JAVA-Swing-Kurs

Begleitmaterial

Paul Fischer

Lehrstuhl Informtik 2, Universität Dortmund
WS 2000/01 2. – 6. April 2001

Version 1.00

Dortmund, den 1. April 2001

JAVA-Swing-Kurs

Version 1.00 vom 1. April 2001
Dieses Material wurde mit LaTeX2ε erstellt

Inhaltsverzeichnis

Kapitel 1

Allgemeines

Dieser Kurs richtet sich an Anfänger in der Graphik-Programmierung in JAVA. Der Kurs behandelt nur wenige, grundlegende graphische Komponenten. Er soll die Fähigkeit vermitteln, eine einfache graphische Benutzeroberfläche zu erstellen. Vorausgesetzt wird die Kenntnis der wichtigsten Konzepte von JAVA (Objektorientierung, Vererbung, Datenkapslung, Umgang mit Packages) und Programmiererfahrung.

Für graphische Oberflächen in JAVA sind vor allem die Bibliotheken AWT und SWING bekannt. Der *Abstract Window Toolkit* (AWT) war die erste solche Bibliothek. Sie stellt die wesentlichen Komponenten zur Gestaltung von graphischen Oberflächen zur Verfügung. Die Erstellung komplexerer Oberflächen ist mit AWT zwar möglich, aber nicht einfach. Die ersten Versionen von AWT waren darüberhinaus extrem fehlerhaft

Die in der Swing-Bibliothek enthaltenen Komponenten weisen eine größere Funktionalität auf, sind einfacher zu handhaben und weit besser implementiert als die von AWT.

In diesem Kurs werden die wichtigsten Komponenten von Swing vorgestellt. Dazu gehören *Frames* (Fenster) zum Anzeigen von Daten und *Dialoge* zum Austausch von Daten zwischen Programm und Benutzer sowie die untergeordneten *Panels* als eigentliche Anzeigeflächen. Zunächst wird der Aufbau und die Funktionalität dieser Komponenten vorgestellt. Dazu gehört auch das *Layout*, das bestimmt wie einzelne Komponenten arrangiert werden.

Die Beispielprogramme sind unabhängig von einer speziellen Entwicklungsumgebung erstellt worden und können von der Kommandozeile aus compiliert und gestartet

werden. Sie sind in einzelne Packages im Haupt-Package `kurs` aufgeteilt. Die Verzeichnisstruktur sieht damit so aus

```
Unix/Linux:     kurs/[packageName]/[dateiName].java
MS-Windows:     kurs\[packageName]\[dateiName].java
```

Zum Compiliernen beziehungsweise Übersetzen genügt es, die folgenden Befehle im Oberverzeichnis von `kurs` einzugeben-

```
javac kurs\[packageName]\[dateiName].java
java  kurs\[packageName]\[dateiName]
```

Alle Beispielprogramme sind bewusst kurz und einfach gehalten, um nicht von den Graphik-Konzepten abzulenken. Das heisst auch, dass manchmal auf sinnvolle Sicherheits- und Plausibilitätsabfragen verziechtet wurde. Ebenso wurde auf die Verwendung von eigenen Exceptions verzichtet.

Bei einer Reihe von Komponenten und Methoden werden nicht alle mölichen Einstellungen und Varianten vorgestellt, sondern nur die grundlegenden. Ein Blick in die Swing-Dokumentation oder andere Literatur ist also ratsam.

1.1 Literatur

Die folgende Liste nennt nur einige Bücher aus der Unmenge der Literatur zu Swing. Bevor man sich ein Buch kauft sollte man es genau prüfen; es gibt sehr viele schlechte Bücher zu JAVA. Weiterhin sind (meist englische) Originalausgaben den zum Teil grauenvollen und teureren Übersetzungen vorzuziehen.

1. Die Swing-Referenz von SUN. Swing-Beschreibung von den Machern von Swing. Ist in vielen Entwicklungsumgebungen schon enthalten. Kann über die folgende URLs

   `http://java.sun.com/j2se/1.3/docs/api/javax/swing/package-summary.html`
   `http://java.sun.com/j2se/1.3/docs.html`

erreicht beziehungsweise heruntergeladen werden.

2. Das Tutorial von Swing zur Entwicklung von graphischen Benutzeroberflächen unter Swing

   `http://java.sun.com/docs/books/tutorial/uiswing/index.html`

3. C. S. Horstmann, G. Cornell, Core Java, Band 1 – Grundlagen, 880 Seiten, Prentice Hall/Markt&Technik Verlag, 1999. Gutes einführendes Buch in Java mit recht großem Swing-Teil. Bei Band 1 ist auch die deusche Übersetzng gelungen.

4. C. S. Horstmann, G. Cornell, Core Java, Band 2 – Advanced Features, 924 Seiten. Prentice Hall, 2000. Stellt weiterführende Konzepte von Java und Swing vor.

5. D. M. Geary, Graphic Java, Band 1 – AWT, 932 Seiten, Prentice Hall/Markt&Technik Verlag, 1999. Als Einführung nicht unbedingt geeignet, zum Nachschlagen brauchbar.

6. D. M. Geary, Graphic Java, Band 2 – Swing, 1620 Seiten, Prentice Hall, 1999. Als Einführung nicht geeignet, eigentlich muss man mindestens AWT beherrschen und die Grundlagen von Swing kennen. Zum Nachschlagen für Fortgeschrittenen brauchbar.

Kapitel 2

Grundlagen von Swing

Die Swing-Bibliothek enthält gut 250 Klassen von denen etwa 40 so genannte *Komponenten* (*components*) sind, also Klassen, die auf dem Bildschirm sichtbare Objekte erzeugen. Diese sind von der Swing-Mutterklasse `JComponent` abgeleitet. Die anderen Klassen sind Hilfsklassen, die zum Beispiel der Mausabfrage dienen. Die Swing-Bibliothek wird mittels

```
import javax.swing.*;
```

eingebunden. Man beachte das „`x`“ in `javax` (Java-Extensions). Meistens ist es nötig, auch (Hilfs-)Klassen der AWT-Bibliothek bereitzustellen, was mit

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

geschieht.

Man unterscheidet in Swing zwischen *schwergewichtigen Komponenten* und *leichtgewichtigen Komponenten*. Leichtgewichtige Komponenten können nicht für sich alleine dargestellt werden; sie müssen in eine schwergewichtige oder leichtgewichtige Komponente eingebettet werden. Jede Einbettung von leichtgewichtigen Komponenten ineinander muss also letztlich in einer schwergewichtigen enden. Dabei sind kreisförmige Einbettungen (A in B, B in C, C in A) ebenso verboten, wie die Einbettung einer Komponente in mehrere andere. Die Struktur der Einbettungen entspricht also einem Wurzelbaum mit der schwergewichtigen Komponente an der Wurzel, siehe Abbildung 2.1.

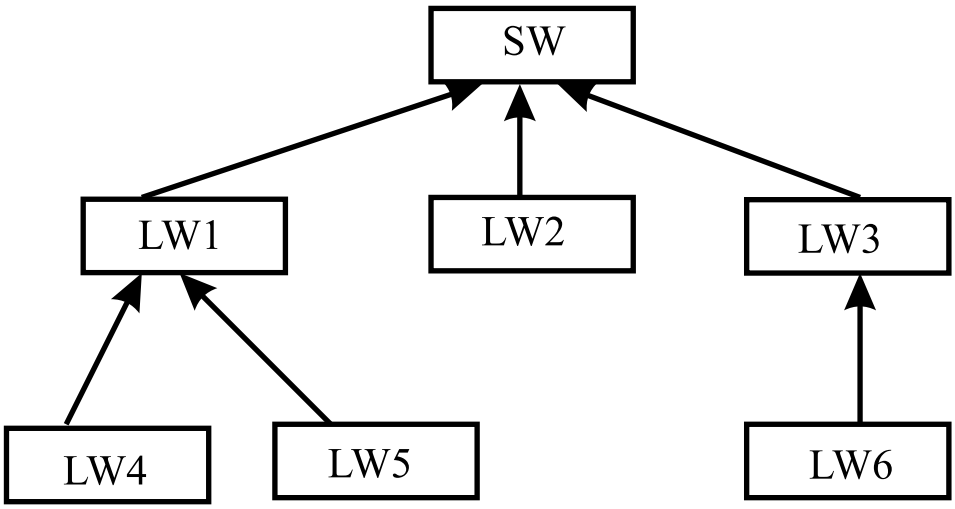

Abbildung 2.1: Struktur der Einbettung von Komponenten als Wurzelbaum. An der Wurzel steht die schwergewichtige Komponente SW, ein Pfeil bedeutet „ist eingebettet in“, zum Beispiel die leichtgewichtige Komponente LW5 ist in LW1 eingebettet.

Die beiden Schwergewichte, die in diesem Kurs behandelt werden, sind der *Rahmen* (frame) und der *Dialog* (dialog). Rahmen entsprechen dem, was man im Allgemeinen *Fenster* nennt. Sie sind die Hauptkomponenten einer graphischen Oberfläche. Sie liefern den Rahmen in den die leichtgewichtigen Komponenten eingebettet werden. Selbst stellen sie keine weiteren Informationen graphisch dar; dies geschieht in den eingebetteten leichtgewichtigen Komponenten. Dialoge dienen der Interaktion mit dem Benutzer und sind meist nur kurz sichtbar, um Information zu liefern oder zu holen. Die beiden anderen Schwergewichte Fenster (windows) (ein etwas irreführender Name) und Applets (applets) behandeln wir hier nicht.

Die Leichtgewichte dienen der eigentlichen Darstellung von Information. Die wichtigste heißt im englischen *Panel*, was man mit *Zeichenfläche* aber auch „Bedienfeld“ übersetzen kann. Dies deutet schon an, dass man Panels sowohl zur Darstellung von Graphik benutzen kann, als auch zur Einbettung von weiteren Bedienelementen. Solche *Bedienelmenete* sind zum Beispiel *Knöpfe* (buttons), *Schieberegler* (slider),
! *Textfelder* (text fields) aber auch Menüs (menus).

Wenn man AWT-Komponenten und Swing-Komponenten mischt, so können überraschende Effekte auftreten. Dies liegt daran, dass die AWT-Gegenstücke mancher leichtgewichtigen Swing-Komponenten schwergewichtig sind. Hier beschränken wir uns aber auf reine Swing-Oberflächen.

Im folgenden werden wir meisten die englischen Begriffe für die Komponenten verwenden, zum Beispiel „Frame“ für „Rahmen“.

Ein Ausrufzeichen im Rand zeigt an, dass hier auf eine mögliche Ursache für Fehlfunktionen oder ungewolltes Verhalten des Programms hingewiesen wird. !

2.1 Frames

Die Klasse, die in Swing die Rahmen definiert heißt `JFrame`. Bei allen Swing-Komponenten wird dem eigentlichen Namen der Buchstabe „`J`“ im Klassennamen vorangestellt. Ein Frame besteht aus einer rechteckigen Fläche, in der sich oben eine *Titelleiste* befindet. Hier werden auch automatisch die Schaltflächen zum Schließen, Verkleinern usw. des Fensters eingefügt. Welche dies sind und wie sie aussehen hängt vom jeweiligen Betriebssystem ab. Darunter ist die Fläche zur Einbettung von Komponenten, die so genannte `contentPane`. Einige betriebssystemabhängige Funktionen wie das Verschieben oder Vergrößern des Frames mit der Maus werden ebenfalls automatisch bereitgestellt. Abbildung 2.2 zeigt den prinzipiellen Aufbau eines Frames.

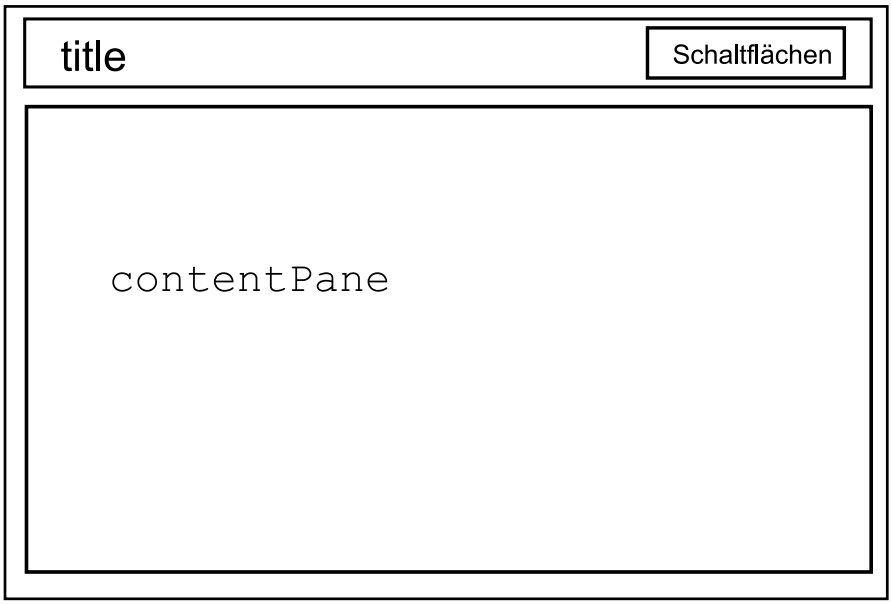

Abbildung 2.2: Der Aufbau eines Frames. In der Titelleiste steht ggf. der im Parameter `titel` des Konstruktors spezifizierte String. Die Lage und das Aussehen der Schaltflächen für Schließen usw. ist betriebssystemabhängig.

Die Konstruktoren der Klasse sind

```
public JFrame();
public JFrame(String title);
```

und erzeugen eine unbetiteltes Fenster beziehungsweise eines mit dem String `title`. Erzeugen heißt hier, dass alle zum Zeichnen des Fensters notwendige Information breitgestellt wird. Das Zeichnen selbst wird aber noch nicht durchgeführt. Dies geschieht durch Aufruf der Methode der Klasse `JFRame`

```
  setVisible(boolean b);\jnidx{setVisible}
```

mit `b = true`. Ein eventuell angezeigter Frame wird mit `setVisible(false)` wieder unsichtbar, wird aber nicht zerstört. Er kann mit `setVisible(true)` jederzeit
! wieder sichtbar gemacht werden (ohne neuen Aufruf des Konstruktors). Dies ist auch zu beachten, wenn man einen Frame mittels der Schaltfläche „Schließen“ in der Titelleiste schließt. Der Frame verschwindet zwar, aber die Anwendung, die ihn angezeigt hat läuft weiter! Um sie zu beenden muss man `System.exit(0);` aufrufen. Damit dies beim Anklicken der Schaltfläche „Schließen“ geschieht, fügen wir dem Rahmen einen *WindowListener* hinzu. Was das genau ist und wie Listener arbeitet wird in Kapitel 4 erklärt. Im Moment nur soviel: der WindowListener lauert auf so genannte *Ereignisse* die der Frame auslösen kann, zum Beispiel Anklicken von Schaltflächen oder Menüs. Erkennt er ein solches Ereignis, reagiert er darauf mit dem vorgesehenen Programmteil. Hier lauert er auf das Anklicken der Schaltfläche
! „Schließen“ und führt dann `System.exit(0);` aus.

Im Allgemeinen wird man nicht einfach `exit(0)` aufrufen sondern vorher noch einige Aufräumarbeiten (Dateien schließen, Daten sichern usw.) erledigen. Wenn eine Anwendung mehrere Frames geöffnet hat, will man eventuell nur einige davon schließen und weiterarbeiten und ruft dann `exit(0)` gar nicht auf.

Im folgenden ist ein Programm aufgeführt, dass einen einfachen Frame definiert.

Datei: `SimpleFrame.java`

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

public class SimpleFrame extends JFrame
{

  public SimpleFrame()
```

```
  {
     setSize(200,200);

     addWindowListener(new WindowAdapter()
         { public void windowClosing(WindowEvent e)
          {
            System.exit(0);
          }
         });

  }

  public void showIt(){
    setVisible(true);
  }//method

  public void showIt(String title){
    setTitle(title);
    setVisible(true);
  }//method

  public void showIt(String title,int x, int y){
    setTitle(title);
    setLocation(x,y);
    setVisible(true);
  }//method

  public void hideIt(){
    setVisible(false);
  }//method
}//class
```

Die Klasse `SimpleFrame` ist von `JFrame` abgeleitet, erbt als deren Funktionalität. Zusätzlich besitzt jeder `SimpleFrame` einen WindowListener der für die Terminierung des aufrufenden Programms sorgt, wenn man die Schaltfläche „Schließen“ anklickt. Es enthält den Konstruktor und Methoden

```
public SimpleFrame()
public void showIt()
public void showIt(String title)
public void showIt(String title,int x, int y)
public void hideIt()
```

Der Konstruktor ist nur um den WindowListener erweitert und den Befehl `setSize(200,200);`, der den Frame auf 200 × 200 Pixel Größe setzt (Außenmaß). Ein *Pixel* ist ein Bild-
! punkt auf dem Bildschirm. Ohne diesen Befehl wäre nur die Titelleiste zu sehen. Die drei `showIt`-Methoden zeigen der Frame ohne Titel, mit Titel bzw. mit Titel und an einer vorgegeben Position an. Dies geschieht mit der -Methode. Die beiden Ganzzahl-Parameter x und y spezifizieren die horizontale beziehungsweise vertika-
! le Position der linken oberen Ecke des Frames. Das JAVA-Koordinatensystem ist „steht auf dem Kopf“, das heißt die positive Richtung der y-Achse zeigt abwärts, siehe Abbildung 2.3. Dies gilt auch für die Koordinatensystem in untergeordneten Komponenten.

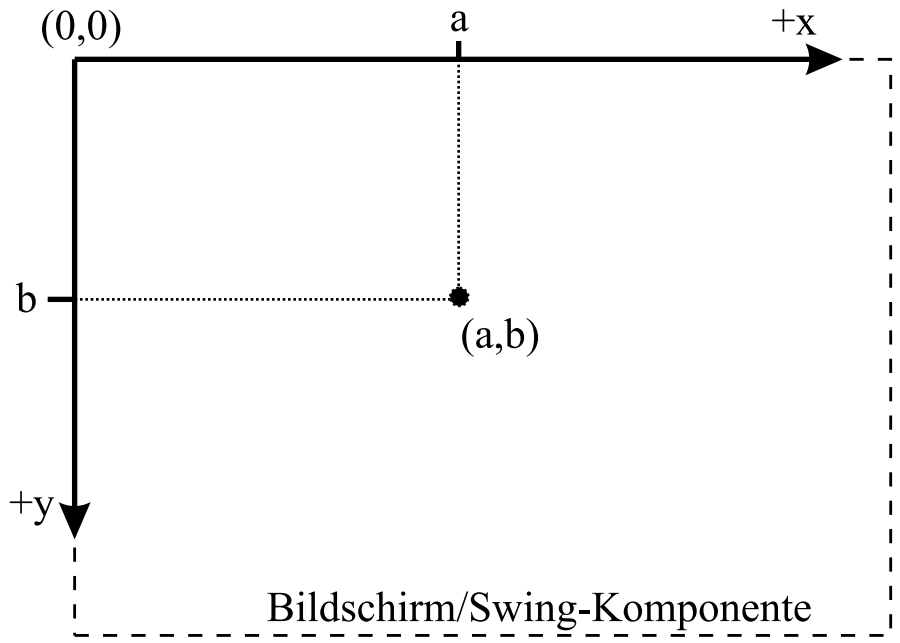

Abbildung 2.3: Das JAVA-Koordinatensystem ist steht auf dem Kopf. Der Ursprung $(0, 0)$ liegt in der linken oberen Ecke des Bildschirms beziehungsweise einer Komponente. Die positive Richtung der x-Achse zeigt nach rechts, die positive Richtung der y-Achse zeigt unten.

Um den Frame anzuzeigen, verwenden wir das folgende Programm. Es erzeugt zwei `SimpleFrames` und zeigt sie an. Der erste wird nur mit einem Titel versehen (zweiter Konstruktor) der zweite obendrein noch um je 150 Pixel nach rechts und unten verschoben.

Datei: `SimpleFrameTest.java`

```
package kurs.SimpleFrame;
```

```
public class SimpleFrameTest
{

  public SimpleFrameTest()
  {
    SimpleFrame SF1 = new SimpleFrame();
    SimpleFrame SF2 = new SimpleFrame();
    SF1.showIt("SimpleFrame 1");
    SF2.showIt("SimpleFrame 2",150,150);
  }

  public static void main(String[] args)
  {
    SimpleFrameTest simpleFrameTest1 = new SimpleFrameTest();
  }
}
```

Das Ergebnis sollte ungefähr so aussehen wie in Abbildung 2.4. Diese Bilder sind unter Microsoft Windows 98 entstanden. Die Farben und Schriftarten der Titelleiste entsprechen den Voreinstellungen, die unter Windows vorgenommen wurden.

Aufgabe 1: Führe die gerade beschriebenen Programme aus. Was passiert, wenn Du auf die „Schließen“-Schaltfläche eines der Fenster klickst und wie ist das zu erklären.

Abbildung 2.4: Ergebnis des oben beschriebenen Programms `SimpleFrameTest`.

Kapitel 3

Panels und Layouts

Panels sind rechteckige Komponenten, die auf zwei Arten benutzt werden: Einmal als *Zeichenfläche* (*canvas*) auf der man Graphiken darstellt und andererseits als *Behälter* (*container*) in die weitere Komponenten eingebettet werden. Die Klasse `JPanel` implementiert die Panels in Swing.

Im folgenden Programm wird die Klasse `ColorPanel` von JPanel abgeleitet. Sie erlaubt in ihren drei Konstruktoren eine farbiges Panel, ein farbiges Panel einer vorgegebenen Breite beziehungsweise ein farbiges Panel mit vorgegebener Breite und Höhe zu erzeugen.

Datei: `ColorPanel.java`

```
package kurs.SimpleFrameWithPanels;

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


public class ColorPanel extends JPanel
{
  /** Ein JPanel mit der Hintergrundfarbe col . */
  public ColorPanel(Color col)
  {
    setBackground(col);
  }

  /** Ein JPanel mit der Hintergrundfarbe col
```

```
   *  und der Breite width .*/
  public ColorPanel(Color col,int width)
  {
    this.setPreferredSize(new Dimension(width,this.getHeight()));
    setBackground(col);
  }
 /** Ein JPanel mit der Hintergrundfarbe col
   *  und der Breite width und Höhe height .*/
  public ColorPanel(Color col,int width,int height)
  {
    this.setPreferredSize(new Dimension(width,height));
    this.setBackground(col);
  }

}
```

Die Farbe wird mittels `setBackground(Color c)` gesetzt. Die Klasse `Color` stammt aus der AWT-Bibliothek. Dort sind einige Farben schon vordefiniert, zum Beispiel rot mit `Color.red`. Man auch Farben im RGB-System (Rot-Grün-Blau-System) selbst kreieren. Die Intensitäten der drei Grundfarben müssen aus dem Intervall $[0; 255]$ stammen.

