Model-GUI-Mediator

Graeme Geldenhuys

2008-09-23

After mastering OOP and Object Persistence, the next thing developers need to conquer is how to present their business objects in the GUI and how for the user to interact with them. This article presents a design pattern that helps solve that obstacle, and it's called Model-GUI-Mediator or MGM for short. The MGM design pattern allows programmers to simulate "object-aware" controls using off-the-shelf GUI controls.

Overview

With today's RAD (Rapid Application Development) IDE's being so popular, developers simply use the technology handed to them, and often without considering other options. RAD IDE's are excellent in producing quick user interfaces in a visual way, but using data-aware components are not always the best option. When the developers use data-aware components, they tie themselves to that particular component suite and bring the Data layer into the GUI layer - hard coding the data structure into their applications. You will quickly be locked into a specific vendor library and database design. Data-aware components might be convenient in some cases, especially for creating prototype applications, but it's not very flexible for production systems that need to be maintained over a long period. Over time maintenance becomes a nightmare when business rules have been scattered all over the place. Some rules lives in Data Modules, some in a separate code unit and some in the GUI forms themselves. Such a design is messy and makes it very hard to extend. You are also limited to the data-ware components included with our IDE, or you need to purchase data-aware components from third party vendors.

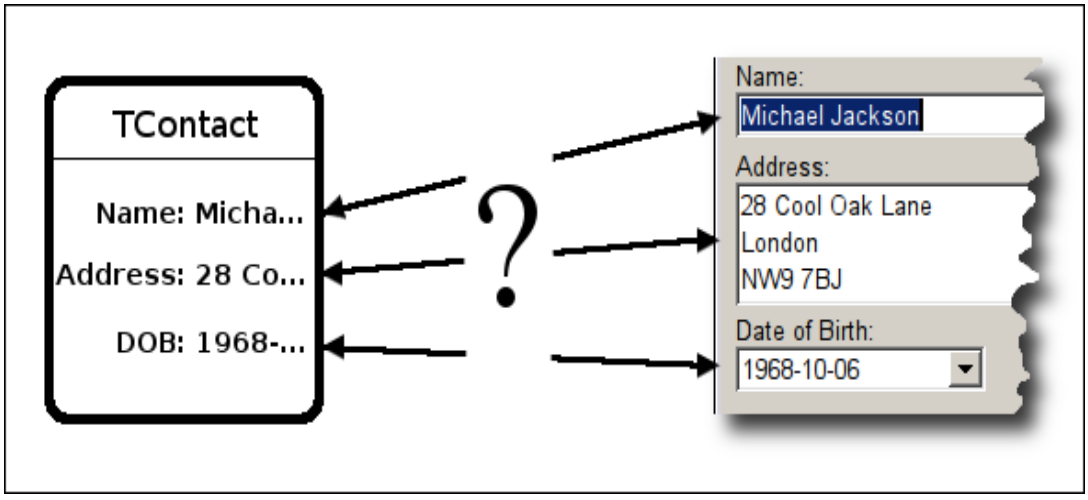

Figure 1: How do we represent and interact with a business object in a GUI?

The Model-GUI-Mediator design pattern, is a quick and easy way to solve these problems. MGM allows you to simulate "object-aware" components using standard off-the-shelf GUI controls. Like other similar design patterns (Model-View-Controller or the newer Model-View-Presenter), MGM keeps the model data and graphical presentation separated. There is also no need to create specialised descendants of GUI components, as in the case of Model-View-Controller or data-aware GUI controls.

MGM is more appropriate with today's RAD IDE's like Delphi, Lazarus, Visual C++, JBuilder etc. All the newer GUI toolkits have their own event handling systems built in. Unlike MVC or MVP, the MGM design patterns works well with today's modern GUI toolkits and take advantage of their existing event models. Take for example the TEdit component in Lazarus. It already comes with an OnChange event, and MGM simply hooks into that event to know when the content in the TEdit component has changed. Compare that to MVC and MVP that recreate the whole event model.

MGM encourages the developers to design and write their programs using a proper object model, whereas data-aware components encourages the developers to adopt the relational database model. By using a proper object model, also has the benefit that it is much easier to implement Unit Testing, which is becoming more and more popular. The data has already been separated from the GUI—a prerequisite for unit testing.

