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
```

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.

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

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.
Eine Komponentengruppe, die nicht dem ContentPane zugeordnet ist und dennoch Ereignis – gesteuert ist, sind die Menüs. Dazu verwendet man zunächst (genau) eine JMenuBar, der man mehrere Objekte vom Typ JMenu und denen wiederum Objekte vom Typ JMenuItem hinzufügt. (s.u.)

Ereignisbehandlung (event handling)

Ein so hinzugefügtes Menü hat zunächst keine Wirkung! Das muss man durch die Methoden zur Ereignisbehandlung ändern. Beispiele von Ereignissen sind ActionEvents bei einem angeklickten Button, MouseEvents und auch das Fenster kennt ein Ereignis, WindowEvent, das z.B. beim Schließen ausgelöst wird. Objekte, die auf Ereignisse reagieren sollen, sind dafür verantwortlich, selbst oder mit Hilfe anderer die Behandlung auszulösen. Dazu muss das Auftreten von Ereignissen beobachtet werden und die notwendigen Methoden sind in interfaces, den Listenern, ausformuliert. Ein Objekt „ist ein Listener“, wenn es die in diesem interface definierten Methoden implementiert.

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);`

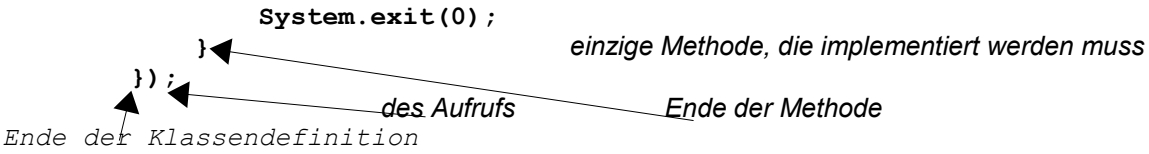
In neueren JAVA-Versionen reicht add ohne Angabe der ContentPane, da der JFrame diese Methode bereitstellt und die Auffordrung an die ContentPane weiterreicht.

ActionListener

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

Aufruf des (Standard-)Konstruktors der *hier beginnenden Klassendefinition*

```
endeButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
```



Der ActionListener führt bei der vom JavaEditor vorgegebenen Variante 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.

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 zunächst selbst dann ein Stringobjekt, wenn der Benutzer Zahlen in das Textfeld eingegeben hat. Da der Benutzer diese Zahlen in der Regel nicht als Text verarbeitet haben möchte, muss eine Möglichkeit geben, diese Zeichenfolgen mit dem Sinn von Zahlen in die entsprechende Zahl zu verwandeln. Dazu stellen die zugehörigen Wrapper-Klassen Integer und Double die entsprechenden Klassenmethoden bereit:

```
Double.parseDouble("3.45“);
```

liefert als Wert die double – Zahl 3.45.

```
Integer.parseInt("123") ;
```

liefert als Wert die int – Zahl 123 zurück. Hier gibt es sogar auch noch die Möglichkeit, andere Zahlenbasen zu verwenden, beispielsweise Hexadezimal:

```
Integer.parseInt("-FF", 16) ;
```

liefert als Wert die int – Zahl -255 zurück.

Listner zu JTextField

Auch einem JTextField können Listener hinzugefügt werden. Bei einem JTextField sind mehrere Aktionen denkbar, nicht nur einfaches Anklicken wie bei einem Button. Es kann beispielsweise ein KeyListener sein, der zur Verarbeitung von Tastaturereignissen dient. Die folgenden Programmzeilen fügen dem Textfeld jTextField1 einen KeyListerner hinzu, der das Drücken einer Taste "beobachtet", also darauf reagiert.

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.

