4 A Traffic Light Application

What's in This Set of Notes?

It is a good idea now to look at an application with a more interesting model component. We will look at the example of a TrafficLight model object with a user interface attached to it. We will also look at how we can use a **Timer** object to have automatic updating of the TrafficLight.

Here are the individual topics found in this set of notes (click on one to go there):

4.1 Application Description

The purpose of this application is to give you more experience in understanding how to create an application by using the following steps:

1. Understand what you want the interface to do.
2. Understand what the model should be and then develop the model.
3. Create the user interface layout.
4. Connect the interface to the model.

You should follow these steps whenever creating an interface in this course.

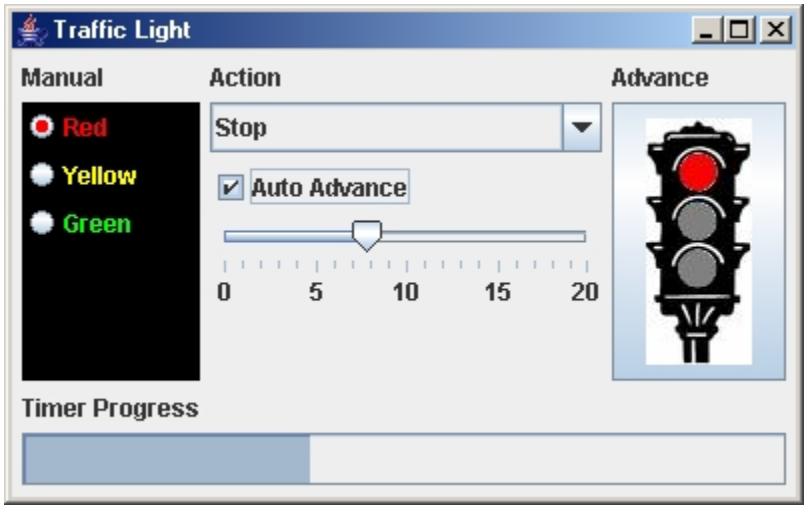
Understanding the interface:

First let us describe the application. The application is a window with the following components:

- A set of 3 **JRadioButtons** to represent the traffic light state (Red, Yellow or Green).

- A **JComboBox** to indicate actions (i.e., Stop, Yield and Go) which correspond to the traffic light state.
- a JButton with a traffic light icon which reflects (and allows advancing of) the state of the traffic light.
- a **JCheckBox** which will allow the light to advance automatically, based on a timer.
- a **JSlider** that will allow the user to adjust the speed of the automatic time advance feature.
- a **JProgressBar** that will indicate how much longer the traffic light will remain in its current state.
- various JLabels to make the window more self-explanatory.

The interface will look something like this:



Here is how the behaviour of the interface is described:

- The radio buttons and the combo box should be linked together such that if the **Red** light is selected, then the **Stop** item is selected automatically in the combo box as well. The same goes for the **Yellow**/**Yield** and **Green**/**Go** options as well. Also, if a selection is made in the combo box, then this selection is also made in the radio button group as mentioned.

- The status pane should show the following messages depending on the state of the lights:
  - Red: "Red Traffic Light Means: 'Stop Please'"
  - Yellow: "Yellow Traffic Light Means: 'Yield to Others'"
  - Green: "Green Traffic Light Means: 'Go Really Fast'"

- The **Advance** button should cause the light to advance to the next state in a cyclic fashion in the order red, green, yellow, red, green, ...

- When the **Auto** button is pressed, the lights will advance automatically such that the red light stays on for 6 seconds, the green for 8 seconds and the yellow for 3 seconds. Pressing the **Advance** button while in **Auto** mode will also advance the state of the light. The progress bar should indicate the number of seconds that the traffic light has been in its current state.

4.2 Developing the Model

How do we make the model ? What is the model ? It is a traffic light.
Two simple ways of representing a traffic light are as:

- an integer from 1 to 3 (where 1 = Red, 2 = Yellow, 3 = Green) or some other numbering scheme
- one of the following Strings: "Red", "Yellow" or "Green"

This would work, but then we don't get to define any behaviour such as **getState()** or **advanceState()**. We should make our own class.
Within this class, we will simply use an integer to store the state.

Note:

- It is always a good idea to make a "stand-alone" model with behaviour such that any kind of user interface can be plugged into it.