Participants and Structure

Model-GUI-Mediator actually uses two classic design patterns together to accomplish it's task. The first of the two patterns is the *Mediator Pattern*[1]. The MGM pattern creates a *mediating view* class that relays all communication between the GUI component and the business object. The second design pattern is the *Observer Pattern*[2]. The business object communicates any changes to the *mediating view* class via observer.

To fully understand the rest of this article, I will first explain the different parts or terms used in the Model-GUI-Mediator design pattern.

Model - This is the business object or objectlist that we want to visualise in the GUI of

1 Design Patterns (aka the Gang-of-Four book): Mediator page 273.
2 Design Patterns (aka the Gang-of-Four book): Observer page 293.

our application, and allow the user to interact with the attributes of the business object via the GUI.

Mediating View - This is the most important part of this pattern, and the part that works the hardest. The mediating view class is an observer of the Model and gets notified of any changes in the Model. The mediating view also relays any communications between the GUI and Model, and vice versa.

GUI - This is the GUI component the user interacts with. For example, a edit box, combo box, memo, list view, tree view etc. When the user interacts with the GUI component, its built in event handling notifies the mediating view class of any changes. In turn the mediating view updates the Model. The GUI component never communicates directly with the Model.

I always find it easier to understand something if I can visualise it. So to help explain the interaction between all the parts of MGM, I will present it in a diagram. Figure 2 shows the interaction of all the parts of MGM, as I just explained.

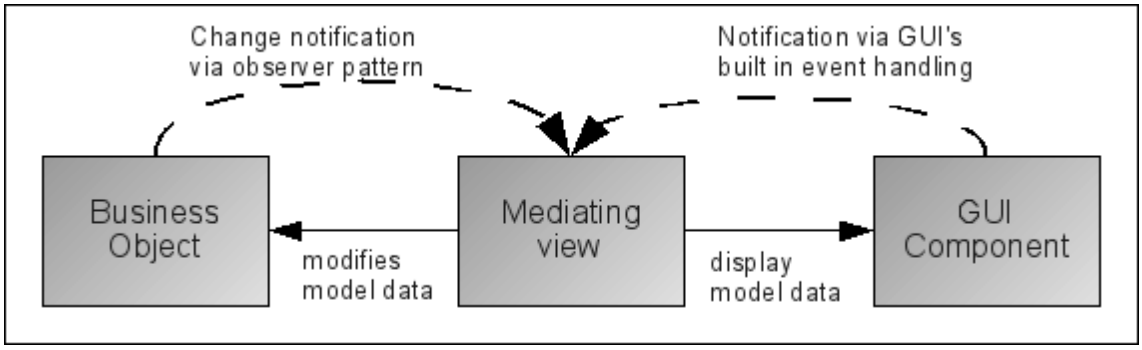

Figure 2: Interaction between all the MGM parts.

Implementation

I will be showing you how to implement the Model-GUI-Mediator using the Object Pascal language. Please note that MGM is not limited to Object Pascal alone. Other developers have implemented MGM in Dolphin Smalltalk, C++, Ruby etc.

I created a UML diagram as shown in Figure 3. This gives you an overview of the MGM pattern we are going to implement.