```
jTextField1.addKeyListener(new KeyAdapter(){
  public void keyPressed(KeyEvent e){
    jTextField2.setText("Taste: "
              +KeyEvent.getKeyText(e.getKeyCode()));
  }
});
```

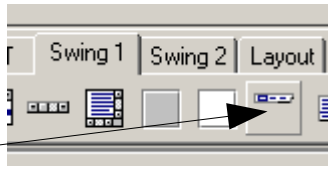
Listener und Adapter

Bemerkenswert ist außer der schon beim ActionListener erläuterten anoymen inneren Klasse, die auch hier verwendet wird, dass wir hinter new nicht KeyListener als Klassennamen angegeben finden.
Das geht zwar prinzipiell, zwingt den Entwickler aber dazu, alle im interface KeyListener geforderten Methoden zu implementieren. Im interface KeyListener ist nämlich nur die Schnittstelle einer solchen Klasse beschrieben, also alles das, was ein KeyListener können muss, wenn er sich als solcher bezeichnen will. Sieht man in der Dokumentation nach, findet man, dass er die Methoden

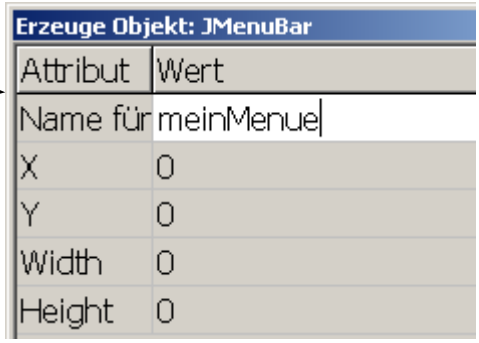
```
public void keyTyped(KeyEvent e) ,
public void keyPressed(KeyEvent e) und
public void keyReleased(KeyEvent e)
```

bereitstellen muss.
Tatsächlich ist in dem o.a. Programmcode aber nur eine der Methoden angegeben und es stellt sich die Frage, wo denn die andern beiden Methoden zu finden sind.
Die Antwort gibt die tatsächlich erzeugte Klasse, nämlich die Adapterklasse KeyAdapter. In dieser Klasse sind die drei Methoden leer implementiert und man braucht für einen zulässigen KeyListener nur die Methoden durch eine eigene Definition zu überschreiben, die man wirklich bereit stellen will. Unser KeyListerner behandelt also tatsächlich nur die Methode keyPressed(...), während die anderen beiden einfach leer sind, also nichts tun. Java erspart uns auf diese Art, Methoden implementieren zu müssen, die gar nicht gebraucht werden.

Menüs



Der Javaeditor stellt eine 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 einfügen können.

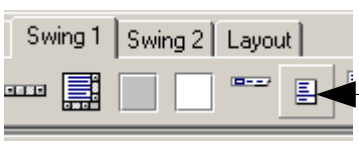


Es werden dann eingefügt

- als Deklaration
  ```
  private JMenuBar meinMenue
  = new JMenuBar();
  ```
  und
- als Definition `setJMenuBar(jmb);`

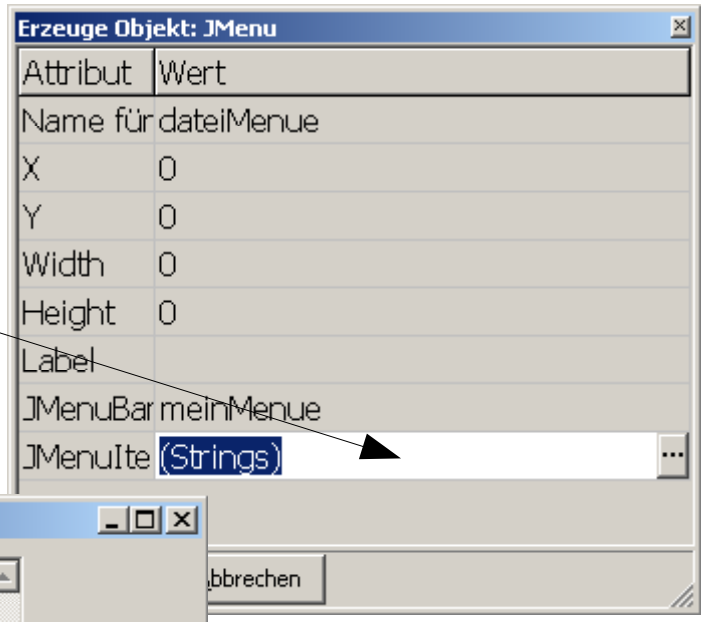
im Konstruktor von von unserer Gui-Klasse.

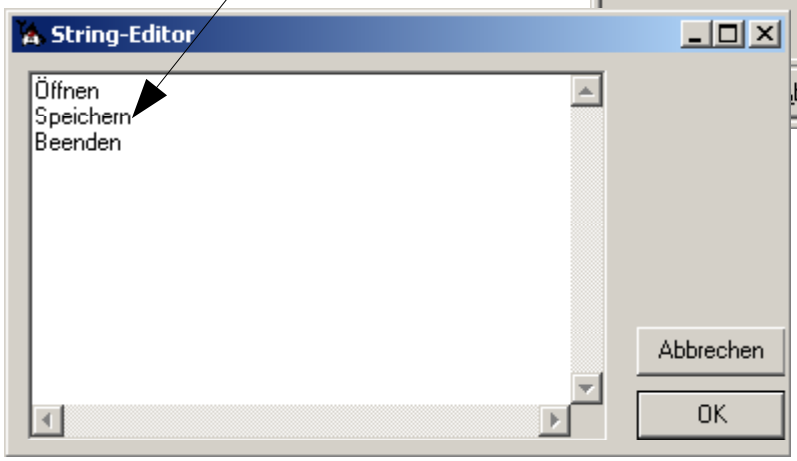
Diese Methode ist also auch wieder eine Methode, die von JFrame bereit gestellt wird.



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 meinMenue). Dann wird im

Programmcode die richtige Verbindung realisiert.
Die einzelnen Menüpunkte sind dem Menü dann per Hand oder durch Einfügen über den Objektinspektor hinzuzufügen. Sie sind Instanzen der Klasse JMenuItem.
Sinnvollerweise macht man das schon mit dem Objektinspektor, indem man die drei Punkte rechts anklickt. Dadurch öffnet sich ein Dialogfenster, in das man zeilenweise die Menüpunkte eintragen kann.





Der Vorteil dieser Vorgehensweise ist, dass nun nicht nur die Objekte angelegt werden, sondern auch schon die notwendigen ActionListener zur Behandlung des Anklickens der Menüpunkte. Passend zum angegebenen Beispiel finden wir bei den Attributen im Kopf:

