

H-MVC Framework

User documentation

Author	Revision	Comments
Olivier Refalo	1.0	Initial release.

Summary

INTRODUCTION	3
SIMPLICITY IS A KEYWORD	4
HOW DOES IT WORK?	4
INTRODUCING THE H	7
DESIGN RULES	8
PROGRAMMING THE FRAMEWORK	8
EVENT	8
PROCESSORS	9
ADAPTERS	12
LINKING EVERYTHING TOGETHER	13
MVC FACADES	14
FIRST STEPS EXAMPLE	15
BUILDING THE MODEL	15
BUILDING THE VIEW	17
LINKING EVERYTHING TOGETHER	20
BUILDING THE MAIN PROGRAM	22
TABLE OF FIGURES	23
REFERENCES	23
AUTHOR	23

Introduction

In terms of code complexity, building a rich UI with widgets is nothing compared to building a JSP page. If you don't pay close attention on how widgets talk to each other, and more precisely, if you don't make your design reusable, you will soon end up with un-maintainable *spaghetti code*.

This is one of main idea behind this framework: Improving development time and maintenance by enforcing code writing and organization rules. In other words giving more flexibility to the GUI design by splitting apart processing from presentation in hierarchical levels.

But understanding how a design works doesn't mean you would implement it correctly. If you consider that many developers aren't experts in the area, you will soon be facing problems that others may have already figured out.

The framework presented in this document provides all the mechanisms to implement a clean room HMVC pattern for rich GUI applications. By clean room I mean a set of structure independent from the actual widget API (AWT, Swing, SWT...etc).

In fact, the only thing left to the programmers is the GUI graphical look, the definition if the actions associated with each widget components and linking components together.

Simplicity is a keyword

This framework is deliberately simple. Simplify is a keyword that all programmers should keep in mind.

Have you noticed something funny about open source projects? They tend to become more and more sophisticated and as a consequence their API becomes more and more complex. A recent study showed that most MS Word users were using only 15% of its features. Is that going to be the same for open source projects?

I don't know... However I believe most programmers (or at least a few of them) are just like me, they like *simple* things. They like to understand what an API is doing without being jammed with features they will never use. Technically understanding an API is actually very important because it makes it faster for the developer to track problems and to extend his design.

How does it work?

First, let's start with the famous MVC pattern; we will come back on the 'H' of H-MVC later.

The MVC is decomposed in three different components called the Model, the View and the Controller. Those components are connected to each other in such a way that the Controller acts as a mediator between the model and the view. Each component can send and receive events from its connected counter part(s).

