The Abdus Salam
International Centre for Theoretical Physics

United Nations
Educational, Scientific
and Cultural Organization

International Atomic
Energy Agency

310/1779-10

Fourth Workshop on Distributed Laboratory Instrumentation Systems
(30 October - 24 November 2006)

Graphical User Interfaces in Java

Ulrich RAICH
P.S. Division
C.E.R.N.
CH-1211 Geneva
SWITZERLAND

Graphical User Interfaces in Java

U. Raich[†*]

[†] *AB Division CERN*
CH-1211 Geneva 23

Lecture given at the:
Third Workshop on
Distributed Laboratory Instrumentation Systems
Trieste, 22 November — 17 December 2004

LNS

[*]Ulrich.Raich@cern.ch

Abstract

The Workshop on **Distributed Laboratory Systems** intends to show how a small distributed control and acquisition system could be set up. With the arrival of the HTTP protocol and the World Wide Web it seems logical to at least include this technology within the Workshop, if not making it its major subject.

This lecture series of 6 lectures is dedicated to programming the WEB. It will first introduce basic HTML programming providing static WEB pages but very quickly introduce user interaction with WEB pages via **G**raphical **U**ser **I**nterfaces. It will first introduce HTML forms and its associated CGI programs and then go into full blown Java applet programming.

Last but not least, the concept of re-usable software components that can be treated by an interactive GUI builder, the Java Beans, will be introduced.

All these concepts will be demonstrated with example programs that form part of the distributed laboratory system.

Once all the building blocks have been explained a complete distributed application accessing the ICTP HC-11 boards is demonstrated. This system consists of a sophisticated GUI application based on home built beans that accesses the HC-11 hardware through a HttpServer on the TINI which in turn interacts with the HC-11 via a serial command/response protocol.

Keywords: LINUX
PACS numbers: 64.60.Ak, 64.60.Cn, 64.60.Ht, 05.40.+j

Contents

1 Basic HTML programming

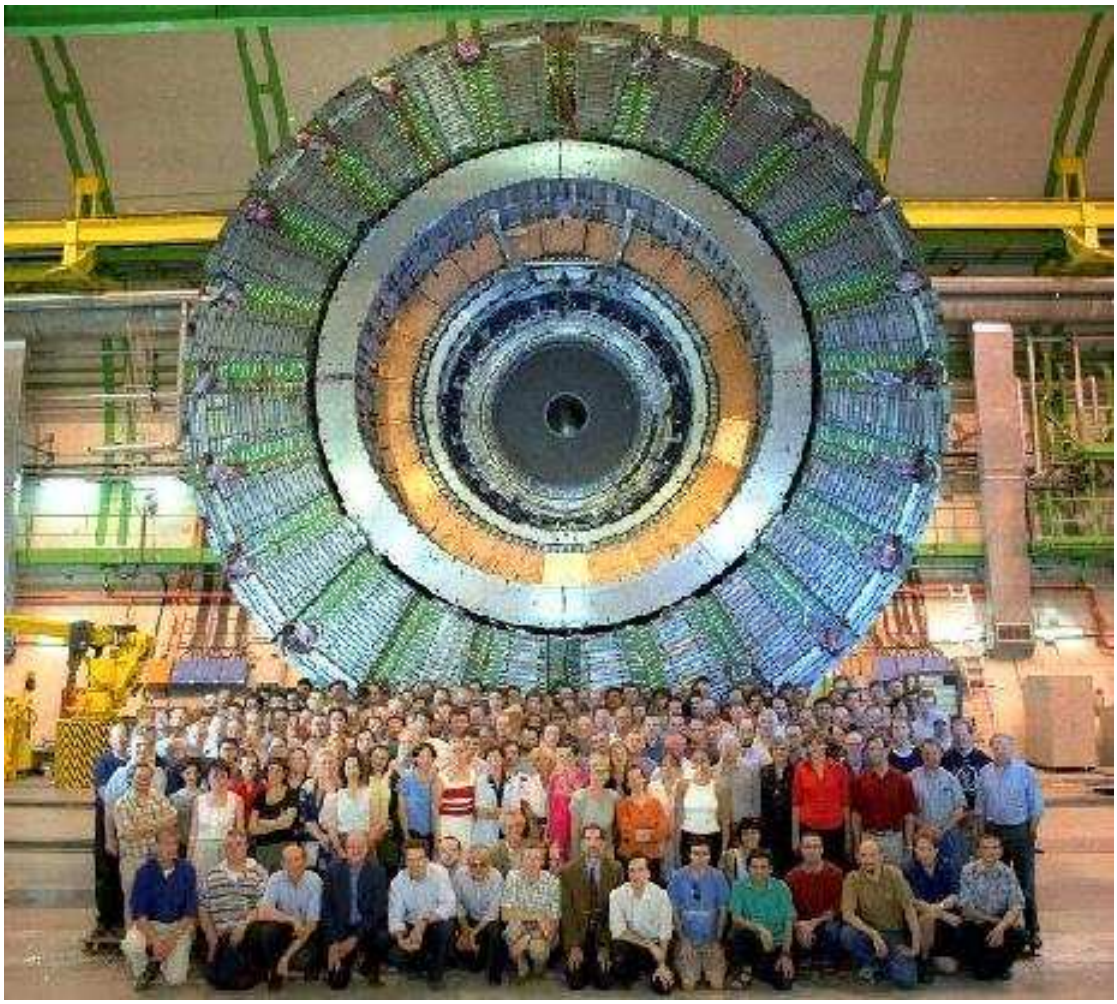
1.1 Introduction

CERN is the European Laboratory for particle physics located in Geneva and one of the largest physics research centers in the world. It possesses several particle accelerators of different types (protons, antiprotons, electrons, heavy ions) providing particle energies of up to 400 GeV. A new proton accelerator, the **L**arge **H**adron **C**ollider (LHC), a ring of 23 km circumference using supraconducting magnets is currently under construction. When going into operation in 2006 it will be the world's highest energy accelerator and a unique research tool on this planet.

Figure 1: Aereal view of CERN's new **L**arge **H**adron **C**ollider (LHC)

In LHC high energy particles will collide in precisely defined interaction points where physics groups will place their detectors. The experiments will be of gigantic dimensions (several stories high and weighting several thousand tons) and groups of some thousand physicists will build the detectors and look after them during the physics runs. Already more than 50% of the world's high energy particle physicists work at CERN. As you may easily understand, the exchange of information within such huge collaborations where subgroups in different universities scattered around the globe work on different subsystems of the detectors is a non-negligible problem. It is therefore not astonishing that the World Wide Web was born at CERN. The original WEB was a means to have detector information stored over many computers in different countries but easily accessible to everyone within the experiment

collaboration.



An experiment at CERN. The image shows *Delphi* which was installed on the now dismantled LEP accelerator. The new experiments foreseen for LHC will even be bigger by an order of magnitude.

Figure 2: Delphi, an Experiment at the LEP Accelerator and its Crew

1.2 WEB client-server model

A major problem for international collaborations is the spread of information across many computers in several labs located in several countries. Nowadays most of the computers are linked through the Internet (btw: the WEB is **NOT** the Internet! the WEB is an application and a protocol running over the Internet) and it seems logical to invent a scheme where a request for information is sent to a server machine, picking up the data in its local file system and sending it in a defined format back to the requester who interprets the result and displays it.

A request is sent in the form of an http (the WEB's protocol running over the Internet) packet often received by an **apache** server which sends back ascii information in the form of an html file which is interpreted by a browser (e.g Netscape or Internet Explorer). This mechanisms works over the internet but also locally using the local file system which can be very interesting for initial testing.

1.3 Static HTML

The first html pages were simple formatted text pages. HTML gives you the possibility to subdivide your text into *headings, paragraphs, lists, tables* etc. and to modify the appearance of text, printing it **bold** or *italic* or you may modify the font.

This is accomplished using so-called HTML tags. <h1> This is a header </h1> is a typical example defining a first level header. <h1> starts the header text </h1> determines its end.

When you start emacs on a new file with `.html` extension, emacs will automatically create an html template of the following form for you:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>
      Workshop on Distributed Laboratory Systems, Trieste 2001
    </title>
  </head>

  <body>
    <h1>
      College on Distributed Laboratory Systems, Trieste 2001
    </h1>
    <hr>
    <address>
      <a href="mailto:Ulrich.Raich@cern.ch">Ulrich Raich
    </a></address>
<!-- Created: Thu Nov  8 23:46:52 CET 2001 -->
<!-- hhmts start -->
<!-- hhmts end -->
  </body>
</html>
```

Even though this is an almost empty page we can still save it as is and load it into netscape:

http://localhost/lecture/HTML/ICTP1.html

and we get what is shown in figure 3.

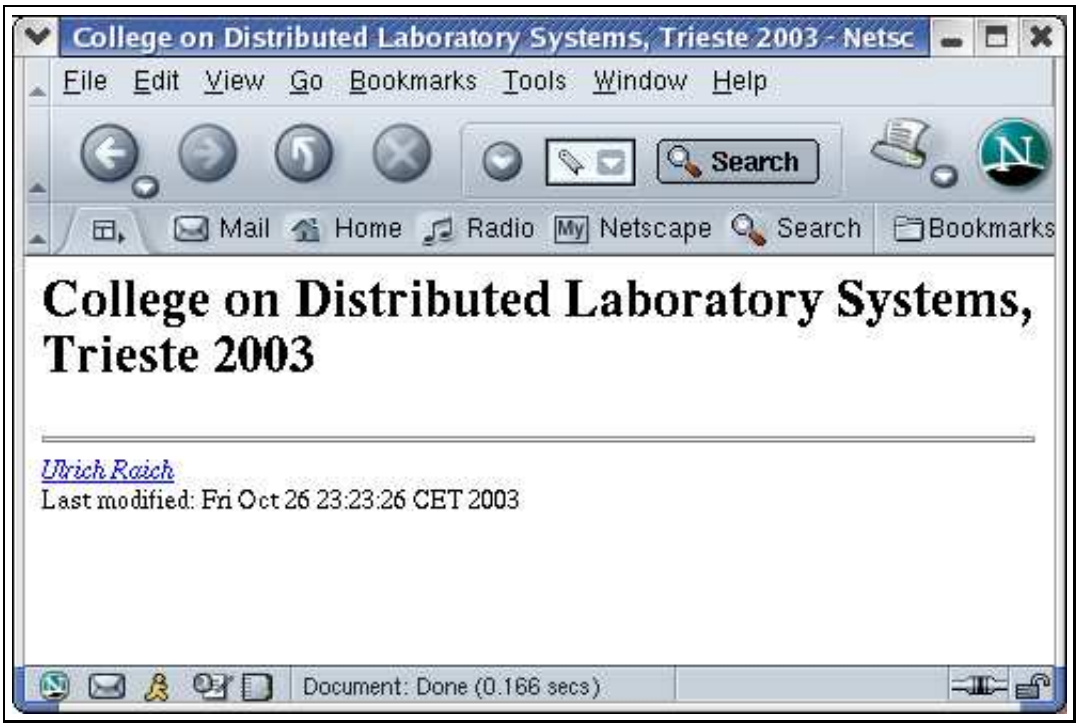


Figure 3: A first and very empty HTML page

Emacs created a lot of different <something></something> **html tags** which define the basic structure of a html document. In the next section we will see many more tags, used for all sorts of structuring, highlighting, layout and the like. For the moment we will simply fill in the empty page with some text.

Here are the beginnings of a *Welcome page* for this Workshop. As you can see, it looks **very** boring but wait... we are going to improve on that! Please note that the screendumps are slightly different. The above one contains the Netscape buttons in order to show that Netscape was used to bring up the page onto the screen, while the one below and all following ones only show the page contents.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>
      Workshop on Distributed Laboratory Systems, Trieste 2001
    </title>
  </head>
  <body>
    <h1>
    Workshop on Distributed Laboratory Systems, Trieste 2001
    </h1>
    In 2001 the Realtime colleges get a new touch. The main language of
```

```
    the college becomes Java, the network gets more attention and a full
    distributed system is demonstrated, including HTML, GUI programming
    with Swing, communication over the network via http, CGI scripts
    and programs and a simple RS-232 protocol in order to communicate
    with a microprocessor system.
    As during the previous years Rinus Verkerk and Abhaya Induruwa
    are the directors of the college.
    We wish you all a lot of fun and hope you don't regret to have
    come to the college.
    Ciao
    <hr>
    <address>
       <a href="mailto:Ulrich.Raich@cern.ch"Ulrich Raich</a>
    </address>
<!-- Created: Thu Nov  8 23:46:52 CET 2001 -->
<!-- hhmts start -->
<!-- hhmts end -->
  </body>
</html>
```

