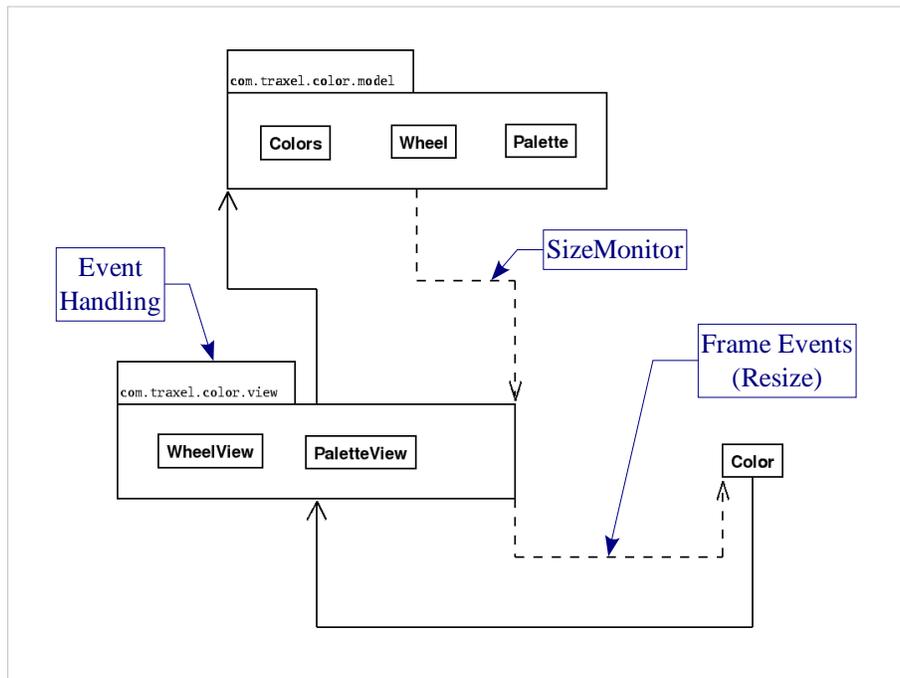


The first pass follows my typical development style for thick UI apps.

The model is clearly separated.

The view has all the meat of what remains.

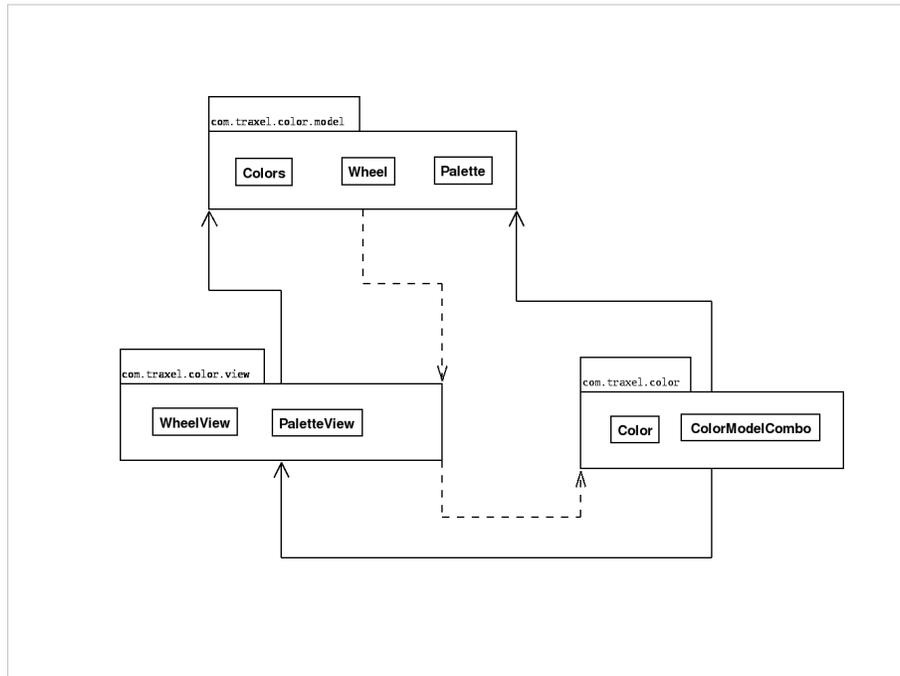
There is one external class to initialize the frames and place the components on screen.