Here is the model code:

```
public class TrafficLight {
    int  currentState;

    // Constructor that makes a red traffic light
    public TrafficLight() {
        currentState = 1;
    }

    // Advance the traffic light to the next state
    public int advanceState() {
        currentState = ++currentState % 3 + 1;
        return currentState;
    }

    // Return the state of the traffic light (as a number from 1 to 3)
    public int getState() {
        return currentState;
    }

    // Set the state of the traffic light (as a number from 1 to 3)
    // If the integer is out of range, do nothing
```

```
    public void setState(int newState) {
        if ((newState > 0) && (newState <4))
            currentState = newState;
    }

    // Return a string representation of the traffic light
    public String toString() {
        String[] colours = {"Red", "Yellow", "Green"};
        return colours[currentState] + " Traffic Light";
    }
}
```

Making a traffic light is easy:

```
    new TrafficLight();
```

We can ask for the state or change the state. A state of 1 is a Red light, 2 is a Yellow light and 3 is a Green light.
The advance method will cause the state to cycle as follows: 1, 3, 2, 1, 3, 2, 1, 3, 2, 1, ...

Notice that there are no **System.out.println()** messages here, nor is there any keyboard input. That is because, those are actually I/O operations and they depend heavily on the type of user interface that will be used. We leave that kinda "stuff" out of the model let the user interface worry about those issues. (Sometimes we may have println statements for debugging/testing purposes, but ultimately these should be removed from the model class code). That way, we can "plug-in" the model into any user interface and the model becomes modular, clean, shared code. So remember,

Model classes should NOT:

- print to the console, nor
- get input from the keyboard

4.3 Designing the User Interface Layout

Now ... the user interface. We will use a GridBagLayout manager. The basic code for building the window is shown below, although we will add more to it. There is nothing tricky about the code.

```
import java.awt.*;
import javax.swing.*;
import javax.swing.*;
import javax.swing.event.*;

public class TrafficLightFrame extends JFrame {

    // These are the window's components
    private JRadioButton[]   buttons = new JRadioButton[3];
    private JButton          advButton;
    private JProgressBar     progressBar;
    private JSlider          slider;
    private JComboBox        actionList;
    private JCheckBox        autoButton;

    public TrafficLightFrame(String title) {
        super(title);
        buildWindow();
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400, 250);
    }

    // Add all components to the frame's panel
    private void buildWindow() {
        GridBagLayout         layout = new GridBagLayout();
        GridBagConstraints    constraints = new GridBagConstraints();
        setLayout(layout);

        // Add all the labels
        JLabel label = new JLabel("Manual");
        constraints.gridx = 0;
        constraints.gridy = 0;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.weightx = 0;
        constraints.weighty = 0;
        constraints.fill = GridBagConstraints.NONE;
        constraints.anchor = GridBagConstraints.NORTHWEST;
        constraints.insets = new Insets(5, 5, 0, 0);
        layout.setConstraints(label, constraints);
        add(label);

        label = new JLabel("Action");
        constraints.gridx = 1;
        layout.setConstraints(label, constraints);
        add(label);

        label = new JLabel("Advance");
        constraints.gridx = 2;
        layout.setConstraints(label, constraints);
        add(label);
```

```
label = new JLabel("Timer Progress");
constraints.gridx = 0;
constraints.gridy = 4;
layout.setConstraints(label, constraints);
add(label);


// Add the Radio Buttons
ButtonGroup lights = new ButtonGroup();
JPanel aPanel = new JPanel();
aPanel.setLayout(new BoxLayout(aPanel, BoxLayout.Y_AXIS));
aPanel.setBackground(Color.black);
for (int i=0; i<3; i++) {
    buttons[i] = new JRadioButton("", false);
    buttons[i].setBackground(Color.black);
    lights.add(buttons[i]);
    aPanel.add(buttons[i]);
}
buttons[0].setText("Red");
buttons[1].setText("Yellow");
buttons[2].setText("Green");
buttons[0].setForeground(Color.red);
buttons[1].setForeground(Color.yellow);
buttons[2].setForeground(Color.green);
constraints.gridx = 0;
constraints.gridy = 1;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
layout.setConstraints(aPanel, constraints);
add(aPanel);

// Make the Actions List
String[] actions = {"Stop", "Yield", "Go"};
actionList = new JComboBox(actions);
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridheight = 1;
constraints.weightx = 1;
constraints.fill = GridBagConstraints.HORIZONTAL;
layout.setConstraints(actionList, constraints);
add(actionList);

// Make the Slider
slider = new JSlider(JSlider.HORIZONTAL, 0, 20, 1);
slider.setMajorTickSpacing(5);
slider.setMinorTickSpacing(1);
slider.setPaintTicks(true);
slider.setPaintLabels(true);
constraints.gridx = 1;
constraints.gridy = 3;
layout.setConstraints(slider, constraints);
add(slider);

// Add the auto checkbox button
autoButton = new JCheckBox("Auto Advance");
constraints.gridx = 1;
```

```
        constraints.gridy = 2;
        constraints.fill = GridBagConstraints.BOTH;
        layout.setConstraints(autoButton, constraints);
        add(autoButton);

        // Add the Advance Picture button
        advButton = new JButton(new ImageIcon("RedLight.jpg"));
        constraints.gridx = 2;
        constraints.gridy = 1;
        constraints.gridheight = 3;
        constraints.weightx = 0;
        constraints.weighty = 1;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(5, 5, 0, 5);
        layout.setConstraints(advButton, constraints);
        add(advButton);

        // Add the progress bar
        progressBar = new JProgressBar(JProgressBar.HORIZONTAL, 0, 8);
        constraints.gridx = 0;
        constraints.gridy = 5;
        constraints.gridwidth = 3;
        constraints.gridheight = 1;
        constraints.weightx = 1;
        constraints.weighty = 2;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(5, 5, 5, 5);
        layout.setConstraints(progressBar, constraints);
        add(progressBar);
    }

    public static void main(String args[]) {
        TrafficLightFrame frame =  new TrafficLightFrame("Traffic Light");
        frame.setVisible(true);
    }
}
```