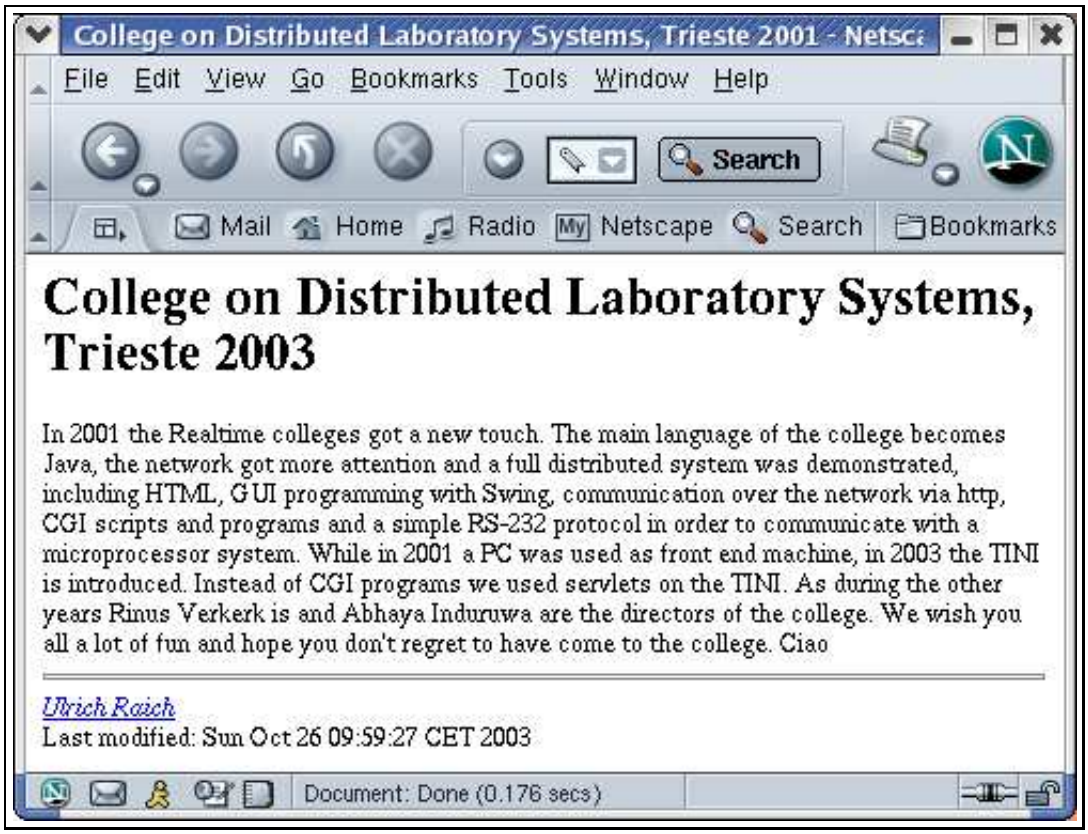


Figure 4: The HTML Page filled with some Text

In order to improve the visual appearance of the page we must first get some structure

into the text which can be done by inserting tags similar to the ones that emacs automatically created for us. Here is a little collection to begin with:

- headings — <h1>..<h6> create a heading of level 1..6;
- paragraph — <p> separates text into paragraphs;
- line break —
- center the text —

In order to modify the appearance of a series of characters we can

- print them bold — <b>
- or italic — <i>
- have them printed like a type writer — <tt>
- change their size — font size = 6
- or their color — <font color = #115577> or <font color = red>
  the color code is: red, green, blue (0000ff is full blue)

```
<h1
  Workshop on Distributed Laboratory Systems, Trieste 2001
</h1
<h2>
  Introduction
</h2>
<p>
In <i><font color=#000088>2001 the Realtime colleges</font></i>
get a new touch.
The main language of
the college becomes <b>Java</b>, the <b>network</b>
gets more attention and a full <b>
distributed system</b> is demonstrated,
including <font color=#880000>HTML,
GUI </font>programming
with <font color=#880000>Swing</font>,
communication over the network via
<font color=#880000>http, CGI </font>scripts
and programs and a simple <font color=#880000>
RS-232</font> protocol in order to communicate
with a microprocessor system.<br>
</pp>
As during the previous years <font size=+4><b>
Rinus Verkerk</b></font> and
<font size=+4><b>Abhaya Induruwa</b></font>
```

```
are the directors of the college.
We wish you all a lot of fun and hope you don't regret to have
come to the college.
</p>
<center>
<font size=22>
<font color=#225588>C</font>
<font color=#ff0022>i</font>
<font color=#007722>a</font>
<font color=#ffff00>o</font>
</font>
</center>
<font size=22>
 <font color=#225588>C</font>
 <font color=#ff0022>i</font>
 <font color=#007722>a</font>
 <font color=#ffff00>o</font>
</center>
```

Of course we are curious to see the improvements in the page. For this reason let us have a look at the current page in figure 5 as displayed with Netscape.

We can do better still:

- At first we will create a table with 2 columns using the <table> tag,
- then we put (aligned to the center) *table headings* (the names of our 2 directors!) enclosed in <th> tags,
- after this, the table body follows with a new row defined by <tr>
- and new table elements with <td>
- Inside those table elements we put the corresponding images.

The final code is shown below:

```
<table align = "center" cols=2 border="0" cellpadding="10"
       cellspacing="5">
 <thead><th align=center>Rinus</th>
 <th align=center>Abhaya</th>
 </thead>
  <tbody>
  <tr><td align=center>
    <img src= http://localhost/ICTP/lecture/images/rinus.jpg>
  </img>
    </td><td align=center>
    <img src=http://localhost/ICTP/lecture/images/abhaya2.jpg>
  </img>
```

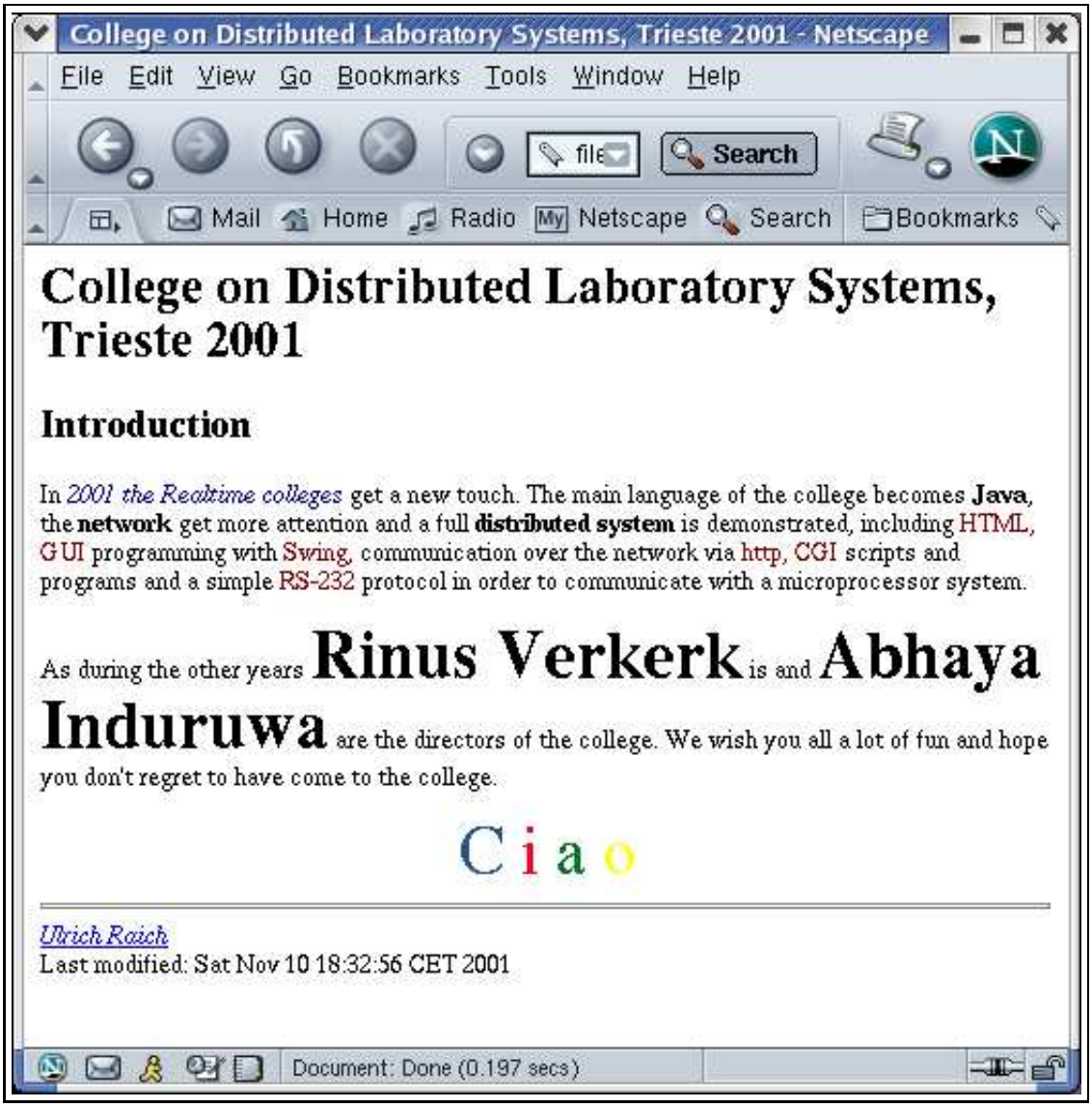


Figure 5: Improving the HTML page with style and color

```
    </td></tr>
 </tbody>
</table>
```

Putting the tag microprocessor around the text *microprocessor* we create a so-called *anchor* (this is why the tag is named "a") which will create a *link* to a **U**nified **R**esource **L**ocator (URL) that can be activated. As we can see, the word *microprocessor* turns blue and gets underlined, indicating it can be activated. Clicking the mouse pointer on it will show an image of our HC-11 microprocessor system. If, instead of specifying the URL of a jpg image you specify the URL of another html page within the a tag, then this html page will be loaded and displayed by your browser. Like this the user can jump from one page to the next.

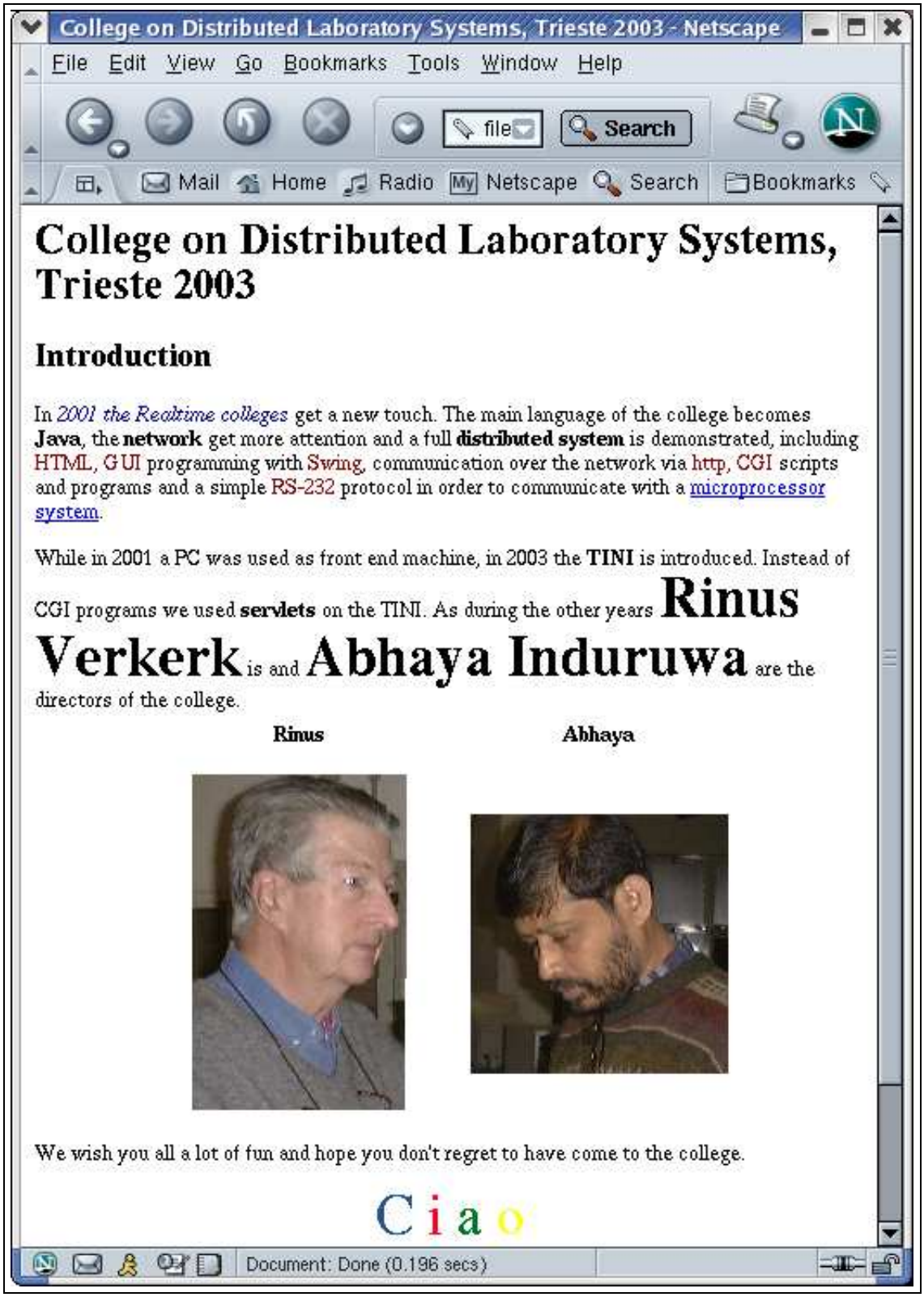


Figure 6: The final Welcome Page for the Workshop

Of course this page does not conform to the latest standards of WEB design but it shows the use of a maximum of tags on a small portion of text. And we can do even quite a bit more:

1.4 Interactive HTML

When surfing the WEB and dowloading software or when buying things on the Internet you have surely already seen some *forms* which you may fill in and submit... Well, where do you actually send them to? And what happens, once you have sent them?

