Model GUI Mediator

Abstract

A MGM object displays a portion of a software application's data to the user, and optionally allows that data to be altered by the user of the application. MGM is specifically suited to the display of application data in contemporary, off-the-shelf GUI control widgets within a modern RAD (Rapid Application Development) IDE. The MGM design pattern empowers a programmer to simulate "object-aware" controls using off-the-shelf GUI controls.

Intent

To manage the display and user manipulation of model data using off-the-shelf GUI graphic controls / widgets. Like Model-View-Controller (MVC), the MGM solution keeps graphical knowledge out of the model, decoupling the model data and GUI.

Overview

Consider the following scenario:

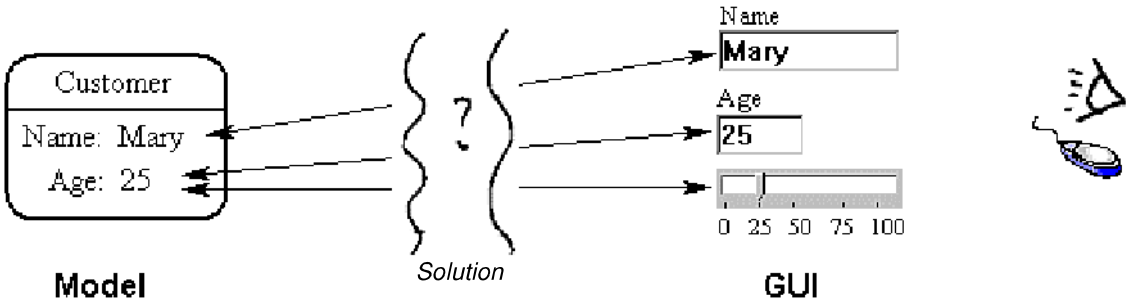


Figure 1. You can change Mary's age by using either the text box or the slider control.

If you find yourself in	{context}	You want to display and possibly modify an attribute of an object using an off-the-shelf GUI control within a modern IDE
for example	{example}	A customer object's age attribute in a GUI text box, also the same age attribute displayed as a GUI slider control
with	{problem}	You want to connect a piece of object data to a particular GUI control
entailing	{forces}	**1)** You want to de-couple model from GUI controls **2)** Each model object might have a corresponding object on the screen **3)** You want to allow multiple presentations of the same model **4)** Changes to the model need to propagate to all presentations **5)** Changes to any presentation changes the model **6)** Users should feel they are directly manipulating model objects & their attributes **7)** You want to use off-the-shelf GUI widgets for presentation **8)** You want to avoid sub-classing or creating special versions of off-the-shelf GUI controls because this is a chore which may even involve registering new widget controls onto an IDE control widget palette.

		9) You want to take advantage of the natural GUI event notification framework available in your programming environment / language / IDE, e.g. like easily filling in the body code for various GUI event procedures that your IDE creates for you. **10)** You want to do something 'MVC-like' using off-the-shelf GUI controls **11)** The traditional MVC framework is unsuitable when implementation involves off-the-shelf GUI controls **12)** The model & GUI need to be encapsulated & loosely coupled, with little or no knowledge of each other **13)** the model & GUI need to be intimately connected in order that the user be able to see the model data and manipulate it
then apply	{design}	Use the Mediator and Observer design patterns
to construct	{solution}	Define a 'mediating view' class that mediates all communication between a GUI control widget and the model. The mediating view takes on the role of 'observer' of particular business objects in the model. In a sense, off-the-shelf GUI widgets are 'adapted' (adapter pattern) to know about the model. This adapter object is known as the 'mediating view'. This MVC-like solution decouples GUI controls and model / business objects by establishing a subscribe-notify protocol between them (via the 'mediating view' class). A further notification mechanism is established (from GUI to mediating view) using the usual event handling mechanism for GUI controls that exists within the language / IDE (Integrated Development Environment).
leading to	{new context}	Off-the-shelf GUI widgets are 'adapted' to know about the model, and become 'object-aware'.