When we run the application at this point, the components all appear on the window, although the interface does not really do anything useful yet. Some interesting things to note are:

- We made an array of **JRadioButtons** so that we can access the buttons by index. This is useful since we need to change the state of a button based on the traffic light's state (which will be an integer).
- The **JRadioButtons** were added to a **JPanel** so that it is easier to keep them grouped together when the window resizes.
- The **JProgressBar** uses a range from 0 to 8. We will see more about this later.
- The **JButton** initially has a picture called **"RedLight.jpg"** representing a red traffic light. This file must be in the same directory as this code as well as two others which will be used later, called **"GreenLight.jpg"** and **"YellowLight.jpg"**.

4.4 Connecting it all Together

Now we must connect the model and the interface together to make it all work.

We begin by adding an instance variable representing the model:

```
// This is the model
private TrafficLight aTrafficLight = new TrafficLight();
```

This model MUST ALWAYS be synchronized with the user interface. That is, the user interface should always reflect perfectly the state of the traffic light. That means, upon startup, the application should show the default traffic light state in the radio buttons, the combo box and the picture on the advance button. To do this, we will make sure that our **update()** method always updates the components properly. In fact, it is best to first write the update() method BEFORE writing any event handlers. That way, the interface always reflects the model, making debugging the event handlers easier. Also, it ensures that you put your code in the correct place.

Always follow these steps:

1. **Create the model**
2. **Create your user interface "look" (i.e., frame with its components)**
3. **Write the update() method to refresh ALL components that may change their look**
4. **Write and test your event handlers one by one**

So now let us write our update method. It is often the case that we write helper methods (one for each component or group of components) to help keep our code neat and tidy. We will start by updating the radio buttons, combo box and advance button only, since they deal directly with the traffic light state:

```
// Update all relevant components according to the traffic light state
public void update() {
    updateRadioButtons();
    updateComboBox();
    updateAdvanceButton();
}
```

Notice the three helper methods. These methods should put the appropriate data into the components based on the current state of the model. These methods will also be used later on when we make changes to the model and need to reflect these changes in the window.

```
private void updateRadioButtons() {
    for (int i=0; i<3; i++)
```

```
            buttons[i].setSelected(aTrafficLight.getState() == (i+1));
    }
    private void updateComboBox() {
        actionList.setSelectedIndex(aTrafficLight.getState() - 1);
    }
    private void updateAdvanceButton() {
        String[] iconNames =
{"RedLight.jpg","YellowLight.jpg","GreenLight.jpg"};
        advButton.setIcon(new ImageIcon(iconNames[aTrafficLight.getState()-
1]));        }
    }
```

Notice a couple of things:

- Each helper update method corresponds to a single component (or group, as with radio buttons).
- Each helper update method relies on the model only, not on any other components (this is clean).

In our constructor, we will call the **update()** method (after we add the components) so that when the frame is created, the components will all be updated to reflect the initial state of the model.

```
    public TrafficLightFrame(String title) {
        super(title);
        buildWindow();

        update();

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400, 250);
    }
```

OK. When we run our window now, it should represent our default traffic light state (which is red). So, the top radio button should be selected, the combo box should indicate "Stop" and the red light image should be on the advance button.

It is now time to write the event handlers.

How can we get the radio buttons to change the state of the model ? We need to add an action listener for each button. We need to write the following in the constructor in order to register the listener for each Radio Button:

```
    // Register the JRadioButton Listeners
    for (int i=0; i<3; i++)
        buttons[i].addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                handleRadioButtonPress((JRadioButton)e.getSource());
            }
        });
```

Then, we can write the **handleRadioButtonPress()** helper method. What should it do ? Do you remember ... its simple ... change the model ... then call **update()**. But how does it change the model ? Well, the model's state should match the index of the button, shouldn't it ?

```
// This is the radio button event handler
private void handleRadioButtonPress(JRadioButton source) {
    for (int i=0; i<3; i++) {
        if (source == buttons[i])
            aTrafficLight.setState(i+1);
    }
    update();
}
```

Notice that the event handler determines which button was pressed by using **`getSource()`** and comparing this to the actual buttons by using the identity operator. Notice also, that we simply ask the model to change its state and then call **`update()`**. The **`update()`** method will take care of making sure that all other components will reflect the recent model changes. If we were to test things now, we would see that by clicking the radio buttons, the ComboBox and icon on the Advance button would be updated properly to reflect the model changes as desired.

