Seventh College on Microprocessor-Based Real-Time Systems in Physics

Abdus Salam ICTP, Trieste.

October 28 – November 22, 2002

Lecture Notes

This chaper contains lecture notes of building GUI.

Contents

Chapter 7

Building GUI With Swing

by Ulrich Raich

7.1 Building Graphical User Interfaces with Swing

7.1.1 General comments on Graphical User Interfaces, the MVC concept

Since the appearance of the first MacIntosh computers the use of a mouse for interaction with a program displaying buttons, menus, sliders, textboxes and the like has become common practice on all desktop computers. These **G**raphical **U**ser **I**nterfaces (GUI) have not only given computer access to many computer illiterate people but they also have revolutionized the way application programs are written. While the use of computers becomes more and more easy or "user friendly", providing (designing and implementing) the programs becomes more and more difficult and time consuming.

GUI programming is inherently difficult: conceptually difficult because the user dictates the sequence of operations to be carried out but difficult also because of the size of GUI libraries which may well contain 10,000 different routines or more needed for creation, layout and interaction with user interface element: the so-called widgets.

When creating a program with a GUI the design will usually be broken up into 3 distinct parts. We will take a calculator program for complex numbers as our example (ComplexCalc) and demonstrate the full design and implementation cycle on this example:

The Model

The first part models the problem. We call this part the **Model**. Our *ComplexModel* models the Calculator and is implemented with classical object oriented programming without connection to graphical user interfaces. Here the real problem solving takes place. Imagine the user wants to perform an addition in our complex calculator. He will first enter the real and imaginary parts of the first number and then press **+**. Then the second number will be entered and finally he will press **=**. The model therefore must be capable of

- keeping the current state of the entered number;
- saving the entered number in internal registers when the operator button (+) is pressed
- keeping the operator in order to know which calculation must be executed once = is pressed.

In order to fulfil these requests the ComplexModel needs to have instance variables allowing to save

- the number currently entered;
- the number that had been entered before the operator button was pressed
- the operator

and it needs methods to

- get the currently entered number
- add a digit to the currently entered number
- clear the currently entered number
- set/get the operator
- get the number entered before pressing the operator button
- execute the calculation
- set the decimal point

We will see the implementaion of the ComplexModel in some detail later.

7.1.2 The View

Graphical User Interfaces consist of a hierarchy of widgets. Here we must distinguish two different widget types: the container widget whose function is to contain other widgets which in turn may be container widgets themselves, and primitive widgets which are the ones that can be seen on the screen and interacted with. Starting from a root widget a tree structure is built with intermediate notes being container widgets and the primitive widgets as leafs.

This static layout of widgets (the buttons, menus, labels, and the like) which show the current state of the model, we call the **View** .

Interactive GUI builders help with the task of creating the widget hierarchy. The dynamic behaviour however is entirely left to the programmer. It is up to him to implement what will happen when a user presses a button or selects an item in a menu. GUI builders can save time but they can be used efficiently only once the programmer perfectly understands the underlying concepts (layout managers, widget resources etc.). For this reason the GUI for the complex calculator is created by hand. Once you understand all the mechnisms involved you are encouraged to try rebuilding the view with help of a GUI builder like Borlands JBuilder™; or *SUN One*™.

In our complex calculator this will be the code that creates text widgets showing the current state of real and imaginary parts, buttons that allow to enter numbers (in addition to the text widgets themselves) and buttons that permit starting the operations like addition, multiplication, etc.

7.1.3 The Controller

Last but not least, we must make our GUI responsive (dynamic). This means that pushing buttons or entering text must provoke changes in the model and the view. Pushing the "add button" must add the number currently visible in the text widgets with the previously entered numbers and display the result on the text widgets. The task of dispatching commands for data entry or calculation to the model and commands for display updates to the view is performed by the **Controller**.

The 3 parts that make up a GUI driven program are therefore the **Model**, **View** and **Controller**, and we speak of the **MVC concept**.

7.1.4 Creating the View

Since a single example says more than thousand words we will go through the design and implementation of the complex calculator step by step. We will start with the GUI (the view), then attach a few simple actions, just in order to demonstrate that we can actually launch actions with our interface (a few steps into the direction of implementing the controller), then we create the model and in the end we put the pieces together in order to get a working program.

Before coding, the widget layout should be done on paper. Here we show where we want to get to and then we slowly work towards this goal.

You can distinguish

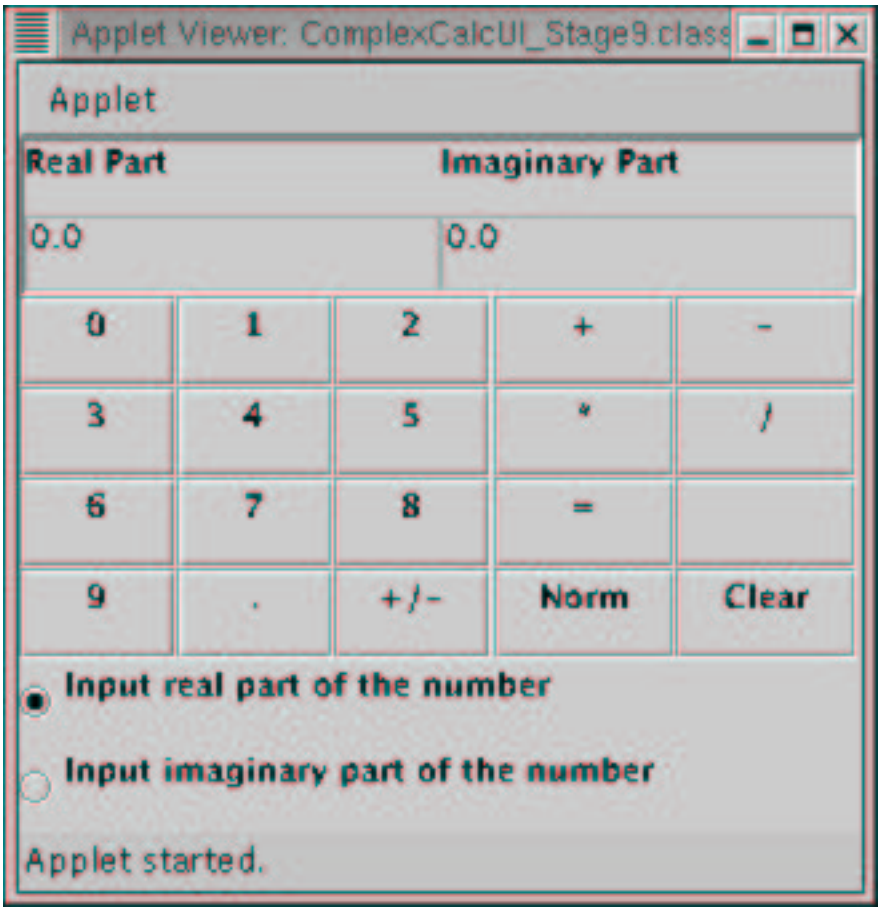
- 2 text widgets and 2 associated labels used for display and data entry for the real - and the imaginary parts of the number.
- Then there is a series of buttons allowing to enter single digits and a radio box with 2 radio buttons, defining into which of the two text widgets the input will go.
- Last but not least, we have the operator buttons "+" "-" ... "=" starting operations on the numbers.

What you don't see are the container widgets allowing to define geometrical relations between these widgets.

7.1.5 The first widget on the screen

In order to get a feeling of what we need to do, we will write a sort of a Swing "hello world" program. **Swing** is the name of the Java class library responsible for the creation of graphical user interfaces. It relies on classes that connect to the native operating system for window creation placement and the like, collected in the **A**bstract **W**indows **T**oolkit (AWT). We create a single text widget and make it appear on the screen.

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



Screenshot of the finished static part of the GUI for the complex number calculator, the View.

Figure 7.1: The Layout of Widgets for the Complex Calculator

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

/**
 * ComplexCalcUI.java
 *
 * Created: Mon Aug 21 21:19:31 2000
 * Stage 1
 * @author Ulrich Raich
 * @version 1.0
 */

public class ComplexCalcUI
{
  public static void main(String arg[])
  {
```

```
    /*
      Create a Panel for real and imaginary part
      text inputs
    */
    JFrame frame =
new JFrame("Beginning of the Complex Calculator");
    JTextField realPartText = new JTextField(30);

    frame.getContentPane().add(realPartText);
    frame.pack();
    frame.setVisible(true);
  }
}// ComplexCalcUI
```

And the result is shown in figure 7.2

Figure 7.2: A Java Application with a single Swing GUI Element

Firstly we need to include a few Swing specific packages which is done by the import statements at the beginning of the code. Then we create a JFrame which is the basic window into which all other GUI elements will be put. Then we create a text input widget of 30 chars in length, put it into the *contentPane* of the base window and make the JFrame (and therefore also the text input) visible on the screen. The application is extremely simple but it already allows you to enter some text.

7.1.6 Java Applets

Now you say: "This is all fine but how do I put this application onto the WEB, after all your lectures have the title: Programming the WEB?"

Well, ehhh. . . oops. . . I cannot! However, what I can do is re-writing the application very slightly, turning the program into an applet. Now I do not have the *main* method any more but there is an *init* method instead. Also this class extends JApplet and it adds the text input into the contentPane of the JApplet instead of creating a JFrame and putting it in there.

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

/**
 * ComplexCalcUI_Stage1.java
 * Created: Mon Aug 21 21:19:31 2000
 * Stage 1
```

```
 * @author Ulrich Raich
 * @version 0.1
 */

public class ComplexCalcUI_Stage1 extends JApplet {
    public ComplexCalcUI_Stage1 () {
    }
/**
 *  Generates a Text widget that will be used for Display
 *  and text entry for the real part of our complex number
 */
    public void init() {
        /*
          Create a Panel for real and imaginary part
          text inputs
        */
        JTextField realPartText;
        realPartText = new JTextField();

        this.getContentPane().add(realPartText);
    }
}// ComplexCalcUI_Stage1
```

However now we have the problem that there is no *main* method any more and

java ComplexCalcUI_Stage1

will result in an error message. We have to integrate the applet into a html page.

```
<html>
<head>
<title> A Calculator for Complex Numbers </title>
<!-- Changed by: Uli Raich, 12-Feb-2000 -->
<h1> A Calculator for Complex Numbers  </h1>
<body bgcolor="#c4c4c4">
<div align="center">
<applet code="ComplexCalcUI_Stage1.class" height=50 width=150>
</applet>
</div>
</body>
</html>
```

Save this html page into a file named *ComplexCalc.stage1.html* and run **appletviewer ComplexCalc.stage1.html** Your applet will appear on the screen. “Wait!” you say, “this is still not what I wanted. I wanted to have a WEB page with everything you showed us in the section called *Introduction* in Chapter 1 and the section called *CGI Programming* in Chapter 2 and my applet appearing on this page and I want to visualise all this in my Netscape browser!”

Unfortunately here things become really clumsy. Most WEB browsers come with a rather outdated **J**ava **VirtualMachine** (JVM) which is unable to run Swing applets. Older java versions were delivered with only the **A**bstract **W**indows **T**oolkit installed but without the Swing classes. You can therefore **not** run Swing applets in these browsers, unless you install a *plugin* implementing a more modern JVM.

Now the problem arises of how to distinguish those 2 JVMs, the one packaged in the standard Netscape distribution and the one added through the plugin. The answer is **new tags** within the html file.

When creating a real Java program and not just a simple ICTP college exercise the code is usually subdivided into modules stored in several files. When compiling, this will create several .class files. In addition you may need image or video or audio files for your applet. In order to improve applet loading time you can package all these files into a single jar (java archive) file which is very similar to tar files. This simple Makefile shows you how the jar file for the first stage of the Complex Calculator has been created:

```
# This makefile creates the jar file for the Complex Calculator
# in a single JAR file.

INSTALLDIR=/var/www/html/ICTP/lecture/

CLASSFILES= \
        ComplexCalcUI_Stage1.class

JARFILE= ../jars/complexCalcStage1.jar

all:    $(JARFILE)

# Create a JAR file with a suitable manifest.

$(JARFILE): $(CLASSFILES)

        echo "Name: ComplexCalcUI.class" >> manifest.tmp
        echo "Java-Bean: False" >> manifest.tmp

        echo "" >> manifest.tmp

        jar cfm $(JARFILE) manifest.tmp *.class

        @/bin/rm manifest.tmp

# Rule for compiling a normal java file
%.class: %.java
        export CLASSPATH; CLASSPATH=. ; \
        javac $<

clean:
```

```
        /bin/rm -f *.class
        /bin/rm -f *~
        /bin/rm -f $(JARFILE)

install:
        cp $(JARFILE) $(INSTALLDIR)/jars
        cp ComplexCalcStage1-Netscape.html $(INSTALLDIR)/HTML/complexCalc
```

The following html file defines that the class ComplexCalcUI_Stage1.class should be loaded from the java archive ../jars/ComplexCalcUI_Stage1.jar and should be executed.

```
<html>
<head>
<title> The beginning of the Complex Calculator GUI <\title>
<!-- Changed by: Uli Raich, 02-Mar-2000 -->
<h1> The beginnings of the Complex Calculator GUI </h1>
<body bgcolor="#c4c4c4">
<div align="center">
<br><br>
<EMBED type=application/x-java-applet
java_docbase=file:///none width=150 height=50
code=ComplexCalcUI_Stage1.class
archive=../jars/ComplexCalcUI_Stage1.jar>
<p>
</div>
</body>
</html>
```