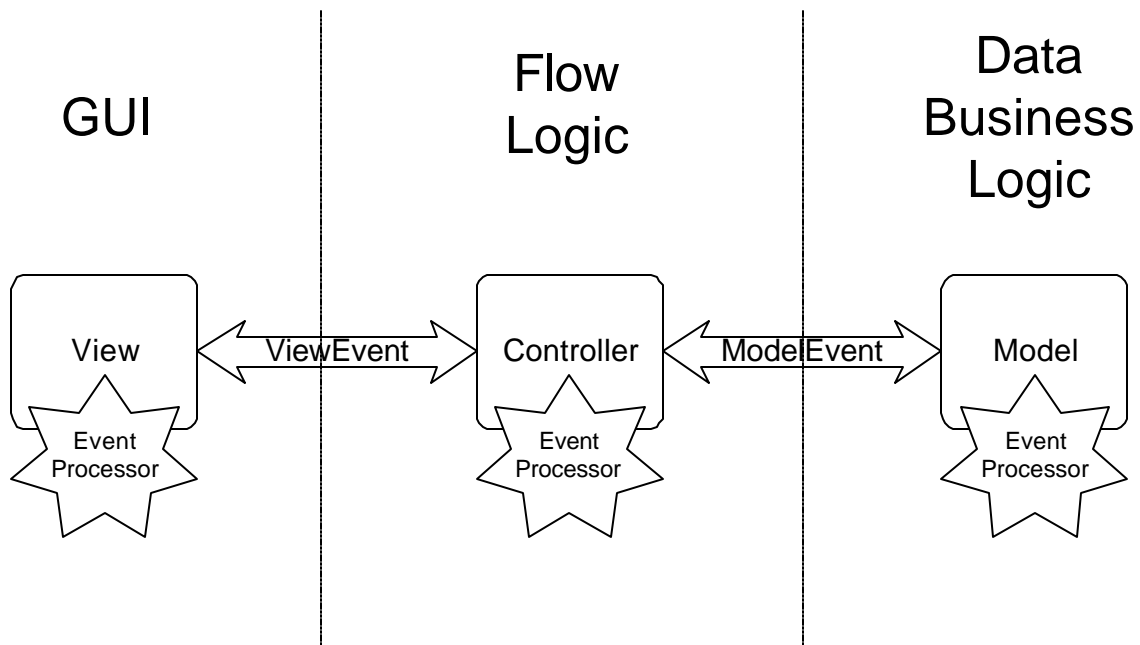


Figure 1: MVC pattern

Let's see how MVC can help us in organizing and reusing our code:

- ?? The View is where the interactions with the GUI are programmed. This is typically the place where user input validation rules are programmed. The view fires events (called ViewEvents) to the controller, which will process them accordingly. The View also receives view events from the controller to perform GUI updates.
- ?? The Controller is a real mediator¹: it receives events from either the model or the view and decides which component must be triggered to process the action. There is no business logic here; it's just a relay, which "forwards" events from one component to the other.
- ?? The Model contains all the rest! In other words, anything, which is not GUI related and process flow related. Typically, Business logic and data model methods would be defined here. The Model, as the above components can send and receive events (called ModelEvents).

¹ See Mediator pattern

Because these components exchange events, they are not tightly coupled and it becomes easy to replace one by another. Plus, you can associate several of them together triggering several GUI components when their associated data has changed. The Events or processed by EventProcessors which will be introduced later in this document.

To clarify the whole thing, let's pick the flag example:

In an application GUI, you would like to display a flag status as a String value and a graphic. The String and the graphic or obviously pointing to the same data, the information is just displayed in two different ways. The model would be holding the flag state as an integer value: 0 for red, 1 for orange, 2 for green. There are two views:

?? A string view that converts the model integer value in its string equivalent.

?? A flag view that converts the same integer value in color signals.

The controller would be the link between the views and the model, propagating updates and status events over the system.

Please read the first steps examples for a complete demonstration.

Introducing the H

MVC is a very helpful pattern, but in a complex component architecture like Swing, it lacks flexibility. For instance, let's imagine that a developer already built a MVC GUI widget. You would like to reuse it in your own application without changing his code. Well this is where the 'H' magic comes to the rescue. Controllers can be linked together in what we call a "chain of responsibility", forming a tree of MVC components. Typically, the tree top hierarchy would be a JFrame or a JApplet (with their associated controller and view). When a leaf component doesn't know how to process a given event, it just delegates it to its parent. The event is lost if no controller knows how to process it.

