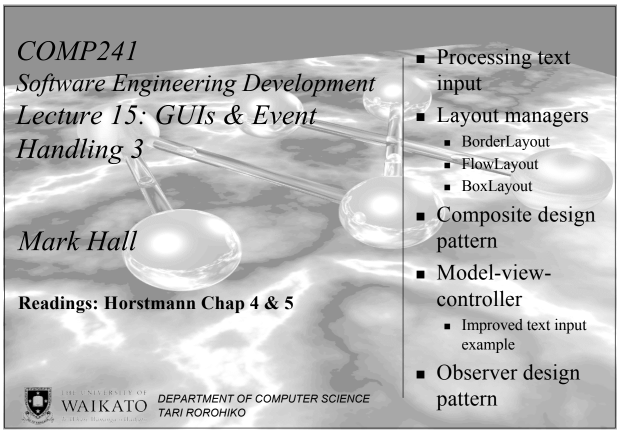


COMP241
Software Engineering Development
Lecture 15: GUIs & Event Handling 3

Mark Hall

Readings: Horstmann Chap 4 & 5

DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

- Processing text input
- Layout managers
  - BorderLayout
  - FlowLayout
  - BoxLayout
- Composite design pattern
- Model-view-controller
  - Improved text input example
- Observer design pattern

Processing Text Input

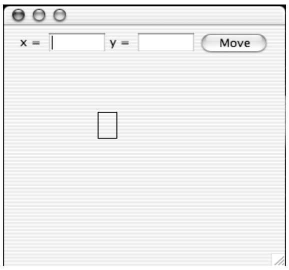
- Most graphical programs collect text input through *text fields*
- The Java Swing GUI libraries have a `JTextField` class for text input
  - When you construct a text field, you supply the width (approx number of characters)

    `JTextField mXField = new JTextField(5);`
  - You can type additional characters, but then part of the content of the field becomes invisible
- You will want to *label* each text field
  - Use a `JLabel`:

    `JLabel mXLabel = new JLabel(“x = ” );`
- Finally, you want to give the user an opportunity to enter information in all text fields before processing it
  - Need a button that the user can press to indicate that the input is ready for processing

Processing Text Input

- `TextInputExample`
  - Similar to `MouseExample2`, but has text fields to allow the user to set the *x* and *y* coordinates of the box



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

public class TextInputExample extends JPanel {
  // x, y, width, height
  private Rectangle mBox =
    new Rectangle(100, 100, 20, 30);

  // text fields for the x and y coordinates
  private JTextField mXField = new JTextField(5);
  private JTextField mYField = new JTextField(5);

  // a button to allow the user to update the
  // rectangle location
  private JButton mMoveButton =
    new JButton("Move");

  . . . // continued on next slide
```

```
. . . // continued from previous slide

public TextInputExample() {
  super();

  // install an ActionListener to move the rectangle
  mMoveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // reset the coordinates of mBox
      int x = Integer.parseInt(mXField.getText());
      int y = Integer.parseInt(mYField.getText());
      mBox.setLocation(x, y);
      repaint(); // ask the JPanel to refresh itself
    }
  });

  // set up the panel
  add(new JLabel("x = ")); add(mXField);
  add(new JLabel("y = ")); add(mYField);
  add(mMoveButton);
}

... // paintComponent and main method ommitted
```

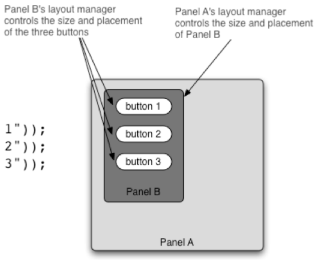
Swing Components

- Components can be nested
  - In Swing, virtually *all* components are capable of holding other components
  - Most of the time, you’ll add *user interactive* components (e.g. buttons, lists etc.) into *background* components (e.g. frames and panels)
    - With the exception of **`JFrame`**, though, the distinction between *interactive* and *background* components is artificial
  - Just about all Swing widgets extend from `javax.swing.JComponent`

Layout Managers

- Control the size and placement of components