```
float r = 100;
float g = 120;
float b = 100;
Color scheusslich = new Color(r,g,b);
```

Aufgabe 2: Definiere einige Farben im RGB-System, unter anderem schwarz, weiß, cyan, gelb, und margenta.

Aufgabe 3: Suche Literatur, die das RGB-System beschreibt. Finde heraus welche anderen Farbsysteme es noch gibt.

Die Größe der Panels wird mit der Methode `setPreferredSize(Dimension d)` eingestellt. Die Klasse `Dimension` ist auch eine Hilfsklasse der AWT-Bibliothek. Mit `Dimension(w,h)`, wobei $w, h \in \mathbb{N}$, werden die Breite (width) auf w und die Höhe
! (height) auf h gesetzt. Die Dimensionsangaben sind nur **Empfehlungen** für die

Größe einer Komponente. Abhängig von den aktuellen Größen anderer Komponenten können bei der Darstellung abweichende Werte gewählt werden. Diese Werte werden vom später noch zu behandelnden `LayoutManager` zur Laufzeit festgelegt. Dies geschieht auch, wenn die Preferred-Size nicht gesetzt wurde.

Nun wollen wir unsere ColorPanels aber auch sehen. Dazu dient das weiter unten angegebene Programm. Es leitet einen `SimplePanelFrame` vom `SimpleFrame` ab, womit für korrektes Schließen gesorgt wäre. Es werden fünf `ColorPanels` in den Farben weiß, rot, gelb, grün und blau erzeugt, das weiße mit vorgegebener Breite von 50. Anschließend werden die fünf Panels in den Frame eingebettet. Zur Einbettung in Frames ist es notwendig noch anzugeben, dass man in die `contentPane` einbetten möchte. Eine Referenz auf diese erhält man mittels der Methode `getContentPane` von `JFrame`.

Kommen wir nun zur Platzierung der einzelnen Panels in der contentPane. Um die angestrebte Plattformunabhängigkeit von JAVA sicherzustellen, muss man dem System eine gewisse Flexibilität bei der Umsetzung der Graphik erlauben. Der Programmierer beschreibt daher nur die Anordnungsstruktur der Komponenten nicht aber die absoluten Positionen. Er gibt also Regeln vor wie zum Beispiel „Komponente A soll rechts von Komponente B liegen“ und nicht „Komponente A mit Breite b und Höhe h liegt auf der Position (x, y) und Komponente B mit Breite b' und Höhe h' liegt auf der Position (x', y')“. Es ist dann die Aufgabe des *Layout-Managers* der Komponente, in die eingebettet wird, die wirkliche Platzierung vorzunehmen. Allerdings hat der Benutzer die Wahl zwischen mehreren Layout-Managern. !

Ein `JFrame` hat standardmäßig ein so genanntes `BorderLayout`. Es erlaubt die Platzierung einer großen zentralen Komponente und von vier schmalen Komponenten an den Rändern. Die Positionen werden durch die Strings `Center`, `North`, `South`, `East` und `West` spezifiziert. Fehlt eine der Rand-Komponenten, dehnt sich die zentrale entsprechend aus.

Aufgabe 4: Finde heraus, was passiert, wenn die zentrale Komponente fehlt.

Die Zentrale Komponente stellt im Allgemeinen die Hauptinformationen dar. Die „South“-Komponente zum Beispiel eine Statusleiste (zeigt Statusinformation an zum Beispiel Zeilenzahl eines Textes). Die „West“-Komponente wird manchmal für Knöpfe zur Auswahl von Werkzeugen oder Edier-Modi verwendet.

Eingefügt wird eine Komponente KOMP in die ContentPane an Position POS mittels

```
this.getContentPane.add(KOMPONENTE,POS);
```

Im folgenden ist der Code für den `SimplePanelFrame` angegeben und im Anschluss daran der Code für das Startprogramm `SimplePanelFrameTest`, das einen solchen Frame erzeugt und anzeigt.

Datei: `SimplePanelFrame.java`

```
package kurs.SimpleFrameWithPanels;

import java.awt.*;
import javax.swing.JFrame;
import kurs.SimpleFrame.*;

public class SimplePanelFrame extends SimpleFrame
{

  public SimplePanelFrame()
  {
    ColorPanel CPWest   = new ColorPanel(Color.white,50);
    ColorPanel CPEast   = new ColorPanel(Color.red);
    ColorPanel CPNorth  = new ColorPanel(Color.yellow);
    ColorPanel CPSouth  = new ColorPanel(Color.green);
    ColorPanel CPCenter = new ColorPanel(Color.blue);
    this.getContentPane().add(CPWest,"West");
    this.getContentPane().add(CPEast,"East");
    this.getContentPane().add(CPNorth,"North");
    this.getContentPane().add(CPSouth,"South");
    this.getContentPane().add(CPCenter,"Center");
  }//constructor

}//class
```

Datei: `SimplePanelFrameTest.java`

```
package kurs.SimpleFrameWithPanels;

public class SimplePanelFrameTest
{
```

```
  public SimplePanelFrameTest()
  {
   SimplePanelFrame SPF = new SimplePanelFrame();
   SPF.showIt();
  }
  public static void main(String[] args)
  {
    SimplePanelFrameTest simplePanelFrameTest1 = new SimplePanelFrameTest();
  }
}
```

Wir wollen nun noch zwei weitere Layout-Manager kennenlernen, das *Fluss-Layout* (`FlowLayout`) und das *Raster-Layout* (`GridLayout`).

Das FlowLayout ordnet die einzubettenden Komponenten „zeilenweise“ von links nach rechts an. Es respektiert (meistens) die vom Benutzer gewünschten Dimensionen (`preferredSize`). Die Komponenten `K` werden in der Mutterkomponente `M` in der Reihenfolge angeordnet, wie die Aufrufe `M.add(K)` auftreten. Passt eine Komponente nicht mehr in die aktuelle „Zeile“, wird eine neue Zeile begonnen. Die Höhen dieser Zeilen werden zur Laufzeit aus den Höhen der einzubettenden Komponenten so bestimmt, dass keine Überlappungen entstehen. Zwischen den Komponenten einer Zeile und den Zeilen selbst werden Lücken gelassen. Diese lassen sich einstellen. Auch die Positionen der eingebetteten Komponenten in der Zeile (zentriert, am links-/rechtsbündig usw.) lassen sich einstellen.

```
FlowLayout()
Flowlayout(int ausrichtung)
Flowlayout(int ausrichtung, int habst, int vabst)
```

Der erste Konstruktor liefert ein zentriertes Layout mit je 5 Pixeln zwischen den Komponenten einer Zeile und zwischen den Zeilen. Beim zweiten Konstruktor kann man die Ausrichtung der Komponenten in der Zeile festlegen:

`ausrichtung` $\in$ {`FlowLayout.RIGHT`, `FlowLayout.LEFT`, `FlowLayout.CENTER`}

(es gibt weitere Möglichkeiten). Der dritte Konstruktor erlaubt zusätzlich die Angabe eines horizontalen Abstandes *habst* und eines Zeilenabstandes *vabst*. Das Programm `LayoutTest` gibt dazu Beispiele.

Der Layout-Manager wird bei jedem Neuzeichnen der zugehörigen Komponente aktiv; zum Beispiel auch nach einer Größenänderung des Frames. Passen dann mehr !

Komponenten in eine Zeile, wird dies auch realisiert! Der optische Eindruck ändert sich dann. Man probiere dies an den beiden Frames mit Flow-Layout im Programm `LayoutTest` aus.

Das zweite hier beschriebene Layout `GridLayout` ordnet die Komponenten rasterförmig an. Der Mutterkomponente (hier ein Frame) wird ein $z \times s$ Raster überlagert, wobei z die Zeilenzahl und s die Spaltenzahl ist. Alle Zeilen und Spalten sind **gleich hoch** beziehungsweise **breit**.In jede der $z \cdot s$ so entstandene *Zellen* wird eine Komponente eingebettet, und zwar zeilenweise oben beginnend. Gibt es mehr Zellen als einzubettende Komponenten, so versucht der Layout, alle **Zeilen** zu füllen und erzeugt eventuell weniger Spalten als gefordert! Hat man mehr Komponenten als Zellen, so werden Spalten angefügt. Im Grunde genommen wird sie Spaltenanzahl also ignoriert. Setzt man die Zeilenanzahl aber auf 0 und die Spaltenanzahl auf $s > 0$, so gibt es immer genau s Spalten und die Zeilenzahl wird angepasst. Im Programm `LayoutFrame` wird eine 2×4 Raster für nur fünf Komponenten gefordert und man erhält ein 2×3 Raster. Die gewünschten Größen der Panels werden vom Grid-Layout ignoriert; die Panels füllen die Zellen ganz aus.

Der (wichtigste) Konstruktor für GridLayouts erhält die Zeilenzahl z und die Spaltenzahl s als Argumente. Auch hier lassen sich Abstände zwischen den Zeilen (*vabst*) beziehungsweise Spalten (*habst*)einstellen.

```
GridLayout(int z, int s);
setVgap(int vabst);
setHgap(int habst);
```

Die beiden folgenden Codestücke `LayoutFrame` und `LayoutTest` geben Beispiele für die gerade beschriebenen Layouts. Das erste definiert einen Frame, dessen Layout man über einen String einstellen kann: Standard Flow-Layout (`Flow1`), Flow-Layout mit größerem horizontalen und vertikalen Abstand und linksbündiger Ausrichtung (`Flow2`) und ein 2×4 GridLayout (`Girg`). Das Programm `LayoutTest` zeigt je einen Frame von jedem Typ an.

Datei: `LayoutFrame.java`

```
package kurs.Layouts;


import java.awt.*;
import javax.swing.JFrame;
import kurs.SimpleFrameWithPanels.ColorPanel;
```

```
import kurs.SimpleFrame.*;


public class LayoutFrame extends SimpleFrame
{

  public LayoutFrame(String layoutType)
  {

    if (layoutType.equals("Flow1"))
     {
       FlowLayout FLay = new FlowLayout();
       this.getContentPane().setLayout(FLay);
      }
    else
      {
       if (layoutType.equals("Flow2"))
        {
         FlowLayout FLay = new FlowLayout(FlowLayout.LEFT,40,30);
         this.getContentPane().setLayout(FLay);
        }
       else
        {
         if (layoutType.equals("Grid"))
          {
            GridLayout GLay = new GridLayout(2,4);
            this.getContentPane().setLayout(GLay);
          }
         else
          {
            System.out.println("Fehler: Layout-Typ nicht korrekt.");
            System.exit(0);
          }
        }
     }
    ColorPanel CP1 = new ColorPanel(Color.red,30,30);
    ColorPanel CP2 = new ColorPanel(Color.yellow,40,20);
    ColorPanel CP3 = new ColorPanel(Color.green);
    ColorPanel CP4 = new ColorPanel(Color.blue);
    ColorPanel CP5 = new ColorPanel(Color.white,80,20);
    this.getContentPane().add(CP1);
    this.getContentPane().add(CP2);
    this.getContentPane().add(CP3);
    this.getContentPane().add(CP4);
    this.getContentPane().add(CP5);
```

```
  }
}
```

Datei: `LayoutTest.java`

```
package kurs.Layouts;


public class LayoutTest
{
  public LayoutTest()
  {
   LayoutFrame FLF1 = new LayoutFrame("Flow1");
   FLF1.showIt("Flow Layout",60,60);

   LayoutFrame FLF2 = new LayoutFrame("Flow2");
   FLF2.showIt("Flow Layout",300,60);

   LayoutFrame GLF  = new LayoutFrame("Grid");
   GLF.showIt("Grid Layout",540,60);

   }

  public static void main(String[] args)
  {
    LayoutTest flowLayoutTest1 = new LayoutTest();
  }
}
```

Aufgabe 5: Experimentiere mit dem Programm `LayoutTest` und berichte über die Erfahrungen. Möglichkeiten sind: Was passiert bei Fenstervergrößerungen, -verkleinerungen. Füge mehr ColorPanels hinzu, ändere die Größen deren Größen. Ändere die Spalten- und Zeilenzahlen beim Grid-Layout.

Kapitel 4

Textfelder, Label, Knöpfe, Aufpasser und eine erste GUI

In diesem Kapitel wollen wir weitere Komponenten von Swing kennenlernen und eine eine erste *Benutzeroberfläche* (graphical user interface, *GUI*) entwerfen. Unsere GUI soll so aussehen wie in Abbildung 4.1. Ihre Funktionalität lässt sich so beschreiben: Rechts von dem Text „Bitte Text eingeben:“ befindet sich ein edierbares Feld in dem wir Text eingeben und ändern können. Wenn man auf die Schaltfläche „Übernehmen“ klickt, soll dieser Text in der nächsten Zeile wieder holt werden, das heißt er soll rechts von „Eingegeben wurde:“ erscheinen. Gleichzeitig soll der Inhalt des Eingabefeldes gelöscht werden. Wir benutzen zur Realisierung dieser Benutzeroberfläche drei neue Swing-Komponenten, die wir nun kurz erklären:

4.1 Label

Ein *Label* (Etikett trifft die Sache nur halb) dient der Darstellung von Text, den der Benutzer nicht ändern soll und kann. Die Klasse `JLabel` realisiert dies in Swing. Hier einige der Konstruktoren und Methoden. Es sei angemerkt, dass `JLabel` weit mehr Möglichkeiten eröffnet, als hier dargestellt werden können. Insbesondere gibt Methoden zur Auswahl eines Schriftsatzes und zur Darstellung von Graphiken.

```
public JLabel(String beschriftung)

public String getText()
```

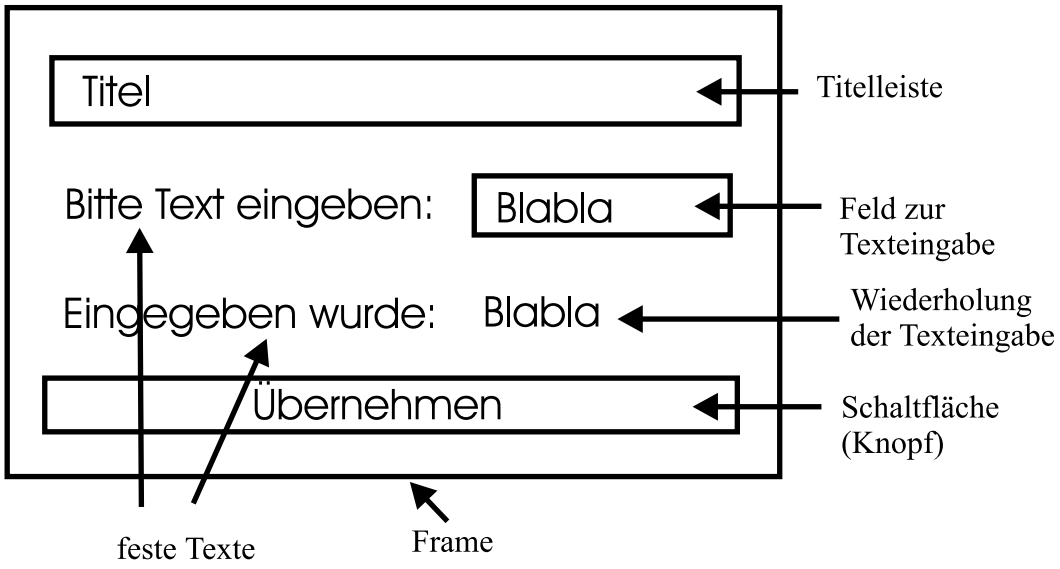

Abbildung 4.1: Unsere erste Benutzeroberfläche.

```
public void setText(String beschriftung)
public void setForeground(Color c)
public void setBackground(Color c)
public void setOpaque(boolean b)
```

Der Konstruktor erzeugt eine Label mit dem Text `beschriftung`. Die Methode `setText(String beschriftung)` setzt den Text eines existierenden Labels entsprechend. Mit `getText` kann man den Text eines Labels abfragen. Die beiden anderen
! set-Methoden ändern die Farbe der Schrift beziehungsweise des Hintergrundes. Zur Änderung der Hintergrundfarbe muss das Label undurchsichtig sein! Dies erreicht man mittels `setOpaque(true)`. Anderenfalls sieht man unabhängig von der gesetzten Hintergrundfarbe des Labels die Farbe der unterliegenden Komponente.

4.2 Textfelder

Textfelder zeigen eine einzelne Zeile Text an und erlauben, diesen zu edieren. Die Klasse `JTextField` implementiert dies in Swing. Auch diese Klasse ist sehr mächtig, so dass wir nur einige Aspekte davon besprechen können. Die Funktionen zur Edierung werden automatisch bereitgestellt. Es genügt, ein Textfeld anzuklicken und man kann den darin enthaltenen Text mit der Tastatur edieren.

Die Semantik der hier vorgestellten Konstruktoren und Methoden entspricht denen der Klasse `JLabel`. Es ist zu beachten, dass `getText()` nicht notwendigerweise den zuletzt mittels des Konstruktors oder mittels `setText` gesetzten Text zurückliefert, sondern den, der gerade im Textfeld steht und der eventuell vom Benutzer geändert wurde.

```
public JTextField(String anfangsText);

public String getText();

public void setText(String beschriftung);
public void setForeground(Color c);
public void setBackground(Color c);
```

4.3 Knöpfe

Knöpfe, manchmal auch *Schaltflächen* genannt, sind rechteckige Flächen, die beim Anklicken eine Meldung an das Programm geben, auf die es eventuell reagiert. Dazu muss man jedem Knopf einen *Aufpasser* (*Listener*) zuordnen, der auf das Anklicken wartet und dann eine vorher festgelegte Komponente benachrichtigt.

Die Klasse `JButton` ist Swing-Implementierung eines Knopfes. Uns interessiert hier zunächst nur der Konstruktor, der einen String zur Beschriftung des Knopfes mitbekommt, und die Methode, die dem Knopf einen Aufpasser zuordnet.

```
public JBUtton(String beschriftung);
public void addActionListener(ActionListner aufpasser);
```

Im nächsten Abschnitt beschreiben wir den Aufpasser.

4.4 Aufpasser und Ereignisse

Für unseren Knopf Implementieren wir einen `ActionListener`. Dies ist ein **JAVA-Interface** mit nur einer einzigen Methode

```
public void actionPerformed(ActionEvent evt)
```

Das Betriebssystem und die JAVA-Laufzeitumgebung bemerken das *Ereignis* (*event*) „der Knopf wurde angeklickt“, erzeugen ein Objekt vom Typ `ActionEvent` und übergeben es dem ActionListener. Daraufhin wird des `actionPerformed`-Methode aktiv. In der `actionPerformed`-Methode muss also stehen, was beim Anklicken des Knopfes zu geschehen hat. Ein `ActionEvent` enthält Informationen über die Art des Ereignisses und die Quelle, die es ausgelöst hat. Genaueres dazu später.

Die Zuordnung des Aufpassers zu einer Komponente erfolgt mittels der Methode `addActionListener`.

```
KOMPONENTE.addActionListener(AUFPASSER);
```

4.5 Der Zusammenbau der GUI

Die GUI wollen wir nun aus den bisher beschriebenen Komponenten zusammenbauen. Die Explosionszeichnung in Abbildung 4.2 verdeutlicht unseren Plan. In den Frame „kleben“ wir unten (South) eine Schaltfläche, darüber (Center) ein Panel (`TextInputPanel`), das die einzelnen Text-Komponenten aufnimmt. Das Layout dieses Panels ist ein 2×2-Grid-Layout, in das wir die Label und das Textfeld zeilenweise hineinkleben. Das Textfeld ist also die zweite einzufügende Komponente. Wir haben im Layout einen Abstand zwischen den Zeilen und Spalten festgelegt und die Komponenten unterschiedlich gefärbt, um den Aufbau der GUI zu verdeutlichen. Man beachte, dass das Hinzufügen von Komponenten in `JFrame` und `JPanel` unterschiedlich vonstatten geht. Während man im `JFrame` in die `contentPane` einfügt,
! fügt man im `JPanel` direkt ein.

```
im JFrame:  this.getContentPane().add(KOMPONENTE);
im JPanel:  this.add(KOMPONENTE);
```

Um die gewünschte Funktionalität zu erreichen, versehen wir das `TextInputPanel` mit der Methode `eingabeUebernehmen`. Diese Methode schreibt den Text aus dem Textfeld `eingabeFeld` in das Label `anzeige` und verwendet dazu die Methoden `getText` (von `JTextField`) und `setText` (von `JLabel`). Anschließend wird der Inhalt von des `JTextField` mittels `setText()` gelöscht

Die Klasse `UeberListener` implementiert das Interface `ActionListener`. Wie bereits erwähnt, muss man die Methode `actionPerformed` mit Inhalt füllen. In unserem Beispiel ruft diese einfach die Methode `eingabeUebernehmen` des `TextInputPanel`

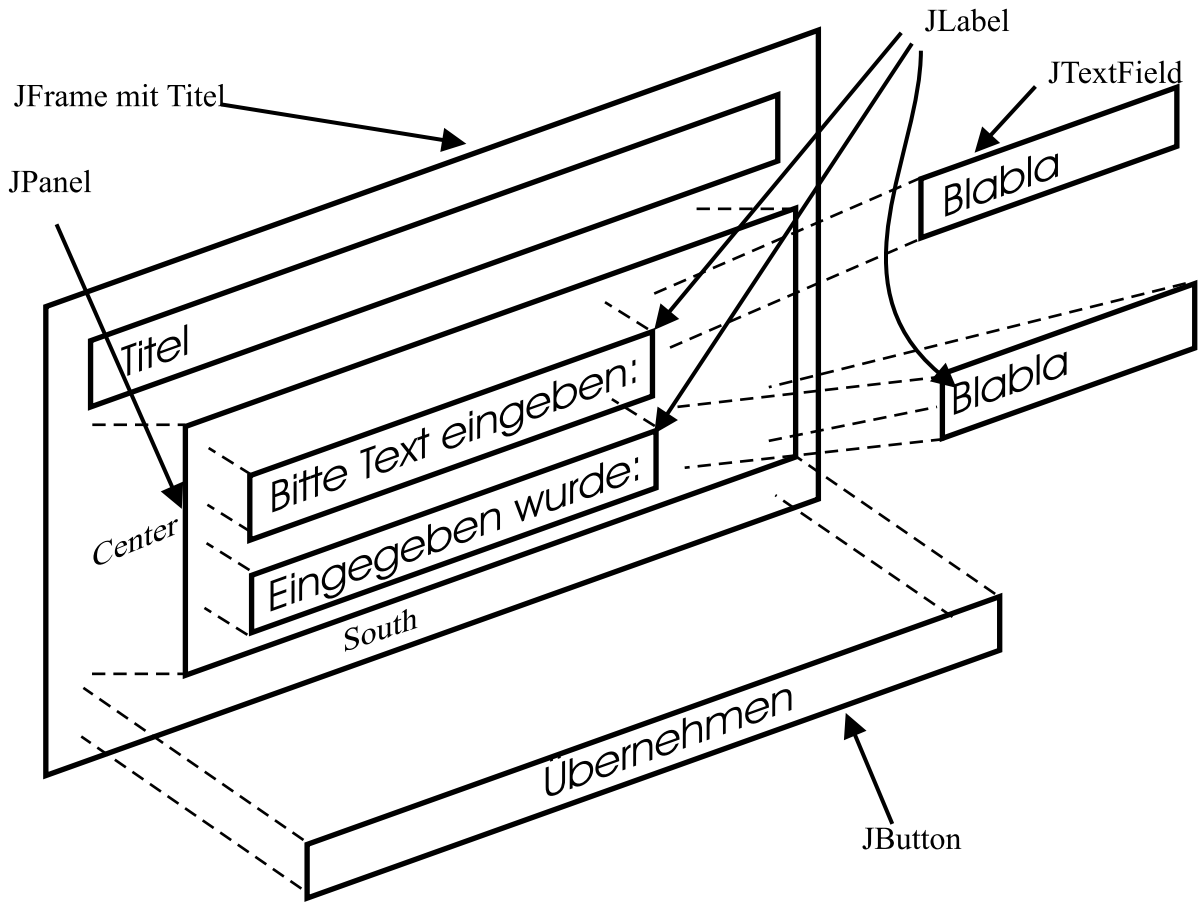

Abbildung 4.2: Der Aufbau der Benutzeroberfläche als Explosionszeichnung.

auf. Damit dem `UeberListerner` das `TextInputPanel` bekannt ist, übergeben wir ihm im Konstruktor eine Referenz darauf.