To get this to happen for selecting combo box items as well, we merely add an **ActionListener** to the ComboBox (again in out constructor):

```
// Register the JComboBox Listener
actionList.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        handleComboBoxSelection((JComboBox)e.getSource());
    }
});
```

Now here is the helper method which simply determines the index of the selected item and updates the model accordingly

```
// The ComboBox selection event handler
private void handleComboBoxSelection(JComboBox source) {
    aTrafficLight.setState(source.getSelectedIndex() + 1);
    update();
}
```

Once again, we can use **`getSource()`** to get the component (i.e. the combo box). Alternatively, we could have just accessed the **actionList** instance variable (simpler code, but it would be dependent on the variable name ... which is not too bad). After making these changes, both the radio buttons and combo box are "synchronized" in that selection from one causes selection in the other. Neat isn't it ?

There is a slight problem. The **setSelectedIndex()** call in our **updateComboBox()** method will generate an **ActionEvent** again, leading to another call to **update()** and we are led into an endless loop as with our "todo List" example. So we need to disable the **ActionListener** for the combobox while we are updating. We can store the **ActionListener** into an instance variable when we create it and then remove/add it in the **update()** method:

```
    // Add this new instance variable
    private ActionListener    comboBoxListener;

    public TrafficLightFrame(String title) {
        ...
        actionList.addActionListener(comboBoxListener = new
ActionListener() {
            public void actionPerformed(ActionEvent e) {

handleComboBoxSelection((JComboBox)e.getSource());
            }
        });
        ...
    }

    public void update() {
        actionList.removeActionListener(comboBoxListener);
        updateRadioButtons();
        updateComboBox();
        updateAdvanceButton();
        updateProgressBar();
        actionList.addActionListener(comboBoxListener);
    }
```

So what about the Advance button ? Well, it too should advance the state of the model and then update all components. We add this code to the constructor:

```
        advButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                handleAdvanceButtonPress();
            }
        });
```

And then add this very simple helper method:

```
    // This is the Advance button event handler
    private void handleAdvanceButtonPress() {
        aTrafficLight.advanceState();
        update();
    }
```

Wow! Isn't this getting really easy now. We just ask the model to do the advancing of state and then call **update()**. Now everything is just peachy.

4.5 Hooking up the Timer

Our last step is to get the **Timer** working properly. This is actually quite easy. We will add an instance variable to hold the **Timer** so that we can grab it whenever we want to Start or Stop it:

```
private Timer  aTimer;
```

Now we must make the timer. To make one, we simply call a constructor which specifies the number of milliseconds that we would like between timer events as well as the listener (i.e., event handler). As it turns out, the listener is simply an ActionListener ... just as with JButtons. We can write the following code in our constructor to generate a timer tick twice per second (i.e., 1secons / 500ms = 2 seconds):

```
// Add a timer for automode.  Set it to go off every 500 milliseconds
aTimer = new Timer(500, new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        handleTimerTick();
    }
});
```

After doing this, the timer has NOT yet started and so no events are actually generated. We have to explicitly start and/or stop the timer with separate methods. When do we want the timer to start anyway ? Well, if the "Auto Advance" checkbox is turned on, we should start the timer. If it is turned off, we should stop the timer. We need to add an event handler for the check box. We can do this by adding the following code to the constructor:

```
autoButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        handleAutoButtonPress((JCheckBox)e.getSource());
    }
});
```

And here is the helper method:

```
// This is the Auto button event handler
private void handleAutoButtonPress(JCheckBox source) {
    if (source.isSelected())
        aTimer.start();
    else
        aTimer.stop();
    update();
}
```

As can be seen, when the checkbox is turned on, the timer is started. When turned off, it is stopped.
Let us now look at what we must do on each timer event. We can write the following code as our **TimerEventHandler**:

```
// This is the Timer event handler
private void handleTimerTick() {
    aTrafficLight.advanceState();
```

```
        update();
    }
```

Wow! It is the same code as the advance button. In fact, we could have used the exact same event handler ! If you were to test this, you would see the traffic light change state every 0.5 seconds as long as the check box remains selected. Once turned off, the advancing stops. Notice that the advance button will still cause the light to change state as well.

We forgot one of our criteria ... we must have the lights remain in a certain state for different amounts of time. Remember, red for 6 seconds, green for 8 seconds, then yellow for 3 seconds. We will have to keep a counter of some kind to keep track of how long the light has been in the current state. Once it has been on long enough, we advance.

