

# My Speech: Interactive Software Paradigms for Statistical Visualisation

: D

February 16, 2007

## Slide 1 - Outline

Hi everyone, this talk is about what I've been working on pretty recently and it's about a number of software paradigms to help programmers build interactive graphics applications that are reusable, scalable and more maintainable, particularly to those who use OpenGL as their graphics library of choice. But, to give you a better appreciation of why these paradigms are useful I would spend the first part of the presentation looking at the basic idea behind them before getting into a more detailed discussion.

So here's the outline of the talk. (NEXT) I'll start talking briefly about some motivation for the research that shaped up this presentation. (NEXT) Then I'll describe a basic but very important idea underlying a number of software paradigms that I studied, using an example. Now because of time constraints I would only have time to give an overview about one of them, and provide a list of related paradigms at the end. And the one I'd talk about in this session is called (NEXT) Taligent's Model-View-Presenter. There's another widely accepted, simplified version used in Dolphin Smalltalk but I wouldn't get into it here.

## Slide 2 - Motivation

(NEXT) Now here's how I got started with the research. Two years ago I was working on a prototype of a statistical visualisation toolkit. The toolkit comes with an extensive documentation that will help programmers to build applications with plots in 3 dimensions or other complex scenes using the OpenGL library. It is also possible to put GUI controls on top or alongside the plot with the help of GUI toolkits like Gtk+. (NEXT) The toolkit

reached some degrees of success but I found that the applications I built weren't very scalable. It's just while the toolkit could let programmers build highly interactive plots, I could yet come up with a satisfactory way to define linking in a non-trivial fashion between graphical objects within a plot or between plots. Further I found that some components have to go through a lot of fiddling around to be reused in another application. Therefore I looked into Model-View-Controller and Model-View-Presenter to see which one of these paradigms will do the best job of solving my problems.

(NEXT) MVC, and more recently MVP have received a lot of hype in the software development world these days. Unfortunately, they're also amongst the most widely misinterpreted, because not only are they much more simply said than done, but most importantly, when you look on the web, there're actually two widely accepted versions for each of MVC and MVP, and quite often, various programmers will suggest their own interpretations while still claiming it MVC/MVP. So unless you stick to the original and read the other articles with a fairly critical mind you're quite likely to get lost.

The good news is, half a year ago Martin Fowler wrote a web article called GUI architectures. He does a good job of making people aware of the different versions of MVC's and MVP's. His review on the two versions of MVC's are particularly credible because he had experience in seeing how the original implementations work. However he hasn't delved deep enough into the MVP's to conclude which version is the better of the two. There were also a lot of doubts about the actual implementations of Taligent's MVP before I can make a judgment call. (NEXT) Therefore, I wrote a toy example using the paradigm and the experience I gained becomes the main theme of this talk.

## Slide 3 - Widget Based Paradigm

Before looking into Taligent's MVP, we'll look at how things used to be done. We'll see some problems associating with this approach and we'll see the basic ideas behind how these problems are tackled in MVC and MVP.

(NEXT) The application you see on the diagram is a mesh of a rosenbrock-bivariate normal mixture. There are various things that a user can customise, like the range of x and y axes, the resolution/smoothness of the mesh, and the function parameters where a dialogue will pop-up when I press this button. To implement all of these, (NEXT) I would first create the GUI's using a view

composer, (NEXT) and register all the possible events/signals for the widgets like clicking on a button or dragging the mouse on the drawing area. (NEXT) After that, I would get the view composer to generate a template of all the functions that handle the events or signals. (NEXT) Within each signal handler, I would basically code up everything in that function like updating the values in other parts of the GUI, perform the necessary computations and finally request the plot window to redraw itself. This method of first laying out the widgets and program according to them is known as the widget based paradigm.

