- Home
- Design Patterns
- Blog
- Products
- Sitemap
- Contact

AndyPatterns

Refactoring to PureMVC - Java version

Here is a Java version of the tutorial I originally wrote in wxPython. It leaves out most of the commentary and concentrates on the code, so please read the original tutorial for more detail about the meaning of the refactoring steps.

I have used the same class names and the logic is identical (except for the parameters to the notification message DATA_SUBMITTED, which I have swapped around due to the fact that in the Java PureMVC framework the getBody() parameter is of type object whereas in dynamic Python the type of this paramter is more flexible – more explanation is found below).

Step 1. Create a Basic Java GUI application form

Create a Java GUI application. Either hand code it or e.g. use Netbeans to kick start you. Then drop a

```
javax.swing.JTextField jTextField1
```

to the form. Don't bother adding any behaviour yet, lets get cracking with the refactoring.

Step 2. Import PureMVC and add a mediator

Here is the mediator.

```
/*
 * MyFormMediator.java
 */
package org.andy;

import org.puremvc.java.patterns.mediator.Mediator;

import org.puremvc.java.interfaces.INotification;
import javapuremvcminimal01.JavaPureMVCMinimal01Form;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyFormMediator extends Mediator implements ActionListener {

    public static final String NAME = "MyFormMediator";

    public MyFormMediator(JavaPureMVCMinimal01Form viewComponent)
    {
        super(NAME, null);
        setViewComponent(viewComponent);
        viewComponent.inputFieldTxt.addActionListener(this);
    }

    public void actionPerformed(ActionEvent evt) {
        JavaPureMVCMinimal01Form form = (JavaPureMVCMinimal01Form) viewComponent;

        form.inputFieldTxt.setText(form.inputFieldTxt.getText().toUpperCase());
    }
}
```

Here is the application startup code, which was generated automatically by Netbeans 6.5 and the startup method was modified to create the PureMVC facade and mediator.

```
/*
 * JavaPureMVCMinimal01App.java
 */

package javapuremvcminimal01;

import org.jdesktop.application.Application;
import org.jdesktop.application.SingleFrameApplication;

import org.puremvc.java.patterns.facade.Facade;
import org.andy.MyFormMediator;

/**
 * The main class of the application.
 * Based on a Netbeans 6.5 template.
 */
public class JavaPureMVCMinimal01App extends SingleFrameApplication {
    public JavaPureMVCMinimal01Form myForm;

    /**
     * At startup create and show the main frame of the application.
     */
    @Override protected void startup() {
        myForm = new JavaPureMVCMinimal01Form(this);
        Facade mvcfacade = Facade.getInstance();
        mvcfacade.registerMediator(new MyFormMediator(myForm));
        show(myForm);
    }
    …
    …
}
```

Since we need access to the textField in the swing form generated by netbeans, we declare a public variable for ourselves. Netbeans seems to make all the gui elements private – so we are fighting that.

```
/*
 * JavaPureMVCMinimal01Form.java
 *  (by default Netbeans names this JavaPureMVCMinimal01View but I renamed it).
 */

package javapuremvcminimal01;

import javax.swing.JFrame;

public class JavaPureMVCMinimal01Form extends FrameView {

    public javax.swing.JTextField inputFieldTxt;  // ADD

    public JavaPureMVCMinimal01Form(SingleFrameApplication app) {
        super(app);

        initComponents();
        inputFieldTxt = jTextField1;  // ADD

        …
        …

}
```

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.

```
/*
 * DataSubmittedCommand.java
 */

package org.andy;

import org.puremvc.java.interfaces.ICommand;
import org.puremvc.java.interfaces.INotification;
import org.puremvc.java.patterns.command.SimpleCommand;

import javapuremvcminimal01.JavaPureMVCMinimal01Form;

public class DataSubmittedCommand extends SimpleCommand implements ICommand {

    public void execute(INotification notification)
    {
        String mydata = (String) notification.getType();
        JavaPureMVCMinimal01Form viewComponent = (JavaPureMVCMinimal01Form) notification.getBody();
        viewComponent.inputFieldTxt.setText(mydata.toUpperCase());
    }
}
```

When the execute method gets called the first thing the execute code should do is decode the notification message for juicy information. Its fairly arbitrary how you use the notification class fields .getBody() and getType(). In the java implementation the only limitation is that getBody() holds an object and getType() a string. Use your own convention on how to use these two parameters, which can be different for each unique notifcation message - just make sure your `sendNotification` sends the right things to match how you are decoding things.