These forms contain text input panels, buttons, pulldown selection menus, and the like. In order to create a button we must first tell the browser that we are going to use a form and then declare the button itself:
`<form method="POST" action="http://localhost/cgi-bin/showString.cgi">`.

The *form* tag declares that now interactive elements are permitted and that the settings will be sent using the **POST** method. There are several ways of treating post requests. The traditional way was the use of **C**ommon **G**ateway **I**nterface scripts which we will use in this chapter. Another way is the use of servlets extending the http servers' capabilities which will be treated in the last chapter of this lecture series. Let us consider that the post requests will be sent to a CGI program on localhost, namely `cgi-bin/showString.cgi`. Each WEB server uses a directory tree for storage of html pages, images, video clips, audio clips, and cgi programs which, in the case of the **apache** WEB server on RedHat Linux, is /var/www, where you will find the subdirectories *html* and *cgi-bin* and maybe some more.

HTML uses 2 different methods for sending user created information to the WEB server:

- the **GET**
- and the **POST** action.

GET passes the information in evironment variables (and here we will stop mentioning this method!) while with **POST** the WEB server will start the CGI program and send the user created information as a text string on standard input to it.

Time for an example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>>Becoming Interactive</title>
  </head>
  <body>
    <h1>>Becoming Interactive</h1>
    <h3>>Buttons, Menus and the like </h3>
    <p>
    Up to now all pages we got were purely static. I mean,
    all pages were prepared before and simply displayed
    <i> as is</i>. Now we want to figure out
    how we can make pages interactive: The page we will
    see now will be created on the fly
    depending on settings provided by the user.
```

```
    This is accomplished with so-called
    <b> forms</b>.
    <hr>
    <form method="POST" action="http://PCUli/cgi-bin/showString.cgi">
    So, let us create a
    <input type="checkbox" name="LED_data" value="b1">;
    checkbox button (bit 1) and then
    <input type="checkbox" name="LED_data" value="b2">
    another one (bit 2)<br>
    </form>
    <hr>
    <address
    >><a href="mailto:Ulrich.Raich">>Ulrich Raich</a>
    </address>
<!-- Created: Sun Nov 11 13:50:25 CET 2001 -->
<!-- hhmts start -->
Last modified: Sun Nov 11 15:09:33 CET 2001
<!-- hhmts end -->
  </body>
</html>
```

And here is the result:

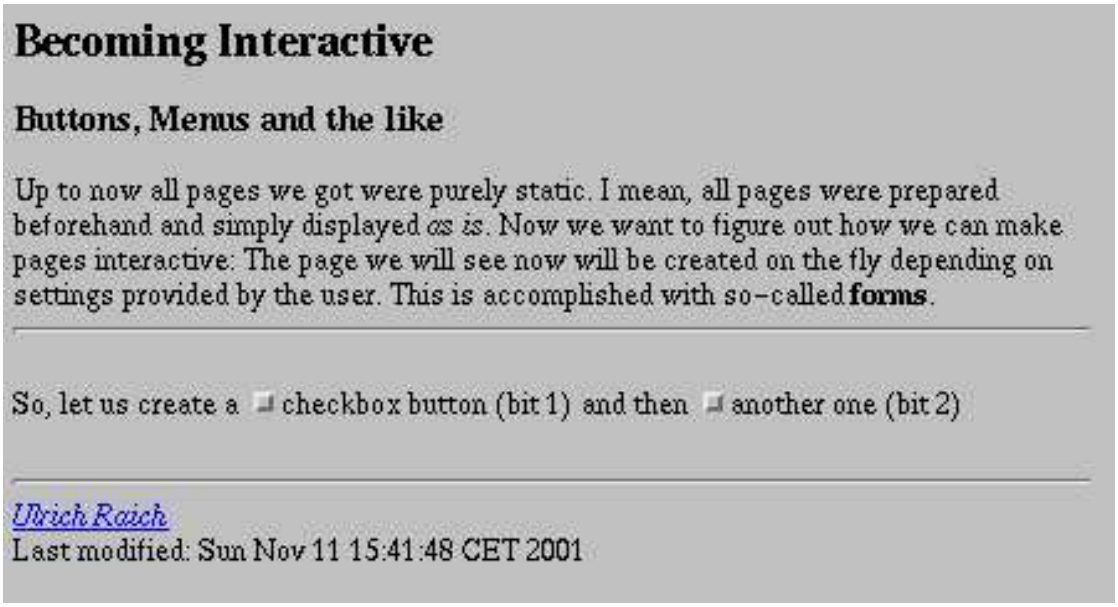


Figure 7: Interactive HTML, a first try

So far, so good. Now we have 2 buttons on the screen, we can activate those buttons but that's it. We still cannot send the information to the CGI program. This will be accomplished by adding a **submit** button to the page:
<input type="submit" name="submit1" value="Submit Request">

Writing the really tiny C program named showString.c, compiling it to showString.cgi and installing it in the directory /var/www/cgi-bin will tell us what is actually going on:

```
/* showString.c */
#include <stdio.h>
 int main()  {
  char lineBuffer[1000];
  int  retCode;
  printf("Content-type:text/html\n\n");
   while (fgets(lineBuffer,999,stdin) != NULL)
     printf("%s\n",lineBuffer);}
```

As you can see, this program simply reads the standard input and it writes the characters it receives on standard input back to the standard output. The result, when the two checkbox buttons are pushed, is the following:

`LED_data=b1&LED_data=b2&submit1=Submit+Request`.

Now it also becomes clear what the parameters to the input tag mean: the *type* describes the type of the GUI element and the *name* and *value* create "name/value" pairs that are sent in the form: name=value e.g. LED_data=b1 to the cgi program. The name value pairs are separated by an ampersand. All the cgi program needs to do is picking up these name/value pairs, interpreting them and executing whatever is appropriate.

Here are some more tags including an example which enable us to create a nice interactive little WEB page. Please note that I also modified the name of the CGI program! A complete html example is also given.

- **<input type="checkbox" name="LED_data" value="b2" checked>**

  which creates the checkbox toggle button. If you mention *checked* the button will be *on* by default.

- **<input type="radio" name="command" value="read"checked>**

  creates a radio button with a name/value pair of "command=read" which by default is off. When you have several buttons of this type only a single one can be active (you automatically deactivate the previously selected button when selecting a new one).

- **<input type="text" name="LCD_data" maxlength=16 value="! Hi all !">**
  creates a text input widget of maximum 16 char with a default text:
  "! Hi all !"

- **<input type="reset" value="Reset Request">** resets all input elements to their default values.

The promissed example then looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Becoming Interactive</title>
  </head>
  <body>
    <h1>Becoming Interactive</h1>
    <h2>Buttons, Menus and the like </h2>
    <p>
    Up to now all pages we got were purely static. I mean,
    all pages were prepared beforehand and simply displayed
    <i>as is</i>. Now we want to figure out how we can
    make pages interactive: The page we will see now will
    be created on the fly depending on settings provided
    by the user. This is accomplished with so-called
    <b>forms</b>.
    <hr>
    <form method="POST"
    action="http://tini/servlet/HC11Servlet">
    So, let us create a
    <input type="checkbox" name="LED_data" value="b1">
    checkbox button (bit 1) and then
    <input type="checkbox" name="LED_data" value="b2">
    another one (bit 2)<br>
    or some radiobox
    <input type="radio" name="command" value="read"> read
    <input type="radio" name="command" value="write" checked>
    write<br>
    or we can create some
    <input type = "text" name ="LCD_Data"
    MAXLENGTH=16 VALUE="! Hi all !">
    and let;s say a <select name="device">
    <option value="LCD"> LCD </option>
    <option selected value="LEDs"> LEDs </option>
    <option value="Switches"> Switches </option>
    </select>
    <br><br><br><br><br>
    and the 2 more buttons for resetting the values to default
    and for submit:<br>
    <input type="submit"
    name="submit1" value="Submit Request">
    <input type="reset" value="Reset Request">
    </form>
    <hr>
    <address><a href="Ulrich.Raich@cern.ch">
    Ulrich Raich</a></address>
```

```
<!-- Created: Sun Nov 11 13:50:25 CET 2001 -->
<!-- hhmts start -->
Last modified: Sun Nov 11 16:53:04 CET 2001
<!-- hhmts end -->
  </body>
</html>
```

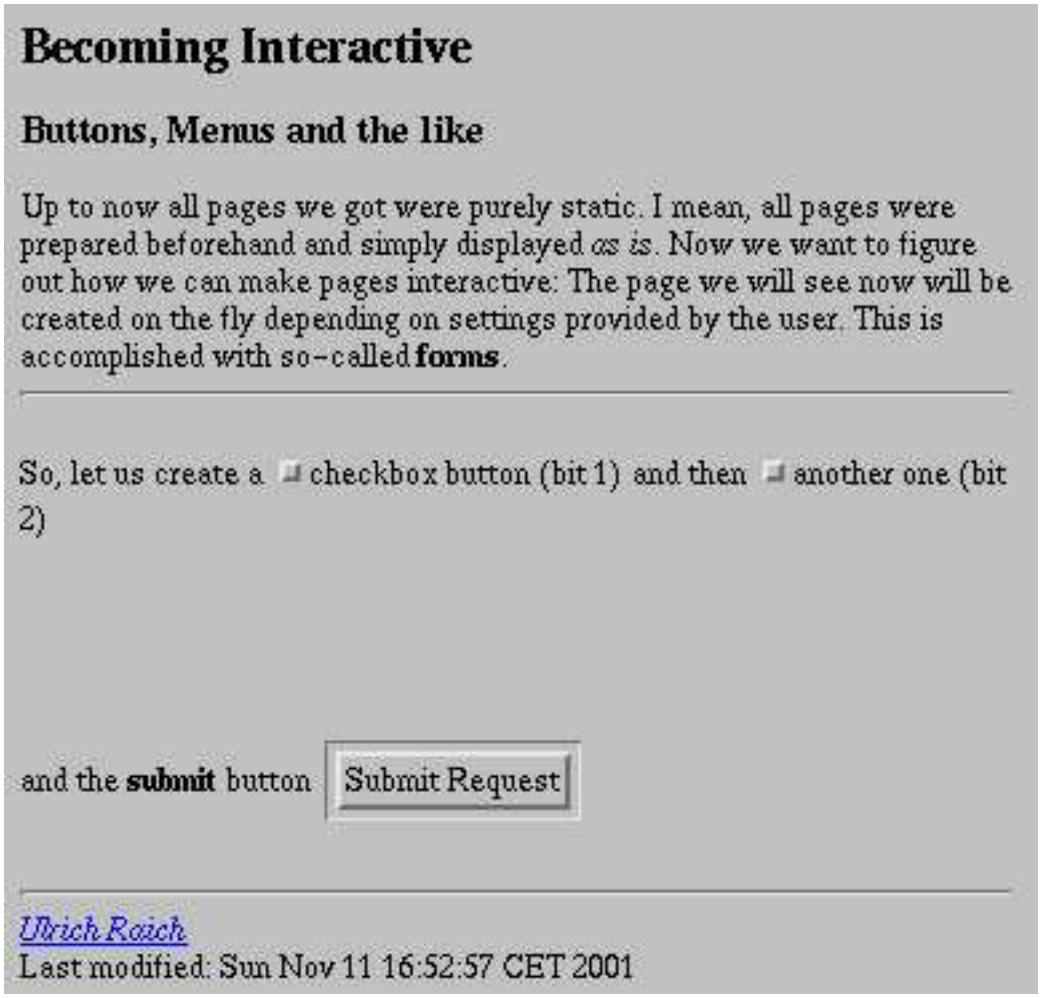


Figure 8: Finished interactive WEB page

This finished interactive WEB page sends data that correspond to settings on the ICTP IO board. "LED_data=b1", the text that will be sent to *tini/servlet/HC11Servlet* will be interpreted as a command switching bit 1 of the LEDs on. If in addition "command=write" is sent, then this information will be passed onto the ICTP_IO protocol and sent to the HC-11 via a serial protocol. You will see the LED light up.

Finally there is a WEB page which refers to all devices available on the ICTP HC-11 boards. The devices are accessed in the same way as in the example above where only LEDs and LCD screen are treated. For more details on the HC-11 boards, see the next chapter.