```
public UeberListener(TextInputPanel TIP)
```

In der Klasse `FirstGUI` erzeugen wir eine Instanz eines UeberListeners und ordnen ihn dem Knopf mit der Inschrift „Übernehmen“ zu.

```
UeberListener Ulist = new UeberListener(TIP);
UeberKnopf.addActionListener(Ulist);
```

Im folgenden findet sich ein Listing der drei Dateien `FirstGUI.java`, `TextInputPanel`, `UeberListener` sowie der Startdatei `FirstGuiTest`.

Datei: FirstGUI.java

```
package kurs.GUI;

import java.awt.*;
import java.awt.event.*;
import javax.swing.JButton;
import kurs.SimpleFrame.SimpleFrame;

public class FirstGUI  extends SimpleFrame
{
  private  TextInputPanel TIP;

  public FirstGUI()
  {
    TIP = new TextInputPanel(Color.yellow);
    this.getContentPane().add(TIP,"Center");
    JButton UeberKnopf = new JButton("Übernehmen");
    UeberKnopf.setBackground(Color.green);
    UeberKnopf.setForeground(Color.red);
    this.getContentPane().add(UeberKnopf,"South");

    UeberListener Ulist = new UeberListener(TIP);

    UeberKnopf.addActionListener(Ulist);
    }
}
```

Datei: TextInputPanel.java

```
package kurs.GUI;

import java.awt.*;
import javax.swing.*;
import kurs.SimpleFrameWithPanels.ColorPanel;


public class TextInputPanel extends ColorPanel
{
  private JLabel anzeige;
  private JTextField eingabeFeld;
```

```
  public TextInputPanel(Color c)
  {
    super(c);
    this.setLayout(new GridLayout(2,2,10,10));
    JLabel bitteLabel = new JLabel("Bitte Text eingeben:");
    JLabel eingLabel  = new JLabel("Eingeben wurde:");
    anzeige           = new JLabel("");
    eingabeFeld       = new JTextField("");

    bitteLabel.setOpaque(true);
    bitteLabel.setBackground(Color.red);
    eingLabel.setOpaque(true);
    eingLabel.setBackground(Color.red);
    anzeige.setOpaque(true);
    anzeige.setBackground(Color.black);
    anzeige.setForeground(Color.white);

    this.add(bitteLabel);
    this.add(eingabeFeld);
    this.add(eingLabel);
    this.add(anzeige);
  }

  public void eingabeUbernehmen()
   {
     anzeige.setText(eingabeFeld.getText());
     eingabeFeld.setText("");
    }

}
```

Datei: `UeberListener.java`

```
package kurs.GUI;

import java.awt.event.*;

public class UeberListener implements ActionListener
{

  private TextInputPanel TIP;
```

```
  public UeberListener(TextInputPanel t)
  {
   TIP = t;
  }

 public void actionPerformed(ActionEvent evt)
   {
     TIP.eingabeUbernehmen();
   }
}
```

Datei: `FirstGUITest.java`

```
package kurs.GUI;

public class FirstGUITest
{

  public FirstGUITest()
  {
    FirstGUI FGUI = new FirstGUI();
    FGUI.setSize(300,120);
    FGUI.showIt("Die erste GUI");
  }
  public static void main(String[] args)
  {
    FirstGUITest firstGUITest1 = new FirstGUITest();
  }
}
```

Es fällt auf, dass die Implementierung des Aufpassers `UeberListener` als eigene Klasse recht viel Aufwand erfordert, wenn man die gering Funktionalität bedenkt. Man implementiert daher solche einfachen Listener oft *anonym* und *on the fly*. Eine solche Implementierung ist dann sinnvoll, wenn der Listener nur in einer Klasse benutzt wird und selbst nicht viel Code enthält. Will man einen Listener in mehreren Klassen verwenden oder führt er selbst komplexes Aufgaben durch, so ist die Implementierung in einer eigenen Klasse zu empfehlen. Eine On-the-fly-Implementierung findet sich im folgenden Listing:

Datei: FirstGUI2.java

```
package kurs.GUI;

import java.awt.*;
import java.awt.event.*;
import javax.swing.JButton;
import kurs.SimpleFrame.SimpleFrame;

public class FirstGUI2 extends SimpleFrame
{
  private  TextInputPanel TIP;

  public FirstGUI2()
  {
    TIP = new TextInputPanel(Color.yellow);
    this.getContentPane().add(TIP,"Center");
    JButton UeberKnopf = new JButton("Übernehmen");
    UeberKnopf.setBackground(Color.green);
    UeberKnopf.setForeground(Color.red);
    this.getContentPane().add(UeberKnopf,"South");

//Anonyme Listener-Implementierung:

    UeberKnopf.addActionListener(new ActionListener()
     {
       public void actionPerformed(ActionEvent evt)
        {
         TIP.eingabeUbernehmen();
        }
     }
    );
  }
}
```

Kapitel 5

Ein einfacher Texteditor

In diesem Kapitel wollen wir einen einfachen Texteditor erstellen. Das Programm soll die folgenden Funktionen besitzen:

- Eine Textdatei unserer Wahl laden können.
- Den Text einer geladenen Datei bearbeiten können.
- Den Text einer geladenen Datei (unter gleichem Namen) speichern können.

Im folgenden beschreiben wir nun zunächst wie man in JAVA Textdateien einliest und schreibt. Dann stellen wir zwei neue Swing-Komponenten vor. Die eine ist ein bereits vordefinierter Dialog zur Dateiauswahl. Die andere zeigt Text an und stellt auch die Funktionen eines Texteditors bereit. Anschließend ist es einfach, den eigentlichen Editor zu erstellen.

5.1 Textdateien lesen und schreiben

Texte in Java bestehen aus Folgen von Zeichen in der *Unicode*-Darstellung. Dieser Zeichensatz umfasst mehr als 16000 Zeichen, darunter auch die des chinesischen und japanischen Zeichensatzes. Jedes Zeichen wird in 2 Byte (16 Bit) dargestellt. Viele Textdateien (besonders solche, die die von anderen Anwendungen erstellt wurden) enthalten Daten im *ASCII*-Format, das nur 256 Zeichen enthält und mit einem Byte (8 Bit) pro Zeichen auskommt. Wenn man nicht aufpaßt werden 2 aufeinanderfolgende ASCII-Zeichen beim Einlesen in ein JAVA-Programm als ein einziges

!

Unicode-Zeichen interpretiert. Der Effekt kann ganz lustig sein, ist meisten aber ungewollt. Wir beschreiben in den folgenden Abschnitten, wie man ASCII-Text in JAVA erfolgreich behandelt.

Die Ein- und Ausgabe ist in JAVA mit dem Konzept der *Datenströme* (*streams*) realisiert. Ein solcher Datenstrom ist eine Folge von Daten (Zeichen oder andere Objekte) aus der Daten entnommen werden können (*Eingabestrom*, *input stream*) oder zu der Daten hinzugefügt werden können (*Ausgabestrom*, *output stream*). Im Allgemeinen wird nicht beim Lesen oder Schreiben eines einzelnen Datums ein Zugriff auf das Speichermedium (Festplatte) erfolgen, sondern es werden ganze Datensätze auf einmal geladen und im schnellen Hauptspeicher verwaltet. Ob und wie das geschieht, ist jedoch abhängig vom Betriebssystem und der Virtual Machine.

Auf die Datenströme kann man so genannte *Filter* aufsetzen, die den speziellen Datentypen des Stromes angepasst sind. Für ASCII-Datenströme enden die Namen dieser Filter auf *Reader* (Eingabe) oder *Writer* (Ausgabe).

Die Bibliothek `java.io` stellt die zur Dateibehandlung notwendigen Klassen und Methoden bereit. Das Einbinden geschieht mittels

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

5.1.1 Die Klasse `File`

Die Klasse `File` realisiert Dateien JAVA. Wir beschreiben nun die von uns verwendeten Konstruktoren und Methoden

```
File(String dateiname)

boolean exists();
boolean delete();
boolean renameTo(File newName);
boolean createNewFile();
```

Der Konstruktor legt ein Dateiobjekt mit dem im String `dateiname` spezifizierten
! Namen an. Es wird weder eine vorhandene Datei (zum Beispiel zum Lesen) geöffnet
! noch eine Datei auf dem Speichermedium angelegt. Wenn man keinen Pfad angibt, wird das aktuelle Arbeitsverzeichnis gewählt, was nicht immer gewünscht ist. Mögliche Dateinamen sind zum Beispiel

```
"testdatei.txt"
"C:/JAVA/kurs/FileIO/scr/testdatei.txt"
"C:\\JAVA\\kurs\\FileIO\\scr\\testdatei.txt"
"~/JAVA/kurs/FileIO/testdatei.txt"
```

Man beachte, dass unter MS-Windows sowohl der normale Bruchstrich (/) als auch ein *Rückstrich* (backslash, \) als Trennzeichen zwischen Verzeichnissen erlaubt ist. Wegen der speziellen Bedeutung des Rückstrichs als ESCAPE-Zeichen in JAVA-Strings muss man ihn allerdings verdoppeln. !

Nun zu den oben angegeben Methoden.

`exists()` liefert `true`, wenn die Datei existiert und `false` sonst.

`delete()` Löscht die Datei, die die `File`-Variable spezifiziert. Liefert bei Erfolg `true` und `false` sonst.

`renameTo(FileNew)` Benennt die in der `File`-Variable spezifizierte Datei in den Namen um, den `FileNew` spezifiziert. Liefert bei Erfolg `true` und `false` sonst.

`createNewFile()` Erzeugt eine neue, leere Datei mit dem in der File-Variablen spezifizierten Pfad und Namen, alledings nur dann, wenn diese Datei nicht existiert. Der Boolesche Rückgabewert zeigt an, ob es geklappt hat.

5.1.2 Die Klassen `FileReader` und `FileWriter`

Die Klassen `FileReader` und `FileWriter` stellen die Methoden zum Lesen beziehungsweise Schreiben von ASCII Dateien bereit. Zunächst die Klasse `FileReader`.

```
FileReader(File datei)

void close()
int  read()
int  read(char[] puffer,int anfang,int laenge)
```

`FileReader(File datei)` Dieser Konstruktor öffnet die Datei `datei` zum Lesen.

`close()` Beendet den FileReader und schließt die zugehörige Datei.

`read()` Liest ein einzelnes Zeichen aus der Datei und liefert die Unicode-Darstellung in einem Ganzzahlwert zurück.

`read(puffer,anfang,laenge)` Liest `laenge` viele Zeichen aus der Datei und schreibt sie in das Char-Array puffer ab der Position `anfang` des Puffers. Gibt die Anzahl der gelesenen Zeichen zurück oder -1, wenn das Dateiende vorzeitig erreicht wurde.

Der Konstruktor öffnet die Datei `datei` zum Lesen.

Die meisten Konstruktoren und Methoden der Bibliothek `java.io` werfen Exceptions, wenn Fehler auftreten, meistens `IOExceptions`. Sie können also nur in `try catch` Blöcken verwendet werden, oder man muss die Ausnahmen weiterwerfen. Weiterhin sollte man das Schließen des Eingangsstroms mittels `close()` nicht vergessen.

Die Klasse `FileWriter` stellt die Methoden zum schreiben von ASCII-Dateien bereit.

```
FileWriter(File datei)

void close()
void write(char[] puffer)
void read(char[] puffer,int anfang,int laenge)
```

`FileWriter(File datei)` Dieser Konstruktor öffnet die Datei `datei` zum Schreiben, ist sie schon vorhanden wird ihr Inhalt gelöscht.

`close()` Beendet den FileWriter und schließt die zugehörige Datei.

`write(puffer)` Schreibt den Inhalt des Char-Array `puffer` in die Datei.

`read(puffer,anfang,laenge)` Schreibt `laenge` viele Zeichen des Char-Array `puffer` in die Datei. Begonnen wird an Position `anfang` des Puffers.

5.1.3 Einfache Manipulationen von Textdateien

Das unten angegeben Programm `FileReadWrite` setzt die gerade besprochenen Konzepte um. Es liest aus der Textdatei `testtext.txt` einige Zeichen ein und speichert diese zu einer eventuellen Bearbeitung in einem Puffer (Char-Array). Der Puffer wird in die Standardausgabe (Bildschirm) geschrieben. Dann wird ein Teil des Puffers in die Textdatei `testtext.txt` geschrieben. Existiert letztere Datei bereits, so wird sie vorher gelöscht. Wichtig ist, dass man die Konstante `path` abhängig vom eigenen System korrekt setzt und mit einem Bruchstrich (`/`) abschließt. !

Bei der Bildschirmausgabe fällt auf, dass die Zeilenumbrüche wie in der Datei erscheinen. Der FileReader liest also die Zeilenumbruch Zeichen (\n) mit und speichert sie im Puffer ab.

Datei: `FileReadWrite.java`

```
package kurs.TextIO;

import java.io.*;

public class FileReadWrite
{
 // die Konstante path enthält den Stammpfad, bitte entsprechend setzten
 public static String path = "PATH/";


 public FileReadWrite(String ReadFileName,String WriteFileName)
  {
    //Puffer zur Aufnahme der Zeichen
    char[] buffer = new char[128];

    // Anlegen zweier Variable vom Typ File
    File readfile  = new File(path+ReadFileName);
    File writefile = new File(path+WriteFileName);
//LESEN:
    if (readfile.exists())
    {
     try
     {
        // Ein FileReader anlegen
        FileReader fr = new FileReader(readfile);

        // lies die Datei ein und speichert 100 Zeichen in buffer
```

```
        // ab der Stelle 5
        fr.read(buffer,5,100);

        // Dateizugriff beenden
        fr.close();
     }
     catch (Exception ex)
     {
      System.out.println("Problem opening file "+readfile.getName());
     }
    }//if
    else
    {
      System.out.println("File not found "+readfile.getName());
    }
    // Inhalt von Buffer ausgeben
    System.out.println("Buffer>"+buffer+"<Buffer");

//SCHREIBEN:
     try
     {
      if (writefile.exists())
      {
        writefile.delete();
      }
      if (writefile.createNewFile())
        {
        // Ein FileWriter kann Zeichen(arrays) schreiben
        FileWriter fw = new FileWriter(writefile);

        // Schreibt aus dem Puffer 40 Zeichen  ab dem 7ten

        fw.write(buffer,7,40);

        // Dateizugriff beenden
        fw.close();

    }//if
    else
    {
      System.out.println("File not created "+writefile.getName());
    }
    }
     catch (Exception ex)
     {
      System.out.println("Problem opening file "+writefile.getName());
```

```
    }

  }

  public static void main(String[] args)
   {
    FileReadWrite RWF = new FileReadWrite("testtext.txt","testtext2.txt");
   }
}
```

5.1.4 Eleganter Dateizugriff

Das folgende Programm `ElegantRead` benutzt einen `BufferedReader`, der das zeilenweise Einlesen von Text aus einer Textdatei mit Zeilenumbrüchen erlaubt.

Datei: `ElegantRead.java`

```
package kurs.TextIO;

import java.io.*;

public class ElegantRead
{
 // die Variable path enthält den Stammpfad, bitte entsprechend setzten
 public static String path = "PATH/";

  public ElegantRead( String ReadFileName)
  {
    File readfile = new File(path+ReadFileName);
    String line;
    try
    {
      BufferedReader bfr = new BufferedReader(new FileReader(readfile));
      while((line = bfr.readLine()) != null)
      {
        System.err.println("gelesen>"+line+"<");
      }
    }
    catch (Exception ex)
    {
      System.out.println("Problem opening file "+readfile.getName());
```

```
    }

  }

   public static void main(String[] args)
   {
    ElegantRead ER = new ElegantRead("testtext.txt");
   }
}
```

5.2 Text aus einer Datei anzeigen

Wir wollen nun den Text aus unserer Datei nicht nur auf dem Bildschirm anzeigen lassen, sondern in einem Fenster. Dazu verwenden wir die Swing-Komponente `JEditorPane`. Es sei angemerkt, dass auch andere Swing-Komponenten in Frage gekommen wären, nämlich `JTextArea` oder `JTextPane`. Die beiden letztgenannten stützen sich auf ein so genanntes Dokumenten-Modell ab, dessen Behandlung den Rahmen eines einwöchigen Kurses sprengen würde. Wir benutzen daher `JEditorPane` als einfachste dieser Komponenten.

Um den Text in eine `JEditorPane` zu bekommen kann man die Methode `read` mit einem Reader aufrufen, der mit der entsprechenden Datei initialisiert wurde.

```
JEditorArea()

void   read(Reader leser, Object beschreibung)
String getText()
```

Das zweite Argument der `read`-Methode interessiert uns hier nicht; es dient dazu bei bestimmten Texttypen (zum Beispiel HTML) weiter Information zur Darstellung zu übergeben. Da wir nur ASCII-Texte betrachten setzen wir `beschreibung = null`. Die Methode `getText` liefert den Inhalt der EditorPane als einen einzigen String zurück, ggf. mit Zeilenumbrüchen.

Der folgende Programm-Code `TextAnzeigeFrame` realisiert eine Textanzeige. In einen von `SimpleFrame` abgeleiteten Rahmen kleben wir zentral eine `JEditorPane`. Der Konstruktor bekommt einen Dateinamen `fileName` als String übergeben. Wir legen wie in Abschnitt 5.1.3 einen `FileReader` für die Datei `fileName` an und rufen die `read`-Methode der `JEditorPane` damit auf. Der Reader wird gestartet und der

gelesenen Text in die `JEditorPane` übernommen. Die Datei `TextAnzeigeTest` ist lediglich die Startdatei.

```
package kurs.TextAnzeige;

import java.awt.*;
import javax.swing.*;
import java.io.*;
import kurs.SimpleFrame.SimpleFrame;

public class TextAnzeigeFrame extends SimpleFrame
{
  private JEditorPane TextAnzeigePanel;

  public TextAnzeigeFrame(String filename)
  {
   TextAnzeigePanel = new JEditorPane();
   this.getContentPane().add(TextAnzeigePanel,"Center");

   File readfile  = new File(filename);

   try{
    FileReader fr = new FileReader(readfile);
    TextAnzeigePanel.read(fr,null);
   }catch(IOException e){
   System.out.println("Probleme beim Öffnen oder Lesen von "+readfile.getName());
   }
  }
}
```

Datei: `TextAnzeigeTest.java`

```
package kurs.TextAnzeige;


public class TextAnzeigeTest
{

  // Bitte path richtig setzen!
  private static String path = "PATH/";
```

```
  private static String fileName = "testtext.txt";


  public TextAnzeigeTest(String filename)
  {
    TextAnzeigeFrame TAF = new TextAnzeigeFrame(filename);
    TAF.showIt("Text anzeigen");
  }
  public static void main(String[] args)
  {
    TextAnzeigeTest textAnzeigeTest1 = new TextAnzeigeTest(path+fileName);
  }
}
```

Das Resultat sollte wie in Abbildung 5.1 aussehen. Offenbar sind ist der Frame zu klein um alle Zeilen der Datei anzuzeigen. Vergrößern des Fensters hilft nur dann, wenn alle Zeilen der Datei in die Höhe des Bildschirms passen. Wir werden das Problem im nächsten Abschnitt mittels *Rollens* (*Scrolling*) lösen.

! **Wir können den angezeigten Text bereits edieren! Mit der Maus lässt sich eine *Einfügemarke* (*cursor*) positionieren. Wir können löschen, einfügen, auswählen und sogar kopieren (mit CNTR-C und CNTR-V).**

5.3 Scrolling

Unter *Rollen* (*Scrolling*) von Fensterinhalten versteht man das Darstellen eines Ausschnitts der gesamten Information in einem Fenster, wobei man den sichtbaren Ausschnitt interaktiv verschieben (neudeutsch *scrollen*) kann. In Abbildung 5.2 ist der prinzipielle Aufbau eines Scroll-Mechanismus dargestellt. In den Frame ist eine Fläche eingebettet, die in einem Bereich, dem sog. *Viewport* einen Ausschnitt des eigentlichen Dokuments anzeigt. Zusätzlich gibt es zwei Schieberegler mit denen der sichtbare Ausschnitt im Dokument horizontal und vertikal verschoben werden kann. In Swing realisiert die Komponente `JScrollPane` diesen Mechanismus. Unser Text wird weiterhin in einer `JEditorPane` dargestellt. Die ScrollPane gibt aber nur den Blick auf einen Ausschnitt davon frei.

