Graphical User Interfaces (GUI) - I

1. Overview

The learning objectives of this laboratory session are:

- To get a clear understanding of the way to build a Graphical User Interface.
- To acquire knowledge concerning the use of important classes and interfaces used in application which have a GUI.
- To acquire hands-on programming experience in managing the events generated by the controls used in a GUI.

The mouse is automatically handled by most components, and thus, in general, you do not have to care about it. For instance, if a user clicks a Swing button (`JButton`), you will receive an `ActionEvent`, but you do not need to know (and shouldn't care) if that was caused by a click on the button or by using a shortcut key.

Graphics. If you draw your own graphics (e.g., in a `JPanel`) and you need to know whether the user clicked, then you must know about the events generated by a mouse. You can easily add a mouse listener to a `JPanel`. More about this in another laboratory guide.

2. Containers and Components

Java packages containing graphics related stuff are:
`Java.awt`
`Javax.swing`

AWT is an interface for the native GUI code which an operating system (OS) provides, a wrapper for graphics objects of the OS. This makes it platform dependent. For the components provided by **Swing**, the Java Virtual Machine (JVM) takes care of their look, and thus offers OS independence. More, Swing comes with an extended range of components and facilities. There are corresponding classes in Swing for the AWT classes (found in the `java.awt` package). They all have a "J" prefix (e.g.: `JButton, JTextField, JLabel, JPanel, JFrame,` or `JApplet)`.

A part of the Swing class hierarchy is shown in Figure 1:

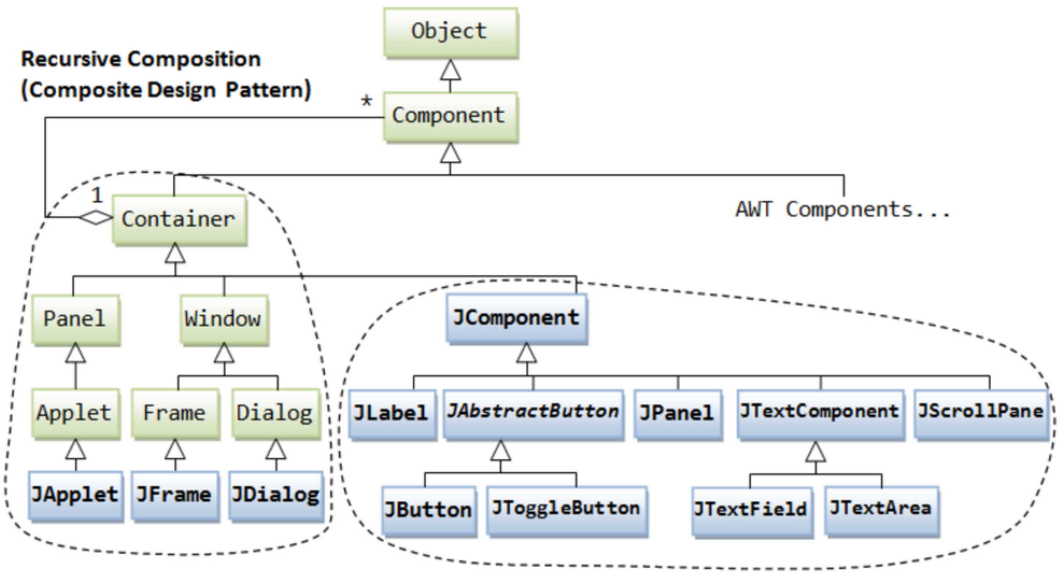

Figure 1. A class hierarchy for Swing (green: AWT).

Graphics elements have two types:

1. ***Components***: objects with a graphic representation which ca be displayed on the screen, and which can interact with the user. Examples: **`JButtons, JLabels, JTextField`** etc.)
2. ***Containers***: components that can contain other components. In Swing, containers such as **`JFrame, JPanel`**, are used to contain components. Components can be arranged inside a container by specifying a layout (`FlowLayout, GridLayout, BoxLayout,` etc.). A container can contain other containers. To arrange the layout you can use contained containers.

Figure 2 shows an example of a simple GUI.