## Slide 4 - Problems with Widget Based Paradigm

(NEXT) The paradigm results in code that is very easy to understand because you'd know exactly what's going to happen when you look at the code. However, it has its own drawbacks:

(NEXT) The main problem is that the user interface components that a user creates are not fine-grained enough. The UI components created are usually in form of a whole dialogue with all event handling of widgets stuck in one place. Now what if I want a new component that is slightly different in functionality later on? In that case I'd either have to copy and modify the old component, or risk breaking the code and factor out the common bits. And neither option is appealing.

(NEXT) Secondly, the backend is what actually does all the real work when the user interface is there mainly to make things easier for users. But now it's all too easy to fall into the trap of having the user interface, which is the less important part, to directly effect how you should implement the crux of the application.

(NEXT) Thirdly, the paradigm does not force you to be careful about separating your backend from your frontend, or what other papers refer to as domain logic and interface logic respectively. One example would be that you can find yourself using a widget as your data storage because the data structure is so simple that you couldn't be bothered making copies of it. The problem is that you risk breaking your code when your client suddenly wants you to change the look-and-feel of the UI.

(NEXT) This fourth point I'll illustrate in more detail on the next slide.

## Slide 5 - Problems with Widget Based Paradigm (ctn.)

(NEXT) Suppose I was asked to implement an object movement simulator, and let's assume for now that we're interested in patterns in cattle movement. The simulator would generate the change in locations every half a second, which will be plotted on a grid. Then we would have this object selector that allows us to change the colour of the selected points, as a way of tracking the positions of some cattle. Now suppose I was told that the application is an one-off so just forget all about proper program structures. Well, should be able to do it in a week, perhaps.

The question mark here shows you that life is never this simple, and two days later the user would tell me that there're serious overplotting issues on the grid and (NEXT) he wants to have a 3D histogram showing the density of the grid, and preferably a stacked histogram to show how the selected cattle are distributed. Well, that's not too bad, because the second window is just a "slave" of the first. The main window can just give the subwindow all the necessary data to render the histogram.

(NEXT) But wait. The user comes back the next day as he sees that one location consistently has a high density and he wants to see if it is from the same group of cattle or it just happens to be a good partying venue. Since you have an object selector in the main window wouldn't it be a trivial matter of adding another one to select the bars in the subwindow? Well, nope. It's just the second window now needs to actively request for the necessary data from the main window to handle the selection, while the main window has to know about the colour and grouping information of the subwindow to display the plot correctly. Or more simply put, they become tightly coupled. Now without any refactoring it can suddenly become a 2 week job because we have to take great care of all the dependencies between windows, especially the possibility of cyclic ones. (NEXT) Further, imagine how you would add or remove the dependencies when there are 4 windows depending on each other, and worse still, when you have to remove one of them later on. The result is seldom pretty.

## Slide 6 - The two Stage Solution

(NEXT) So what can we do? First we need to identify and separate the data objects out from the GUI's as shown in the diagram. Then we make the plot

windows observing the data objects. By observing we actually mean that, the plot windows will have direct access to the relevant data objects but the data objects would only know they've a number of unknown objects attaching to them. (NEXT) Now whenever the plot windows change the data, (NEXT) the modified data objects will notify all the interested windows, raising the flag "I've changed!" and not much else. Finally, upon notification, (NEXT) the plot windows will retrieve the required data and update themselves to reflect the changes, through the orange arrows.

And as we'll see, the Observer pattern is actually used quite heavily in the MVP framework because this allows us to define loose coupling between objects.

## **Slide 7 - Consequence: View as Observer to Data**

We actually gain a lot already just by doing what was proposed in the last slide. (NEXT) The code will certainly be more complex, (NEXT) but now the data objects do not need to know what is referring to them anymore, and it only requires a well-designed interface for other objects to gain access to it. (NEXT) So your data can now be reused in other applications. (NEXT) Then while plot windows seem to visually link to each other, they're completely unaware of the others. Therefore you can add or remove as many windows as you like without having to touch other components.