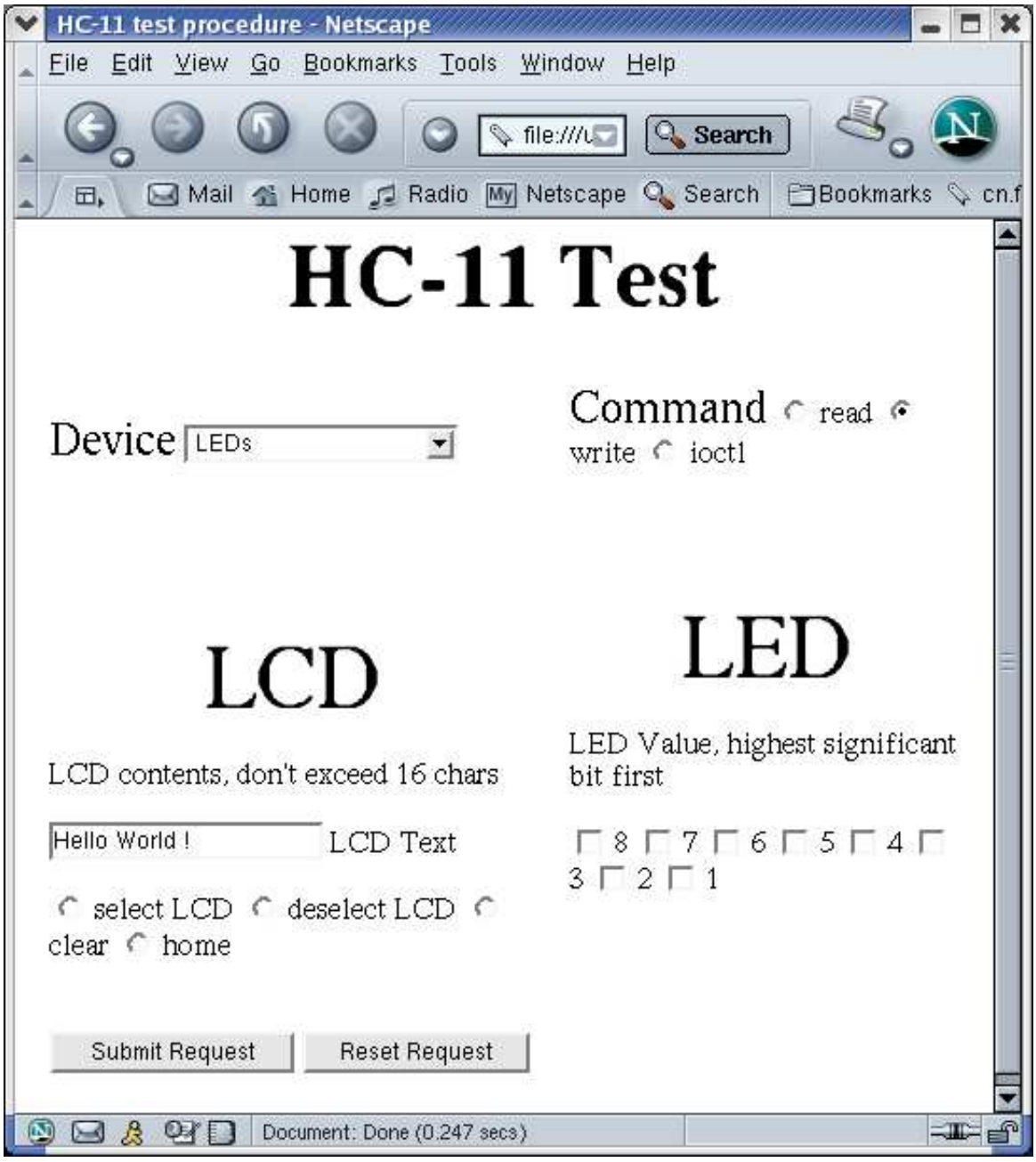


Figure 9: Accessing all HC-11 devices via dynamic html

These programs are really interactive!
Of course it is impossible to go through all the features of HTML in just one lecture. For this reason we give at least a *link*[1] to a listing of some more HTML tags.

[1] http://werbach.com/barebones/

2 Accessing Hardware

2.1 Legacy equipment

Imagine... You have this oscilloscope that you bought some years ago or this multimeter or whatever other equipment which you would like to use in a measurement setup and you want this to be accessible through the network. How can you interface these devices such that they are visible on the net? Many commercial devices are delivered with some kind of computer interface, mostly RS232 or Centronix or GPIB using a well defined communication protocol. For the ICTP colleges we have designed and built a microprocessor board associated with an I/O board (C.S. Ang) which possesses an RS-232 interface and which uses a parallel interface in order to drive an LCD panel, an LED panel and a series of switches. A multiplexed ADC is implemented on the microcontroller chip itself. The chip is a Motorola HC-11 microcontroller featuring 128 bytes of RAM, 2 kBytes of EEProm a parallel interface, a serial interface, timer, ADC ... all on-chip. This means that the processor board consists only of the microcontroller itself with very few support electronics. It connects through a flat cable to the I/O board (see figure 10, The microcontroller is mostly hidden by the flat cable.

Let us consider the HC-11 + I/O board as the *commercial* equipment which we want to interface to the internet. In oder to make the device internet ready we need some interface box that has an Ethernet connection on one side and a serial interface on the other and the Tini is capable of implemented exactly this type of box. The Tini can be accessed through TCP/IP and it has a serial line that can be connected to the HC-11.

2.2 Software in the HC-11

In order to be able to communicate with the HC-11 a communications protocol must be defined that allows access to HC-11 I/O devices. On the HC-11 a server program implementing this protocol has been installed. Before being able to download and debug the server however a simple debug monitor is needed which is started on reset. The HC-11 EEPROM has therefore been subdivided into 2 sections of 1 kBytes each where the first one (at address F800) is used by the protocol server and the second one (address FC00) by the debug monitor. Both programs have been implemented in HC-11 assembly language.

The monitor has the following commands:

- gxxxx goto xxxx, start program at address xxxx
- s single step
- c continue program from the *current address*
- p program the EEPROM, used for downloding of programs
- v verify the EEPROM against the contents of a HC11 binary file (s19)
- r print register contents
- m examine and modify memory
- b set breakpoint (only a single breakpoint is possible)
- d display breakpoint
- u uninstall breakpoint

This monitor will normally only be used by people who want to implement a new ICTP_IO protocol server with features deferring from the supplied one. It can be accessed by the TINI

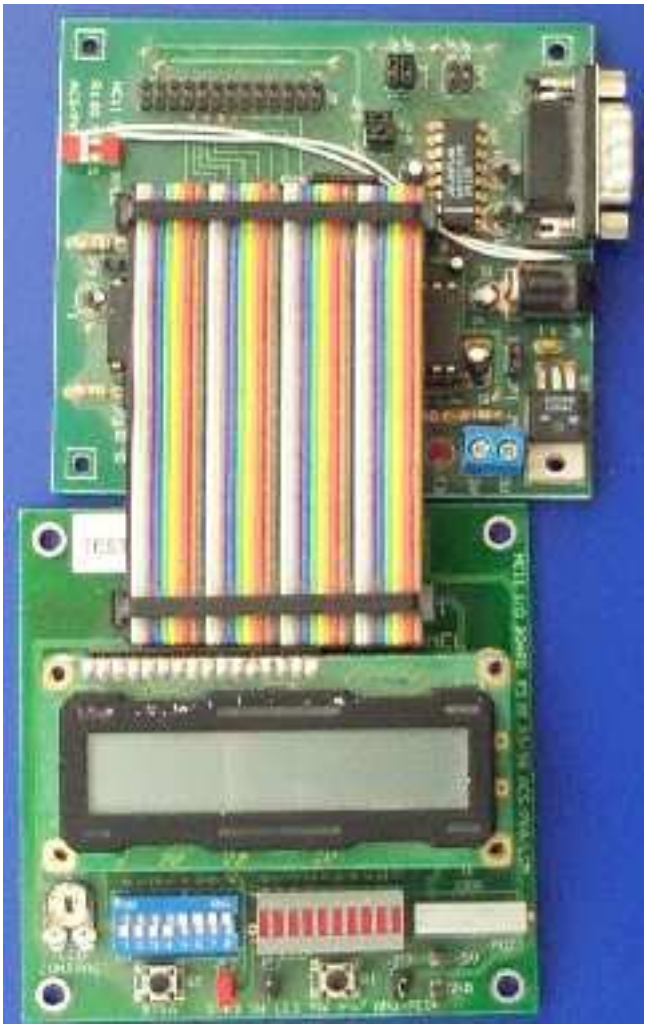


Figure 10: The HC11 microcontroller and I/O board

through the *TiniTerm* terminal emulator, a program that takes input from the connected telnet session and transfers them to the serial line. Answers coming back from the HC-11 are read off the serial line and printed in the telnet session.

2.3 The ICTP_IO protocol

After reset the HC-11 starts the monitor and the monitor command `gf800` accessed through TiniTerm, transfers control to the protocol server.

The protocol has been given the name ICTP_IO protocol and borrows heavily from the interface of UNIX drivers. Of course different servers are possible treating different devices or treating devices in a different way.

The server provided as an example is capable of reading and writing the devices available on the HC-11 boards. It does this by waiting for messages to be sent to it via the serial line which have the following form:

- a header of 4 bytes, containing, in order, a device number, a command code, an error code and a datasize field specifying how many bytes must be read or witten
- in case of writing: the data must be attached in addition

The server analyses this ICTP_IO message, executes the command specified and send back a reply message of the same form in which the error code and, in case of a read command, the data read from the device are modified.

In order to ease the use of the protocol on the client side a support class named tini_ictp_io has been developed making it easy to create ICTP_IO messages and sending them to the HC-11. A typical client program (I used it to test the support class) looks as follows:

```
/*
 * ICTP_IO_test.java
 *
 * Created on September 1, 2003, 11:03 PM
 */

package tini_ictp_io;
/**
 *
 * @author  uli
 */
public class ICTP_IO_test {


    /** Creates a new instance of ICTP_IO_test */
    public ICTP_IO_test() {

    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        byte LEDdata1 = 0x55;
        byte LEDdata2 = -0x56; // should be 0xaa
        String LCDString = new String("Ca marche!");
        byte[] LCDbyteArray,LCDbyteArrayRead;
        byte[] switchValue = new byte[1];

        TINI_ICTP_IO tiniConn = new TINI_ICTP_IO();
        tiniConn.setDebug(false);

        for (int i=0; i<1;i++) {
```

```
    if (tiniConn.open()!= TINI_ICTP_IO.ICTP_IO_SUCCESS)
    {
        System.out.println("Could not open HC-11 connection, giving up ...");
        System.exit(-1);
    };
    if (tiniConn.getDebug())
        System.out.println("--------------------------------------- LED ---")
    if (tiniConn.write(TINI_ICTP_IO.ICTP_IO_LED,LEDdata1)
     != TINI_ICTP_IO.ICTP_IO_SUCCESS)
        System.out.println("Could not write to  HC-11 connection");

    if (tiniConn.getDebug())
        System.out.println("--------------------------------------- LED ---")
    if (tiniConn.write(TINI_ICTP_IO.ICTP_IO_LED,LEDdata2)
      != TINI_ICTP_IO.ICTP_IO_SUCCESS)
        System.out.println("Could not write to  HC-11 connection");

}

if (tiniConn.getDebug())
    System.out.println("------------------------------------ ioctl LCD ---")


if (tiniConn.ioctl(TINI_ICTP_IO.ICTP_IO_LCD, (byte)TINI_ICTP_IO.ICTP_IO_CLEAR)
 != TINI_ICTP_IO.ICTP_IO_SUCCESS) {
    System.out.println("Error  on ioctl to select LCD");
}
if (tiniConn.ioctl(TINI_ICTP_IO.ICTP_IO_LCD, (byte)TINI_ICTP_IO.ICTP_IO_HOME)
         != TINI_ICTP_IO.ICTP_IO_SUCCESS) {
    System.out.println("Error  on ioctl to select LCD");
}

LCDbyteArray = LCDString.getBytes();
    if (tiniConn.getDebug())
        System.out.println("--------------------------------------- LCD ---")
if (tiniConn.write(TINI_ICTP_IO.ICTP_IO_LCD,LCDbyteArray)
  != TINI_ICTP_IO.ICTP_IO_SUCCESS)
    System.out.println("Could not write to  HC-11 connection");

if (tiniConn.open()!= TINI_ICTP_IO.ICTP_IO_SUCCESS) {
    System.out.println("Could not open HC-11 connection, giving up ...");
    System.exit(-1);
};
tiniConn.setDebug(true);

if (tiniConn.getDebug())
```

```
            System.out.println("------------------------------------ Switches ---");
        if (tiniConn.read(TINI_ICTP_IO.ICTP_IO_SWITCHES,switchValue)
           != TINI_ICTP_IO.ICTP_IO_SUCCESS)
            System.out.println("Could not read from HC-11 connection");
        else
            System.out.println("Switch value : " + switchValue[0]);

        LCDbyteArrayRead = new byte[16];
        for (int i=0;i<LCDbyteArrayRead.length;i++)
            LCDbyteArrayRead[i]=' ';
        System.out.println("Before ictp_io_read LCD");
        if (tiniConn.read(TINI_ICTP_IO.ICTP_IO_LCD,LCDbyteArrayRead)
           != TINI_ICTP_IO.ICTP_IO_SUCCESS)
            System.out.println("Could not read from HC-11 connection");
        else
            System.out.println("LCDtext : " + new String(LCDbyteArrayRead));
        if (tiniConn.close() != TINI_ICTP_IO.ICTP_IO_SUCCESS)
            System.out.println("Could not close  HC-11 connection");

    }

}
```

As you can see the class TINI_ICTP_IO provides methods for opening and closing a serial connection to the HC-11, methods to read and write HC-11 devices and ioctl methods for certain controls of the LCD like *clear* and *home*. The following devices are available and defined as public constants:

- **ICTP_IO_SERVER**: When reading this pseudo device an identifier for the server type is returned. It is possible to run different servers on the same type of hardware. On the HC-11 for example the ADC can be used to read a single value but it may also be used as slow sampling ADC showing the time development of a signal. Such a facility can be implementd in a different server which would have a different ID number.
- **ICTP_IO_SWITCHES**: The dual inline switches on the I/O board
- **ICTP_IO_ADC**: The ADC on the HC-11 chip. Reading this device returns a single byte value corresponding to the current input voltage connected to the ADC.
- **ICTP_IO_BUTTONS**: Reads the current state of two push buttons on the I/O board
- **ICTP_IO_LED**: The LED chain on the I/O board
- **ICTP_IO_LCD**: The LCD display on the I/O board.