Discussion

The MGM (Model-GUI-Mediator) pattern balances the basic forces of the 'model to GUI' mapping problem in a different priority than MVC (Model-View-Controller), because it introduces a powerful new force – that of being relevant to a modern programmer who wants to use off-the-shelf GUI controls within a modern RAD (Rapid Application Development) IDE. Delphi, Visual C++, JBuilder, C++ Builder, VB etc. are examples of such environments. Such environments have a built-in event handling framework for such GUI controls, where the programmer can easily add code to respond to various events e.g. the OnTextChanged event. Unlike MVC, MGM works well in this context

The MGM pattern encourages designing and programming in terms of a proper object model. Unfortunately, the convenience of off-the-shelf 'data-aware' controls is often compelling due to the easy, instant mapping of GUI controls to relational database data attributes and the subsequent automatic synchronization between those GUI controls and database data. This situation encourages programmers to adopt relational database technology to model their application. This is often overkill and introduces a dependency on relational database engines. This in turn increases application size, makes application installation more complex. Most of all, it discourages

programmers from designing and coding their application in terms of a pure object model.

The MGM pattern on the other hand, encourages the use of an object model instead of a relational database model, since some of the convenience of 'object-aware' GUI controls can now be simulated (though you will still need to implement your own persistence functionality).

Also Known As

Model-View-Graphic (MVG) [Jezequel, Train, Mingins 2000, "Design Patterns and Contracts"]
Model-View-Presenter (MVP) [Joele Coutaz, Andy Bower (Dolphin Smalltalk)]

Solution

The Model GUI Mediator pattern is actually 2 classic patterns overlaid on each other: Mediator and Observer. A mediating 'view' class mediates all communication between the GUI component and model. The model communicates with the mediating view(s) using observer.

MGM uses the following terms:

- **Model** is a business object or collection / composite of application business objects – which you want to visualize.
- **Mediating View** is the heart of the pattern. It is an observer of business objects, mediating between model and GUI, and between GUI and model. It knows how to display the model/business object, talking to a particular GUI control (or in more complex cases, drawing onto a rectangular region) in order to achieve the presentation effect. Conversely, the GUI graphic control widget talks to the mediating view object in order to make changes to the model.
- **GUI** (Graphical User Interface) is a GUI control widget e.g. a text edit control, combo box, slider, treeview, form etc. Programmer code triggered by the IDE's normal GUI event handling mechanism is used to both respond to user input and then talk to the mediating view in order to affect changes to the model.

Note: The meaning of 'view' varies slightly depending on the author and pattern. Is a 'view' the actual display widget or a class that talks to a display widget? A MVC view is a class that renders onto a GUI widget and often performs roles that are routinely handled by a modern GUI widget. A MVP view is a GUI. A Document-View 'view' is usually a GUI widget or form. A MVG view is the same as a MGM 'mediating view', which is equivalent to the MVP 'presenter'.

The interaction of the observer pattern notifications, the GUI control event notifications and the view object's mediation in the MGM pattern are illustrated thus:

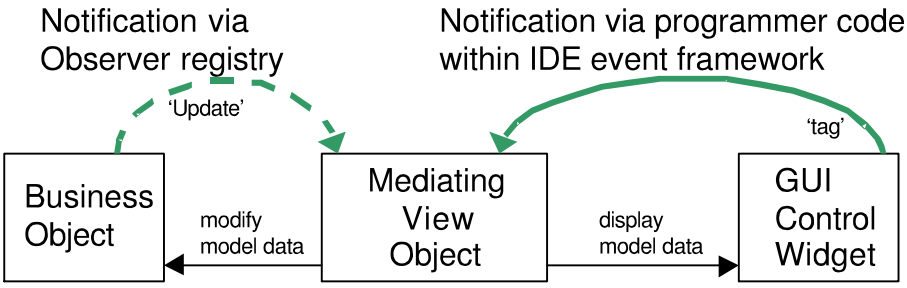


Fig. 2. How the MGM view class mediates.