But doesn't this have something to do with the model ? Shouldn't the traffic light itself know how long to remain in each state ? It looks like we may want to adjust our model. We will need to add some kind of counter to our model that counts how long the traffic light remains in a certain state. We can add the following instance variable to our **TrafficLight** model class:

```
    private int     stateCount;    // amount of time in this
state
```

We can also choose a maximum amount of time that the traffic light remains in any particular state. This can be used by the progress bar later. We will define the following static constant:

```
    public static final int  MAX_TIME_COUNT = 8;
```

We should set the **stateCount** to zero in our constructor and reset this counter to zero whenever we make state changes.

```
    public TrafficLight() {
        currentState = 1;
        stateCount = 0;
    }
    public int advanceState() {
        currentState = ++currentState % 3 + 1;
        stateCount = 0;
        return currentState;
    }
    public void setState(int newState) {
        if ((newState > 0) && (newState <4)) {
            currentState = newState;
            stateCount = 0;
        }
    }
```

We will likely want a get method for this counter too:

```
public int getStateCount() {
    return stateCount;
}
```

Now how (and when) do we go about making the traffic light advance automatically over time ? Well, we will let the user interface decide on how fast to make the timer go, but we will want a method in our model that "simulates" the passing of time. We can write a method called advanceTime() that will be called once per time unit (e.g., once per second) to advance the time (i.e., our stateCount counter). So then in this method we just have to check and see if the counter reached its limit according to what state it is in:

```
// Simulate a single time unit of time passing by
public void advanceTime() {
    // advance the time spent in the current state
    stateCount++;

    // The number of seconds (i.e., time units) that
    // the traffic light remains in each state
    int[] stateTimes = {6, 3, 8};

    // Check if we have reached time limit for current state
    if (stateCount > stateTimes[currentState-1])
        advanceState();
}
```

So now we can get back to our user interface. We need to change the call from **advanceState()** to **advanceTime()** in our timer tick event handler:

```
private void handleTimerTick() {
    aTrafficLight.advanceTime();
    update();
}
```

So what about the progress bar ? How can we have it reflect the current count ? We first need to go back to our code that built our window and now use the static constant that we defined in our model representing the maximum limit for the progress bar:

```
progressBar = new JProgressBar(JProgressBar.HORIZONTAL, 0,
TrafficLight.MAX_TIME_COUNT);
```

We will need to update the progress bar every time that the traffic light increments its stateCount. We will write an update method for this:

```
// Update all relevant components according to the traffic light state
public void update() {
    updateRadioButtons();
    updateComboBox();
    updateAdvanceButton();
```

```
        updateProgressBar();

    }

    // Update the progress bar
    private void updateProgressBar() {
        progressBar.setValue(aTrafficLight.getStateCount());
    }
```

That's it. The progress bar will now show the amount of time that the traffic light remains in its current state.

The last remaining task for us to complete is to get the slider working. We need to register for a **stateChanged** event. We can add this to our constructor:

```
    slider.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
            handleSlider((JSlider)e.getSource());
        }
    });
```

Now of course, we need to write the event handler. We only want to make a change to the timer delay when the user lets go of the slider. So while the user is adjusting the value, we do not want to handle the event. We can check for this with a **getValueIsAdjusting()** method call to our slider. Then, we can get the value of the slider with **getValue()**. It will return an integer within the range that we specified when creating the slider (from 0 to 20 in our case). Lastly, we just need to call **setDelay(int)** for our timer, passing it in the new delay value. Of course, if the delay is 0, we probably want to pick a HUGE delay such as Integer.MAX_VALUE. Also, we will need to restart the timer after making the change (as long as it has already been started by the check box):

```
    // This is the Slider event handler
    private void handleSlider(JSlider source) {
        if (!source.getValueIsAdjusting()) {
            int delay = source.getValue();
            if (delay > 0) {
                aTimer.setDelay(1000/delay);
                if (aTimer.isRunning())
                    aTimer.restart();
            }
            else
                aTimer.setDelay(Integer.MAX_VALUE);
            update();
        }
    }
```

Well ... that is it! We are done.

4.6 Splitting up the Model, View and Controller

Taking a look at our traffic light application, we notice a couple of things:

- The **TrafficLightFrame** class mixes together code related to
  - how the interface looks and
  - how the interface behaves.
- There is a problem if some other "entity" changes the **TrafficLight** model.....then the **TrafficLightFrame** GUI will not be updated.

We will now make a distinction between a view and a controller.

Recall that a ***view*** is:

- the part of the application that specifies how the model is shown visually.
- the part of the code that deals with the appearance of the interface.

and a ***controller*** is:

- the part of the application that specifies how the model interacts with the view.
- the part of the code that deals with the behaviour of the interface.
- code that serves as a "mediator" between the model and the view.