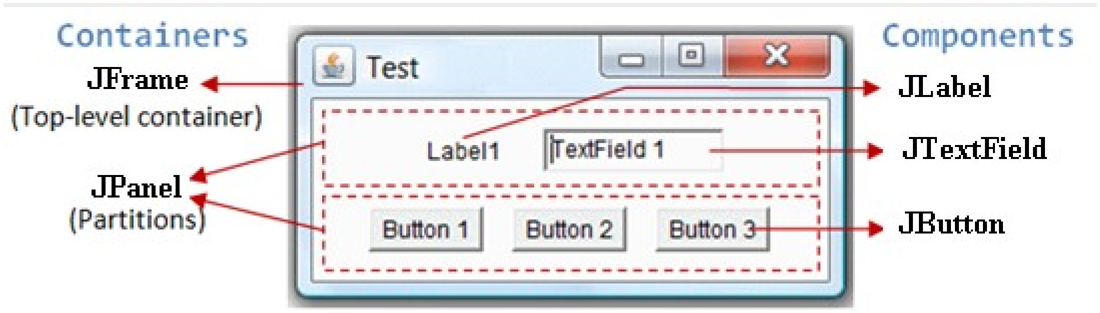

Figure 2. A simple GUI.

2.1 Arranging Components Inside Containers

To customize component layout in containers you must specify that. There are many layout managers. Some of the most common are presented below:

- **`FlowLayout`** (default for `JPanel`) – arranges elements line by line, as you would write on a page. (An example is shown in Figure 3.)

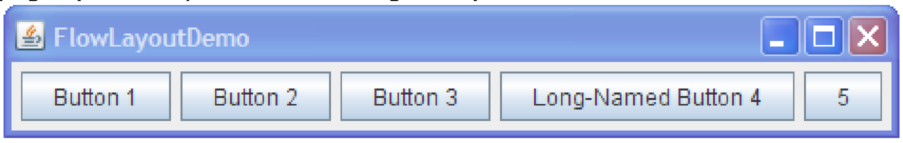

Figure 3. A FlowLayout Example.

The Java code for the example of Figure 3 might be:

```
FlowLayout experimentLayout = new FlowLayout();
...
    compsToExperiment.setLayout(experimentLayout);

    compsToExperiment.add(new JButton("Button 1"));
    compsToExperiment.add(new JButton("Button 2"));
    compsToExperiment.add(new JButton("Button 3"));
    compsToExperiment.add(new JButton("Long-Named Button 4"));
    compsToExperiment.add(new JButton("5"));
```

- **`GridLayout`** – places components in a matrix specified by its numbers of rows and columns. For an example see Figure 4.

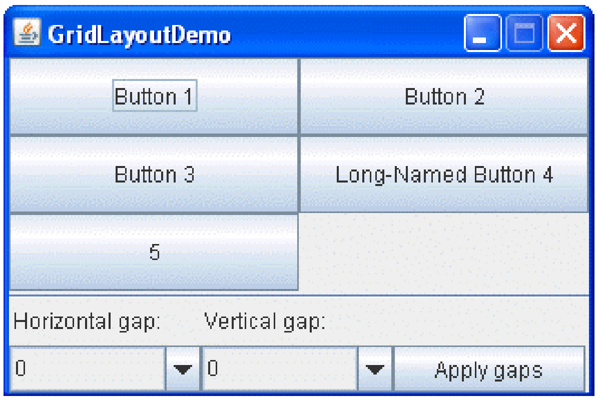

Figura 4. A GridLayout Example.