And the result is shown in figure 7.3

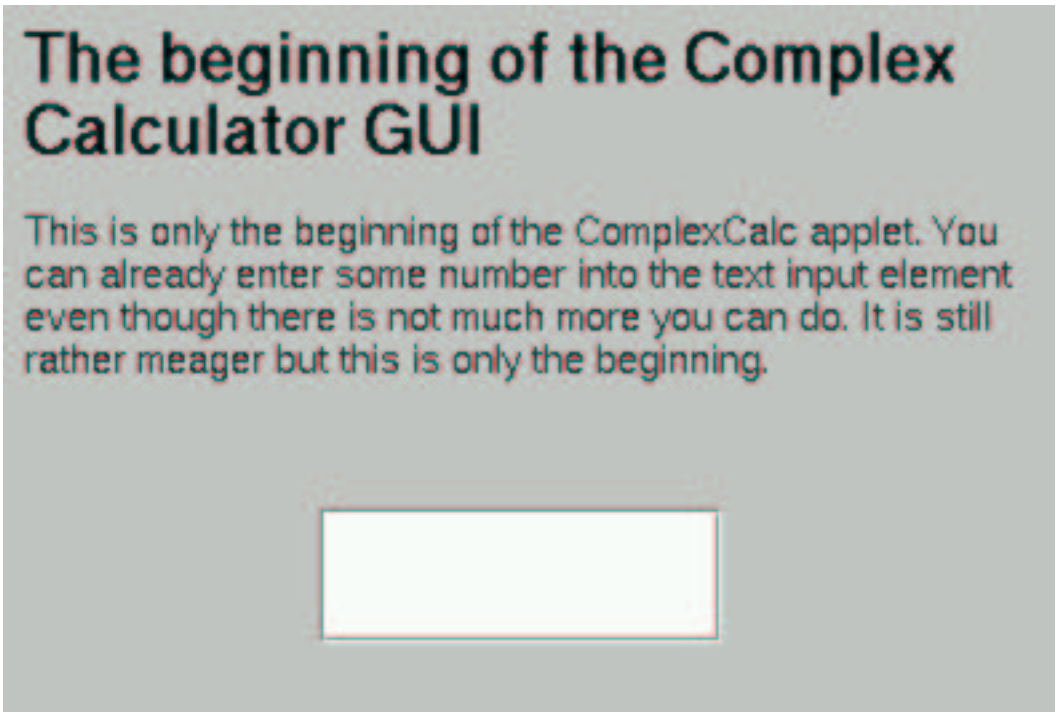


Figure 7.3: The first Java applet within a WEB page as seen by Netscape

Notice that when clicking on the text widget the caret will appear and you will be able to enter text into it.

7.1.7 The Applet life cycle

Applets encounter several important milestones in their life. Firstly they are created when the browser brings up the html page into which they are embedded for the first time. At this moment the **init** method is called.

When you switch to another page the applet is stopped (its **stop** method is called) and restarted (the **start** method is called) when you come back to the page. Finally, when you exit the browser the **destroy** will be given a chance to do some cleanup that might be necessay.

The applet therefore has the following methods which may be overridden or not:

- init
- start
- stop
- destroy

One more thing that is worth mentioning: System.out.println will not print anything onto the applet area; this text goes into the *system console*. When rewriting the *HelloWorld* program from the lectures on basic Java into an applet we will have to do graphics drawing in order to get the text onto the html page.

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

/**
 * ComplexCalcUI.java
 *
 *
 * Created: Mon Aug 21 21:19:31 2000
 * Stage 1
 * @author Ulrich Raich
 * @version 1.0
 */

public class HelloWorld extends JApplet {
    public HelloWorld () {

    }

/**
 *  Generates a Text widget that will be used for Display
 *  and text entry for the real part of our complex number
 */
  public void paint(Graphics g)
```

```
  {
    g.drawString("Hello World !",10,20);
  }

}// HelloWorld
```

Figure 7.4: The Hello World Program re-written as an Applet

Unfortunately explaining the Graphics class which permits drawing in Java, or its more modern Graphics2d or even 3d counterparts, goes largely beyond the scope of this Workshop. We could actually give a 4 weeks course on Java graphics alone. For this reason you are referred to the Java API documentation of the SUN Java tutorials.

7.1.8 Adding more Elements, Layout Management

A typical graphical user interface, except for a *hello world* style program consists of a whole series of GUI elements. In our example we first want to add a label describing the text intput. Of course you have no problem of creating such a label:

```
realPartLabel = new JLabel("Real Part");
```

will do the trick, however the question arises where this new widget will be situated on the screen. To keep things simple for the moment we will create a *vertical box* and add the label and the text to the box where they will appear one on top of each other, as you may expect. The box is then added to the content pane of the applet and we are done.

```
Box realPartBox;
JLabel     realPartLabel;
JTextField realPartText;
/*
  create the widgets
*/
realPartBox = Box.createVerticalBox();
realPartLabel = new JLabel("Real Part");
realPartText = new JTextField();
/*
  Place the label and the text widget in the box
*/

realPartBox.add(realPartLabel);
realPartBox.add(realPartText);
this.getContentPane().add(realPartBox);
```

And here is the result:

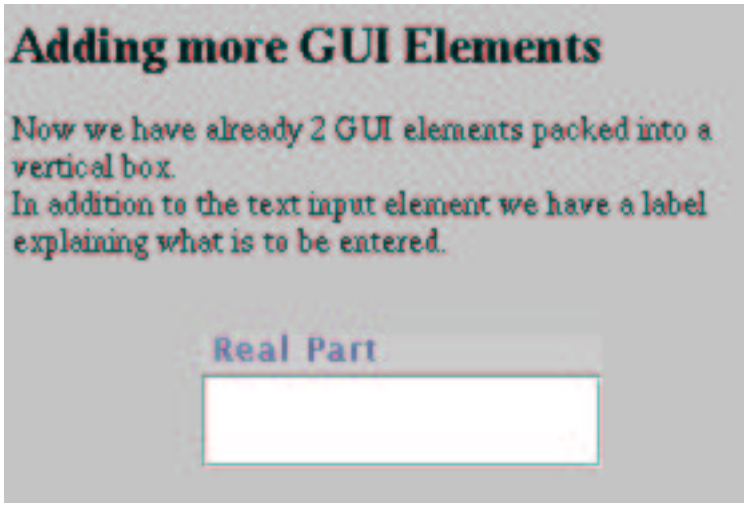


Figure 7.5: Two Applets in a Vertical Box

Since there is the equivalent to the vertical Box also in horizontal direction many layout problems can be solved this way. When many GUI elements are used however, this method gets very clumsy.

The BorderLayout

For this reason Java uses the concept of layout managers in order to define how the children of container widgets are placed. The default Layout manager of JApplet is the **BorderLayout** which has 5 fields into which the elements can be placed.

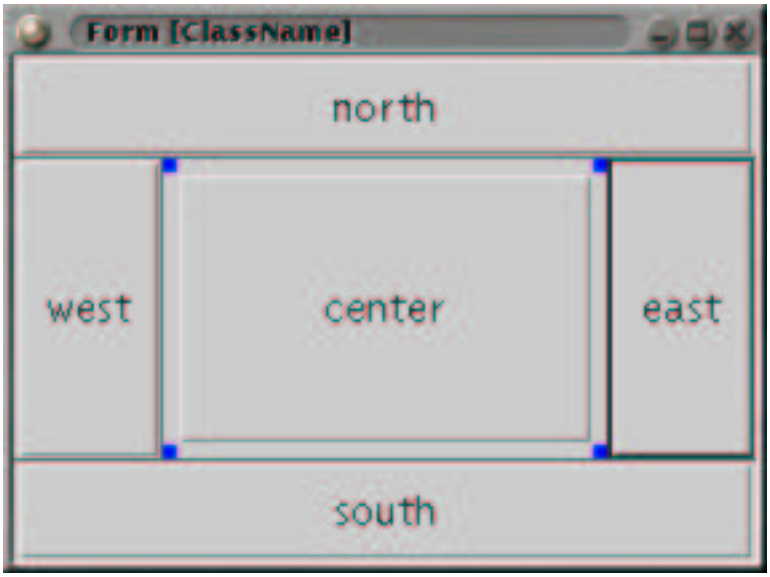


Figure 7.6: The BorderLayout

Instead of creating a vertical box and inserting our label and text widget in there, we could have taken advantage of the border layout manager associated with JApplet and placed the label in the North area of the layout while the text would have gone into the South area. The end result would have been essentially the same because the unused areas are shrunk to zero size.

```
public void init() {
    /*
      Create a Panel for real and imaginary part
      text inputs
    */
    JLabel     realPartLabel;
    JTextField realPartText;
    /*
      create the widgets
    */
    realPartLabel = new JLabel("Real Part");
    realPartText = new JTextField();
    /*
      Place the label and the text widget the content pane
```

```
      using the default border layout manager
    */

    this.getContentPane().add(realPartLabel,"North");
    this.getContentPane().add(realPartText,"South");

}
```

The GridLayout

Unfortunately the BorderLayout does not really map onto our problem. We want to have our widget laid out in a regular grid. When only looking at the 2 text input areas we would like to have them arranged as a 2x2 matrix. This is exactly what the GridLayout does. We will therefore create a GridLayout manager, attach it to the applets content pane and then insert our label widgets, which will go into the first row followed by text widgets which will appear below.

In addition, to make things look even prettier, we surround the 4 widgets with a **border**, the **BevelBorder**, which allows to make the widgets look lowered into the screen by setting its bevelType to Bevel.LOWERED (a constant defined in the BevelBorder object). Again there are many different border types and you are invited to look for the keywords: BevelBorder, LineBorder, EtchBorder, TitledBorder. . . in the Java API docs.

Now the Calculator display is already almost done.

In the next step we add all the buttons needed for number entry and for entry of commands like add, sub, div, mult, clear. In contrast to the GridLayout, the GridBagLayout allows the creations of elements of different size which are realised using so-called GridBagConstraints. These constraints are imposed on the element to be entered into the GridBag. gridx and gridy define the position while weightx and weighty define the amount of space (in %) to be taken up by the element. Like this we can optimise the calculator layout since the display area needs less space than all the buttons. Width and height can be used if you want to have one element take more than one slot in x or y direction.

```
/**
 *  In this stage we create the rest of the widgets.
 *  Now the number output widgets as well as the
 *  number and command buttons are ready for use.
 *  All that is missing are the labels on those buttons
 */
    public void init() {
/*
  Create a Panel for real and imaginary part
  text inputs
*/
JPanel             calcPanel;
```

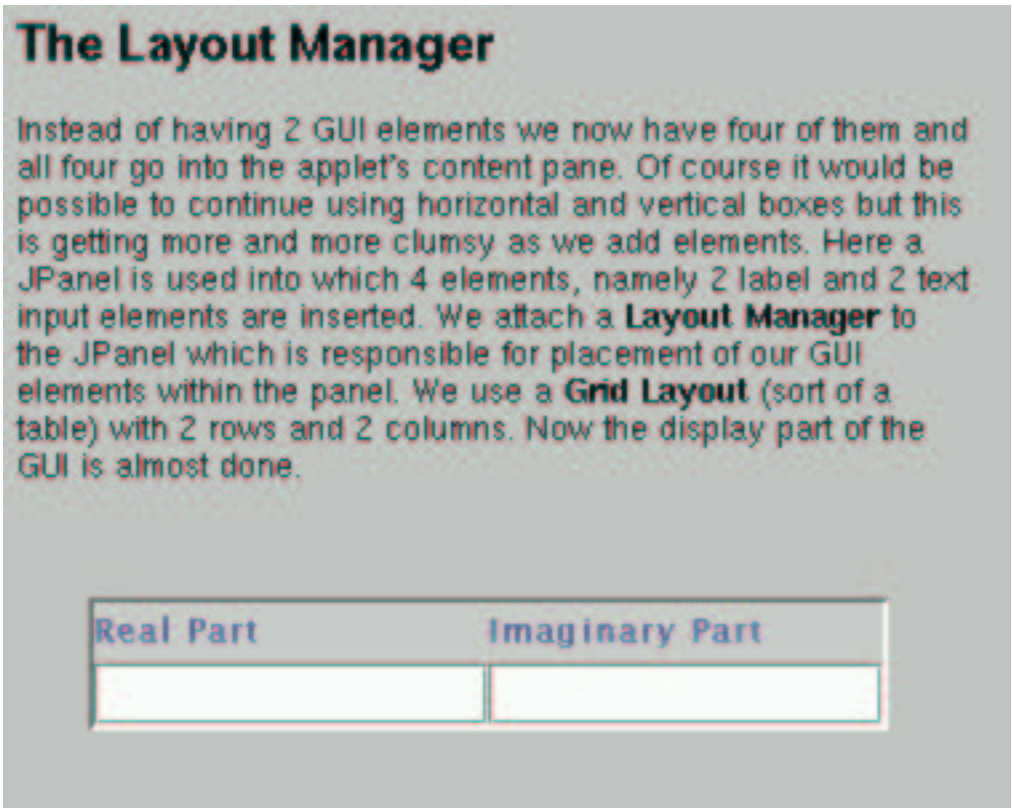


Figure 7.7: The Display Part of the Calculator

```
GridBagLayout      gridBagLayout = new GridBagLayout();
GridBagConstraints gridBagConstraints = new GridBagConstraints();
GridLayout         numberLayout;
GridLayout         numberInputLayout;
JPanel             numberPanel;
Box                inputBox;
JButton[]          numberInputButton;
JButton[]          operatorInputButton;
JPanel             numberInputPanel;
JPanel             operatorInputPanel;
JLabel             realPartLabel;
JTextField         realPartText;
JLabel             imagPartLabel;
JTextField         imagPartText;
BevelBorder        TextBorder;
/*
  create the widgets
*/
```