```
JScrollPane(JComponent KOMPONRENTE);
```

Abbildung 5.1: Resultat des Programms `TextAnzeigeTest`. Man sieht, dass nicht alle Zeilen der Textdatei sichtbar sind.

Dieser Konstruktor erzeugt eine `JScrollPane`, in deren Viewport ein Ausschnitt der Komponente KOMPONENTE dargestellt. Passt die ganze Komponente in den Viewport, werden die Schieberegler ausgeblendet. Dieses Verhalten lässt sich ändern.

Im folgenden ist der Code des Programms `TextAnzeigeScrollFrame` angegeben, der eine rollbare Textdarstellung erlaubt. Auf die Angabe des Startprogramms verzichten wir.

Datei: `TextAnzeigeScrollFrame.java`

```
package kurs.TextAnzeigeScroll;

import java.awt.*;
```

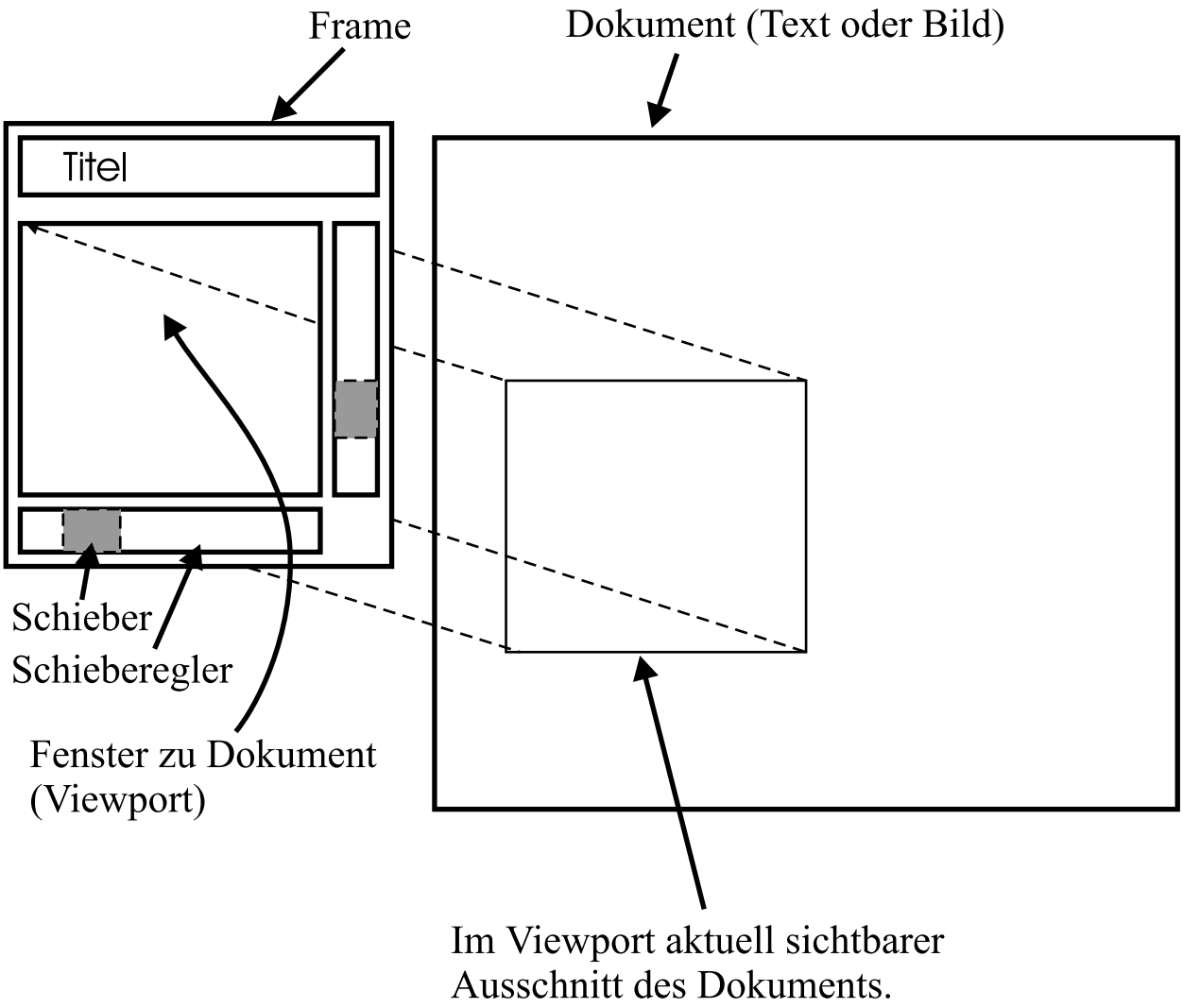

Abbildung 5.2: Struktureller Aufbau einer scrollbaren Darstellung.

```
import javax.swing.*;
import java.io.*;
import kurs.SimpleFrame.SimpleFrame;

public class TextAnzeigeScrollFrame extends SimpleFrame
{
  private JEditorPane TextAnzeigePanel;

  public TextAnzeigeScrollFrame(String filename)
  {

   TextAnzeigePanel = new JEditorPane();
```

```
   JScrollPane JSP  = new JScrollPane(TextAnzeigePanel);

   this.getContentPane().add(JSP,"Center");

   File readfile  = new File(filename);

   try{
    FileReader fr = new FileReader(readfile);
    TextAnzeigePanel.read(fr,null);
   }catch(IOException e){
   System.out.println("Probleme beim öffnen oder Lesen von "+readfile.getName());
   }
  }
}
```

In Abbildung 5.3 sieht man das Ergebnis dieses Programms.

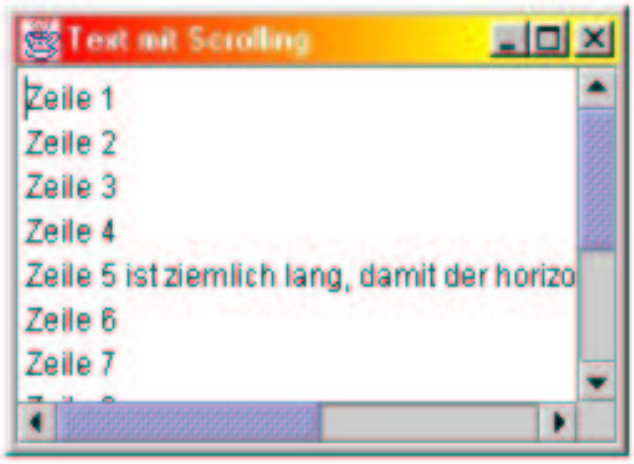

Abbildung 5.3: Resultat des Programms `TextAnzeigeScrollTest`. Der Frame wurde mit der Maus ein wenig vergrößert.

5.4 Menüs

Für unseren geplanten Texteditor sind sicherlich Menüs ein wichtiges Hilfsmittel. Um Menüs in Swing nutzbar zu machen, benötigen wir drei Swing-Komponenten. Ersten `JMenuBar` die *Menüleiste* die im Frame zwischen der Titelleiste und der ContextPane eingefügt wird. Zweitens `JMenu`, eine Klasse die Menüs mit Ihrem Titel in die Menüleiste einfügt. Drittens `JMenuItem`, die Klasse die die Menüeinträge definiert, die der Benutzer wählen kann. Siehe auch Abbildung 5.4. Die Grundfunktionen von Menüs, zum Beispiel dass ein Menü beim Anklicken aufklappt, werden wieder automatisch bereitgestellt. Von der Klasse `JMenuBar` benötigen wir nur den Konstruktor

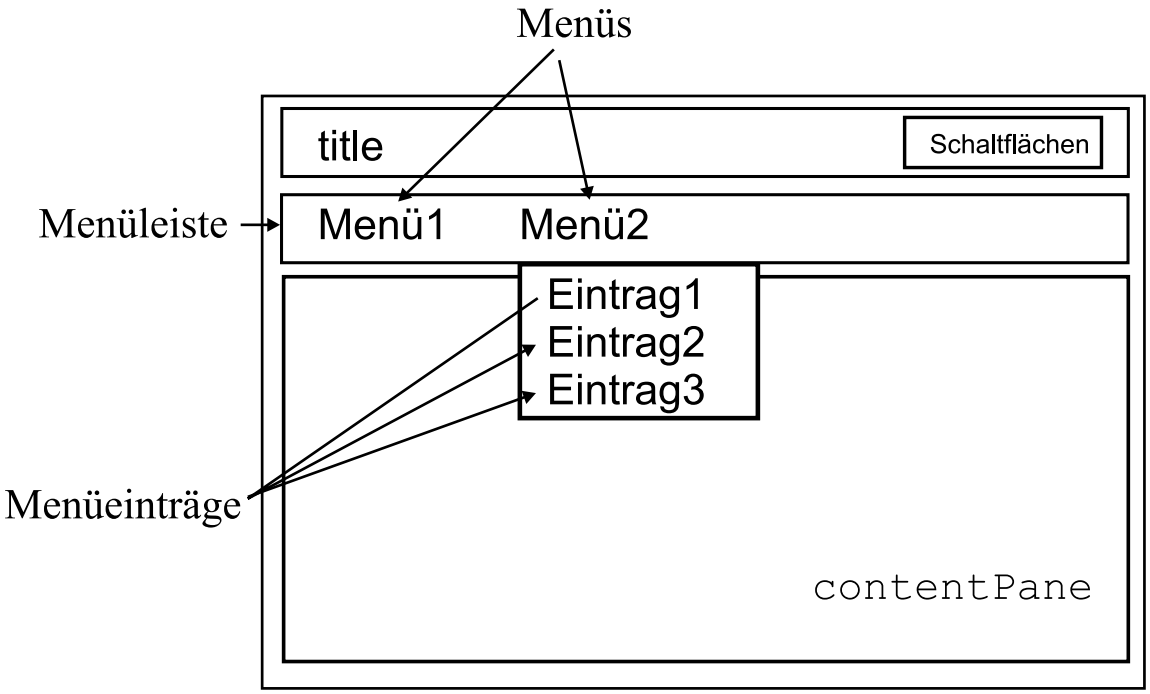

Abbildung 5.4: Ein Frame mit zwei Menüs. Das zweite Menü ist aufgeklappt und sein drei Einträge sind sichtbar und können angeklickt werden.

und die Methode mit der ein Menü in die Menüleiste eingehängt wird.

```
JMenuBar()
add(JMenu menue)
```

Um die Menüleiste in einen Frame einzufügen benutzen man die folgende Methode der Klasse `JFrame`.

```
setJMenuBar(JMenuBar menueLeiste)
```

Von der Klasse `JMenu` benötigen wir den Konstruktor der den Titel des Menüs als String erhält, die Methode zum Einhängen eines Menüeintrages und Methode zum Einfügen eines Trennstriches (*Separators*).

```
JMenu(String menuetitel)
add(JMenuItem menueEintrag)
addSeparator()
```

Von der Klasse `JMenuItem` benötigen wir den Konstruktor der den Titel des Menüeintrages als String erhält und die Methode zum Zuordnen eines Aufpassers, der eine Aktion beim Anklicken des Menüpunktes auslöst.

```
JMenuItem(String menuePunkt)
addActionListener(ActionListener aufpasser)
```

Damit haben wir alles zusammen um Menüs einsetzen zu können. Den Aufpasser definieren wir in der **internen** Klasse `TAListener` (für TextAnzeigeListener), damit er Zugriff auf die lokalen Komponenten des Frames hat. Der Listener fragt in diesem Falle zunächst ab, ob der Auslöser (source) des Ereignisses `evt` ein Menüeintrag war. dies geschieht mittels

```
if(evt.getSource() instanceof JMenuItem)
```

Wenn dem so ist, entnimmt der Listener dem Ereignis `evt` das `actionCommand`. Ein solches Kommando kann man jeder ereignisauslösenden Komponente zuordnen, also jedem Knopf oder Menüeintrag. Macht man das nicht explizit, so wird als Kommando automatisch der String der Beschriftung genommen, hier also „Öffnen“, „Speichern“ usw. Der Listener fragt jetzt in einer `if-then-else`-Kaskade ab, welcher Menüpunkt ihn aktiviert hat. Er aktiviert dann den entsprechenden Code. Im Beispiel `TextAnzeigeMenuFrame` ist dies einfach nur eine `println`-Anweisung, die noch einmal den gewählten Menüeintrag auf der Konsole anzeigt. Später werden wir an diesen Stellen „richtigen“ Code einfügen. Auf die Beschreibung des Startprogramms verzichten wir wieder.

Datei: TextAnzeigeMenuFrame.java

```
package kurs.TextAnzeigeMenu;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import kurs.SimpleFrame.SimpleFrame;

public class TextAnzeigeMenuFrame extends SimpleFrame
{
  private JEditorPane TextAnzeigePanel;

  public TextAnzeigeMenuFrame(String filename)
  {
   // Menüleiste einfügen
   JMenuBar menueLeiste = new JMenuBar();
   this.setJMenuBar(menueLeiste);

   // Erstes Menue mit Einträgen erzeugen und in die
   // Menüleiste einhängen.

   JMenu fileMenu = new JMenu("Datei");
   menueLeiste.add(fileMenu);

   JMenuItem openItem  = new JMenuItem("Öffnen");
   JMenuItem saveItem  = new JMenuItem("Speichern");
   JMenuItem closeItem = new JMenuItem("Schliessen");

   fileMenu.add(openItem);
   fileMenu.add(saveItem);
   fileMenu.addSeparator();
   fileMenu.add(closeItem);


   // Zweites Menü

   JMenu searchMenu = new JMenu("Suchen");
   menueLeiste.add(searchMenu);

   JMenuItem searchItem = new JMenuItem("Suchen");
   searchMenu.add(searchItem);

   // Jedem Menüeintrag den Listener zuordnen
```

```
 TAListener TALi = new TAListener();
 openItem.addActionListener(TALi);
 saveItem.addActionListener(TALi);
 closeItem.addActionListener(TALi);
 searchItem.addActionListener(TALi);

 TextAnzeigePanel = new JEditorPane();
 JScrollPane JSP  = new JScrollPane(TextAnzeigePanel);

 this.getContentPane().add(JSP,"Center");

 File readfile  = new File(filename);

 try{
  FileReader fr = new FileReader(readfile);
  TextAnzeigePanel.read(fr,null);
 }catch(IOException e){
 System.out.println("Probleme beim öffnen oder Lesen von "+readfile.getName());
 }



}

class TAListener implements ActionListener{
  public void actionPerformed(ActionEvent evt)
   {
   if (evt.getSource() instanceof JMenuItem)
   {
    String command = evt.getActionCommand();
    // command enthält jetzt die Beschriftung des gewählten Menüeintrags
    if(command.equals("Öffnen"))
    {
     System.out.println("Menüeintrag "+command+" gewählt");
     // CODE zum Öffnen hier einfügen
    }
    else if(command.equals("Speichern"))
    {
     System.out.println("Menüeintrag "+command+" gewählt");
     // CODE zum Speichern hier einfügen
    }
    else if (command.equals("Schliessen"))
    {
     System.out.println("Menüeintrag "+command+" gewählt");
     // CODE zum Schliessen hier einfügen
    }
```

```
      else if (command.equals("Suchen"))
      {
       System.out.println("Menüeintrag "+command+" gewählt");
       // CODE zum Suchen hier einfügen
      }

      }//if instanceOf
    }// method actionPerformed
   }//class TAListener

}//class TextAnzeigeMenuFrame
```

5.5 Dateiauswahldialoge

Im Abschnitt 5.2 haben wir eine feste Datei eingelesen. Im Allgemeinen will man aber die zu bearbeitende Datei zur Laufzeit frei wählen können. Daher fügen wir nun einen „Dateiauswahldialog“ in unser Programm ein, der beim Anklicken des Menüeintrages „Öffnen“ angezeigt wird. In diesem Dialog können wir wie gewohnt eine Datei graphisch auswählen. Beim Schließen des Dialoges wird die gewählte Datei an das aufrufende Programm zurückgegeben.

Die Swing-Bibliothek enthält bereits einen solchen Dialog in der Klasse `JFileChooser`. Wir stellen nur die für unsere Anwendung wichtigen Methoden vor. Die Klasse ist sehr mächtig und bietet viele Möglichkeiten zu individueller Gestaltung. Man kann in einem `JFileChooser` so navigieren, wie man es von den Dateiauswahldialogen des jeweiligen Betriebssystems gewohnt ist.

```
JFileChooser(String startVerzeichnis);

int  showOpenDialog(Component mutter);
int  showSaveDialog(Component mutter);
File getSelectedFile();

public static final int CANCEL_OPTION // Konstante
public static final int APPROVE_OPTION // Konstante
```

`JFileChooser(String startVerzeichnis)` Erzeugt einen Dateiauswahldialog, der aber noch nicht angezeigt wird. Das Startverzeichnis wird als String übergeben.

`showOpenDialog(Component mutter)` Zeigt den Dateiauswahldialog zum Öffnen von Dateien an. Die Dateien des Startverzeichnisses werden angezeigt. Die Komponente `mutter` ist im Allgemeinen die Komponente, aus der heraus `showOpenDialog` aufgerufen wurde. Dies ist deswegen wichtig, weil Dateiauswahldialoge *modal* sind, das heißt, während sie sichtbar sind blockieren sie die Mutterkomponente. Erst wenn der Dialog wieder unsichtbar ist, kann dort weitergearbeitet werden. Der Dialog liefert beim Schließen einen Ganzzahlwert zurück. Es ist eine der beiden oben angegeben Konstanten. `APPROVE_OPTION` zeigt an, dass den Knopf „Open“ beziehungsweise „Öffnen“ angeklickt wurde. `CANCEL_OPTION` zeigt an, dass den Knopf „Cancel“ beziehungsweise „Abbrechen“ angeklickt wurde.

!

`showSaveDialog(Component mutter)` Zeigt den Dateiauswahldialog zum Speichern von Dateien an. Ansonsten wie `showOpenDialog`.

`getSelectedFile()` Liefert die ausgewählte Datei in einer Variablen vom Type `File` zurück, falls `APPROVE_OPTION` gewählt wurde.

Abbildung 5.5 zeigt das Ergebnis.

Wir fügen nun an der mit `// CODE zum Öffnen hier einfügen` gekennzeichneten Stelle im Programm `TextAnzeigeMenuFrame` den entsprechenden Code ein. Zunächst erzeugen wir einen Dateiauswahldialog und zeigen ihn dann als Dialog zum Öffnen von Dateien an. Das Startverzeichnis ist in der String-Variablen `path` gegeben, die vorher gesetzt werden muss. Bei Wahl einer Datei (`APPROVE_OPTION`) lesen wir die Datei ein und zeigen sie an.

Das Codestück, das bisher die vorgegebene Datei geladen hat, löschen wir. Ebenso übergeben wir dem Editor nicht mehr einen Dateinamen.

Wir geben nun nur noch die zu ergänzenden Code-Stücke an; den kompletten Editor findet man im Programm-Listing `Editor` auf Seite 58

```
JFileChooser FC = new JFileChooser(path);

int returnVal = FC.showOpenDialog(EditorFrame);
if(returnVal == JFileChooser.APPROVE_OPTION)
{
     File selectedFile =  FC.getSelectedFile();
     try{
      FileReader fr = new FileReader(selectedFile);
      TextAnzeigePanel.read(fr,null);
```

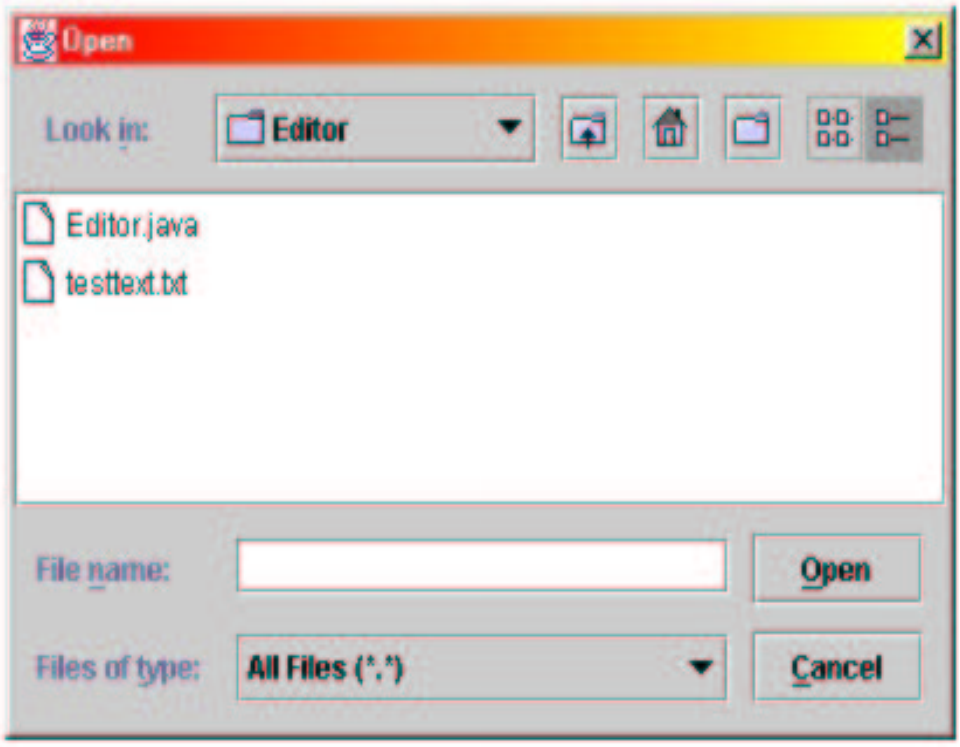

Abbildung 5.5: Ein Dateiauswahldialog.

```
      fr.close();
     }catch(IOException e){
     System.out.println("Probleme beim öffnen oder Lesen von "
                        + selectedFile.getName());
     }
}
```

5.6 Auf dem Weg zum Editor

Nun fügen wir den Code ein, der beim Wählen der Menüeinträge „Speichern“ und „Schließen“ ausgeführt werden soll.

Beim Menüeintrag „Schließen“ gehen wir brutal vor und beenden einfach die Anwendung.

```
exit(0);
```

Aufgabe 6: Überlege Dir, was man sinnvollerweise beim Schließen eines Editors alles machen sollte.

Beim Menüeintrag „Speichern“ schreiben wir den aktuell in dem EditorPane `TextAnzeigePanel` vorhandenen Text in die Datei zurück.