The Java code for the example of Figure 4 might be:

```
GridLayout experimentLayout = new GridLayout(0,2);
...
        compsToExperiment.setLayout(experimentLayout);

        compsToExperiment.add(new JButton("Button 1"));
        compsToExperiment.add(new JButton("Button 2"));
        compsToExperiment.add(new JButton("Button 3"));
        compsToExperiment.add(new JButton("Long-Named Button 4"));
        compsToExperiment.add(new JButton("5"));
```

- **`BorderLayout`** – places components in up to five areas: east, west, south, north and center; remaining space is placed in the center (see Figure 5)

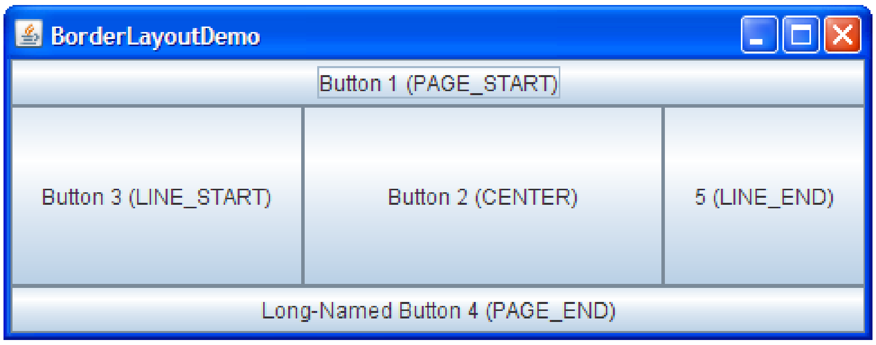

Figura 5. A BorderLayout Example.

The Java code for the example of Figure 5 might be:

```
...//Container pane = aFrame.getContentPane()...
JButton button = new JButton("Button 1 (PAGE_START)");
pane.add(button, BorderLayout.PAGE_START);

//Make the center component big, since that's the
//typical usage of BorderLayout.
button = new JButton("Button 2 (CENTER)");
button.setPreferredSize(new Dimension(200, 100));
pane.add(button, BorderLayout.CENTER);
...
```

- There are other layout managers, such as: **`BoxLayout`, `CardLayout`, `SpringLayout`** etc. (Details about those can be found in [1].)

2.2 Developing a Simple GUI Application

A simple example, consisting only of the layout code, (no code for interaction) is shown below.

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

class MyPanel extends JPanel {

      public static void main() {
             JFrame frame = new JFrame ("Simple Frame");
             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
             frame.setSize(300, 120);

            JPanel panel1 = new JPanel();
            JPanel panel2 = new JPanel();

            JLabel l = new JLabel ("Label1 ");
            JTextField tf = new JTextField("TextField1");
            panel1.add(l);
            panel1.add(tf);
            panel1.setLayout(new FlowLayout());

            JButton b1 = new JButton("Button 1");
            JButton b2 = new JButton("Button 2");
            JButton b3 = new JButton("Button 3");
            panel2.add(b1);
            panel2.add(b2);
            panel2.add(b3);

            JPanel p = new JPanel();
            p.add(panel1);
            p.add(panel2);
            p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));

            frame.setContentPane(p);
            frame.setVisible(true);
       }
}
```

3. Event Handling

3.1 Events

Events come from the controls as a result of user interaction. When we define a user interface, we need a way to get information from a user. Buttons, menus, sliders, mouse clicks, etc., generate events when a user interacts with them. The event objects from the user interface are passed from an **event source** – such as a button or a mouse click, to an **event listener** – a programmer provided method which processes event objects.
Every input control (such as `JButton, JSlider`, ...) requires an event listener. **If you want a control to do something when a user interacts with it, then you have to provide a listener.**

There are many kinds of events. The usual ones are shown in Table 1.

Event Source	Method to add listener	Listener method
JButton JTextField JMenuItem	addActionListener()	actionPerformed(actionEvent e)
JSlider	addChangeListener()	stateChanged(ChangeEvent e)
JCheckBox	addItemListener()	itemStateChanged()
Key on component	addKeyListener()	keyPressed(), keyReleased(), **keyTyped()**
Mouse on component	addMouseListener()	**mouseClicked()**, mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
Mouse move on component	addMouseMotionListener()	mouseMoved(), mouseDragged(),
JFrame	addWindowListener()	windowClosing(WindowsEvent e), …

Table 1. Event sources, listener add methods and methods invoked in a listener.

To work with events, you need to use some predefined classes. You can use **import clauses** to do that:

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

3.2 Listeners

An event listener is invoked when the user interacts with the interface, which produces an event. Many events come from the user interface, but they may also have other sources such a `Timer`. After the creation of a button, add a listener to it. For instance, use

