Model-View-Controller
The hard problem in O-O programming is deciding what objects to have, and what their responsibilities are.

*Design Patterns* describe the higher-level organization of solutions to common problems.

Design patterns are a major topic in O-O design.
The MVC pattern

- **MVC** stands for Model-View-Controller
- The **Model** is the actual internal representation
- The **View** (or a View) is a way of looking at or displaying the model
- The **Controller** provides for user input and modification
- These three components are usually implemented as separate classes
The Model

- Most programs are supposed to do work, not just be "another pretty face"
  - but there are some exceptions
  - useful programs existed long before GUIs
- The **Model** is the part that does the work--it *models* the actual problem being solved
- The **Model** should be independent of both the Controller and the View
  - But it provides services (methods) for them to use
- Independence gives flexibility, robustness
The Controller

- The Controller decides what the model is to do
- Often, the user is put in control by means of a GUI
  - in this case, the GUI and the Controller are often the same
- The Controller and the Model can almost always be separated (what to do versus how to do it)
- The design of the Controller depends on the Model
- The Model should not depend on the Controller
The View

- Typically, the user has to be able to see, or view, what the program is doing
- The View shows what the Model is doing
  - The View is a passive observer; it should not affect the model
- The Model should be independent of the View, but (but it can provide access methods)
- The View should not display what the Controller thinks is happening
Combining Controller and View

- Sometimes the Controller and View are combined, especially in small programs.
- Combining the Controller and View is appropriate if they are very interdependent.
- The Model should still be independent.
- *Never* mix Model code with GUI code!
Separation of concerns

- As always, you want code independence
- The Model should not be contaminated with control code or display code
- The View should represent the Model as it really is, not some remembered status
- The Controller should *talk to* the Model and View, not *manipulate* them
  - The Controller can set variables that the Model and View can read
The “Reverser” program

- In this program we combine the Controller and the View (indeed, it’s hard to separate them)
- The Model, which does the computation (reversing the string), we put in a separate class
A bunch of `import` statements, then...

```java
public class ReverserGUI extends JFrame {
    ReverserModel model = new ReverserModel();
    JTextField text = new JTextField(30);
    JButton button = new JButton("Reverse");

    public static void main(String[] args) {
        ReverserGUI gui = new ReverserGUI();
        gui.create();
        gui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    ...
}
```
The **create** method

```java
private void create() {
   setLayout(new GridLayout(2, 1));
    add(text);
    add(button);

    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
            String s = text.getText();
            s = model.reverse(s);
            text.setText(s);
        }
    });

    pack();
    setVisible(true);
}
```
public class ReverserModel {

    public String reverse(String s) {
        StringBuilder builder = new StringBuilder(s);
        builder.reverse();
        return builder.toString();
    }

}
The Bouncing Ball Applet

- Each click of the Step button advances the ball a small amount
- The step number and ball position are displayed in the status line
The Ball Applet: Model

- The Ball Applet shows a ball bouncing in a window
- The Model controls the motion of the ball
- In this example, the Model must know the size of the window
  - so it knows when the ball should be made to bounce
- The Model doesn’t need to know anything else about the GUI
java.util provides an Observer interface and an Observable class

An Observable is an object that can be “observed”

An Observer is “notified” when an object that it is observing announces a change

Here’s an analogy:
- An Observable is like a Button
- An Observer is like a Listener
- You have to “attach” a Listener to a Button

Another analogy:
- An Observable is like a bulletin board
- An Observer is like someone who reads the bulletin board
An **Observable** is an object that can be “observed”

An **Observer** is “notified” when an object that it is observing announces a change

- When an **Observable** wants the “world” to know about what it has done, it executes:
  - `setChanged();`
  - `notifyObservers(); /* or */ notifyObservers(arg);`
  - The `arg` can be any object

- The **Observable** doesn’t know or care “who is looking”

- But you have attach an **Observer** to the **Observable** with:
  - `myObservable.addObserver(myObserver);`
  - This is best done in the controller class – *not in the model class!*
- **Observer** is an interface
- An **Observer** implements
  
