Model View Controller

Dr Ian Bayley [1]

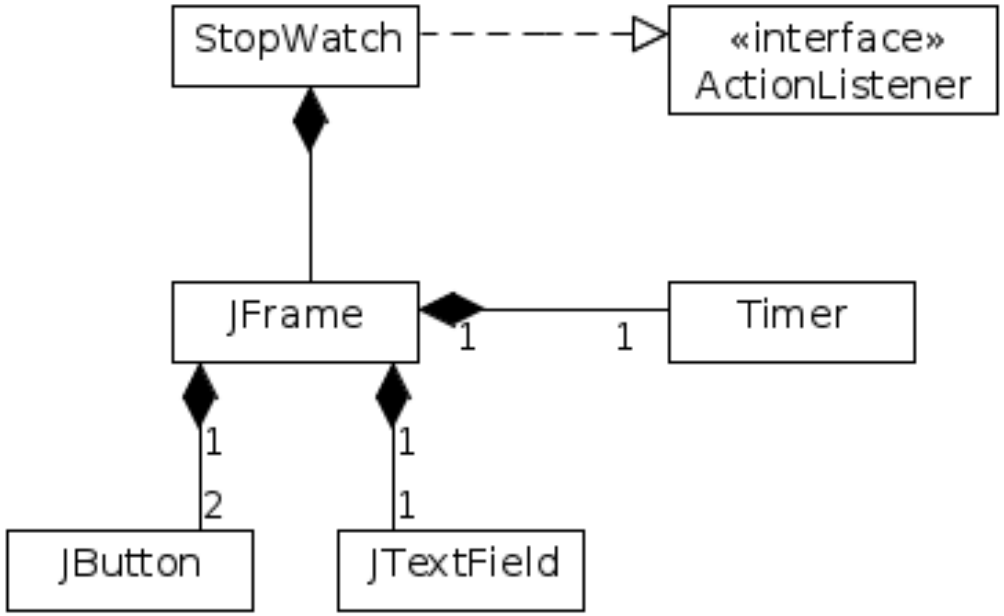
Oxford Brookes

Advanced O-O Prog

[1] with thanks to David Sutton

Outline

Screenshot and Design of Stopwatch Program



Code for Stopwatch Program I

```
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.NumberFormat;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.Timer;

public class StopWatch implements ActionListener {
  private JFrame frame = new JFrame("Digital Stopwatch");
  private JPanel buttonPanel = new JPanel();
```

Code for Stopwatch Program II

```
private JButton startButton = new JButton("Start");
private JButton stopButton = new JButton("Stop");
private JTextField timeField = new JTextField("");
private Timer timer;
private long startTime;

public StopWatch() {
  buttonPanel.setLayout(
    new BoxLayout(buttonPanel, BoxLayout.X_AXIS));
  buttonPanel.add(startButton);
  buttonPanel.add(stopButton);
  stopButton.setEnabled(false);
  startButton.addActionListener(this);
  stopButton.addActionListener(this);
  Container contentPane = frame.getContentPane();
  contentPane.setLayout(
```

Code for Stopwatch Program III

```
    new BoxLayout(contentPane, BoxLayout.Y_AXIS));
  timeField.setEditable(false);
  contentPane.add(timeField);
  contentPane.add(buttonPanel);
  timer = new Timer(100,this);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.pack();
  frame.setVisible(true);
}

public void actionPerformed(ActionEvent e) {
  if (e.getSource() == startButton) {
    startTime = e.getWhen();
    timer.start();
    stopButton.setEnabled(true);
    startButton.setEnabled(false);
```

Code for Stopwatch Program IV

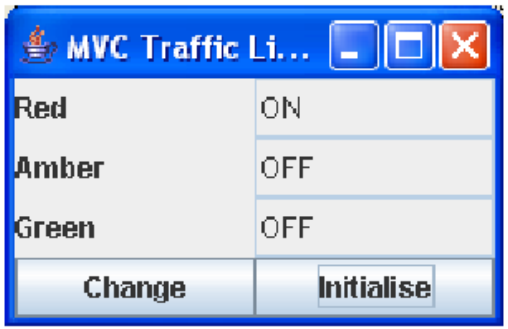
```
    }
    if (e.getSource() == timer) {
      setTimeField();
    }
    if (e.getSource() == stopButton) {
      setTimeField();
      timer.stop();
      stopButton.setEnabled(false);
      startButton.setEnabled(true);
    }
  }
  private void setTimeField() {
    long elapsed = System.currentTimeMillis() - startTime;
    long centisecs = elapsed/10;
    long seconds = centisecs/100;
    long minutes = seconds/60;
```