Also, recall that it is ALWAYS a good idea to separate the model, view and controller (MVC):

- code is cleaner and easier to follow when the view and controller are separated
- we may want to have multiple views and controllers on the same model.

Consider this second point for a moment. If we have multiple controllers and views on the same model (i.e., two windows for the same traffic light), then we have two different windows affecting the model.

How can we cleanly allow one view to change the model and have the other view updated automatically ?

We could have each view/controller keep track of all other views and controllers. This could get messy, especially if we get more controllers and views later on down the road. A nice solution is to have the model inform all interested parties whenever it has changed. Things brings up the notion of a commonly used Object-Oriented Design pattern called the "**subject/observer**" pattern.

If we make our model inform all interested applications when it has changed, then it must somehow keep track of all applications that need to be updated when a change occurs. The model becomes a kind of ***subject*** which the applications ***observe***. To make a clean connection

between the two, we will make our model class implement a standardized **Subject** interface that allows observers to register or un-register with it. We will write the following interface:

```
public interface Subject {
    public void registerObserver(Observer observer);
    public void unregisterObserver(Observer observer);
}
```

The model should therefore keep track of the applications that have registered with it so that it can later update them. The applications should make sure that they have an **update()** method so that this will all work. To do this, we will have the applications implement an **Observer** interface.

```
public interface Observer {
    public void update();
}
```

As a result, we will have to change the model code so that it:

- keeps a list of all Observers that have registered
- informs all Observers whenever it changes (i.e., whenever its instance variables change)

We will also have to go through our interface code and:

- split up the view and controller into two separate classes
- remove all the calls to **update()** from the controller (except maybe one for initializing)

The Model:

Let us take a look at how the model now looks. Notice the changes highlighted:

```
// Making the TrafficLight model implement the Subject interface allows
// it to inform all of the observers whenever there has been a change
import java.util.ArrayList;
public class TrafficLight implements Subject {

    public static final int  MAX_TIME_COUNT = 8;

    private int  currentState;  // 1=red, 2=yellow, 3=green
    private int  stateCount;    // amount of time in this state

   ArrayList<Observer>     observers = new ArrayList<Observer>();

    // Constructor that makes a red traffic light
    public TrafficLight() {
        currentState = 1;
        stateCount = 0;
    }

    // Advance the traffic light to the next state
    public int advanceState() {
```

```
        currentState = ++currentState % 3 + 1;
        stateCount = 0;

        updateObservers(); // Tell the observer applications about this change

        return currentState;
    }

    // Simulate a single time unit of time passing by
    public void advanceTime() {
        // The number of seconds (i.e., time units) that
        // the traffic light remains in each state
        int[] stateTimes = {6, 3, 8};

        // advance the time spent in the current state
        stateCount++;

        updateObservers(); // Tell the observer applications about this change

        // Check if we have reached time limit for current state
        if (stateCount > stateTimes[currentState-1])
            advanceState();
    }

    // Return the amount of time spent in the current state
    public int getStateCount() {
        return stateCount;
    }

    // Return the state of the traffic light (as a number from 1 to 3)
    public int getState() {
        return currentState;
    }

    // Set the state of the traffic light (as a number from 1 to 3)
    // If the integer is out of range, do nothing
    public void setState(int newState) {
        if ((newState > 0) && (newState <4)) {
            currentState = newState;
            stateCount = 0;

            updateObservers(); // Tell the observer applications about this change

        }
    }

    // Return a string representation of the traffic light
    public String toString() {
        String[] colours = {"Red", "Yellow", "Green"};
        return colours[currentState] + " Traffic Light";
    }

   public void registerObserver(Observer observer) {
```

```
        observers.add(observer);
    }

    public void unregisterObserver(Observer observer) {
        observers.remove(observer);
    }

    // This method is called whenever there is a change to the model.
    // It informs all registered observer applications of the change.
    private void updateObservers() {
        for (Observer anObserver: observers)
            anObserver.update();
    }

}
```

The View:

What portion of code represents the view ? All of the stuff related to adding Frames/Panels/Components etc.. We can take the **TrafficLightFrame** class and "strip away" all of the behaviour and model related stuff (i.e., remove the Listeners etc..) If we do this, then the controller MUST add the behaviour-related stuff (i.e., event handlers, updates, listeners etc..).

One problem is that the controller must be able to access the view's components in order to add listeners, get their contents, change them etc.. One solution to this problem is to make instance variables for all the components and then supply **public** "**get**" methods for each.