These are the methods available:

- **open():** Opens the serial line to the HC-11 and sends an ICTP_IO message with the command code set to *open*. Waits for the return mesaage and passes on the error code.
- **close():** Sends a close message to the HC-11
- **write(byte dev, byte data):** Write a byte to the HC-11 device specified by *device* This call is typically used to write the LED chain.

- **write(byte dev, byte data[]):** Write a series of bytes to the HC-11 device specified by *device* This call is typically used to write the LCD.
- **read(byte dev, byte[] data):** Read data from a device. Any device is readable, even the LEDs or LCD. In case of the HC-11 switches of the ADC data read fron the device are returned otherwise it will be the data last written to the device.
- **ioctl(byte dev, byte ioctCmd):** The LCD has additional features that allow the user to clean the entire LCD display with a single command or to put the cursor back to he beginning of the line (*home* the cursor). These are ioctl command of which the following are available:

  - **ICTP_IO_SELECT**: The HC-11 multiplexes the data lines between the LCD and the LEDs. ICTP_IO_SELECT selects the LCD.
  - **ICTP_IO_DESELECT**: Same as above but the LEDs are selected. When sending a wite command to LCD or LEDs this selection is down automatically by the server.
  - **ICTP_IO_CLEAR**: Clears the LCD display
  - **ICTP_IO_HOME**: Put the LCD cursor to home position

The TINI_ICTP_IO class hides the protocol itself as well as all the difficulties connected to the serial interface. You may want to check how exactly this is done by looking at the source code of the class provided in the appendix of these notes.

2.4 Accessing the TINI

Now that we know how to access the HC-11 though a class on the TINI we must figure out how to access the TINI over the network. Naturally there are different solutions like writing dedicated programs using sockets. The easiest way accessing the internet and the most commonly used one is *browsing* WEB browsing. We have already seen how to write WEB pages that can access dynamic information. In order to be able to follow this path, we would need a WEB server running on the TINI. A WEB server is a server program understanding the http protocol and such a server including extensions to allow creation of dynamic html pages through so-called **servlets** is available on the TINI. Each WEB server expects html files in a predefined directory on its file system, */docs* in the case of TINI. In order to check TINI access we take one of the html files described in the previous chapters and put it into the TINI's /docs directory. Starting a WEB browser on the PC and entering *http://mytini/mypage.html* as URL will take use to our html file, If no filename is given we are directed to the root of the TINI's html tree, a file named *index.html*. None of the pages we saw in the previous chapter however is capable of creating WEB pages dynamically which is needed if we want to display the state of our HC-11 hardware. Traditionally the creation of dynamic page content was accomplished with so-called **C**ommon **G**ateway **I**interface (CGI) scripts. A even more powerful solution is the use of servlets, pieces of Java code extending the server. Servlets must be compiled and linked into the server before installation. The next task will therefore be the addition of a HC11Servlet which will do the access to the HC-11 through the TINI_ICTP_IO class and create dynamically a html page displayin the result.

2.5 Java Servlets

A servlet extends the Http server functionality. In the chapter on applets we will see that it does this in a way which is very similar to the applet increasing the functionality of a WEB browser on the client side.

Servlets are normal Java code which create HTML code and send it back to the browser. They have a life cycle which is similar to that of an applet. A servlet has an init method, a service method and a destroy method. The init method is executed when the servlet is first loaded. Servlets are not "run" in the same sense as Java applets or applications, they extend server functionality. In order to access a servlet it must be installed in the hosting server and a servlet service request must be made via a client request.

Http servlets are accessed by the user entering a URL or by an HTML form action like the one explained in the previous chapter. For example: If you enter the URL: http://tini/servlet/SnoopServ then the SnoopServlet is requested to dynamically create an HTML page and send it to the browser for interpretation. If the submit button on the hc11.html page is pressed then a http request will be sent to the HC11Servlet which will execute a ICTP_IO command: read, write, ioctl on the HC-11 and return the result in form of an html page to the browser.

Since servlets that use the http protocol are very common an http specific helper class HttpServlet is provided. Http defines a set of text based requests called http methods. These methods include

- GET
- POST
- PUT
- DELET
- TRACE
- CONNECT
- OPTIONS

Since we are only using the post method, we will concentrate on this. The http message contains information about the the type of request, the requestor, the host, the browser etc. which can be consulted by the servlet. Before going any further let's have a look at an example. This is a code snippet that was extracted from the SnoopServlet which is provided with the TiniHttpServer bye Smart Software Consulting (www.smartsc.com).

```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class SnoopServlet
extends HttpServlet
{
public void doPost( HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
doGet( req, res);
}

public void doGet( HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
PrintWriter out;

res.setContentType( "text/html");

int i;

out = res.getWriter();


out.println( "<html>");
out.println( "<head><title>Snoop Servlet</title></head>");
out.println( "<body>");
out.println( "<h1>Request information:</h1>");
out.println( "<pre>");

print( out, "Request method", req.getMethod());
print( out, "Request URI", req.getRequestURI());
print( out, "Request protocol", req.getProtocol());
print( out, "Servlet path", req.getServletPath());
print( out, "Path info", req.getPathInfo());
print( out, "Path translated", req.getPathTranslated());
print( out, "Query string", req.getQueryString());
print( out, "Content length", req.getContentLength());
print( out, "Content type", req.getContentType());
print( out, "Server name", req.getServerName());
print( out, "Server port", req.getServerPort());
print( out, "Remote user", req.getRemoteUser());
print( out, "Remote address", req.getRemoteAddr());
print( out, "Remote host", req.getRemoteHost());
print( out, "Authorization scheme", req.getAuthType());

out.println( "</pre>");
enum = req.getHeaderNames();
if( enum.hasMoreElements())
```

```
{
out.println( "<h1>Request headers:</h1>");
out.println( "<pre>");
while( enum.hasMoreElements())
{
String name = (String)enum.nextElement();
out.println( " " + name + ": " + req.getHeader(name) );
}
out.println( "</pre>");
}

enum = req.getParameterNames();
if( enum.hasMoreElements())
{
out.println( "<h1>Servlet parameters (Single Value style):</h1>");
out.println( "<pre>");
while( enum.hasMoreElements())
{
String name = (String)enum.nextElement();
out.println( " " + name + " = " + req.getParameter(name) );
}
out.println( "</pre>");
}

enum = req.getParameterNames();
if( enum.hasMoreElements())
{
out.println( "<h1>Servlet parameters (Multiple Value style):</h1>");
out.println( "<pre>");
while( enum.hasMoreElements())
{
String   name = (String)enum.nextElement();
String[] vals = req.getParameterValues( name);
// out.println( " " + name + " = " + req.getParameter(name) );

if (vals != null)
{
out.print( "<b> " + name + " = </b>");
out.println( vals[0]);
for( i = 1; i<vals.length; i++)
out.println( "           " + vals[i]);
}
out.println( "<p>");
}
out.println("</pre>");
}
```

```
out.println("</body></html>");
}
```

As we can see this servlet does not directly implement the init, service and delete methods mentionned above. These methods are implemented in the HpptServlet code. However we have the possibility to override the doGet and doPost methods which, as you might expect, are the methods that treat *get* and *post* http requests. A HttpServletRequest object is passed into the method and from this object all sorts of information on the request can be retrieved. req.getMethod will tell us which http method has been used and should be *POST* in our case. As you can see we can also check from which browser the request comes, on which machine the browser is running etc.

We also see how the servlet dynamically creates the html page and sends it via the *Writer* out back to the browser. The other interesting point in this example is the way the name/value pairs can be found. In order to select the LED the following code was supplied in the html file:

```
<select name="device">
<option selected value="LEDs"> LEDs </option>
```

which was converted by the browser to a url-encoded string of this kind:
`device=LED&LED_data=b2&submit1=Submit+Request`.

The name/value pairs can be found back from *enum = req.getParameterNames();* and *vals = req.getParameterValues(name);* All that needs to be done is the interpretation of the name/value pairs in order to extract

- the device number
- the command code
- the data values

The following extract from the HC11Servlet contains all the Strings that must be interpreted.

```
/*
 * strings that are needed for comparison
 */
private static final String ICTP_IO_DEVICE = new String("device");
private static final String ICTP_IO_LED    = new String("LEDs");
private static final String ICTP_IO_LED_DATA = new String("LED_data");
private static final String ICTP_IO_LED_BIT1 = new String("b1");
private static final String ICTP_IO_LED_BIT2 = new String("b2");
private static final String ICTP_IO_LED_BIT3 = new String("b3");
private static final String ICTP_IO_LED_BIT4 = new String("b4");
private static final String ICTP_IO_LED_BIT5 = new String("b5");
private static final String ICTP_IO_LED_BIT6 = new String("b6");
```

```
private static final String ICTP_IO_LED_BIT7 = new String("b7");
private static final String ICTP_IO_LED_BIT8 = new String("b8");
private static final String ICTP_IO_LED_NUM_DATA = new String("intData");
private static final String ICTP_IO_LCD      = new String("LCD");
private static final String ICTP_IO_LCD_DATA = new String("LCD_data");

private static final String ICTP_IO_SWITCHES = new String("Switches");
private static final String ICTP_IO_BUTTONS  = new String("Buttons");
private static final String ICTP_IO_ADC      = new String("ADC");

private static final String ICTP_IO_COMMAND   = new String("command");
private static final String ICTP_IO_CMD_READ  = new String("read");
private static final String ICTP_IO_CMD_WRITE = new String("write");
private static final String ICTP_IO_CMD_IOCTL = new String("ioctl");
private static final String ICTP_IO_LCD_IOCTL = new String("ioctlCmd");
private static final String ICTP_IO_IOCTL_SELECT   = new String("select");
private static final String ICTP_IO_IOCTL_DESELECT = new String("deselect");
private static final String ICTP_IO_IOCTL_CLEAR    = new String("clear");
private static final String ICTP_IO_IOCTL_HOME     = new String("home");
```

Once the device number, the command code and the data are known the call to the ICTP_IO object can be made and the command is sent to the server on the HC-11. The error code coming back from the HC-11 as well as possible data are returned to the caller in form of html text in the form *Error Code:* or *ICTP_IO Data:* respectively. In case the requester is a html page these data are simply printed in case it is a more complex program that creates the URL string itself it is this program that must interpret the html text coming back to it. For further details about the HC11Servlet please refer to the appendix in which you will find the full source code.

3 Building Graphical User Interfaces with Swing

3.1 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.

3.2 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 SUNs Forte4J™.

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.

3.3 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**.

3.4 Creating the View

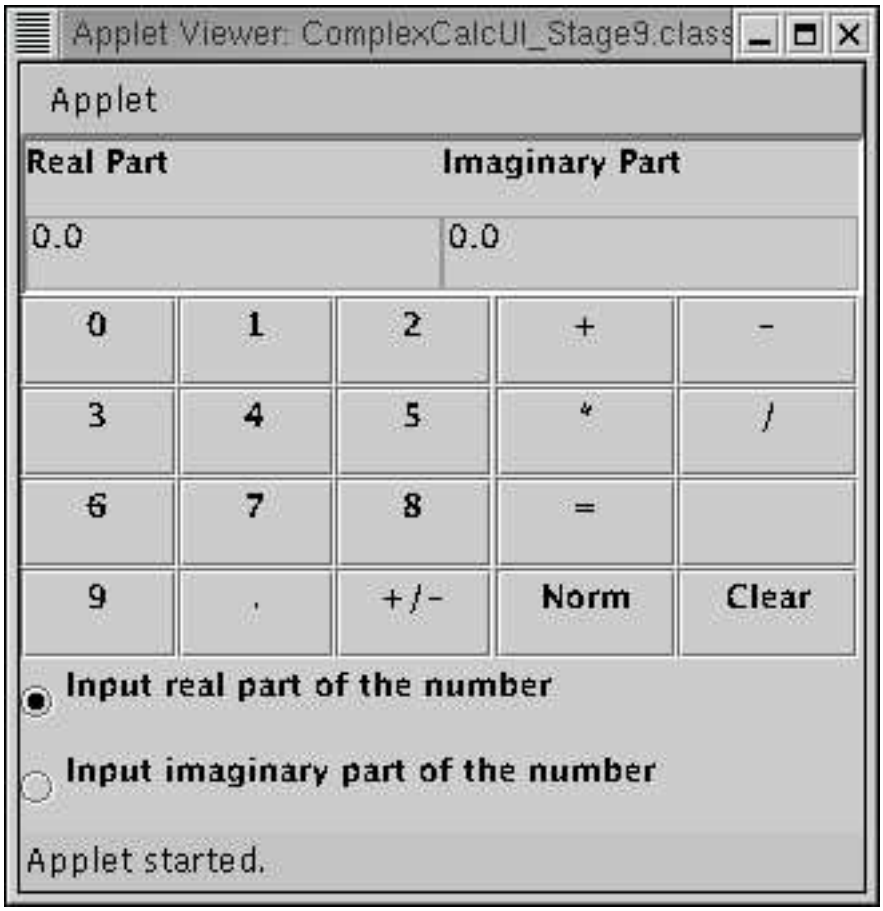
Since a single example says more than thousand words we will go through the design and implementation of the complex calculator step by step. We will start with the GUI (the view), then attach a few simple actions, just in order to demonstrate that we can actually launch actions with our interface (a 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.



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

Figure 11: The Layout of Widgets for the Complex Calculator

3.5 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.*;
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 12

Figure 12: 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.

