SWL: Tool Tutorial

Swing

Ons Seddiki

Software Technologies Research Group
University of Bamberg

What is Swing?

- Java's platform independent GUI Framework
- Draws GUI elements by itself, not through the OS
  - a lightweight UI system
- Different pluggable look and feel customizations possible
- A Swing GUI runs in its own thread

Working with Swing

Using Swing:

- located in package javax.swing.*
- Best learnt through experimentation
- GUI builders are seldom the best solution
  - Fast at first, but slower in the long term

The Oracle Docs are an excellent resource for learning:

- `http://docs.oracle.com/javase/tutorial/uiswing/`
- Contains many runnable examples
- Directly executable through Java Web start technology
  - Download the .jnlp file
  - Run it with Java Web Start launcher
  - For windows: `C:\Java\jre7\bin\javaws.exe`
  - For linux: IcedTea Java Web Start or similar
- Also contains many code samples

Swing Elements

Hierarchy

- Every on-screen GUI component must be part of a containment hierarchy.
  - In other words, a tree of Swing elements
- Each component may only be contained once

JFrame

- Top-level container (essentially a window)

JPanel (Content Pane)

- A container providing layout options within a JFrame

JComponent

- widgets appearing on screen, e.g. JLabel, JButton

Component Hierarchy in Code

Note how the hierarchy of JFrame > JPanel > JLabel is built through calls to the add() methods:

```
1  public class SwingTest1 extends JFrame {
2
3      public SwingTest1() {
4
5          this.setSize(500,500);
6          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7
8          JPanel thePanel = new JPanel();
9          JLabel label1 = new JLabel("Hello World");
10
11         thePanel.add(label1);
12         this.add(thePanel);
13
14         this.setVisible(true);
15     }
16
17      public static void main(String[] args) {
18         new SwingTest1();
19     }
20 }
```

Modifying/Reading a Components State

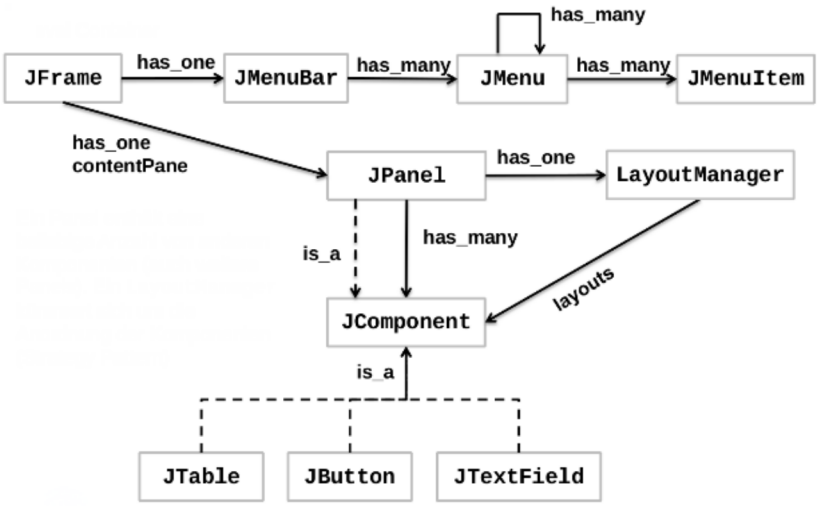
Components may be modified and queried through a number of methods after their creation (lines 6-8):

```
1  JPanel thePanel = new JPanel();
2
3  JLabel label1 = new JLabel("Hello World");
4  thePanel.add(label1);
5
6  label1.setText("Hello Aliens");
7  label1.setToolTipText("Here's a tooltip!");
8  System.out.println(label1.getText());
9
10 JButton button2 = new JButton("Say Hello");
11 thePanel.add(button2);
12
13 this.add(thePanel);
14 this.setVisible(true);
```