Event handling in this version is done entirely in the view.

There is a weak connection from the model to the view – it needs to know the size of the panel to calculate the location of colors on screen.

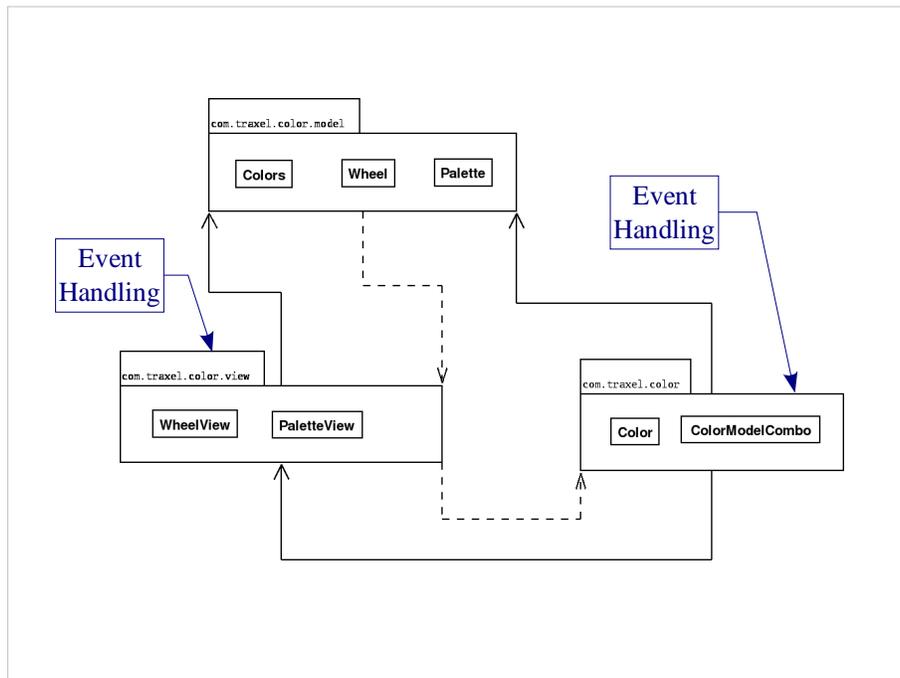
The view panels have a weak connection to the frame, to detect frame resizing (this is all handled internally by the Swing components).



On the second pass, I added a combo to control the color model.

This component doesn't feel like a view, so I put it in the same external package with Color.

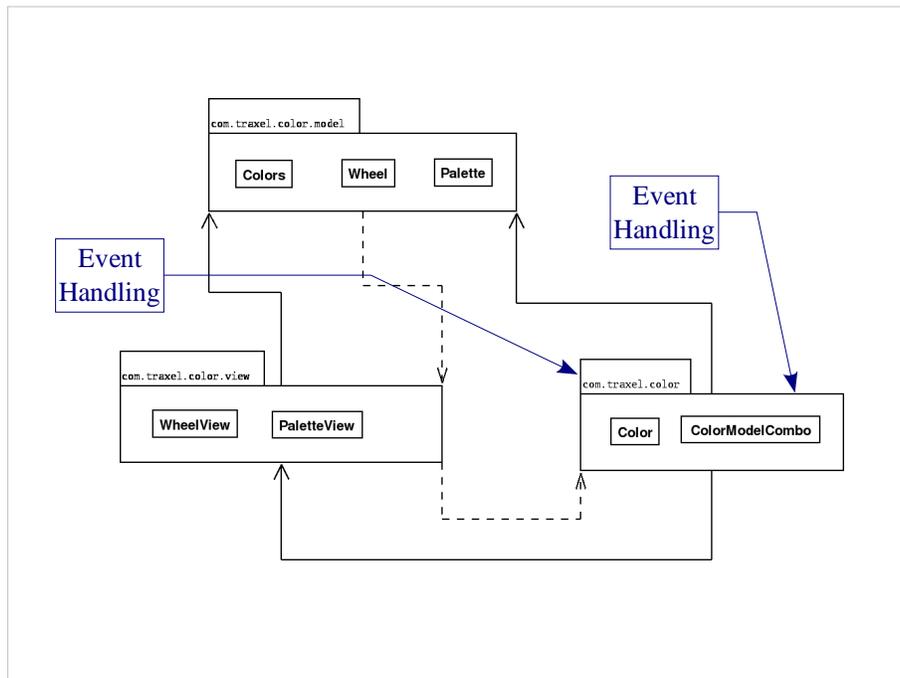
ColorModelCombo builds a new palette and hands it to PaletteView when the user makes a selection.



