TWISTING THE TRIAD

The evolution of the Dolphin Smalltalk MVP application framework.

Tutorial Paper for ESUG 2000

Andy Bower, Blair McGlashan
Object Arts Ltd.

Model View Presenter (MVP) is a modern user interface framework for Smalltalk. Derived from the Taligent C++ system of the same name. MVP is currently the key UI framework in Dolphin Smalltalk. This paper discusses the qualities of MVP and why we (the Dolphin design team) chose to adopt it over and above two previous framework designs (that were tried, and yet rejected) based around "widgets" and Model View Controller (MVC). We will attempt show how, by rotating (or twisting) the MVC triad, one can produce an "Observer" based framework that is easy to use and more flexible than those currently available in other Smalltalk environments.

Introduction

Back in the summer of 1995, we were starting to turn our minds to the creation of a suitable User Interface model for our new implementation of the Smalltalk language; Dolphin Smalltalk. Our fledgling VM was working well, the compiler was (for the most part) emitting correctly optimised bytecodes and most of the Smalltalk-80 base classes had by now been implemented as part of the Dolphin boot image. The next essential stage before we could start building "real" applications was to supply a Windows-based GUI framework. This paper discusses the evolutionary stages that we went through before arriving at our current offering, the Model-View-Presenter framework.

Widgets

Since our original remit when designing Dolphin was to provide a truly object-oriented replacement for client side programming environments such as Visual Basic (and, indeed, another client side 4GL called IS/2) it seemed appropriate to start off with a framework similar to the ones used by these systems. Both Visual Basic and IS/2 use an interface paradigm that we have tended to describe as *Widget-Based*. By this, we mean that the normal method of system building consisted of "drawing" various application screens by laying out user interface components within windows and dialog boxes and then attaching pieces of code to these elements in order to build up the logic of the application. At that time Visual Basic didn't allow for a component approach, i.e. being able to build new widgets and reusing them in future designs. However, it was always our intention that Dolphin should allow the hierarchical composition of components that could themselves be re-used if required.

Hence, we set about building a widget-based framework for Dolphin Smalltalk. The process took about three months and it was only towards the end of this period that we became highly dissatisfied with the results. Yes, with a widget-based system it is easy to avoid having to think about the (required) separation between the user interface and the application domain objects, but it is all too easy to allow one's domain code to become inextricably linked with the general interface logic. However, since this was Smalltalk, it was also a simple matter to refactor these sorts of issues. No, that was not the problem; it was much more that the widget system was just not flexible enough. We didn't know at the time, but were just starting to realise, that Smalltalk thrives on plugability and the user interface components in our widget framework were just not fine-grained enough.

One example of this deficiency surfaced in our *SmalltalkWorkspace* widget. This was originally designed as a multiline text-editing component with additional logic to handle user interface commands such as Do-it, Show-it, Inspect-it etc. The view itself was a standard Windows text control and we just attached code to it to handle the workspace functionality. However, we soon discovered that we also

wanted to have a rich text workspace widget too. Typically the implementation of this would have required the duplication of the workspace logic from the

SmalltalkWorkspace component or, at least, an unwarranted refactoring session. It seemed to us that the widget framework could well do with some refactoring itself!

Model-View-Controller

Interestingly, until that time, we had little idea what Model-View-Controller (MVC) was all about. Of course, we knew it was the fundamental UI framework in Smalltalk–80, but we didn't quite know how one would go about building an application using it. This was before the days of Squeak and VisualWorks Non-Commercial so our research has to be conducted using a rather expensive copy of VisualWorks 2.5. This soon led us to believe that MVC was the way to go.

So then, in late 1995, we began ripping the widget framework out of Dolphin and replacing it with our own implementation of MVC. It is of course pretty tricky to replace one UI framework with another when the code browsers being used are themselves implemented in the framework being replaced. After two months of this level of excitement the job was almost complete but, on stepping back, we were still not completely satisfied with the results.