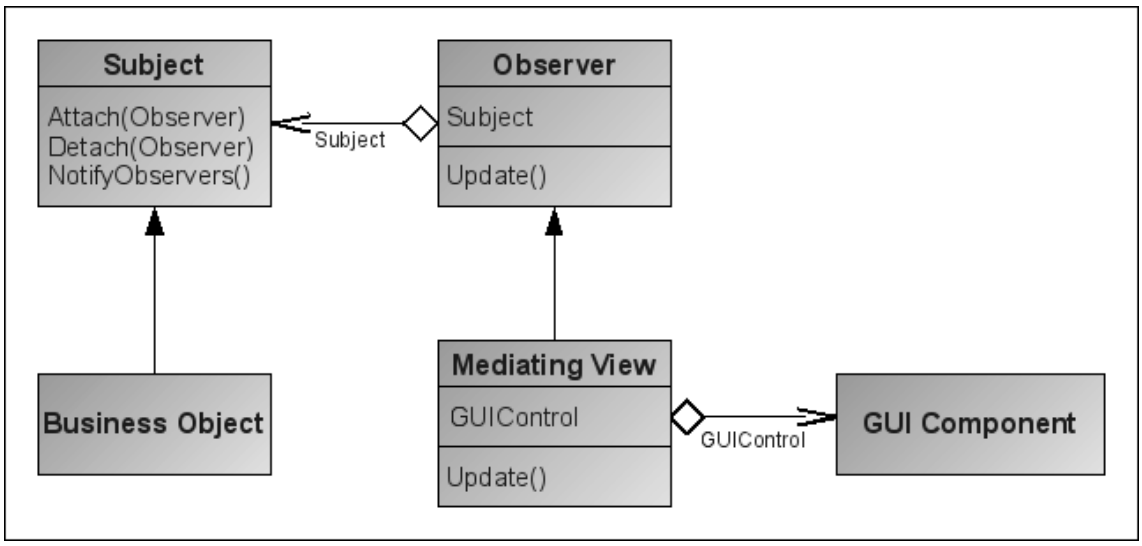

Figure 3: A UML diagram showing the structure of the MGM pattern.

To implement the MGM pattern, we first need to implement the helper pattern - Observer. I will briefly explain the Observer pattern, but I recommend you research the pattern further, to get a full understanding of how it works. As a matter of interest, if you are already using something like a Object Persistence Framework (OPF) for persisting your objects, the OPF's base object class probably has the Observer implemented already. Observer is a commonly used pattern. tiOPF (TechInsite Object Persistence Framework)[3] is one example of an OPF framework that has built in support for Observer.

The intent of the Observer pattern as described in the *Design Patterns* book: "Define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically."

With that new found knowledge and the UML diagram in Figure 3 to guide us, we need to start by creating an abstract `TObserver` class. This is a really easy class to implement, as just one method is needed. See Listing 1. This implementation supports multiple subjects for each observer. The subject passed to the `Update` method lets the observer determine which subject changed when it observers more than one. In this article we will keep it simple an only limit ourselves to observing one subject.

Listing 1:
```
TObserver = class(TObject)
public
  { The method that notifies us that ASubject has changed }
  procedure Update(ASubject: TSubject); virtual; abstract;
end;
```

The next abstract class we need, to complete the Observer design pattern implementation is the `TSubject` class. The Subject class is what will be observed. It can also have any number of Observer objects observing it. Listing 2 shows the interface of our `TSubject` class.

3 TechInsite tiOPF - http://www.tiopf.com

```
Listing 2:
TSubject = class(TObject)
private
  FObserverList: TList;
public
  constructor Create; virtual;
  destructor Destroy; override;
  { Attach a new observer }
  procedure AttachObserver(AObserver: TObserver); virtual;
  { Detach an existing observer }
  procedure DetachObserver(AObserver: TObserver); virtual;
  { Notify all the attached observers about a change }
  procedure NotifyObservers; virtual;
end;
```

Internally we have a field variable, `FObserverList`, of type `TList`. This will store the list of Observer references—objects observing the Subject. We also have a method called `AttachObservers`, which is how an Observer object registers itself with the Subject, so it can be notified on any changes in Subject. We also have a `DetachObservers` method which does the opposite, by unregistering an Observer with the Subject ,so it doesn't get notified of any changes in the Subject anymore. The last method is called the `NotifyObservers` method. Whenever the internal state of Subject changes, `NotifyObservers` is called. The implementation of `NotifyObservers` loops through all registered Observers stored in `FObserverList` and calls the `Update` method of each of those Observers. When it calls `Update`, it passed itself as a parameter, so the Observer knows which subject has changed. Listing 3 shows the implementation of `NotifyObservers`.

```
Listing 3:
procedure TSubject.NotifyObservers;
var
  i: Integer;
  lObserver: TObserver;
begin
  if not Assigned(FObserverList) then
    Exit; //==>
  for i := 0 to FObserverList.Count-1 do
  begin
    lObserver := TObserver(FObserverList.Items[i]);
    if Assigned(lObserver) then
      lObserver.Update(self);
  end;
end;
```

That is all we need for a basic Observer pattern! Like I mentioned earlier, this is a very simplified implementation of the Observer pattern, but it will do fine for what we need in MGM. I still recommend you research the Observer pattern further, to fully learn what other features it can have and alternative—more complete—implementations.

Model

Now that we have the Observer implemented we can move to the next step. Our ex-

ample application is a very basic Address Book type application. Our main form will show a list of contacts. We will be able to insert, edit and delete a contact. Listing 4 shows the class interface of our business object we are going to observe and create mediating views for.