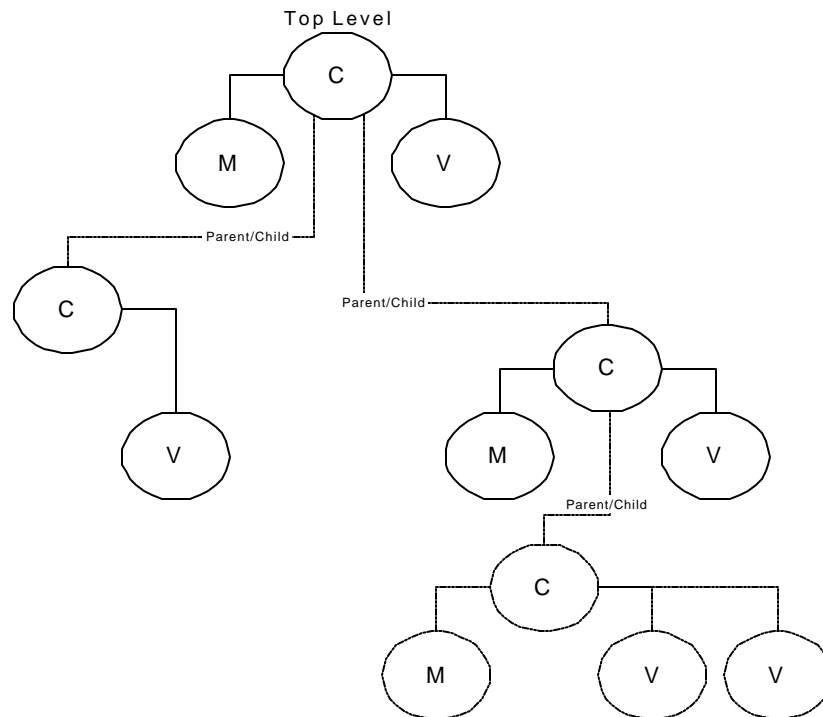


Figure 2: HMVC

Design rules

There are a few important rules about this pattern:

1. The Model and the View don't know anything about each other. They use the controller to exchange information.
2. There is no constraints on how components talk to each other. It can be a direct method call, listeners... this framework implements a listener approach.
3. Finally don't be surprised if you see other declination of this pattern. MVC has been widely used in the computer industry and hence widely changed. But the main idea remains the same.

Just for information, the framework implements late instantiation in order to minimize memory usage.

Programming the framework

Enough for the theory, now let's see the beast...

At this point, you should understand that:

- ?? MVC is based on three components, each of them having its responsibility.
- ?? Model and View use the controller to exchange events.
- ?? The controller controls the flow logic.
- ?? Each component can fire and receive events.

Event

An Event is an object that transits from one component to another. It usually holds data and triggers an EventProcessor on the destination component. It's important to notice that Events are a one-way communication elements: **they do not return a value**. They hold an event type (which is an integer value), a pointer to the source of the event and an optional data.

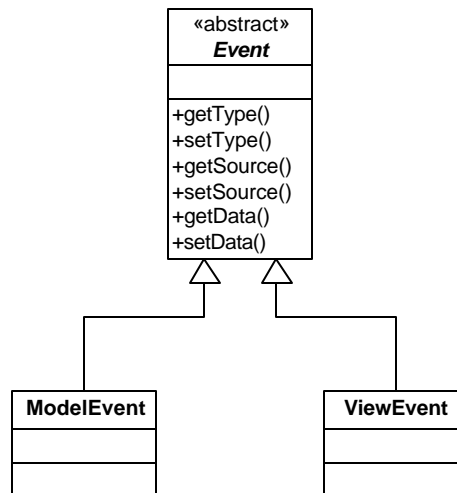


Figure 3: Event UML

There are two kinds of events:

1. GUI related (called ViewEvents) which transit from the View to the Controller
2. Process related (called ModelEvent) which transit between the model and the controller.

In the example below, we create an ModelEvent holding the value 15 and triggering an update.

```
Event e=new ModelEvent(DefaultEventType.UPDATE, this, new Integre(15));
```

Processors

Although you can extend or compose components, they usually don't hold the application logic: Processors do.

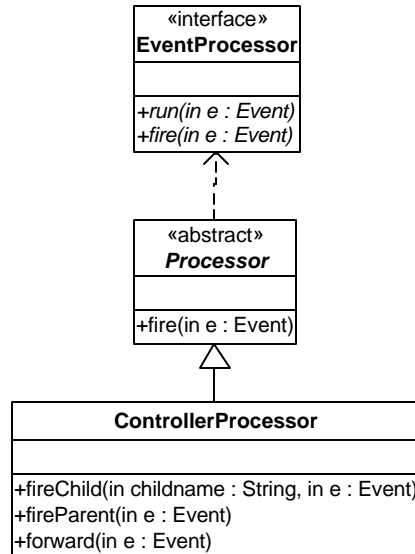


Figure 4: Processor UML

Let's first talk about the basics:

1. As you can see in the figure above, there are two kinds of Processors: ControllerProcessors and Processors. Controller processors are used exclusively in the controller and provide extra methods.
2. Each processor has a run() method which is called when a specific event is triggered. At build time, Processors are linked with components and are associated with an event to reply to.