In MVC, it is a view's responsibility to display the data held by a model object. A controller can be used to determine how low-level user gestures are translated into actions on the model. The various components, M, V and C are all pluggable for maximum flexibility. Generally, a view and controller are directly linked together (usually by an instance variable pointing from one to the other) but a view/controller pair is only indirectly linked to the model. By "indirect", I mean that an Observer relationship[1] is set up so that the view/controller pair knows about the existence of the model but not vice versa. The advantage of this comes because it is then possible to connect multiple views/controller pairs to the same model so that several user interfaces can share the same data. Unfortunately, it is the nature of this indirect link that causes the problems with MVC.

In MVC, most of the application functionality must be built into a model class known as an Application Model[2] (see figure 1). It is the responsibility of the application model to be the mediator between the true domain objects and the views and their controllers. The views, of course, are responsible for displaying the domain data while the controllers handle the raw user gestures that will eventually perform actions on this data. So the application model typically has methods to perform menu command actions, push buttons actions and general validation on the data that it manages. Nearly all of the application logic will reside in the application model classes. However, because the application model's role is that of a go-between, it is at times necessary for it to gain access to the user interface directly but, because of the Observer relationship between it and the view/controller, this sort of access is discouraged.

---

[1] Smalltalk these days has two standard schemes for implementing an Observer relationship. The original, Smalltalk-80 approach was to use a mechanism known as "Dependency". More recently, an additional "Event" mechanism has been introduced into most class libraries. The latter has a number of performance and aesthetic advantages over the former and so is the mechanism most commonly used for Observer in Dolphin Smalltalk. However, there are pros and cons to each approach.

[2] These are the VisualWorks terms and we consider their implementation here since it is generally acknowledged that the VisualWorks implementation of MVC contains a number of improvements on the original in Smalltalk-80.

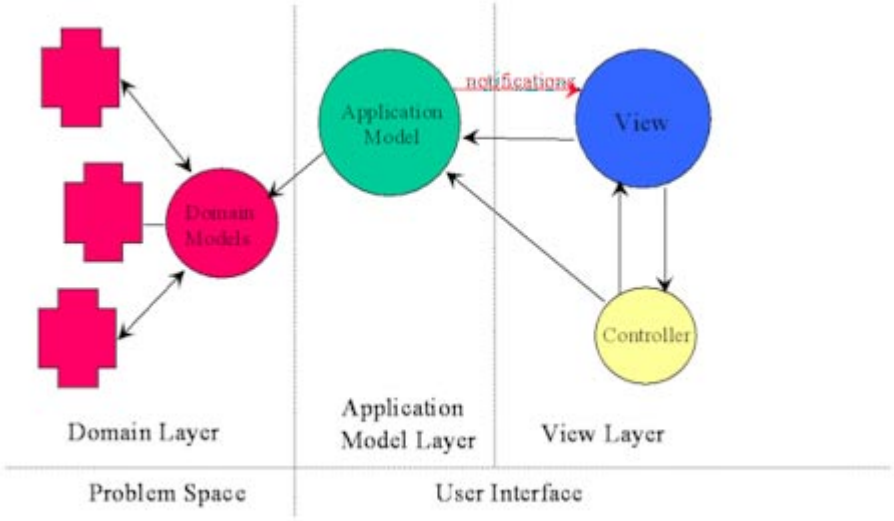

Figure 1: A typical MVC triad

For example, let's say one wants to explicitly change the colour of one or more views dependent on some conditions in the application model. The correct way to do this in MVC would be to trigger some sort of event, passing the colour along with it. Behaviour would then have to be coded in the view to "hang off" this event and to apply the colour change whenever the event was triggered. This is a rather circuitous route to achieving this simple functionality[3] and typically it would be avoided by taking a shortcut and using *#componentAt:* to look up a particular named view from the application model and to apply the colour change to the view directly. However, any direct access of a view like this breaks the MVC dictum that the model should know nothing about the views to which it is connected. If nothing else, this sort of activity surely breaks the possibility of allowing multiple views onto a model, which must be the reason behind using the Observer pattern in MVC in the first place.