```
 btn.addActionListener( listener_object);
 // the listener object is of the type ButtonListener defined below
```

Upon click on the button, the `actionPerformed()` method is invoked (method defined in the listener's class). This method receives an `ActionEvent` object as a parameter.

Below is an example of a class which implements a listener:

```
class ButtonListener implements ActionListener{
     public void actionPerformed(ActionEvent e){
         // perform smth. When the button is pressed
         ++count;
         tf.setText(count + "");
     }
 }
```

Listeners may be implemented as anonymous inner classes. Here is an example:

```
btnCount.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
            // perform smth. When the button is pressed
            ++count;
            tf.setText(count + "");
        }
    });
```

4. The Model-View-Controller (MVC) Pattern

Many user interfaces rely on the **Model-View-Controller (MVC)** pattern. The idea behind this pattern is to separate programs into three parts: a **Model**, a **View** (creates what the user sees, interacts with the Model as needed), and **Controller** (responds to user requests, interacts with both Model and View).

The role of the MVC architecture parts (according to [2])**:**

- **Model** – encapsulates the data specific to an application and define the logic and computation that manipulate and process that data. In enterprise software, a model often serves as a software approximation of a real-world process. The model doesn't know anything about views and controllers. When a model changes, typically it will notify its observers that a change has occurred.
- **View –** renders the contents of a model. It specifies exactly how the model data should be presented. If the model data changes, the view must update its presentation as needed. This can be achieved by using a *push model*, in which the view registers itself with the model for change notifications, or a *pull model*, in which the view is responsible for calling the model when it needs to retrieve the most current data
- **Controller** – The controller translates the user's interactions with the view into actions that the model will perform. In a stand-alone GUI client, user interactions could be button clicks, menu selections, etc.

4.1 An application Based on MVC: A Simple Calculator

Our simple calculator follows the MVC concept. Its separates the user interface in a View, which creates the presentation, and interacts with the Model as needed), and a Controller, which responds to user interaction, and communicates with both the View and the Model, as needed. There are variations in the literature, but all ideas follow this basic philosophy. The model is simple, and can be used with simple method calls. For more complex interactions (e.g. asynchronous Model updates), then the use of an Observer pattern (with listeners) may be necessary. Its GUI is shown in Figure 6.

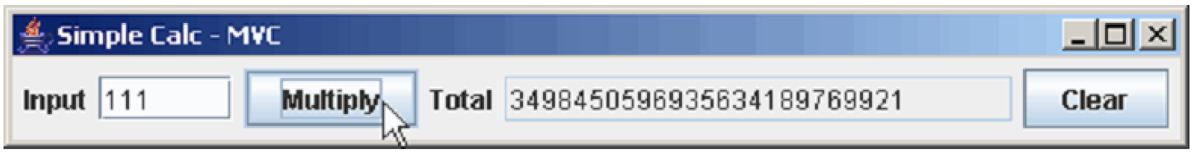

Figure 6. GUI for a simple calculator.

The main Program
The main program initializes the interface and connects all parts.

```
// CalcMVC.java -- Calculator in MVC pattern.
// Fred Swartz -- December 2004

import javax.swing.*;

public class CalcMVC {
    //... Create model, view, and controller.  They are
    //    created once here and passed to the parts that
    //    need them so there is only one copy of each.
    public static void main(String[] args) {

        CalcModel      model      = new CalcModel();
        CalcView       view       = new CalcView(model);
        CalcController controller = new CalcController(model, view);

        view.setVisible(true);
    }
}
```

The View:

```
// CalcView.java - View component
//    Presentation only.  No user actions.
// Fred Swartz -- December 2004

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

class CalcView extends JFrame {
    //... Constants
    private static final String INITIAL_VALUE = "1";

    //... Components
    private JTextField m_userInputTf = new JTextField(5);
    private JTextField m_totalTf     = new JTextField(20);
    private JButton    m_multiplyBtn = new JButton("Multiply");
    private JButton    m_clearBtn    = new JButton("Clear");

    private CalcModel m_model;