```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("button 1"));
panelB.add(new JButton("button 2"));
panelB.add(new JButton("button 3"));
panelA.add(panelB);
```


Panel B's layout manager controls the size and placement of the three buttons
Panel A's layout manager controls the size and placement of Panel B
button 1
button 2
button 3
Panel B
Panel A

How does the layout manager decide?

- Layout scenario:
  1. Make a panel and add three buttons to it
  2. The panel's layout manager asks each button how big the button prefers to be
  3. The panel's layout manager uses its layout policies to decide whether it should respect all, part, or none of the button's preferences
  4. Add the panel to a frame
  5. The frame's layout manager asks the panel how big the panel prefers to be
  6. The frame's layout manager uses its layout policies to decide whether it should respect all, part, or none of the panel's preferences
- **Different layout managers have different polices**
  - **Using layout managers is another example of the *Strategy design pattern***

Three layout managers: border, flow and box

- `BorderLayout`
  - A `BorderLayout` manager divides a background component into five regions
  - You can only add one component per region
  - Components layed out by this manager usually don't get to have their preferred size
  - `BorderLayout` is the default layout manager for a fame

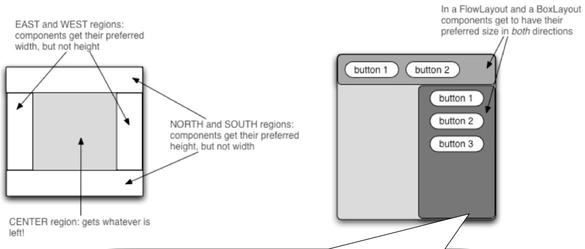
Three layout managers: border, flow and box

- `FlowLayout`
  - Acts kind of like a word processor, except with components rather than words
  - Each component is the size it wants to be and are laid out left to right in the order that they are added
  - "Word wrap" is turned on, so when a component won't fit horizontally, it drops to the next "line" in the layout
  - `FlowLayout` is the default layout manager for a panel

Three layout managers: border, flow and box

- `BoxLayout`
  - A `BoxLayout` manager is like `FlowLayout` in that each component gets to have its own size, and the components are placed in the order that they are added
  - `BoxLayout` can stack the components *vertically* or horizontally
  - Instead of having automatic "component wrapping" you can **force** the components to start a new line

Layout Manager Policies


EAST and WEST regions: components get their preferred width, but not height
NORTH and SOUTH regions: components get their preferred height, but not width
CENTER region: gets whatever is left!
In a FlowLayout and a BoxLayout components get to have their preferred size in *both* directions
button 1
button 2
button 1
button 2
button 3

```
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
```

Composite Design Pattern

- The creation of GUI layouts in Java is a good example of the *Composite* design pattern
- The intent of the Composite design pattern is *to allow the creation of complex objects using simple parts; individual objects and compositions of objects can be treated uniformly*
  - That is, a complex object (composed of many simple parts) can itself be treated as a simple object
- The key to the Composite pattern is an abstract class that represents both simple objects and their containers
  - Swing objects are **both** `Containers` and `JComponents`
  - `JComponents` can be nested arbitrarily (with the aid of layout managers)

Improving TextInputExample

- One problem with `TextInputExample` from before is that we added text fields and a button to the same area that the rectangle gets drawn on
  - Setting *y* to zero results in the rectangle getting drawn over top of the text fields and button!
- In this case the controls should not be part of the drawing area
  - The responsibility of the `JPanel` should just be to draw the rectangle
  - If we separate the controls from the view then we can easily change the controls without having to modify the view code

Model-View-Controller

- Design pattern that advocates separating data (model) from user interface (controller/view) concerns
  - Changes to the UI do not affect data handling
  - Data can be reorganized without changing the UI
  - Decouple data access and application logic from data presentation and user interaction

Model-View-Controller

- Model — holds the *information* in some data structure
- View — renders the information in some way
- Controller — each view has a controller that processes user interaction
- Example interaction:
  - Controller tells model to change/update data
  - Model notifies all views of a change in the model
  - All views repaint themselves
  - During painting, each view asks the model for the current data values
- Improved `TextInputExample`
  - In the spirit of model-view-controller—model & view collapsed into one

```java
public class RectanglePanel extends JPanel {
  private static final int PANEL_WIDTH = 300;
  private static final int PANEL_HEIGHT = 300;

  // x, y, width, height
  private Rectangle mBox = new Rectangle(100, 100, 20, 30);

  public RectanglePanel() {
    setPreferredSize(new Dimension(PANEL_WIDTH, PANEL_HEIGHT));
  }