From a processor, it is possible to trigger the connected component by calling:

```
Void fire(Event e)
```

Please note that it is only possible to send events compatible with the component you are in. Check the table below for a summary:

From component	Allowed trigger
<i>Model</i>	ModelEvent
<i>View</i>	ViewEvent
<i>Controller</i>	ModelEvent & ViewEvent

See the example below:

```
Model m=new ModelImpl();
Processor p=new Processor() {
    private Object v=null;
    public void run(Event e) {
        System.out.println("Select event triggered");
        v=e.getData();
        fire(DefaultEventType.UPDATE, this, v);
    }
};
m.register(DefaultEventType.SELECT, p);
```

The code snapshot above configures the model component to accept SELECT events. When the model receives a SELECT event, it will display the string “Select event triggered” on the console. The processor then updates the local variable and fires an UPDATE event.

Controllers have a special processor called a ControllerProcessor, which adds three methods:

```
void fireParent(Event e)
void fireChild(String _childname, Event e)
void forward(Event e)
```

Use the two first methods to propagate events in the HMVC hierarchy. The last one is used to propagate an Event from the Model to the View and vice versa.

The code bellow will fires an UPDATE event to the model and propagate the event to its parent when a SET event is received on the Controller.

```
Controller c=new ControllerImpl();
Processor p=new ControllerProcessor() {
    public void run(Event e) {
        Event me= new ModelEvent(DefaultEventType.UPDATE, this);
        this.fire(me);
        this.fireParent(me);
    }
};
c.register(DefaultEventType.GET, p);
```