At this stage, the event handling shows up in two different places.

When the user clicks on the wheel view to change the primary color of the palette, the event is handled in `WheelView`.

When the user changes the color model with the color model combo, the event is handled in `Color` (which has a listener on `ColorModelCombo`).

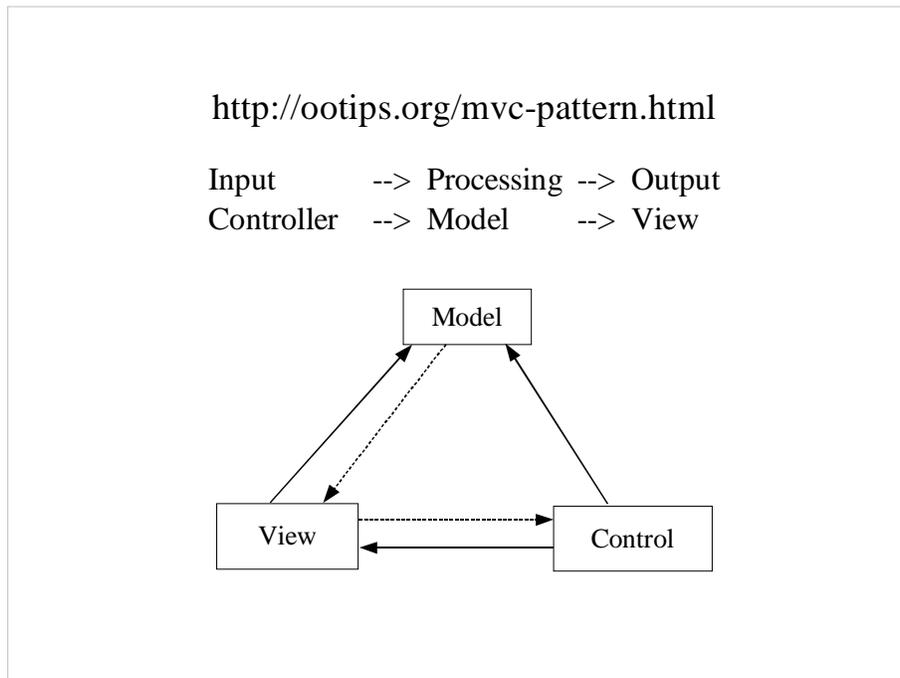


Looking at the code, I decided that the event handling code in WheelView looked more like the event handling code in Color than it did like anything else in WheelView.

So I moved the primary color selection handling code from WheelView to Color.

At this point I'm pretty happy with the model / UI separation, but I still don't know what a control is.

So I decided it was time to go and figure out what this whole MVC thing is and see how hard it would be to refactor my code to use it.



I Googled MVC and the second link took me to a page on OOTips.org.