3.6 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 13

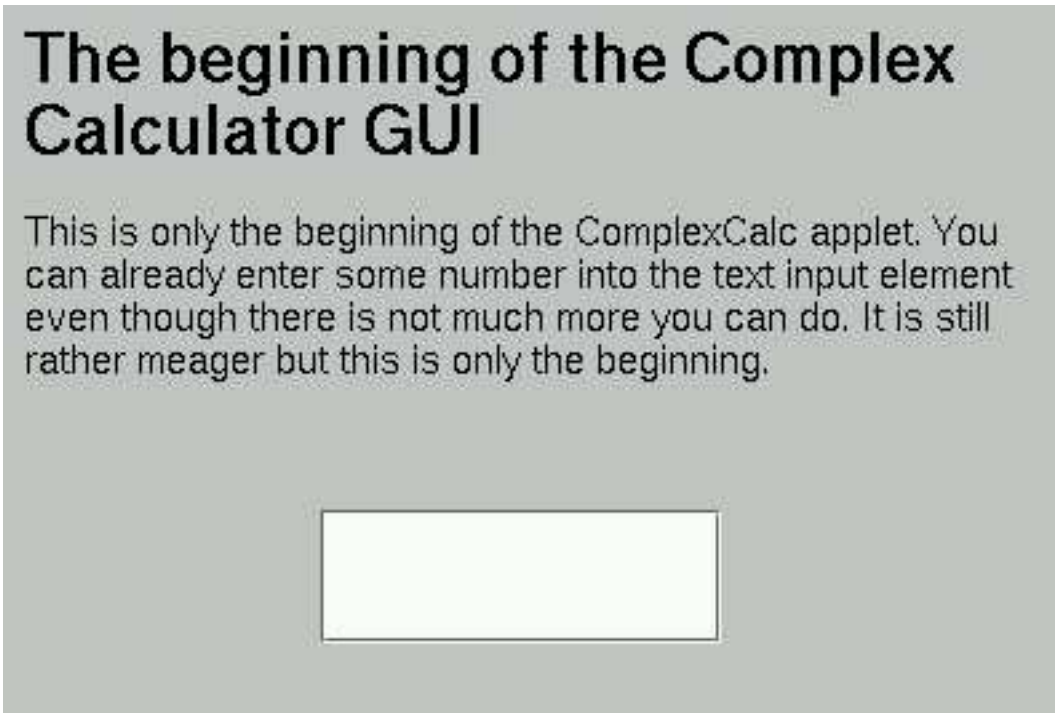


Figure 13: The first Java applet within a WEB page as seen by Netscape

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

3.7 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 14: 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.

3.8 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:

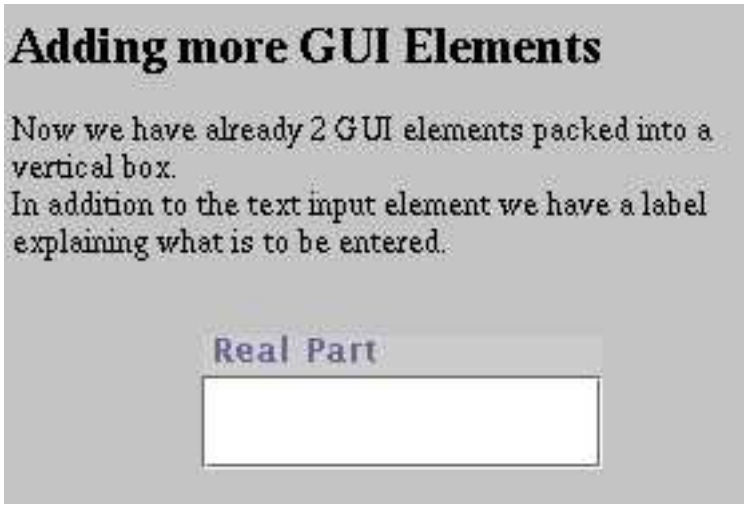


Figure 15: Two Applets in a Vertical Box

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.

3.8.1 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.

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

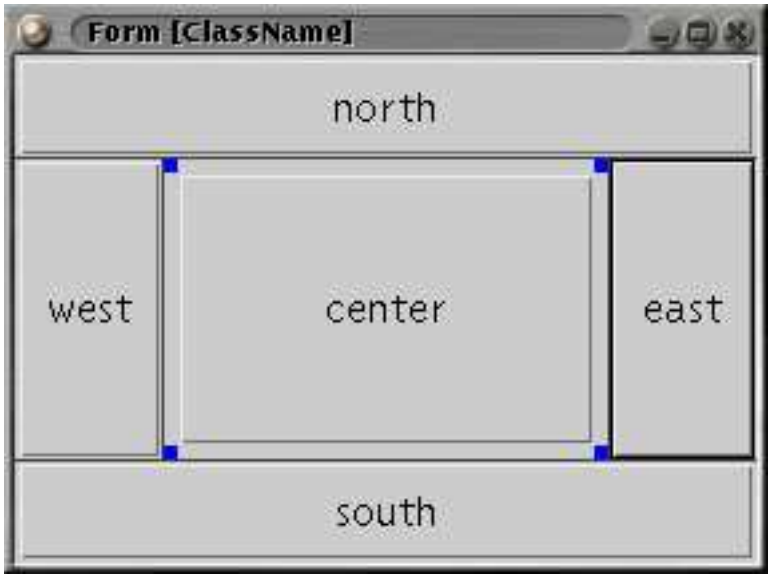


Figure 16: The BorderLayout

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");

}
```

3.8.2 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.

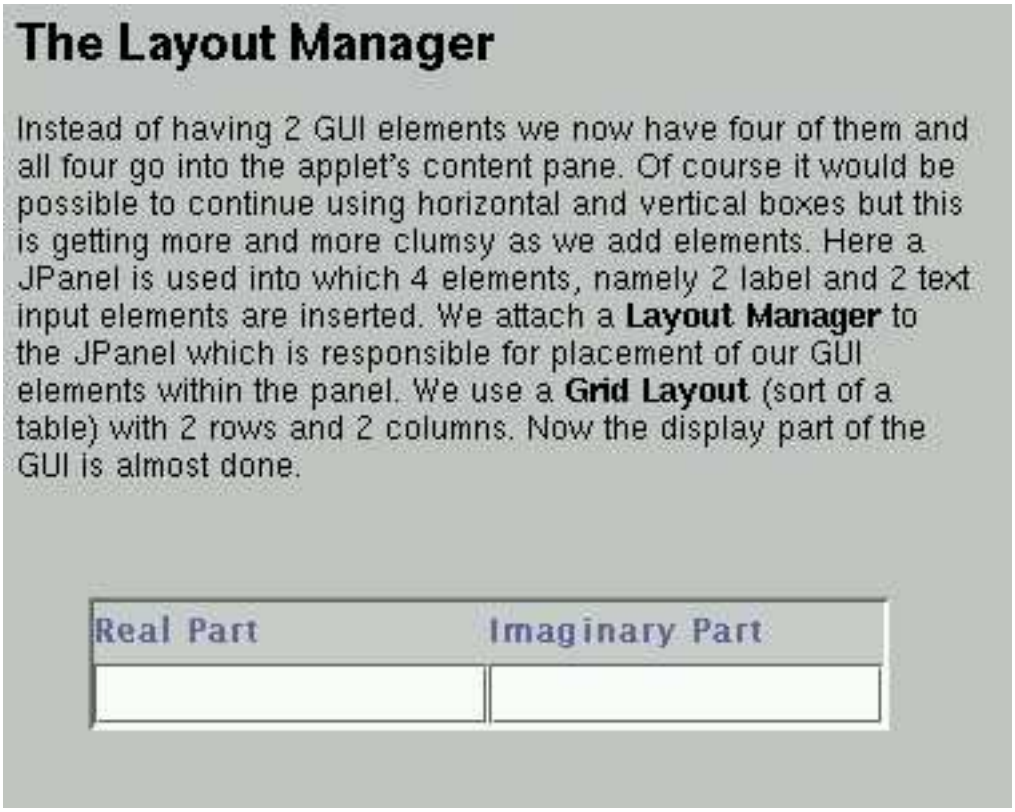


Figure 17: The Display Part of the Calculator

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;
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

Figure 18: All the Calculator Widgets are there!

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.

3.9 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 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;
    }
    }
}
```

3.10 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 19

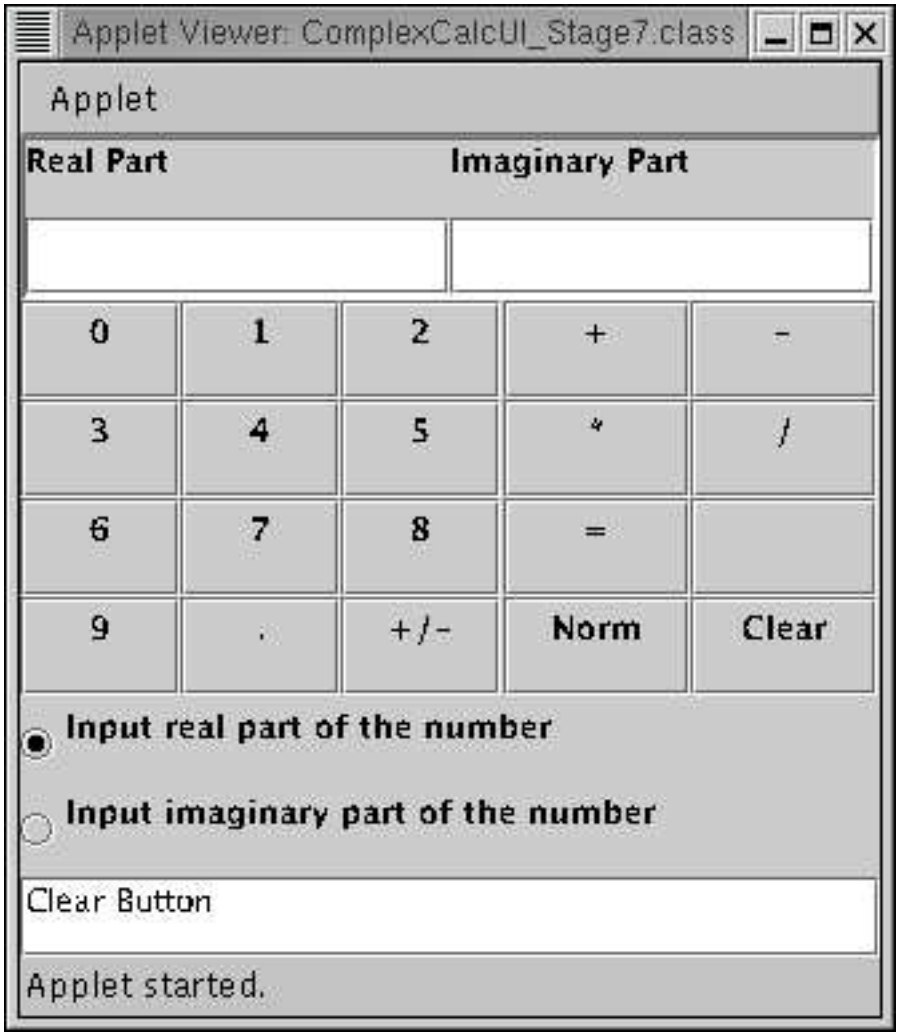


Figure 19: First Test of Activation

4 Software Components, Java Beans

4.1 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.

4.2 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.

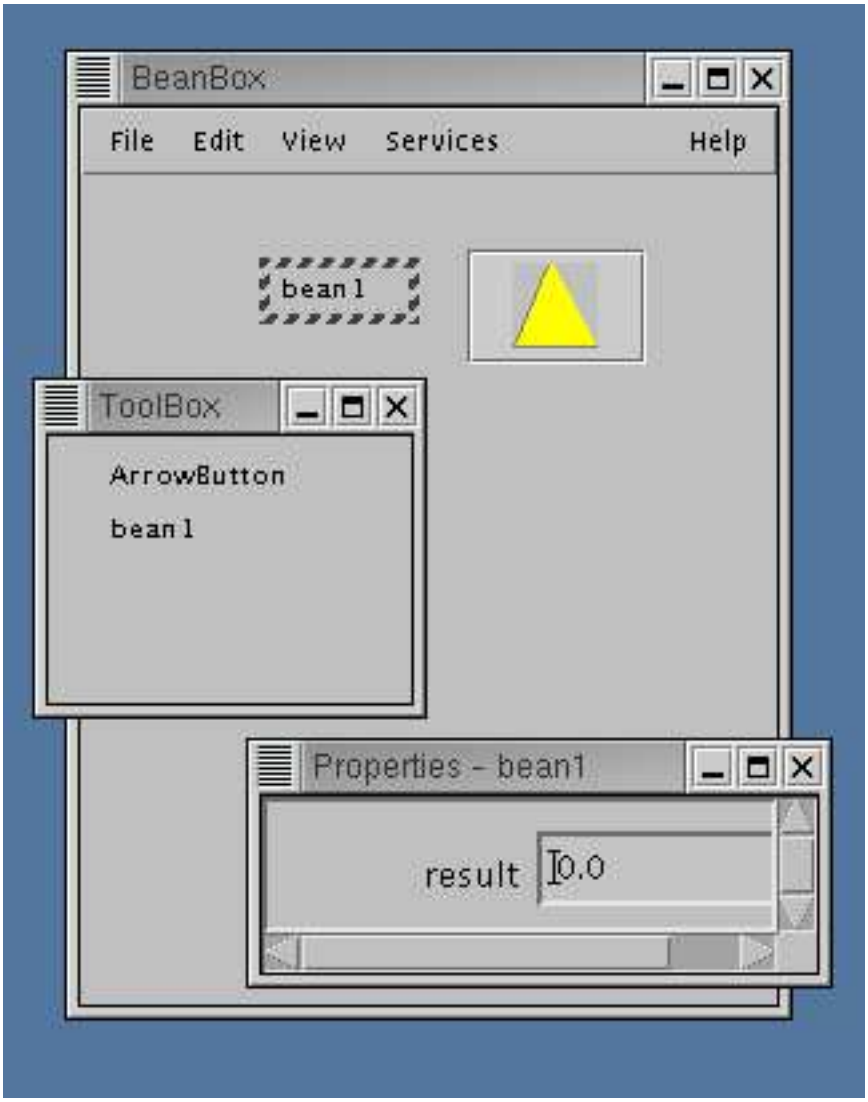


Figure 20: Bean1 sitting in the beanbox

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
```

4.3 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());
        }
    }
```