```
calcPanel = new JPanel(gridBagLayout);
inputBox  = Box.createHorizontalBox();
numberInputLayout = new GridLayout(4,3);
numberInputPanel   = new JPanel(numberInputLayout);
operatorInputPanel = new JPanel(numberInputLayout);
inputBox.add(numberInputPanel);
inputBox.add(operatorInputPanel);

numberInputButton = new JButton[12];
for (int i=0;i<12;i++)
    {
numberInputButton[i] = new JButton();
numberInputPanel.add(numberInputButton[i]);
    }
operatorInputButton = new JButton[12];
for (int i=0;i$<$12;i++)
    {
operatorInputButton[i] = new JButton();
operatorInputPanel.add(operatorInputButton[i]);
    }
numberPanel = new JPanel();
numberLayout  = new GridLayout(2,2);
numberPanel.setLayout(numberLayout);
realPartLabel = new JLabel("Real Part");
realPartText = new JTextField();
imagPartLabel = new JLabel("Imaginary Part");
imagPartText = new JTextField();
TextBorder = new BevelBorder(BevelBorder.LOWERED);
numberPanel.setBorder(TextBorder);
/*
  Place the label and the text widget in the box
*/
numberPanel.add(realPartLabel);
numberPanel.add(imagPartLabel);
numberPanel.add(realPartText);
numberPanel.add(imagPartText);

gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 0;
gridBagConstraints.gridwidth = 1;
gridBagConstraints.gridheight = 1;
gridBagConstraints.weightx = 100;
gridBagConstraints.weighty = 20;
gridBagConstraints.fill = GridBagConstraints.BOTH;
```

```
gridBagLayout.setConstraints(numberPanel,gridBagConstraints);
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 1;
gridBagConstraints.gridwidth = 1;
gridBagConstraints.gridheight = 1;
gridBagConstraints.weightx = 100;
gridBagConstraints.weighty = 80;
gridBagLayout.setConstraints(inputBox,gridBagConstraints);
calcPanel.add(numberPanel);
calcPanel.add(inputBox);
this.getContentPane().add(calcPanel);
    }
```

Once the labels are on the widget and 2 radio buttons, deciding into which field (the real or the imaginary one) button input has to go, the GUI is entirely finished. An interesting point may be the way how the numbers are put onto the number buttons:

```
numberInputButton = new JButton[12];
for (int i=0; i<10; i++)
    {
numberInputButton[i] = new JButton(Integer.toString(i));
numberInputPanel.add(numberInputButton[i]);
    }
```

The Integer class provides a conversion method from integer to String which we take advantage of in order to convert the loop index into a String, which is used as a button label when creating the JButton.

7.1.9 The Event Delegation Model

The GUI of the complex calculator is more or less ready and you have seen that it is a rather tedious task to put all the widgets together. For this reason graphical user interface builders have been built allowing you to create the GUI in an interactive manner. You click on graphical representations of JLabel, JButton, JTextField ... and place them in a container widget on the screen. At the same moment the GUI elements are created and visualised such that you can see what the final result is going to be. In order to make this possible the elements must be built in a well defined way such that they can act as software components which can be connected to other components a bit like when building a model out of Lego^{TM}; blocks (did you play with Legos when you were a child?). How exactly the software components, the Java Beans, are built, we will see in the next chapter.

Even though our complex calculator GUI may look quite pretty, it is not of much use yet. The reason is that nothing happens when we click the number

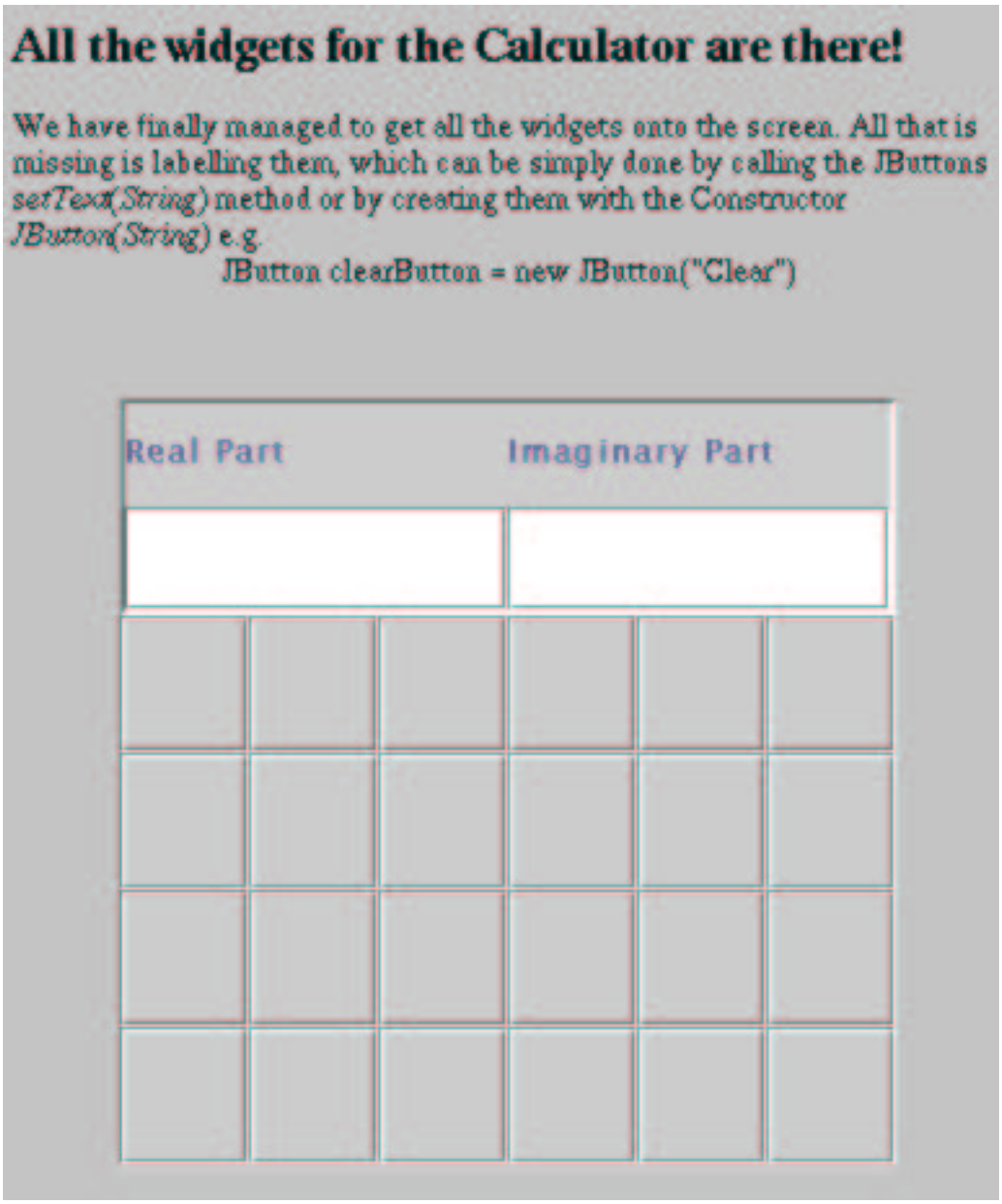


Figure 7.8: All the Calculator Widgets are there!

or operator buttons. We therefore have to look into the problem of activating the GUI. What actually happens when you press a number button?

A button click will be seen by the operating system which will pass this information to the Swing button. The sequence of *button down – button up* will be interpreted as a *button press* or, in other words, an **activation** of the button. The button will create an **ActionEvent** and send it to all **ActionListeners** attached

to it. This means that, in order to interact with the button, we will have to create an ActionListener and add it to the JButton's list of ActionListeners.

The ActionListener is a Java Interface with just a single method: **actionPerformed(ActionEvent)**. As explained in the introduction to GUI programming, this is usually implemented in the **Controller**. In order to know from which Object the event originated (which element was the **event source**) this information is ported in the ActionEvent. The event's method getSource() will return the Object that sent the event.

In our example, and for the moment, only JButtons can be event sources and each of our buttons has got a label on it. In order to find out which button had been pressed we therefore first get the reference to the button that triggered the event and then we read its label with the button's getText() method. Since we want to demonstrate the handling of events another textfield has been added to the complex calculator's user interface to display some text identifying the event. We will also need a simple method that allows us to write to this textfield from an outside object:

```
/*
 add the controller containing the action listeners
*/
 ComplexCalcController7 complexCalcController =
                        new ComplexCalcController7(this);
 numberInputButton = new JButton[12];
  for (int i=0; i<10; i++)
 {
numberInputButton[i] = new JButton(Integer.toString(i));
numberInputPanel.add(numberInputButton[i]);
numberInputButton[i].addActionListener(complexCalcController);
  }
numberInputButton[10] = new JButton(".");
numberInputPanel.add(numberInputButton[10]);
numberInputButton[10].addActionListener(complexCalcController);

numberInputButton[11] = new JButton("+/-");
numberInputPanel.add(numberInputButton[11]);
numberInputButton[11].addActionListener(complexCalcController);

operatorInputButton = new JButton[12];
operatorInputButton[0] = new JButton("+");
operatorInputPanel.add(operatorInputButton[0]);
/* activate the thing */
operatorInputButton[0].addActionListener(complexCalcController);
debugText = new JTextField(20);
        and so on ...
/**
```

```
 * Writes debug text to the debug text field
 * Used for demonstration of events
 */
    public void setDebugText(String debug)
    {
      debugText.setText(debug);
      return;
    }
```

This method is used by the controller which will find out the event source and print a text identifying the source. We do not use System.out.println in order to visualise the text on the page. Here of course, since this is only debug information we could have used System.out.println and either tested the widget with the appletviewer or we could have observed the print result on the system console.

```
/**
 * ComplexCalcController7.java
 * Created: Sat Aug 26 22:17:25 2000
 *
 * @author Ulrich Raich
 * @version 0.1
 */
public class ComplexCalcController7 implements ActionListener {
    ComplexCalcUI_Stage7 parent;
    public ComplexCalcController7 (ComplexCalcUI_Stage7 p) {
      parent = p;
    }
    public void actionPerformed(ActionEvent e)
    {
JButton activatedButton;
String  buttonLabel;
activatedButton = (JButton)e.getSource();
buttonLabel = activatedButton.getText();
if (buttonLabel.equals("+"))
    {
parent.setDebugText("Add Button");
return;
    }
if (buttonLabel.equals("-"))
    {
parent.setDebugText("Sub Button");
return;
    }
```

```
if (buttonLabel.equals("*"))
    {
parent.setDebugText("Mult Button");
return;
    }
if (buttonLabel.equals("/"))
    {
parent.setDebugText("Div Button");
return;
    }
if (buttonLabel.equals("+/-"))
    {
parent.setDebugText("Change Sign Button");
return;
    }
if (buttonLabel.equals("="))
    {
parent.setDebugText("Equals Button");
return;
    }
if (buttonLabel.equals("."))
    {
parent.setDebugText("Dot Button");
return;
    }
if (buttonLabel.equals("Clear"))
    {
parent.setDebugText("Clear Button");
return;
    }
if (buttonLabel.equals("Norm"))
    {
parent.setDebugText("Norm Button");
return;
    }
for (int i=0; i<10; i++)
    if (buttonLabel.equals(Integer.toString(i)))
    {
parent.setDebugText("Number:" + i);
return;
    }
    }
}
```

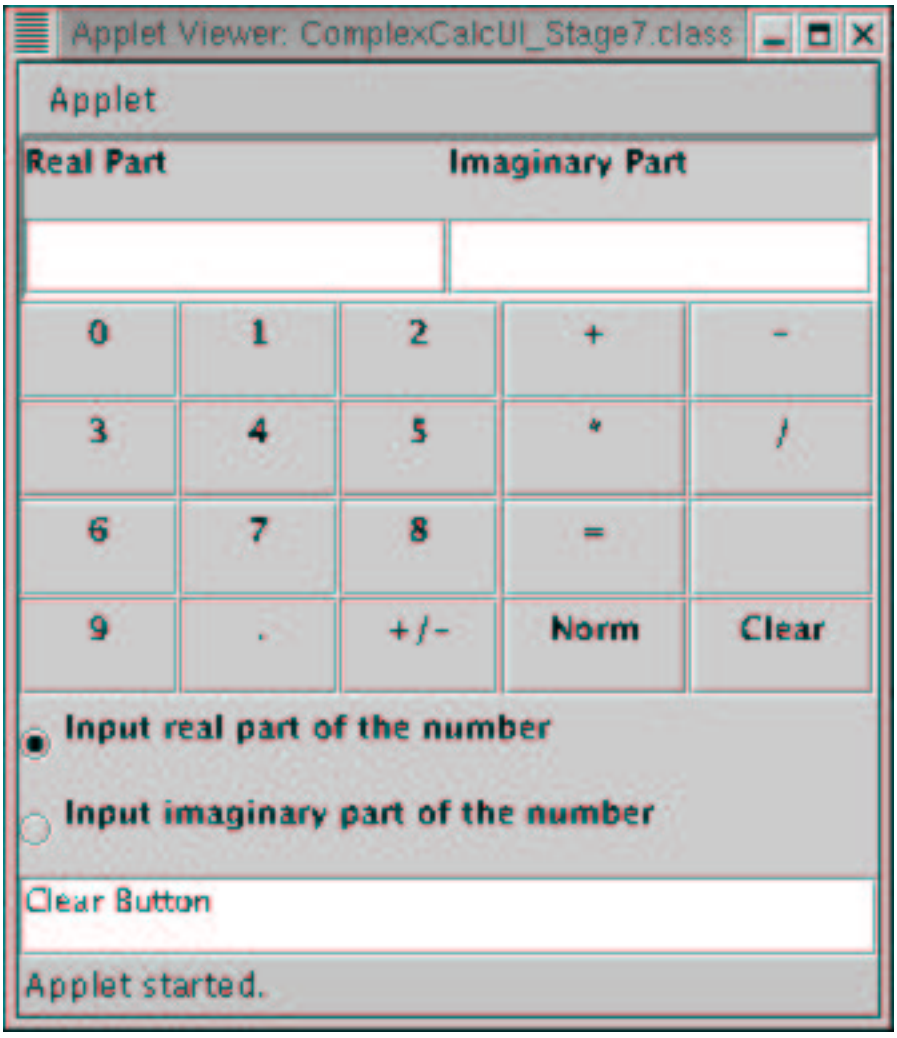


Figure 7.9: First Test of Activation

7.1.10 The Calculator Model

The most complex part of any GUI base program is the **model** and at the same time it is the one I explain least. It is the model that implements the actual problem solving. The *user interface* merely provides pretty buttons, pull-down menus and the like while it does not do much, seen from the functional point of view. The *controller* simply receives events from the user interface and dispatches them to the model. Again not much is done from the functional point of view. It is the model that does the actual data treatment. On the other hand, programming the model only uses ”standard” programming concepts and there is nothing new to be learned.

In the case of the calculator you will find all the routines that are needed to handle digits newly entered, which are added to the already available digits,

the switch from entering numbers from the real part entry field to the imaginary part field, the conversion of the series of digits entered into doubles and later complex numbers, the handling of operators ("+", "-", "*", "/") and of the course calculations themselves (when "=" is pressed).

The whole program now works as follows: The user presses a button (e.g. a digit) which triggers an *actionEvent*. This event is captured and interpreted by the controller which in turn informs the model which action needs to be taken. To do this, it calls a model method.

The method modifies the model's internal state (a digit is added to the number entered) which must be reflected in the user interface. The controller therefore informs the view by calling one of its methods, which in turn updates its display on the screen.

You will find the complete source code of the model in the appendix. Please have a look at it. Note that most of the work is actually the conversion from doubles (well, actually Complex, the type used for calculations) to byte arrays and back. Each time a modification is made, the conversions are performed in order to make sure that the double respresentation and the byte arrays always correspond.

And the final result will essentially look like figure 7.9

7.2 Software Components, Java Beans

7.2.1 What is a Bean?

Beans are **re-usable software modules** with strictly defined interfaces that can be hooked together by a graphical user interface builder. Of course, most beans will be visible user interface elements (all of the Swing components like JButton, JLabel, JTextField... as well as the container components are beans) but even the visibility is not a necessary criterion.

Apart from obeying certain rules, beans are just ordinary Java objects. Just like any other object, beans have internal variables, here called properties, which may be modified by the objects methods.

As we have seen in the calculator example, the objects that make up the calculator applet interact with each other by means of events. Since beans, in order to be treated by a GUI builder must be clearly defined entities with no cross links, beans only use events for their communication.

The questions we now have to ask are: "How does the GUI builder discover the capabilities of the bean? How does it know, which properties are implemented in the bean and how to read and modify their values? How does it know which events are used for communication and which bean is considered to create the event and which one is capturing it?"

The magic buzz-word is **introspection**. Let us first have a look at properties. If a bean has got a property called *prop* then it must implement two methods named: `void` **`setProp`**`(type val);` `type` **`getProp`**`()` in order to expose them.

The very first beans example does not attempt to get a usable bean, it simply tries to show the sequence of steps needed to create a bean. The bean itself is a simple java object, not extending any Swing component which means that it is not going to be visible. Note that the bean implements the Serializable interface which does not require any supplementary code but add the capability to the bean to save itself onto a file (serialize itself). This functionality is very important because, after having built an application with a large number of beans, which have all been customized (their properties have been set) we do not want to loose this work but we want to be able to save it.