Listing 4:
```
{ our business object we are going to observe }
TContact = class(TSubject)
private
  FAddress: string;
  FDateOfBirth: TDateTime;
  FName: string;
  procedure SetAddress(const AValue: string);
  procedure SetDateOfBirth(const AValue: TDateTime);
  procedure SetName(const AValue: string);
public
  constructor Create; override;
  property Name: string
            read FName write SetName;
  property Address: string
            read FAddress write SetAddress;
  property DateOfBirth: TDateTime
            read FDateOfBirth write SetDateOfBirth;
end;
```

The Contact object has three properties. A `Name` property which is of a basic string type. An `Address` property which will store multi-line address information. And lastly, we have a `DateOfBirth` property. which is of a TDateTime type. Our business object or model implements the *Subject* interface of the Observer pattern. All we had to do to accomplish that, was to let our `TContact` class descend from the `TSubject` class. By doing that, we now have access to the `AttachObserver`, `DetachObserver` and `NotifyObservers` methods inside our business object.

As you might have noticed, we also have setter methods defined for each of the `TContact`'s properties, e.g. `SetName` or `SetDateOfBirth`. This is very important. When a business object attribute changes, e.g. `Name` or `DateOfBirth` the business object needs to somehow inform all interested observers of the change. The way we do that, is to call the `NotifyObservers` method inside the setter method. Listing 5 shows how this is done for the `DateOfBirth` property.

Listing 5:
```
procedure TContact.SetDateOfBirth(const AValue: TDateTime);
begin
  if FDateOfBirth = AValue then
    Exit; //==>
  FDateOfBirth := AValue;
  NotifyObservers;
end;
```

Mediator

I am going to implement the mediators in two phases. A base mediator class for each GUI component we want to represent. These mediator classes are generic, and we will be able to reuse them over and over for different business objects.

The second phase of the mediator implementation is to create the *mediating view* class, which is specific to each business object. This class will complete the mapping between the Model attribute and the GUI component.

A mediator observers the Model, so we will start by deriving a base mediator class from the `TObserver` class. Listing 6 shows the class declaration for our `TBaseMediator` class.

Listing 6:

```
TBaseMediator = class(TObserver)
protected
  { Subscribe to the Subject }
  procedure DoAttachment(ASubject: TSubject);
  { placeholder method where we update the
    Subject due to GUI changes }
  procedure DoGuiToObject; virtual;
  { placedolder method where we update the GUI
    due to Subject changes }
  procedure DoObjectToGui; virtual;
end;
```

This class has three methods: `DoAttachment`, `DoGuiToObject` and `DoObject-ToGui`. The purpose of the `DoAttachment` method is to register the Mediator class as an observer of the Subject. I implemented this for convenience, so we do not have to call SomeSubject.AttachObserver(SomeMediator) manually at a later stage in our program. Listing 7 shows the implementation of `DoAttachement`. Nothing too complicated. We do a quick sanity check to make sure a valid subject has been passed in as the `ASubject` parameter. We then set a internal field to point to the Subject, in case we need to reference the Subject later. We then call the Subject's `AttachObserver` method passing in the instance of the mediator as `self`. The mediator class can now be notified of any attribute changes happing in the Subject.

Listing 7:

```
procedure TBaseMediator.DoAttachment(ASubject: TSubject);
begin
  if ASubject = nil then
    raise Exception.Create('You must specify a Subject' +
      ' to be observed.');
  { set internal subject reference for later use }
  FSubject := ASubject;
  { Register as an observer of Subject }
  FSubject.AttachObserver(self);
end;
```

The `DoGuiToObject` and `DoObjectToGui` are template methods. They do not do anything yet, but will be implemented later in the more specific mediating view classes. The `DoGuiToObject` will be used to update the Model object, based on information from the GUI component. The `DoObjectToGui` does the opposite, it will be used to

update the GUI component based on a data from the Model object.

Now that we have our base mediator class complete, we can start implementing GUI component specific mediators. I will only show you the implementation for a Text Edit component. The sample code accompanied on the Toolbox DVD includes the rest of the mediators used by our example project. Listing 8 shows the class declaration for the `TEditMediator`.

