AndyPatterns

- Refactoring to PureMVC - Java version

Refactoring to PureMVC

PureMVC - What's all the fuss about?

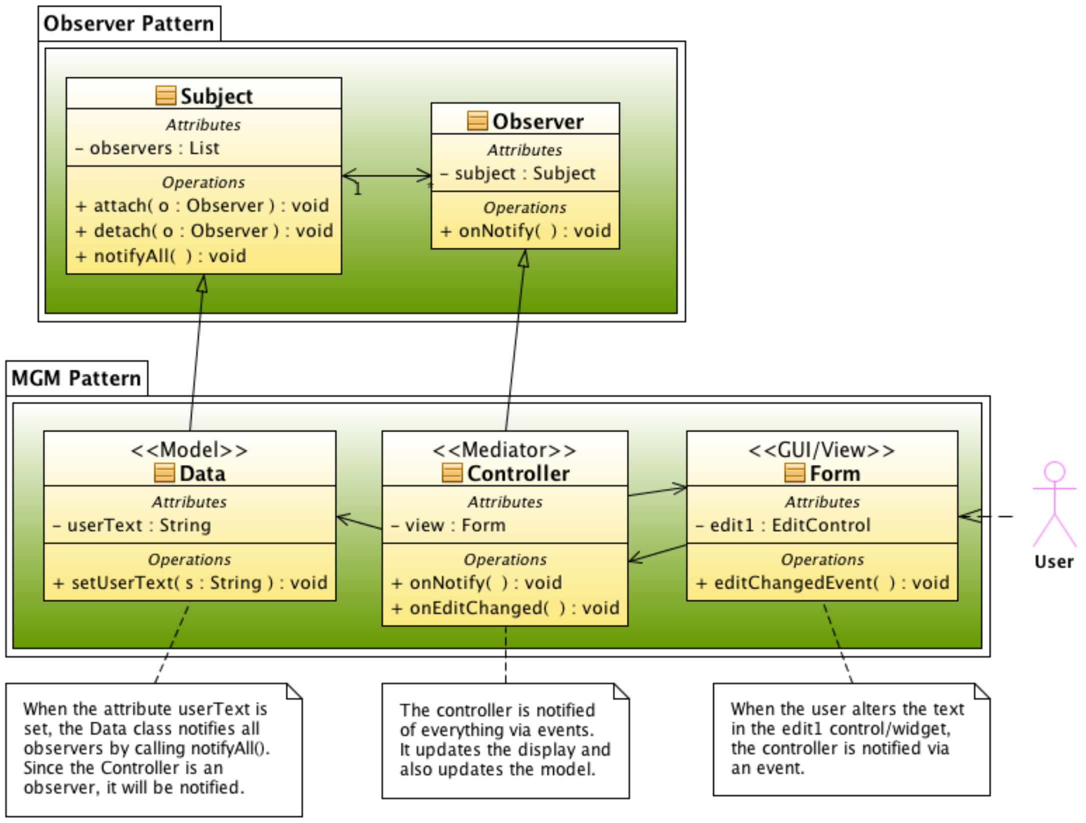
I have long been fascinated by the MVC (Model View Controller) architectural pattern, first conceived in the 1960's. What it promises, to those who can fathom its mysteries, is an orderly way of organising your application architecture. You define a de-coupled domain/business model that is oblivious to any GUI that might be displaying it. A mediating controller class usually looks after the dirty details of moving data between the model and GUI. You can even have multiple GUI representations of your single model e.g. a pie chart view and a bar chart view simultaneously displaying the same information out of the model.

There are many variants of MVC, and people use different terminology to mean the same thing, and conversely people use the same terminology to mean different things e.g. the view could be the GUI or it could be the mediating class looking after the GUI, which means it could actually end up meaning the controller...

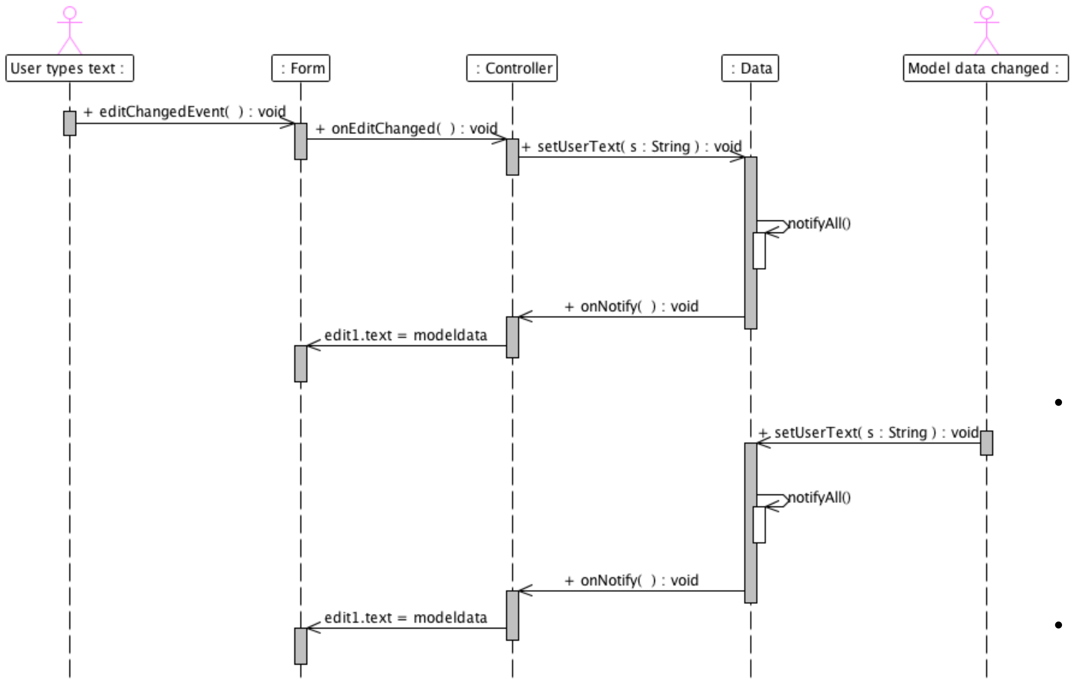
Some History

I once wrote up a pattern called MGM (Model Gui Mediator) which describes a version of MVC that I thought made sense in today's modern programming age. Today, GUI views were usually comprised of sophisticated controls/widgets rather than laboriously handcrafted graphic code that needed custom controller code to handle the intricate details of interacting with the view/GUI. Most of that low level controller interaction is now built in to the off the shelf widgets/controls.

The controller is now just a mediator between model and view – hence the “M” in MGM (Model-Gui-Mediator). You might prefer to think of the mediator as the controller, so the pattern could well have been called MGC (Model-Gui-Controller). Many modern variants of MVC e.g. MVP (Model View Presenter) and PAC (Presentation Abstraction Controller) and others probably fit into the basic idea of MGM.


Observer Pattern
Subject
Attributes
- observers : List
Operations
+ attach(o : Observer) : void
+ detach(o : Observer) : void
+ notifyAll( ) : void
1
*
Observer
Attributes
- subject : Subject
Operations
+ onNotify( ) : void
MGM Pattern
<<Model>>
Data
Attributes
- userText : String
Operations
+ setUserText(s : String) : void
<<Mediator>>
Controller
Attributes
- view : Form
Operations
+ onNotify( ) : void
+ onEditChanged( ) : void
<<GUI/View>>
Form
Attributes
- edit1 : EditControl
Operations
+ editChangedEvent( ) : void
User
When the attribute userText is set, the Data class notifies all observers by calling notifyAll(). Since the Controller is an observer, it will be notified.
The controller is notified of everything via events. It updates the display and also updates the model.
When the user alters the text in the edit1 control/widget, the controller is notified via an event.

The behaviour of the MGM architecture is as follows:



MGM pattern sequence diagram. M=Model (Data class) G=Gui (Form with an edit control on it) M=Mediator (Controller class)

What we have illustrated above are two use cases:

- In the first the user types some text into an edit field on a form, which causes the model to change – the model broadcasts the change and the GUI is updated (watch out for infinite loops here of course, which can be avoided by the gui update not triggering another change event).
- The second use case is the model changing for some reason (e.g. being loaded from a file or some other part of the system altering the data) and the model broadcasting the change – again the Controller is observing and intercepts the broadcast and updates the GUI.