(NEXT) When you look at the name of the two paradigms, Model is actually the data and View refers to the visual representation of them. The third component, which it's either called Controller or Presenter, aims to take out the event interpretations and data manipulations part from the View and make the GUI as lightweight as possible, because they are usually very hard to test and debug. The paradigms have their responsibilities distributed somewhat differently, but the idea I just talked about actually form the basis of all 4 versions of the MVC/MVP paradigms.

## **Slide 8 - Taligent's MVP**

(NEXT) We now turn our focus onto the Taligent's MVP framework. You might expect that all versions of MVC/MVP will have three components like this but Taligent's MVP actually has (NEXT) six. (BACK) The Controller

component in the classic MVC used to receive all the events, interpret them and make changes to the Model. (NEXT) In this version of MVP they split the Controller into three extra levels of abstractions, the Interactor, Command and Selection. Then the remaining part of the Controller is renamed to Presenter.

And here are the basic responsibilities of the individual components in more detail. I have some slight modifications to the framework for my own use but I'll put up the original version first before stating the modifications.

As I mentioned before, View would represent the data that are stored in the Domain model. In some cases, the data could be stored in other softwares such as a database, and the Domain Model can act as a mediator between the View's and these other softwares as well.

Selection supposedly defines a simple collection of the selected data, and provide an implementation to show them on the screen. In my implementation however, the definition and the rendering of the selection was moved to Interactor and View respectively, while the Selection component gets reduced to a storage for the selected data. This is due to the idiosyncrasies of OpenGL in regards to object selection, and the fact that this configuration actually allows the Selection component to be shared between triads, which is required for plot linking at times.

In the current implementation, the Command component receives command objects created by the Interactor, each of which encapsulates an algorithm to be applied to the data defined by the Selection component. The component also has a Command history so the opeartions can be undone/redone.

Interactor is responsible for creating the command objects and pass it down to the Command component, although the exact timing of this creation process in the original version remained unclear. It is also the place to receive all the user gestures and events, and map them to changes in the data. The original paper actually mentioned that Interactor will first delegate these user actions to the Presenter, where it will process them further before issuing the appropriate commands. But then I had to leave this step when I was implementing the example because most of the effort was put on clarifying the structure of the Command component.

The final responsibility of the Interactor is to activate or deactivate parts of the GUI in response to changes in Selections.

Finally, on top of possibly interpreting the user gestures further and invoking commands, Presenter is also the creator of all the other components, and therefore has access to every single one of them.

Then here are what the links in the diagram indicate. In what follows, the green arrows indicate the Observer-Subject relationship which I mentioned before. Violet indicates direct access and therefore the source will have reference to the target component. The brown ones are links that exist historically in MVC but isn't documented in the MVP paper. Finally the dashed lines are optional links. And as you would expect, the Data component here is also optional.

(NEXT) An important point to note also is that while the “triads” can be applied on a per window basis, more proper implementations actually apply the construct to data structures as simple as strings or vectors of integers and use them to build more complex components. That's how the additional granularity is attained in these implementations.

## **Slide 9 - Taligent's MVP (Modified for OpenGL)**

(NEXT) The modified version is shown on this diagram. We have two additional arrows, one from Interactor to Domain Model and one from View to Selection. I also employ the Interactor-View link. The reason of their existence will be explained in the conference paper but I have to skip over them and just state their uses in the control flow of MVP instead. The only thing is, the Interactor-Domain Model link, in saturated red, is the one I feel the least comfortable with and I had extensive documentation in my example code to limit its usage.

## **Slide 10 - Control Flow (Modified Version)**

(NEXT) Now, we get onto the things that hasn't been talked about a lot and that's the control flow of the paradigm. There're a number of seemingly viable implementations for some of the steps but I only have time to present the one I did use in my toy example.

