GUI

Das Ziel bei der Entwicklung von BlueJ war, eine Oberfläche zu bieten, bei der man ohne eine GUI auskommt. Das Erstellen einer GUI (graphical user interface), also einer grafischen Benutzerschnittstelle zu einer Java- Anwendung, fällt mit dem Javaeditor (http://lernen.bildung.hessen.de/informatik/javaeditor/index.htm) sehr viel leichter als mit BlueJ. In der drag and drop – Oberfläche stehen uns nicht nur die Hilfsmittel bereit, um einfach eine einfache Oberfläche aufzubauen, sondern es wird jeweils auch ein wichtiger Teil des Programmcodes mit generiert.

JFrame

Erzeugt man einen JFrame durch Anklicken des entsprechenden Buttons in der Standard-leiste, dann sollte man unbedingt zunächst einen neuen Ordner für das neue Projekt erstellen. Man gibt der Gui-Klasse, die man erstellen will, einen Namen (hier einfach Gui). Der Javaeditor erzeugt dann den Klassentext und ein zugehöriges Formular. Die einzelnen Abschnitte sind kurz erläutert.

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

/**
  *
  * Beschreibung
  *
  * @version 1.0 vom ...
  * @author
  */

public class Gui extends JFrame {
  // Anfang Variablen
  // Ende Variablen

  public Gui(String title) {
    // Frame-Initialisierung
    super(title);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) { System.exit(0); }
    });
    int frameWidth = 300;
    int frameHeight = 300;
    setSize(frameWidth, frameHeight);
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    int x = (d.width - getSize().width) / 2;
    int y = (d.height - getSize().height) / 2 ;
    setLocation(x, y);
    Container cp = getContentPane();
    cp.setLayout(null);
    // Anfang Komponenten
    // Ende Komponenten
    setResizable(false);
    setVisible(true);
  }
  // Anfang Ereignisprozeduren
  // Ende Ereignisprozeduren

  public static void main(String[] args) {
    new Gui("Gui");
  }
}
```

Unsere Klasse erbt von JFrame, ...

... enthält einen Konstruktor, der ...

... den Konstruktor von JFrame aufruft, ...

... fügt einen WindowListener hinzu, ...

... damit wir das Ereignis verarbeiten können, dass unser Fenster geschlossen werden soll.

Größe und Position werden definiert.

Zur contentPane s.u.

ContentPane, LayoutManager und main-Methode

Ein großer Teil der benötigten Eigenschaften und Methoden wird schon von JFrame geerbt. Jeder JFrame hat z.B. eine ContentPane, die wir uns mit der Methode getContentPane() vom Frame – Objekt geben lassen können. Das ContentPane – Objekt ist die Inhaltsfläche des Fensters.
In unserem Beispiel bekommt das ContentPane – Objekt keinen LayoutManager. Die Layout – Manager ermöglichen standardisierte Layouts der Oberfläche, in unserem Beispiel ist das Layout daher absolut und statisch.
Am Ende der Klassendefinition finden wir die main – Methode der Klasse. Sie ermöglicht das Erstellen eines ausführbaren Programmes. Erst so ist der Zugriff über das jeweilige Betriebssystem und die JAVA runtime-environment z.B. auf einen jar-file möglich.

Hinzufügen von Komponenten

Klicken wir im Javaeditor - Fenster die Lasche Swing1 an, dann können wir mit drag and drop einen JButton hinzufügen, indem wir ihn irgendwo auf dem Formular ablegen.
Im Programmtext wird er
deklariert und definiert: `private JButton jButton1 = new JButton();`
seine Position festgelegt: `jButton1.setBounds(8, 8, 75, 25);`
Beschriftung festgelegt: `jButton1.setText("jButton1");`
und er wird der ContentPane hinzugefügt: `cp.add(jButton1);`

ActionListener

Damit der Button auf Aktionen des Benutzers reagieren kann, muss er einen ActionListener hinzugefügt bekommen.