What is interesting about MGM is that it shows how simple the MVC pattern really can be, when understood in terms of today's modern GUI components. It also serves as a way of contrasting what PureMVC is offering. Read on.

Along comes PureMVC

I've since (2008-2009) become quite interested in the PureMVC framework because it adds a few things that were missing from MGM.

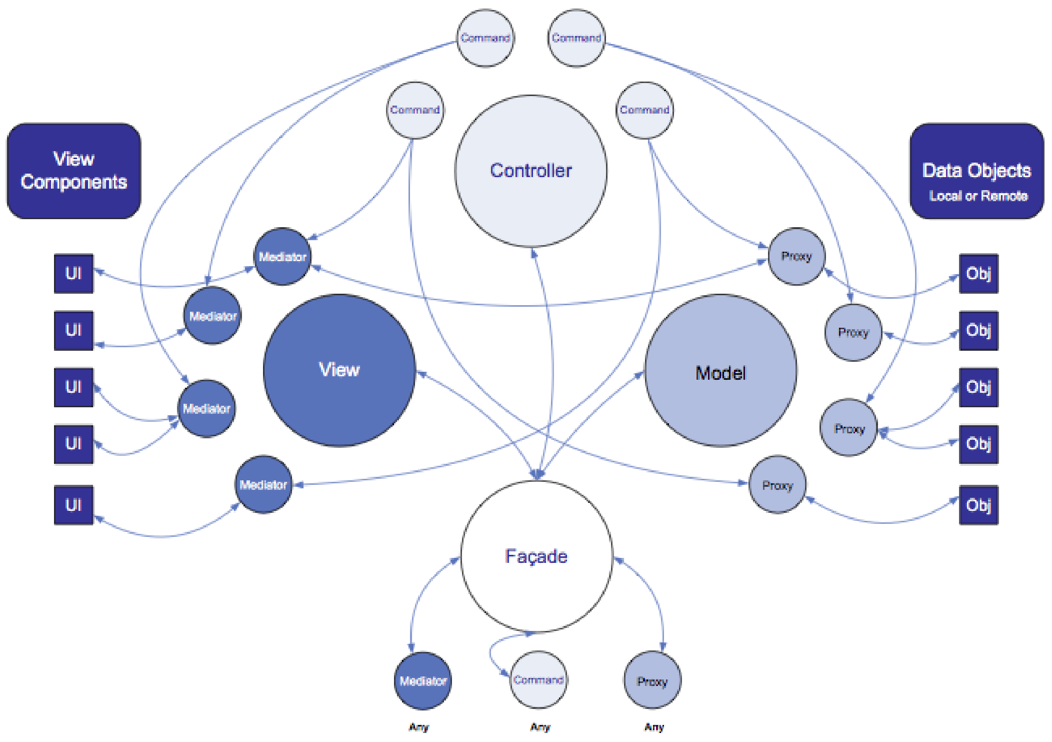
PureMVC Conceptual Architecture

Firstly PureMVC explicitly adds command classes – rather than just burying behaviour in controllers/mediators, we have an official way of organising behaviour. Sure, in MVC and MGM, any controller is free to delegate and invoke command classes to perform behaviour in a more organised way - but in PureMVC the details of this are a little more spelled out.

Secondly PureMVC adds service location, so that you end up with a nice de-coupled design – events are raised and anyone can register interest in an event/message. Thus the whole architecture becomes a lot more pluggable, flexible and neat.

Thirdly I believe PureMVC addresses scalability – for example it tells you exactly how to add multiple mediators to the system. Multiple commands and data proxies are also supported. You simply register all these classes and specify what noification messages are of interest to each of them.

PureMVC rules



The model notifies, but doesn't listen and ultimately knows nothing about anything else in the system. Mediators notify and listen to messages. Commands are invoked (cmd.execute()) automatically by certain messages and they send out messages if they want to.

The facade is the communication hub.

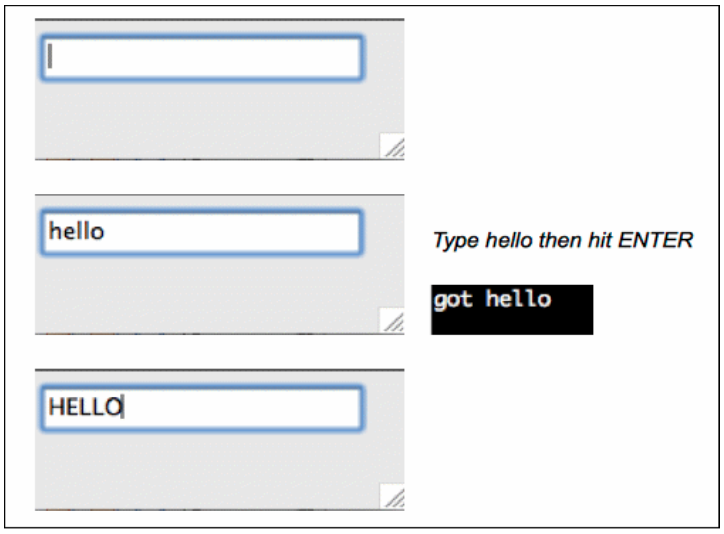
Mediators look after gui elements, and this forms the view. Commands forms the controller. Proxies wrap model elements and this forms the model. *Note that in the above diagram, there is a one to many relationship between the View and the Mediators, and between the Controller and the Commands, and between the Model and the Proxies.*

Refactoring to PureMVC

To show you just how useful and simple incorporating PureMVC can be, I have designed a step by step tutorial on how you introduce a PureMVC command driven, message notification driven architecture to your existing wxpython application.

We will start with a simple application that doesn't even have a model. We will first add a mediator, then a command class and then a proper model.

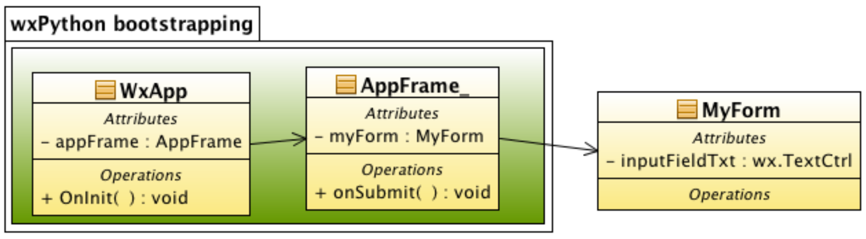
The application is a simple form which displays a textfield. When you hit ENTER, it converts anything you type into uppercase and displays it in the textfield. There is no "model" and no PureMVC architecture.



Three screenshots of our simple wxPython application.
We will use this as our starting point and refactor to PureMVC.

Step 1. Starting Point

Here is the code for our simple wxpython application illustrated above. It has no proper model and does not use PureMVC. This is our starting point.