  ```java
  public void update(Observable obs, Object arg)
  ```
- This method is invoked whenever an **Observable** that it is “listening to” does an `addNotify()` or `addNotify(arg)`
- The *obs* argument is a reference to the observable object itself
- If the **Observable** did `addNotify()`, the *arg* is **null**
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td></td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td></td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>Model</td>
<td>No collaborators...</td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------</td>
<td></td>
</tr>
<tr>
<td>Set initial position</td>
<td>....but provide access methods to allow view to see what is going on</td>
<td></td>
</tr>
<tr>
<td>Move one step</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
import java.util.Observable;

class Model extends Observable {
    final int BALL_SIZE = 20;
    int xPosition = 0;
    int yPosition = 0;
    int xLimit, yLimit;
    int xDelta = 6;
    int yDelta = 4;
    // more...
void makeOneStep() {
    xPos = xPos + xDelta;
    if (xPos < 0) {
        xPos = 0;
        xDelta = -xDelta;
    }
    // more...
if (xPosition >= xLimit) {
    xPosition = xLimit;
    xDelta = -xDelta;
}

// still more...
yPosition += yDelta;
    if (yPosition < 0 || yPosition >= yLimit) {
        yDelta = -yDelta;
        yPosition += yDelta;
    }
    setChanged();
    notifyObservers();
}  // end of makeOneStep method
}  // end of Model class
<table>
<thead>
<tr>
<th>Model</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set initial position</td>
<td>No collaborators...</td>
</tr>
<tr>
<td>Move one step</td>
<td>....but provide access methods to allow view to see what is going on</td>
</tr>
</tbody>
</table>
The Ball Applet: View

- The View needs access to the ball’s state (in this case, its x-y location)
- For a static drawing, the View doesn’t need to know anything else
Get necessary info from Model

Paint the ball

View
import java.awt.*;
import java.util.*;

class View extends Canvas implements Observer {
    Controller controller;
    Model model;
    int stepNumber = 0;

    View (Model model) {
        this.model = model;
    }

    // more...
public void paint(Graphics g) {
    g.setColor(Color.red);
    g.fillOval(model.xPosition, model.yPosition,
               model.BALL_SIZE, model.BALL_SIZE);
    controller.showStatus("Step " +
                         (stepNumber++) +
                         ", x = " + model.xPosition +
                         ", y = " + model.yPosition);
} // end paint method
public void update(Observable obs, Object arg) {
    repaint();
}

} // end class
View (repeated)

<table>
<thead>
<tr>
<th>View</th>
<th>Get necessary info from Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paint the ball</td>
<td></td>
</tr>
</tbody>
</table>
The Ball Applet: Controller

- The Controller tells the Model what to do
- The Controller tells the View when it needs to refresh the display
- The Controller doesn’t need to know the inner workings of the Model
- The Controller doesn’t need to know the inner workings of the View
Controller

Create Model
Create View
Give View access to
Model
Tell Model to advance
Tell View to repaint

Model
View
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Controller extends JApplet {
    JPanel buttonPanel = new JPanel();
    JButton stepButton = new JButton("Step");

    Model model = new Model();
    View view = new View();

    // more...
public void init() {

    // Lay out components
    setLayout(new BorderLayout());
    buttonPanel.add(stepButton);
    this.add(BorderLayout.SOUTH, buttonPanel);
    this.add(BorderLayout.CENTER, view);

    // more...

Controller III

// Attach actions to components
stepButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        model.makeOneStep();
    }
});

// more...
Controller IV

// Tell the View about myself (Controller) and
// about the Model
model.addObserver(view);
view.controller = this;

} // end init method

// more...
public void start() {
    model.xLimit =
        view.getSize().width - model.BALL_SIZE;
    model.yLimit =
        view.getSize().height - model.BALL_SIZE;
    repaint();
} // end of start method

} // end of Controller class
Controller (repeated)

<table>
<thead>
<tr>
<th>Controller</th>
<th>Model</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create View</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Give View access to Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tell Model to advance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tell View to repaint (via Observer)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Controller (repeated)
Key points

- A Model does the “business logic”
  - It should be *I/O free*
  - Communication with the Model is via methods
  - This approach gives maximum flexibility in how the model is used

- The Controller organizes the program and provides input (control) to the Model

- The View displays what is going on in the model
  - It should never display what *should* be going on in the model
  - For example, if you ask to save a file, the View shouldn’t itself tell you that the file has been saved—it should tell you what the model reports

- Especially in small programs, the Controller and View are often combined
The End