```
import java.beans.*;
import java.io.Serializable;

/**
 * bean1.java
 *
 *
 * Created: Sat Nov 24 13:13:42 2001
 *
 * @author <a href="mailto:uli@localhost.localdomain">Ulrich Raich</a>
 * @version
 */

public class bean1 implements Serializable{
  public bean1 (){

  }
    double result;
    /**
     * Get the value of result.
     * @return value of result.
     */
    public double getResult() {
      return result;
    }

    /**
     * Set the value of result.
     * @param v  Value to assign to result.
     */
    public void setResult(double  v) {
      this.result = v;
    }
}// bean1
```

The bean only has a single property of type double called result and it exposes a get and a set method for this property. In order to make a bean out of this code

it must be compiled and the resulting class file must be packed together with a manifest file into a jar packet. Here is the Makefile that does exactly this:

```
# This makefile delivers the bean1 bean into the beansbox
# in a single JAR file.

BEANSDIR= /opt/ICTP/lectures/lectureNotes/Java/code/beans/BDK1.1/jars
CLASSFILES= \
        bean1.class

JARFILE= ../../jars/bean1.jar

all: $(JARFILE)

# Create a JAR file with a suitable manifest.

$(JARFILE): $(CLASSFILES) $(DATAFILES)
        echo "Name: bean1.class" >> manifest.tmp
        echo "Java-Bean: True" >> manifest.tmp
        echo "" >> manifest.tmp
        jar cfm $(JARFILE) manifest.tmp bean1.class

        @/bin/rm manifest.tmp

# Rule for compiling a normal .java file
%.class: %.java
        export CLASSPATH; CLASSPATH=. ; \
        javac $<

install:
        cp $(JARFILE) $(BEANSDIR)
clean:
        /bin/rm -f *.class
        /bin/rm -f *.ser
        /bin/rm -f $(JARFILE)
```

The manifest contains the following text:

```
Name:  bean1.class
Java-Bean:  True
```

It states that the content of this package contains a bean of name bean1. It is possible to package several beans into a single jar package if they are denoted as described above.

7.2.2 The Beanbox

As we said at the beginning, this bean does not provide much functionality and it will not even be visible when used in a GUI builder.

SUN provides a very simplistic GUI builder designed for testing user created beans in its **B**eans **D**evelopment **K**it, the **beanbox**. This program will search

through a directory of jar files and make all beans found in this directory available for test. All we have to do is therefore copying our beans jar file into the directory searched by the beanbox.

Once the beanbox is started we find back the bean in its toolbox. Clicking the text will modify the cursor to a cross, clicking on the BeanBox now, make the bean appear on the screen. The BeanBox is in *design mode* in which case the bean is displayed with this strange border indicating that in the later application or applet it will be invisible.

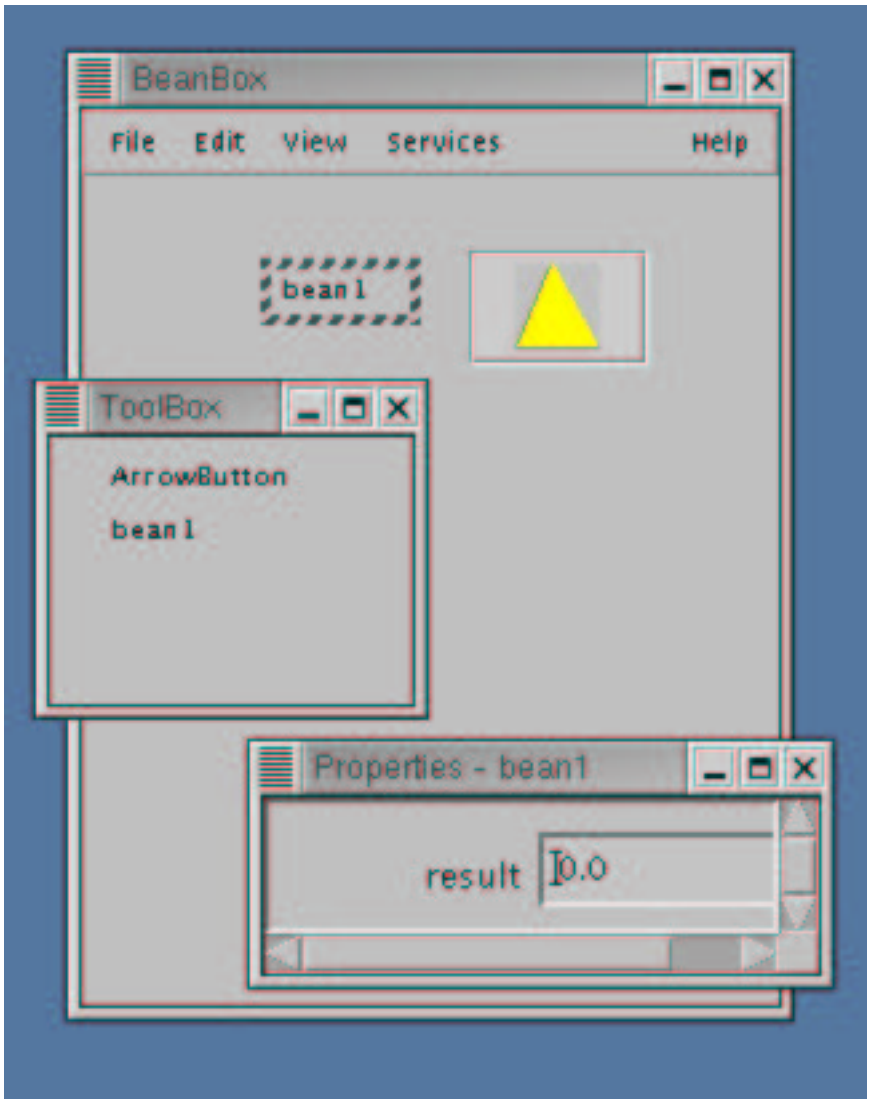


Figure 7.10: Bean1 sitting in the beanbox

In order to provide a little more realistic example, let us consider an arrow button, which is a simple JButton that shows an arrow pointer in any direction

up, down, left or right. From this description it becomes clear immediately that we will extend a JButton, use icons for the arrows and have a property that can be *up*, *down*, *left* or *right*. Please note the way, the icons are read: We use a URL in order to get at the gif files that are transferred in the jar package. Due to security reasons an applet is not allowed to access the local file system. We can get at resources within the jar file though.

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

public class ArrowButton extends JButton implements Serializable {
    Image arrowImage;

    public ArrowButton() {
        super();
        createIcons();
        arrowDirection = ARROW_UP;
        this.setIcon(images[arrowDirection]);
    }

    public ArrowButton(int direction) {
        super();
        createIcons();
        if ((direction < 0) || (direction > MAX_DIRECTIONS))
            arrowDirection = ARROW_UP;
        else
            arrowDirection = direction;

        this.setIcon(images[direction]);
    }

  public void createIcons()
  {
        try {
          java.net.URL url = getClass().getResource("images/up.gif");
          images[ARROW_UP] = new ImageIcon(url);
        } catch (Exception e)
          {
            System.out.println(e.getMessage());
          }
        try {
          java.net.URL url = getClass().getResource("images/down.gif");
          images[ARROW_DOWN] = new ImageIcon(url);
        } catch (Exception e)
          {
            System.out.println(e.getMessage());
          }
        try {
```

```
          java.net.URL url = getClass().getResource("images/left.gif");
          images[ARROW_LEFT] = new ImageIcon(url);
        } catch (Exception e)
          {
            System.out.println(e.getMessage());
          }
        try {
          java.net.URL url = getClass().getResource("images/right.gif");
          images[ARROW_RIGHT] = new ImageIcon(url);
        } catch (Exception e)
          {
            System.out.println(e.getMessage());
          }
  }
    /**
       * Get the value of arrowDirection.
       * @return Value of arrowDirection.
       */
    public int getArrowDirection() {return arrowDirection;}
    /**
       * Set the value of arrowDirection.
       * @param v  Value to assign to arrowDirection.
       */
    public void setArrowDirection(int  v)
    {
        if ((v < 0) || v > MAX_DIRECTIONS)
            return;
        this.arrowDirection = v;
        if (images[v] == null)
            images[v] = new ImageIcon(
                                    "images/"
                                    + imageFilename[v]);
        this.setIcon(images[v]);

    }
    ImageIcon[] images = new ImageIcon[4];

    private             int arrowDirection;
    public static final int ARROW_UP    = 0;
    public static final int ARROW_DOWN  = 1;
    public static final int ARROW_RIGHT = 2;
    public static final int ARROW_LEFT  = 3;

    private final int MAX_DIRECTIONS = 4;

    static ImageIcon DownPic;
    static final String[] imageFilename =
{"up.gif", "down.gif", "right.gif","left.gif"};
   }
```

The Makefile needs some brush-up as well since we have to add the images into the jar package. The code snippet below is of course not complete, the rest of the Makefile does not change however.

```
ICONS=images
all: $(JARFILE)
# Create a JAR file with a suitable manifest.
$(JARFILE): $(CLASSFILES) $(DATAFILES)
        echo "Name: ArrowButton.class" >> manifest.tmp
        echo "Java-Bean: True" >> manifest.tmp
        echo "" >> manifest.tmp
        jar cfm $(JARFILE) manifest.tmp ArrowButton.class $(ICONS)/*
        @/bin/rm manifest.tmp
```

7.2.3 The BeanInfo class

When we instantiate the ArrowButton bean in the beanbox, it will appear as a normal button and the arrow will be seen normally (this is now a visible bean since it is extended from a Swing Component!). However we can also see that the property box is entirely full even though we only defined the arrowDirection property with set/get methods. The reason for this is the subclassing of JButton. We do not only see the ArrowButtons properties but all the properties of its super class. Note however that the arrowDirection property can actually be changed with the property editor, which will make the arrow turn.