Another irritating feature of MVC, at least with respect to Dolphin, was that the idea of a controller did not fit neatly into the Windows environment. Microsoft Windows, like most modern graphical operating systems, provides a set of native widgets from which user interfaces can be constructed. These "windows" already include most of the controller functionality embodied as part of the underlying operating system control. We found that, in order to create a sensible controller hierarchy, it was necessary to "break out" this inherent functionality and route it to various *Controller* subclasses. Even having done so, the ability to plug and play with these controller classes was severely limited by what the Windows OS would actually allow. Eventually, we decided that this approach was not appropriate so we stripped these controller classes away leaving us with a mainly vestigial *Controller* hierarchy.

Twisting the triad: Model-View-Presenter

By now we were once more becoming somewhat disheartened with our framework approach. We were left holding two possibilities that, although adequate, were flawed in several respects (and, when one is engaged on a quest to create the *perfect* development environment, such flaws seem all the more debilitating).

So, where were we? We liked the Observer aspect of MVC and the flexibility that came from its pluggable nature but it just didn't seem correct that the link between the application model and view was an indirect one. Also, the requirement for "controllers" in the Windows environment seemed out-moded.

---

[3] There are other reasons for disliking this approach. In VisualWorks MVC a composite view is built as a Canvas and installed in a *#windowSpec* method on the application model. In order to add the behaviour required by this example it would be necessary to create a new view class that would not otherwise be necessary.

To be honest, we didn't feel confident about tinkering with these aspects of MVC purely as the result of our own intuition; we'd wasted enough time already. We were on the point of giving up and resigning ourselves to using our existing interpretation of MVC when a colleague[4] asked if we had looked at Taligent's Model View Presenter framework? Of course we hadn't, so we tracked down a couple of books and a paper on the Web. The latter (with all the original references to "C++" globally replaced with "Java") can now be found at http://www.ibm.com/java/education/mvp.html and makes interesting reading.

On reading this paper, we were intrigued at how the Taligent people had appeared to uncover the same weakness in MVC that we had, and how they'd solved the problem by rotating (or as we like to say, "twisting") the triad through 60°. So what, then, are the components of a MVP triad?

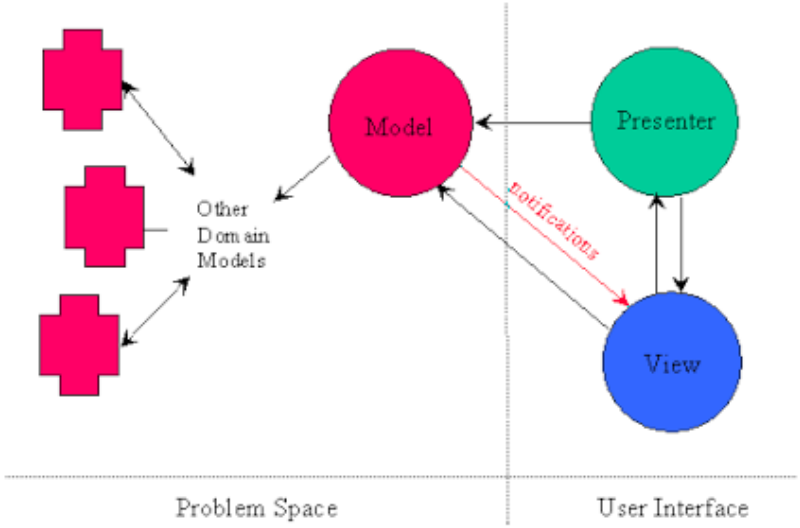

Figure 2: The MVP Triad

The Model

This is the data upon which the user interface will operate. It is typically a domain object and the intention is that such objects should have no knowledge of the user interface. Here the M in MVP differs from the M in MVC. As mentioned above, the latter is actually an Application Model, which holds onto aspects of the domain data but also implements the user interface to manipulate it. In MVP, the model is purely a domain object and there is no expectation of (or link to) the user interface at all.

The View

The behaviour of a view in MVP is much the same as in MVC. It is the view's responsibility to display the contents of a model. The model is expected to trigger appropriate change notifications whenever its data is modified and these allow the view to "hang off" the model following the standard Observer pattern. In the same way as MVC does, this allows multiple views to be connected to a single model.