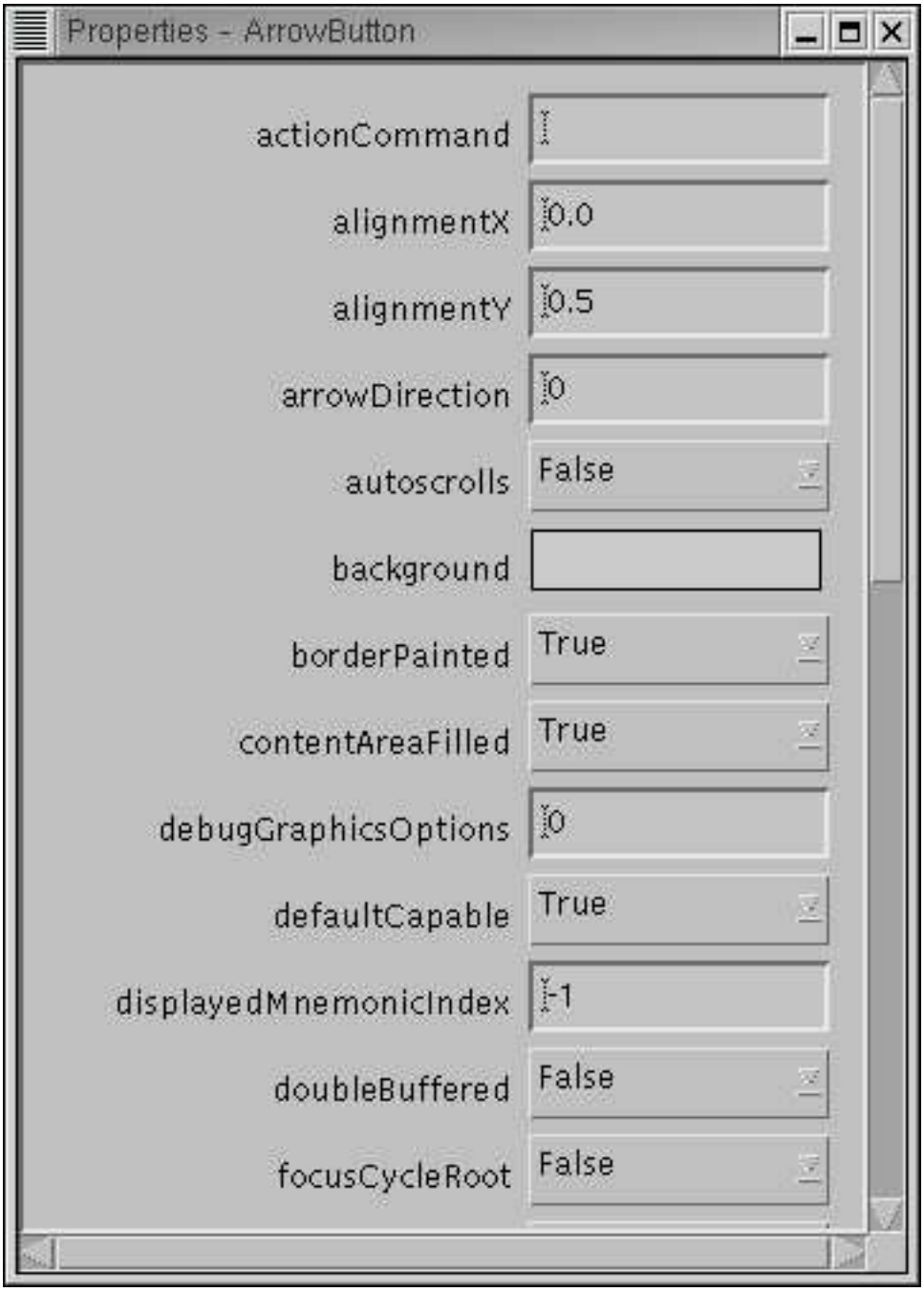


Figure 21: The Properties of the ArrowButton

```
    private final static Class beanClass = ArrowButton2.class;
}
```

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.

4.4 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;
}
```

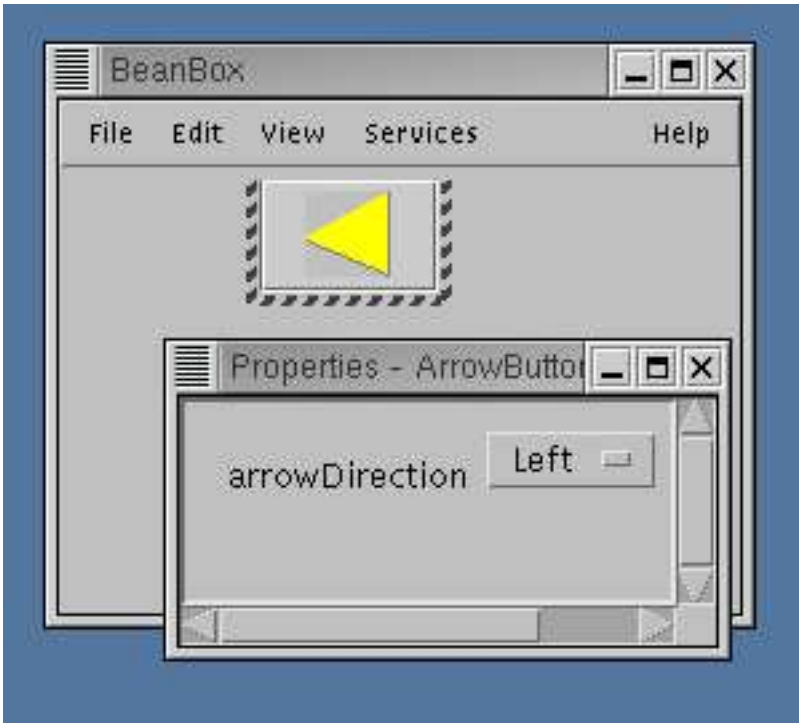


Figure 22: The Property Editor for the Arrow Directions

4.5 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 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 Action-Event 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());
        }
    }
```

4.6 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;
  }

}
```

5 Putting things together

In the first chapters a desciption was given on how to access the HC-11 hardware. After that the construction of applications and applets with a sophisticated graphical user interface was shown and finally the building blocks of these GUIs, namely beans, were shown. Having all these elements ready we now want to put things together and build an application with a nice GUI that accesses the HC-1 hardware. This GUI uses the theoretical concept of model-view-controller as explained in the previous chapter. Please refer to fig. 23.

The application consists of the HC-11 photograph which has been made *active*. When the user passes the mouse over the LCD it becomes high-lighted and text may be entered. The same is true for the LEDs, When you point to one of the LEDs in the photograph, a button appears. This active photograph implements the View (IOBoardView). When a LED button is pushed an ledActionEvent is fired, when you type a carriage return in the LCD a lcdActionEvent is the result.

Since the IOBoard is implemented as a bean, any other bean capable of treating ledActionEvents of lcdActionEvents can be connected to it. Such a bean is the IOBoardmodel which acts as am actionListener for those two events. Once a ledActionEvent is captured by the model, it extracts the new led values from the ledActionEvent and communicates this

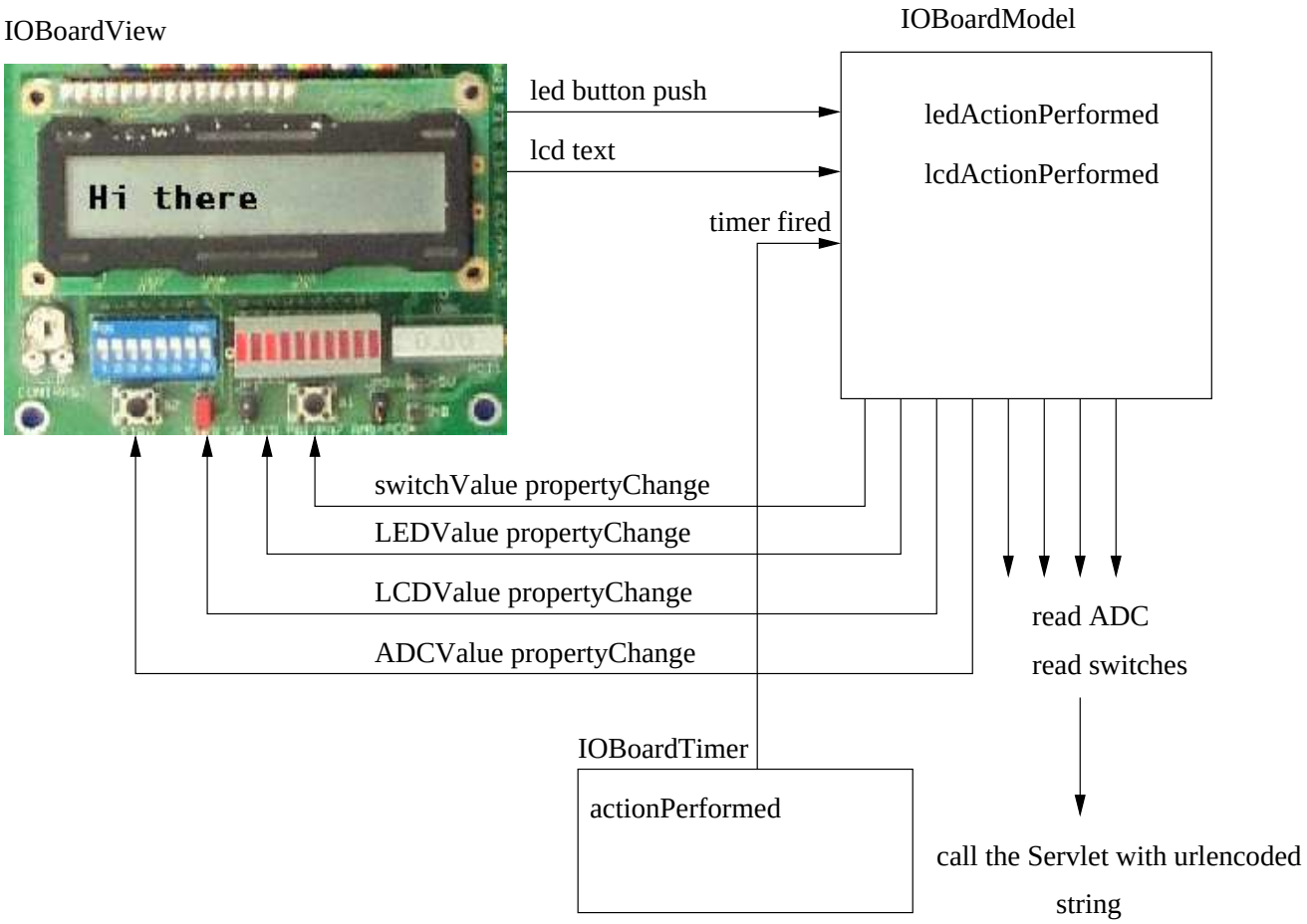


Figure 23: Architecture of a complete application

new value to the HC11Servlet running on the TINI. In order to do so it must create a URL and send a URL encoded string to the servlet containing the name/value pair *intData=0x01* (if only the first LED is to be switched on). Here is the code snippet showing how this is done. First a httl connection is opened to the machine we want to task to. *hostname* is the name of the TINI whose Http server we want to access.