```
private JMenuBar meinMenue = new JMenuBar();
private JMenu dateiMenue = new JMenu("Datei");
private JMenuItem JMenuItem1 = new JMenuItem("Öffnen");
private JMenuItem JMenuItem2 = new JMenuItem("Speichern");
private JMenuItem JMenuItem3 = new JMenuItem("Beenden");
```

Im Konstruktor muss dann stehen:

```
setJMenuBar(meinMenue);
meinMenue.add(dateiMenue);
dateiMenue.add(JMenuItem1);
JMenuItem1.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    JMenuItem1_ActionPerformed(evt);
  }
});

dateiMenue.add(JMenuItem2);
JMenuItem2.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    JMenuItem2_ActionPerformed(evt);
  }
});

dateiMenue.add(JMenuItem3);
JMenuItem3.addActionListener(new ActionListener() {
```

```
    public void actionPerformed(ActionEvent evt) {
      JMenuItem3_ActionPerformed(evt);
    }
  });
```

Selbstverständlich sind dann die eigentlichen Ereignismethoden noch zu schreiben.

Eine TextArea mit Scollbalken

Eine JTextArea ist geeignet, um mehrzeiligen Text darzustellen. Dabei kann es passieren, dass auch dieses Fenster nicht zur Darstellung des gesamten Textes ausreicht. Man möchte dann gern ein Scrollen des Textes ermöglichen.
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);    (bzw. einfach add(scroll);)
```

Inhalte werden dann einzeln in neue Zeilen gebracht mit:

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

return und newline

Eine Buttongroup für JRadioButtons

Wir bauen eventuell die Auswahl des neu einzufügenden Möbeltyps mit einer Gruppe von JRadioButtons ein. Eine Gruppe bedeutet, dass aus dieser Gruppe maximal einer ausgewählt sein kann. Dies gelingt mit folgenden Deklarationen und Definitionen:

```
private JPanel jButtonGroup1Panel = new JPanel();
private ButtonGroup jButtonGroup1 = new ButtonGroup();
private JRadioButton jRadioButtonStuhl = new JRadioButton("Stuhl");
  ...
```

weiter unten dann:

```
jButtonGroup1Panel.setBounds(16, 56, 185, 129);
jButtonGroup1Panel.setLayout(new GridLayout(5, 1));
// Stuhl
jRadioButtonStuhl.setActionCommand("Stuhl");
jButtonGroup1.add(jRadioButtonStuhl);
jButtonGroup1Panel.add(jRadioButtonStuhl);
// Tisch
...
// Auswahl erzwingen
jRadioButtonStuhl.setSelected(true);
cp.add(jButtonGroup1Panel);
...
```

weiter unten dann die Behandlung der Auswahl:

```
ButtonModel selected = jButtonGroup1.getSelection();
if (selected.getActionCommand().equals("Stuhl"))
   <das, was dann passieren soll>
```

Jeweils auch entsprechend für die weiteren Möbeltypen.