One significant difference in MVP is the removal of the controller. Instead, the view is expected to handle the raw user interface events generated by the operating system (in Windows these come in as WM_xxxx messages) and this way of working fits more naturally into the style of most modern operating systems. In some cases, such as a *TextView*, the user input is handled directly by the view and used to make changes to the model data. However, in most cases the user input events are

[4] This was Doug Simmonds who was, at the time, working for Intuitive Systems Ltd

actually routed via the presenter and it is this which becomes responsible for how the model gets changed [5].

The Presenter

While it is the view's responsibility to display model data it is the presenter that governs how the model can be manipulated and changed by the user interface. This is where the heart of an application's behaviour resides. In many ways, a MVP presenter is equivalent to the application model in MVC; most of the code dealing with how a user interface works is built into a presenter class. The main difference is that a presenter is *directly* linked to its associated view so that the two can closely collaborate in their roles of supplying the user interface for a particular model.

Benefits of MVP

So the effect of this "twist" is that the presenter, where the data manipulation part of a user interface is handled, is also allowed direct access to the view, where the data display is implemented. This can be very handy at times and is one of the most obvious benefits over MVC where the application model only has an indirect link to its associated view. The Dolphin implementation of MVP also manages to dispense with the idea of a controller, which seems to make the framework "fit" better with the underlying Windows operating system.

Compared with our original widget framework, MVP offers a much greater separation between the visual presentation of an interface and the code required to implement the interface functionality. The latter resides in one or more presenter classes that are coded as normal using a standard class browser. The window layouts for most applications are created using a tool known as the View Composer which is used to create an *instance* of the view required[6]. These view instances are held in an internal binary form by a Resource Manager. Normally, one or more view instances can be associated with any presenter class and a presenter can specify which particular view is required when it is launched. Hence it is easy for an MVP application to have one or more "skins" that can be selected as required. For example, the Dolphin development environment has three versions of the standard Class Hierarchy Browser which are all driven by the same *ClassBrowserShell* presenter class. We have the standard browser view, a simplified version for beginners (offering fewer options) and an alternative version where the class hierarchy is represented as a diagram rather than a standard tree view. This level of flexibility would not be possible with a framework based solely on widgets.

Future possibilities

Because of the clean separation between data (the model), the display of this data (the view) and the handling of updates to this data (the presenter), the MVP framework lends itself quite readily to a number of future enhancements.

Schematic diagrams

When we started out with Dolphin we had always been quite taken with the "wiring metaphor" behind VisualSmalltalk's PARTS and VisualAge's Composition Editor. However, both of these

---

[5] The original Taligent specification for MVP included an additional pluggable object called an Interactor. This had much the same responsibility as an MVC Controller. An Interactor takes user interface gestures and translates these into appropriate manipulation messages to the Presenter. In Dolphin, we chose (for the sake of simplicity) not to build Interactors into the basic framework. It is however possible to use such interactors on an ad hoc basis when necessary and the *MouseTracker* class is an example of such a device.

[6] Some other Smalltalk frameworks (e.g. WindowBuilder) tend to emit view descriptions as new classes. While the representation of a view as code has some benefits (one being the ability to check such views into a standard change control system) we have chosen not to adopt this scheme within MVP. Our reasoning has always been that a typical composite view offers no new "behaviour" and therefore should not be represented by a new class but, rather, as an aggregation of view instances.

systems suffer from the same problem in that the wiring, which is effectively part of the interface functionality, is mixed up with the visual presentation. This almost invariably seems to lead to a spaghetti wiring situation that most people find hard to read and to maintain.