How can we avoid that all those properties, which we want to keep as default values, will be proposed for change? This can be done with a **BeanInfo** class. If our bean is named ArrowButton, then we will have to define a new class named ArrowButtonBeanInfo and here it is:

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class ArrowButton2BeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor arrowDirection =
                        new PropertyDescriptor("arrowDirection",
beanClass);
            PropertyDescriptor rv[] = {arrowDirection};
            return rv;
        } catch (IntrospectionException e) {
            throw new Error(e.toString());
        }
    }
    private final static Class beanClass = ArrowButton2.class;
}
```

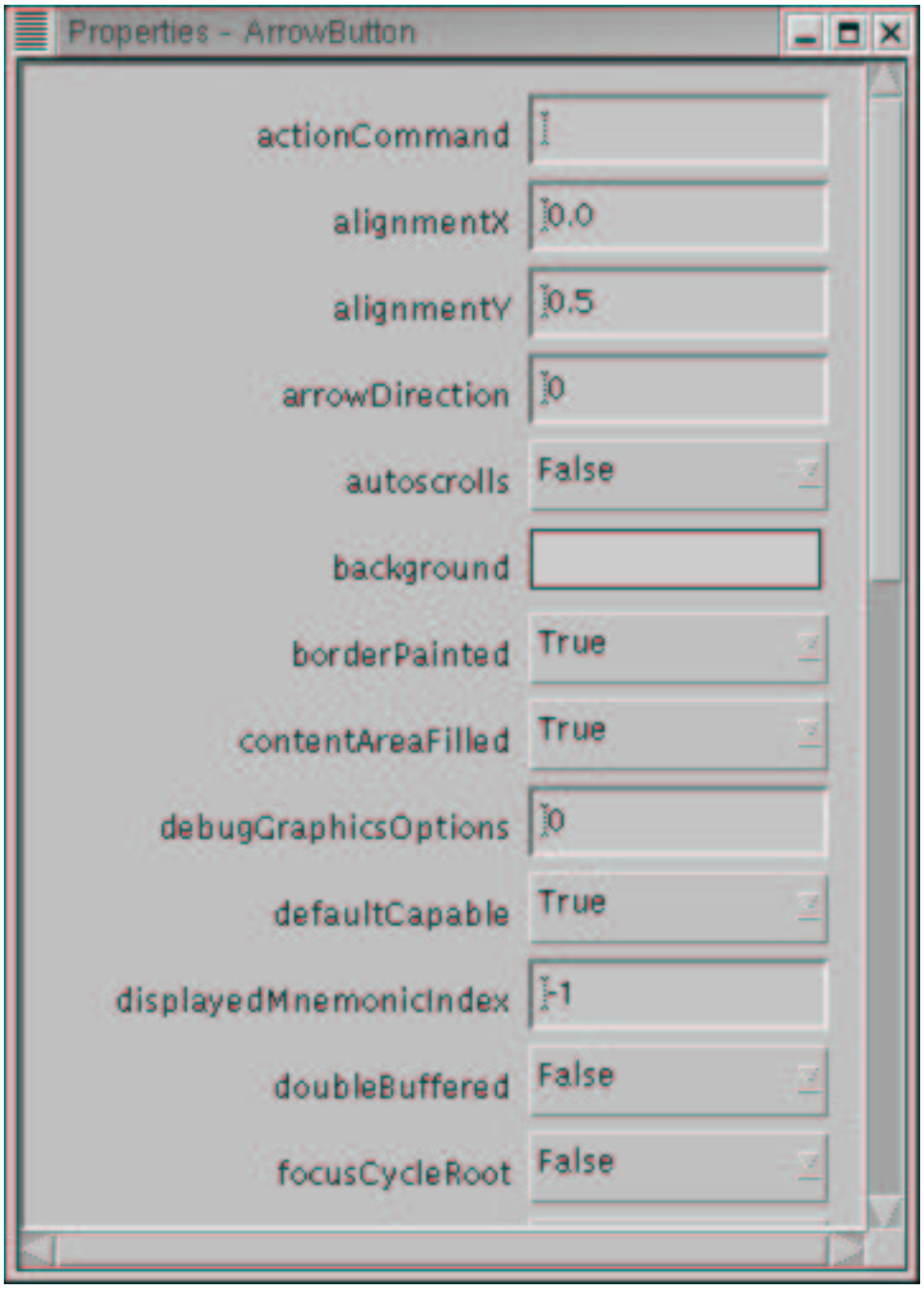


Figure 7.11: The Properties of the ArrowButton

An array of *PropertyDescriptors* is created and returned in the *getPropertyDescriptors()* method. Only those properties explicitly exposed in the PropertyDescriptor will be proposed for customisation.

Still the property editor is not as nice as it could be because the direction can actually only take 4 values while it is implemented and seen by the property editor as an integer. In order to give the user only those 4 possibilities we will have to customise the property editor.

7.2.4 A customised Property Editor

Of course there are different possible ways of proposing to change the arrow directions. When we have a set of possible values then implementing just a few methods of the PropertyEditor interface and announcing this property editor in the bean info file is enough. It is however also possible to write a custom property editor as we would have been obliged to do if we wanted to enter complex numbers.

Here is the property editor and a screen dump showing the result.

```
import java.beans.*;

/**
 * ArrowButtonDirectionNameEditor.java
 *
 *
 * Created: Sun Jan 16 15:58:36 2000
 *
 * @author Ulrich Raich
 * @version
 */

public class ArrowButton3DirectionNameEditor extends
    PropertyEditorSupport {

    public String[] getTags()
    {
        String directions[] = {"Up","Down","Left","Right"};
        return directions;
    }
    public String getAsText() {
        Integer direction = (Integer)getValue();
        switch(direction.intValue()) {
        case ArrowButton3.ARROW_UP:    return "Up";
        case ArrowButton3.ARROW_DOWN:  return "Down";
        case ArrowButton3.ARROW_LEFT:  return "Left";
        case ArrowButton3.ARROW_RIGHT: return "Right";
        default: return null;
        }
    }

    public void setAsText(String text)
throws IllegalArgumentException
```

```
        {
            if (text.equals("Up"))
               setValue(new Integer((int)ArrowButton3.ARROW_UP));
            if (text.equals("Down"))
               setValue(new Integer((int)ArrowButton3.ARROW_DOWN));
            if (text.equals("Left"))
               setValue(new Integer((int)ArrowButton3.ARROW_LEFT));
            if (text.equals("Right"))
               setValue(new Integer((int)ArrowButton3.ARROW_RIGHT));
        }

    public String getJavaInitializationString() {
        return (String)getValue();
    }

} // ArrowButtonDirectionNameEditor
```

. . . and the modified bean info

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class ArrowButton3BeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor arrowDirection =
                        new PropertyDescriptor("arrowDirection",
beanClass);

            arrowDirection.setPropertyEditorClass(
                        ArrowButton3DirectionNameEditor.class);

            PropertyDescriptor rv[] = {arrowDirection};
            return rv;
        } catch (IntrospectionException e) {
            throw new Error(e.toString());
        }
    }

    private final static Class beanClass = ArrowButton3.class;
}
```

7.2.5 Beans and Events

Since beans are handled as autonomous entities it is not possible to directly call methods of a bean from within another bean. You can never be sure that the

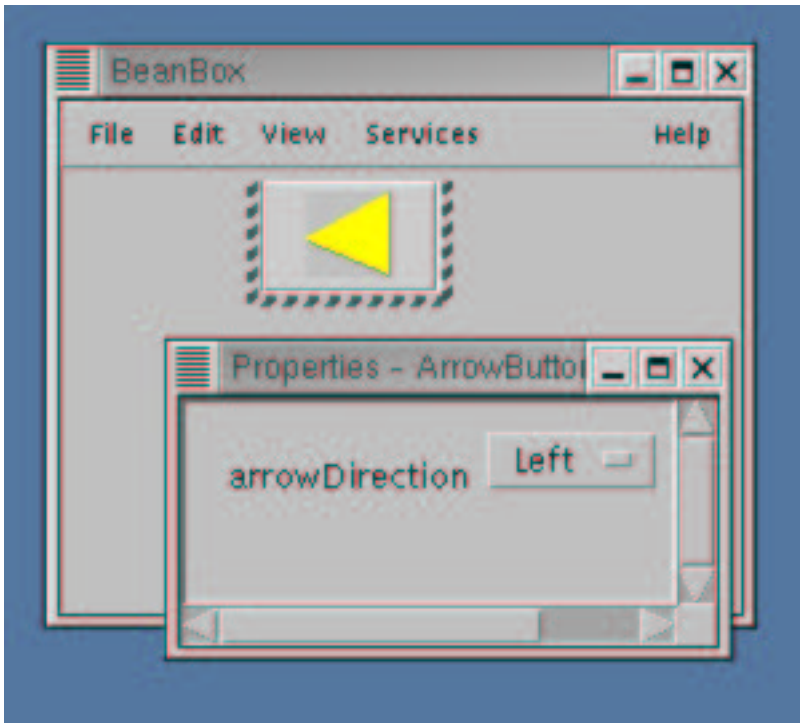


Figure 7.12: The Property Editor for the Arrow Directions

user of the GUI builder has actually instantitated both beans. How then can beans communicate with each other? Again this is done using **events**. Our ArrowButton, since it is a subclass of JButton is able to create ActionEvents. These ActionEvents can be caught by any ActionListener. The BeanBox is capable to connect a bean's method to an event by interspersing so called event adapter classes (other methods for doing the same thing exist).

If we add an increment method to our bean1 we can connect this method to the ActionEvent created when the ArrowButton is pressed. Each time the user presses the button, the number will be increased by one.

Looking at the events presented by the beanbox the user will again be bewildered by their big number. As with the properties this is due to the subclassing and as with properties we can restrict the events displayed to the essentials using the bean info class.

This method must be added to the bean info class in order to achieve the clean-up:

```
    public EventSetDescriptor[] getEventSetDescriptors() {
        try {
            EventSetDescriptor push =
new EventSetDescriptor(beanClass,
                           "action",
```

```
                            java.awt.event.ActionListener.class,
                            "actionPerformed");
            push.setDisplayName("action");
            EventSetDescriptor[] rv = { push };
            return rv;
        } catch (IntrospectionException e) {
            throw new Error(e.toString());
        }
    }
```

7.2.6 Bounded Properties

When developing the complex calculator we have already seen that the GUI part of the project needed update each time the model changes its internal state. It seems therefore logical to provide a PropertyChangeEvent which the model fires each time any of its properties changes. The ComplexCalcUI then only needs to provide the necessary methods that take an PropertyChangeEvent as parameter just as our simple beans has done for ActionEvents.

Once these conditions are fulfilled then we can connect the Model with the userinterface by means of an automatically generated event adapter. Of course the model, which becomes an event source, must supply additional code such that an PropertyChangeListener can be added and removed to a list of objects to be informed of property changes.

One following code shows the simple invisible bean created at the beginning of this chapter augmented with the capability of firing PropertyChangeEvent s. In addition we create a NumberField bean which can take the PropertyChangeEvents and display the latest number stored in the bean firing the events. A PropertyChangeEvent always contains the old and the new values and it is therefore simple to perform the updates of the NumberF ield. Now we can connect the Arrowbutton to the simple bean for increment and decrement using ActionEvents. The simple bean in turn is connected to the Numberfield through the PropertyChangeEvent for display of its current state.

```
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;
import javax.swing.*;
import java.beans.*;

/**
 * bean3.java
 *
 *
 * Created: Sat Nov 24 13:13:42 2001
 *
 * @author <a href="mailto:uli@localhost.localdomain">Ulrich Raich<</a>>
```

```
 * @version
 */

public class bean3 implements Serializable {
  public bean3 (){
    changes  = new  PropertyChangeSupport(this);
  }
  private double result;

  /**
   * Get the value of result.
   * @return value of result.
   */
  public double getResult() {
    return result;
  }

  /**
   * Set the value of result.
   * @param v  Value to assign to result.
   */
  public void setResult(double  v) {
    double oldValue;
    oldValue = result;
    changes.firePropertyChange("value",
new Double(oldValue),new Double(v));
    this.result = v;
  }

  /**
   * increments the value
   */
  public void increment(ActionEvent e)
  {
    double oldValue;
    oldValue = result;
    result+=1.0;
    changes.firePropertyChange("value",new Double(oldValue),
                               new Double(result));
  }

  /**
   * decrements the value
   */
  public void decrement(ActionEvent e)
  {
    double oldValue;
    oldValue = result;
    result-=1.0;
    changes.firePropertyChange("value",
```

```
new Double(oldValue),new Double(result));
  }
  /*
    here we collect all Listeners to whom we sent the
    propertyChange events
  */

  public void addPropertyChangeListener(PropertyChangeListener l)
  {
    changes.addPropertyChangeListener(l);
  }
  public void removePropertyChangeListener
(PropertyChangeListener l)
  {
    changes.removePropertyChangeListener(l);
  }

    private PropertyChangeSupport  changes;
}// bean3



import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;
import javax.swing.*;
import java.beans.*;
import java.text.*;

public class NumberField extends JTextField implements
                     PropertyChangeListener,Serializable {

  double number;
  public NumberField() {
    super(16);
  }

  /**
   * Get the current value
   * @return double value
   */
  public double getNumber()
  {
    return number;
  }

  /**
   * Set the value
   * @param v  Value to assign
   */
```

```
  public void setNumber(double v)
  {
    String valString;
    DecimalFormat df = new DecimalFormat("0.0######");
    valString = df.format(v);
    setText(valString);
    number = v;
    return;
  }

  public void propertyChange(PropertyChangeEvent e)
  {
    String valString;
    double newValue;
    newValue = ((Double)e.getNewValue()).doubleValue();
    DecimalFormat df = new DecimalFormat("0.0######");
    valString = df.format(newValue);
    setText(valString);
    number = newValue;
    return;
  }

}
```

7.3 Conclusions and Acknowlegements

7.3.1 Conclusions

This ends our excursion into the world of WEB programming. Of course these lectures will only allow you a quick glimpse on the opportunities opened by this computer science field.

If we managed to stimulate your curiosity and we showed you how to go on from here, then these lectures were a success. Please note that, even though we are using Linux during the college, this is by no means a requirement for WEB programming. All we showed you during these lectures can be applied to the operating systems like MS Windows of MacOS.

7.3.2 Acknowlegements

Giving these lectures would not have been possible without the consent of my employer CERN. Also my wife Dong Ye and my children Melanie and David have suffered seeing their husband and father sitting in front of the computer for too long hours. Thanks for their understanding.