Adapters

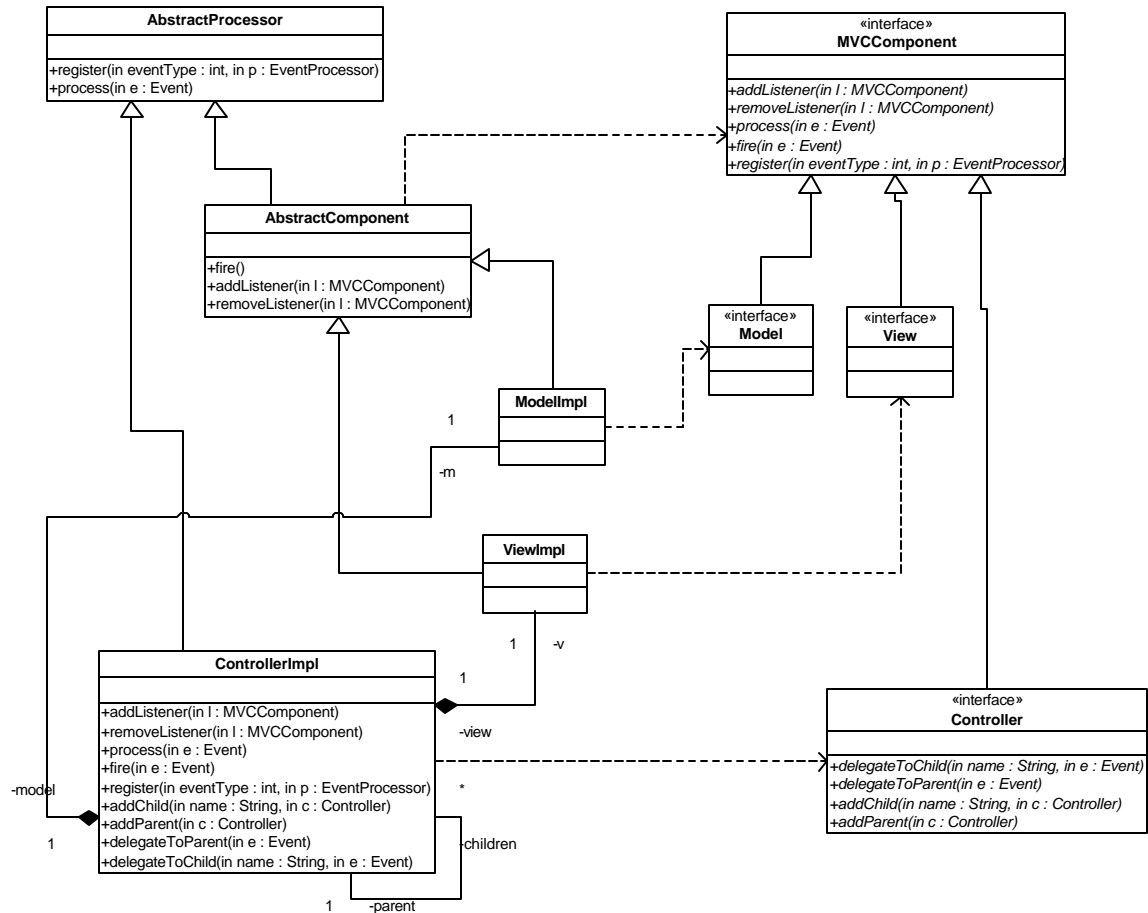


Figure 5: HMVC UML

Most of you already know that multiple inheritance is forbidden in Java. But the API must be flexible enough to provide this feature by either extending or composing a given component. This mechanism is designed in Java by using an Interface and an implementation class. Programmers who need to extend the class would extend the implementation while the ones who need to compose the class would implement the interface and link all methods with the composed implementation. See the examples below:

```
class MyModel extends ModelImpl {  
  
...  
  
}
```

Figure 6: Enhancement by inherence

```
class MyView extends JPanel implements View {  
private View v=new ViewImpl();  
  
public void addListener(Listener l) {  
    v.addListener(l);  
}  
  
// etc.. do the same for all View methods  
  
}
```

Figure 7: Enhancement by composition

Linking everything together

Now that we saw how to create and configure components, let's see how we can link these objects together. Each component has methods to add and remove listeners. The term *Listener* means that the object (the listener) is listening to events fired in the component it is attached to.

```
void addListener(MVCCComponent _sl);  
  
void removeListener(MVCCComponent _sl);
```

Let me give a simple example to clarify the process. In the example below, we will setup a Controller and attach a listener to it. The Listener object is a Model, hence it will be triggered when the controller fires ModelEvents.

```

//Create the Controller
Controller c=new ControllerImpl();

Processor p=new ControllerProcessor() {
    public void run(Event e) {
        // fire all listeners listening to ModelEvents
        this.fire(new ModelEvent(ModelEventType.SELECT, this));
    }
};
c.register(DefaultEventType.OPEN, p);

// Now create the model

Model m=new ModelImpl();

// and setup a processor which will be called when a ModelEvent
// of type SELECT is raised.

p=new Processor() {
    public void run(Event e) {
        test=true;
    }
};
m.register(DefaultEventType.SELECT, p);

// Finally attach the model to the controller.

c.addListener(m);

```

The example demonstrated how it is possible to make one component listening the events raised in another component. You can of course add more than one listener.

MVC Facades

Linking components together soon becomes a pain. That's the reason why we created a Façade (called MVC) which links together the components passed as a parameter. The Façade only accepts one model/view/controller, if you need to link the controller to multiple views (or models), you will have to extend this class and write your own implementation.

First steps example

We saw the theory, an API presentation, what about a real example?

We will build a simple application: a temperature converter. The application will provide an easy way to convert Celcius to Fahrenheit and vice versa.