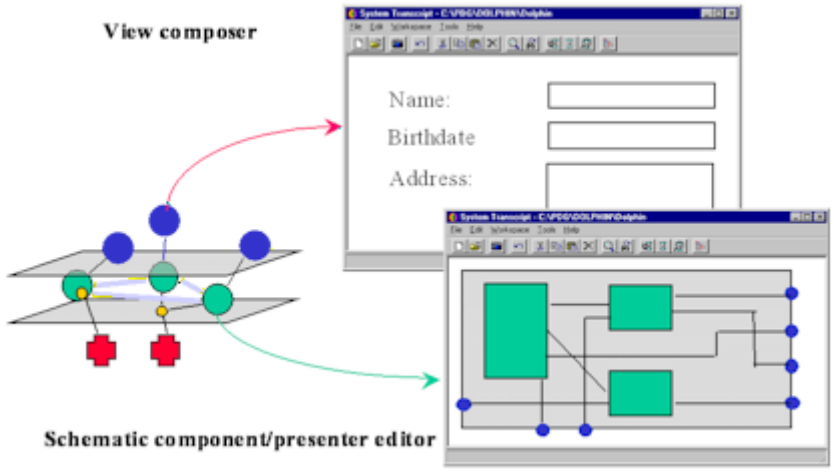

Figure 3: A Schematic Editor

In retrospect it seems obvious that the logic of an application, which is described by the wiring, should be kept wholly separate from the window layout. The two have completely different forces acting on them; the wiring is intended to be viewed by the programmer whereas the audience for the window layout is the end-user[7]. Hence, it was our original intention with MVP to have a View Composer for laying out the visual aspect of an application but to supplement this with a Schematic Composer which would be able to lay out the application logic as a schematic design. Such a Schematic Composer would automatically generate methods within the appropriate presenter classes to implement the schematic wiring's behaviour. You can still see the hooks for this in some of Dolphin's MVP methods such as *Presenter>>createSchematicWiring*.

Portable MVP

One of the most common criticisms that comes (mainly) from the Java camp is that it is not straightforward to take a complete application from one vendor's Smalltalk and move it with little change to another's. In fact, domain code will usually transfer with little change (and the efforts at the recent Camp Smalltalk's have gone a long way towards improving this situation still further). What is still quite apparent is that there is currently no portable GUI framework that can be used to move user interfaces between the different Smalltalk variants.

A possible solution to this might be to develop a "Portable MVP" framework for each of the popular Smalltalk environments. The Model and Presenter class hierarchies and the other sub-frameworks[8] that are part of Dolphin MVP should port with relative ease since they don't tend to make assumptions about the underlying operating system or user interface. Similarly, since most of the code for an application's user interface is written on the presenter side, the applications themselves should also be eminently portable.

The main development outlay with such a scheme would be to create a View class hierarchy for each of the target Smalltalks. This is perhaps not as onerous as it sounds since it would not be necessary from the outset to build a collection of views as sophisticated as those provided by Dolphin's native

[7] For example, an electrician will never attempt to repair a printed circuit board without reference to a schematic diagram. The board layout is constrained by the physical dimensions of the components and will, very likely, be unintelligible to the engineer. A schematic, however, is specifically layout to be easy to read at a functional level.

[8] MVP includes a number of additional sub-frameworks such as Command Routing, Drag and Drop and Resource Management.

widget set. A good deal of mileage could be gained from a straightforward subset that was at least sufficient to get simple development tools (such as those used by Camp Smalltalk) on-screen in a portable way. For example, at CS1[9], a concerted group of Smalltalkers was able to port the domain code for the Refactoring Browser over to most of the available Smalltalk dialects in a couple of days. However, some months later, the user interface remains an outstanding issue for most of these systems.

Conclusion

Over the last few years our experience with Model-View-Presenter has been enough to reaffirm the original decision to run with this new framework. The flexibility it provides over that of a simple widget based approach such as that of Visual Basic or Java's AWT cannot be disputed. Although it is ostensibly very similar to standard MVC, the rotation of the triad into the MVP format gives a more regular result that improves the consistency and "feel" of programming in a number of areas.

Having said this, MVP is still a major source of confusion for Dolphin newcomers. This may be due to a lack of documentation or, perhaps more likely, the added complexity that a pluggable framework exhibits, especially when compared to the widget approach that most non-Smalltalk programmers are familiar with. The majority of people who "stick with it", however, come to recognise the advantages it provides, especially in the ease of maintenance of MVP applications and in enabling re-use. Nevertheless, it remains our aim to further reduce the barriers to MVP's acceptance by implementing additional tools (or Wizards) that will, hopefully, ease the initial programming burden when creating MVP applications in future.

[9] Camp Smalltalk 1, San Diego, March 2000