7.4 Appendixes

7.4.1 HC-11 test procedure

```
<html>
<head>
<title>HC-11 test procedure</title>
</head>

<h1><center><font size = 7>HC-11 Test</font></center></h1>
<form method="POST" action="http://localhost:8080/cgi-bin/hc11.cgi">
<p>
<table  border="0" cellpadding="10" cellspacing="0">
<TR><TD>
<font size=5>Device</font>
<select name="device">
<option value="LCD"> LCD </option>
<option selected value="LEDs"> LEDs </option>
<option value="Switches"> Switches </option>
<option value="Buttons"> Buttons </option>
<option value="ADC"> ADC </option>
<option value="Scope"> Digital Scope Trace </option>
</select>
</td>
<td>
<p><font size=5>Command</font>
<input type="radio" name="command" value="read"> read
<input type="radio" name="command" value="write" checked> write
<input type="radio" name="command" value="ioctl"> ioctl
<p>
</td></tr>
<tr><td>
<br>
<br><br>
<center> <font size=7>LCD</font></center>
<br>
LCD contents, don’t exceed 16 chars
<p>
<input type = "text" name ="LCD_Data" MAXLENGTH=16 VALUE="Hello World !">
LCD Text
<p>
<input type="radio" name="lcdIoctl" value="select"> select LCD
<input type="radio" name="lcdIoctl" value="deselect"> deselect LCD
<input type="radio" name="lcdIoctl" value="clear"> clear
<input type="radio" name="lcdIoctl" value="home"> home
<p>
</td>
<td>
<center><font size=7>LED</font></center>
```

```
<br>
LED Value, highest significant bit first
<p>
<input type="checkbox" name="LED_data" value="b8"> 8
<input type="checkbox" name="LED_data" value="b7"> 7
<input type="checkbox" name="LED_data" value="b6"> 6
<input type="checkbox" name="LED_data" value="b5"> 5
<input type="checkbox" name="LED_data" value="b4"> 4
<input type="checkbox" name="LED_data" value="b3"> 3
<input type="checkbox" name="LED_data" value="b2"> 2
<input type="checkbox" name="LED_data" value="b1"> 1
<p>
</td></tr>
<tr><td>
<p>
<input type="submit" name="submit1" value="Submit Request">
<input type="reset" value="Reset Request">
</td></tr>
</table>

</form>
</body>
</html>
```

7.4.2 The GCI program for the HC-11 interface

```
#include <stdio.h>
#include <cgic.h>
#include <ICTP_IO.h&gt
#include <protocol.h>
#include <string.h>
#include <stdarg.h>

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

int  Device();
int  Command();

int  checkPars();
int  protSendMsg();
void reportError(int);
void cgi_fprintf(FILE *stream, char *fmt,...);

/*
   needed as globals
*/
```

```
int cmdChoice;
int deviceChoice;
int ioctlChoice;
int dataSize;
int ICTP_IO_debugLevel = ICTP_IO_DEBUG_ERROR;
int ICTP_IO_msgType    = ICTP_IO_HTML;

char *cmds[] = {
  "invalid",
  "read",
  "write",
  "ioctl"
};

char *ioctlCmds[] = {
  "serverId",
  "monitor",
  "select",           /* LCD ioctl cmds */
  "deselect",
  "clear",
  "home"
};


char *devices[] = {
  "ServerId",
  "Switches",
  "ADC",
  "Buttons",
  "LEDs",
  "LCD",
  "Serial",
  "Scope"
};


int cgiMain()
{
  int retCode;
  //#define DEBUG 1

  ICTP_IO_setMsgType(ICTP_IO_msgType);
  ICTP_IO_setDebugLevel(ICTP_IO_debugLevel);


#if DEBUG
/* Load a saved CGI scenario if we're debugging */
cgiReadEnvironment("/tmp/capcgi.dat");
#endif
```

```
if (ICTP_IO_msgType == ICTP_IO_HTML)
  cgiHeaderContentType("text/html");
else
  cgiHeaderContentType("text/plain");

if ((retCode = Device())  != ICTP_IO_SUCCESS)
  {
    reportError(retCode);
    cgi_fprintf(cgiOut, "Device\n");
    return 0;
  }

if ((retCode = Command()) != ICTP_IO_SUCCESS)
  {
    reportError(retCode);
    cgi_fprintf(cgiOut, "Command\n");
    return 0;
  }
else
    cgi_fprintf(cgiOut, "Command ok\n");

if ((retCode = checkPars()) != ICTP_IO_SUCCESS)
  {
    reportError(retCode);
    cgi_fprintf(cgiOut, "checkPars\n");
    return 0;
  }
else
  {
    /*
      open the connection to the HC11
    */

    if ((retCode = ICTP_IO_Open()) != ICTP_IO_SUCCESS)
      {
cgi_fprintf(cgiOut,
"Could not open HC-11 connection, error: %d\n \n",
retCode);
reportError(retCode);
exit(0);
      }

    if ((retCode = protSendMsg()) == ICTP_IO_SUCCESS)
      cgi_fprintf(cgiOut,
"%s successfully executed\n",cmds[cmdChoice]);
    else
      cgi_fprintf(cgiOut, "%s failed\n",cmds[cmdChoice]);
    reportError(retCode);

    if (ICTP_IO_Close() != ICTP_IO_SUCCESS)
```

```
      {
cgi_fprintf(cgiOut,
"Could not close HC-11 connection, error: %d\n \n",
retCode);
      }
   }// if checkPars
return 0;
}

int Device() {
cgiFormResultType retCode;
retCode = cgiFormSelectSingle("device",
devices, 7, &amp;deviceChoice, 0);

switch (retCode)
  {
  case cgiFormSuccess:
    cgi_fprintf(cgiOut, "Selected Device: %s\n",
    devices[deviceChoice]);
    return ICTP_IO_SUCCESS;
    break;
  case cgiFormNotFound:
    cgi_fprintf(cgiOut,"Could not find Form \n");
    break;
  case cgiFormNoSuchChoice:
    cgi_fprintf(cgiOut,"Could not find Choice \n");
    break;
  default:
    cgi_fprintf(cgiOut,"Error Code unknown %d \n", retCode);
    break;
  }
return ICTP_IO_ILLEGAL_DEVICE;
}

int Command() {
cgiFormResultType retCode;
retCode = cgiFormRadio("command", cmds, 4, &amp;cmdChoice, 0);
switch (retCode)
  {
  case cgiFormSuccess:
    cgi_fprintf(cgiOut, "Command: %s Code: %d\n",
cmds[cmdChoice],cmdChoice);
    return ICTP_IO_SUCCESS;
    break;
  case cgiFormNotFound:
    cgi_fprintf(cgiOut,"Cmd: Could not find Form \n");
    break;
  case cgiFormNoSuchChoice:
    cgi_fprintf(cgiOut,"Cmd: Could not find Choice \n");
    break;
```

```
  default:
    cgi_fprintf(cgiOut,"Cmd: Error Code unknown \n");
    break;
  }
return ICTP_IO_ILL_REQUEST;
}


int checkPars()
{
  int retCode;
  if ((deviceChoice+1 < ICTP_IO_SWITCHES) |
(deviceChoice+1 > ICTP_IO_SERIAL))
    {
      cgi_fprintf(cgiOut,"Illeagal Device Code: %d\n \n",
deviceChoice);
      return ICTP_IO_ILLEGAL_DEVICE;
    }

  if ((cmdChoice < ICTP_IO_OPEN) |
(cmdChoice > ICTP_IO_CLOSE))
    {
      cgi_fprintf(cgiOut,"Illeagal Command Code: %d\n \n",
cmdChoice);
      return  ICTP_IO_ILL_REQUEST;
    }
  /*
    now check for data size and data
  */

  /*
     most times data size = 1
  */

  switch (cmdChoice)
    {
    case ICTP_IO_WRITE:
      cgi_fprintf(cgiOut,"checkPars: write\n");

      switch (deviceChoice)
{
case ICTP_IO_LCD:
  dataSize = 16;
  return ICTP_IO_SUCCESS;
case ICTP_IO_LED:
  return ICTP_IO_SUCCESS;
default:
  cgi_fprintf(cgiOut,"Devices are read only!\n");
  return ICTP_IO_RDONLY;
}
```

```
      break;

    case ICTP_IO_READ:
      switch (deviceChoice)
{
case ICTP_IO_SERVER:
case ICTP_IO_SWITCHES:
case ICTP_IO_BUTTONS:
case ICTP_IO_LED:
case ICTP_IO_ADC:
  dataSize = 1;
  /*
  if (dataSize != 1)
    {
      cgi_fprintf(cgiOut,"Illegal Data Size: %d\n \n",
dataSize);
      return ICTP_IO_OUT_OF_RANGE;
    }
  */
  break;
case ICTP_IO_LCD:
  dataSize=16;
  break;
default:
  cgi_fprintf(cgiOut,"Devices are write only!\n");
  return ICTP_IO_WRONLY;
  break;
}

      break;

    case ICTP_IO_IOCTL:
      retCode = cgiFormRadio("lcdIoctl", ioctlCmds, 7,
&amp;ioctlChoice, 0);
      if (retCode == cgiFormSuccess)
  cgi_fprintf(cgiOut, "Ioctl Command: %s \n",
  ioctlCmds[ioctlChoice]);
      else
{
  retCode = cgiFormInteger("ioctlCmd", &amp;ioctlChoice, 0);
  if (retCode == cgiFormSuccess)
    {
      cgi_fprintf(cgiOut, "ioctl command: %d\n",ioctlChoice);
    }
  else
    {
      cgi_fprintf(cgiOut, "No ioctl code found.\n");
      return FALSE;
    }
}
```

```
      switch (deviceChoice)
{
case ICTP_IO_LCD:
  switch (ioctlChoice)
    {
    case LCD_SELECT:
    case LCD_DESELECT:
    case LCD_CLEAR:
    case LCD_HOME:
    cgi_fprintf(cgiOut,"Ioctl code valid for LCD %d\n",
ioctlChoice);
      break;
    default:
    cgi_fprintf(cgiOut,"Ioctl code not valid for LCD %d\n",
ioctlChoice);
      return FALSE;
    }
  break;
default:
  cgi_fprintf(cgiOut,
"Device does not support any ioctl calls!\n");
  break;
}
    }
  return ICTP_IO_SUCCESS;
}

char *LED_BitNames[] = {
  "b8","b7","b6","b5","b4","b3","b2","b1",
};

int protSendMsg()
{
  unsigned char mask;
  unsigned char LED_Data;
  unsigned char dataBuffer[128];
  char          LCD_Data[17];
  int           LED_Bits[8];
  int           i,invalid,ledData;
  int           retCode;

  cgi_fprintf(cgiOut,"command code: %d\n",cmdChoice);

  switch (cmdChoice)
    {

    case ICTP_IO_WRITE:

      cgi_fprintf(cgiOut,"Executing Write \n");
```

```
      if (ICTP_IO_write)
{
  switch (deviceChoice)
    {
    case ICTP_IO_LED:
      /*
get the bits, construct the data byte
then call ICTP_IO_write
      */
      retCode = cgiFormCheckboxMultiple("LED_data",
LED_BitNames, 8,
LED_Bits,
&amp;invalid);
      if (retCode == cgiFormSuccess)
{
  mask = 0x80;
  LED_Data = 0;
  for (i=0; (i < 8); i++) {
    if (LED_Bits[i])
      LED_Data |= mask;
    mask = mask >> 1;
    }
  cgi_fprintf(cgiOut, "LED data: %2x\n", LED_Data);
  retCode = ICTP_IO_write(ICTP_IO_LED,
&amp;LED_Data,1);
  break;
}

      retCode = cgiFormInteger("intData", &amp;ledData, 0);
      if (retCode == cgiFormSuccess)
{
  cgi_fprintf(cgiOut, "Integer Data: %d\n",ledData);
  LED_Data = ledData;
  retCode = ICTP_IO_write(ICTP_IO_LED,&amp;LED_Data,1);
  reportError(retCode);
}
      else
{
  /* if none are checked */
  cgi_fprintf(cgiOut,
"Nothing checked, data set to zero\n",ledData);
  LED_Data = 0;
  retCode = ICTP_IO_write(ICTP_IO_LED,&amp;LED_Data,1);
  reportError(retCode);
}
      break;

    case ICTP_IO_LCD:
retCode = cgiFormStringNoNewlines("LCD_Data", LCD_Data, 17);
      switch (retCode)
```

```
{
case cgiFormSuccess:
  cgi_fprintf(cgiOut,
"LCD_Data fetched: %s\n",LCD_Data);
  retCode =
ICTP_IO_write(ICTP_IO_LCD,LCD_Data,strlen(LCD_Data));
  break;
case cgiFormTruncated:
  cgi_fprintf(cgiOut,
"LCD_Data fetched, result code: cgiFormTruncated\n");
  retCode = -11;
  break;
case cgiFormEmpty:
  cgi_fprintf(cgiOut,
"LCD_Data fetched, result code: cgiFormEmpty\n");
  retCode = -11;
  break;
case cgiFormNotFound:
  cgi_fprintf(cgiOut,
"LCD_Data fetched, result code: cgiFormNotFound\n");
  retCode = -11;
  break;
case cgiFormMemory:
  cgi_fprintf(cgiOut,
"LCD_Data fetched, result code: cgiFormMemory\n");
  retCode = -11;
  break;
default:
  cgi_fprintf(cgiOut,
"LCD_Data fetched,
unexpected result code: %s\n",
retCode);
  retCode = -11;
  break;
}
      break;
    }
}
      break;

    case ICTP_IO_READ:
      cgi_fprintf(cgiOut,"Executing read \n");

      retCode = ICTP_IO_read(deviceChoice,dataBuffer,dataSize);
      reportError(retCode);
      if (retCode == ICTP_IO_SUCCESS)
{
  cgi_fprintf(cgiOut, "ICTP_IO Data: ");
  for (i=0;i<dataSize;i++)
    cgi_fprintf(cgiOut, "0x%02x ",dataBuffer[i]);
```

```
  cgi_fprintf(cgiOut,"\n");
}
      return retCode;

    case ICTP_IO_IOCTL:
      cgi_fprintf(cgiOut,"Executing ioctl\n");
      if (deviceChoice == ICTP_IO_LCD)
retCode = ICTP_IO_ioctl(deviceChoice,ioctlChoice);
      break;
    }
  return retCode;
}

void reportError(int retCode)
{
  if (ICTP_IO_msgType == ICTP_IO_HTML)
    {
      fprintf(cgiOut, "<h1><center><font size = 7>\n");
      if (retCode == ICTP_IO_SUCCESS)
{
  fprintf(cgiOut,"ICTP_IO Protocol Success<BR>\n");
  fprintf(cgiOut, "Error Code: %03d<BR>\n",
  retCode);
}
      else
{
  fprintf(cgiOut, "ICTP_IO Protocol Error:<BR>\n");
  fprintf(cgiOut, "Error Code: %d<BR>\n",
  retCode);
}
      fprintf(cgiOut,"</font></center></h1>\n");
    }
  else
    {
      if (retCode == ICTP_IO_SUCCESS)
fprintf(cgiOut,"ICTP_IO Protocol Success\n");
      else
fprintf(cgiOut, "ICTP_IO Protocol Error! Error Code: %d\n",
retCode);
    }
}

void cgi_fprintf(FILE *stream, char * fmt, ...)
{
  char *outString;
  char *inPtr,*outPtr;
  int  i;

  va_list ap;
  va_start(ap,fmt);
```

```
  switch (ICTP_IO_msgType)
    {
    case ICTP_IO_TEXT:
      vfprintf(stream,fmt,ap);
      break;

    case ICTP_IO_HTML:
      /*
replace \n with <BR>
      */
      outString = (char *) malloc(strlen(fmt)+3*countCRs(fmt)+1);
      outPtr = outString;
      inPtr = fmt;
      while (*inPtr != '\0')
if (*inPtr != '\n')
  *outPtr++ = *inPtr++;
else
  {
    inPtr++;
    *outPtr++='<';
    *outPtr++='B';
    *outPtr++='R';
    *outPtr++='>';
  }

      *outPtr='\0';

      vfprintf(cgiOut,outString,ap);
      fflush(stream);
      va_end(ap);
    }
}

int countCRs(char *inString)
{
  /*
    count no of \n
  */
  int  i,count;
  char *strPtr;
  count = 0;
  strPtr = inString;
  for (i=0;i<strlen(inString);i++)
    if (*strPtr++ == '\n')
      count++;

  return count;
}
```