```
    public int write(int dev, int data)
    {
      int retCode;
try {
    url = new URL("http://" + hostname + HTTPSERVER);
    connection = url.openConnection();
} catch (Exception e)
  {
    System.out.println("Could not open URL connection\n");
    System.out.println(e.getMessage());
    return OPEN_FAILURE;
  }
```

```
connection.setDoOutput(true);

return sendWriteCmd(ICTP_IO_WRITE,dev,data);
    }
```

Once the connection is established we can send the http code:

```
    cmdString = "command="+cmds[cmd];
    devString = "device="+devices[dev];
    tmpLedData = data & 0xff;
    ledData = "intData=" + "0x" +Integer.toHexString(tmpLedData);
    urlString = cmdString + "&" + devString + "&" + ledData + "&" + submitString;
    sendURLString(urlString);
```

As you can see the string

```
command=write&device=LED&intData=0x01}
```

is created and sent to the TINI. This explains how LEDs and LCD are written. However, how do we know if the switches have been changed by somebody of if the ADC has a new value? The answer is simple: We periodically read their values and check. In order to do this in regular intervals we use a Timer bean which fires an actionPerformed event at an interval that can be predetermined. The actionPerformed event is captured by the model which interogates the HC11Servlet for the current value. If a value change is seen the model fires a propertyChange event, which in turn is captured by the view. The View then does the necessary updates on the display.

6 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.

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 Appendixes

7.1 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>
```

7.2 The servlet treating the post requests

```
// HC11Servlet.java - A servlet that gives access to the HC11 device.
//
// Copyright (C) 2003  Ulrich.Raich
//
// This program is free software; you can redistribute it and/or
// modify it under the terms of the GNU General Public License
// as published by the Free Software Foundation; either version 2
// of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
//
```

```
// Ulrich.Raich
// AB Division
// CERN
// CH-1211 Geneva 23
// Switzerland
//
// email: Ulrich.Raich@cern.ch
//

import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;
import tini_ictp_io.TINI_ICTP_IO;

/** This is the servlet that allows access to the HC-11 devices
 * It uses the TINI_ICTP_IO class in which implements the ICTP_IO
 * serial protocol on the TINI. It gets request to read/write hc11 devices
 * like LCD, LEDs Switches etc. and it executes the corresponding commands
 * and returns the results.
 * @author Ulrich Raich
 * @version 0.5
 */
public class HC11Servlet
extends HttpServlet
{
    /** serve http post requests
     * @param req the http request
     * @param res the response
     */
     public HC11Servlet()
     {
        super();
        /*
         * create a new connection to the hc11
         */
        tini_ictp_io = new TINI_ICTP_IO();
        tini_ictp_io.setDebug(true);
     }
        public void doPost( HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
doGet( req, res);
}

public void doGet( HttpServletRequest req, HttpServletResponse res)
```

```
throws ServletException, IOException
{
PrintWriter out;
Enumeration     enum;
                byte[]          retData;
                int             retCode;
int i;
                int tmp;
                boolean debug = true;

res.setContentType( "text/html");

                retCode = TINI_ICTP_IO.ICTP_IO_SUCCESS;

// Get writer
out = res.getWriter();

out.println( "<html>");
out.println( "<head><title>HC11 Servlet</title></head>");
out.println( "<body>");

enum = req.getHeaderNames();
                if( enum.hasMoreElements()) {
                    if (debug) {
                        out.println( "<h1>Request headers:</h1>");
                        out.println( "<pre>");
                    }
                    while( enum.hasMoreElements()) {
                        String name = (String)enum.nextElement();
                        if (debug)
                            out.println( " " + name + ": " + req.getHeader(name) );
                    }
                    if (debug)
                        out.println( "</pre>");
                }

enum = req.getParameterNames();
                if( enum.hasMoreElements()) {
                    if (debug) {
                        out.println( "<h1>Servlet parameters (Multiple Value style):</h1>")
                        out.println( "<pre>");
                    }
                    while( enum.hasMoreElements()) {
                        String   name = (String)enum.nextElement();
                        String[] vals = req.getParameterValues( name);
                        String parameter;
                        if (debug)
```

```
    out.println( " " + name + " = " +    req.getParameter(name) );
        /*
         * check for the names defined for the ICTP_IO protocol
         * first try to get the command to be executed
         * Only read, write and ioctl are implemented
         */
parameter = req.getParameter(name);
System.out.println("name: " + name + ", parameter: " + parameter);
if (name.equals(ICTP_IO_COMMAND)) {
    if (parameter.equals(ICTP_IO_CMD_READ))
        servletCmd = TINI_ICTP_IO.ICTP_IO_READ;
    else if (parameter.equals(ICTP_IO_CMD_WRITE))
        servletCmd = TINI_ICTP_IO.ICTP_IO_WRITE;
    else if (parameter.equals(ICTP_IO_CMD_IOCTL))
        servletCmd = TINI_ICTP_IO.ICTP_IO_IOCTL;
    else
        System.out.println("Invalid command code");
}

        /*
         * then check for the device that has been selected
         */
else if (name.equals(ICTP_IO_DEVICE)) {
    if (parameter.equals(ICTP_IO_LED))
        servletDevice = TINI_ICTP_IO.ICTP_IO_LED;
    else if (parameter.equals(ICTP_IO_LCD))
        servletDevice = TINI_ICTP_IO.ICTP_IO_LCD;
    else if (parameter.equals(ICTP_IO_SWITCHES))
        servletDevice = TINI_ICTP_IO.ICTP_IO_SWITCHES;
    else if (parameter.equals(ICTP_IO_BUTTONS))
        servletDevice = TINI_ICTP_IO.ICTP_IO_BUTTONS;
    else if (parameter.equals(ICTP_IO_ADC))
        servletDevice = TINI_ICTP_IO.ICTP_IO_ADC;
    else
        System.out.println("Invalid device code");
}


else if (name.equals(ICTP_IO_LED_NUM_DATA)) {
    System.out.println("LED data from direct value!");
    LED_data = (byte) (Integer.decode(parameter)).intValue();
    System.out.println("LED data from direct value: " + LED_data);
}

else if (name.equals(ICTP_IO_LED_DATA)) {
    System.out.println("LED data are coming!");
        /*
```

```
                 * check for values to be written
                 */

            if (vals != null) {
                LED_data = 0;
                for( i = 0; i<vals.length; i++) {
                    if (vals[i].equals(ICTP_IO_LED_BIT1))
                        LED_data |= 0x1;
                    else if (vals[i].equals(ICTP_IO_LED_BIT2))
                        LED_data |= 0x2;
                    else if (vals[i].equals(ICTP_IO_LED_BIT3))
                        LED_data |= 0x4;
                    else if (vals[i].equals(ICTP_IO_LED_BIT4))
                        LED_data |= 0x8;
                    else if (vals[i].equals(ICTP_IO_LED_BIT5))
                        LED_data |= 0x10;
                    else if (vals[i].equals(ICTP_IO_LED_BIT6))
                        LED_data |= 0x20;
                    else if (vals[i].equals(ICTP_IO_LED_BIT7))
                        LED_data |= 0x40;
                    else if (vals[i].equals(ICTP_IO_LED_BIT8))
                        LED_data |= 0x80;
                }
                out.print( "<b> " + name + " = </b>");
                out.println( vals[0]);
                for( i = 1; i<vals.length; i++)
                    out.println( "           " + vals[i]);
            }
            System.out.println("LED_data: " + LED_data);
            out.println( "<p>");
        }
        else if (name.equals(ICTP_IO_LCD_DATA)) {
            System.out.println("LCD data are coming!");
            LCD_data = new byte[parameter.length()];
            LCD_data = parameter.getBytes();
            System.out.println("Writing " + parameter + "to LCD");

        }
        else if (name.equals(ICTP_IO_LCD_IOCTL)) {
            if (parameter.equals(ICTP_IO_IOCTL_SELECT))
                servletIoctlCmd = TINI_ICTP_IO.ICTP_IO_SELECT;
            else if (parameter.equals(ICTP_IO_IOCTL_DESELECT))
                servletIoctlCmd = TINI_ICTP_IO.ICTP_IO_DESELECT;
            else if (parameter.equals(ICTP_IO_IOCTL_HOME))
                servletIoctlCmd = TINI_ICTP_IO.ICTP_IO_HOME;
            else if (parameter.equals(ICTP_IO_IOCTL_CLEAR))
                servletIoctlCmd = TINI_ICTP_IO.ICTP_IO_CLEAR;
```

```
                            else
                                out.println("Illegal ioctl cmd: " + parameter);
                        }
                    }
                    out.println("</pre>");
                }
                if ((servletDevice == TINI_ICTP_IO.ICTP_IO_LED) &&
(servletCmd == TINI_ICTP_IO.ICTP_IO_WRITE))
    out.println("LED_data: " + LED_data);
else if ((servletDevice == TINI_ICTP_IO.ICTP_IO_LCD) &&
(servletCmd == TINI_ICTP_IO.ICTP_IO_WRITE))
    out.println("LCD_data: " + new String(LCD_data));
                out.println("<br>device = " + servletDevice);
                out.println("<br>command = " + servletCmd);
                out.println("</body></html>");
                /*
                 * now execute the command
                 */

                out.println("before execution");
                if (tini_ictp_io.open() != TINI_ICTP_IO.ICTP_IO_SUCCESS) {
                    out.println("HC11 Servlet: Error when opening the HC-11 connection");
                    return;
                }
                switch (servletCmd) {
                    case TINI_ICTP_IO.ICTP_IO_WRITE:
if (servletDevice == TINI_ICTP_IO.ICTP_IO_LED)
    {
System.out.println("Writing " + LED_data + " to device "
+ servletDevice);
retCode = tini_ictp_io.write((byte)servletDevice,LED_data);
    }
else if (servletDevice == TINI_ICTP_IO.ICTP_IO_LCD)
    {
System.out.println("Writing " + LCD_data + " to device "
+ servletDevice);
retCode = tini_ictp_io.write((byte)servletDevice,LCD_data);
    }
break;
                    case TINI_ICTP_IO.ICTP_IO_READ:
                        if (servletDevice == TINI_ICTP_IO.ICTP_IO_LCD) {
                            if (debug)
                                out.println("16 chars to be read from LCD");
                            retData = new byte[16];
                        }
                        else {
                            retData = new byte[1];
```

```
                        }
                        if ((retCode = tini_ictp_io.read((byte)servletDevice, retData)) !=
TINI_ICTP_IO.ICTP_IO_SUCCESS)
                            out.println("Fatal error when trying to read ICTP_IO device");
                        else

                            if (retData.length > 1) {
                                out.println("Read from device: " + servletDevice + " " +
new String(retData));
                                out.print("ICTP_IO Data: ");
                                for (i=0;i<retData.length;i++) {
                                    tmp = ((int)retData[i]) & 0xff;
                                    out.print("0x" + Integer.toHexString(tmp) + " ");
                                }
                                out.println();
                            }
                            else {
                                out.println("Read from device: " + servletDevice + " " +
                                Integer.toHexString((int)retData[0]));
                                tmp = ((int)retData[0]) & 0xff;
                                out.println("ICTP_IO Data: " + "0x" + Integer.toHexString(tm
                            }
                        break;
                    case TINI_ICTP_IO.ICTP_IO_IOCTL:
                        if (servletDevice != TINI_ICTP_IO.ICTP_IO_LCD)
                            out.println("ioctl command are only valid on the LCD device");
                        else
                        {

                        if ((retCode = tini_ictp_io.ioctl((byte)servletDevice, servletIoctl(
 != TINI_ICTP_IO.ICTP_IO_SUCCESS)
                            out.println("Fatal error when trying to ioctl ICTP_IO device");
                        }
                        break;
                    default:
                    out.println("HC11 Servlet: Illegal command");
                }
                out.println("Error Code: " + "0x" + Integer.toHexString(retCode));
        // needed for the client

                if (tini_ictp_io.close() != TINI_ICTP_IO.ICTP_IO_SUCCESS) {
                    out.println("HC11 Servlet: Error when closing the HC-11 connection");
                    return;
                }
        }

private void print( PrintWriter out, String name, String value)
```

```
{
out.print( " " + name + ": ");
out.println( value == null ? "&lt;none&gt;" : value);
}

private void print( PrintWriter out, String name, int value)
{
out.print( " " + name + ": ");
if (value == -1)
{
out.println( "&lt;none&gt;");
}
else
{
out.println( value);
}
}
        /*
         * strings that are needed for comparison
         */
        private static final String ICTP_IO_DEVICE = new String("device");
        private static final String ICTP_IO_LED    = new String("LEDs");
        private static final String ICTP_IO_LED_DATA = new String("LED_data");
        private static final String ICTP_IO_LED_BIT1 = new String("b1");
        private static final String ICTP_IO_LED_BIT2 = new String("b2");
        private static final String ICTP_IO_LED_BIT3 = new String("b3");
        private static final String ICTP_IO_LED_BIT4 = new String("b4");
        private static final String ICTP_IO_LED_BIT5 = new String("b5");
        private static final String ICTP_IO_LED_BIT6 = new String("b6");
        private static final String ICTP_IO_LED_BIT7 = new String("b7");
        private static final String ICTP_IO_LED_BIT8 = new String("b8");
        private static final String ICTP_IO_LED_NUM_DATA = new String("intData");
        private static final String ICTP_IO_LCD      = new String("LCD");
        private static final String ICTP_IO_LCD_DATA = new String("LCD_data");

        private static final String ICTP_IO_SWITCHES = new String("Switches");
        private static final String ICTP_IO_BUTTONS  = new String("Buttons");
        private static final String ICTP_IO_ADC      = new String("ADC");

        private static final String ICTP_IO_COMMAND   = new String("command");
        private static final String ICTP_IO_CMD_READ  = new String("read");
        private static final String ICTP_IO_CMD_WRITE = new String("write");
        private static final String ICTP_IO_CMD_IOCTL = new String("ioctl");
        private static final String ICTP_IO_LCD_IOCTL = new String("ioctlCmd");
        private static final String ICTP_IO_IOCTL_SELECT   = new String("select");
        private static final String ICTP_IO_IOCTL_DESELECT = new String("deselect");
        private static final String ICTP_IO_IOCTL_CLEAR    = new String("clear");
```

```
        private static final String ICTP_IO_IOCTL_HOME     = new String("home");

        private int servletCmd;
        private int servletDevice;
        private int servletLCDdata;
        private int servletLEDdata;
        private byte servletIoctlCmd;
        private byte LED_data;
        private byte[] LCD_data;

        TINI_ICTP_IO tini_ictp_io;
}
```

7.3 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;
  }
}
```

References

[1] David Flanagan (1997). *Java in a nutshell: A desktop quick reference*. Second Edition. O'Reilly.

[2] Patrick Chan and Rossana Lee (1997). *The Java class libraries: An annotated reference*. Addison Wesley.

[3] *The Java tutorial. A practical guide for programmers*. Sun Microsystems. Available online at `http://java.sun.com`