```
{
FileWriter fw = new FileWriter(selectedFile);
fw.write(TextAnzeigePanel.getText());
fw.close();
}catch(IOException e){
 System.out.println("Probleme beim Schreiben in "+selectedFile.getName());
}
```

Damit sind die Punkte des ersten Menüs alle mit Leben erfüllt. Das zweite Menü „Suchen“ dient dazu, die im nächsten Kapitel vorgestellten Dialoge zu verwenden.

Aufgabe 7: Untersuche, was beim mehrmaligen Laden von Dateien geschieht.

Aufgabe 8: Füge einen Menüeintrag „Speichern unter“ in das Dateimenü ein. Beim Anklicken soll man mittels Dateiauswahldialoges aufgefordert werden, Ort und Dateinamen für die Abspeicherung anzugeben. Füge Code ein, der dies alles leistet.

Aufgabe 9: Füge am unteren Rand des Editor-Rahmens eine „Status-Leiste“ ein, die Zusatzinformation anzeigt, zum Beispiel den Namen der geladenen Datei oder ihre Länge.

Kapitel 6

Dialoge

Dialoge dienen dem Austausch von Information zwischen dem Benutzer und dem laufenden Programm. Wir wollen unseren Editor aus dem Kapitel 5 mit einer Suchfunktion ausstatten. Beim Anklicken des Menüeintrages „Suchen“ soll ein Dialog erscheinen, der uns zur Eingabe des gesuchten Wortes auffordert und uns die Möglichkeit gibt, zu wählen, ob bei der Suche die Groß-Klein-Schreibung beachtet werden soll oder nicht.

Die Swing-Klasse für Dialoge ist `JDialog`. Ein Dialog verhält sich in vielem wie ein Frame. Der wichtigste Unterschied für uns ist, dass ein Dialog *modal* seien kann. Ist er modal, so blockiert er seine Mutterkomponente während er sichtbar ist. Erst wenn er unsichtbar geworden ist, kann man in der Mutterkomponente weiterarbeiten.

Für unseren Dialog wählen wir ein Panel mit 4×2-Grid-Layout, das wir in die Content-Pane des Dialogs kleben. Die Komponenten ordnen wir in diesem Raster an. Abbildung 6.1 zeigt das Ergebnis. Aus der Klasse `JDialog` benötigen wir nur den Konstruktor und einige Methoden zum Hinzufügen von Komponenten, die denen aus der Klasse `JFrame` entsprechen. Neu ist die `pack`-Methode, die alle Komponenten in einem Frame, Dialog oder Panel so dicht wie möglich zusammenpackt. Wir rufen den default-Konstruktor mittels des `super`-Befehls auf. Übergeben wird die Mutterkomponente, der Titel des Dialogs und ein `boolean` um die Modalität zu setzen.

```
JDialog(Frame mutter, String title, boolean modal)

pack()
```

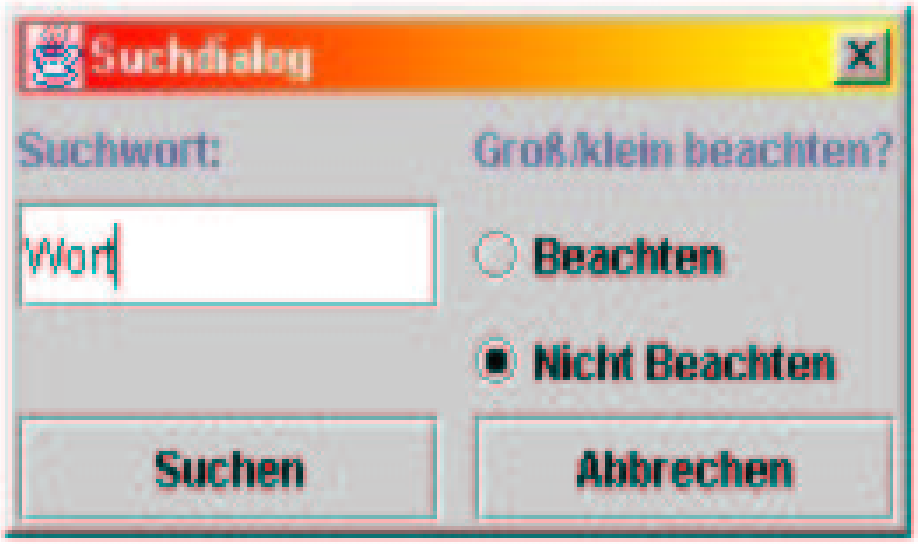

Abbildung 6.1: Der Such-Dialog.

6.1 Radiotasten (radio button)

Die beiden Knöpfe unten („Suchen“ und „Abbrechen“) sind `JButton`s wie wir sie bereits in `FirstGUI` in Kapitel 4 kennen gelernt haben. Neu sind die beiden Knöpfe rechts („Beachten“ und „Nicht Beachten“). Es sind so genannte *Radiotasten* (*radio button*), von denen wie bei Stationstasten eines Radios immer nur eine gedrückt seien darf. Mittels Mausklick schaltet man die Knöpfe ein oder aus. Die zugehörige Klasse ist `JRadioButton`.

```
JRadioButton(String beschriftung)

setSelected(boolean gedrueckt)
setActionCommand(String command)
```

Mit der Methode `setSelected` kann man festlegen, ob der Knopf gedrückt sein soll oder nicht. Oftmals erhalten Radiotasten keine Beschriftung sondern ein Symbol. Daher wird ihnen auch nicht automatisch ein ActionCommand zugeordnet wie den `JButton`. Damit man später feststellen kann, welcher Knopf gedrückt wurde, ordnet

man jedem eine individuelles Kommando mittels der Methode `setActionCommand`. Dies muss nicht die Beschriftung sein.

Bis jetzt ist die Radiotasteneigenschaft (genau eine gedrückt) noch nicht gegeben. Man könnte sich ja durchaus mehrere Tastenblöcke vorstellen, von denen immer je eine Taste gedrückt ist. Wir müssen unsere RadioButtons als gruppieren. Dies übernimmt die Klasse `ButtonGroup`. Unten sind der Konstruktor die Methode zum hinzufügen eines Knopfes und die Methode zur Bestimmung des gedrückten Knopfes, genauer zur Bestimmung des ActionCommands des gedrückten Knopfes, angegeben.

```
JButtonGroup();

add(JButton knopf)
String getSelection().getActionCommand()
```

6.2 Informationsaustausch zwischen Programm und Dialog

Unser Dialog ist nun fertig. Wie aber kommen die Daten über das Suchwort usw. in das Programm? Man könnte der `showIt` Methode einen Rückgabewert – zum Beispiel einen String – geben, in den man die Information hineinpackt. Das Programm, das den Dialog aktiviert hat, kann die Daten dann aus dem String herausnehmen.

Eleganter und dem objektorientierten Konzept von Java besser angepasst ist jedoch das folgende Vorgehen. Man definiert ein Klasse, deren Objekte die zu transferierenden Daten aufnehmen können. Ein solches Objekt wird vom Hauptprogramm erzeugt und dem Dialog in der Methode `showIt` übergeben, genauer die Referenz auf das Objekt wird dem Dialog übergeben. Der Dialog kann dem Objekt Daten entnehmen oder Daten eintragen. Nach Beendigung des Dialoges enthält das Objekt dann ggf. neue Daten, die das Hauptprogramm nutzen kann.

In dem unten stehenden Beispiel `InfoTransferObject` ist eine auf unsere Anwendung zugeschnittene Klasse definiert. Eine Instanz eines `InfoTransferObject` enthält drei Felder: einen String und zwei Boolesche Werte. Diese können mittels der definierten `get-` und `set`-Methoden gelesen beziehungsweise gesetzt werden. Der Dialog nutzt die Methode `setAll` um das zu suchende Wort in das Feld `suchwort` einzutragen. Weiterhin wird im Feld `grossklein` eintragen, ob die Groß/Kleinschreibung zu beachten ist und in das Feld `suchen` wird `true` eingetragen, wenn man zum Verlassen des Dialogs den Knopf „Suchen“ angeklickt hat. Hat man „Abbrechen“ angeklickt, so wird `false` eingetragen.

Das aufrufende Programm `Editor` kann nun feststellen, ob es suchen soll (falls nämlich `getSearch() == true`), welches Wort es suchen soll (nämlich `getSearchWord()`) und ob es dabei Groß/Kleinschreibung beachten soll (falls nämlich `getCaseSensitive() == true`).

Dem Dialog haben wir mit der Klasse `searchListener` einen Aufpasser mitgegeben, der die beiden Knöpfe „Suchen“ und „Abbrechen“ im Auge behält. Wird einer davon angeklickt, so schreibt der Aufpasser die Daten in das `InfoTransferObject` und macht den Dialog unsichtbar. In diesem Moment wird die Blockierung des Editors durch den Dialog aufgehoben und dieser kann dem `InfoTransferObject` die Daten entnehmen und mit der Suche beginnen.

In unserm Beispiel beschränkt sich sie Suche darauf, festzustellen, wie oft das gesuchte Wort im Text vorkommt. Auch können nur ganze Wörter, keine beliebigen Teilstrings, gesucht werden.

Unseren Dialog möchten wir nur einmal erzeugen, das heißt, den Konstruktor nur einmal aufrufen, und ihn dann „recyclen“. Dies geschieht, indem wir in `Editor` mittels `setVisible(true)` anzeigen lassen und in `searchDialog` mittels `setVisible(false)` unsichtbar machen. Um multiple Instanzen des Dialogs zu verhindern benutzen wir den folgenden Code, der den Dialog nur erzeugt, wenn die Referenz auf ihn noch `null` ist. Ebenso benötigen wir nur eine Instanz von `InfoTransferObject`.

```
if (SDia == null)
 {
   SDia = new SearchDialog(parent);
   ITO  = new InfoTransferObject();
 }
```

Die Variablen `SDia` und `ITO` müssen natürlich vorher deklariert worden sein.

```
 private SearchDialog SDia;
 private InfoTransferObject ITO;
```

Die folgenden Listings enthält den JAVA-Code der Dialogklasse `SearchDialog` und der Klasse `InfoTransferObject`.

Datei: `SearchDialog.java`

```
package kurs.Editor;
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class SearchDialog extends JDialog
{
  private JPanel hauptPanel = new JPanel();
  private JTextField suchTextFeld = new JTextField();
  private JRadioButton yesButton  = new JRadioButton("Beachten");
  private JRadioButton noButton   = new JRadioButton("Nicht Beachten");
  private JButton searchButton    = new JButton("Suchen");
  private JButton cancelButton    = new JButton("Abbrechen");
  private InfoTransferObject ITO;
  private ButtonGroup gruppe = new ButtonGroup();

  public SearchDialog(Frame frame)
  {
    super(frame,"Suchdialog",true);

    JLabel aufforderung = new JLabel("Suchwort:");
    JLabel gkFrage      = new JLabel("Groß/klein beachten?");
    JLabel fueller      = new JLabel();

    this.getContentPane().setLayout(new BorderLayout());
    this.setLocation(300,300);
    this.getContentPane().add(hauptPanel,"Center");

    hauptPanel.setLayout(new GridLayout(4,2,10,0));
    hauptPanel.add(aufforderung);
    hauptPanel.add(gkFrage);
    hauptPanel.add(suchTextFeld);
    hauptPanel.add(yesButton);
    hauptPanel.add(fueller);
    hauptPanel.add(noButton);
    hauptPanel.add(searchButton);
    hauptPanel.add(cancelButton);

    gruppe.add(yesButton);
    gruppe.add(noButton);
    yesButton.setActionCommand("beacht");
    noButton.setActionCommand("nicht");
    yesButton.setSelected(true);
    noButton.setSelected(false);

    searchListener SLis = new searchListener();
    searchButton.addActionListener(SLis);
```

```
    cancelButton.addActionListener(SLis);

    this.pack();
  }

  public void showIt(InfoTransferObject i)
  {
    ITO = i;
    setVisible(true);
  }

  class searchListener implements ActionListener
  {
   public void actionPerformed(ActionEvent evt)
     {
       String suchText = suchTextFeld.getText();
       boolean grkl = (gruppe.getSelection().getActionCommand().equals("beacht"));

       String command = evt.getActionCommand();
       if(command.equals("Abbrechen"))
        {
         ITO.setAll(suchText,grkl,false);
         setVisible(false);
        }
       else if (command.equals("Suchen"))
        {
         ITO.setAll(suchText,grkl,true);
         setVisible(false);
        }//ifelse
     }//method
  }//internal class
}
```

6.3 Vordefinierte Optionsdialoge

Für viele einfache Dialoge gibt es in Swing bereits vordefinierte Schablonen. Es sind dies zum Beispiel Meldungen an den Benutzer, die er nur zur Kenntnis nehmen kann („Datei Win.ini wurde gelöscht“), oder Fragen mit wenigen Alternativen („Festplatte formatieren? Ja/Nein“). Durch Wahl eines Typs (zum Beispiel `MessageDialog`, `MessageDialog`, `OptionDialog`) und Übergabe von vordefinierten Konstanten können diese Dialoge sehr vielseitig gestaltet werden.

Wir benutzen einen solchen Dialog, um nach einer Suche dem Benutzer die Anzahl der gefundenen Übereinstimmungen anzuzeigen. Hier geben wir nur den entsprechenden Code an, näheres dazu findet man in der Referenz der Swing-Klasse `JOptionPane`.

```
JOptionPane.showMessageDialog(Component mutter,
 String titel,
 String meldung,
 int type);
```

Damit ist unser Editor fertig. Der Code ist im folgenden Listing zu finden. Gegenüber dem Programm `TextAnzeigeMenuFrame` ist Code für die Menüereignisse eingefügt worden und einige dazu nötige Variablen sind hinzugekommen. Der Editor benötigt die Klassen `SearchDialog` und `InfoTranferObject`.

Datei: `Editor.java`

```
package kurs.Editor;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.util.*;
import kurs.SimpleFrame.SimpleFrame;

public class Editor extends SimpleFrame
{
  private JEditorPane TextAnzeigePanel;
  private static final String path = "D:/JAVA/Kurs/scr/kurs/Editor/";
  private File selectedFile;
  private SearchDialog SDia;
  private InfoTransferObject ITO;

  public Editor()
  {
   // Menüleiste einfügen
   JMenuBar menueLeiste = new JMenuBar();
   this.setJMenuBar(menueLeiste);

   // Erstes Menue mit Einträgen erzeugen und in die
   // Menüleiste einhängen.
```

```
 JMenu fileMenu = new JMenu("Datei");
 menueLeiste.add(fileMenu);

 JMenuItem openItem  = new JMenuItem("Öffnen");
 JMenuItem saveItem  = new JMenuItem("Speichern");
 JMenuItem closeItem = new JMenuItem("Schliessen");

 fileMenu.add(openItem);
 fileMenu.add(saveItem);
 fileMenu.addSeparator();
 fileMenu.add(closeItem);


 // Zweites Menü

 JMenu searchMenu = new JMenu("Suchen");
 menueLeiste.add(searchMenu);

 JMenuItem searchItem = new JMenuItem("Suchen");
 searchMenu.add(searchItem);

 // Jedem Menüeintrag den Listener zuordnen
 TAListener TALi = new TAListener(this);
 openItem.addActionListener(TALi);
 saveItem.addActionListener(TALi);
 closeItem.addActionListener(TALi);
 searchItem.addActionListener(TALi);

 TextAnzeigePanel = new JEditorPane();
 JScrollPane JSP  = new JScrollPane(TextAnzeigePanel);

 this.getContentPane().add(JSP,"Center");
}

public void searchWord(String word, boolean caseSensitive)
{
  String Text = TextAnzeigePanel.getText();
  int count = 0;
  StringTokenizer stok = new StringTokenizer(Text," \n\t,.;");
  while (stok.hasMoreTokens())
  {
    if(caseSensitive)
     {
       if(stok.nextToken().equals(word))
       {
```

```
            count++;
          }
       }
      else
       {
         if(stok.nextToken().equalsIgnoreCase(word))
         {
           count++;
         }
       }
   }
  String result =  "Das Wort \""+word+"\" kommt "+count+"-mal vor.";
  JOptionPane.showMessageDialog(this,
  result,"Suchergebnis",
  JOptionPane.INFORMATION_MESSAGE);

}

public static void main(String[] args)
{
  Editor Edi = new Editor();
  Edi.showIt("Javakurs Editor");
}

class TAListener implements ActionListener{
  private JFrame parent;

  TAListener(JFrame p)
  {
    parent = p;
  }

  public void actionPerformed(ActionEvent evt)
   {
   if (evt.getSource() instanceof JMenuItem)
   {
    String command = evt.getActionCommand();
    // command enthält jetzt die Beschriftung des gewählten Menüeintrags
    if(command.equals("Öffnen"))
    {
     System.out.println("Menüeintrag "+command+" gewählt");
     JFileChooser FC = new JFileChooser(path);

     int returnVal = FC.showOpenDialog(parent);
     if(returnVal == JFileChooser.APPROVE_OPTION)
     {
```

```
          selectedFile =  FC.getSelectedFile();
          try{
            FileReader fr = new FileReader(selectedFile);
            TextAnzeigePanel.read(fr,null);
            fr.close();
          }catch(IOException e){
          System.out.println("Probleme beim öffnen oder Lesen von "+selectedFile.getName())
          }
         }//if
       }
       else if(command.equals("Speichern"))
       {
        System.out.println("Menüeintrag "+command+" gewählt");
        try{
          FileWriter fw = new FileWriter(selectedFile);
          fw.write(TextAnzeigePanel.getText());
          fw.close();
          }catch(IOException e){
           System.out.println("Probleme beim Schreiben in "+selectedFile.getName());
          }
       }
       else if (command.equals("Schliessen"))
       {
        System.out.println("Menüeintrag "+command+" gewählt");
        System.exit(0);
       }
       else if (command.equals("Suchen"))
       {
        System.out.println("Menüeintrag "+command+" gewählt");
        if (SDia == null)
        {
          SDia = new SearchDialog(parent);
          ITO  = new InfoTransferObject();
        }
        SDia.showIt(ITO);
        if(ITO.getSearch() && (ITO.getSerchWord().length() > 0))
         {
          searchWord(ITO.getSerchWord(),ITO.getCaseSensitive());
         }

       }

       }//if instanceOf
     }// method actionPerformed
    }//class TAListener
}
```

Kapitel 7

Einfache Graphik mit Swing

Zur Darstellung von Graphiken in Swing benutzt man Panels, das heißt die Klasse `JPanel`, als Zeichenflächen. Die ganze rechteckige Fläche eines Panels kann als *Zeichenfläche* genutzt werden. Die Klasse `JPanel` enthält die Methode `paintComponent`. Diese Methode zeichnet das zugehörige Panel. Wir haben diese Methode bei der Besprechung des Panels in Kapitel 3 nicht erwähnt, weil sie vom Betriebssystem beziehungsweise der *Virtual Machine* (VM) von Java automatisch aufgerufen wird wenn ein Neuzeichnen des Panels ansteht, zum Beispiel bei Größenänderungen der Mutterkomponente. Die Methode `paintComponent` sollte **nie** direkt aufgerufen werden. ! Wenn man ein Neuzeichnen einer Swing-Komponente erreichen will, so ruft man die `repaint()`-Methode dieser Komponente auf. Das Neuzeichnen geschieht nicht ! sofort; `repaint` teilt der (VM) nur mit, dass neu gezeichnet werden soll. Die VM zeichnet dann zu einem geeigneten Zeitpunkt neu. Insbesondere bei Verwendung ! von parallelen Prozessen (*Threads*) führt das oft zu lustigen Effekten (halbfertige Zeichnungen usw.).

7.1 Die Methode `paintComponent`

Die Methode `paintComponent` ist wie folgt definiert:

```
public void paintComponent(Graphics g)
```

Übergeben wird ein Parameter g vom Typ `Graphics`. Dies ist eine abstrakte Klasse aus der AWT-Bibliothek. Ein Objekt dieser Klasse entspricht einem *Graphic context* in C oder C++. Die `Graphics`-Objekte bilden den „Vermittler“ zwischen der

Java-Anwendung und dem Teil des Betriebssystems, das das eigentliche Zeichnen vornimmt. Die Klasse `Graphics` stellt auch die *Graphik-Befehle* zum Zeichen von Linien usw. breit. Mehr dazu im Abschnitt 7.2 Da wir `paintComponent` ohnehin nicht selbst aufrufen wollen, soll uns dies als Erklärung der Klasse `Graphics` genügen.

Um nun eigene Zeichnungen in ein Panel einzufügen, erweitern wir `paintComponent` um eigene Zeichen-Befehle. Dies geschieht, in dem wir die Methode `paintComponent` *überschreiben*. Natürlich möchten wir sicherstellen, dass die ursprüngliche Funktion von `paintComponent`, nämlich das bloße Panel zu zeichnen, erhalten bleibt. Wir rufen daher beim Überschreiben zunächst die Original-Methode mittels der `super`-Methode auf. Dann folgen unsere eigenen Graphik-Anweisungen. Die Struktur der überschriebenen Methode sieht somit so aus:

```
public void paintComponent(Graphics g)  {
     super.paintComponent(g);

     // Eigene Graphik-Befehle

}
```

7.2 Die Graphik-Befehle

Kommen wir nun zu den Befehlen, die die Klasse `Graphics` zum Zeichnen bereitstellt. Aus der Fülle von etwa 40 Zeichen-Befehlen wollen wir nur einige vorstellen.