Building the model

Let's start with the easy part: The model. Finding the model actions in this application is trivial: It is the conversion formulas defined by the following equations.

$$\text{Celcius} = \left(\frac{5}{9}\right) \text{ Fahrenheit} + 32$$

$$\text{Fahrenheit} = \left(\frac{9}{5}\right) (\text{Celcius} - 32)$$

So what do we need to build out a model? Well, a public interface like this one would make it. But that's not really pluggable.

```
class ConverterModel {  
  
    double toCelcius(double fahrenheit);  
    double toFahrenheit(double celcius);  
  
}
```

Let's see how this definition translates using the HMVC framework:

```

package com.crionics.hmvc.example;

import com.crionics.hmvc.Event;
import com.crionics.hmvc.ModelEvent;
import com.crionics.hmvc.ModelImpl;
import com.crionics.hmvc.Processor;

public class ConverterModel extends ModelImpl {

    public ConverterModel() {

        // TO_CELCUS Processor
        Processor p = new Processor() {
            public void run(Event e) {
                double temp = ((Double) e.getData()).doubleValue();
                double celcus = Math.round((temp * 9 / 5) + 32);
                fire(new ModelEvent(ConverterEventType.UPDATE_FARENHEIT,
this, new Double(celcus)));
            }
        };
        register(ConverterEventType.TO_FARENHEIT, p);

        // TO_FARENHEIT Processor
        p = new Processor() {
            public void run(Event e) {
                double temp = ((Double) e.getData()).doubleValue();
                double far = Math.round((temp - 32) * 5 / 9);
                fire(new ModelEvent(ConverterEventType.UPDATE_CELCUS,
this, new Double(far)));
            }
        };
        register(ConverterEventType.TO_CELCUS, p);

    }

}

```

This is it! We just built a pluggable component, which replies to events: `TO_FARENHEIT` and `TO_CELCUS`. The events carry a double value, which represents the initial temperature to convert from.

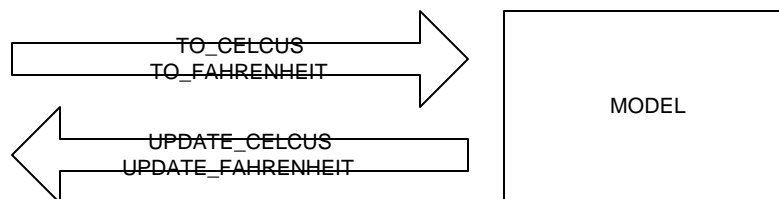
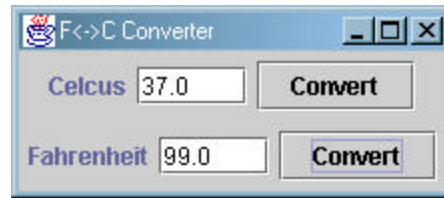


Figure 8: Model Events Diagram

The processors also fire an update event upon computation. We will see in the next step how these updates interact with the GUI.

Building the view



On the view side, we need to provide two things:

- ~~✍~~ A way to send the chosen temperature to the converter model.
- ~~✍~~ A way to display the computed value in the appropriate field.