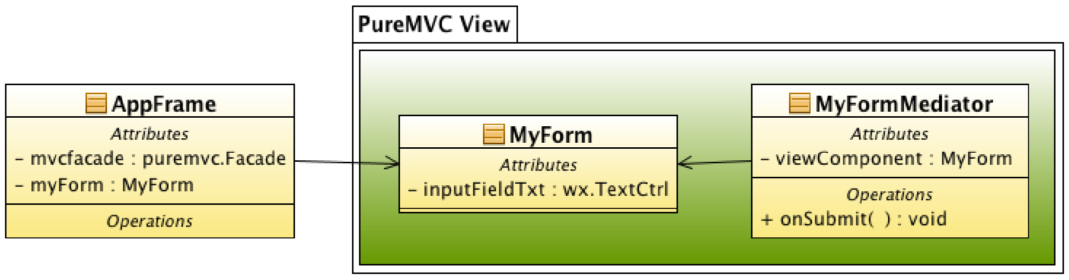


Starting point – a simple wxpython app - a single form with a textfield control.

```
import wx

class MyForm(wx.Panel):

    def __init__(self, parent):
        wx.Panel.__init__(self,parent,id=3)
        self.inputFieldTxt = wx.TextCtrl(self, -1, size=(170,-1), pos=(5, 10), style=wx.TE_PROCESS_ENTER)

class AppFrame(wx.Frame):
    myForm = None

    def __init__(self):
        wx.Frame.__init__(self,parent=None, id=-1, title="Refactoring to PureMVC",size=(200,100))
        self.myForm = MyForm(self)

        self.myForm.Bind(wx.EVT_TEXT_ENTER, self.onSubmit, self.myForm.inputFieldTxt)

    def onSubmit(self, evt):
        mydata = self.myForm.inputFieldTxt.GetValue()
        print "got", mydata
        self.myForm.inputFieldTxt.SetValue(mydata.upper())


class WxApp(wx.App):
    appFrame = None

    def OnInit(self):
        self.appFrame = AppFrame()
        self.appFrame.Show()
        return True

if __name__ == '__main__':
    wxApp = WxApp(redirect=False)
    wxApp.MainLoop()
```

Step 2. Import PureMVC and add a mediator

Let's add a mediator and create a PureMVC "view". Mediators in PureMVC are classes which look after a gui – e.g. a wxpython form. Mediators can be built that look after larger or smaller chunks of your gui – depending on your situation. In our case we will build a mediator to look after the one form and its single textfield.



Step 2 – the Mediator now intercepts the ENTER key event and performs the uppercasing behaviour in its onSubmit() handler.

Steps:

1. Add the puremvc import statements
2. Add a Mediator class
3. Move the binding and onSubmit method out of the AppFrame and into the Mediator class
4. Have the AppFrame create a PureMVC facade and register a mediator instance with the facade

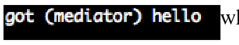
Bold text indicates the new code.

```
import wx

import puremvc.interfaces #ADD
import puremvc.patterns.mediator #ADD

class MyForm(wx.Panel):

    def __init__(self, parent):
        wx.Panel.__init__(self,parent,id=3)
        self.inputFieldTxt = wx.TextCtrl(self, -1, size=(170,-1), pos=(5, 10), style=wx.TE_PROCESS_ENTER)

class MyFormMediator(puremvc.patterns.mediator.Mediator, puremvc.interfaces.IMediator): #ADD CLASS
    NAME = 'MyFormMediator'

    def __init__(self, viewComponent):
        super(MyFormMediator, self).__init__(MyFormMediator.NAME, viewComponent)

        self.viewComponent.Bind(wx.EVT_TEXT_ENTER, self.onSubmit, self.viewComponent.inputFieldTxt)

    def listNotificationInterests(self):
        return []

    def handleNotification(self, notification):
        pass

    def onSubmit(self, evt):
        mydata = self.viewComponent.inputFieldTxt.GetValue()
        print "got (mediator)", mydata
        self.viewComponent.inputFieldTxt.SetValue(mydata.upper())

class AppFrame(wx.Frame):
    myForm = None

    def __init__(self):
        wx.Frame.__init__(self,parent=None, id=-1, title="Refactoring to PureMVC",size=(200,100))
        self.myForm = MyForm(self)

        self.mvcfacade = puremvc.patterns.facade.Facade.getInstance()  #ADD
        self.mvcfacade.registerMediator(MyFormMediator(self.myForm ))  #ADD
~~self.myForm.Bind(wx.EVT_TEXT_ENTER, self.onSubmit, self.myForm.inputFieldTxt) #DELETE~~

~~ def onSubmit(self, evt):~~
~~ mydata = self.myForm.inputFieldTxt.GetValue()~~
~~ print "got", mydata~~
~~ self.myForm.inputFieldTxt.SetValue(mydata.upper())~~

class WxApp(wx.App):
    appFrame = None

    def OnInit(self):
        self.appFrame = AppFrame()
        self.appFrame.Show()
        return True

if __name__ == '__main__':
    wxApp = WxApp(redirect=False)
    wxApp.MainLoop()
```

So now, instead of our AppFrame class binding to the ENTER key event and handling it via an onSubmit method, the mediator now does this – binding to the ENTER event in the constructor of the mediator.

All the AppFrame does now is create a PureMVC facade and register a mediator instance with the facade. Notice that we pass the GUI object, in this case the form, to the mediator as a parameter to the mediator's constructor – the GUI object is referred to by the mediator as the viewComponent.

The application now behaves exactly as before, except we now get the following diagnostic message:

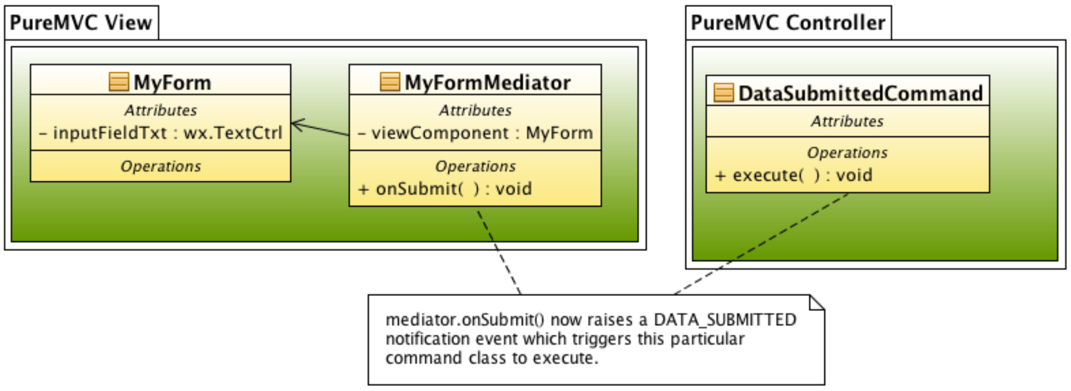


```
got (mediator) hello
```

which proves the mediator is active. Of course this step doesn't really buy us any functionality yet, but at least we are on the road...

Step 3. Move the business logic into a Command class

Let's now move the behaviour (changing text to uppercase) out of the mediator and into a proper command class. This makes the mediator dumber – which is good, as all we want the mediator to do is look after the GUI, not house application logic.



Step 3 – the uppercasing behaviour is moved to the command class.

Steps:

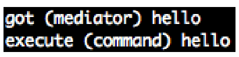
1. Import the puremvc command import
2. Create a command class `DataSubmittedCommand` and implement execute
3. Move the logic for updating the GUI display with uppercase text - out of the mediator and into the command object's execute method. The mediator now simply raises a DATA_SUBMITTED notification
4. Define our own facade class called `AppFacade`and
    1. Define the DATA_SUBMITTED message
    2. Override getInstance() to implement the singleton design pattern (just return a new `AppFacade()`)
    3. Override initializeController() registering the DataSubmittedCommand class, associating it with the DATA_SUBMITTED message. In other words, whenever the DATA_SUBMITTED message is raised, DataSubmittedCommand.**execute**() is called.
5. Use our new concrete Facade rather than the default Facade in the AppFrame class startup code.

```
import wx

import puremvc.interfaces
import puremvc.patterns.mediator
import puremvc.patterns.command #ADD

class MyForm(wx.Panel):

    def __init__(self, parent):
        wx.Panel.__init__(self,parent,id=3)
        self.inputFieldTxt = wx.TextCtrl(self, -1, size=(170,-1), pos=(5, 10), style=wx.TE_PROCESS_ENTER)

class MyFormMediator(puremvc.patterns.mediator.Mediator, puremvc.interfaces.IMediator):
    NAME = 'MyFormMediator'

    def __init__(self, viewComponent):
        super(MyFormMediator, self).__init__(MyFormMediator.NAME, viewComponent)

        self.viewComponent.Bind(wx.EVT_TEXT_ENTER, self.onSubmit, self.viewComponent.inputFieldTxt)

    def listNotificationInterests(self):
        return []

    def handleNotification(self, notification):
        pass

    def onSubmit(self, evt):
        mydata = self.viewComponent.inputFieldTxt.GetValue()
print"got (mediator)", mydata
 ~~self.viewComponent.inputFieldTxt.SetValue(mydata.upper())~~
        self.sendNotification(AppFacade.DATA_SUBMITTED, mydata, self.viewComponent)  #ADD

class DataSubmittedCommand(puremvc.patterns.command.SimpleCommand, puremvc.interfaces.ICommand): #ADD CLASS
    def execute(self, notification):
        print "execute (command)", notification.getBody()
        mydata = notification.getBody()
        viewComponent = notification.getType()
        viewComponent.inputFieldTxt.SetValue(mydata.upper())

class AppFacade(puremvc.patterns.facade.Facade): #ADD CLASS

    DATA_SUBMITTED = "DATA_SUBMITTED"

    @staticmethod
    def getInstance():
        return AppFacade()

    def initializeController(self):
        super(AppFacade, self).initializeController()

        super(AppFacade, self).registerCommand(AppFacade.DATA_SUBMITTED, DataSubmittedCommand)

class AppFrame(wx.Frame):
    myForm = None
    mvcfacade = None  #ADD

    def __init__(self):
        wx.Frame.__init__(self,parent=None, id=-1, title="Refactoring to PureMVC",size=(200,100))
        self.myForm = MyForm(self)

 ~~self.mvcfacade = puremvc.patterns.facade.Facade.getInstance()~~
        self.mvcfacade = AppFacade.getInstance()  #ADD
        self.mvcfacade.registerMediator(MyFormMediator(self.myForm ))

class WxApp(wx.App):
    appFrame = None

    def OnInit(self):
        self.appFrame = AppFrame()
        self.appFrame.Show()
        return True

if __name__ == '__main__':
    wxApp = WxApp(redirect=False)
    wxApp.MainLoop()
```