How to implement MGM

In a nutshell, assuming your model already implements observable, (1) derive a 'mediating view'

class from observer, code its update method to display a business object attribute in a GUI control. (2) Subscribe the mediating view object to a business object, point the mediating view object to a GUI control and point the GUI control to the mediating view object. (3) Using your IDE framework add GUI event code that responds to events and modifies the business object via its mediating view object. (4) Ensure any relevant business object attribute changes trigger a NotifyOfUpdate broadcast (observer pattern). Detailed steps are listed below:

Here is an example implementation of the MGM pattern:

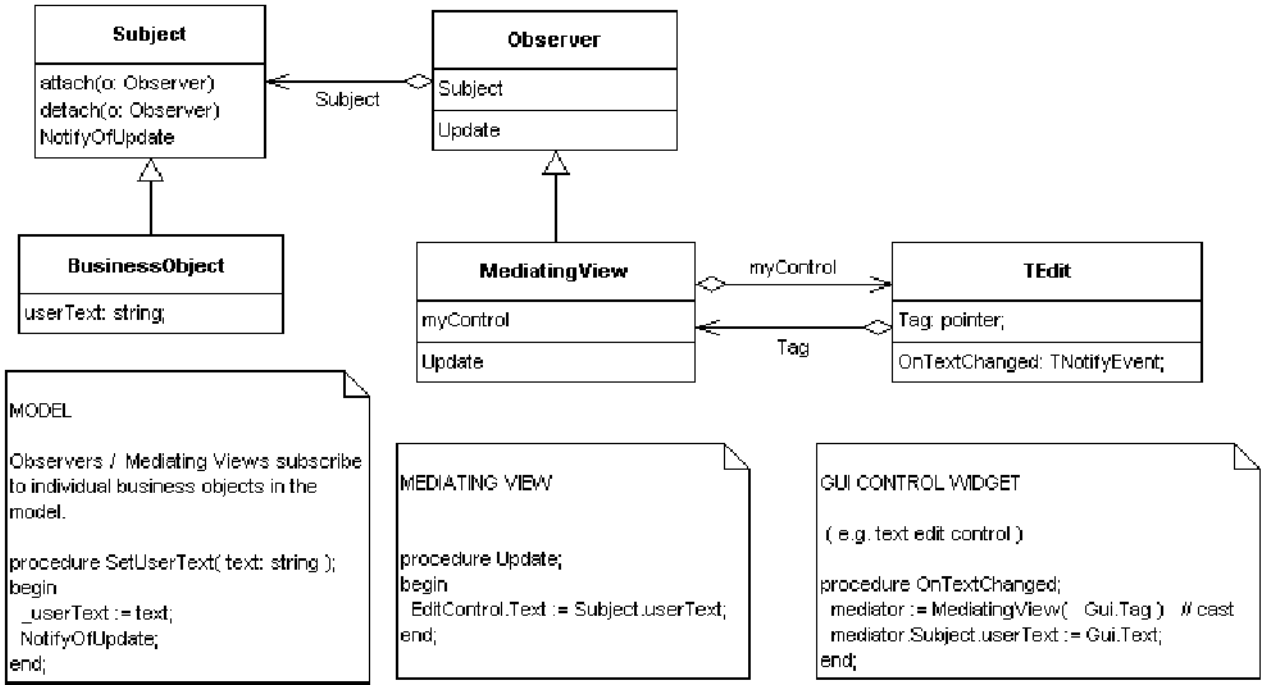


Fig. 3. MGM pattern for use with off-the-shelf GUI controls.
Comments boxes contains fragments of Delphi Object Pascal code

Detailed Steps

1. MODEL: Ensure your model implements observable

1.1. Have all business objects inherit from a base business object class that itself inherits from Observer (also known as Subject). Alternatively, have your base business object class support the Observer interface. The required methods are `attach`, `detach` and `NotifyOfUpdate`.

1.2. At runtime, you will typically be associating a mediating view object with a particular business object in your model (via `attach`/`detach` Observer). E.g.

```
Customer23.attach( Customer_NAME_TextEdit_MediatingView1 );
Customer27.attach( Customer_AGE_TextEdit_MediatingView2 );
Customer27.attach( Customer_AGE_Slider_MediatingView3 );
```

Delphi Object Pascal code.

