The simplest Model View Controller (MVC) Java example

Joseph Mack

jmack (at) austintek (dot) com

15 Aug 2011, released under GPL-v3

Abstract

The simplest MVC Java example I could think of; I wanted the MVC version of "Hello World!".

Material/images from this webpage may be used, as long as credit is given to the author, and the url of this webpage is included as a reference.

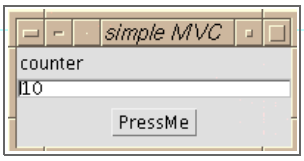
GPL source code is at MVC.tar.gz (http://www.austintek.com/mvc/files/MVC.tar.gz).

Table of Contents

1. Introduction

Most MVC examples show code doing something interesting. Here, to make the MVC functionality clear, the model does almost nothing (it has a counter) and the model, view and controller are separate classes.

Figure 1. MVC GUI



The code here is based on Joseph Bergin's well explained Building Graphical User Interfaces with the MVC Pattern (http://csis.pace.edu/~bergin/mvc/mvcgui.html). Bergin's model has been simplified (Model class) and Bergin's GUI class, which contains both the controller and the view, has been separated into two classes (Controller class, View class), resulting one class for each of the model, view and controller. (Amir Salihefendic, Model View Controller: History, theory and usage http://amix.dk/blog/post/19615, notes that historically the controller and view were coded together as the user interface.)

The best explanation of design patterns I've found is by Bob Tarr CMSC491D Design Patterns In Java (http://userpages.umbc.edu/~tarr/dp/fall00/cs491.html) and CMSC446 Introduction To Design Patterns (http://userpages.umbc.edu/~tarr/dp/spr06/cs446.html).

2. The Model

The model has state (a counter). The model accepts a command to change state (increment the counter). The model emits notices of change of state, and the new state.

The model has no code specifying where the notices are sent or from where it will accept the command. This makes the model reusable. A non-reusable piece of code *RunMVC* tells model to send notices to view and to accept commands from controller.

```
//Model.java
//(C) Joseph Mack 2011, jmack (at) wm7d (dot) net, released under GPL v3 (or any later version)

//inspired by Joseph Bergin's MVC gui at http://csis.pace.edu/~bergin/mvc/mvcgui.html

//Model holds an int counter (that's all it is).
//Model is an Observable
//Model doesn't know about View or Controller

public class Model extends java.util.Observable {

        private int counter;    //primitive, automatically initialised to 0

        public Model(){

                System.out.println("Model()");

                /**
                Problem initialising both model and view:

                On a car you set the speedometer (view) to 0 when the car (model) is stationary.
                In some circles, this is called calibrating the readout instrument.
                In this MVC example, you would need two separate pieces of initialisation code,
                        in the model and in the view. If you changed the initialisation value in one
                        you'd have to remember (or know) to change the initialisation value in the other.
                        A recipe for disaster.

                Alternately, when instantiating model, you could run

                setValue(0);

                as part of the constructor, sending a message to the view.
                This requires the view to be instantiated before the model,
                otherwise the message will be send to null (the unitialised value for view).
                This isn't a particularly onerous requirement, and is possibly a reasonable approach.

                Alternately, have RunMVC tell the view to intialise the model.
                The requires the view to have a reference to the model.
                This seemed an unneccesary complication.

                I decided instead in RunMVC, to instantiate model, view and controller, make all the connections,
                then since the Controller already has a reference to the model (which it uses to alter the status of the model),
                to initialise the model from the controller and have the model automatically update the view.
                */

        } //Model()

        //uncomment this if View is using Model Pull to get the counter
        //not needed if getting counter from notifyObservers()
        //public int getValue(){return counter;}

        //notifyObservers()
        //model sends notification to view because of RunMVC: myModel.addObserver(myView)
        //myView then runs update()
        //
        //model Push - send counter as part of the message
        public void setValue(int value) {

                this.counter = value;
                System.out.println("Model init: counter = " + counter);
                setChanged();
                //model Push - send counter as part of the message
                notifyObservers(counter);
                //if using Model Pull, then can use notifyObservers()
                //notifyObservers()

        } //setValue()

        public void incrementValue() {

                ++counter;
                System.out.println("Model     : counter = " + counter);
                setChanged();
                //model Push - send counter as part of the message
                notifyObservers(counter);
                //if using Model Pull, then can use notifyObservers()
                //notifyObservers()

        } //incrementValue()

} //Model
```

The Model has

- a counter

- a method to increment the counter (the Model's main functionality)
- a method to intitialise the counter
- a null constructor (well, the constructor prints a notice to the console)

> **Note**
> The Model doesn't know about (*i.e.* have a reference to) what is viewing or controlling it. All it knows is its status. This is required for the Model to be reusable.