Listing 8:
```
TEditMediator = class(TBaseMediator)
private
  FGUIControl: TfpgEdit;
  { event handler to detect changes in the GUI component }
  procedure InternalOnChange(Sender: TObject);
public
  { A custom constructor so we can pass in both the GUI
    control and the Subject we want to observe. }
  constructor CreateCustom(AControl: TfpgEdit;
      ASubject: TSubject);
  { A read-only property pointing to the GUI Control
    we are managing. }
  property GUIControl: TfpgEdit read FGUIControl;
end;
```

We declared a custom constructor, so that when we create a instance of this mediator class, we can pass in both the GUI control and the Model. The two parts the mediator needs to communicate with. To refresh your memory on how the mediator interacts with the GUI control and the business object, refer back to Figure 2. The constructor is quite straight forward. We call the default `Create` constructor. We then assign the GUI control instance to a internal field variable. This is so we can reference the GUI control at a later stage, e.g. when we need to update the GUI control when the business object we are observing has changed. We assign an event handler to the GUI control's `OnChange` event. This will allow the mediator to react to any data changes made by the user in the Text Edit component. This is where MGM takes advantage of the built in event handling of the modern GUI frameworks. We can hook into any appropriate event of the GUI component, it doesn't have to be the `OnChange` event. Other events like `OnExit` (when the component looses focus) could also have been used. And lastly we call the `DoAttachement` method, which subscribes to the Model. Listing 9 shows the code for the constructor.

Listing 9:
```
constructor TEditMediator.CreateCustom(
    AControl: TfpgEdit; ASubject: TSubject);
begin
  Create;
  FGUIControl := AControl;
  FGUIControl.OnChange := @InternalOnChange;
  DoAttachment(ASubject);
end;
```

Now that we have a generic Text Edit mediator, we can move on to phase two—creating a *mediating view* for our `TContact` business object, and for a specific text property of that business object. Because we implemented most of the work already in the generic mediator class, the mediating view class is very simple and quick to do. All that re-

mains, is for us to tell the mediator how to update the GUI and how to update the Model. For that, we need to implement the two template methods: `DoGuiToObject` and `DoObjectToGui`.

For this example, I am going to create a mediating view for the TContact.Name property. When I create the mediating views, I like to name the classes as verbose as possible, so that when I look at the class name, I can clearly see what that class is meant for. I tend to use the following naming convention, but you are obviously allowed to use whatever you feel comfortable with:

Model class name + Property Name + Component + "View"

So with that being said, here follows the class declaration for our first mediating view.

```
TContactNameEditView = class(TEditMediator)
protected
  procedure DoGuiToObject; override;
  procedure DoObjectToGui; override;
end;
```

And below you can see the one line implementation of `DoGuiToObject`. We simply get the text value from the GUI component and assign it to the `Name` property of our TContact instance.

```
TContact(Subject).Name := GUIControl.Text;
```

The next method follows the same style, but in the opposite direction. Here follows the one line implementation for `DoObjectToGui`. We assign the `Name` property of our TContact instance to the `Text` property of our GUI component.

```
GUIControl.Text := TContact(Subject).Name;
```

As I mentioned earlier, we did all the hard work in the generic mediators, so now our concrete mediating view classes are a lot more simplistic and quick to implement. One has to love Object Oriented Programming! :-)

GUI

I have already touched on some of the GUI component topics in the Mediator section, but I will now go into a bit more detail. So how do we solve the last part of our puzzle?

Simply drop a standard GUI control onto a form, or create a GUI control at runtime. In our example, we place a Text Edit component, like the TEdit from Lazarus or Delphi's component palette, onto a form. The Model-GUI-Mediator pattern can be applied to just about any component, even something like a TPaintbox, but I will get into that a little bit later.

Now that we have our GUI component, we need to somehow associated it with our mediating view class. We first create a private field variable of type `TContactNameEditView`. This will hold the instance of our mediating view class.

Then we need to instantiate our mediating view, and at the same time we will associate our GUI component and Model (our Contact object) with the mediating view.

Below is the line of code to accomplish this. We can place this line somewhere in our Form's OnCreate or OnShow event handler.

```
FmedName := TContactNameEditView.CreateCustom(edtName, Data);
{ create more mediating views here... }
```

`FmedName` is our mediating view instance variable. The `edtName` parameter is our TEdit component instance, and `Data` is our TContact business object instance, or Model. Now that we have our GUI all hooked up, we need to tell our business object to make sure all the GUI components show the correct initial information. We only need to do this once, and we do it after we created our mediating views. To do that, we simply call the business object's `NotifyObservers` method, as show below.

```
{ Now all observers will have a chance to
  update themselves }
Data.NotifyObservers;
```

All that still remains, is some clean-up code. You must remember to free the `FmedName` instance when the Form is destroyed. And that's it, our GUI in all done!