I'll start doing a little demo of the toy example before using it to demonstrate some of the steps in the control flow. This toy example is just a simple

colour selector. The circle is a HSV colour wheel, and you can customise the number of colours to be displayed on the wheel, using this spinbutton. Then you can customise the range of values for hue, saturation and value given the other two fixed. So at this point, the hue starts from 0 with 360 as the total increment while saturation and value are kept fixed at 0.5. And I can do various things to change the appearance of the wheel so I can obtain different sets of colours. Now I can click Ctrl left button to select the colours, and when I'm done I can either save it or click this button to display my selection on the subwindow. In this subwindow, I can change the colours one at a time using this colour selection dialogue and the change will be reflected in the main window. This window is actually pretty useless, but I'm doing it so that I can demonstrate linking like you'd do in statistical plots.

I'll describe the control flow of all the different steps in the conference paper, but I only have time to talk about two of the more significant ones.

*Since I'm not talking anymore here's the control flow of all of them. Corresponding slides at the end*

## Triad Creation

As the caption suggests, this is when all the components are created, and it's the time when you see the widgets appearing on the screen.

The individual components except the extra data are supposedly created by the Presenter, but for pluggability issues, I also had a version where I would create all the components outside and wire them all up in the Presenter. The extra data and the connection of it would in any case be done by the Domain Model.

## Rendering the Data

After you see the window, the process of rendering or representing the data is carried out whenever you see the appearance or values of a widget changing, and not restricted to the colour wheel in the drawing area. And the mechanism is just what I talked about in the first half of the presentation.

## Update Available Commands

I haven't implemented this feature in my toy example yet, but one example you can think of is the activation or deactivation of the menu items after



you selected some words in a word processor. This activation or deactivation is determined by the set of available commands, and in turn determined by the state of the Selection component. The method I'm presenting here is actually inspired by Joanna Carter's implementation in her tutorial series. The Command component would contain a list of strings of all possible commands. Now when a selection is made, the Selection component will notify the Command component, and the component will respond by splitting the list of strings into two sets - available and not available. Interactor will then be notified by the Command and this is the place where the activation and deactivation take place.

## **Selecting Observations and Rendering Selected Obs**

(NEXT) When I click on the pieces on the colour wheel, I was actually selecting some data and the colour pieces will have white line loops around them. In my implementation, the Interactor would interpret the mouse click and decide that I'm selecting some data. The detection of the selected objects would then be carried out through the View using the selection buffer in OpenGL, and when we know what the objects are, we would update the indices of the colour array in the Selection. The View would be notified of the change and it would respond by rendering the screen again with the selected objects highlighted.

## **User Gestures/Events Interpretation and Command Execution**

(NEXT) Next one is to execute the commands when the users perform some actions other than selections. Towards the end of the demonstration I changed the colour of a square and this would be one of the situations when this is happening. Under the current implementation, the Interactor will receive the signal, grab the current colour of the colour selection dialogue and create the "set-colour" command objects, using the colour and these two links. The Command component will then accept the created object and modify the data through link 2, 3 and 4.

Now that the data gets modified, the interested View's would update the changes, and that's when we see the colour on the two windows changing simultaneously.

## Slide 11 - Conclusion

(NEXT) To conclude, benefits of Model-View separation and having the View observing the data are obvious. (NEXT) But at this point we still cannot conclude on the best suited framework amongst the various versions of MVC's/MVP's for this particular toolkit.

(NEXT) The finer separation of Taligent's MVP looks good on paper and Mike Potel clearly addressed all the benefits of the extra separation in his paper. However some further studies are required to assess its effectiveness in practice, as well as the effect of the additional links in the modified version.

(NEXT) One major drawback of the framework though comes in its rather steep learning curve and proper documentations and assisting tools will be necessary.

There're also a number of mysteries remaining but I cannot cover them all today, and they'll be addressed in the conference paper as well.

As promised, here's a list of related paradigms, roughly in their chronological order, which ends my talk. Thank you for your patience, as it cures my insomnia every time I read it at heart.

## Slide 12 - Quote of the day

This last one is to show you how Martin Fowler started his research in MVC's...

## Slide 13 - Mysteries of Taligent's MVP

There're actually a number of mysteries behind the paradigm that go unaddressed. But I guess I've no time for that and I'll skip over them. Instead I'd present the modified version of the paradigm and talk briefly about the control flow.