1.3. When a business object attribute changes, e.g. `Age` or `UserText` the business object usually will need to send a `NotifyOfUpdate` message to inform all interested observers (including view mediators). E.g. The following 'setter' method (coded in Delphi / Object Pascal) is executed every time a customer object's age is changed:

```
procedure Customer.SetAge( newAge: integer );
begin
  _age := newAge; // Set the private field representing the customer's age.
  NotifyOfUpdate; // Loop through attached observers & call Update on each.
end;
```

Delphi Object Pascal code.

2. MEDIATOR: Define a base 'mediating view' class that implements Observer – providing a virtual method called '`Update`'.

2.1. Sub-class to create a new mediating view class for each model attribute-GUI control mapping. For example, to display a customer object's address in a user editable text edit control, create a *Customer_AGE_TextEdit_MediatingView* class.

2.2. Define a pointer from each mediating view to a specific GUI control class. Thus the 'mediating view' class ends up having two pointers: a pointer to the business object (e.g. a property called '`subject`' via observer pattern) and also a pointer '`myControl`' to a GUI control (e.g. a pointer to a TEdit text control).

2.3. Have each mediating view class override the '`Update`' method and add GUI specific code that updates the relevant GUI control's display with the current value of the relevant business object attribute. In the following example, `EditControl` is the mediating view's pointer to its associated GUI control, and `userText` is the business object attribute being displayed. Note also that `EditControl.Text` is a reference to the GUI control's built in '`text`' property, which represents what is physically displayed to the user. E.g.

```
procedure Update;
begin
  EditControl.Text := Subject.userText;
end;
```

Delphi Object Pascal code.

2.4. At runtime, have the mediating view object observe a particular business object/model (via `attach/detach` Observer). *Same as step 1.2, above.*

Fig 4. Typical off-the-shelf GUI graphic controls

3. GUI: Drop a GUI control onto a form, or create a GUI control at runtime. Examples of GUI controls in Windows would be a 'static text control' & 'edit box'. Examples within the Delphi IDE are TEdit, Tmemo, TTrackBar & TTreeView. See *fig. 4.* for a typical palette of off-the-shelf GUI control widgets.

3.1. Associate the GUI control with its mediating view object via the GUI's '`tag`' property or via a lookup registry, *more details on these techniques below*. For example, the constructor method for mediating views could always take a GUI control as a parameter, and set up the relevant associations then. E.g.

```
MediatingView1 := TCust_AGE_TextEdit_MediatingView.Create( edit1 );
                  // 'edit1' is an instance of a GUI text edit control
```

***A note on 'tags'*:**
A GUI control's '`tag`' property may be called something different in your particular IDE. In Delphi & Visual Basic it's called 'tag'. In X-windows it's called a 'resource'. In Visual C++ under the Microsoft Windows environment you can associate a 32-bit value (e.g. a reference to an associated mediating view object) with a control using `SetWindowLong/GetWindowLong` which are Windows API functions so this is not a solution specific just to C++ :

Simulating the tag property using Visual C++ code
in a Microsoft Windows API environment:

```
::SetWindowLong( aEdit.m_hWnd, GWL_USERDATA, (long) pMediator );
                  // Equivalent to aEdit.Tag = (integer) pMediator

pMediator = (CMediator *) ::GetWindowLong( aEdit.m_hWnd, GWL_USERDATA );
                  // Equivalent to pMediator = (CMediator) aEdit.Tag

// In the above example we assume the following type definitions

CEdit aEdit;        // A text edit GUI control (CEdit is an MFC class)
CMediator *pMediator;   // A pointer to a view mediator class
```

The above is C++ code:

3.2. Using your particular IDE, code the GUI control as normal, intercepting various user events e.g. `onTextChanged`. The GUI control widget event code can determine which mediating view object is 'looking after it' by looking in its own 'tag' property/pointer.