Note the addition of a JButton component to the GUI

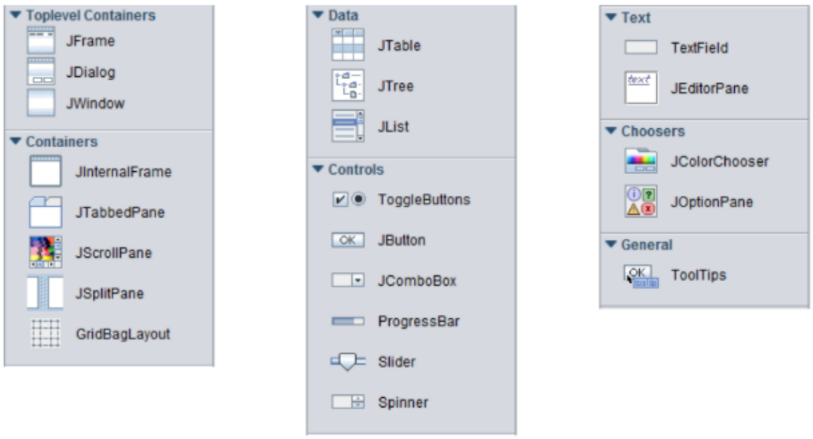
- looks like a button, but currently has no associated functionality

Hierarchy diagram



From the module DSG-AJP-B

Swing Components



From SwingSet3 – https://java.net/projects/swingset3

Specifying layouts and accessing the content pane

The content pane of a JFrame may be set and retrieved directly

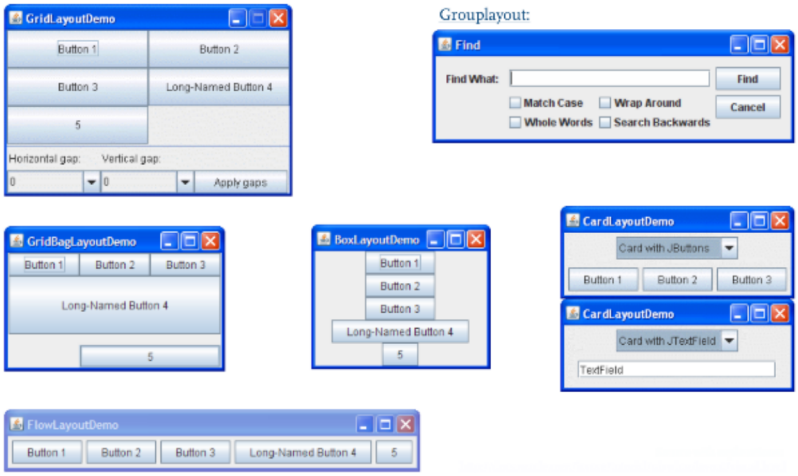
- setContentPane() has the same functionality as add() in the previous examples

Here we are retrieving the content pane to apply a layout to it

- This layout allows component placement relative to panel edges
- Many other layouts available (grid layout is especially useful)

```
1  JPanel contentPane = new JPanel();
2  this.setContentPane(contentPane);
3
4  BorderLayout layout = new BorderLayout();
5  this.getContentPane().setLayout(layout);
6
7  JTextArea textArea = new JTextArea("I'm at the top!");
8  JTextField textField = new JTextField("I'm at the bottom!");
9
10 this.getContentPane().add(textArea, BorderLayout.NORTH);
11 this.getContentPane().add(textField, BorderLayout.SOUTH);
```

Swing Layouts



From http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

Scrolling and Tables

To allow the display of a JTable larger than the window size, place it inside a JScrollPane:

```
String[] columnNames = {
        "First Name",
        "Last Name",
        "Age",
        "Vegetarian"};

Object[][] data = {
        {"Kathy", "Smith", new Integer(5), new Boolean(false)},
        {"John", "Doe", new Integer(3), new Boolean(true)},
        {"Sue", "Black", new Integer(2), new Boolean(false)},
        {"Jane", "White", new Integer(20), new Boolean(true)},
        {"Joe", "Brown", new Integer(10), new Boolean(false)}
    };

final JTable table = new JTable(data, columnNames);

JScrollPane scrollPane = new JScrollPane(table);
```

Adapted from http://docs.oracle.com/javase/tutorial/uiswing/components/table.html

Interactive GUIs

Behavior can be added to the GUI through listeners.

In general, a listener's job is to respond to the occurrence of an event (usually by executing some code).

In Swing, a listener object is attached to a component such that when an action is performed on that component, a piece of code is executed.