Notice that we had to create our own concrete Facade class instead of merely creating an instance of the default Facade class that PureMVC provides. This is because we now want to start defining our own messages (e.g. DATA_SUBMITTED).

Also notice (and this is the crux of this refactoring step) that the mediator no longer performs the business logic of converting the text to uppercase. Now instead it simply raises a notification DATA_SUBMITTED which the command class then picks up and acts upon.

Again, as with any good refactoring step, the behaviour of the application is exactly the same as before except for our diagnostic message which now emits:



```
got (mediator) hello
execute (command) hello
```

Step 4. Move the ability to access the GUI out of the Command class

Hey - isn't the command class doing too much now? Notice, in the previous refactoring step, that the command class is not only converting the text to upercase (our simple business logic), but is also getting a bit too big for its boots – it is also stuffing the result back into the gui itself – altogether bypassing the meditor. We are actually encouraging this because we pass both the gui textfield data and a reference to the textfield as part of the notification message. This gives the command class direct access to the gui. The command class should really simply raise a notification and let the mediator do what it was designed to do, and stuff the uppercase text into the appropriate part of the GUI. Let's now do this.

Let's now have the command raise a notification message after it has done its work, and let the *mediator* look after putting the result back into the gui.

Steps:

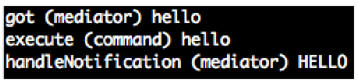
1. Create a new message type **`DATA_CHANGED`** in the facade - don't register it against a command since it is simply a message that will be listened for by the existing meditor. In PureMVC, message notifications can be associated with the triggering of commands or simply be listened for by mediators.
2. Add the message **`DATA_CHANGED`** to the list of messages the mediator is interested in viz.
```
def listNotificationInterests(self):
return [ AppFacade.DATA_CHANGED ]
```
3. Inside the mediator's handleNotification method, check for the message matching **`DATA_CHANGED`** and move the logic that updates the gui in here.
4. We stop passing the meditor's view component as part of the message to the command – the command class doesn't need that reference to the gui anymore.

Ironically the code that updates the GUI started in wxapp, then moved to the meditor, then into the command class, then end up now back in the meditor again! What sort of refactoring is this!?

Well the point is that we have separated two aspects of that code - the uppercase logic is now correctly in the command class and the updating of the GUI correctly in the mediator class. The roles are being performed by the correct classes. The meditor is the only one that knows about the intricacies of the GUI. And its all notification message driven and nicely de-coupled – the command class is triggered in response to an abstract notification `DATA_SUBMITTED` and sends the result back to the mediator using another abstract notification `DATA_CHANGED`. PureMVC is working as intended (though we don't have a proper model yet).

```
import wx

import puremvc.interfaces
import puremvc.patterns.mediator
import puremvc.patterns.command

class MyForm(wx.Panel):

    def __init__(self, parent):
        wx.Panel.__init__(self,parent,id=3)
        self.inputFieldTxt = wx.TextCtrl(self, -1, size=(170,-1), pos=(5, 10), style=wx.TE_PROCESS_ENTER)

class MyFormMediator(puremvc.patterns.mediator.Mediator, puremvc.interfaces.IMediator):
    NAME = 'MyFormMediator'

    def __init__(self, viewComponent):
        super(MyFormMediator, self).__init__(MyFormMediator.NAME, viewComponent)

        self.viewComponent.Bind(wx.EVT_TEXT_ENTER, self.onSubmit, self.viewComponent.inputFieldTxt)

    def listNotificationInterests(self):
        return [ AppFacade.DATA_CHANGED ]   #ADD

    def handleNotification(self, notification):
        if notification.getName() == AppFacade.DATA_CHANGED:                    #ADD
            print "handleNotification (mediator)", notification.getBody()     #ADD
            mydata = notification.getBody()                                  #ADD
            self.viewComponent.inputFieldTxt.SetValue(mydata)                  #ADD

    def onSubmit(self, evt):
        mydata = self.viewComponent.inputFieldTxt.GetValue()
        print "got (mediator)", mydata
        self.sendNotification(AppFacade.DATA_SUBMITTED, mydata, self.viewComponent)

class DataSubmittedCommand(puremvc.patterns.command.SimpleCommand, puremvc.interfaces.ICommand):
    def execute(self, notification):
        print "execute (command)", notification.getBody()
        mydata = notification.getBody()
~~ viewComponent = notification.getType()~~
~~ viewComponent.inputFieldTxt.SetValue(mydata.upper())~~
        self.sendNotification(AppFacade.DATA_CHANGED, mydata.upper())   #ADD


class AppFacade(puremvc.patterns.facade.Facade):

    DATA_SUBMITTED = "DATA_SUBMITTED"
    DATA_CHANGED = "DATA_CHANGED"   #ADD

    @staticmethod
    def getInstance():
        return AppFacade()

    def initializeController(self):
        super(AppFacade, self).initializeController()

        super(AppFacade, self).registerCommand(AppFacade.DATA_SUBMITTED, DataSubmittedCommand)

class AppFrame(wx.Frame):
    myForm = None
    mvcfacade = None

    def __init__(self):
        wx.Frame.__init__(self,parent=None, id=-1, title="Refactoring to PureMVC",size=(200,100))
        self.myForm = MyForm(self)

        self.mvcfacade = AppFacade.getInstance()
        self.mvcfacade.registerMediator(MyFormMediator(self.myForm ))

class WxApp(wx.App):
    appFrame = None

    def OnInit(self):
        self.appFrame = AppFrame()
        self.appFrame.Show()
        return True

if __name__ == '__main__':
    wxApp = WxApp(redirect=False)
    wxApp.MainLoop()
```

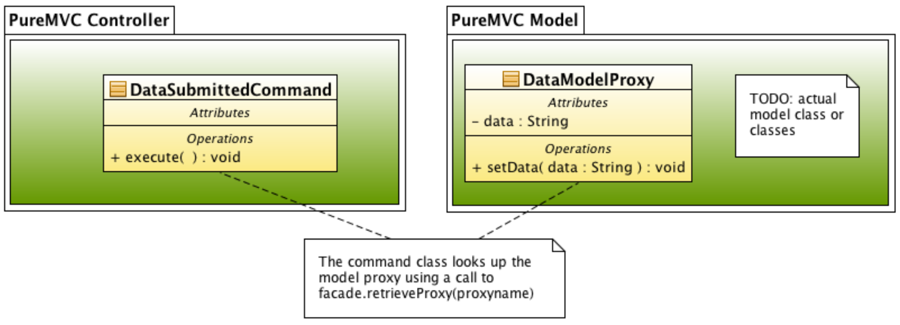
Our diagnostic now shows how the flow of execution moves from the GUI to the mediator, to the command, and then back to the mediator again.



```
got (mediator) hello
execute (command) hello
handleNotification (mediator) HELLO
```

Step 5. Add a Model

You've been waiting for this step, I know. Let's add a proper model.