```
drawLine(int xstart, int ystart, int xende, int yende)
drawOval(int xlinks, int yoben, int breite, int höhe)
drawString(String text,int links, int unten)

fillOval(int xlinks, int yoben, int breite, int höhe)

setColor(color farbe)
```

`drawLine` zeichnet eine Stecke zwischen den Punkten mit den Koordinaten $(xstart, ystart)$ und $(xende, yende)$.

`drawOval` zeichnet den Rand einer Ellipse, deren Achsen parallel zu den Koordinatenachsen liegen. Die linke obere Ecke des die Ellipse umschließenden achsenparallelen Rechtecks ist $(xlinks, yoben)$, wir nennen dies den *Bezugspunkt*.

Die waagerechte Achse der Ellipse ist *breite* + 1 Pixel lang, die senkrechte *hoehe* + 1.

`drawString` zeichnet einen den Text `text`. Legt man ein achsenparalleles Rechteck um den Text, so hat dessen untere linke Ecke die Koordinaten (*links*, *unten*) (*Bezugspunkt*). Dies stimmt nur bedingt: Unterlänge von Buchstaben können auch tiefer reichen. !

`fillOval` zeichnet eine ausgefüllte Ellipse, sonst wie `drawOval`.

`setColor` setzt die aktuelle Zeichenfarbe auf `farbe`.

Aufgabe 10: Erkläre was passiert, wenn man den `super`-Aufruf in `paintComponent` wegläßt.

7.3 Eine einfache Graphikanwendung

Die folgenden beiden Klassen definieren einen Frame `SimpleGraphicsFrame` und ein Panel `SimpleGraphicsPanel`. In die PaintComponent Methode des Panels sind einige `draw`-Befehle eingehängt worden. Beim der ersten Ellipse und beim Text folgt je ein `fillOval`-Befehl, der den jeweiligen Bezugspunkt kennzeichnet. Die Klasse mit der Startmethode geben wir nicht an.

Datei: `SimpleGraphicsFrame.java`

```
package kurs.SimpleGraphics;

import java.awt.*;
import javax.swing.JFrame;
import kurs.SimpleFrame.*;

public class SimpleGraphicsFrame extends SimpleFrame
{
  SimpleGraphicsPanel SGP = new SimpleGraphicsPanel();

  public SimpleGraphicsFrame()
  {
    this.setTitle("Einfache Graphik");
    this.getContentPane().add(SGP);
```

```
    pack();
  }
}
```

Datei: SimpleGraphicsPanel.java

```
package kurs.SimpleGraphics;

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

public class SimpleGraphicsPanel extends JPanel
{
   public SimpleGraphicsPanel()
   {
     this.setBackground(Color.white);
     this.setPreferredSize(new Dimension(300,300));
   }

   public void paintComponent(Graphics g)
   {
     super.paintComponent(g);
     g.setColor(Color.black);
     g.drawLine(10,10,100,100);
     g.setColor(Color.red);
     g.drawLine(10,100,100,10);
     g.setColor(Color.green);
     g.drawOval(120,60,70,40);
     g.fillOval(119,59,3,3);
     g.setColor(Color.cyan);
     g.fillOval(10,120,100,60);
     g.setColor(this.getBackground());
     g.fillOval(50,140,100,60);
     g.setColor(Color.blue);
     g.drawString("Swing ist gut.",100,200);
     g.fillOval(99,199,3,3);
  }
 }
```

Kapitel 8

Mausabfragen

Um die Maus[1] in einer Swing-Komponenten `SK` abzufragen, muss man dieser Komponenten einen Aufpasser zuordnen, der die Maus überwacht. Da man manchmal nur am Status der Maustasten interessiert ist und manchmal nur an Mausbewegungen, hat man in JAVA für beide Fälle je einen Listener definiert. Es sind dies die beiden Interfaces `MouseListener` beziehungsweise `MouseMotionListener` aus der Bibliothek `java.awt.event`. Man fügt einen Aufpasser je nach Typ mit einem der folgenden Befehle zur Komponente `SK` hinzu

```
SK.addMouseListener(MouseListener mausAupasser);
SK.addMouseMotionListener(MouseMotionListener bewegungsAupasser);
```

Wichtig ist: Der Aufpasser wird nur aktiv, wenn die Maus in der Komponente ist, der er zugeordnet wurde. Klickt man also irgendwo anders, so wird der Aufpasser nichts unternehmen. !

8.1 Maustasten

Das Interface `MouseListener` erlaubt Reaktionen auf die Bedienung der Maustasten und auf das Eintreten der Maus in eine Swing-Komponente sowie das Verlassen einer solchen. Es verlangt die Implementation *aller* der folgenden Methoden:

[1] Wir reden im diesem Kapitel oft von der „Maus“, wenn wir eigentlich den „Bezugspunkt des Mauszeigers meinen“.

```
void mouseClicked(MouseEvent e)
void mouseEntered(MouseEvent e)
void mouseExited(MouseEvent e)
void mousePressed(MouseEvent e)
void mouseReleased(MouseEvent e)
```

`mouseClicked` Diese Methode wird automatisch aufgerufen, wenn eine Maustaste geklickt wird. Welche Taste es ist, kann man dem übergebenen Mouse-Event entnehmen; mehr dazu unten.

`mouseEntered` Diese Methode wird automatisch aufgerufen, wenn die Maus die Komponente betritt, der der MouseListener zugeordnet wurde.

`mouseExited` Diese Methode wird automatisch aufgerufen, wenn die Maus die Komponente verlässt, der der MouseListener zugeordnet wurde.

`mousePressed` Diese Methode wird automatisch aufgerufen, wenn eine Maustaste gedrückt wird. Welche Taste es ist, kann man dem übergebenen Mouse-Event entnehmen; mehr dazu unten.

`mouseReleased` Diese Methode wird automatisch aufgerufen, wenn eine Maustaste nach Drücken wieder losgelassen wird. Welche Taste es ist, kann man dem übergebenen Mouse-Event entnehmen; mehr dazu unten.

Oftmals ist man nur an einer der genannten Methoden interessiert. Die anderen vier muss man dann zwar trotzdem implementieren, aber mit leerem Methodenrumpf. Um dies zu vermeiden, haben die Entwickler von JAVA so genannte *Adapter-Klassen* als „syntaktischen Zucker“ geschaffen. Die Namen der Adapter**klassen** stimmen mit denen der Listener**interfaces** überein, und sie implementieren alle vom Interface geforderten Methoden. Verwendet man Adapterklassen, so muss man nur die **benötigten** Methoden **überschreiben**, statt **alle** zu **implementieren**. Die Klassen sind `MouseAdapter` beziehungsweise `MouseMotionAdapter`.

8.2 Mausbewegungen

Das Interface `MouseMotionListener` beziehungsweise die zugehörige Adapterklasse `MouseMotionAdapter` besitzen die folgenden Methoden:

```
void mouseMoved(MouseEvent e)
void mouseDragged(MouseEvent e)
```

`mouseMoved` Diese Methode wird automatisch aufgerufen, sobald die Maus bewegt wird. Dies natürlich nur, wenn sie in der zugeordneten Komponente ist.

`mouseDragged` Wird aufgerufen, wenn zusätzlich zur Bewegung noch eine Maustaste gedrückt ist.

8.3 Mausereignisse

Die gerade beschriebenen Methoden bekommen als Parameter ein Maus-Ereignis (`MouseEvent`) übergeben. Dies enthält Information über das Ereignis, zum Beispiel die Koordinaten, an denen es aufgetreten ist oder die Maustaste, die benutzt wurde. Einige dieser Informationen kann man einfach durch Methoden der Klasse `MouseEvent` selbst abfragen. Bei anderen erfordert dies viel Aufwand, den man aber durch Verwendung von *Hilfsklassen* (*Utilities*) verringern kann.

Die Abfrage der Position des Mauszeigers, an der das Ereignis ausgelöst wurde geschieht auf die erste Art so:

```
int getX()
int getY()
```

Die Abfrage der gedrückten Taste ist in AWT recht aufwendig und erfordert die Kenntnis der AWT-internen Kodierung von Ereignissen. Die Klasse `SwingUtilities` stellt komfortable Methoden dazu bereit, deren Namen wohl selbsterklärend sind:

```
boolean SwingUtilities.isLeftMouseButton(MouseEvent e)
boolean SwingUtilities.isMiddleMouseButton(MouseEvent e)
boolean SwingUtilities.isRightMouseButton(MouseEvent e)
```

8.4 Anwendung der Mausabfragen

Nun wollen wir die gerade vorgestellten Methoden zur Mausabfrage anwenden. Als Komponente für die Mausabfragen verwenden wir ein 300×300 Pixel großes Panel, das wir in der Klasse `MouseEventPanel` definieren. Es wird als zentrale Komponente in einen Rahmen der Klasse `MouseEventFrame` geklebt. Das Panel besitzt zunächst noch keine Methoden; es dient nur als Testfläche für die Maus.

Darunter (South) kleben wir ein weiteres Panel vom Typ `StatusPanel`, das uns sie Mausposition im `MouseEventPanel` in JAVA-Pixel-Koordinaten zeigen soll. Diese Panel hat ein 1×3 Raster-Layout, in das 3 Label eingefügt werden. Das erste enthält den Text „Position“ das zweite und dritte die x- beziehungsweise y-Koordinate der Maus. Das Status-Panel hat Methoden zum Setzen der Koordinaten und zum Setzen der Schriftfarbe.

Die Funktionalität soll wie folgt sein:

- Solange sich die Maus im `MouseEventPanel` befindet, sollen im jnidxbspStatusPanel die aktuellen Maus-Koordinaten ständig angezeigt werden.
- Ist die Maus nicht im `MouseEventPanel` sollen die Koordinaten $(-1, -1)$ angezeigt werden.
- Ist die Maus im `MouseEventPanel` so soll die Anzeige im Status-Panel rot sein und sonst schwarz.

Für die erste Funktion verwenden wir der Bequemlichkeit wegen einen Adapter-Klasse. Die Klasse `MausPositionsAdapter` überschreibt die Methode `mouseMoved` der Oberklasse `MouseAdapter` so, dass in ihr die aktuellen Mauskoordinaten im Status-Panel gesetzt werden. Damit ein Zugriff auf das Status-Panel möglich ist, wird dem Konstruktor `MausPositionsAdapter` eine Referenz auf das Status-Panel übergeben. Da bei jeder Mausbewegung die Methode `mouseMoved` aufgerufen wir, sind im Status-Panel immer die aktuellen Koordinaten zu sehen. Der Adapter wird dem `MouseEventPanel` mittels `addMouseMotionListener` zugeordnet.

Für die zweite und dritte Funktion übernimmt ein Maus-Adapter, nämlich die Klasse `MausAdapter` die Arbeit. Die dort überschriebenen Methoden `mouseEntered` und `mouseExited` setzen die Farben im Status-Panel und beim Verlassen des `MouseEventPanel` auch die Koordinaten auf $(-1, -1)$. Der Adapter wird dem `MouseEventPanel` mittels `addMouseListener` zugeordnet.

Damit ist unsere erste mausfähige Anwendung fertig. Die Klassen – diesmal inclusive der Startklasse `MouseEventTest` – sind im Folgenden angegeben.

Datei: `MouseEventFrame.java`

```
package kurs.MouseEvents;

import java.awt.*;
```

```
import kurs.SimpleFrame.SimpleFrame;
public class MouseEventFrame extends SimpleFrame
{
  MouseEventPanel MEPanel = new MouseEventPanel();
  StatusPanel     StPanel = new StatusPanel();

  public MouseEventFrame()
  {
    this.setTitle("Mausabfrage");
    this.getContentPane().add(MEPanel,"Center");
    this.getContentPane().add(StPanel,"South");
    pack();

    MausPositionsAdapter MPosAdpt = new MausPositionsAdapter(StPanel);
    MEPanel.addMouseMotionListener(MPosAdpt);

    MausAdapter MAdpt = new MausAdapter(StPanel);
    MEPanel.addMouseListener(MAdpt);

  }
}
```

Datei: `MouseEventPanel.java`

```
package kurs.MouseEvents;

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

public class MouseEventPanel extends JPanel
{
   public MouseEventPanel()
  {
      setBackground(Color.white);
      setPreferredSize(new Dimension(300,300));
  }
}
```

Datei: `MausPositionsAdapter.java`

```
package kurs.MouseEvents;

import java.awt.event.*;

public class MausPositionsAdapter extends MouseMotionAdapter
{
  private StatusPanel SP;

  public MausPositionsAdapter(StatusPanel s)
  {
    SP = s;
  }
  public void mouseMoved(MouseEvent evt)
   {
   SP.setCoordinates(evt.getX(),evt.getY());
   }
}
```

Datei: MausAdapter.java

```
package kurs.MouseEvents;

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

public class MausAdapter extends MouseAdapter
{
private StatusPanel SP;

  public MausAdapter(StatusPanel s)
  {
    SP = s;
  }

  public void mouseEntered(MouseEvent e)
   {
     SP.setColor(Color.red);
   }

  public void mouseExited(MouseEvent e)
   {
     SP.setColor(Color.black);
     SP.setCoordinates(-1,-1);
```

```
    }
}
```

Datei: `MouseEventTest.java`

```
package kurs.MouseEvents;

public class MouseEventTest
{

  public MouseEventTest()
  {
    MouseEventFrame MEF = new MouseEventFrame();
    MEF.showIt();
  }
  public static void main(String[] args)
  {
    MouseEventTest mouseEventTest1 = new MouseEventTest();
  }
}
```

Kapitel 9

Interaktive Graphik

In diesem Kapitel wollen wir die in Kapitel 8 entwickelte Anwendung um einen interaktiven Teil erweitern. Es sollen die folgenden Funktionen implementiert werden:

- Beim Klicken der linken Maustaste soll ein kleiner gefüllter Kreis an dieser Position gezeichnet werden.
- Beim Klicken der rechten Maustaste soll der der Position am nächsten gelegenen Kreis entfernt werden. Dies aber nur, wenn die Entfernung zwischen der Mausposition und dem nächstgelegenen Kreismittelpunkt weniger als 30 Pixel (in der Euklidischen Metrik des $\mathbb{R}^2$) beträgt. Haben mehrere Kreise denselben Abstand, so kann irgendeiner davon gewählt werden.
- In der Status-Leiste soll zusätzlich die aktuelle Anzahl der Kreise Angezeigt werden.

Beginnen wir mit dem letzten Punkt: In der Klasse Status Panel wird ein 1×5 Raster-Layout verwendet, das zusätzlich zwei Label für den Text „Kreise:“ und die Anzahl der Kreise enthält. Hinzu kommt eine Methode, mit der man die Anzahl der Kreise setzen kann.

9.1 Eine einfache Datenstruktur für geometrische Objekte

Die Kreise zeichnen zu lassen, ist mit unserem Wissen über Mausabfragen sicher nicht schwierig. Wir müssen hier aber etwas Verwaltungsaufwand treiben, um Kreise auch wieder löschen zu können. Die beiden in diesem Abschnitt beschriebenen Datenstrukturen `Kreis` und `KreisVerwaltung` stellen eine Möglichkeit dar, graphische Objekte zu verwalten. Sie sind leicht auf komplexere graphische Objekte als Kreise erweiterbar, siehe Kapitel 10

Die Klasse `Kreis` hat zwei ganzzahlige Felder `x` und `y` die die Koordinaten des Mittelpunktes des Kreises auf nehmen. Der Radius ist konstant 7. Dem objektorientierten Paradigma von JAVA folgend, kann auch nur diese Klasse wissen, wie ein Kreis zu zeichnen ist. Daher besitzt `Kreis` auch eine Methode `zeichne`, die dies übernimmt. Zum Zeichnen ist ein Graphik-Refernz (`Graphics`) notwendig, die der Methode `zeichne` als Parameter übergeben wird. Zusätzlich verfügt die Klasse `Kreis` über die Methode `distance(int x1, int y1)`, die den Euklidischen Abstand des Kreismittelpunktes zu dem Punkt mit den Koordinaten (x_1, y_1) bestimmt. Dies ist zum Löschen wichtig.

Die Klasse `KreisVerwaltung` übernimmt den Hauptteil der Arbeit, außer der eigentlichen Graphik. Ihr Kernstück ist ein Objekt vom Type `Vector`, also eine in JAVA vordefinierte dynamische Datenstruktur. Vektoren in JAVA dürfen nicht mit
! den gleichnamigen algebraischen Objekten verwechselt werden. JAVA-Vektoren verwalten beliebige JAVA-Objekte auf. Man kann Objekte einfügen, entfernen und sich die Referenz auf ein Objekt geben lassen, ohne es zu entfernen. Enthält ein Vektor n Objekte, so sind diese von 0 bis $n-1$ durchnummeriert.

Der in der `KreisVerwaltung` verwendete Vektor `Kreise` erhält nur Objekte vom Typ `Kreis`. Neben dem Konstruktor bietet `KreisVerwaltung` die folgenden Methoden:

```
void add(Kreis k)
void removeNearest(int x, int y)
zeichne(Graphics g)
anzahl()
```

`add(k)` Fügt den Kreis `k` in den Vektor `Kreise` ein.

`removeNearest(x,y)` Bestimmt unter allen Kreisen, die aktuell im Vektor `Kreise` gespeichert sind einen, dessen Mittelpunkt den kleinsten Abstand zu dem

Punkt mit den Koordinaten (x, y) hat. Ist dieser Abstand kleiner als 30 (Pixel), so wird der Kreis aus dem Vektor `Kreise` entfernt.

`zeichne()` Zeichnet alle Kreise, die aktuell im Vektor `Kreise` gespeichert sind durch aufrufen von deren `zeichne`-Methoden. Das erhaltene `Graphics`-Objekt `g` wird dabei weitergegeben.

`anzahl()` Liefert die aktuelle Anzahl der im Vektor `Kreise` gespeicherten Kreise.

Datei: `Kreis.java`

```
package kurs.InteractiveGraphic;

import java.awt.Graphics;

public class Kreis
{
  private int x,y;

  public Kreis(int x, int y)
  {
   this.x = x;
   this.y = y;
  }

  public void zeichne(Graphics g)
  {
   g.fillOval(x-3,y-3,7,7);
  }

  public double distance(int x1, int y1)
  {
   return(Math.sqrt(Math.pow(x-x1,2)+Math.pow(y-y1,2) ));
  }

}
```

Datei: `KreisVerwaltung.java`

```
package kurs.InteractiveGraphic;
```

```
import java.awt.event.MouseMotionAdapter;
import java.awt.Graphics;
import java.util.Vector;

public class KreisVerwaltung
{
  private Vector Kreise;

  public KreisVerwaltung()
  {
   Kreise = new Vector();
  }

  public void add(Kreis k)
   {
     Kreise.add(k);
   }

  public void removeNearest(int x1, int y1)
  {
    Kreis kr;
    double minDist = Double.MAX_VALUE;
    int    minDistIndex = -1;
    for (int i=0 ; i < Kreise.size() ; i++)
    {
      kr = (Kreis)(Kreise.elementAt(i));
      if(kr.distance(x1,y1) < minDist)
      {
       minDist = kr.distance(x1,y1);
       minDistIndex = i;
      }// if
    }//for i
    if ((minDistIndex >= 0) && (minDist < 30))
     {
       Kreise.removeElementAt(minDistIndex);
     }//if
  }//methode

  public void zeichne(Graphics g)
  {
    for (int i=0 ; i < Kreise.size() ; i++)
    {
      ((Kreis)(Kreise.elementAt(i))).zeichne(g);
    }//for i
   }//methode
```

```
  public int anzahl(){
    return(Kreise.size());
  }//methode
}
```

9.2 Erweiterungen der anderen Klassen

Die Frame-Klasse `InteractiveFrame` ist gegenüber der Klasse `MouseEventFrame` aus Kapitel 8 unverändert.

Die Panel-Klasse `InteractivePanel` wird gegenüber der Klasse `MouseEventPanel` aus Kapitel 8 erweitert. Hinzu kommt ein Objekt vom Typ `KreisVerwaltung`. Außerdem gibt es Methoden zum Hinzufügen und Löschen eines Kreises. Sie tun eigentlich nichts anders als die Daten der Mausposition an die gleichnamigen Methoden der Klasse `KreisVerwaltung` weiterzureichen. Außerdem kann die Anzahl der Kreise abgefragt werden. Auch dabei wird die Methode `anzahl` von `KreisVerwaltung` bemüht.

Datei: `InteractivePanel.java`

```
package kurs.InteractiveGraphic;

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

public class InteractivePanel extends JPanel
{

KreisVerwaltung Kreise;

public InteractivePanel()
  {
      Kreise = new KreisVerwaltung();
      setBackground(Color.white);
      setPreferredSize(new Dimension(300,300));
  }

public  void paintComponent(Graphics g)
 {
```

```
   super.paintComponent(g);
   Kreise.zeichne(g);
 }

public void addKreis(int x, int y)
{
 Kreise.add(new Kreis(x,y));
 repaint();
}

public void removeNearest(int x, int y)
{
  Kreise.removeNearest(x,y);
  repaint();
}
public int getKreiszahl(){
   return(Kreise.anzahl());
  }
}
```

Von den beiden Adaptern wird nur der `MausAdapter` um die Methode `mouseClicked` erweitert. Abhängig davon, ob die linke oder rechte Maustaste[1] geklickt wurde, wird entweder ein Kreis mit den aktuellen Mauskoordinaten dem Panel hinzugefügt (mittels `addKreis(e.getX(),e.getY())`) oder der nächstgelegene wird gelöscht (mittels `removeNearest(e.getX(),e.getY())`). In jedem Falle wird die Anzahl der Kreise in der Statusleiste auf den neuesten Stand gebracht (mittels `setKreiszahl(getKreiszahl())`). Der Konstruktor der Klasse `MausAdapter` erhält nun Referenzen auf beide benutzten Panels.

Datei: `MausAdapter.java`

```
package kurs.InteractiveGraphic;

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

public class MausAdapter extends MouseAdapter
{
private StatusPanel SP;
private InteractivePanel IAP;
```

[1] Wir gehen von einer Zweitastenmaus aus.