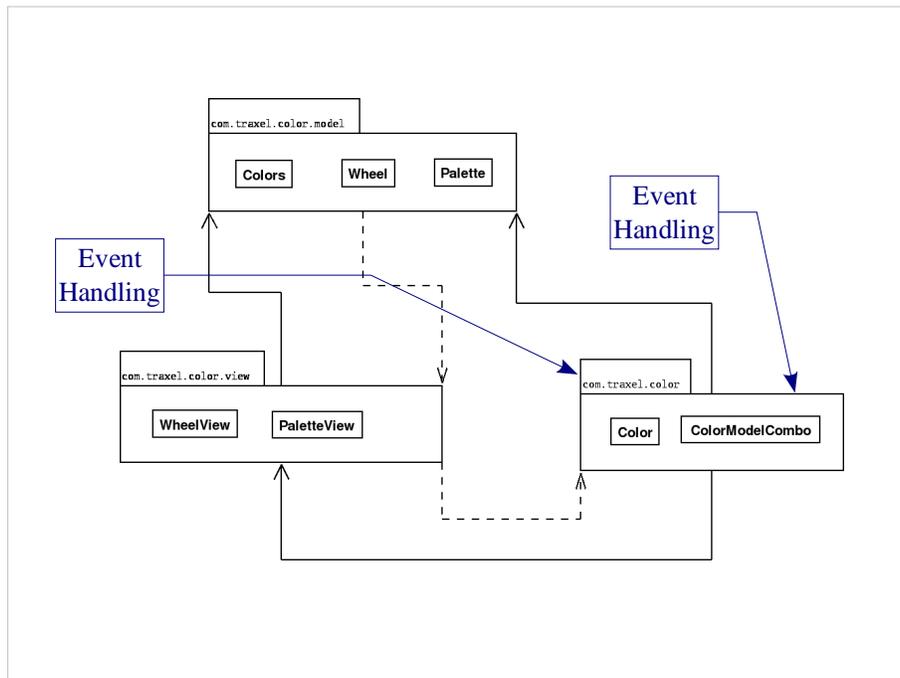
It starts by explaining that MVC is an attempt to model traditional Input / Processing / Output into the GUI world.

Near the bottom of the page is this simple diagram.

Model which handles the state and logic.

View which presents a visual image of the model.

Control I don't yet understand, but I know it's related to the user input.

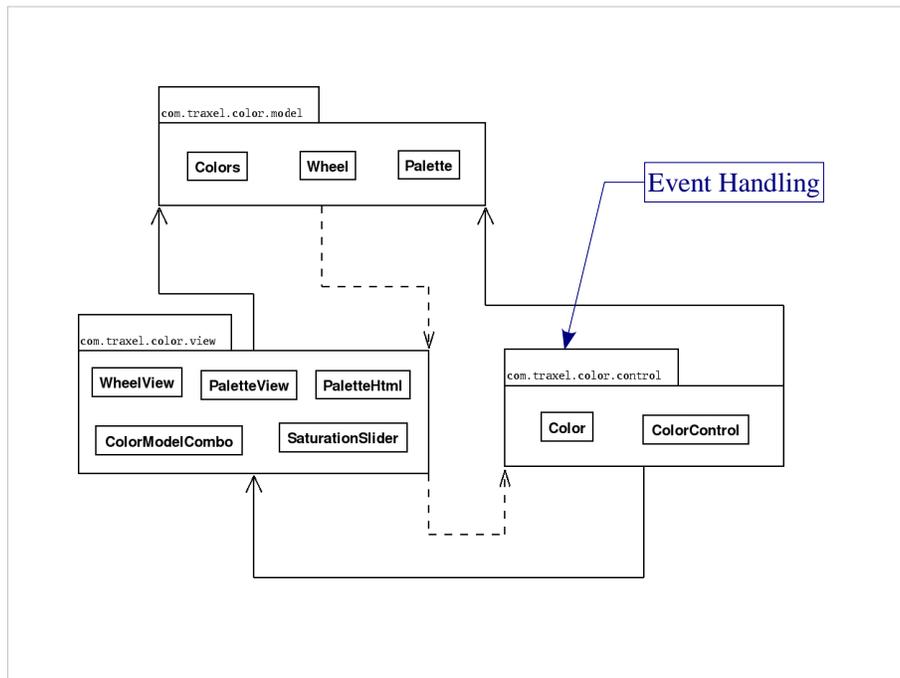


Looking back at my code, this is starting to look pretty familiar.

Model holding all the state and logic.

View creating a visual image of the Model

And this other thing where all the user input is being handled.