Code for Stopwatch Program V

```
    long hours = minutes/60;
    NumberFormat nf = NumberFormat.getNumberInstance();
    nf.setMinimumIntegerDigits(2);
    String time = "" + nf.format(hours) + ":" +
      nf.format(minutes%60) + ":" +
      nf.format(seconds%60) + "." +
      nf.format(centisecs%100);
    timeField.setText(time);
  }
  public static void main(String[] args) {
    new StopWatch();
  }
```

Outline

Screenshot and Code of Traffic Light Program




```
class TrafficLight extends JFrame {
    void change() {
        if (greenField.getText.equals(“ON”)) {
            greenField.setText(“OFF”);
            amberField.setText(“ON”);
        }
    }
}
```

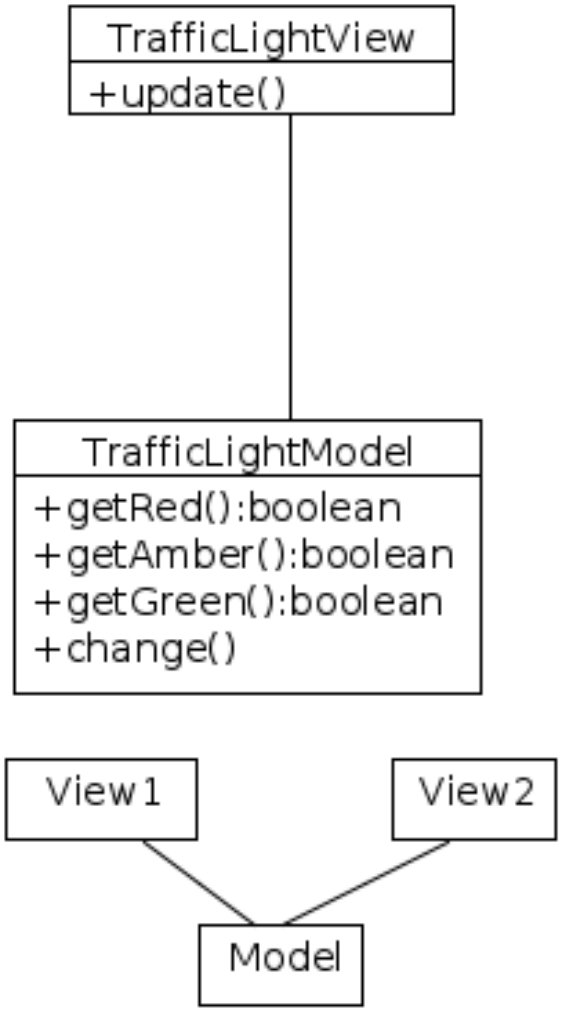
Problem with Traffic Light Program

- the code that models the state and changes it is mixed up with code that displays the state in the form of lights
- if I change how lights are displayed (eg coloured circles) then I need to modify the code that changes the state
- in general, we are violating an important Design Principle: that each class should only have one responsibility
- so there should be one class for the state (called the model) and one class for the view
- when the state changes, then the view need not change and vice versa

Outline

Separating Model and View

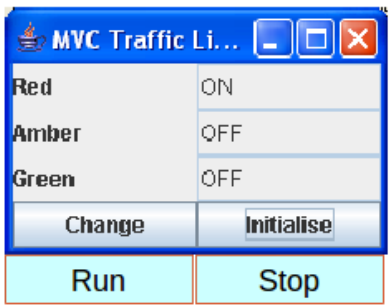
- responsibility of the view:
  - display lights by observing model
  - update the lights when the model changes
- responsibility of the model:
  - store the state
  - change the state when required
- model calls `update` to say it has changed
- view calls `getRed` etc to see what the change is

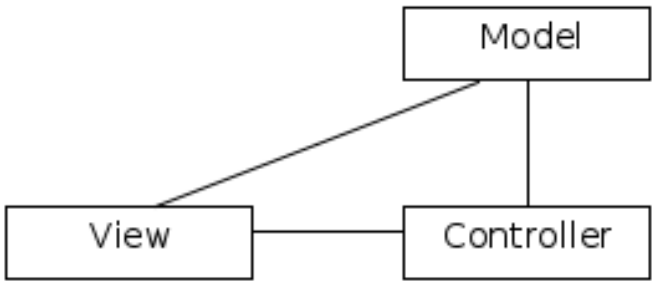


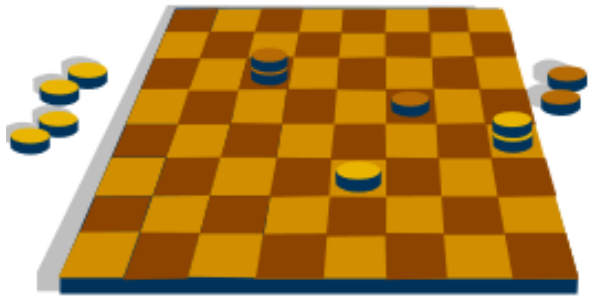
- note that model does not depend on view
- ie it makes no assumptions about it so we can have many views