In the example so far, in the notification message DATA_SUBMITTED I use the getBody() field to pass a reference to the viewcomponent, i.e. the form. This is only a temporary situation, we will later change this so that the command simply raises a message and let the mediator deal with the viewcomponent (which is the mediators job). I use the getType() parameter of the notification message to hold the text of the edit control at the time when the use hit ENTER.

Now the mediator has changed a little. It no longer does the work of uppercase, it delegates this entirely to the command class simply by raising a `DATA_SUBMITTED` message:

```
/*
 * MyFormMediator.java
 */
package org.andy;

import org.puremvc.java.patterns.mediator.Mediator;

import org.puremvc.java.interfaces.INotification;
import javapuremvcminimal01.JavaPureMVCMinimal01Form;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyFormMediator extends Mediator implements ActionListener {

    public static final String NAME = "MyFormMediator";

    public MyFormMediator(JavaPureMVCMinimal01Form viewComponent)
    {
        super(NAME, null);

        setViewComponent(viewComponent);
        viewComponent.inputFieldTxt.addActionListener(this);
    }

    public void actionPerformed(ActionEvent evt) {
        JavaPureMVCMinimal01Form form = (JavaPureMVCMinimal01Form) viewComponent;

        ~~form.inputFieldTxt.setText(form.inputFieldTxt.getText().toUpperCase());~~
        this.sendNotification(AppFacade.DATA_SUBMITTED, viewComponent, form.inputFieldTxt.getText());
    }
}
```

Note that another way to send a notification message (instead of `sendNotification`) is:

```
        import org.puremvc.java.patterns.observer.Notification;

        this.facade.notifyObservers(new Notification(AppFacade.DATA_SUBMITTED,
              viewComponent, form.inputFieldTxt.getText()));
```

however this is a bit PureMVC old school and not quite as clean as what we have used.

We also have needed to create our own concrete Facade class so that we can define our own message types.

```
/*
 * AppFacade.java
 */

package org.andy;

import org.puremvc.java.patterns.facade.Facade;

public class AppFacade extends Facade {

    public static final String DATA_SUBMITTED = "DATA_SUBMITTED";

    private static AppFacade instance = null;

    public static AppFacade getInst()
    {
        if (instance == null) {
            instance = new AppFacade();
        }
        return (AppFacade) instance;
    }

    @Override
    protected void initializeController()
    {
        super.initializeController();

        registerCommand(DATA_SUBMITTED, DataSubmittedCommand.class);
    }
}
```

And of course we instantiate our own concrete facade instead of the base class facade:

```
/*
 * JavaPureMVCMinimal01App.java
 */
    …
    …
    @Override protected void startup() {
        myForm = new JavaPureMVCMinimal01Form(this);

        ~~Facade mvcfacade = Facade.getInstance();~~
        Facade mvcfacade = AppFacade.getInst();

        mvcfacade.registerMediator(new MyFormMediator(myForm));
        show(myForm);
    }
```