    //======================================================= constructor
    /** Constructor */
    CalcView(CalcModel model) {
        //... Set up the logic
        m_model = model;
        m_model.setValue(INITIAL_VALUE);

        //... Initialize components
        m_totalTf.setText(m_model.getValue());
        m_totalTf.setEditable(false);

        //... Layout the components.
        JPanel content = new JPanel();
        content.setLayout(new FlowLayout());
        content.add(new JLabel("Input"));
        content.add(m_userInputTf);
        content.add(m_multiplyBtn);
        content.add(new JLabel("Total"));
        content.add(m_totalTf);
        content.add(m_clearBtn);

        //... finalize layout
        this.setContentPane(content);
        this.pack();

        this.setTitle("Simple Calc - MVC");
        // The window closing event should probably be passed to the
        // Controller in a real program, but this is a short example.
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    void reset() {
        m_totalTf.setText(INITIAL_VALUE);
    }

    String getUserInput() {
        return m_userInputTf.getText();
    }

    void setTotal(String newTotal) {
        m_totalTf.setText(newTotal);
    }

    void showError(String errMessage) {
        JOptionPane.showMessageDialog(this, errMessage);
    }

    void addMultiplyListener(ActionListener mal) {
        m_multiplyBtn.addActionListener(mal);
    }

    void addClearListener(ActionListener cal) {
        m_clearBtn.addActionListener(cal);
    }
}
```

The Controller:

```
// CalcController.java - Controller
//    Handles user interaction with listeners.
//    Calls View and Model as needed.
// Fred Swartz -- December 2004

import java.awt.event.*;

public class CalcController {
    //... The Controller needs to interact with both the Model and View.
    private CalcModel m_model;
    private CalcView  m_view;

    //========================================================
constructor
    /** Constructor */
    CalcController(CalcModel model, CalcView view) {
        m_model = model;
        m_view  = view;

        //... Add listeners to the view.
        view.addMultiplyListener(new MultiplyListener());
        view.addClearListener(new ClearListener());
    }


    //////////////////////////////////////// inner class MultiplyListener
    /** When a mulitplication is requested.
     *  1. Get the user input number from the View.
     *  2. Call the model to mulitply by this number.
     *  3. Get the result from the Model.
     *  4. Tell the View to display the result.
     * If there was an error, tell the View to display it.
     */
    class MultiplyListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String userInput = "";
            try {
                userInput = m_view.getUserInput();
                m_model.multiplyBy(userInput);
                m_view.setTotal(m_model.getValue());

            } catch (NumberFormatException nfex) {
                m_view.showError("Bad input: '" + userInput + "'");
            }
        }
    }//end inner class MultiplyListener


    //////////////////////////////////////////// inner class ClearListener
    /**  1. Reset model.
     *   2. Reset View.
     */
    class ClearListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            m_model.reset();
            m_view.reset();
        }
    }// end inner class ClearListener
}
```

The Model:

```
// CalcModel.java
// Fred Swartz - December 2004
// Model
//     This model is completely independent of the user interface.
//     It could as easily be used by a command line or web interface.

import java.math.BigInteger;

public class CalcModel {
    //... Constants
    private static final String INITIAL_VALUE = "0";

    //... Member variable defining state of calculator.
    private BigInteger m_total;  // The total current value state.

    //============================================================ constructor
    /** Constructor */
    CalcModel() {
        reset();
    }

    //================================================================== reset
    /** Reset to initial value. */
    public void reset() {
        m_total = new BigInteger(INITIAL_VALUE);
    }

    //============================================================= multiplyBy
    /** Multiply current total by a number.
    *@param operand Number (as string) to multiply total by.
    */
    public void multiplyBy(String operand) {
        m_total = m_total.multiply(new BigInteger(operand));
    }

    //================================================================== setValue
    /** Set the total value.
    *@param value New value that should be used for the calculator total.
    */
    public void setValue(String value) {
        m_total = new BigInteger(value);
    }