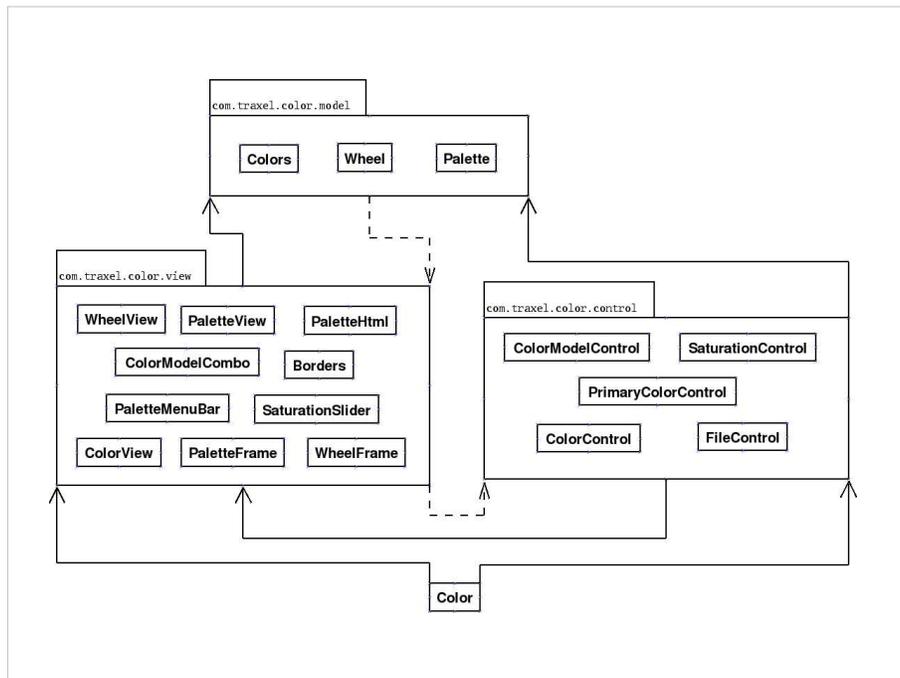


As I was polishing the code, the control simply fell out in front of my eyes.

Ron Crocker at OOPSLA put it this way; “the whole point of a pattern is that it's what happens naturally if you take the time to mercilessly refactor your code.”

So I moved the ColorModelCombo into the view, and created SaturationSlider.

Then I created ColorControl as the central class for all event handling, and changed the package name to control.



Then some more merciless refactoring, and given my preference for all things fine-grained, I reached the current state of the application.

There is now a view for each component, including the Frames.

And I've encapsulated the controls based on their conceptual function – what they do from the user's perspective regardless of what object they control.

```

package com.traxel.color.control;

class PrimaryColorControl extends ColorControl {

    PrimaryColorControl( ColorView view ) {
        super( view );
        controlPrimaryColor();
        setPrimary( Colors.BLUE );
    }

    private void controlPrimaryColor() {
        MouseListener colorControl;

        colorControl = new MouseAdapter() {
            public void mouseClicked( MouseEvent e ) {
                int x;
                int y;
                Color color;

                x = e.getX();
                y = e.getY();
                color = getWheel().getBaseColor( x, y );
                setPrimary( color );
            }
        };
        getWheelView().addMouseListener( colorControl );
    }
}

```

This is the primary color control.

It passes the view to the superclass (which just has a bunch of convenience methods shared by all the controls).

It then initializes the event handler and connects it to the WheelView, and sets the initial state of the controlled concept.

When invoked, the event handler (AKA: the control) gets the position of the click, uses the Wheel to translate the coordinates into a color, and uses a convenience method in the superclass that tells the palette to change its primary color.

```

package com.traxel.color.control;

class SaturationControl extends ColorControl {

    SaturationControl( ColorView view ) {
        super( view );
        controlSaturation();
        setSaturation( 7 );
    }

    private void controlSaturation() {
        ChangeListener saturationControl;

        saturationControl = new ChangeListener() {
            public void stateChanged( ChangeEvent e ) {
                setSaturation
                    ( getSaturationSlider().getValue() );
            }
        };
        getSaturationSlider().addChangeListener
            ( saturationControl );
    }
}

```

The SaturationControl is very similar. It passes the view to the superclass, initializes the event handler, and sets the initial state. This event handler simply grabs the current value from the slider and calls the superclass's convenience method, which tells the palette to update its saturation.