  // allow users to change the coordinates of the rectangle
  public void setCoordinates(Point coords) {
    mBox.setLocation(coords.x, coords.y);
    repaint();
  }

  public void paintComponent(Graphics g) {
    // first let the superclass erase the old contents
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.draw(mBox); // now draw our box
  }
}
```

```java
public class RectangleController extends JPanel {
  private JTextField mXField = new JTextField(5);
  private JTextField mYField = new JTextField(5);
  private JButton mMoveButton = new JButton("Move");
  private RectanglePanel mView; // ref. to the model/view

  public RectangleController(RectanglePanel view) {
    mView = view;

    mMoveButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // reset the coordinates of mView's rectangle
        int x = Integer.parseInt(mXField.getText());
        int y = Integer.parseInt(mYField.getText());
        mView.setCoordinates(new Point(x, y));
      }
    });

    // set up the panel
    add(new JLabel("x = ")); add(mXField);
    add(new JLabel("y = ")); add(mYField);
    add(mMoveButton);
  }
}
```

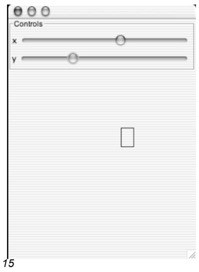
```
import java.awt.event.*;
import java.awt.*;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.BorderFactory;

public class RectangleApplication {
  public static void main(String [] args) {
    JPanel holderPanel = new JPanel();
    holderPanel.setLayout(new BorderLayout());
    RectanglePanel view = new RectanglePanel();
    RectangleController control =
      new RectangleController(view);
    control.setBorder(BorderFactory.
             createTitledBorder("Controls"));
    holderPanel.add(control, BorderLayout.NORTH);
    holderPanel.add(view, BorderLayout.CENTER);
    JFrame myFrame = new JFrame();
    myFrame.setContentPane(holderPanel);
    myFrame.pack();
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    myFrame.setVisible(true);
  }
}
```

Observer Design Pattern

- In model-view-controller, the *views* are observers of the model
  - That is, they are interested in knowing about changes to the model
    - Model notifies all views of changes
  - This is basically the event/listener mechanism we have seen with GUI events

RectangleApplication2

- Replace the `RectangleApplication`'s `RectangleController` with one that uses `JSliders`



```
public class RectangleController2 extends JPanel {
  private JSlider mXSlide;
  private JSlider mYSlide;
  private RectanglePanel mView;
  public RectangleController2(RectanglePanel view) {
    mView = view;
    Dimension viewD = mView.getPreferredSize();
    mXSlide = new JSlider(0, (int)viewD.getWidth());
    mYSlide = new JSlider(0, (int)viewD.getHeight());

    mXSlide.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        // reset the x coordinate of the view's rectangle
        mView.setCoordinates(new Point(mXSlide.getValue(),
                      mYSlide.getValue()));
      }});
    mYSlide.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        // reset the y coordinate of the view's rectangle
        mView.setCoordinates(new Point(mXSlide.getValue(),
                      mYSlide.getValue()));
     }});
    . . . // continued on next slide
```

```
    . . . // continued from previous slide

    // set up the panel
    JPanel xP = new JPanel();
    xP.setLayout(new BorderLayout());
    xP.add(new JLabel("x"), BorderLayout.WEST);
    xP.add(mXSlide, BorderLayout.CENTER);
    JPanel yP = new JPanel();
    yP.setLayout(new BorderLayout());
    yP.add(new JLabel("y"), BorderLayout.WEST);
    yP.add(mYSlide, BorderLayout.CENTER);

    setLayout(new GridLayout(2,1));
    add(xP); add(yP);
  }
}
```

```
public class RectangleApplication2 {
  public static void main(String [] args) {
    JPanel holderPanel = new JPanel();
    holderPanel.setLayout(new BorderLayout());
    RectanglePanel view = new RectanglePanel();

    // use the new controller panel
    RectangleController2 control =
      new RectangleController2(view);

    control.setBorder(BorderFactory.
             createTitledBorder("Controls"));
    holderPanel.add(control, BorderLayout.NORTH);
    holderPanel.add(view, BorderLayout.CENTER);
    JFrame myFrame = new JFrame();
    myFrame.setContentPane(holderPanel);
    myFrame.pack();
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    myFrame.setVisible(true);
  }
}
```