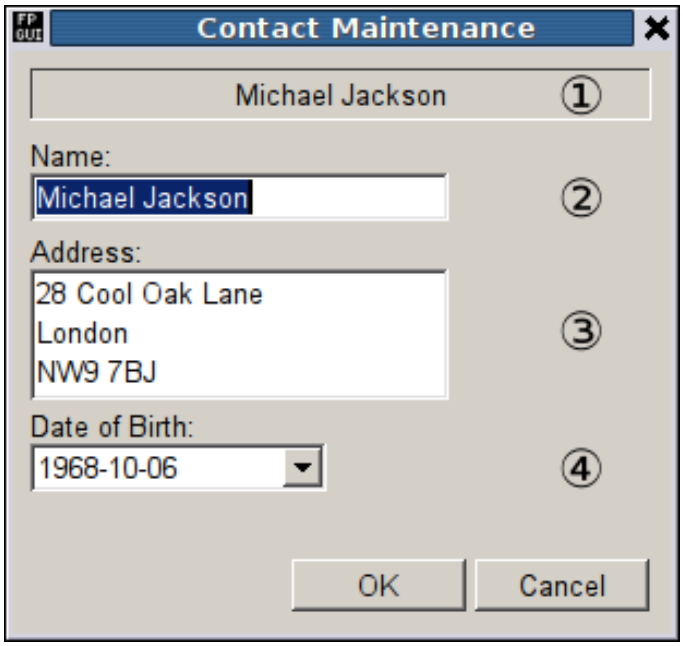

Figure 4: Our example application with four different mediators.

A screenshot of our example application is shown in Figure 4. I highlighted, using numbers, the four different mediators that it uses.

- (1) uses a read-only mediator that uses a TPanel GUI component. It observes the TContact.Name property.
- (2) uses a text edit mediator that uses the TEdit GUI component. It also observes the TContact.Name property. As you can see, you are not limited to one mediat-

or per business object attribute. The nice thing here is that while you update the name in the text edit, the name in the read-only panel is updated at the same time. The text is always in sync. The mediator updates the Name property as the user types in text, reacting to the TEdit.OnChange event.

- (3) uses a multi-line edit mediator that uses the TMemo GUI component. It observes the TContact.Address property. In this mediator, the Address property is only updated when the TMemo looses focus, reacting to the TMemo.OnExit event.
- (4) uses a date edit mediator that uses the TCalendarComboBox GUI component. This mediator observes the TContact.DateOfBirth property. The mediator updates the DateOfBirth property when a date has been selected, reacting to the TCalendarComboBox.OnChange event.

Variations and Improvements

The MGM implementation I showed you here is a somewhat simplified implementation —purely to make it easier to explain and understand. Once you understand the basics of Model-GUI-Mediator, you will find more and more ways to use it and how to improve it to suite your needs.

I have only shown the basic edit mediators. A one-to-one relationship with a business object attribute. One attribute, one mediator and one GUI component.

Alternatively, a single mediator could also take on the responsibility for maintaining multiple GUI components. The disadvantage of this is that the mediator might become to complex and take on to many roles.

Another alternative is where you have multiple mediators feeding information to a single read-only GUI component e.g. a TMemo. This could be implemented in an application that maybe needs to log attribute changes from various business objects to a single “logging” screen.

Then we also have the case of Object Lists. All the mediators we spoke of so far only worked with a single business object. You can also implement mediator that observer a Object List class, and sends out notifications to observers when a object was added or deleted. Such a mediator could use GUI component like a Lazarus TListBox or TComboBox etc. The list mediator could manage the items being displayed in such components.

Then we also get the more advanced mediators. Those that manage composite GUI components. For example a Lazarus TTreeview, TListView or TStringGrid. Composite GUI controls are collections of GUI controls that are themselves part of a larger GUI control. If we use the Treeview as an example. The treeview is a collection of TTreeNode components. We could implement a composite mediator following the same idea - maintaining the treeview and the treenodes.

If you would like to look at a more complete MGM implementation with many advanced features, you can take a look at the tiOPF project. The source code units of interest are: `tiBaseMediator.pas`, `tiFormMediator.pas` and the `tiMediators.pas` units. tiOPF is freely available on Sourceforge.net.

As you can see, there are many variations of how you can use the Model-GUI-Mediator

design pattern. I have only listed a few ideas, but I am pretty sure you will come up with even more, to better suite your needs. MGM is a very versatile pattern, and is a great way to keep your business objects and business rules separated from your GUI layer. Something that is very important in a well designed OOP based application.