In this code *Model*:**notifyObservers(counter)** pushes the model's status to the View. In Bergin's code, the generic *Model*:**notifyObservers()** (*i.e.* without sending any status information), requires View to then pull from the Model (in *View*:**model.getValue()**). (The code for pull in the Model is commented out.) If pull is used, the View needs a reference to the Model (code is also commented out in View), making the View less reusable.

By extending *Observable*, Model has the method *Observable*:**addObservers()** to send (change of) status messages to the view. With the code for the Model not having a reference to the View (at least as an attribute), how does the Model know where view is? A non-reusable glue class (my term) *RunMVC* instantiates the objects model, view and controller and tells them what they need to know about each other. *RunMVC*:**myModel.addObserver(Observer myView)** gives the model a reference to the view. Although there is no mention of view in the model code (thus making Model reusable), the model is being passed a reference to the view. This can be done because Model is an *Observable* and an *Observable* knows the declaration of an *Observer* (like view).

To make the *Observable* model reusable, it must not have references to arbitary classes. We need to pass model a reference to the *Observer* view. The declaration of *Observable* knows what an *Observer* is. As long as view extends *Observer*, we can pass a reference to view, without having to hard code view into model. The important part is that we can make model reusable by building in the declaration of *Observer*, allowing model to accept a reference to view .

3. The View pt1

View is the UI (it's a GUI). View shows the model's state, and allows the user to enter commands which will change the model's state.

> **Warning**
> This code works fine, but is not reusable. It's what you might write for a first iteration. If you just want the reusable code and aren't interested in the explanation, hop to The View pt2.

```
//View.java
//(C) Joseph Mack 2011, jmack (at) wm7d (dot) net, released under GPL v3 (or any later version)

//inspired by Joseph Bergin's MVC gui at http://csis.pace.edu/~bergin/mvc/mvcgui.html

//View is an Observer

import java.awt.Button;
import java.awt.Panel;
import java.awt.Frame;
import java.awt.TextField;
import java.awt.Label;
import java.awt.event.WindowEvent;      //for CloseListener()
import java.awt.event.WindowAdapter;    //for CloseListener()
import java.lang.Integer;               //int from Model is passed as an Integer
import java.util.Observable;            //for update();

class View implements java.util.Observer {

        //attributes as must be visible within class
        private TextField myTextField;
        private Button button;

        //private Model model;          //Joe: Model is hardwired in,
                                        //needed only if view initialises model (which we aren't doing)

        View() {
                System.out.println("View()");

                //frame in constructor and not an attribute as doesn't need to be visible to whole class
                Frame frame             = new Frame("simple MVC");
                frame.add("North", new Label("counter"));

                myTextField             = new TextField();
                frame.add("Center", myTextField);

                //panel in constructor and not an attribute as doesn't need to be visible to whole class
                Panel panel             = new Panel();
```

```
                button                  = new Button("PressMe");
                panel.add(button);
                frame.add("South", panel);

                frame.addWindowListener(new CloseListener());
                frame.setSize(200,100);
                frame.setLocation(100,100);
                frame.setVisible(true);

        } //View()

        // Called from the Model
        public void update(Observable obs, Object obj) {

                //who called us and what did they send?
                //System.out.println ("View      : Observable is " + obs.getClass() + ", object passed is " + obj.getClass());

                //model Pull
                //ignore obj and ask model for value,
                //to do this, the view has to know about the model (which I decided I didn't want to do)
                //uncomment next line to do Model Pull
                //myTextField.setText("" + model.getValue());

                //model Push
                //parse obj
                myTextField.setText("" + ((Integer)obj).intValue());    //obj is an Object, need to cast to an Integer

        } //update()

        //to initialise TextField
        public void setValue(int v){
                myTextField.setText("" + v);
        } //setValue()

        public void addController(Controller controller){
                System.out.println("View      : adding controller");
                button.addActionListener(controller);   //need controller before adding it as a listener
        } //addController()

        //uncomment to allow controller to use view to initialise model
        //public void addModel(Model m){
        //      System.out.println("View      : adding model");
        //      this.model = m;
        //} //addModel()

        public static class CloseListener extends WindowAdapter {
                public void windowClosing(WindowEvent e) {
                        e.getWindow().setVisible(false);
                        System.exit(0);
                } //windowClosing()
        } //CloseListener

} //View
```

As is required for reusability, the View doesn't know about (*i.e.* have a reference to) the Model. (If you use the view to initialise the model, and we didn't, then the view needs a refence to the model.)