    //================================================================== getValue
    /** Return current calculator total. */
    public String getValue() {
        return m_total.toString();
    }
}
```

5. Lab Tasks

5.1 Study and understand the examples provided here (more in [3]).

5.2 In the simple application add listeners to three buttons as follows:
- Button1 should count the number of button presses on it and will update the label.
- Button2 should read a text input in the `JTextField` and should also display it on the label. To get the text in the input field use the `getText()` method (which returns a `String`).
- When Button3 is pressed, a random color should be generated and applied to the panel.

5.3 Using the simple calculator as a base, add a new button for the "add" operation. The new GUI should look like what is illustrated in Figure 7.

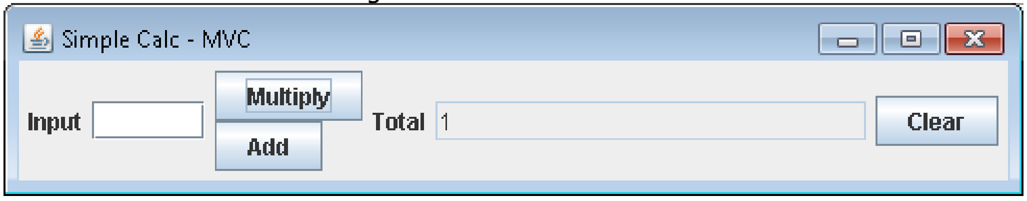

Figure 6. The new simple calculator interface.

5.4 Using the MVC architecture, develop a program which simulates a currency converter. As a GUI example use the Official National Bank of Romania site: http://www.cursbnr.ro/convertor-valutar (cf. figure 8)

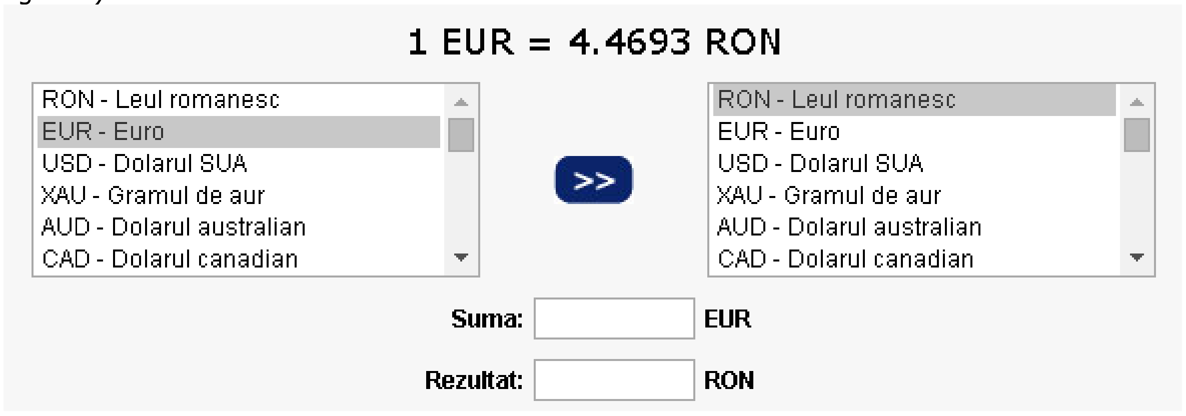

Figura 7. Interfața pentyru convertorul valutar.

Hints: use `JBomboBox`si [4] to select the currency. Implement just three options: RON, EUR, USD.

6. References

[1] Oracle, „Java Tutorials. A Visual Guide to Layout Managers,” 2015. [Interactiv]. Available: https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html. [Accesat 27 November 2016].

[2] R. Eckstein, „Java SE Application Design With MVC,” March 2007. [Interactiv]. Available: http://www.oracle.com/technetwork/articles/javase/index-142890.html. [Accesat 27 November 2016].

[3] C. H. Chuan, „Yet another insignificant programming notes,” January 2016. [Interactiv]. Available: http://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html. [Accesat 27 November 2016].

[4] Oracle, „Java Tutorials. How to Use Combo Boxes,” 2015. [Interactiv]. Available: https://docs.oracle.com/javase/tutorial/uiswing/components/combobox.html. [Accesat 27 November 2016].