7.4.3 The full source code of the Complex Calculator

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

/**
 * ComplexCalcUI_Stage9.java
 *
 *
 * Created: Mon Aug 21 21:19:31 2000
 * Stage 7
 * @author Ulrich Raich
 * @version 0.1
 */

public class ComplexCalcUI_Stage9 extends JApplet {
    private JTextField         realPartText;
    private JTextField         imagPartText;
    private JRadioButton           realPartSelector,imagPartSelector;
    boolean                        debug = false;

    public ComplexCalcUI_Stage9 () {

    }

/**
 *  In this stage be put a border around the 2 widgets
 *  Making the box look lowered
 */
    public void init() {
/*
  Create a Panel for real and imaginary part
  text inputs
*/

JPanel             calcPanel;
GridBagLayout      gridBagLayout = new GridBagLayout();
GridBagConstraints gridBagConstraints = new GridBagConstraints();
GridLayout         numberLayout;
GridLayout         numberInputLayout;
JPanel             numberPanel;
Box                inputBox;
JButton[]          numberInputButton;
JButton[]          operatorInputButton;
Box                selectorBox;
JPanel             numberInputPanel;
JPanel             operatorInputPanel;
JLabel             realPartLabel;
```

```
JLabel             imagPartLabel;

BevelBorder        TextBorder;
if(debug)
  System.out.println("Version 9");
/*
  create the widgets
*/
calcPanel = new JPanel(gridBagLayout);
inputBox  = Box.createHorizontalBox();

numberInputLayout = new GridLayout(4,3);
numberInputPanel   = new JPanel(numberInputLayout);
operatorInputPanel = new JPanel(numberInputLayout);

inputBox.add(numberInputPanel);
inputBox.add(operatorInputPanel);

/*
  create the label and the text fields for
  entry of complex numbers
*/
numberPanel = new JPanel();
numberLayout  = new GridLayout(2,2);
numberPanel.setLayout(numberLayout);
realPartLabel = new JLabel("Real Part");
realPartText = new JTextField(15);
realPartText.setText("0.0");
realPartText.setEditable(false);
imagPartLabel = new JLabel("Imaginary Part");
imagPartText = new JTextField(15);
imagPartText.setEditable(false);
imagPartText.setText("0.0");

TextBorder = new BevelBorder(BevelBorder.LOWERED);
numberPanel.setBorder(TextBorder);
/*
  Place the label and the text widget in the box
*/

numberPanel.add(realPartLabel);
numberPanel.add(imagPartLabel);
numberPanel.add(realPartText);
numberPanel.add(imagPartText);

/*
  create the controller containing the action listeners
  and pass it the instances of realPartText,imagPartText...
  which are needed when treating the Action events
*/
```

```
ComplexCalcController complexCalcController =
    new ComplexCalcController(this);

/*
  create a RadioBox for selection into which
          TextField the button input should go
*/
selectorBox  = Box.createVerticalBox();
realPartSelector = new JRadioButton(
"Input real part of the number");
realPartSelector.setActionCommand("SetReal");
realPartSelector.addActionListener(complexCalcController);
realPartSelector.setSelected(true);

imagPartSelector = new JRadioButton(
"Input imaginary part of the number");
imagPartSelector.addActionListener(complexCalcController);
imagPartSelector.setActionCommand("SetImag");

/*
  Group them into a Radio Box
*/
ButtonGroup buttonGroup = new ButtonGroup();
buttonGroup.add(realPartSelector);
buttonGroup.add(imagPartSelector);

selectorBox.add(realPartSelector);
selectorBox.add(imagPartSelector);

numberInputButton = new JButton[12];
for (int i=0;i<10;i++)
    {
numberInputButton[i] = new JButton(Integer.toString(i));
numberInputPanel.add(numberInputButton[i]);
numberInputButton[i].addActionListener(complexCalcController);
    }

numberInputButton[10] = new JButton(".");
numberInputPanel.add(numberInputButton[10]);
numberInputButton[10].addActionListener(complexCalcController);

numberInputButton[11] = new JButton("+/-");
numberInputPanel.add(numberInputButton[11]);
numberInputButton[11].addActionListener(complexCalcController);

operatorInputButton = new JButton[12];

operatorInputButton[0] = new JButton("+");
operatorInputPanel.add(operatorInputButton[0]);
/* activate the thing */
```

```
operatorInputButton[0].addActionListener(complexCalcController);

operatorInputButton[1] = new JButton("-");
operatorInputButton[1].addActionListener(complexCalcController);
operatorInputPanel.add(operatorInputButton[1]);

operatorInputButton[3] = new JButton("*");
operatorInputPanel.add(operatorInputButton[3]);
operatorInputButton[3].addActionListener(complexCalcController);

operatorInputButton[4] = new JButton("/");
operatorInputPanel.add(operatorInputButton[4]);
operatorInputButton[4].addActionListener(complexCalcController);

operatorInputButton[8] = new JButton("=");
operatorInputPanel.add(operatorInputButton[8]);
operatorInputButton[8].addActionListener(complexCalcController);

/* this button is not used yet */
operatorInputButton[9] = new JButton();
operatorInputPanel.add(operatorInputButton[9]);

operatorInputButton[10] = new JButton("Norm");
operatorInputPanel.add(operatorInputButton[10]);
operatorInputButton[10].addActionListener(complexCalcController);

operatorInputButton[10] = new JButton("Clear");
operatorInputPanel.add(operatorInputButton[10]);
operatorInputButton[10].addActionListener(complexCalcController);

/*
  get the proportions right between
          the display and the button part
*/
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 0;
gridBagConstraints.gridwidth = 1;
gridBagConstraints.gridheight = 1;
gridBagConstraints.weightx = 100;
gridBagConstraints.weighty = 15;
gridBagConstraints.fill = GridBagConstraints.BOTH;
gridBagLayout.setConstraints(numberPanel,gridBagConstraints);

gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 1;
gridBagConstraints.gridwidth = 1;
gridBagConstraints.gridheight = 1;
gridBagConstraints.weightx = 100;
gridBagConstraints.weighty = 70;
gridBagLayout.setConstraints(inputBox,gridBagConstraints);
```

```
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 2;
gridBagConstraints.gridwidth = 1;
gridBagConstraints.gridheight = 1;
gridBagConstraints.weightx = 100;
gridBagConstraints.weighty = 15;
gridBagLayout.setConstraints(selectorBox,gridBagConstraints);

calcPanel.add(numberPanel);
calcPanel.add(inputBox);
calcPanel.add(selectorBox);

this.getContentPane().add(calcPanel);
    }

    /**
       * Get the value of realPartText.
       * @return Value of realPartText.
       */
    public JTextField getRealPartText() {return realPartText;}
    /**
       * Get the value of imagPartText.
       * @return Value of imagPartText.
       */
    public JTextField getImagPartText() {return imagPartText;}
    /**
       * Get the value of imagPartText.
       * @return Value of imagPartText.
       */

    public void setRealPartText(String s)
    /**
       * Write the String into imagPartText.
       */
    {
      if(debug)
System.out.println("UI setText text: " + s);
      realPartText.setText(s);
    }

    public void setImagPartText(String s)
    /**
       * Write the String into imagPartText.
       */
    {
imagPartText.setText(s);
    }

  /**
```

```
   * Get the value of debug.
   * @return value of debug.
   */
  public boolean isDebug() {
    return debug;
  }

  /**
   * Set the value of debug.
   * @param v  Value to assign to debug.
   */
  public void setDebug(boolean  v) {
    this.debug = v;
  }

}// ComplexCalcUI_Stage9


import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;
/**
 * ComplexCalcController.java
 *
 *
 * Created: Sat Aug 26 22:17:25 2000
 *
 * @author Ulrich Raich
 * @version 0.1
 */


public class ComplexCalcController implements ActionListener {
    static final int          plus=1;
    static final int          minus=1;
    ComplexCalcUI_Stage9      complexUI;
    int                       operator;
    ComplexModel              model;
    boolean                   debug=false;

    public ComplexCalcController (ComplexCalcUI_Stage9 ui)
    {
/* save the identifier to the View */
complexUI = ui;
model = new ComplexModel();
if(debug)
  System.out.println("ComplexController");
    }
```

```
    /**
     * Get the value of debug.
     * @return value of debug.
     */
    public boolean isDebug() {
      return debug;
    }

    /**
     * Set the value of debug.
     * @param v  Value to assign to debug.
     */
    public void setDebug(boolean  v) {
      this.debug = v;
    }

    private double getRealPartFromUI()

    {
JTextField  rp;
        String      numberString;
double      value;

rp = complexUI.getRealPartText();
numberString = rp.getText();
value = Double.parseDouble(numberString);
System.out.println("real part value: " + value);
return(value);
    }

    private double getImagPartFromUI()
    {
JTextField  ip;
        String      numberString;
double      value;

ip = complexUI.getRealPartText();
numberString = ip.getText();
value = Double.parseDouble(numberString);
if(debug)
  System.out.println("real part value: " + value);
return(value);
    }

    public void actionPerformed(ActionEvent e)
    {
JButton     activatedButton;
String      buttonLabel;
double      value;
byte[]      buttonChars;
```

```
if(debug)
  System.out.println("Class name:" +
e.getSource().getClass().getName());
if (e.getActionCommand().equals("SetReal"))
  {
    if(debug)
      System.out.println("Real part ");
    model.setReal();
    return;
  }
if (e.getActionCommand().equals("SetImag"))
  {
    if(debug)
      System.out.println("Imag part ");
    model.setImag();
    return;
  }

activatedButton = (JButton)e.getSource();
buttonLabel = activatedButton.getText();

if (buttonLabel.equals("+"))
    {
      if(debug)
System.out.println("Add Button");
      model.setOperator(ComplexModel.ADD);
      model.copyNumbers();
      return;
    }
if (buttonLabel.equals("-"))
    {
      if(debug)
System.out.println("Sub Button");
      model.copyNumbers();
      model.setOperator(ComplexModel.SUB);
      return;
    }
if (buttonLabel.equals("*"))
    {
      if(debug)
System.out.println("Mult Button");
      model.copyNumbers();
      model.setOperator(ComplexModel.MUL);
      return;
    }
if (buttonLabel.equals("/"))
    {
      model.setOperator(ComplexModel.DIV);
      model.copyNumbers();
```

```
      if(debug)
System.out.println("Div Button");
      return;
    }
if (buttonLabel.equals("Norm"))
    {
      model.setOperator(ComplexModel.NORM);
      if(debug)
System.out.println("Norm Button");
      model.execute();
      setResult();
      return;
    }
if (buttonLabel.equals("+/-"))
    {
      if(debug)
System.out.println("Change Sign Button");
      model.changeSign();
      setResult();
      return;
    }
if (buttonLabel.equals("="))
    {
      if(debug)
System.out.println("Equals Button");
      model.execute();
      setResult();
      return;
    }
if (buttonLabel.equals("."))
    {
      if(debug)
System.out.println("Dot Button");
      model.setPoint();
      return;
    }
if (buttonLabel.equals("Clear"))
  {
    model.clear();
    setResult();
  }
for (int i=0;i<10;i++)
    if (buttonLabel.equals(Integer.toString(i)))
    {
      if(debug)
{
  System.out.println("Number:" + i);
  System.out.println("Actual result:" +
model.getResult().toString());
}
```