We will make our views as separate **JPanels** that hold the entire contents of the window:

```
import java.awt.*;
import javax.swing.*;
public class TrafficLightPanel extends JPanel {

    // These are the components
    private JRadioButton[]   buttons = new JRadioButton[3];
    private JButton          advButton;
    private JProgressBar     progressBar;
    private JSlider          slider;
    private JComboBox        actionList;
    private JCheckBox        autoButton;

    // Make some get methods so that the controller can access this view
    public JRadioButton  getButton(int i)   { return buttons[i]; }
    public JButton       getAdvanceButton() { return advButton; }
    public JProgressBar  getProgressBar()   { return progressBar; }
    public JSlider       getSlider()        { return slider; }
    public JComboBox     getActionList()    { return actionList; }
    public JCheckBox     getAutoButton()    { return autoButton; }

    public TrafficLightPanel() {
        GridBagLayout         layout = new GridBagLayout();
```

```
GridBagConstraints    constraints = new GridBagConstraints();
setLayout(layout);

// Add all the labels
JLabel label = new JLabel("Manual");
constraints.gridx = 0;
constraints.gridy = 0;
constraints.gridwidth = 1;
constraints.gridheight = 1;
constraints.weightx = 0;
constraints.weighty = 0;
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.NORTHWEST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(label, constraints);
add(label);

label = new JLabel("Action");
constraints.gridx = 1;
layout.setConstraints(label, constraints);
add(label);

label = new JLabel("Advance");
constraints.gridx = 2;
layout.setConstraints(label, constraints);
add(label);

label = new JLabel("Timer Progress");
constraints.gridx = 0;
constraints.gridy = 4;
layout.setConstraints(label, constraints);
add(label);


// Add the Radio Buttons
ButtonGroup lights = new ButtonGroup();
JPanel aPanel = new JPanel();
aPanel.setLayout(new BoxLayout(aPanel, BoxLayout.Y_AXIS));
aPanel.setBackground(Color.black);
for (int i=0; i<3; i++) {
    buttons[i] = new JRadioButton("", false);
    buttons[i].setBackground(Color.black);
    lights.add(buttons[i]);
    aPanel.add(buttons[i]);
}
buttons[0].setText("Red");
buttons[1].setText("Yellow");
buttons[2].setText("Green");
buttons[0].setForeground(Color.red);
buttons[1].setForeground(Color.yellow);
buttons[2].setForeground(Color.green);
constraints.gridx = 0;
constraints.gridy = 1;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
layout.setConstraints(aPanel, constraints);
add(aPanel);
```

```
        // Make the Actions List
        String[] actions = {"Stop", "Yield", "Go"};
        actionList = new JComboBox(actions);
        constraints.gridx = 1;
        constraints.gridy = 1;
        constraints.gridheight = 1;
        constraints.weightx = 1;
        constraints.fill = GridBagConstraints.HORIZONTAL;
        layout.setConstraints(actionList, constraints);
        add(actionList);

        // Make the Slider
        slider = new JSlider(JSlider.HORIZONTAL, 0, 20, 1);
        slider.setMajorTickSpacing(5);
        slider.setMinorTickSpacing(1);
        slider.setPaintTicks(true);
        slider.setPaintLabels(true);
        constraints.gridx = 1;
        constraints.gridy = 3;
        layout.setConstraints(slider, constraints);
        add(slider);

        // Add the auto checkbox button
        autoButton = new JCheckBox("Auto Advance");
        constraints.gridx = 1;
        constraints.gridy = 2;
        constraints.fill = GridBagConstraints.BOTH;
        layout.setConstraints(autoButton, constraints);
        add(autoButton);

        // Add the Advance Picture button
        advButton = new JButton(new ImageIcon("RedLight.jpg"));
        constraints.gridx = 2;
        constraints.gridy = 1;
        constraints.gridheight = 3;
        constraints.weightx = 0;
        constraints.weighty = 1;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(5, 5, 0, 5);
        layout.setConstraints(advButton, constraints);
        add(advButton);

        // Add the progress bar
        progressBar = new JProgressBar(JProgressBar.HORIZONTAL, 0,
TrafficLight.MAX_TIME_COUNT);
        constraints.gridx = 0;
        constraints.gridy = 5;
        constraints.gridwidth = 3;
        constraints.gridheight = 1;
        constraints.weightx = 1;
        constraints.weighty = 2;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(5, 5, 5, 5);
        layout.setConstraints(progressBar, constraints);
        add(progressBar);
```

```
    }
}
```

Notice the following:

- This "view" code has no listener stuff, nor event handlers nor update methods
- We do not need to import the event packages anymore
- All of the instance variables related to the **Timer** have been removed (since this is behaviour related)
- There is no main method

The Controller:

What portion of code represents the controller ? All of the stuff related to adding listeners, event handlers, update methods etc.. as well as any behaviour-related code such as the **Timer** code. All of the stuff that we removed from the **View** must be added here. We will put it all in the **TrafficLightFrame** class that will serve as a *Mediator* between the model and the view. It will also contain the "main" method which will be responsible for coordinating the startup of the application. The controller will keep hold of the model and the view (as instance variables) as part of this coordination.