PureMVC Controller
DataSubmittedCommand
Attributes
Operations
+ execute( ) : void
PureMVC Model
DataModelProxy
Attributes
- data : String
Operations
+ setData(data : String) : void
TODO: actual model class or classes
The command class looks up the model proxy using a call to facade.retrieveProxy(proxyname)

Adding a model to our architecture.

In the PureMVC framework's way of looking at things, the Model should be wrapped by a “Model Proxy” class. Just like the mediator wraps and looks after the GUI, the proxy wraps and looks after the model. In this refactoring step, we are going to add the Proxy class and use it as the model.

From the command class point of view, instead of putting the result of the business logic straight back into the GUI, the command class now writes the uppercase string to the model proxy. The model proxy then sends out a notification that the model data has changed - which the mediator picks up and acts on by updating the form inputfield control/widget. Basically we have a complete PureMVC architecture functioning now (see sequence diagram below).

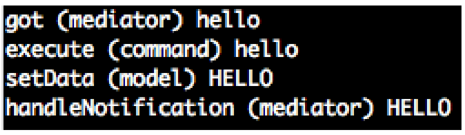
Steps:

1. Import the proxy namespace
2. Define a class **`DataModelProxy`** which will hold the data (the string we are entering in and upper-casing). The proxy class sends a notification whenever the data changes, enabling e.g. the mediator to update the gui.
3. Create and register the the Proxy class in the startup phase
4. Change the command so that it looks up the model proxy and updates it with the uppercase string.
5. Mediator is told about the data that changes by the model proxy raising a DATA_CHANGED message. In response, the meditor takes the data and puts it in the GUI.

When the mediator has been notified by the model - how does the mediator get access to the model proxy data? Well it can be given a reference to the data proxy so that it can get the data itself, or the notification message can contain the actual data (remember that notification messages have two additional parameters after the message name – getBody() and getType() and it is up to you what you put there. Finally, the Mediator is within its rights to look up the data proxy itself if it wants using the **`self.facade.retrieveProxy(...)`** lookup technique.

```
import wx

import puremvc.interfaces
import puremvc.patterns.mediator
import puremvc.patterns.command
import puremvc.patterns.proxy #ADD

class MyForm(wx.Panel):

    def __init__(self, parent):
        wx.Panel.__init__(self,parent,id=3)
        self.inputFieldTxt = wx.TextCtrl(self, -1, size=(170,-1), pos=(5, 10), style=wx.TE_PROCESS_ENTER)

class MyFormMediator(puremvc.patterns.mediator.Mediator, puremvc.interfaces.IMediator):
    NAME = 'MyFormMediator'

    def __init__(self, viewComponent):
        super(MyFormMediator, self).__init__(MyFormMediator.NAME, viewComponent)

        self.viewComponent.Bind(wx.EVT_TEXT_ENTER, self.onSubmit, self.viewComponent.inputFieldTxt)

    def listNotificationInterests(self):
        return [ AppFacade.DATA_CHANGED ]

    def handleNotification(self, notification):
        if notification.getName() == AppFacade.DATA_CHANGED:
            print "handleNotification (mediator)", notification.getBody()
            mydata = notification.getBody()
            self.viewComponent.inputFieldTxt.SetValue(mydata)

    def onSubmit(self, evt):
        mydata = self.viewComponent.inputFieldTxt.GetValue()
        print "got (mediator)", mydata
        self.sendNotification(AppFacade.DATA_SUBMITTED, mydata, self.viewComponent)
```

```
class DataSubmittedCommand(puremvc.patterns.command.SimpleCommand, puremvc.interfaces.ICommand):
    def execute(self, notification):
        print "execute (command)", notification.getBody()
        mydata = notification.getBody()
~~self.sendNotification(AppFacade.DATA_CHANGED, mydata.upper())~~
        self.datamodelProxy = self.facade.retrieveProxy(DataModelProxy.NAME)  #ADD
        self.datamodelProxy.setData(mydata.upper())                          #ADD


class DataModelProxy(puremvc.patterns.proxy.Proxy): #ADD

    NAME = "DataModelProxy"

    def __init__(self):
        super(DataModelProxy, self).__init__(DataModelProxy.NAME, [])
        self.data = ""

    def setData(self, data):
        self.data = data
        print "setData (model)", data
        self.sendNotification(AppFacade.DATA_CHANGED, self.data)

class AppFacade(puremvc.patterns.facade.Facade):

    DATA_SUBMITTED = "DATA_SUBMITTED"
    DATA_CHANGED = "DATA_CHANGED"

    @staticmethod
    def getInstance():
        return AppFacade()

    def initializeController(self):
        super(AppFacade, self).initializeController()

        super(AppFacade, self).registerCommand(AppFacade.DATA_SUBMITTED, DataSubmittedCommand)


class AppFrame(wx.Frame):
    myForm = None
    mvcfacade = None

    def __init__(self):
        wx.Frame.__init__(self,parent=None, id=-1, title="Refactoring to PureMVC",size=(200,100))
        self.myForm = MyForm(self)

        self.mvcfacade = AppFacade.getInstance()
        self.mvcfacade.registerMediator(MyFormMediator(self.myForm ))
        self.mvcfacade.registerProxy(DataModelProxy())   #ADD

class WxApp(wx.App):
    appFrame = None

    def OnInit(self):
        self.appFrame = AppFrame()
        self.appFrame.Show()
        return True

if __name__ == '__main__':
    wxApp = WxApp(redirect=False)
    wxApp.MainLoop()
```

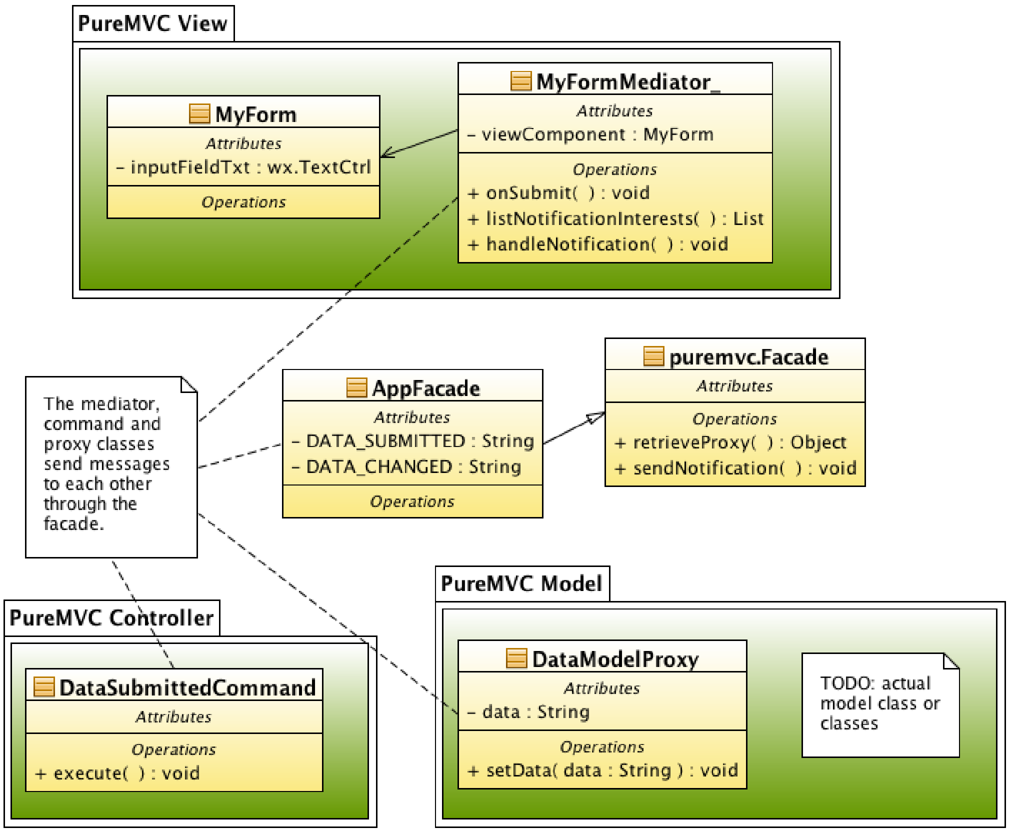


```
got (mediator) hello
execute (command) hello
setData (model) HELLO
handleNotification (mediator) HELLO
```