Have all such GUI event code talk back to the mediating view object, which will in turn, modify the appropriate model data. Alternatively, a more tightly coupled approach is to have the mediator merely provide a reference to the appropriate model business object so that the GUI control's event code can make the modifications to the relevant model business data itself. E.g.

```
procedure GUIControl.OnTextChanged (Sender: TObject);
  mediator := MediatingView(  Sender.Tag );  // Cast generic tag into View type.
  mediator.Subject.userText := Sender.Text; // Sender is a ref to a GUI Control.
end;
```

Delphi Object Pascal code.

Notice that the above `OnTextChanged` code does not need to call `mediator.Subject.NotifyOfUpdate` since the setter function `Customer.SetUserText` found on the subject's userText property would have already called `.NotifyOfUpdate` (see step 1.3)

3.3. As an alternative to the 'tag' technique, set up a registry object that associates 'mediating view' objects with GUI controls, and have your GUI control event handling code consult the registry at runtime to find its associated mediating view object. E.g.

```
procedure GUIControl.OnTextChanged (Sender: TObject);
  mediator := MVRegistry.FindMediatingView( sender ) // Sender is a GUI Control
  ...
  ...
end;
```

Delphi Object Pascal code.

Thus the resulting mediating view class acts as a mediator linking the interface of the GUI control to the interface of the business object/model. Another way to look at it is that the mediating class adapts the interface of the GUI class so that it can observe the model. The MGM mediating view perhaps might also be characterised as a type of two-way adaptor (GOF mention this term).

The mediating view class encapsulates the role of 'presenting' the business object data onto the GUI and in the other direction, of modifying the business object as a result of user actions in the GUI.

Where a smart GUI component is not being used, it is entirely possible for the mediating view class to draw a complex series of paint/draw operations onto a dumb canvas. Again, having the detailed drawing logic reside in the mediating view object, is a neat way of encapsulating that logic.

Why not MVC?

Why not use the classic Model-View-Controller (MVC) instead of Model-GUI-Mediator (MGM)?

1. MVC is actually a pattern of the internals of a GUI widget as well as of Model to GUI coupling. MVC thus involves unneeded complexity, when using ready-to-use, off-the-shelf widgets.

2. MVC's view and controller classes have many roles, and when some of those roles are taken over by a GUI widget, the model specific presentation & mediation roles have nowhere to go in the MVC paradigm. Only by introducing a new 'mediating view' class (as we do in MGM) can these roles be properly allocated and encapsulated.

3. MVC relies on both view and controller classes implementing Observer and subscribing to the model directly. When view and controller roles are combined into off-the-shelf widgets, in order to stay consistent with MVC, GUI widgets would somehow need to implement Observer. Sub-classing standard GUI widgets (and potentially also adding them to the particular development environment's tool palette) is a complex chore that can be avoided by creating an intermediate adapter / wrapper class that implements Observer. The adapter class (known as the 'mediating view' class in MGM) subscribes to the model and thereafter, when notified, forwards messages to and from the unmodified GUI widget.

Why not Document-View?

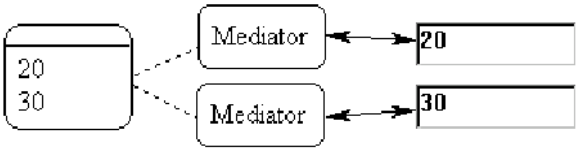
The Document-View (DV) architecture combines the roles of MVC's view and controller into a 'view' class. Typically used in a Microsoft C++ multiple-inheritance environment, GUI control widgets, especially GUI forms, are sub-classed from the view class, which implements observer. The model, in order to benefit from being observed, inherits from the 'document' class. DV users often avoid sub-classing GUI controls by introducing an adapter class which inherits from the 'view' class. The resulting pattern then closely resembles MGM.