```
      buttonChars = buttonLabel.getBytes();
      if(debug)
System.out.println("Button Byte" +
buttonChars[0]);
      model.addDigit(buttonChars[0]);
      setResult();
    }
    }

  private void setResult()
  {
    Complex result = model.getResult();
    DecimalFormat df = new DecimalFormat("0.0######");
    String resultString=df.format(result.getReal());
    complexUI.setRealPartText(resultString);
    resultString=df.format(result.getImaginary());
    complexUI.setImagPartText(resultString);
  }

  private void clearResult()
  {
    complexUI.setRealPartText("0.0");
    complexUI.setImagPartText("0.0");
  }

}// ComplexCalcController
/**
 * ComplexModel.java
 *
 *
 * Created: Sun Nov 11 17:59:36 2001
 *
 * @author <a href="Ulrich.Raich@cern.ch">Ulrich Raich</a>
 * @version
 */
import java.text.*;

public class ComplexModel {

  public static final int INVALID=0;
  public static final int ADD    =1;
  public static final int SUB    =2;
  public static final int MUL    =3;
  public static final int DIV    =4;
  public static final int NORM   =5;

  Complex firstNumber,result;
  int     operator;
  boolean realPoint,imagPoint;
  boolean real = true;
```

```
  boolean newNumber = true;
  byte[]  realIntPart,imagIntPart;
  byte[]  realFloatPart,imagFloatPart;
  int     realIntIndex,realFloatIndex;
  int     imagIntIndex,imagFloatIndex;
  boolean debug = false;

/**
 * The ComplexModel is a <b>model</b> for the complex
 * calculator. It has properties to save the numbers
 * that are about to be entered and it stores the number
 * entered before and operator. The operator is stored
 * as well.
 */

  public ComplexModel (){
    realIntPart   = new byte[50];
    realFloatPart = new byte[50];

    imagIntPart   = new byte[50];
    imagFloatPart = new byte[50];

    realPoint = imagPoint = false;
    result = new Complex(0.0,0.0);
    firstNumber = new Complex(0.0,0.0);
    clearAll();
    operator = INVALID;
  }
  public ComplexModel (Complex c){
    this();
    result=c;
  }
  public ComplexModel (double r, double i){
    this();
    result.setReal(r);
    result.setImaginary(i);
  }

  public void setOperator(int op)
  {
    if ((operator < ADD) && (operator > DIV))
      return;
    if(debug)
      System.out.println("Operator set to " + op);
    operator = op;
    newNumber = true;
  }

  public int getOperator()
  {
```

```
    return operator;
  }

  /**
   * Get the value of debug.
   * @return value of debug.
   */
  public boolean isDebug() {
    return debug;
  }

  /**
   * Set the value of debug.
   * @param v  Value to assign to debug.
   */
  public void setDebug(boolean  v) {
    this.debug = v;
  }

  public void execute()

  {
    boolean tmpRealImag;
    String  resultString;
    byte[]  tmpInt,tmpFloat;
    int     tmpIntIndex,tmpFloatIndex;
    int i,j;

    tmpInt = new byte[50];
    tmpFloat= new byte[50];

    if ((operator < ADD) || (operator > NORM))
      return;
    result=getResult();
/* assemble byte strings into complex */
    switch(operator) {
    case ADD:
      result=firstNumber.add(result);
      clearFirstNumber();
      if(debug)
System.out.println("Execution result:" +
result.getReal() + " " +
result.getImaginary());
      break;
    case SUB:
      result=firstNumber.sub(result);
      clearFirstNumber();
      if(debug)
System.out.println("Execution result:" +
result.getReal() + " " +
```

```
result.getImaginary());
      break;
    case MUL:
      result=firstNumber.mul(result);
      clearFirstNumber();
      if(debug)
System.out.println("Execution result:" +
result.getReal() + " " +
result.getImaginary());
      break;
    case DIV:
      result=firstNumber.div(result);
      clearFirstNumber();
      if(debug)
System.out.println("Execution result:" +
      result.getReal() + " " +
      result.getImaginary());
      break;
    case NORM:
      result.setReal(result.norm());
      result.setImaginary(0.0);
      clearFirstNumber();
      if(debug)
System.out.println("Execution result:" +
result.getReal() + " " +
result.getImaginary());
      break;
    }

    newNumber = true;
    clearFirstNumber();
    tmpInt = double2IntArray(result.getReal());
    copyByteArray(tmpInt,realIntPart);
    realIntIndex = tmpInt.length;
    if(debug)
      System.out.println("execute: realIntPart: " +
new String(tmpInt));

    tmpFloat = double2FloatArray(result.getReal());
    copyByteArray(tmpFloat,realFloatPart);
    realFloatIndex = tmpFloat.length;
    if(debug)
      System.out.println("execute: realFloatPart: " +
new String(tmpFloat));

    tmpInt = double2IntArray(result.getImaginary());
    copyByteArray(tmpInt,imagIntPart);
    imagIntIndex = tmpInt.length;

    tmpFloat = double2FloatArray(result.getImaginary());
```

```
    copyByteArray(tmpFloat,imagFloatPart);
    imagFloatIndex = tmpFloat.length;


  }

  public void clearFirstNumber()
  {
    firstNumber.setReal(0.0);
    firstNumber.setImaginary(0.0);
    return ;
  }
  public Complex getFirstNumber()
  {
    return firstNumber;
  }
  public Complex getResult()
  {
    return result;
  }

  private void assembleResult()
  {
    double real,imag;
    byte[] intTmp,floatTmp;

    intTmp   = adaptByteArray(realIntPart,  realIntIndex);
    floatTmp = adaptByteArray(realFloatPart,realFloatIndex);
    if (intTmp.length == 0)
      {
if(debug)
  System.out.println("zero length byte array");
return;
      }
    String numberString = new String(intTmp) + "."
+ new String(floatTmp);
    if(debug)
      System.out.println("assembleResult: real numberString: " +
numberString);
    real = Double.parseDouble(numberString);

    intTmp   = adaptByteArray(imagIntPart,  imagIntIndex);
    floatTmp = adaptByteArray(imagFloatPart,imagFloatIndex);

    numberString = new String(intTmp) + "." +
new String(floatTmp);
    imag = Double.parseDouble(numberString);

    if(debug)
      System.out.println("assembleResult: imag numberString: " +
```

```
numberString);
    result = new Complex(real,imag);
  }

  private byte[] adaptByteArray(byte[] unadaptedByteArray,int index)
  {

    byte[] adaptedByteArray;
    if (index==0)
      {
adaptedByteArray   = new byte[1];
adaptedByteArray[0]='0';
return adaptedByteArray;
      }
    else
      {
adaptedByteArray   = new byte[index];
for (int i=0;i<index;i++)
  {
    adaptedByteArray[i] = unadaptedByteArray[i];
    if(debug)
      System.out.println(i +
"Copy char:" +
new String(adaptedByteArray));
  }
      }
    return adaptedByteArray;
  }

  public String[] getResultStrings()
  {
    String[] complexStrings = new String[2];
    byte[] intTmp,floatTmp;

    if(debug)
      System.out.println("realIntIndex : " + realIntIndex);

    intTmp   = adaptByteArray(realIntPart,  realIntIndex);
    floatTmp = adaptByteArray(realFloatPart,realFloatIndex);

    complexStrings[0] = new String(intTmp) + "." + new String(floatTmp);

    intTmp   = adaptByteArray(imagIntPart,  imagIntIndex);
    floatTmp = adaptByteArray(imagFloatPart,imagFloatIndex);

    complexStrings[1] = new String(intTmp) + "."
+ new String(floatTmp);

    if(debug)
      System.out.println("result Strings: " +
```

```
complexStrings[0] + " " +
        complexStrings[1]);
    return complexStrings;
  }

  public void copyNumbers()
  {
    firstNumber = new Complex(result);
    result.setReal(0.0);
    result.setImaginary(0.0);
    clearAll();
  }

  public void clearAll()
  {
    clear();
    real = !real;
    clear();
    real = !real;
  }

  public void clear()
  {
    if (real)
      {
realIntPart[0]   = '0';
realFloatPart[0] = '0';
result.setReal(0.0);
realIntIndex   = 1;
realFloatIndex = 0;
      }
    else
      {
imagIntPart[0]   = '0';
imagFloatPart[0] = '0';
result.setImaginary(0.0);
imagIntIndex   = 1;
imagFloatIndex = 0;
      }
    realPoint = imagPoint = false;
  }

  public void addDigit(byte digit)
  {
    String resultString;
    double newRealPart,newImagPart;
    String integerPartString,floatPartString;
    byte[] intBytes,floatBytes;
    int    intIndex,floatIndex;
    byte[] intPart,floatPart;
```

```
    boolean point;

    if (newNumber) {
      clear();
      newNumber = false;
    }
    if(debug)
      System.out.println("addDigit: " + digit);
    if (real)
      {
intIndex   = realIntIndex;
floatIndex = realFloatIndex;
intPart    = realIntPart;
floatPart  = realFloatPart;
point      = realPoint;
      }
    else
      {
intIndex   = imagIntIndex;
floatIndex = imagFloatIndex;
intPart    = imagIntPart;
floatPart  = imagFloatPart;
point      = imagPoint;
if(debug)
  System.out.println("Imag addDigit");
      }

    if (point)
      {
if (floatIndex > 7)
  return;
floatPart[floatIndex] = digit;
floatIndex++;

floatBytes = new byte[floatIndex];
for (int i=0;i<floatIndex;i++)
  floatBytes[i] = floatPart[i];
resultString = new String(floatBytes);
if(debug)
  System.out.println("New result: " + resultString);

newRealPart = Double.parseDouble(resultString);
if(debug)
  System.out.println(
"new value:" + new Double(newRealPart).toString());
result.setReal(newRealPart);
      }
    else
      {
if (intIndex > 7)
```

```
  return;
if(debug)
  System.out.println("intIndex: " + intIndex);
if ((intIndex == 1)&&(intPart[0]=='0'))
  {
    if (digit == '0')
      return;
    intPart[0] = digit;
  }
else
  {
    intPart[intIndex] = digit;
    intIndex++;
  }

intBytes = new byte[intIndex];
for (int i=0;i<intIndex;i++)
  intBytes[i] = intPart[i];

resultString = new String(intBytes);
if(debug)
  System.out.println("New result: " + resultString);

newRealPart = Double.parseDouble(resultString);
if(debug)
  System.out.println("new value:" + new Double(newRealPart).toString());
result.setReal(newRealPart);

      }
    if (real)
      {
realIntIndex = intIndex;
realFloatIndex= floatIndex;
      }
    else
      {
imagIntIndex = intIndex;
imagFloatIndex= floatIndex;
      }
    assembleResult();
  }

  public void setPoint()
  {
    if (real)
      realPoint = true;
    else
      imagPoint = true;
  }
```

```
  public void setReal()
  {
    real = true;
  }

  public void setImag()
  {
    real = false;
  }

  public void changeSign()
  {
    if (real)
      {
realIntPart = changeSign(realIntPart);
if (realIntPart[0] == '-')
  realIntIndex++;
else
  realIntIndex--;

if(debug)
  {
    System.out.println("After Change Sign " +
       new String(realIntPart) + "." +
       new String(realFloatPart));
    System.out.println("realIntIndex: " + realIntIndex);
  }
      }
    else
      {
imagIntPart = changeSign(imagIntPart);
if (imagIntPart[0] == '-')
  imagIntIndex++;
else
  imagIntIndex--;
      }
    assembleResult();
  }

  public byte[] changeSign(byte[] inArray)
  {
    byte[] outArray = new byte[50];
    if (inArray[0] == '-')
      {
for (int i=0;i<inArray.length-1;i++)
  outArray[i] = inArray[i+1];
return outArray;
      }
    else
      {
```

```
outArray[0] = '-';
for (int i=0;i<inArray.length-1;i++)
  outArray[i+1] = inArray[i];
return outArray;
      }
  }

  public void copyByteArray(byte[] src, byte[] dest)
  {
    for (int i=0;i<src.length;i++)
      dest[i] = src[i];
  }

  public double byteArray2Double(byte[] intArray, byte[] floatArray)
  {
    double result;
    String numberString = new String(intArray) +
                                              "." +
new String(floatArray);

    result = Double.parseDouble(numberString);
    if (debug)
      System.out.println("byteArray2Double: double value " + result);
    return result;
  }


  /**
   * extracts the integer part from a double
   * in form of a character array
   *
   */

  public byte[] double2IntArray(double val)
  {
    DecimalFormat df;
    String        valString;
    byte[]        intArray,tmpAll;
    int           i,length;

    df = new DecimalFormat("0.0######");
    valString=df.format(val);
    /*
      extract the byte array from the String
      should be in the form ii.fff
      with a least 1 i and 1 f
    */

    length = 0;
    tmpAll = valString.getBytes();
```

```
    for (i=0;i<tmpAll.length;i++)
      {
/*
  get at the position of the decimal point
*/
if (tmpAll[i] == '.')
  break;
else
  length++;
      }
    intArray = new byte[length];
    for (i=0;i<length;i++)
      intArray[i] = tmpAll[i];
    return intArray;
  }

  public byte[] double2FloatArray(double val)
  {
    DecimalFormat df;
    String        valString;
    byte[]        floatArray,tmpAll;
    int           i,length,dotPos;

    df = new DecimalFormat("0.0######");
    valString=df.format(val);
    /*
      extract the byte array from the String
      should be in the form ii.fff
      with a least 1 i and 1 f
    */

    tmpAll = valString.getBytes();
    for (i=0;i<tmpAll.length;i++)
      {
/*
  get at the position of the decimal point
*/
if (tmpAll[i] == '.')
  break;
      }
    dotPos=i;
    length = tmpAll.length - dotPos - 1;

    floatArray = new byte[length];
    for (i=0;i<length;i++)
      floatArray[i] = tmpAll[dotPos+i+1];
    return floatArray;
  }
}
```