Separating Controller from Model and View

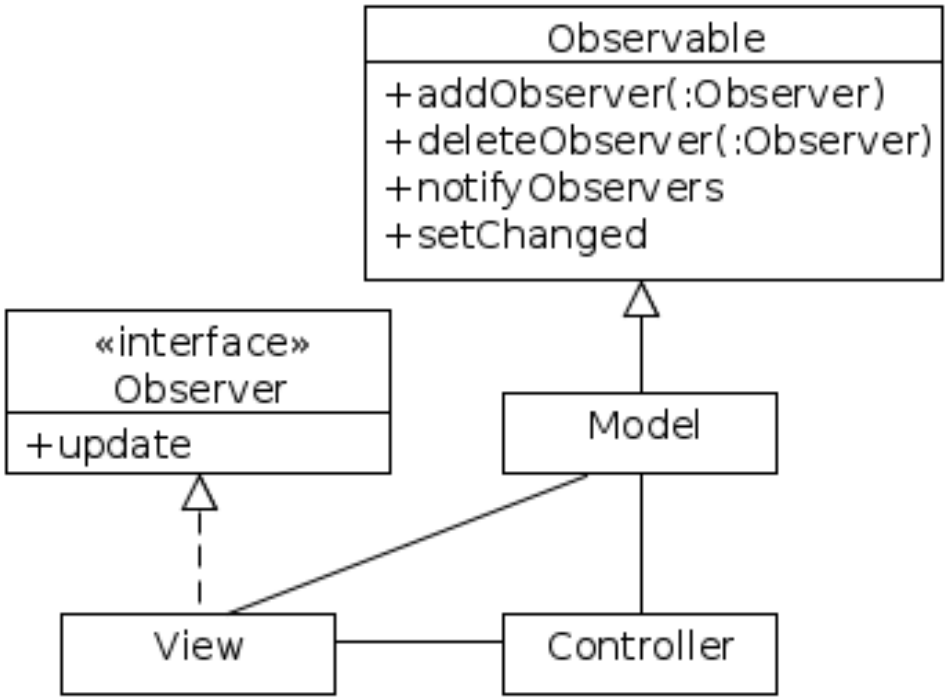
- Controller is a separate class with responsibility for converting mouse clicks into methods that change the model
- it may decide not to convert some mouse clicks
- it may also change the view eg by disabling buttons
  - eg Run Button disable Change and initialise but enable Stop button
- if we were to implement a game of draughts using MVC, what would be the responsibility of Model, View, and Controller?







Implementing MVC in Java



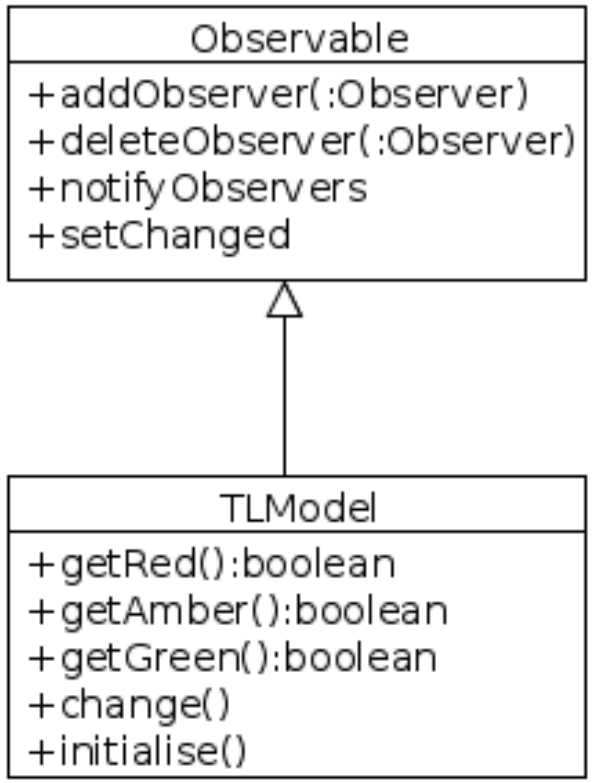
- an instance of the Observer pattern (see later) supported by the Java class library