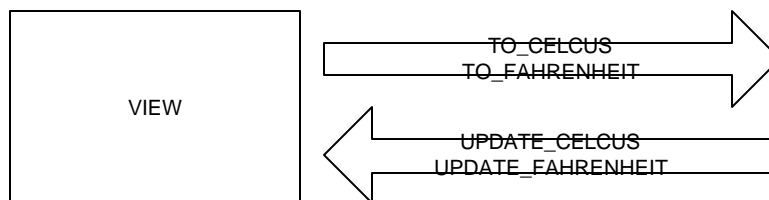


Figure 9: View Events Diagram

Optionally, The view can also validate the input in order to make sure the temperature is a decimal value.

```

package com.crionics.hmvc.example;

import com.crionics.hmvc.Event;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;

import com.crionics.hmvc.Processor;
import com.crionics.hmvc.ViewEvent;
import com.crionics.hmvc.swing.JPanelView;

public class ConverterView extends JPanelView {
    GridLayout gridLayout1 = new GridLayout();
    JPanel upP = new JPanel();
    JPanel downP = new JPanel();
    JLabel celcusL = new JLabel();
    JTextField celcusTF = new JTextField();
    JButton celcusB = new JButton();
    JLabel farL = new JLabel();
    JTextField farTF = new JTextField();
    JButton farB = new JButton();

    public ConverterView() {
        try {
            jbInit();
        } catch (Exception e) {
            e.printStackTrace();
        }

        Processor p = new Processor() {
            public void run(Event _e) {
                celcusTF.setText(_e.getData().toString());
            }
        };

        register(ConverterEventType.UPDATE_CELCUS, p);

        p = new Processor() {
            public void run(Event _e) {
                farTF.setText(_e.getData().toString());
            }
        };
        register(ConverterEventType.UPDATE_FAHRENHEIT, p);
    }
}

```

```

public static void main(String arg[]) {
    JFrame f = new JFrame();
    f.getContentPane().add(new ConverterView());
    f.pack();
    f.show();
}

private void jbInit() throws Exception {
    GridLayout1.setRows(2);
    GridLayout1.setColumns(1);
    this.setLayout(GridLayout1);
    celcusL.setText("Celcus");
    celcusTF.setText("30");
    celcusTF.setColumns(5);
    celcusB.setText("Convert");
    celcusB.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            celcusB_actionPerformed(e);
        }
    });

    farL.setText("Fahrenheit");
    farTF.setText("80");
    farTF.setColumns(5);
    farB.setText("Convert");
    farB.addActionListener(new java.awt.event.ActionListener(){
        public void actionPerformed(ActionEvent e) {
            farB_actionPerformed(e);
        }
    });

    this.add(upP, null);
    upP.add(celcusL, null);
    upP.add(celcusTF, null);
    upP.add(celcusB, null);
    this.add(downP, null);
    downP.add(farL, null);
    downP.add(farTF, null);
    downP.add(farB, null);
}

void celcusB_actionPerformed(ActionEvent _e) {

    // validate the data.
    Double d = null;
    try {
        String s = celcusTF.getText();
        d = Double.valueOf(s);
    } catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(
            this,
            "Celcus temperature is invalid",
            "Format Exception",
            JOptionPane.ERROR_MESSAGE);
        return;
    }
}

```

```

        fire(new ViewEvent(ConverterEventType.TO_FAHRENHEIT, this,
d));
    }

    void farB_actionPerformed(ActionEvent _e) {
        // validate the data.
        Double d = null;

        try {
            String s = farTF.getText();
            d = Double.valueOf(s);
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(
                this,
                "Fahrenheit temperature is invalid",
                "Format Exception",
                JOptionPane.ERROR_MESSAGE);
            return;
        }
        fire(new ViewEvent(ConverterEventType.TO_CELCUS, this, d));
    }
}





```

The source code looks complicated, but actually most of it is GUI related.

Linking everything together

Now that we have a model and a view component, we just need to link them together using a controller. Our controller is very simple: it just forwards events from the view to the model and vice versa.

But in more sophisticated applications, the controller may be doing other tasks such as:

-  Converting events
-  Formatting events
-  Checking event ranges
-  Validation...etc.

```

package com.crionics.hmvc.example;

import com.crionics.hmvc.*;
import com.crionics.hmvc.swing.*;

public class ConverterController extends ControllerImpl {

    /**
     * Constructor for ConverterController
     */
    public ConverterController() {
        super();

        ControllerProcessor p = new ControllerProcessor() {
            public void run(Event _e) {
                forward(_e);
            }
        };

        register(ConverterEventType.TO_CELCUS, p);
        register(ConverterEventType.TO_FAHRENHEIT, p);
        register(ConverterEventType.UPDATE_CELCUS, p);
        register(ConverterEventType.UPDATE_FAHRENHEIT, p);
    }
}

```

Below is the complete diagram describing how components talk to each other.