```
jButton1.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    jButton1ActionPerformed(evt);
  }
```

Der ActionListener führt hier die Reaktion nicht selbst aus, sondern reicht sie an eine weitere Methode weiter[1], die mit `jButton1ActionPerformed(evt)` bezeichnet ist und zunächst leer ist. An dieser Stelle ist die tatsächliche Aktion zu programmieren, die das Anklicken des Buttons auslösen soll.
ActionListener ist eine Schnittstelle. In ihr ist also nur beschrieben, was ein ActionListener können muss, wenn er denn erzeugt wird. Die Definition der Schnittstelle ActionListener fordert allein eine Methode[2], nämlich die oben angegebene Methode
`public void actionPerformed(ActionEvent evt)`.
In unserem Beispiel wird tatsächlich ein ActionListener – Objekt erzeugt, was man an dem Auftauchen von **new** erkennen kann. Bemerkenswert ist, dass dies ohne eine vorherige Definition einer ActionListener – Klasse geschieht.
Statt dessen wird eine anonyme innere Klasse verwendet. Da nur dies eine Objekt zu erzeugen ist, das sofort an das Button – Objekt gekoppelt wird und auf das auf einem anderen Wege als über den zugehörigen Button nie zugegriffen werden darf, ist diese Technik hier angemessen.

1 Zum Modellierungsaspekt, der zum Ausgliedern in diese Methode führt, gibt es einen gut passenden Text aus Guido Krüger: GoTo Java 2, der auszugsweise hinten an diesen Text angefügt ist.
2 Das ist nämlich das Einzige, was man mit einem Button tun kann: mit ihm mitteilen, dass etwas getan werden soll. Ereignisse können dazu Maus- und Tastaturereignisse sein.

JTextField

Entsprechend lässt sich auch ein JTextField einfügen. Grundlegende Eigenschaft ist hier der Textinhalt des Feldes, den man mit den Methoden
`jTextField1.setText("jTextField1");` setzen und mit
`jTextField1.getText();` lesen kann. Der Rückgabewert der letzten Methode ist ein Stringobjekt, das man nun verarbeiten kann. Er ist auch dann ein Stringobjekt, wenn Zahlen eingegeben worden sind. Da dies natürlich ein sehr häufiges Problem ist, stellt die Klasse String Methoden bereit, die den Text in ein entsprechende Zahl verwandeln.

Menüs

Der Javaeditor stellt auch einfache Möglichkeiten bereit, Menüs zu realisieren. Ebenfalls in der Funktionsleiste Swing1 finden wir ein Symbol für eine JMenuBar, die wir durch Anklicken und konfigurieren im Dialog (prinzipiell braucht man nichts zu tun) einfügen können. Es werden dann eingefügt
`private JMenuBar jmb = new JMenuBar();` bei den Deklarationen und
`setJMenuBar(jmb);` bei den Definitionen im Konstruktor. Diese Methode ist also auch wieder eine Methode, die von JFrame bereit gestellt wird. Dieser JMenuBar fügt man Menüs mit dem in der Funktionsleiste Swing1 daneben liegenden Symbol für ein JMenu hinzu. Auch hier öffnet sich ein Konfigurationsdialog, bei dem wir in das Feld für die JMenuBar den richtigen Namen eintragen sollten (hier also jmb).
Die einzelnen Menüpunkte sind dem Menü dann per Hand oder durch Einfügen über den Objektinspektor hinzuzufügen. Sie sind Objekte der Klasse JMenuItem
Ebenfalls per Hand sind ihnen die entsprechenden Methoden zur Ereignisbehandlung hinzuzufügen.
Eine funktionierende Anwendung zu realisieren, sollte mit diesen Hilfen gelingen. Mit weiteren Komponenten sollte man sich dann beschäftigen, wenn man sie braucht. SUN stellt mit seinen HOW-TOs Hilfe auf der homepage bereit.
Nur ein kleiner Tip: Bei JTextArea kann das Scrollen automatisch mit implementiert werden, wenn nicht die JTextArea selbst auf die ContentPane gebracht wird, sondern eine JScrollPane, der dann wiederum die JTextArea hinzugefügt wird[1]. Ein Beispiel:

```
ausgabe = new JTextArea();
ausgabe.setText(null);
JScrollPane scroll = new JScrollPane(ausgabe);
scroll.setBounds(160, 10, 200, 300);
contentPane.add(scroll);
```

Inhalte werden dann einzeln in neue Zeilen gebracht mit:

```
ausgabe.append(<ein String>+"\r\n")
```

return und newline

Ein etwas schwierigeres Problem ist das Realisieren von Tabellen, das deswegen gesondert erläutert wird.

GUI und Tabellen

Tabellen, die mit Hilfe einen grafischen Oberfläche dargestellt werden sollen, bestehen neben der grafischen Komponente immer aus einer Modellkomponente, die eine Repräsentation der Daten darstellt. Folgerichtig stellt Swing dazu eine Klasse mit einem abstrakten Datenmodell bereit.
Sie heißt AbstractTableModel, eine konkrete Modellklasse muss also von dieser abstrakten Klasse erben. Da hier keine Schnittstelle vorliegt, bringt diese Klasse schon Methoden mit und nur einige müssen implementiert werden.

1 Siehe auch das Vorgehen bei der Tabelle.

Diese Unterscheidung ist hier wichtig, da u.a. die Methode public boolean isCellEditable schon existiert, die standardmäßig alle Felder so setzt, dass keine Werte eingegeben werden können. Ist allein eine Darstellung von Ergebnissen gewünscht, ist das die richtige Wahl, sonst aber sind ggf. die Werte anders zu setzen. Das eigentlich Bemerkenswerte ist aber, dass das Modell einen Listener zugeordnet bekommen kann, so dass auf Veränderungen der Werte in der Tabelle durch den Benutzer reagiert werden kann.

Die hier betrachtete Anwendung nutzt die Tabelle einfach zur Darstellung der Wertetabelle einer Funktion. Sie verwendet dabei für das AbstractTableModel eine private innere Klasse.