1. `View` subscribes by calling `addObserver`
2. (later) `Model` publishes by calling `update`
3. View requests new state by calling `getGreen` etc

Outline

Code and Class Diagram for Traffic Light Model I

```
import java.util.Observable;
public class TLModel extends Observable {
  private boolean red;
  private boolean amber;
  private boolean green;
  public boolean getRed() {
    return red;
  }
  public boolean getAmber() {
    return amber;
  }
  public boolean getGreen() {
    return green;
  }
```



Code and Class Diagram for Traffic Light Model II

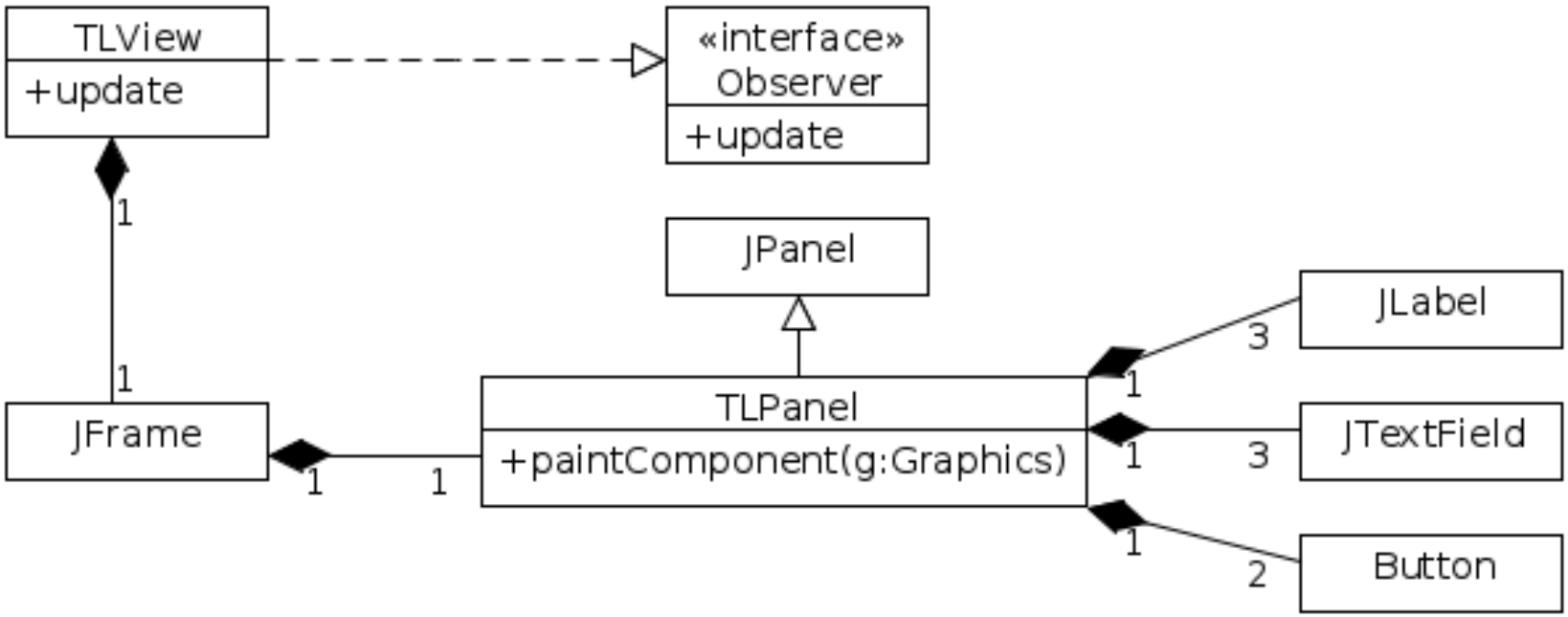
```
public void change(){
  if (red && !amber && !green) {
    amber = true;
  } else if (red && amber && !green){
    red = false; amber = false; green = true;
  } else if (!red && !amber && green){
    green = false; amber = true;
  } else { //just assume amber (is this wise?)
    red = true; amber = false; green = false;
  }
  setChanged();
  notifyObservers();
}
```

Code and Class Diagram for Traffic Light Model III

```
  public void initialise() {
    red = true;
    amber = false;
    green = false;
    setChanged();
    notifyObservers();
  }
  public TLModel() {
    initialise();
  }
}
```