```
  public MausAdapter(StatusPanel s,InteractivePanel i)
  {
    SP  = s;
    IAP = i;
  }

  public void mouseEntered(MouseEvent e)
   {
     SP.setColor(Color.red);
   }

  public void mouseExited(MouseEvent e)
   {
     SP.setColor(Color.black);
     SP.setCoordinates(-1,-1);
    }

  public void mouseClicked(MouseEvent e)
  {
   if(SwingUtilities.isLeftMouseButton(e))
   {
    IAP.addKreis(e.getX(),e.getY());
    SP.setKreiszahl(IAP.getKreiszahl());
   }
   else
   {
    IAP.removeNearest(e.getX(),e.getY());
    SP.setKreiszahl(IAP.getKreiszahl());
   }
  }

}
```

Damit ist unsere interaktive graphische Anwendung komplett und kann genutzt werden. Abbildung 9.1 zeigt das Resultat.

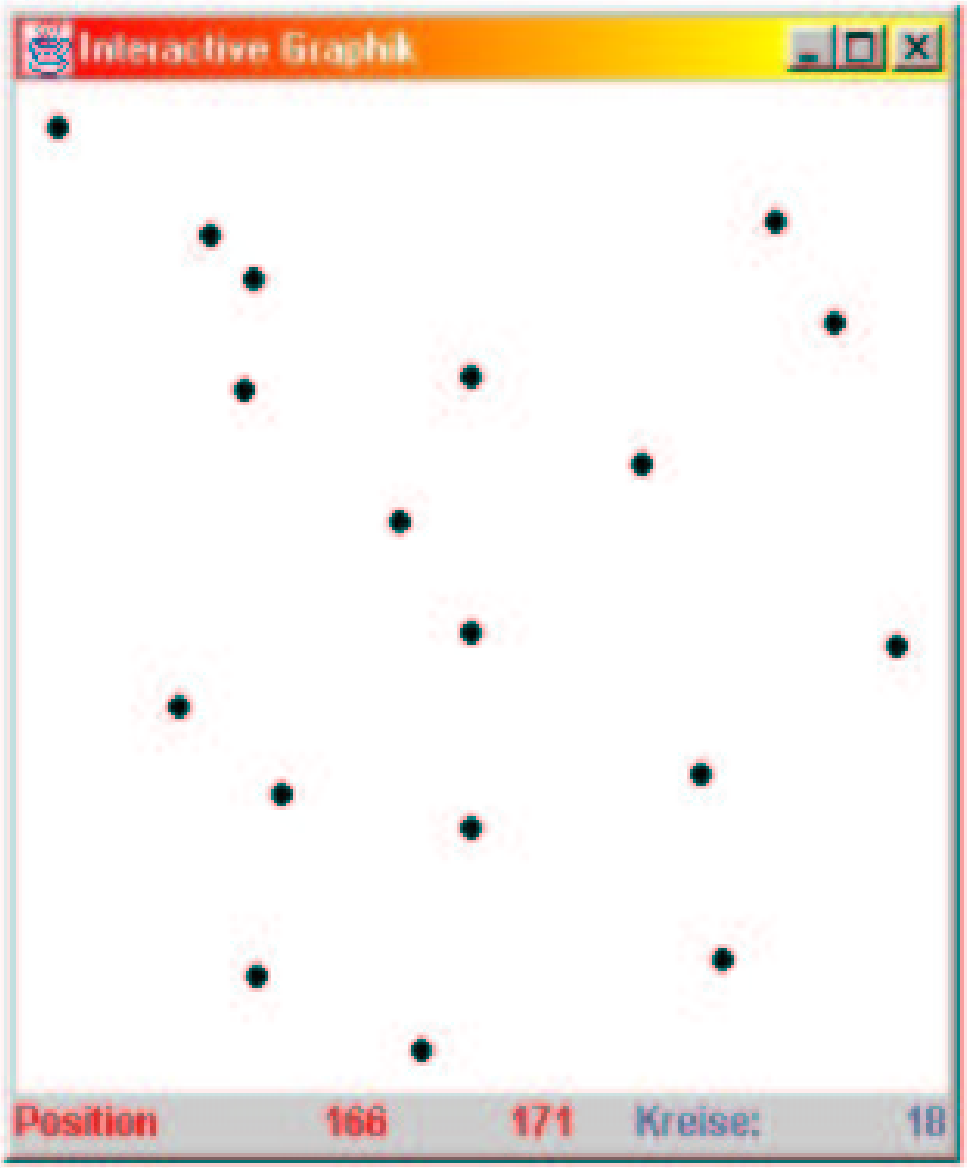

Abbildung 9.1: Die interaktive Anwendung.

Kapitel 10

Vektor-Graphik

Dieser Abschnitt führt keine neuen Swing-Komponenten ein. Es soll vielmehr der Umgang mit graphischen Objekten exemplarisch beschrieben werden.

Bei Graphiken unterscheidet man im Wesentlichen zwei Typen: *Pixelgraphiken* und *Vektorgraphiken*. Bei Pixelgraphiken ist die Größe der Graphik (im Bildpunkten) festgelegt und für jeden Bildpunkt ist die Helligkeit und/oder der Farbwert (in einem geeigneten Farbsystem) gegeben. Beim Vergrößern oder Verkleinern einer Pixelgraphik müssen daher Bildpunkte neu berechnet (interpoliert) beziehungsweise zusammengefasst (gemittelt) werden. Pixelgraphiken benutzt man vor allem für Darstellungen von Bildern und Fotos. Typische Formate sind GIF, TIFF, JPEG und PNG. Vektorgraphiken sind keine expliziten Darstellungen sondern *Beschreibungen* von Graphiken. Diese Beschreibungen kann man benutzen, um die Graphik in beliebiger Größe darzustellen. Die Beschreibung enthält im Allgemeinen die Cartesischen Koordinaten der graphischen Objekte, also zum Beispiel „es gibt eine grüne Linie von Punkt $(0, 0)$ nach $(1, 2)$“. Es ist klar, dass sich diese Art der Darstellung eher für detailärmere Zeichnungen eignet. Formate sind zum Beispiel `FIG` (Xfig) `CDR` (CorelDraw).

Im Folgenden wird das Programmpaket `Rebo` beschrieben, das interaktive und passive Vektorgraphiken darstellen, speichern und laden kann. Hier kurz eine Zusammenfassungen der Funktionen

- Es werden einige grundlegende graphische Objekte bereitgestellt: Strecke, Dreieck, Kreis und Polygonzug.
- Andere Programme können solche Objekte erzeugen und an Rebo übergeben, wo sie dargestellt werden.

- Andere Programme können vorhandene graphische Objekte löschen.
- Mit der Maus können Strecken der Graphik hinzugefügt werden.
- Mit der Maus können vorhandene graphische Objekte gelöscht werden.
- Die Graphiken können gespeichert und geladen werden.

Das `Rebo`-Paket ist eigentlich eine Erweiterung des Pakets `kurs.InteractiveGraphic` aus Kapitel 9, allerdings eine erhebliche. Wegen der größeren Anzahl von Graphiktypen und der umfangreicheren Funktionalität werden viele Hilfsklassen definiert, die allerdings unabhängig von Swing sind. Für ein Listing ist das Paket mit immerhin 19 Klassen zu groß, es jedoch von der Web-Seite der Kurses heruntergeladen werden.

10.1 Koordinatensysteme und Skalierungen

Das in Abschnitt 2.1 vorgestellte Koordinatensystem von Swing ist diskret (pixelorientiert) und „steht auf dem Kopf“ (linksorientiert). Das Cartesische Koordinatensystem ist hingegen kontinuierlich und rechtsorientiert. Man könnte natürlich cartesische Koordinaten durch (Ab-)Rundung in Pixelkoordinaten verwandeln und damit zeichnen. Liegen aber alle Cartesischen Koordinaten im Intervall $[0, 1)$ so würden sie alle auf 0 abgerundet; die resultierende Zeichnung wäre ein einziger Bildpunkt. Wir müssen also die Größe der Cartesischen Koordinaten berücksichtigen und diese vor der Rundung mit einem geeigneten Faktor multiplizieren. Dieser Faktor muss – abhängig von der aktuellen Größe der Zeichenfläche in Pixeln – so gewählt werden, dass die Zeichnung möglichst groß aber trotzdem vollständig zu sehen ist. Diesen Vorgang nennt man *skalieren*.

Das `Rebo`-Paket verwendet zwei Koordinatensysteme. Die Klasse `RealPoint` definiert Punkte in Cartesischen Koordinaten durch zwei `double`-Werte. Die Klasse `PixelPoint` definiert Bildpunkte durch zwei `int`-Werte. Die Klasse `GraphicsUtilities` stellt Methoden zur Umwandlung des einen Punkt-Typs in den anderen bereit. Dazu ist natürlich die Kenntnis der **aktuellen** Breite und Höhe der Zeichenfläche `GrahpicPanel` nötig. Außerdem wird die Darstellung aufgerichtet (rechtsorientiert) und es wird ein Rahmen der Zeichenfläche berücksichtigt, so dass die Zeichnung nicht ganz bis an den Rand der Fläche reichen. Die Breiten der Ränder werden durch Konstanten in `GraphicsConst` festgelegt.

In Kapitel 9 konnten nur Kreise fester (Pixel-)Größe gezeichnet werden. Hier werden allgemeinere graphische Objekte erlaubt. Für die Skalierung ist es wichtig, die Ausmaße eines solchen graphischen Objekts zu kennen. Jedem ist daher eine

`ScaleObjekt` zugeordnet. Dieses enthält die maximalen und minimalen (Cartesischen) x- und y-Koordinaten des graphischen Objekts. Zusätzlich kann man darin die aktuelle Größe der Zeichenfläche abspeichern. Damit stellt ein `ScaleObjekt` alle Information bereit, die man zur Umwandlung von Cartesischen in Pixelkoordinaten (oder umgekehrt) benötigt.

10.2 Graphische Objekte

Die Klasse `GraphicObjekt` definiert die graphischen Objekte allerdings teilweise abstrakt. Die davon abgeleiteten Objekte (zum Beispiel `Circle`) müssen diese Methoden dann implementieren. So ist die Methode `draw` zum Zeichen des Objektes für Kreise eine andere als für Polygonzüge und muss daher in den abgeleiteten Klassen `Circle` und `OpenPolygon` definiert werden.

Neben den schon erwähnten Objekten gibt es noch Klassen für Strecken (`Line`) und Dreiecke (`Triangle`). In den Konstruktoren dieser Klassen werden jeweils die relevanten Parameter des Objekts als Cartesische Koordinaten (`RealPoint`) übergeben. Bei einem Dreieck sind es zum Beispiel die drei Eckpunkte, beim Kreis der Mittelpunkt und der Radius. Die Graphik-Objekte speichern somit Vektordarstellungen, die unabhängig von der Darstellung auf dem Bildschirm sind. Erst in den `draw`-Methoden der Graphik-Objekte werden diese mit Hilfe des Skalierungs-Objekts der aktuellen Größe der Zeichenfläche angepasst.

Zu jedem Graphik-Objekt gehört ein `anchor`. Dies ist ein Punkt, der als Referenzpunkt für Auswahl eines Graphik-Objekts mit der Maus dient. Er wird auf dem Bildschirm mit einem kleinen, orangefarbenen Kreis gekennzeichnet. Es ist hier immer der erste im Konstruktor übergebenen Punkt, bei Kreisen der Mittelpunkt. Im `Rebo`-Paket wird der Referenzpunkt zum Löschen benutzt. Beim Klick mit der rechten Maustaste wird ein Graphik-Objekt gelöscht, dessen Referenzpunkt am dichtesten am Mauszeiger liegt, allerdings nur dann, wenn der Abstand kleiner als 30 Pixel ist.

Die Hauptarbeit in diesem Paket leistet die Klasse `GraphicAdministration`, die die `KreisVerwaltung` aus Kapitel 9 ersetzt. Sie verwaltet die Graphik-Objekte, fügt welche hinzu oder löscht sie und sorgt dafür, dass alle gezeichnet werden. Damit alle Graphik-Objekte in die Zeichenflächen passen, berechnet die `GraphicAdministration` auch die minimalen und maximalen x- und y-Koordinaten, die in den Graphik-Objekten vorkommen (in der Methode `updateScale`). Diese Werte werden zusammen mit der aktuellen Größe der Zeichenfläche im Skalierungs-Objekt `mainScale` gespeichert. Dieses Skalierungs-Objekt wird jedem Graphik-Objekt zum Zeichnen

übergeben, so dass alle gleich skaliert werden.

10.3 Weitere Funktionen

Das Status-Panel zeigt nun auch noch die aktuelle Anzahl der Graphiken an sowie die Pixel- und Cartesischen Koordinaten.

Mit der linken Maustaste können Strecken gezeichnet werden. Erster Klick: Streckenanfang (und Referenzpunkt), zweiter Klick: Streckenende. Dritter Klick: Anfang der nächsten Strecke usw.

Das Programm `ReboTest` erzeugt einige Graphik-Objekte und lässt sie zeichnen. Eines davon (die Raute oben im Bild) wird von `ReboTest` nach 8 Sekunden wieder entfernt. Damit das geht, muss `ReboTest` eine „Referenz“ auf die Raute besitzen. Hier wir beim Einfügen einer Graphik mittels `addGraphic` der Graphik eine eindeutige Nummer (Griff/handel) zugeordnet und dem einfügenden Programm zurückgegeben. Diese Nummer benutzt `ReboTest`, um die Raute löschen zu lassen.

10.4 Speichern von Vektor-Graphiken

Wir speichern eine Graphik als eine Liste von Beschreibungen der in ihr vorkommenden Graphik-Objekte. Die Dateien erhalten die Endung `.rvg` für „Rebo-Vektor-Graphik“. Zur Demonstration speichern wir im ASCII-Format jeweils ein Graphik-Objekt pro Zeile. Die Graphik-Objekte besitzen die Methode `toString`. Sie erzeugt eine Zeichenkette, die mit der Beschreibung der Klasse beginnt (zum Beispiel „OPOL“ für `OpenPolygon`). Dann folgen durch Tabulatoren (`\t`) separiert die Punkte in der Form: x-Koordinate, y-Koordinate (und ggf. weitere Parameter wie der Radius beim Kreis).

Zum Laden löse man die folgende

Aufgabe 11: Ergänze das Rebo-Paket so, dass auch `rvg`-Dateien eingelesen werden können und die darin gespeicherte Graphik wieder angezeigt wird. Dies erfordert wahrscheinlich den Eingriff in viele Klassen.

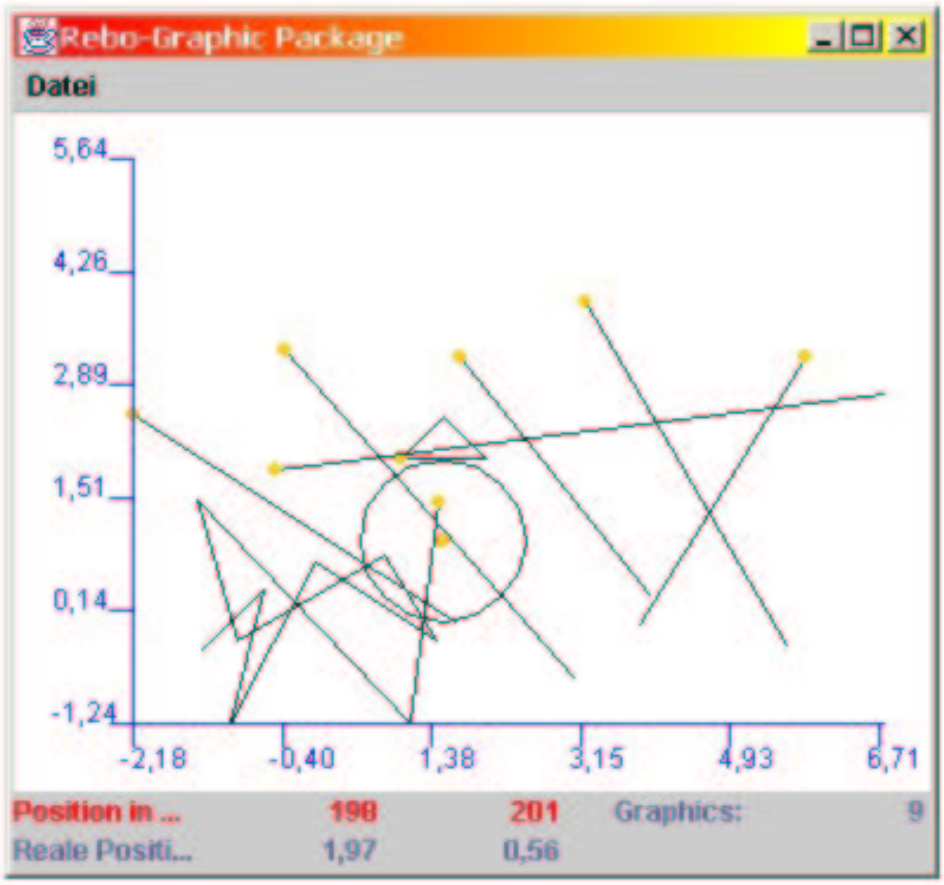

Abbildung 10.1: Beispiel für Rebo.

10.5 Anmerkungen zum Rebo-Paket

- Das `Rebo`-Paket ist nur rudimentär, es ist eher für die passive Darstellung als für die interaktive Erzeugung von Graphiken geeignet. Zu einem echten Graphik-Editor gehört viel mehr. Das Paket stellt aber alle wesentlichen Konzepte zur passiven und interaktiven Graphik-Darstellung bereit.

- Einige Teile sind nicht besonders effizient Implementiert, um den Code noch halbwegs überschaubar zu halten. Zum Beispiel kann man die Bestimmung der maximalen (Cartesischen) Bildgröße besser lösen als durch den Durchlauf durch die Skalierungs-Objekte aller Graphik-Objekte.

- Bei der Implementierung der Klassen `PixelPoint` und `RealPoint` wurden die Felder `x` und `y` als `public` deklariert und auf die entsprechenden `set-` und `get-`Methoden verzichtet. Dies ist die einzige Konzession an die Geschwindigkeit

(und der einzige Verstoß gegen die Objektorientierung), da diese Zugriffe sehr oft erfolgen.

- Die Benutzung der Referenzpunkt (`anchor`) zu Auswahl von Graphik-Objekten ist nicht unbedingt benutzerfreundlich. Eigentlich möchte man „irgendwo auf oder in die Nähe“ eines Objektes klicken können, um es auszuwählen. Damit das effizient möglich ist, sind komplexe Datenstrukturen notwendig. Wer Interesse hat, darüber etwas zu lernen, dem sei die Spezialvorlesung *Algorithmische Geometrie* empfohlen.

Kapitel 11

Tabellen

Für die Darstellung von zweidimensionalen Tabellen bietet Swing mit der Klasse `JTable` ein vordefinierte Komponente. Diese Klasse ist sehr leistungsfähig und umfangreich. So kann in den Zellen einer Tabelle nicht nur Text angezeigt werden, sondern es können auch Graphiken oder andere Swing-komponenten eingebettet werden. Wir reißen nur drei mögliche Verwendungen an.

11.1 Einfache statische Daten-Tabelle

Will man nur Daten tabellarisch Darstellen, die sich nicht ändern und die nicht ediert werden sollen, so kann man diese Daten als zweidimansionales Array im Konstruktor übergeben. Das Array wird als (eindimansionales) Array von Zeilen interpretiert. Zusätzlich muss man noch in einem weiteren die Spaltenüberschriften übergeben.

```
JTable(Object[][] inhalt,Object[] spaltenNamen)
```

Man beachte, dass der Typ der Arrays `Oject` ist. Man kann also beliebige Objekte übergeben, solange man sicherstellt, dass sie in den Zellen dargestellt werden können. Es sollte Klar sein, dass alle Zeilen gleichlang sind. Die Zeilen- und Spaltennumerierung beginnt jeweils bei Null. Durch die Einbettung in eine `JScrollPane` wird sichergestellt, dass die Splatenüberschriften sichtbar werden und auch beim vertikalen Rollen stets angezeigt werden.

Das folgende Listing legt eine Tabelle mit zwei Zeilen und 3 Spalten an.

Datei: `StaticTableFrame.java`

```
package kurs.Table;

import java.awt.*;
import javax.swing.*;
import kurs.SimpleFrame.SimpleFrame;

public class StaticTableFrame extends SimpleFrame
{

  public StaticTableFrame()
  {

    String[][] inhalt = {{"Zelle 1","Zelle 2","Zelle 3"},{"Zelle 4",
    "Zelle 5","Zelle 6"}};
    String[] spaltenNamen = {"Spalte 1","Spalte 2","Spalte 3"};
    JTable JTab = new JTable(inhalt,spaltenNamen);
    JScrollPane SP = new JScrollPane(JTab);
    this.getContentPane().add(SP);
  }

  public static void main(String[] args)
  {
    StaticTableFrame STF = new StaticTableFrame();
    STF.showIt();
  }
}
```

11.2 Tabellen-Modelle

Bei komplexeren Inhalten oder großen Tabellen ist die gerade beschriebene Methode der expliten Datenübergabe an `JTable` sicher ungeeeignet. Hier bietet Swing mit dem sogenannten *Tabellen-Modell* eine elegante Lösung. Ein Tabellen-Modell ist eine Beschreibung des Tabelleninhalts; die Tablle muss also gar nicht explizit Vorliegen. Jedesmal wenn die Tabelle gezeichnet wird, werden die aktuellen Daten der einzellen Zellen aus dem Modell abgerufen.

Der folgende Code erzeugt eine $(n \times n)$-Tabelle, in der die Zweierpotenzen stehen.

Benutzt wird das Modell `RiceTableModel`[1], das das abstrakte `AbstractTableModel` implementiert. Die Klasse `AbstractTableModel` verlangt die Implementation der folgenden drei Methoden.

```
public int getRowCount()
public int getColumnCount()
public Object getValueAt(int r, int c)
```

Die beiden ersten liefern die Zahl der Zeilen beziehungsweise Spalten der Tabelle zurück. Die Methode `getValueAt(r,c)` liefert eine Referenz auf den Inhalt der c-ten Zelle in der r-ten Zeile.

Datei: `RiceTableFrame.java`

```
package kurs.Table;

import java.awt.*;
import javax.swing.*;
import kurs.SimpleFrame.SimpleFrame;

public class RiceTableFrame extends SimpleFrame
{

  public RiceTableFrame()
  {
    RiceTableModel RTM = new RiceTableModel(4);
    JTable JTab = new JTable(RTM);
    JScrollPane SP = new JScrollPane(JTab);
    this.getContentPane().add(SP);

  }

  public static void main(String[] args)
  {
    RiceTableFrame TF = new RiceTableFrame();
    TF.showIt();
  }
}
```

[1] Der Name leitet sich von der Legende über die Belohnung des Erfinders des Schachspiels her.

Datei: RiceTableModel.java

```
package kurs.Table;

import javax.swing.table.*;


public class RiceTableModel extends AbstractTableModel
{
 private int size;

  public RiceTableModel(int s)
  {
   size = s;
  }

  public int getRowCount()
  {
     return(size);
  }

  public int getColumnCount()
  {
   return(size);
  }

  public Object getValueAt(int r, int c)
  {
    return(new Long ((long)Math.pow(2,r*size+c)));
  }

}
```

Abschließend die Bemerkung, dass Swing-Tabellen spalten-orientiert sind, so gibt es zum Beispiel für Spalten eigene Modelle.

11.3 Komplexere Inhalte in Tabellen

In diesem Abschnitt konstruieren wir eine Tabelle mit verschiedenen Typen von Einträgen, die zum Teil auch edierbar sind. Wie oben erwähnt, sind Swing-Tabellen

spalten-orientiert. Beim Entwurf einer Tabelle (oder eines Tabellen-Modells) sollte man daher möglichst darauf achten, dass in jeder Spalte Daten des gleichen Typs stehen. Hier enthät die erste Spalte Zahlen (JAVA-Type `Number`), die zweite Wahrheitswerte (JAVA-Type `Boolean`) und die dritte Bilder (JAVA-Type `ImageIcon`).

Die Klasse `JTable` besitzt für einige Objekttypen spezielle Routinen, un diese zu zeichnen. So werden der Typ `Number` in geeignetem Zahlenformat dargestellt, der Typ `String` als Text, der Typ `Boolean` als ankreuzbares Kästchen (Checkbox) und `ImageItem` als Bild. Dies erledigt wieder die Methode `getValueAt`

```
public Object getValueAt(int r, int c)
 {
    Object result = new Object();
    switch(c){
     case 0: result = new Integer(r); break;
     case 1: result = new Boolean(bv[r]);break;
     case 2: result = new ImageIcon(Path+"Bild"+r+".png");  break;
     default:
    }
   return(result);
 }
```

Andere Typen werden mittels ihrer `toString`-Methoden als Zeichenketten dargestellt. Diese Routinen heißen *Renderer* (engl. to render: wiedergeben/darstellen). Sie sind spaltengebunden, das heißt, jede Spalte hat einen einheitlichen Renderer. Man muss dem System allerdings mitteilen, welchen Typ die Daten haben, das heißt aus welcher Klasse sie stammen. Dazu überschreibt man im Tabellen-Modell die Methode `getColumnClass`. In unserem Beispiel liefert diese Methode die Klasse des ersten Eintrages jeder Spalte.