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

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.

```
/*
 * MyFormMediator.java
 */
package org.andy;

import org.puremvc.java.patterns.mediator.Mediator;

import org.puremvc.java.interfaces.INotification;
import javapuremvcminimal01.JavaPureMVCMinimal01Form;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyFormMediator extends Mediator implements ActionListener {

    public static final String NAME = "MyFormMediator";

    public MyFormMediator(JavaPureMVCMinimal01Form viewComponent)
    {
        super(NAME, null);

        setViewComponent(viewComponent);
        viewComponent.inputFieldTxt.addActionListener(this);
    }

    public void actionPerformed(ActionEvent evt) {
        JavaPureMVCMinimal01Form form = (JavaPureMVCMinimal01Form) viewComponent;

        ~~form.inputFieldTxt.setText(form.inputFieldTxt.getText().toUpperCase());~~
        this.sendNotification(AppFacade.DATA_SUBMITTED, viewComponent, form.inputFieldTxt.getText());
    }

    @Override
    public String[] listNotificationInterests()
    {
        return new String[] {AppFacade.DATA_CHANGED};
    }

    @Override
    public void handleNotification(INotification notification)
    {
        if(notification.getName().equals(AppFacade.DATA_CHANGED))
        {
            System.out.println("handleNotification (mediator) " + notification.getType());

            String mydata = (String) notification.getType();
            JavaPureMVCMinimal01Form form = (JavaPureMVCMinimal01Form) viewComponent;
            form.inputFieldTxt.setText(mydata);
        }
    }
}

/*
 * DataSubmittedCommand.java
 */

package org.andy;

import org.puremvc.java.interfaces.ICommand;
import org.puremvc.java.interfaces.INotification;
import org.puremvc.java.patterns.command.SimpleCommand;

~~import javapuremvcminimal01.JavaPureMVCMinimal01Form;~~

public class DataSubmittedCommand extends SimpleCommand implements ICommand {

    public void execute(INotification notification)
    {
        String mydata = (String) notification.getType();
        ~~JavaPureMVCMinimal01Form viewComponent = (JavaPureMVCMinimal01Form) notification.getBody();~~
        ~~viewComponent.inputFieldTxt.setText(mydata.toUpperCase());~~
        this.sendNotification(AppFacade.DATA_CHANGED, null, mydata.toUpperCase());
    }
}
```

Notice that the mediator is putting the result of the command behaviour (the uppercasing of user entered text) into the gui – the command is not touching the GUI anymore, which explains why the command no longer needs to import the GUI form class `JavaPureMVCMinimal01Form`.

The command class `DataSubmittedCommand` simply raises a notification, passing the changed text as part of the notifcation message (we are using the getType() parameter to hold the uppercased string – though we could have passed it around in the getBody() parameter).

Finally we need to add the new notification message type to the facade:

```
/*
 * AppFacade.java
 */
package org.andy;

import org.puremvc.java.patterns.facade.Facade;

public class AppFacade extends Facade {

    public static final String DATA_SUBMITTED = "DATA_SUBMITTED";
    public static final String DATA_CHANGED = "DATA_CHANGED";
    …
    …
}
```

Step 5. Add a Model

Let's add a proper model. For now we leave out the real model behind the proxy and use the proxy class as both proxy and model.

```
/*
 * DataModelProxy.java
 */
package org.andy;

import org.puremvc.java.patterns.proxy.Proxy;

public class DataModelProxy extends Proxy {

    public static final String NAME = "DataModelProxy";

    private String data;

    public DataModelProxy()
    {
        super(NAME, null);
        this.data = "";
    }

    public void setData(String data) {
        this.data = data;
        System.out.println("setData (model) " + data);
        this.sendNotification(AppFacade.DATA_CHANGED, null, this.data);
    }
}
```

We alter the command class to poke the uppercase string into the model. We then let the model notify the world that the model data has altered.

```
/*
 * DataSubmittedCommand.java
 */

package org.andy;

import org.puremvc.java.interfaces.ICommand;
import org.puremvc.java.interfaces.INotification;
import org.puremvc.java.patterns.command.SimpleCommand;


public class DataSubmittedCommand extends SimpleCommand implements ICommand {

    public void execute(INotification notification)
    {
        String mydata = (String) notification.getType();
        ~~this.sendNotification(AppFacade.DATA_CHANGED, null, mydata.toUpperCase());~~
        DataModelProxy datamodelProxy = (DataModelProxy) facade.retrieveProxy(DataModelProxy.NAME);
        datamodelProxy.setData(mydata.toUpperCase());
    }
}
```