- Examples of actions:
  - button1 clicked
  - component loses focus
  - window moved
- Examples of listener responses:
  - display a value in textField1
  - process some user input
  - close the program

Attaching a listener

There is a listener for each type of interaction the user may perform:

- Low-level actions: MouseListener, KeyListener
- Logical actions: ActionListener (focused on here)

The listener object must implement the ActionListener interface

- Essentially, this means providing the method:
  public void actionPerformed(ActionEvent e)

An instance of the listener must then be attached to the component via addActionListener():

```
1 button1 = new JButton("Click me!");
2 ListenForButton1 lForButton1 = new ListenForButton1();
3 button1.addActionListener(lForButton1);
```

Attaching a listener

On the previous slide the listener was declared as a named class

- the source of the action should be tested to prevent misuse

```
1 private class ListenForButton1 implements ActionListener {
2
3     @Override
4     public void actionPerformed(ActionEvent e) {
5         if (e.getSource() == button1) {
6             System.out.println("Button 1 clicked\n");
7         }
8     }
9 }
```

It is also possible to write an in-line anonymous classes:

```
1 button1.addActionListener(new ActionListener() {
2     @Override
3     public void actionPerformed(ActionEvent e) {
4         System.out.println("Button 1 clicked\n");
5     }
6 });
```

Structuring your GUI

It is recommended to structure your GUI and its connection to your application logic using the Model-View-Controller architecture.

The view is the Swing component of your application.

- Contains the components and component listeners that implement the visual functionality of your GUI.

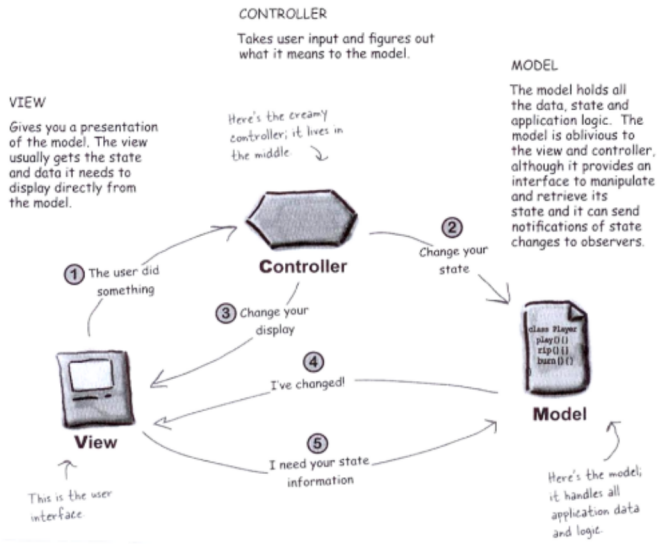
The model is where the core of your application resides

- The view requests data to populate the display
- The controller manipulates the model

The controller provides the glue between the view and the model.

- It is responsible for manipulating the model in response to user actions performed on the view.

The Model-View-Controller (MVC) architecture



E. Freeman and E. Freeman. Head First Design Patterns, O'Reilly, 2004.

MVC component interactions in detail

View ! Controller: User actions performed in the view are passed to the controller to decide how to proceed.

Controller ! View: In some cases, updates to the view may be directly requested by the controller.

Controller ! Model: User actions requiring the model are passed through (possibly with some preprocessing occurring in the controller).

Model ! View: The view registers with the model to listen for state changes relevant to itself. The model then issues notifications of such changes to the view.

View ! Model: The view requests data relevant for display from the model.

MVC in Swing/Java

Swing actually implements a mini-MVC framework of its own: JComponents comprise the view, listeners correspond to the controller and the model maintains the state of JComponents

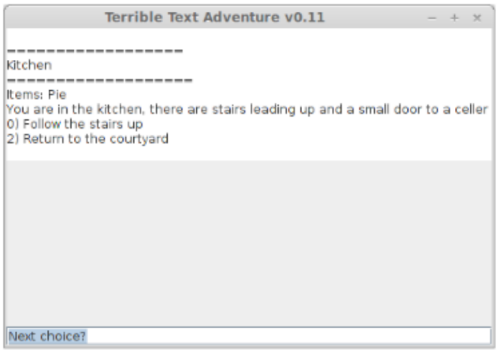
- However, we will not further focus on these details here

Implementing MVC in Java

- The model, view and controller elements should each have a corresponding package in your Java application so they can be further internally structured using classes
- The view listens for changes in the model via the observer pattern
  - The view implements the Observer interface
  - The model implements the Obvserable interface

Running Example

We shall now see how to construct a GUI for our Game example and structure it using MVC.



The user enters their choice of which room to enter next in a JFormattedTextField and the result of their choice will be displayed in a JTextArea.

