the difference is small.

Since Passive View makes the widgets entirely humble, without even a mapping present, Passive View eliminates even the small risk present with Presentation Model. The cost however is that you need a Test Double to mimic the screen during your test runs - which is extra machinery you need to build.

A similar trade-off exists with Supervising Controller. Having the view do simple mappings introduces some risk but with the benefit (as with Presentation Model) of being able to specify simple mapping declaratively. Mappings will tend to be smaller for Supervising Controller than for Presentation Model as even complex updates will be determined by the Presentation Model and mapped, while a Supervising Controller will manipulate the widgets for complex cases without any mapping involved.

Further Reading

For recent articles that develop these ideas further, take a look at my bliki.

Acknowledgements

Vassili Bykov generously let me have a copy of Hobbes - his implementation of Smalltalk-80 version 2 (from the early 1980's)which runs in modern VisualWorks. This provided me with a live example of Model-View-Controller which was extremely helpful in answering detailed questions of how it worked and how it was used in the default image. Many people in those days considered it impractical to use a virtual machine. I wonder what our prior selves would have thought to see me running Smalltalk 80 in a virtual machine written in VisualWorks running in the VisualWorks virtual machine on Windows XP running in a VMware virtual machine running on Ubuntu.

Significant Revisions

18 Jul 06: First publication in development website.