1 Siehe auch das Vorgehen bei der Tabelle.

GUI und Tabellen

Ein etwas schwierigeres Problem ist das Realisieren von Tabellen, das deswegen hier ausführlicher gesondert erläutert wird.
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.
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);
        }
    }
```

JList

Etwas einfacher als Tabellen, aber immer noch nicht ganz ohne: Dateneingabe und Zugriffssteuerung bei einer JList.
Auch in diesem Fall:

- Sinnvollerweise packt man die JList in eine JScrollPane, damit bei zu vielen Zeilen automatisch das Scrollen eingeschaltet wird.
- Mit der JList ist eine Datenstruktur verknüpft, die ihr beim Erzeugen übergeben werden muss.
- Dafür gibt es hier zwei Varianten, einmal ein Array, als Alternative aber auch einen echten Sammlungstyp, Vector, der sich ähnlich verhält wie eine ArrayList, insbesondere aber auch die entsprechenden Zugriffsmethoden wie add(Object obj) bereitstellt.

Möglich also:

```
private Vector jList1Daten = new Vector();                    // leer erzeugt
private JList jList1 = new JList(jList1Daten);                // verknüpft
private JScrollPane jScrollPane = new JScrollPane(jList1);    // scrollen
```

Fügt man der JList einen Eintrag hinzu, dann kann man das über die add-Methode des Vectorobjektes machen:

```
jList1Daten.add("der neue Eintrag");
jList1.updateUI();
```

Die zweite Zeile ist notwendig, damit die JList die Änderung auch übernimmt, also die Darstellung aktualisiert.
Für das Auslesen von Inhalten ist es normalerweise notwendig, zunächst gezielt einzelne Elemente zu markieren. Dazu fügen wir -ähnlich wie bei einem Button- einen Listener hinzu:

```
jList1.addListSelectionListener(new ListSelectionListener(){
    public void valueChanged(ListSelectionEvent e){
       // macht nichts, was aber möglich wäre:
       jTextField2.setText("index: "+jList1.getSelectedIndex());
    }
});
```

Die Gui kann nun auf das Ändern der Auswahl reagieren; JList stellt auch noch weitere Methoden bereit. Eine ganz wichtige verwenden wir beim Auslesen:

```
jTextField1.setText((String)jList1.getSelectedValue());
```

überträgt beispielsweise den Inhalt der (ersten) ausgewählten Zeile in ein Textfeld.

Mit diesen Hilfen sollte eine Gui für den Raumplaner zu erstellen sein, bei der die Objekte in der JList angezeigt und ausgewählt werden können.

Sichern in eine Datei

Unsere Gui-Elemente halten die Daten nur temporär, also nur solange das Programm läuft. Wollen wir die Daten dauerhaft sichern, beispielsweise auf der Festplatte, dann brauchen wir einen anderen Dateityp. Auch dafür stellt Java natürlich Werkzeuge bereit. Mit dem Javaeditor können wir per Mausklick die „FileChooser“ einfügen und zwar sowohl für den Öffnen-Dialog als auch für den Speichern-Dialog. Klicken wir beide an, ohne die Vorgaben im Dialogfenster zu verändern, dann finden wir zwei Attribute

```
private JFileChooser jfco = new JFileChooser();
private JFileChooser jfcs = new JfileChooser();
```

und zwei fast gleiche Methoden für open (save entsprechend):

```
public String jfcoOpenFilename() {
  jfco.setDialogTitle("Öffne Datei");
  if (jfco.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
    return jfco.getSelectedFile().getPath();
  } else {
    return null;
  }
}
```

Damit ist es aber noch nicht getan und das ist auch sinnvoll, denn dies ist der Teil, welcher der View-Komponente zugeordnet ist.

Eine Extraklasse für die Textdatei

Die tatsächlichen Schreib- und Lesebefehle sollten wieder in der Klasse Datei zu finden sein. Hier wird es nun etwas komplizierter:

```
/**
 * Der in der ArrayList vorhandene Text wird in die Datei geschrieben.
 * ACHTUNG: bestehende Dateien werden überschrieben !!!
 */
public void dateiSchreiben(){
  try {
      PrintWriter ausgabe
      = new PrintWriter(new BufferedWriter(new FileWriter(dateiName)));
      for (int i=0; i<text.size();i++) {
          ausgabe.write(text.get(i));
          if (i<text.size()-1) ausgabe.write("\n");
      }
      ausgabe.close();
  } catch (IOException e){
      System.out.println("Fehler beim Erstellen der Datei !");
  }
}
```

Da beim schreibenden und lesenden Zugriff immer mit Fehlern gerechnet werden muss, sollten wir diese mit einem try – catch – Block abfangen. Ob das Abfangen, wie oben angegeben, allein in einer Konsolenmeldung besteht oder besser ein Dialogfenster (showDialog...) bemüht wird, sei dahin gestellt.

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], ...

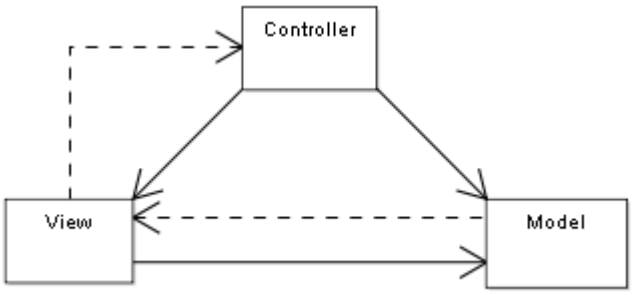
Bei den Swing-Dialogelementen wird eine vereinfachte Variante von MVC verwendet, die auch als Model-Delegate-Prinzip bezeichnet wird. Hierbei wird die Funktionalität von View und Controller in einem UI Delegate zusammengefaßt. Dadurch wird einerseits die Komplexität reduziert (oft ist der Controller 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.“

MVC nicht nur bei Swing

Das Entwurfsmuster MVC ist keine Erfindung von SUN oder den Java-Entwicklern, sondern entstammt schon Smalltalk, der ersten „reinen“ objekt-orientierten Sprache. Grafisch veranschaulicht[5] bestehen folgende Beziehungen zwischen diesen "logischen Baugruppen":

1 Guido Krüger...; Handbuch der Java-Programmierung; Addison-Wesley, 2007, ISBN 3-8273-2373-8
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
5 Bildquelle. Wikipedia (public domain)
Die folgende Erläuterung ist der dort abgedruckten entlehnt.

Die Komponenten



Das Modell [englisch: model]

Das Modell enthält die eigentliche inhaltliche Problemlösung. Dabei ist es oft nicht allein in der Lage, die mit der Anwendung beabsichtigten Geschäftsprozesse zu steuern, sondern muss das ggf. dem Controller überlassen. Wichtig ist aber, dass [siehe das Bild] das Modell von Präsentation und Steuerung unabhängig ist. Es könnte also [und das kennen wir so von BlueJ] für sich allein arbeiten, allerdings kann dann davon keiner etwas "wissen". Ändert sich sein Zustand, müssen View (und Controller) informiert werden.

Präsentation [englisch: view]

Die view-Komponente hat zwei Aufgaben[1]:

- Darstellung der benötigten Daten aus dem Modell und
- Entgegennahme von Benutzerinteraktionen.

Wichtig ist:

1. Sie kennt sowohl ihre Steuerung [um sie über Anforderungen zu informieren] als auch das Modell [um es über Inhalte zu informieren].
2. Sie ist nicht selbst für die Weiterverarbeitung der vom Benutzer übergebenen Daten verantwortlich.

Die view-Komponente ist in der Lage, sich Daten aus dem Modell zu besorgen. Bei für sie relevanten Änderungen von Daten im Modell wird sie informiert und besorgt sich selbst daraufhin die aktualisierten Daten. Das ist eine Anwendung des Beobachter–Entwurfsmusters [englisch: Observer-Pattern].

Steuerung [englisch: controller]

Die view-Komponente steuert die Bearbeitung der Anforderungen, sie arbeitet also selbst nur auf Logikebene, nicht aber auf der Datenebene. Die Steuerung entscheidet , welche Daten im Modell geändert werden müssen und fordert das Modell dazu auf. In der Regel wird das Modell danach die views über die Änderungen informieren.

1 In vielen Fällen gibt es mehrere views.