Finally we need to instantiate and register the DataModelProxy instance in the application's startup code:

```
/*
 * JavaPureMVCMinimal01App.java
 */

package javapuremvcminimal01;

import org.jdesktop.application.Application;
import org.jdesktop.application.SingleFrameApplication;
import org.puremvc.java.patterns.facade.Facade;
import org.andy.MyFormMediator;

public class JavaPureMVCMinimal01App extends SingleFrameApplication {
    public JavaPureMVCMinimal01Form myForm;

    /**
     * At startup create and show the main frame of the application.
     */
    @Override protected void startup() {
        myForm = new JavaPureMVCMinimal01Form(this);
        Facade mvcfacade = AppFacade.getInst();
        mvcfacade.registerMediator(new MyFormMediator(myForm));
        mvcfacade.registerProxy(new DataModelProxy());
        show(myForm);
    }
    …
    …
}
```

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.

```
/*
 * Data.java
 */
package org.andy;

public class Data {

    public String data;

    public Data() {
        this.data = "Hello - hit enter";
    }
}

/*
 * DataModelProxy.java
 */
package org.andy;

import org.puremvc.java.patterns.proxy.Proxy;

public class DataModelProxy extends Proxy {

    public static final String NAME = "DataModelProxy";
    ~~private String data;~~
    private Data realdata;

    public DataModelProxy()
    {
        super(NAME, null);
        ~~this.data = "";~~
        this.realdata = new Data();
        this.sendNotification(AppFacade.DATA_CHANGED, null, this.realdata.data);
    }

    public void setData(String data) {
        ~~this.data = data;~~
        this.realdata.data = data;
        System.out.println("setData (model) " + data);
        ~~this.sendNotification(AppFacade.DATA_CHANGED, null, this.data);~~
        this.sendNotification(AppFacade.DATA_CHANGED, null, this.realdata.data);
    }
}
```

I have make the model string data public to keep the example simple. Feel free to add setters and getters.

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

Let's now create a startup command and move as much of the startup logic into there.

```
/*
 * StartupCommand.java
 */
package org.andy;

import org.puremvc.java.interfaces.ICommand;

import org.puremvc.java.interfaces.INotification;
import org.puremvc.java.patterns.command.SimpleCommand;

import javapuremvcminimal01.JavaPureMVCMinimal01App;

public class StartupCommand extends SimpleCommand implements ICommand {

    public void execute(INotification notification)
    {
        System.out.println("startup execute (command) " + notification.getBody());

        JavaPureMVCMinimal01App app = (JavaPureMVCMinimal01App) notification.getBody();
        facade.registerMediator(new MyFormMediator(app.myForm));
        facade.registerProxy(new DataModelProxy());
    }
}
```

We need register the startup command with the facade and define a STARTUP message which will be used to trigger it:

```
/*
 * AppFacade.java
 */
package org.andy;

import org.puremvc.java.patterns.facade.Facade;
import javapuremvcminimal01.JavaPureMVCMinimal01App;

public class AppFacade extends Facade {

    public static final String STARTUP = "STARTUP";
    public static final String DATA_SUBMITTED = "DATA_SUBMITTED";
    public static final String DATA_CHANGED = "DATA_CHANGED";

    private static AppFacade instance = null;

    public static AppFacade getInst()
    {
        if (instance == null) {
            instance = new AppFacade();
        }
        return (AppFacade) instance;
    }

    @Override
    protected void initializeController()
    {
        super.initializeController();

        registerCommand(STARTUP, StartupCommand.class);
        registerCommand(DATA_SUBMITTED, DataSubmittedCommand.class);
    }

    public void startup(JavaPureMVCMinimal01App app)
    {
        this.sendNotification(STARTUP, app, null);
    }
}
```