The View doesn't hold a reference to the controller either. However the View sends button actions to the controller, so the button must be given a reference to the controller. Here is the relevant code (from View), which is called by the glue class *RunMVC*).

```
        public void addController(Controller controller){
                button.addActionListener(controller);
        } //addController()
```

For View to accept the controller reference (parameter to **addController()**), View must have the declaration of *Controller*. The consequence of this requirement is that View is dependant on *Controller* and hence not reusable.

We didn't have the same reusability problem with Model, because Java had declared a base class *Observer*. The problem of View not being reusable comes about because Java doesn't have a base class *Controller*. Why isn't there a base class *Controller*? According to google, no-one has even thought about it. I mused about the central role of the 30yr old MVC to OOP design patterns, and wondered why someone hadn't written a controller base class. The base class had to be extendable by classes that listen (presumably to UIs). Then I realised that *Controller* is an *ActionListner*.

Here's the new View code

```
import java.awt.event.ActionListener;
.
.
        public void addController(ActionListener controller){
                button.addActionListener(controller);
```

```
        } //addController()
```

This gets us part of the way. Now View is reusable by controllers which are ActionListeners, but not controllers which are other types of Listeners. Presumably View can be extended for all listeners. Perhaps no-one needs a base class for Controller.

Extending the idea of referring to controller by its superclass, going up one more level to *java.util.EventListener* gives an error

```
addActionListener(java.awt.event.ActionListener) in java.awt.Button cannot be applied to (java.util.EventListener)
```

4. The View pt2

```
//View.java
//(C) Joseph Mack 2011, jmack (at) wm7d (dot) net, released under GPL v3 (or any later version)

//inspired by Joseph Bergin's MVC gui at http://csis.pace.edu/~bergin/mvc/mvcgui.html

//View is an Observer

import java.awt.Button;
import java.awt.Panel;
import java.awt.Frame;
import java.awt.TextField;
import java.awt.Label;
import java.awt.event.WindowEvent;      //for CloseListener()
import java.awt.event.WindowAdapter;    //for CloseListener()
import java.lang.Integer;               //int from Model is passed as an Integer
import java.util.Observable;            //for update();
import java.awt.event.ActionListener;   //for addController()


class View implements java.util.Observer {

        //attributes as must be visible within class
        private TextField myTextField;
        private Button button;

        //private Model model;          //Joe: Model is hardwired in,
                                        //needed only if view initialises model (which we aren't doing)

        View() {
                System.out.println("View()");

                //frame in constructor and not an attribute as doesn't need to be visible to whole class
                Frame frame             = new Frame("simple MVC");
                frame.add("North", new Label("counter"));

                myTextField             = new TextField();
                frame.add("Center", myTextField);

                //panel in constructor and not an attribute as doesn't need to be visible to whole class
                Panel panel             = new Panel();
                button                  = new Button("PressMe");
                panel.add(button);
                frame.add("South", panel);

                frame.addWindowListener(new CloseListener());
                frame.setSize(200,100);
                frame.setLocation(100,100);
                frame.setVisible(true);

        } //View()

        // Called from the Model
        public void update(Observable obs, Object obj) {

                //who called us and what did they send?
                //System.out.println ("View      : Observable is " + obs.getClass() + ", object passed is " + obj.getClass());

                //model Pull
                //ignore obj and ask model for value,
                //to do this, the view has to know about the model (which I decided I didn't want to do)
                //uncomment next line to do Model Pull
                //myTextField.setText("" + model.getValue());

                //model Push
                //parse obj
                myTextField.setText("" + ((Integer)obj).intValue());    //obj is an Object, need to cast to an Integer

        } //update()

        //to initialise TextField
```

```
        public void setValue(int v){
                myTextField.setText("" + v);
        } //setValue()

        public void addController(ActionListener controller){
                System.out.println("View      : adding controller");
                button.addActionListener(controller);   //need instance of controller before can add it as a listener
        } //addController()

        //uncomment to allow controller to use view to initialise model
        //public void addModel(Model m){
        //      System.out.println("View      : adding model");
        //      this.model = m;
        //} //addModel()

        public static class CloseListener extends WindowAdapter {
                public void windowClosing(WindowEvent e) {
                        e.getWindow().setVisible(false);
                        System.exit(0);
                } //windowClosing()
        } //CloseListener

} //View
```

The view is an *Observer*. The view is a GUI which has

- a TextField to display the status of the model (the value of the counter).
- a Button to communicate with the controller (the controller is a button Listener)..

The trivial functionality of View is a constructor which generates the GUI and a method to initialise the TextField. The interesting functionality of View communicates with the controller and the model.