Intermission – Taking Stock

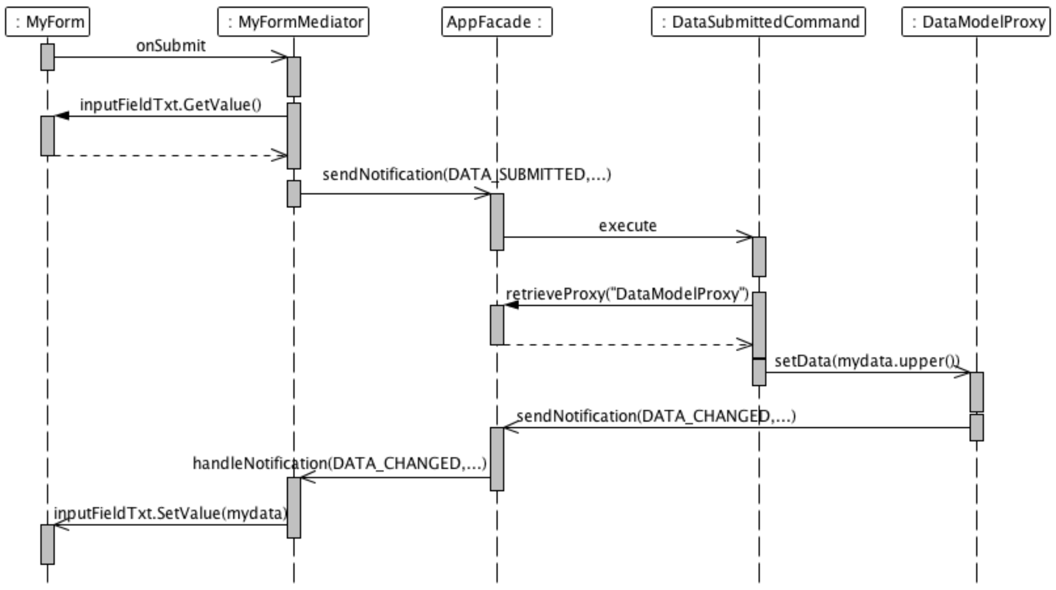
At this stage the PureMVC architecture is fully operational.

Here is the class diagram:



Class diagram of our application thus far.

Here is the sequence diagram:


: MyForm
: MyFormMediator
AppFacade :
: DataSubmittedCommand
: DataModelProxy
onSubmit
inputFieldTxt.GetValue()
sendNotification(DATA_SUBMITTED,...)
execute
retrieveProxy("DataModelProxy")
setData(mydata.upper())
sendNotification(DATA_CHANGED,...)
handleNotification(DATA_CHANGED,...)
inputFieldTxt.SetValue(mydata)

Sequence diagram of what happens after the user types in some text and hits ENTER in the GUI.

Note that the proxy, mediator and command classes actually have a very convenient **sendNotification** method *on themselves* (rather than having a reference to the facade's **sendNotification** method as illustrated in the above sequence diagram). In actuality, these self.**sendNotification** methods ultimately get routed to the facade anyway, so the sequence diagram above is essentially correct.

"Aha" moment – several roles have been distilled out of two hacky lines of code

Its interesting that we have distilled three roles out of the original, two simple lines of code:

```
        mydata = self.myForm.inputFieldTxt.GetValue()
        self.myForm.inputFieldTxt.SetValue(mydata.upper())
```

The above code gets some user input text, converts it to uppercase and stuffs it back into the GUI. After our PureMVC refactoring steps, those roles are now spread out across various classes and a major architectural revamp. The roles are:

- Getting the data in and out of the GUI (mediator)
- Converting the text to uppercase (command)

And we have added in an additional role which the original code didn't do

- Holding and storing an independent representation of the text being displayed (model proxy)

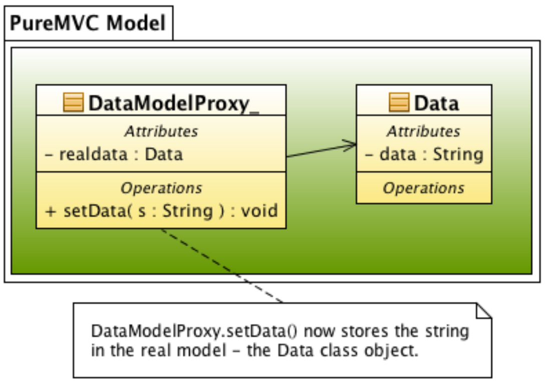
Sure, we've added complexity – boy have we ever. But now we have something amazingly scalable. We won't be scaling this application up in this tutorial, however there are a couple more refactoring steps to do in order to tidy up a few things.

Step 6. Add a real Model behind the Model Proxy and Initialise the Model

Let's now add a proper model that the model proxy wraps. In our simple example so far, this step seems redundant as it may seem easier to keep the string stored in the model proxy. Why not make the model proxy be the model?

In more complex projects, model proxies can become properly useful – possibly representing a number of model objects under the one model proxy, or even returning result sets due to the underlying data model being a database.

Another reason to separate the ModelProxy from the Model is that the ModelProxy class is free to “play ball” with the PureMVC framework e.g. sending out various notification messages whenever data gets changed etc. whereas a Model class is typically oblivious to such things (and probably should remain so). In our example, the DataModelProxy sends out a ï»¿DATA_CHANGED notification – both in the constructor, and also whenever setData() is called.


PureMVC Model
DataModelProxy_
Attributes
- realdata : Data
Operations
+ setData(s : String) : void
Data
Attributes
- data : String
Operations
DataModelProxy.setData() now stores the string in the real model - the Data class object.

Adding a real model behind the model proxy.

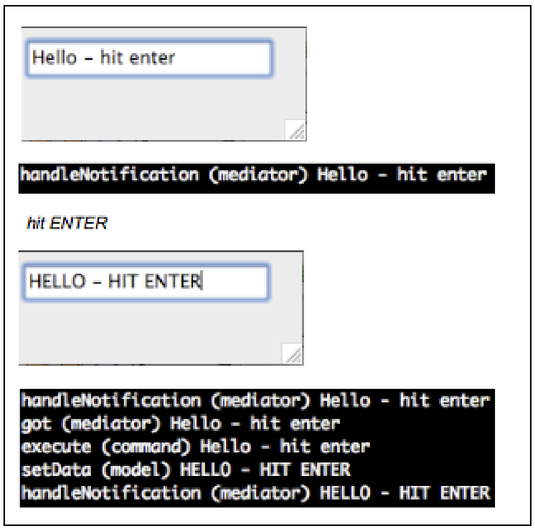
Steps:

1. Add a proper model class “Data” and have the DataModelProxy wrap it.
2. Initialise the model with an initial string - and watch the default text appear in the GUI when the application starts up!

```
import wx

import puremvc.interfaces
import puremvc.patterns.mediator
import puremvc.patterns.command
import puremvc.patterns.proxy

class MyForm(wx.Panel):

    def __init__(self, parent):
        wx.Panel.__init__(self,parent,id=3)
        self.inputFieldTxt = wx.TextCtrl(self, -1, size=(170,-1), pos=(5, 10), style=wx.TE_PROCESS_ENTER)

class MyFormMediator(puremvc.patterns.mediator.Mediator, puremvc.interfaces.IMediator):
    NAME = 'MyFormMediator'

    def __init__(self, viewComponent):
        super(MyFormMediator, self).__init__(MyFormMediator.NAME, viewComponent)

        self.viewComponent.Bind(wx.EVT_TEXT_ENTER, self.onSubmit, self.viewComponent.inputFieldTxt)

    def listNotificationInterests(self):
        return [ AppFacade.DATA_CHANGED ]

    def handleNotification(self, notification):
        if notification.getName() == AppFacade.DATA_CHANGED:
            print "handleNotification (mediator)", notification.getBody()
            mydata = notification.getBody()
            self.viewComponent.inputFieldTxt.SetValue(mydata)

    def onSubmit(self, evt):
        mydata = self.viewComponent.inputFieldTxt.GetValue()
        print "got (mediator)", mydata
        self.sendNotification(AppFacade.DATA_SUBMITTED, mydata, self.viewComponent)

class DataSubmittedCommand(puremvc.patterns.command.SimpleCommand, puremvc.interfaces.ICommand):
    def execute(self, notification):
        print "execute (command)", notification.getBody()
        mydata = notification.getBody()
        self.datamodelProxy = self.facade.retrieveProxy(DataModelProxy.NAME)
        self.datamodelProxy.setData(mydata.upper())

class DataModelProxy(puremvc.patterns.proxy.Proxy):

    NAME = "DataModelProxy"

    def __init__(self):
        super(DataModelProxy, self).__init__(DataModelProxy.NAME, [])
        ~~self.data = ""~~
        self.realdata = Data()   #ADD
        self.sendNotification(AppFacade.DATA_CHANGED, self.realdata.data)   #ADD

    def setData(self, data):
        ~~self.data = data~~
        self.realdata.data = data   #ADD
        print "setData (model)", data
        ~~self.sendNotification(AppFacade.DATA_CHANGED, self.data)~~
        self.sendNotification(AppFacade.DATA_CHANGED, self.realdata.data)   #ADD

class Data: #ADD CLASS
    def __init__(self):
        self.data = "Hello - hit enter"

class AppFacade(puremvc.patterns.facade.Facade):

    DATA_SUBMITTED = "DATA_SUBMITTED"
    DATA_CHANGED = "DATA_CHANGED"

    @staticmethod
    def getInstance():
        return AppFacade()

    def initializeController(self):
        super(AppFacade, self).initializeController()

        super(AppFacade, self).registerCommand(AppFacade.DATA_SUBMITTED, DataSubmittedCommand)


class AppFrame(wx.Frame):
    myForm = None
    mvcfacade = None

    def __init__(self):
        wx.Frame.__init__(self,parent=None, id=-1, title="Refactoring to PureMVC",size=(200,100))
        self.myForm = MyForm(self)

        self.mvcfacade = AppFacade.getInstance()
        self.mvcfacade.registerMediator(MyFormMediator(self.myForm ))
        self.mvcfacade.registerProxy(DataModelProxy())

class WxApp(wx.App):
    appFrame = None

    def OnInit(self):
        self.appFrame = AppFrame()
        self.appFrame.Show()
        return True

if __name__ == '__main__':
    wxApp = WxApp(redirect=False)
    wxApp.MainLoop()
```

The Application Initialisation Sequence – Getting intitial Model data into the GUI

A nice effect here is that the string `"Hello - hit enter"` in the Data class magically appears in the GUI when the application starts up. How cool – some of this PureMVC architecture is starting to pay off. How does this happen? Well, notice that the DataModelProxy sends out a DATA_CHANGED notification in its constructor (when it is also creating the instance of the real model Data class). The mediator intercepts this notification and displays the string in the GUI form.


Hello - hit enter

```
handleNotification (mediator) Hello - hit enter
```

hit ENTER

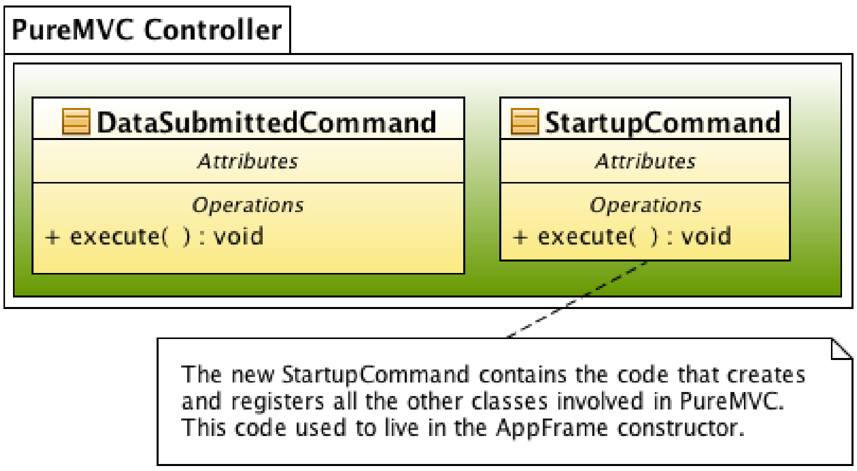
HELLO – HIT ENTER

```
handleNotification (mediator) Hello - hit enter
got (mediator) Hello - hit enter
execute (command) Hello - hit enter
setData (model) HELLO - HIT ENTER
handleNotification (mediator) HELLO - HIT ENTER
```

Two screenshots of our PureMVC wxPython application,
showing how the model data magically appears in the GUI form on application startup.

Step 7. Move the startup code into a startup command class

Personally I'm not convinced this step is really necessary - however this step follows the convention on how PureMVC applications are put together.


PureMVC Controller
DataSubmittedCommand
Attributes
Operations
+ execute( ) : void
StartupCommand
Attributes
Operations
+ execute( ) : void
The new StartupCommand contains the code that creates and registers all the other classes involved in PureMVC. This code used to live in the AppFrame constructor.

The new startup command class.

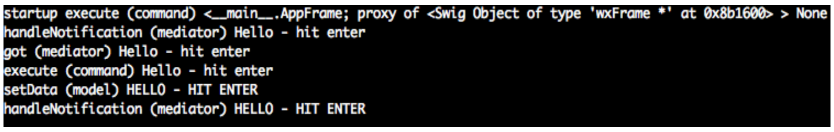
Moving most of the startup code into its own command class may have the benefit of organising your code a little more, at the cost of complexity – I mean, we already have a nice place for constructing classes etc in the constructor of the AppFrame class. Even if we move some startup code into its own startup command class, we still need to create the form and the facade in AppFrame, so why not keep all the startup code in the one place? On the other hand, by moving what we can into the startup command, we may be helping decouple the AppFrame startup code from the knowledge of all the other clases involved in your framework e.g. mediators, commands etc. Certainly in a Flex application where namespaces are tightly controlled, or even a more serious python application where namespaces are more of an issue, having a startup comand class is a good thing.