MGM Variations

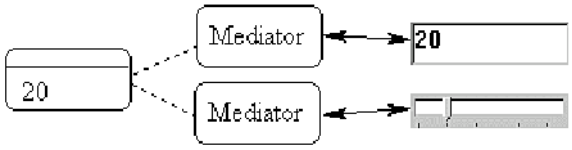
There are a number of variations of the MGM pattern. These variations are generated by the type of association between the model and the mediator, and between the mediator and the GUI. A notation for these variations might be of the form *model-mediator, mediator-GUI.* Thus *1-1, 1-1* would represent the classic one-to-one-to-one MGM pattern:

The classic
1-1 , 1-1
configuration

The next interesting variation of the MGM pattern is one business object to a number of mediator/GUI pairs. The notation for this would be *1-*, 1-1*. This configuration would be useful if you wanted to display different attributes of the same business object, or the same attribute in two different GUI controls:

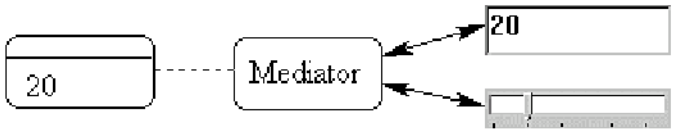


or



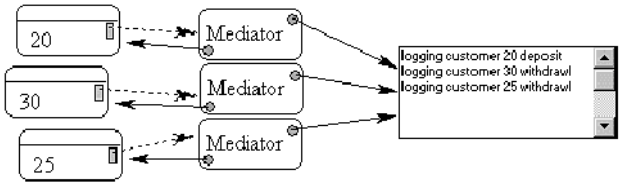
1-, 1-1* *1-*, 1-1*

Alternatively, a single mediator could take on the responsibility for maintaining multiple GUI controls, although this may have the disadvantage of a single mediator with too much responsibility:



*1-1, 1-**

A less common variation of MGM is that of multiple mediators feeding into a single GUI control. This is useful for 'logging information' functionality. One can log aspects of single or multiple business objects. Where the GUI control is read-only, the GUI need not point back to the mediating view, nor is there any need for GUI event code:

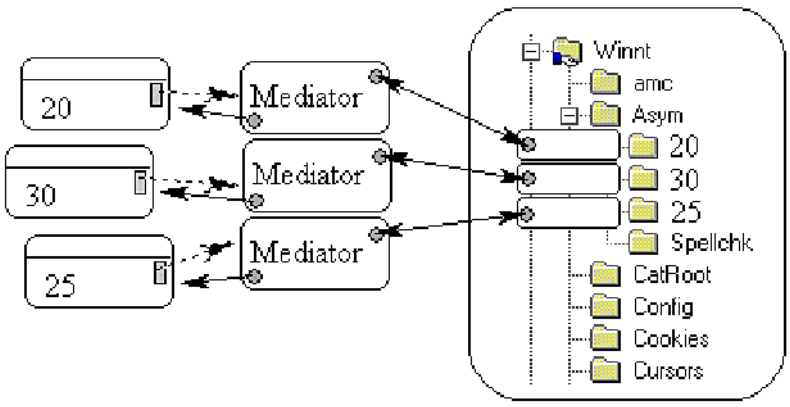


*1-1, *-1*

Variations based on Composite GUI components

Composite GUI components are collections of GUI controls that are themselves part of a larger GUI control. For example a treeview is a collection of tree nodes. A pie chart is a collection of pie segments. In a sense, all GUI controls are (usually) always housed within a larger GUI control – e.g. all GUI controls live on a form, which is itself a GUI control.

A typical MGM configuration for a composite GUI component such as a treeview is a set of one to one to one associations:

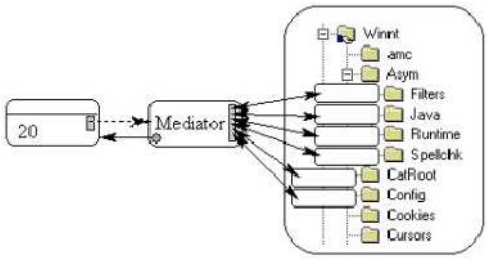


1-1, 1-1

This *1-1, 1-1* treeview configuration could be used, for example, to display all the business objects in your model on a treeview. A single attribute of each business object e.g. 'name' would be the visible representation of each business object on that treeview. Incidentally, if MGM has been

implemented correctly, then the user's in-place editing of a GUI tree node would alter the value of the 'name' attribute not only on the treeview, but also in the associated business object.