- a method **addController(ActionListener controller)**, which attaches the controller as a listener to the button (called by the glue class *RunMVC*).
- the magic part, **update()**, which receives the status message from model.

How does **myView.update()** get updated? (It all happens inside the instance *Observable*:**myModel**.)

- model changes state when the method *Model*:**incrementValue()** is executed (by the controller). After first changing the model's state, *Observable*:**setChanged()** changes the flag *Observable*:**changed** to true.
- next *Model*:**notifyObservers(counter)** is run. **notifyObservers(counter)** is a method of *Observable*. **notifyObservers()** checks that **changed** is true, sets it to false, looks up the vector of observers, in our case finding **myView**, and then runs **myView.update(Observable myModel. Object (Integer)counter)**.
- **myView** now has the reference to the observable **myModel** and a reference to its (new) status. Subsequent commands in **update()** present the model's (new) status to the user.

5. The Controller

Controller is a Listener. It

- has a method **actionPerformed()**, which listens to View's button. When the method receives a button press, it changes the state of Model by running **myModel.incrementValue()**.
- has code which is specific to Model and View. Controller is not reusable.

Controller has the following routine functionality

- a constructor which sends a notice to the console.
- a method to initialise the Model.
- methods to get references to myModel and myView (these methods are run by the glue class *RunMVC*).

```
//Controller.java
//(C) Joseph Mack 2011, jmack (at) wm7d (dot) net, released under GPL v3 (or any later version)

//inspired by Joseph Bergin's MVC gui at http://csis.pace.edu/~bergin/mvc/mvcgui.html

//Controller is a Listener

class Controller implements java.awt.event.ActionListener {

        //Joe: Controller has Model and View hardwired in
        Model model;
        View view;

        Controller() {
                System.out.println ("Controller()");
        } //Controller()
```

```
        //invoked when a button is pressed
        public void actionPerformed(java.awt.event.ActionEvent e){
                //uncomment to see what action happened at view
                /*
                System.out.println ("Controller: The " + e.getActionCommand()
                        + " button is clicked at " + new java.util.Date(e.getWhen())
                        + " with e.paramString " + e.paramString() );
                */
                System.out.println("Controller: acting on Model");
                model.incrementValue();
        } //actionPerformed()

        //Joe I should be able to add any model/view with the correct API
        //but here I can only add Model/View
        public void addModel(Model m){
                System.out.println("Controller: adding model");
                this.model = m;
        } //addModel()

        public void addView(View v){
                System.out.println("Controller: adding view");
                this.view = v;
        } //addView()

        public void initModel(int x){
                model.setValue(x);
        } //initModel()

} //Controller
```

6. Glue code

Here is the code which instantiates the classes and passes references to the classes which need them. It then runs the initialisation code. This code is specific to the model, controller and view. It is not meant to be reusable.

```
//RunMVC.java
//(C) Joseph Mack 2011, jmack (at) wm7d (dot) net, released under GPL v3 (or any later version)

public class RunMVC {

        //The order of instantiating the objects below will be important for some pairs of commands.
        //I haven't explored this in any detail, beyond that the order below works.

        private int start_value = 10;   //initialise model, which in turn initialises view

        public RunMVC() {

                //create Model and View
                Model myModel   = new Model();
                View myView     = new View();

                //tell Model about View.
                myModel.addObserver(myView);
                /*
                        init model after view is instantiated and can show the status of the model
                        (I later decided that only the controller should talk to the model
                        and moved initialisation to the controller (see below).)
                */
                //uncomment to directly initialise Model
                //myModel.setValue(start_value);

                //create Controller. tell it about Model and View, initialise model
                Controller myController = new Controller();
                myController.addModel(myModel);
                myController.addView(myView);
                myController.initModel(start_value);

                //tell View about Controller
                myView.addController(myController);
                //and Model,
                //this was only needed when the view inits the model
                //myView.addModel(myModel);

        } //RunMVC()

} //RunMVC
```

I could have had a **main()** in *RunMVC*. However when writing test code, I find it easier for the executable and jar files all to have the same name. Here's Main.

```
//(C) Joseph Mack 2011, jmack (at) wm7d (dot) net, released under GPL v3 (or any later version)

public class Main{

        public static void main(String[] args){

                RunMVC mainRunMVC = new RunMVC();

        } //main()

} //Main
```

Here's the relevant part of my Makefile

```
jar cfe RunMVC.jar Main *.class
```

7. Conclusion

A simple example of MVC. There is one class for each of Model, View and Controller. Model is reusable. View is reusable as long as Controller is an ActionListener. Controller is not reusable.