A small refactoring of our Game application

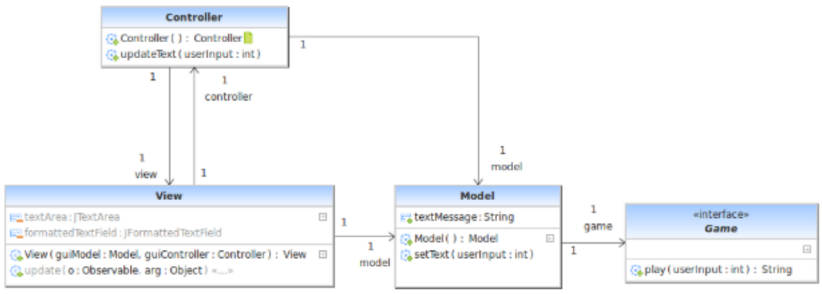
Employ an interface to expose only the aspects of your application necessary for the GUI:

```
1 public interface Game {
2
3     String getInitialText();
4
5     String play(int userInput);
6
7 }
```

and then have your application implement this interface:

```
1 public class GameImpl implements Game {
2
3     @Override
4     public String getInitialText() {
5         ...
6
7     @Override
8     public String play(int userInput) {
9         ...
10 }
```

Running example: communication in the GUI



Note:

- The two JComponents in View that comprise the GUI.
- Model stores the current state of the GUI (the game text) in the member variable textMessage. Hence, the GUI could be regenerated at any time without consulting the model.

After the MVC classes are constructed, View registers as an observer of the Model.

- In the future, changes to the model (updates to textMessage) will result in the update() method of View being called.

Implementation Details: Controller

Controller() constructs View and Model, links the MVC elements together and initializes the model.

```
1  public class Controller {
2
3      private final Model model;
4      private final View view;
5
6      public Controller() {
7          // initialize GUI
8          this.model = new Model();
9          this.view = new View(this.model, this);
10         // initial run of game logic
11         this.model.setInitialText();
12         // make GUI finally visible
13         this.view.setVisible(true);
14     }
15
16     public void updateText(int userInput) {
17         this.model.setText(userInput);
18     }
19
20 }
```

Implementation Details: Model

Model constructs and holds the reference for the application logic (Game)

- Note the convention for notifying observers of changes to the model

```
1  public class Model extends Observable {
2
3      private final Game game;
4      private String textMessage;
5
6      public Model() { this.game = new GameImpl(); }
7
8      public String getTextMessage() { return this.textMessage; }
9
10     public void setInitialText() {...}
11             //body similar to setText()
12
13     public void setText(int userInput) {
14         this.textMessage = this.game.play(userInput);
15
16         this.setChanged();
17         if (this.hasChanged()) {
18             this.notifyObservers();
19         }
20     }
21 }
```

Implementation Details: View (1)

View adds itself as an observer of the model via addObserver

```
1  public class View extends JFrame implements Observer {
2
3      // MVC
4      private final Model model;
5      private final Controller controller;
6      // container
7      private JPanel contentPane;
8      private BorderLayout layout;
9      // widgets
10     private JTextArea textArea;
11     private NumberFormatter numberFormatter;
12     private JFormattedTextField formattedTextField;
13
14     public View(Model guiModel, Controller guiController) {
15         // link MVC parts
16         this.model = guiModel;
17         this.controller = guiController;
18         this.model.addObserver(this);
19
20         initializeView();
21         createAndLinkViewContents();
22     }
```

Implementation Details: View (2)

The method actionPerformed() is called when the user enters a number in the text box

- this is then passed to the controller via updateText()

```
1  private void createAndLinkViewContents() {
2
3      ...
4
5      this.fTextField = new JFormattedTextField(...);
6
7      this.fTextField.addActionListener(new ActionListener() {
8          @Override
9          public void actionPerformed(ActionEvent arg0) {
10             if (fTextField.getText() instanceof String) {
11                 controller.updateText(Integer
12                         .parseInt(fTextField.getText()));
13             }
14             fTextField.setText("Next choice?");
15             fTextField.selectAll();
16         }
17     });
18
19   getContentPane().add(formattedTextField, BorderLayout.SOUTH);
20 }
```

Implementation Details: View (3)

The update() method will be invoked whenever Model calls notifyObservers()

```
1      @Override
2      public void update(Observable o, Object arg) {
3          this.textArea.setText(this.model.getTextMessage());
4          this.repaint();
5      }
6  }
```

Exercise

Implement a basic calculator application with...

- Two text fields for user entry
- A label for displaying the calculation result
- A button for addition, subtraction, multiplication and division
  - When the user clicks an operation button, the result should be directly displayed in the label.

Think about...

- the application behavior if the user enters a non-integer
- how you would structure your application if you were using MVC.