Outline

Class Diagram of Traffic Light View



Code for Traffic Light View I

```
import java.util.Observer;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class TLView implements Observer, ActionListener {
  private static final Dimension PANEL_SIZE =
    new Dimension(200,200);
  private TLModel model;
  private TLController controller;
  private JFrame frame;
  private JPanel panel;
  private JTextField redField = new JTextField(3);
  private JTextField amberField = new JTextField(3);
  private JTextField greenField = new JTextField(3);
```

Code for Traffic Light View II

```
private JLabel redLabel = new JLabel("Red");
private JLabel amberLabel = new JLabel("Amber");
private JLabel greenLabel = new JLabel("Green");
private JButton changeButton = new JButton("Change");
private JButton initialiseButton =
  new JButton("Initialise");
public TLView(TLModel model, TLController controller) {
  this.model = model;
  model.addObserver(this);
  this.controller = controller;
  createControls();
  controller.setView(this);
  update(model, null);
}
public void createControls() {
  frame = new JFrame("MVC Traffic Light Example");
```

Code for Traffic Light View III

```
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPane = frame.getContentPane();
    contentPane.setLayout(
      new BoxLayout(contentPane, BoxLayout.X_AXIS));
    createPanel();
    contentPane.add(panel);
    frame.pack();
    frame.setResizable(false);
    frame.setVisible(true);
  }
  private void createPanel() {
    panel = new JPanel();
    panel.setLayout(new GridLayout(4,2));
    redField.setEditable(false);
    amberField.setEditable(false);
    greenField.setEditable(false);
```

Code for Traffic Light View IV

```
    panel.add(redLabel);
    panel.add(redField);
    panel.add(amberLabel);
    panel.add(amberField);
    panel.add(greenLabel);
    panel.add(greenField);
    changeButton.addActionListener(this);
    panel.add(changeButton);
    initialiseButton.addActionListener(this);
    panel.add(initialiseButton);
    panel.setPreferredSize(PANEL_SIZE);
  }
  public void update(java.util.Observable o, Object arg) {
    redField.setText(model.getRed()?"ON":"OFF");
    amberField.setText(model.getAmber()?"ON":"OFF");
    greenField.setText(model.getGreen()?"ON":"OFF");
```

Code for Traffic Light View V

```
    frame.repaint();
  }
  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == initialiseButton)
      controller.initialise();
    else if (event.getSource() ==changeButton)
      controller.change();
  }
}
```

Outline

Code for Traffic Light Controller

```
public class TLController {
  private TLModel model;
  private TLView view;
  public TLController(TLModel model) {
    this.model = model;
  }
  public void setView(TLView view) {
    this.view = view;
  }
  public void change() {
    model.change();
  }
  public void initialise() {
    model.initialise();
  }
}
```

Outline

Code for Traffic Light Top Level I

```
public class TrafficLightDemo {
  public static void main(String [] args) {
    TLModel model = new TLModel();
    TLController controller = new TLController(model);
    TLView view = new TLView(model, controller);
  }
}
public class TrafficLightDemo {
  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(
      new Runnable() {
        public void run () {createAndShowGUI();}
        }
    );
```

Code for Traffic Light Top Level II

```
  }

public static void createAndShowGUI() {
  TLModel model = new TLModel();
  TLController controller = new TLController(model);
  TLView view = new TLView(model, controller);
  }
}
```

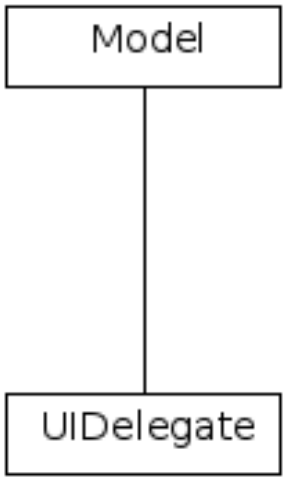
Pros and Cons of MVC

- can change the View without changing the Model and vice versa
  - View changes more often and varies between devices
  - Designing the View requires different (HCI) skills from coding the Model so someone else could do it
- it is easy to have multiple views
- It is easy to automatically test the model
  but difficult to test the view
  and also difficult to test the view and model combined

- However, code is more complex than the alternative
  and less efficient because every change to the model changes every part of the view

Alternatives to MVC and Java library support

- As discussed later, either the View pulls information about the changes from the Model or the Model pushes the information to the View as part of the update notification
- Model-Delegate is an alternative to Model View Controller



The Model-Delegate pattern merges the View and Controller into one class as we see with library text boxes and tables

- a disadvantage of the Java library support for the Observer pattern is that any class that extends `Observable` cannot extend another class
- in particular, you cannot delegate to `Observable` because the `setChanged` method is protected