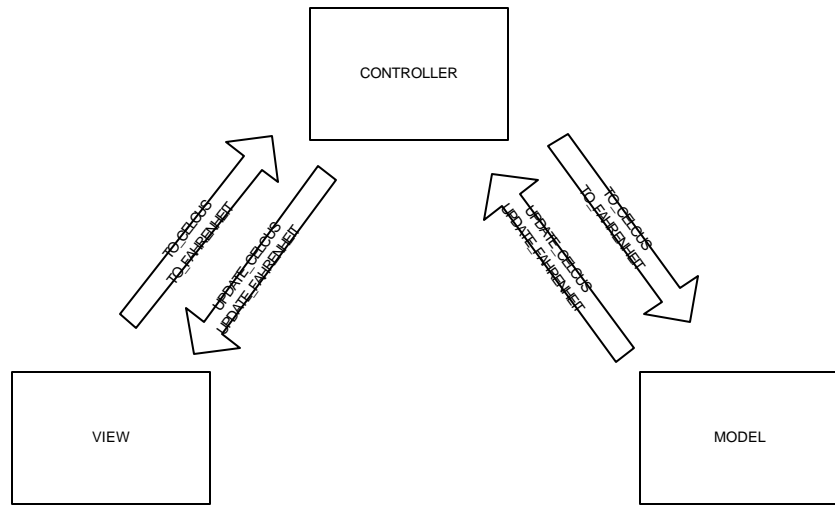


Figure 10: MVC Events Diagram

Building the main program

Here we go; our components are finished. From a UI stand point we have a Jpanel, which holds the entire user interface. Now, how do we popup a window? And how to we make this window HMVC compliant?

```
package com.crionics.hmvc.example;

import com.crionics.hmvc.Controller;
import com.crionics.hmvc.MVC;
import com.crionics.hmvc.MVCImpl;
import com.crionics.hmvc.Model;
import com.crionics.hmvc.View;
import com.crionics.hmvc.swing.DefaultJFrameController;
import com.crionics.hmvc.swing.JFrameView;

public class Main extends MVCImpl {
    /**
     * Constructor for Main
     */
    public Main(Model _m, View _v, Controller _c) {
        super(_m, v, c);
    }

    public static void main(String[] args) {
        // Create the frame
        JFrameView f = new JFrameView("F<->C Converter");

        // Create the top level MVC with a default controller
        // which understands WINDOW_CLOSE events
        Main main = new Main(null, f, new
DefaultJFrameController());

        // Create the converter MVC
        ConverterModel m = new ConverterModel();
        ConverterView v = new ConverterView();
        ConverterController c = new ConverterController();
        MVC converter = new MVCImpl(m, v, c);

        // add the Jpanel to the JFrame
        f.getContentPane().add(v);
        // Link the controllers together
        main.addChild("converter", converter);

        f.pack();
        f.show();
    }
}
```

Table of figures

<i>Figure 1: MVC pattern</i>	5
<i>Figure 2: HMVC</i>	7
<i>Figure 3: Event UML</i>	9
<i>Figure 4: Processor UML</i>	10
<i>Figure 5: HMVC UML</i>	12
<i>Figure 6: Enhancement by inherence</i>	13
<i>Figure 7: Enhancement by composition</i>	13
<i>Figure 8: Model Events Diagram</i>	16
<i>Figure 9: View Events Diagram</i>	17
<i>Figure 10: MVC Events Diagram</i>	21

References

[Javaworld](http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc.html) - <http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc.html>
Very good article, which actually inspired this framework

[The Scope Project](http://scope.sourceforge.net/) - <http://scope.sourceforge.net/>

Author

©2001 Olivier Refalo
<mailto:orefalo@yahoo.com>
<http://www.crionics.com>