And we need to alter the application startup code to do less. Notice that we lose the need for some imports, thus proving that a reason for having the startup command is to decouple and loosen dependencies.

```
/*
 * JavaPureMVCMinimal01App.java
 */

package javapuremvcminimal01;

import org.jdesktop.application.Application;
import org.jdesktop.application.SingleFrameApplication;
~~import org.puremvc.java.patterns.facade.Facade;~~
import org.andy.AppFacade;
~~import org.andy.MyFormMediator;~~
~~import org.andy.DataModelProxy;~~

public class JavaPureMVCMinimal01App extends SingleFrameApplication {
    public JavaPureMVCMinimal01Form myForm;

    /**
     * At startup create and show the main frame of the application.
     */
    @Override protected void startup() {
        myForm = new JavaPureMVCMinimal01Form(this);
        ~~Facade~~ AppFacade mvcfacade = AppFacade.getInst();
        ~~mvcfacade.registerMediator(new MyFormMediator(myForm));~~
        ~~mvcfacade.registerProxy(new DataModelProxy());~~
        mvcfacade.startup(this);
        show(myForm);
    }
    …
    …
}
```

Since we are calling a brand new startup() method on our concrete facade, and this method is not declared in the Facade base class, we need to change the declaration we have been using from Facade to AppFacade. The author of PureMVC recommends this technique of bootstrapping however whether the startup() method officially makes it into the base class in future versions of PureMVC remains to be seen. We don't have to hold our breath for this, as you can see, we simply define our own startup() method on our own concrete facade class.

Step 8. Java specific – Organise the classes into packages

You could potentially move the classes into packages that reflect the roles they are playing. How about:

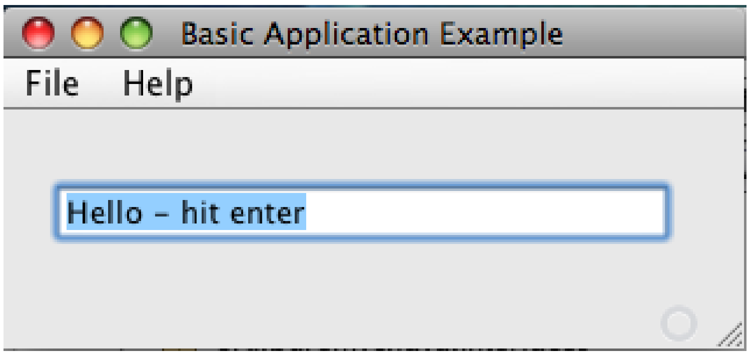
- Data and DataModelProxy → **Model** package
- DataSubmittedCommand and StartupCommand → **Controller** package
- MyFormMediator and JavaPureMVCMinimal01Form → **View** package

and we might as well...

- JavaPureMVCMinimal01App and AppFacade → **App** package

Here is the final code.

Here are a couple of screenshots of the GUI in operation:



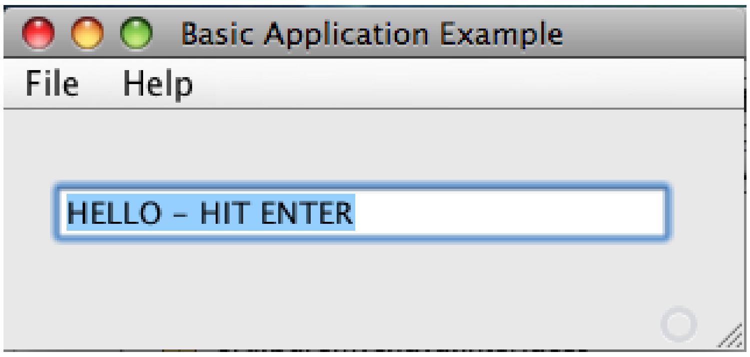
```
compile:

run:

startup execute (command) javapuremvcminimal01.JavaPureMVCMinimal01App@1e67ac

handleNotification (mediator) Hello - hit enter
```

User hits ENTER



```
setData (model) HELLO - HIT ENTER

handleNotification (mediator) HELLO - HIT ENTER
```

Conclusion

We have seen how we can introduce PureMVC into an existing Netbeans application, step by step. I haven't hooked up the about box or the menus into the PureMVC system because these things were created by default by Netbeans and I was focussed simply on duplicating the Python minimalist example. The java source code can be found here.

The Python version of this code contains much more explanation so you may want to study that too, as well as read the more in-depth puremvc refactoring walkthrough based on an idential, python example. Now to put together a C# version? ;-)

-Andy Bulka
March 2009