1. Move the startup code into a startup command class

2. Change our initialisation to invoke the startup command via a notification message

```
import wx

import puremvc.interfaces
import puremvc.patterns.mediator
import puremvc.patterns.command
import puremvc.patterns.proxy

class MyForm(wx.Panel):

    def __init__(self, parent):
        wx.Panel.__init__(self,parent,id=3)
        self.inputFieldTxt = wx.TextCtrl(self, -1, size=(170,-1), pos=(5, 10), style=wx.TE_PROCESS_ENTER)

class MyFormMediator(puremvc.patterns.mediator.Mediator, puremvc.interfaces.IMediator):
    NAME = 'MyFormMediator'

    def __init__(self, viewComponent):
        super(MyFormMediator, self).__init__(MyFormMediator.NAME, viewComponent)

        self.viewComponent.Bind(wx.EVT_TEXT_ENTER, self.onSubmit, self.viewComponent.inputFieldTxt)

    def listNotificationInterests(self):
        return [ AppFacade.DATA_CHANGED ]

    def handleNotification(self, notification):
        if notification.getName() == AppFacade.DATA_CHANGED:
            print "handleNotification (mediator)", notification.getBody()
            mydata = notification.getBody()
            self.viewComponent.inputFieldTxt.SetValue(mydata)

    def onSubmit(self, evt):
        mydata = self.viewComponent.inputFieldTxt.GetValue()
        print "got (mediator)", mydata
        self.sendNotification(AppFacade.DATA_SUBMITTED, mydata, self.viewComponent)

class DataSubmittedCommand(puremvc.patterns.command.SimpleCommand, puremvc.interfaces.ICommand):
    def execute(self, notification):
        print "execute (command)", notification.getBody()
        mydata = notification.getBody()
        self.datamodelProxy = self.facade.retrieveProxy(DataModelProxy.NAME)
        self.datamodelProxy.setData(mydata.upper())

class StartupCommand(puremvc.patterns.command.SimpleCommand, puremvc.interfaces.ICommand): #ADD CLASS
    def execute(self, notification):
        print "startup execute (command)", notification.getBody(), notification.getType()
        wxapp = notification.getBody()

        self.facade.registerMediator(MyFormMediator(wxapp.myForm))
        self.facade.registerProxy(DataModelProxy())

class DataModelProxy(puremvc.patterns.proxy.Proxy):

    NAME = "DataModelProxy"

    def __init__(self):
        super(DataModelProxy, self).__init__(DataModelProxy.NAME, [])
        self.realdata = Data()
        self.sendNotification(AppFacade.DATA_CHANGED, self.realdata.data)

    def setData(self, data):
        self.realdata.data = data
        print "setData (model)", data
        self.sendNotification(AppFacade.DATA_CHANGED, self.realdata.data)

class Data:
    def __init__(self):
self.data="Hello - hit enter"

class AppFacade(puremvc.patterns.facade.Facade):

    STARTUP = "STARTUP"   #ADD
    DATA_SUBMITTED = "DATA_SUBMITTED"
    DATA_CHANGED = "DATA_CHANGED"

    @staticmethod
    def getInstance():
        return AppFacade()

    def initializeController(self):
        super(AppFacade, self).initializeController()

        super(AppFacade, self).registerCommand(AppFacade.STARTUP, StartupCommand)   #ADD
        super(AppFacade, self).registerCommand(AppFacade.DATA_SUBMITTED, DataSubmittedCommand)

    def startup(self, app):   #ADD METHOD
        self.sendNotification( AppFacade.STARTUP, app )
```

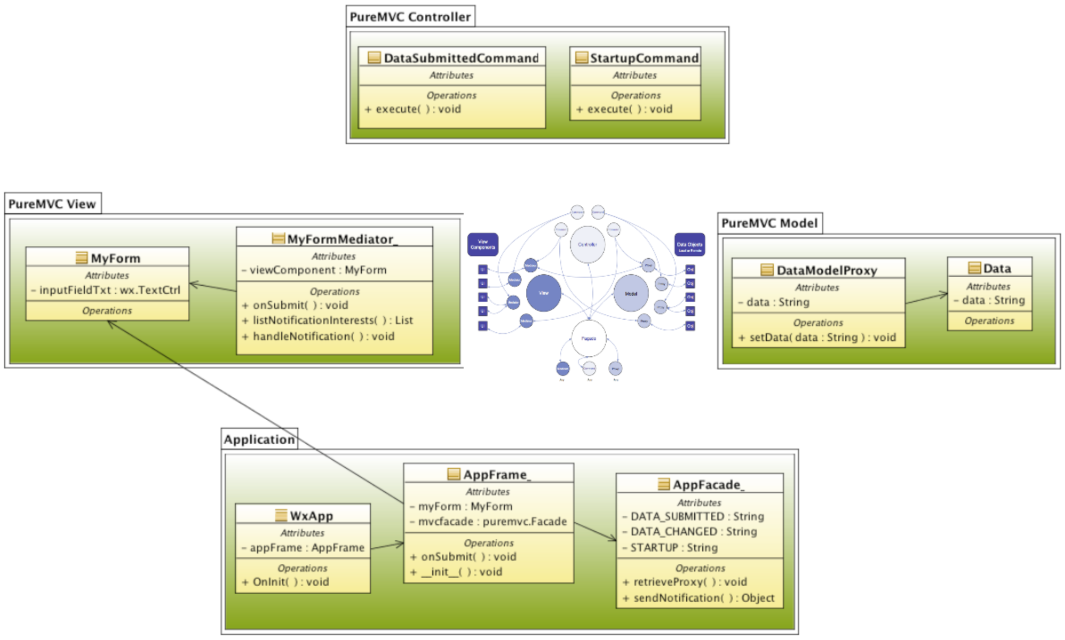
```
class AppFrame(wx.Frame):
    myForm = None
    mvcfacade = None

    def __init__(self):
        wx.Frame.__init__(self,parent=None, id=-1, title="Refactoring to PureMVC",size=(200,100))
        self.myForm = MyForm(self)

        self.mvcfacade = AppFacade.getInstance()
~~ self.mvcfacade.registerMediator(MyFormMediator(self.myForm ))~~
~~ self.mvcfacade.registerProxy(DataModelProxy())~~
        self.mvcfacade.startup(self)   #ADD

class WxApp(wx.App):
    appFrame = None

    def OnInit(self):
        self.appFrame = AppFrame()
        self.appFrame.Show()
        return True

if __name__ == '__main__':
    wxApp = WxApp(redirect=False)
    wxApp.MainLoop()
```



```
startup execute (command) <__main__.AppFrame; proxy of <Swig Object of type 'wxFrame *' at 0x8b1600> > None
handleNotification (mediator) Hello - hit enter
got (mediator) Hello - hit enter
execute (command) Hello - hit enter
setData (model) HELLO - HIT ENTER
handleNotification (mediator) HELLO - HIT ENTER
```

Conclusion

Here is the final UML.



Final UML of our example

You may notice that there are hardly any dependencies between classes. This is because classes communicate with each other via the PureMVC message notification system. This takes the form of a string message broadcast to the "world" e.g. e.g. **facade.sendNotification("DATA_CHANGED", notificationMsg)** - the sender doesn't really care who handles it. Or it can take the form of looking up model instances, again by string references e.g. **facade.retreiveProxy("datamodelproxy1")**. We end up with a very decoupled design. Nice.

Too much Complexity?

This series of refactorings has shown how you can really complicate your application by utilising the PureMVC framework. :-)

Seriously, on the positive side, you now have a scalable architecture where you can add more notification messages, more mediators, more commands – and everything will fit together. I think that a framework has value not just because of what it does, but rather because it guides you in how to structure your application – how to name your classes and where to put them, and how they play together.

If your application is simple it may not be worthwhile incorporating PureMVC, however hopefully this tutorial has shown you that it is not that hard to start with the PureMVC approach. I may have inspired you to use PureMVC with even your simple apps!

Finally, this tutorial has shown that you can hack something up in wxPython and then later, switch to PureMVC when you need to, in a step by step way.

Source code for all the steps is available here.

If you prefer a java example walkthrough, click here.

-Andy Bulka
March 2009

P.S. This blog posting was presented as a talk to the Melbourne Patterns Group on the 1st April, 2009.

Comments

Posted by Arun George on Feb 15th, 2012
Great article on pureMVC and its applicalbility through a simple example. This is cool. A very simple example which explains the complexity :) .

Posted by Byron Harris on May 25th, 2012
Thanks for providing this example. It helps me in applying the concepts in the PureMVC documentation to Python and wxPython.

A few minor points about the example code:

- In the DataSubmittedCommand.execute(), it's not necessary to add attribute datamodelProxy since Command classes are meant to be stateless. Therefore you can just make datamodelProxy a method variable.

- In StartupCommand.execute(), the variable named wxapp is inappropriate named. It's actually an AppFrame instance that is passed.

Posted by T.Javed on Sep 25th, 2012
Thank You so much!

I had read all the pureMVC documentation but was finding it hard to grasp it until I came across this nice little tutorial. It helped me a lot to clearly understand the pureMVC concept and implementation.

cheers!

Posted by Demolishun on Oct 20th, 2012

Putting logic in the controller does not make sense to me:

http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model
http://stackoverflow.com/questions/235233/asp-net-mvc-should-business-logic-exist-in-controllers

I am starting down the path to using pureMVC in my projects, but your example is confusing. It seems to me that the controller is supposed to be light weight and act as glue. What is the reasoning for putting the business logic in the controller and not the model?

When I began to understand MVC it clicked in my head that the model would be independent and could be swapped out just like the view. I am also a little concerned about the AppFrame knowing about the Facade. This makes it harder to swap out the view. Or am I looking at this the wrong way?