```
public Class getColumnClass(int c)
{
         return getValueAt(0, c).getClass();
}
```

Um eine bestimmte Zelle einer Tabelle edieren zu können, muss man dies erlauben. Dazu überschreibt man die Methode `isCellEditable(r,c)` so, dass sie genau für die edierbaren Zellen (r, c) den Wert `true` liefert

```
public boolean isCellEditable(int r, int c) {
```

```
        return(c == 1);
    }
```

Die Einträge in edierbare Zellen müssen ggf. geändert werden. Dies übernimmt die Methode `setValueAt(Object o, int r, int c)` der Klasse `AbstractTableModel` (oder der von ihr abgeleiteten Klasse, hier `MultipleTableModel`). Die Parameter r und c geben Zeile beziehungsweise Spalte an, während das Objekt o der neu Inhalt der Zelle ist. Diese Methode wird automatisch aufgerufen, wenn eine Zelle der Tabelle ediert wurde, die einen der Typen `Number`, `String`, `Boolean` oder `ImageItem` als Inhalt besitzt. (Für andere Typen muss man ein eigene `CellEditor`-Methode schreiben.)

```
  public void setValueAt(Object o, int r, int c)
   {
     if(c == 1){
       bv[r] = (bv[r] ^ true);      // Parity of bv[r] and 1
     }                              // toggles the value of bv[r]
   }
```

In diesem Beispiel wird der r-te Wert des Booleschen Arrays bv geändert, falls die Spaltennummer 1 ist. Hier nun die kompletten Listings der Klassen `MultiTableFrame` und `MultiTableModel`.

Datei: `MultiTableFrame.java`

```
package kurs.Table;

import java.awt.*;
import javax.swing.*;
import kurs.SimpleFrame.SimpleFrame;

public class MultiTableFrame extends SimpleFrame
{

  public MultiTableFrame()
  {
    MultiTableModel MTM = new MultiTableModel();
    JTable JTab = new JTable(MTM);
    JTab.setRowHeight(50);
    JScrollPane SP = new JScrollPane(JTab);
    this.getContentPane().add(SP);
```

```
  }

  public static void main(String[] args)
  {
    MultiTableFrame MTF = new MultiTableFrame();
    MTF.showIt();
  }
}
```

Datei: `MultiTableModel.java`

```
package kurs.Table;

import javax.swing.table.*;
import javax.swing.ImageIcon;

public class MultiTableModel extends AbstractTableModel
{
  public static final String Path = "D:/JAVA/Kurs/scr/kurs/Table/";
  public boolean[] bv= {true,false,true};

  public MultiTableModel()
  {
  }
  public int getColumnCount()
  {
    return(3);
  }

  public int getRowCount()
  {
    return(3);
  }

   public String getColumnName(int col) {
        return("Spalte "+col);
    }

   public Class getColumnClass(int c) {
            return getValueAt(0, c).getClass();
   }
```

```
  public Object getValueAt(int r, int c)
  {
     Object result = new Object();
     switch(c){
      case 0: result = new Integer(r); break;
      case 1: result = new Boolean(bv[r]);break;
      case 2: result = new ImageIcon(Path+"Bild"+r+".png");  break;
      default:
     }
    return(result);
  }

   public boolean isCellEditable(int r, int c) {
        return(c == 1);
    }

   public void setValueAt(Object o, int r, int c)
   {
     if(c == 1){
       bv[r] = (bv[r] ^ true);     // Parity of bv[r] and 1
     }                             // toggles the value of bv[r]
   }
}
```

Kapitel 12

Darstellung von Bäumen

Swing bietet auch Komponenten zur graphischen Darstellung von baumartigen Strukturen. Solche Strukturen kommen recht häufig vor. So sind die Dateisystemen baumartig angelegt, aber auch die Verwaltungsstruktur einer Behörde oder Taxinomie der Biologie sind so organisiert. Die Swing-Komponente `JTree` kann Wurzelbäume darstellen. Der Baum wird unabhängig davon konstruiert und der darstellenden Komponente `JTree` wird eine Referenz auf die Wurzel übergeben. Der Baum selbst wird dazu aus Knoten zusammengebaut, die das Interface `TreeNode` implementieren. Da diese Implementierung ziemlich aufwendig ist, bietet Swing einen vordefinierten Knotentyp `DefaultMutableTreeNode`. Das „mutable“ (veränderbar) bedeutet, dass man die unter einem Knoten liegenden Subbäume in der Darstellung ein- und ausblenden kann.

Der Zusammenbau des Baumes ist damit sehr einfach. Jedem Knoten übergibt man im Konstruktor ein Objekt, das an diesem Knoten gespeichert wird. In der Anzeige wird dieses Objekt dann dargestellt (es muss natürlich eine Methode zur Darstellung besitzen). Außerdem gibt es die Methode `add`, die einem Knoten ein Kind hinzufügt.

```
DefaultMutableTreeNode(Object obj)
add(DefaultMutableTreeNode node)
```

Von der Klasse `JTree` benötigen wir nur den Konstruktor, der wie erwähnt die Wurzel des darzustellenden Baumes erhält. Die unten angegebene Methode fügt Linien in die Darstellung ein, die die Baumstruktur verdeutlichen

```
JTree(TreeNode root)
putClientProperty("JTree.lineStyle","Angled");
```

Der Code im Beispielprogramm `TreeFrame` konstruiert einen Verzeichnisbaum und zeigt ihn an. Die Wurzel ist das Verzeichnis `startDir`.

```
package kurs.Trees;

import java.awt.*;
import kurs.SimpleFrame.SimpleFrame;
import java.io.*;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.*;
import javax.swing.plaf.metal.MetalLookAndFeel;

public class TreeFrame extends SimpleFrame
{

  private  String startDir = "path/";    // Ist individuell
                                         // zu setzen.
  public TreeFrame()
  {
     File startFile = new File(startDir);
    DefaultMutableTreeNode root = setNode(startFile);
    JTree tree = new JTree(root);
    tree.putClientProperty("JTree.lineStyle","Angled");
    this.getContentPane().add(new JScrollPane(tree));

  }

  public DefaultMutableTreeNode setNode(File curFile)
   {
     DefaultMutableTreeNode curNode =
        new DefaultMutableTreeNode(curFile.getName());
     if(curFile.isDirectory())
     {
      File[] files = curFile.listFiles();
       for (int i = 0; i < files.length; i++) {
         curNode.add(setNode(files[i]));
       }//for i
      }
     return(curNode);
     }

  public static void main(String[] args)
  {
```

```
    TreeFrame TF = new TreeFrame();
    TF.showIt("TreeFrame");
  }
}
```

Die Klasse `JTree` bietet sehr viele Möglichkeiten zur individuellen Gestaltung der Anzeige.

Kapitel 13

Anmerkungen

13.1 Anonyme Listener

Listener werden oft *anonym* definiert. Dieses Vorgehen wird auch in manchen einführenden Büchern verwendet und nicht immer ausreichend erklärt. Listener sind *interfaces*, deren **Methoden** implementiert werden müssen. Die Implementierung einer neuen **Klasse** kann oft vermieden werden.

Angenommen, wir wollen einen Frame `XFrame` mit einem Knopf `XButton` implementieren. Auf den `XButton` soll ein Listener aufpassen und beim Anklicken eine Reaktion auslösen. Wir implementieren keine Listener Klasse sondern in der Klasse `XFrame` die Methode `ActionPerformed` des Listener-Interfaces. Damit ist der `XFrame` quasi auch Listener geworden und er wird dem Knopf in mittels `addActionListener(this)` zugeordnet. Die Code-Struktur sieht dann so aus:

```
class XFrame    extends JFrame implements ActionListener
{

//Konstruktor:

    XFrame
    {
     JButton XButton = new  JButton("XKNOPF");
     XButton.addActionListener(this);
    }//constructor
```

```
 //IMPLEMENTATION von ActionListener:

   public void actionPerformed(ActionEvent evt)
   {
   String command = evt.getActionCommand();
     if (command.equals("XKNOPF"))
      {
        // Code, der beim Anklicken von
        // XButton ausgeführt wird
      }
   }//method

}//class
```

13.2 Implementation von graphischen Darstellungen

Grundsätzlich sollte man immer die Implementation der graphischen Objekte (Kreise, Rechtecke usw.) streng von der der darstellenden Komponenten (Panels, usw.) trennen! Mindestens sollten verschiedene Klassen, bei umfangreicheren Projekten eventuell verschieden Packages verwendet werden. Dieses Vorgehen hat sich **sehr** bewährt.

Ein Panel muss und sollte nicht wissen was ein Polygonzug ist, oder wie man es zeichnet. Es muss nur die Methode zum Zeichnen eines Polygonzugs kennen, und diese geeignet in die `paintComponent`-Methode einbauen. Umgekehrt sollte Polygonzug keine Möglichkeit haben in einem Panel irgendetwas zu verändert. Sein ganzes Wissen über Graphik besteht darin, sich selbst zeichnen zu können.

13.3 Begrenzungslinien

Um viele Swing-Komponenten kann man *Begrenzungslinien* (*border*) ziehen lassen. Dies dazu, Komponenten optisch von einander abzusetzen. Dazu stellt Swing die Klasse `Border` bereit sowie die Hilfsklasse `BorderFactory`, die auf einfache Weise eine Begrenzungslinie erzeugt. Um eine `Border` vom Typ „geätzte Rille“ zu erzeugen und einer Komponente `KOMP` zuzuweisen genügt der folgende Code

```
Border rille = BorderFactory.createEtchedBorder();
KOMP.setBorder(rille);
```

Es gibt weitere Typen von Begrenzungslinien. Will man noch einen Text `text` in die bereits erzeugte Begrenzungslinie `b` eintragen, benutzt man

```
Border tb = BorderFactory.createTitledBorder(Border b,String text);
```

Hier ein einfaches Beispiel mit vier Panels, von denen zwei eine einfache Begrenzungslinie und zwei eine mit Titel besitzen.

Datei: `BorderPanel.java`

```
package kurs.Borders;

import java.awt.*;
import javax.swing.JPanel;
import javax.swing.border.*;
import javax.swing.BorderFactory;

public class BorderPanel extends JPanel
{

  public BorderPanel()
  {
     setBackground(Color.lightGray);
     Border rille      = BorderFactory.createEtchedBorder();
     setBorder(rille);
  }

 public BorderPanel(String text)
  {
     setBackground(Color.lightGray);
     Border rille      = BorderFactory.createEtchedBorder();
     Border titelRille = BorderFactory.createTitledBorder(rille,text);
     setBorder(titelRille);
  }

}
```

Datei: `BorderFrame.java`

```
package kurs.Borders;

import java.awt.*;
import kurs.SimpleFrame.SimpleFrame;

public class BorderFrame extends SimpleFrame
{

  public BorderFrame()
  {
    this.getContentPane().setLayout(new GridLayout(2,2));
    BorderPanel BP1 = new BorderPanel();
    BorderPanel BP2 = new BorderPanel();
    BorderPanel BP3 = new BorderPanel("Panel 3");
    BorderPanel BP4 = new BorderPanel("Panel 4");
    this.getContentPane().add(BP1);
    this.getContentPane().add(BP2);
    this.getContentPane().add(BP3);
    this.getContentPane().add(BP4);
  }

  public static void main(String[] args){
    BorderFrame BF = new BorderFrame();
    BF.showIt("Borders");

  }
}
```

13.4 Das Grid-Bag-Layout

Beim Raster-Layout (grid-Layout) aus Kapitel 3 werden alle eingebetteten Komponenten gleichgroß gezeichnet, jeweils eine Komponente in eine Zelle des Rasters. Das hier besprochene *Raster-Beutel-Layout* (`GridBagLayout`) erlaubt es, dass eine Komponente mehrere Zellen des Rasters einnimmt. Diese Zellen bilden einen Beutel (bag) und müssen natürlich ein zusammenhängendes Rechteck im Raster bilden. Dazu gibt man jeder einzubettenden Komponente Information darüber mit, wo und wie groß sie im Raster erscheinen soll. Diese Information wird in der Klasse `GridBagConstraints` (Constraint = Bedingung) abgelegt.

Beginnen wir mit einem Beispiel. Wir wollen sechs Panels in die Content-Pane eines Frames so einkleben, dass sich das in Abbildung 13.1 dargestellte Bild ergibt. Als

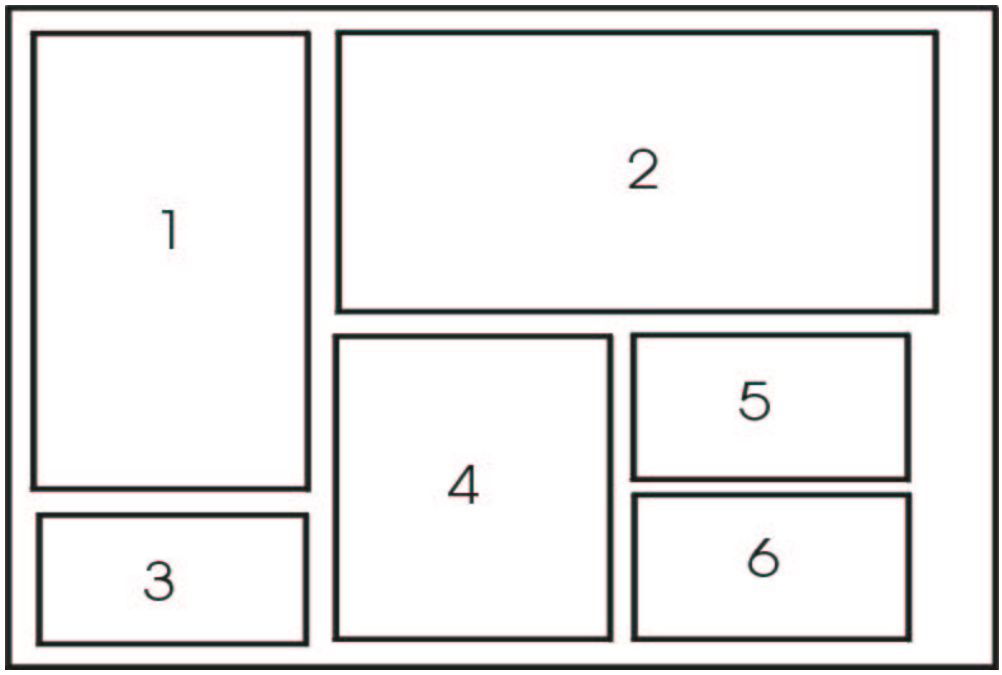

Abbildung 13.1: A

ersten Schritt legen wir dazu die Rastergröße fest. Hier bietet sich ein 4×3 Raster an. Man sollte immer ein Raster mit möglichst wenig Zeilen und Spalten wählen, auch wenn man dann die eine oder andere Komponente etwas vergrößert oder verkleinert. Abbildung 13.2 zeigt das Raster. Damit ist klar, dass das Panel mit der Nummer 1 (P1) eine Spalte breit, drei Zeilen hoch ist und seine linke obere Ecke in der Zelle $(0, 0)$ des Rasters liegt. Panel P2 ist zwei Spalte breit, zwei Zeilen hoch ist und seine linke obere Ecke ist die Zelle $(1, 0)$, usw. Diese Parameter müssen wir für jedes der sechs Panels in `GridBagConstraints` festlegen und die Constraints den Panels zu ordnen. Ein `GridBagConstraints`-Objekt enthält die Felder `gridwidth`, `gridheight`, `gridx` und `gridy` die der Breite beziehungsweise Höhe einer Komponente (in Rasterzellen) beziehungsweise der Position der linke obere Ecke (in Rasterzellen) entsprechen. Zum Setzen dieser Werte stellt `GridBagConstraints` keine `set`-Methoden bereit, die Felder sind `public` und können direkt gesetzt werden. Das folgende Code-Fragment legt ein Grid-Bag-Layout an, erzeugt Constraints für unser Panel P1 und ordnet das Panel mit den zugehörigen Constraints dem Layout zu.

!

```
GridBagLayout GBL = new GridBagLayout();

GridBagConstraints constraints = new  GridBagConstraints();
```

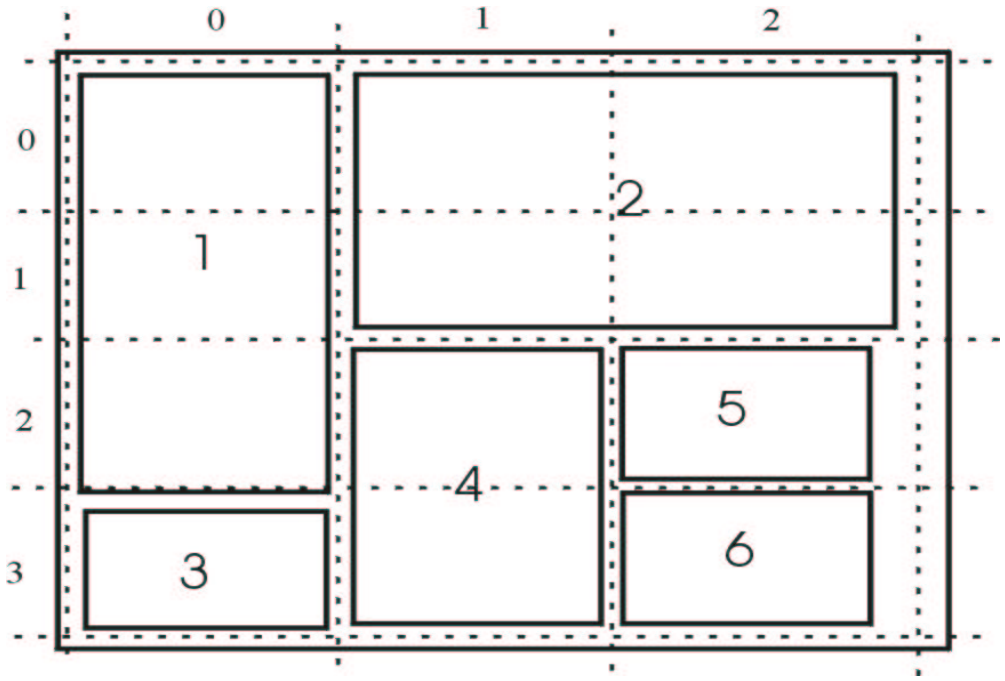

Abbildung 13.2: B

```
constraints.gridwidth =  1;  //Breite
constraints.gridheight = 3;  //Höhe
constraints.gridx = 0;       //Spalte der linken oberen Ecke v. P1
constraints.gridy = 0;       //Zeile  der linken oberen Ecke v. P1

GBL.setConstraints(P1,constraints);
```

`GridBagConstraints`-Objekt enthält weitere Felder, die man setzen sollte, deren Bedeutung wir aber nur kurz beschreiben. Im Feld `fill` wird angegeben, ob Eine Komponente ihren Beutel horizontal oder vertikal oder beides ausfüllen soll. Wir wählen hier die letzte Möglichkeit

```
constraints.fill = GridBagConstraints.BOTH;
```

Weiterhin kann man jeder Komponente ein horizontales Gewicht `weightx` und ein vertikales `weighty` zuordnen, deren Bedeutung wir gleich erläutern. Wir setzen alle Gewichte auf 1.0.

```
constraints.weightx = wx;
constraints.weighty = wy;
```

Das unten angegebene Programm `GridBagFrame` setzt die Vorgaben aus der Abbildung 13.2 um. Damit wir nicht alle Constraint-Definitionen für alle sechs Panels wiederholen müssen, haben wir die Methode `easyConstraints` geschrieben, der wir das Layout, die einzubettende Swing-Komponente und die Constraint-Werte übergeben.

```
package kurs.GridBag;

import java.awt.*;
import javax.swing.JComponent;
import kurs.SimpleFrame.SimpleFrame;
import kurs.Borders.BorderPanel;


public class GridBagFrame extends SimpleFrame
{
  private static final double xw = 1.0;
  private static final double yw = 1.0;

  private GridBagLayout GBL = new GridBagLayout();
  public GridBagFrame()
  {
    this.getContentPane().setLayout(GBL);
    BorderPanel BP1 = new BorderPanel("Panel 1");
    BorderPanel BP2 = new BorderPanel("Panel 2");
    BorderPanel BP3 = new BorderPanel("Panel 3");
    BorderPanel BP4 = new BorderPanel("Panel 4");
    BorderPanel BP5 = new BorderPanel("Panel 5");
    BorderPanel BP6 = new BorderPanel("Panel 6");

    this.getContentPane().add(BP1);
    this.getContentPane().add(BP2);
    this.getContentPane().add(BP3);
    this.getContentPane().add(BP4);
    this.getContentPane().add(BP5);
    this.getContentPane().add(BP6);

    easyConstraints(GBL,BP1,1,3,0,0,xw,yw);
    easyConstraints(GBL,BP2,2,2,1,0,xw,yw);
    easyConstraints(GBL,BP3,1,1,0,3,xw,yw);
    easyConstraints(GBL,BP4,1,2,1,2,xw,yw);
    easyConstraints(GBL,BP5,1,1,2,2,xw,yw);
    easyConstraints(GBL,BP6,1,1,2,3,xw,yw);
```

```
  }

  public void easyConstraints(GridBagLayout GLB,JComponent Comp,
              int w, int h, int x, int y,double wx, double wy){
    GridBagConstraints constraints = new  GridBagConstraints();
    constraints.fill = GridBagConstraints.BOTH;
    constraints.gridwidth =  w;
    constraints.gridheight = h;
    constraints.gridx = x;
    constraints.gridy = y;
    constraints.weightx = wx;
    constraints.weighty = wy;
    GBL.setConstraints(Comp,constraints);
  }

  public static void main(String[] args){
    GridBagFrame GBF = new GridBagFrame();
    GBF.showIt("GridBag");

  }
}
```

In Abbildung 13.3 sehen wir das Resultat. Dabei fällt auf, dass ein 3×3-Raster und nicht wie erwartet ein 4×3-Raster vorliegt. Die Komponente P2 nimmt nicht die Hälfte der Gesamthöhe ein, sondern nur ein Drittel.

Aufgabe 12: Finde heraus, warum das so ist.

Die Gewichte einer Komponente legen fest, wieviel Anteil am Gesamtplatz sie erhält. Man setze zum Beispiel die horizontalen Gewichte `weightx` der Panels P1 und P3 auf 0.5. Damit hat die erste Spalte dieses horizontale Gewicht während die zweite und dritte Spalte je das Gewicht 1.0 haben. Startet man das Programm, so ist die erste Spalte nur halb so breit wie die anderen beiden. Das gesamte horizontale Gewicht von 2.5 verteilt sich auf die drei Spalten im Verhältnis $1 : 2 : 2$.

Um die Breite der zweiten Spalte zu verändern, genügt es das `weightx` von P4 zu verändern, denn dieses Panel alleine definiert diese Spalte. Man sollte mir den gewichten und den anderen Parametern der grid-Bag-Constraints ein bisschen herumspielen, um ein Gefühl für ihre Bedeutung und Wirkung zu bekommen.

Aufgabe 13: Beseitige das weiter oben beschriebene Problem mit Höhe der Kom-

Abbildung 13.3: Ein GrigBagFrame.

ponente P2. Sie soll die Hälfte der Gesamthöhe einnehmen.

Index

A

B

C

D

E

F

G

H

I

J

W

Z