```
private class ModellKlasse extends AbstractTableModel {
   private String[] spaltenBeschriftungen = { "Wert", "Radikand/Wert" };
   private Object[][] daten = new String[maxZahl][2];

   public String getColumnName(int col)
     { return spaltenBeschriftungen[col]; }
   public int getRowCount()
     { return daten.length; }
   public int getColumnCount()
     { return spaltenBeschriftungen.length; }
   public boolean isCellEditable(int row, int col)
     { return false; }
   public Object getValueAt(int row, int col)
     { return daten[row][col]; }
   public void setValueAt(Object value, int row, int col) {
       daten[row][col] = value;
       fireTableCellUpdated(row, col);
       }
   }
```

In der Gui-Klasse wird ein Objekt von diesem AbstractTableModel erzeugt und der Gui hinzugefügt:

```
dasModell = new ModellKlasse();
tabelle = new JTable(dasModell);
scroll = new JScrollPane(tabelle);
scroll.setBounds(160, 10, 200, 300);
contentPane.add(scroll);
```

Die folgende Methode des Gui-Objektes wird zum Setzen der Werte benutzt.

```
private void setzeTabellenDaten() {
    double[][] werte =
       datenModell.holeWerte(); // <holt die Tabellendaten aus dem Modell>
    int schritte = datenModell.holeAnzahlSchritte();  // <s.o.entspr.>
    dasModell.setValueAt("\r\n",0,0);
    for (int i=0; i<schritte; i++) {
        dasModell.setValueAt(werte[0][i]+"\r\n",i,0);
        dasModell.setValueAt(werte[1][i]+"\r\n",i,1);
   }
    for (int i=schritte; i<maxZahl; i++) {  // <löscht den Rest>
        dasModell.setValueAt(""+"\r\n",i,0);
        dasModell.setValueAt(""+"\r\n",i,1);
   }
}
```

Entwurfsmuster MVC

Guido Krüger schreibt in GoToJava2[1] zu den Neuerungen, die in Java durch Swing eingeführt wurden u.a.

„Das Model-View-Controller-Prinzip

Neben den äußerlichen Qualitäten wurde auch die Architektur des Gesamtsystems verbessert. Wichtigste "Errungenschaft" ist dabei das Model-View-Controller-Prinzip (kurz MVC genannt), dessen Struktur sich wie ein roter Faden durch den Aufbau der Dialogelemente zieht. Anstatt den gesamten Code in eine einzelne Klasse zu packen, werden beim MVC-Konzept drei unterschiedliche Bestandteile eines grafischen Elements sorgsam unterschieden:

- *Das **Modell** enthält die Daten des Dialogelements und speichert seinen Zustand.*
- *Der **View** ist für die grafische Darstellung der Komponente verantwortlich.*
- *Der **Controller** wirkt als Verbindungsglied zwischen beiden. Er empfängt Tastatur- und Mausereignisse und stößt die erforderlichen Maßnahmen zur Änderung von Model und View an.*

Das Modell enthält praktisch die gesamte Verarbeitungslogik der Komponente. Ein wichtiger Aspekt ist dabei, dass ein Model mehrere Views gleichzeitig haben kann. Damit Veränderungen des Modells in allen Views sichtbar werden, wird ein Benachrichtigungsmechanismus implementiert, mit dem das Modell die zugeordneten Views über Änderungen informiert. Diese Vorgehensweise entspricht dem Observer-Pattern[2], wie es in Abschnitt 10.3.9 erläutert wurde.

Bei den Swing-Dialogelementen wird eine vereinfachte Variante von MVC verwendet, die auch als Model-Delegate-Prinzip bezeichnet wird. Hierbei wird die Funktionalität von View und Controller in einem UI Delegate zusammengefaßt. Dadurch wird einerseits die Komplexität reduziert (oft ist der Controller so einfach strukturiert, dass es sich nicht lohnt, ihn separat zu betrachten) und andererseits die in der Praxis mitunter unhandliche Trennung zwischen View und Controller aufgehoben[3]. Es kann allerdings sein, dass ein Dialogelement mehr als ein Model besitzt. So haben beispielsweise Listen und Tabellen[4] neben ihrem eigentlichen Datenmodell ein weiteres, das nur für die Selektion von Datenelementen zuständig ist.“

1 Guido Krüger; GoToJava2; Addison-Wesley; es gibt inzwischen eine ganz neue Auflage
2 Das Observer – Pattern, deutsch Beobachter – Muster, ist ein weiteres Entwursmuster der OO
3 Man könnte also betonen: Wichtig ist es, weiterhin nach Möglichkeit die Trennung dieser zusammengefassten Komponente von der Modellkomponente einzuhalten.
4 Siehe AbstractTabelModel