Here is the code:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class TrafficLightFrame extends JFrame implements
Observer {

    private TrafficLight        aTrafficLight;
    private Timer               aTimer;

   // This is the view
   private TrafficLightPanel
aView;

    private ActionListener    comboBoxListener;

    public TrafficLightFrame(String title, TrafficLight
model, TrafficLightPanel view) {
        super(title);
```

```
        aTrafficLight = model;

      aView = view;
      setContentPane(aView);
    //Replace old panel with ours

        // Add the Listeners
        for (int i=0; i<3; i++)
            aView.getButton(i).addActionListener(new
ActionListener() {
                public void actionPerformed(ActionEvent e) {

handleRadioButtonPress((JRadioButton)e.getSource());
                }
            });


aView.getActionList().addActionListener(comboBoxListener =
new ActionListener() {
            public void actionPerformed(ActionEvent e) {

handleComboBoxSelection((JComboBox)e.getSource());
            }
        });
        aView.getSlider().addChangeListener(new
ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                handleSlider((JSlider)e.getSource());
            }
        });
        aView.getAutoButton().addActionListener(new
ActionListener() {
            public void actionPerformed(ActionEvent e) {

handleAutoButtonPress((JCheckBox)e.getSource());
            }
        });
        aView.getAdvanceButton().addActionListener(new
ActionListener() {
            public void actionPerformed(ActionEvent e) {
                handleAdvanceButtonPress();
            }
        });

        // Add a timer for automode.  Set it to go off every
500 milliseconds
        aTimer = new Timer(500, new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
                handleTimerTick();
            }
        });

        // !!! IMPORTANT !!!
        // Register with the model so that when it
    changes, we get informed
        aTrafficLight.registerObserver(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400, 250);
    }


    // This is the Timer event handler
    private void handleTimerTick() {
        aTrafficLight.advanceTime();
        // update() has been removed now
    }

    // This is the Advance button event handler
    private void handleAdvanceButtonPress() {
        aTrafficLight.advanceState();
        // update() has been removed now
    }

    // This is the Auto button event handler
    private void handleAutoButtonPress(JCheckBox source) {
        if (source.isSelected())
            aTimer.start();
        else
            aTimer.stop();
        // update() has been removed now
    }

    // This is the radio button event handler
    private void handleRadioButtonPress(JRadioButton source)
{
        for (int i=0; i<3; i++) {
            if (source == aView.getButton(i))
                aTrafficLight.setState(i+1);
        }
        // update() has been removed now
    }

    // The ComboBox Selection event handler
```

```
    private void handleComboBoxSelection(JComboBox source) {
        aTrafficLight.setState(source.getSelectedIndex() +
1);
        // update() has been removed now
    }

    // This is the Slider event handler
    private void handleSlider(JSlider source) {
        if (!source.getValueIsAdjusting()) {
            int delay = source.getValue();
            if (delay > 0) {
                aTimer.setDelay(1000/delay);
                if (aTimer.isRunning())
                    aTimer.restart();
            }
            else {
                aTimer.setDelay(Integer.MAX_VALUE);
            }
            // update() has been removed now
        }
    }

    // Update the radio buttons according to the traffic
light state
    private void updateRadioButtons() {
        for (int i=0; i<3; i++) {

aView.getButton(i).setSelected(aTrafficLight.getState() ==
(i+1));
        }
    }

    // Update the status pane according to the traffic light
state
    private void updateAdvanceButton() {
        String[] iconNames =
{"RedLight.jpg","YellowLight.jpg","GreenLight.jpg"};
        aView.getAdvanceButton().setIcon(new
ImageIcon(iconNames[aTrafficLight.getState()-1]));
    }


    // Update the combo box according to the traffic light
state
    private void updateComboBox() {

aView.getActionList().setSelectedIndex(aTrafficLight.getStat
```

```
e() - 1);
    }


    // Update the progress bar
    private void updateProgressBar() {

aView.getProgressBar().setValue(aTrafficLight.getStateCount(
));
    }

    // Update all relevant components according to the
traffic light state
    public void update() {

aView.getActionList().removeActionListener(comboBoxListener)
;
        updateRadioButtons();
        updateComboBox();
        updateAdvanceButton();
        updateProgressBar();

aView.getActionList().addActionListener(comboBoxListener);
    }

    public static void main(String args[]) {
        TrafficLight aModel = new TrafficLight();

        // Instantiate three controllers each with their own
views
        // Two controllers (A and B) will share the same
model while
        // the 3rd will be alone with its own model.
        new TrafficLightFrame("Traffic Light A (tied with
B)", aModel,
                                    new
TrafficLightPanel()).setVisible(true);
        new TrafficLightFrame("Traffic Light B (tied with
A)", aModel,
                                    new
TrafficLightPanel()).setVisible(true);
        new TrafficLightFrame("Traffic Light C (alone)", new
TrafficLight(),
                                    new
TrafficLightPanel()).setVisible(true);
    }
}
```