A variation of the treeview configuration is to reduce the number of mediators to one, at the risk of making the mediator heavy in responsibility:



*1-1, 1-**

Such a single mediator looking after many GUI sub-components might be more appropriate when only a single business object is involved. For example, many aspects/attributes of the same business object are being displayed by the one mediator onto a treeview GUI control.

Variations based on Manager Objects

More variations of MGM revolve around aggregating not only GUI controls, but mediators and business objects as well. Such 'Manager' objects can provide functionality for collections of MGM objects. For example, a model business object manager might offer persistence functionality for a collection of business objects. Some mediating views might actually subscribe to and observe business object managers rather than individual business objects. Such managers would observe business objects, and be observed by mediating views. For example, a read-only GUI text label displaying the number of objects existing in the model would be implemented as follows: a 'ModelObjectCounter' mediating view object would be created and would subscribe to the model manager object. When triggered, the mediating view would ask the manager object for the number of business objects that existed and display this number in the appropriate GUI text label control.

Manager objects are also useful on the mediating view side of things. A 'view populator' class should probably be created and be responsible for the important step of creating the initial set of mediating view objects and setting up the relationships between the GUI controls, mediators and business objects (especially for composite GUI controls like treeviews and their nodes). This might include either setting up 'tag' pointers in GUI controls or adding entries to a central mediator to GUI mapping registry. The View Handler pattern (POSA) can be used to manage various 'view managers'. More ambitiously, composite, and other various patterns of mediating views might be formed in order to systematically and generically compose an interface (see MVG).

Limitations

Replacing either the GUI control or business object will affect view mediator code and property types. For example, replacing a 'text edit control' for a 'masked text edit control' (e.g. the new GUI control does some neat validation) means rewriting the mediating view class slightly to deal with the new GUI control. At the very least, the pointer type from the mediating view class to the GUI control has to change (unless the pointer to the GUI control is to a generic base GUI control class). Alternatively (and perhaps preferably), for each new GUI control, a new mediating view class should be created that specifically points to and has code to handle that new GUI control.

Creating a new class for each business object to GUI control mapping results in a proliferation of mediating view classes. To reduce the number of mediating view classes more generic mediating views could be introduced which could handle a wider range of GUI controls, or a broader set of business objects or business object attributes. For example, a CutomerTextEdit mediating view

class could handle the display of any customer attribute in a text edit control. A technique for specifying to the mediating view class exactly which attribute to display would need to be invented.

There is a trade off between placing code that modifies the business object in the GUI event handler, or putting that same code in the mediating view class – in order to keep the GUI event handling code simple and to keep model knowledge encapsulated within the mediating view class. For example, when the user changes the contents of a GUI text field, the GUI event handling code can ask the mediator for the relevant business object, and modify that business object directly. Alternatively, the GUI event handling code could simply notify the mediator, and let the mediator handle the details of the change to the model. Keeping model changes encapsulated in the mediating view class is probably the cleanest solution, minimizing the impact of swapping one GUI control for another.

Consequences

All the forces are resolved in the MGM pattern solution, however there are the various issues raised in the limitations section to consider. If any forces could to be said to be less fully resolved these would fall under the category of encapsulation & loose coupling (see forces **12 & 13).** The GUI can potentially still know too much about the model if implementations allow this, though in most cases this may not be such a bad thing as long as the more important principle is adhered to: viz. that the model is ignorant of the GUI.

Another consequence is that we now need to manage collections of view objects, including their initialisation and shutdown. As mentioned earlier, variations of the View Handler pattern might be used to help deal with these new forces. More ambitious composite views would involve even more design, pattern! and coding work.

Known Uses

Reason! Software for critical thinking (1998-2000), Melbourne University, Australia uses MGM for various treeview and combo box displays.

Dolphin Smalltalk 98 - All of the Dolphin development system tools are built using the MVP (Model View Presenter) framework. MVP is similar to MGM in that a mediating 'presenter' class plays the key role, as the 'mediating view' class does in MGM.

Large scale South African / Australian client server based car management system 1999 uses MGM for all GUI displays.

-Andy Bulka
abulka@netspace.net.au
www.andypatterns.com