Programmieren II

SS 04

Sven Eric Panitz

TFH Berlin

Version 12. Juli 2004

Dieses Skript entstand begleitend zur Vorlesung des SS 03 vollkommen neu. Es stellte somit eine Mitschrift der Vorlesung dar. Diese Version basiert auf der ersten Version erfährt jedoch im Laufe des Semesters einige Umstrukturierungen. Insbesondere das Kapitel über Klassenpfade und Jar-Dateien ist ins Skript Programmieren I[Pan03] gewandert.

Naturgemäß ist es in Aufbau und Qualität nicht mit einem Buch vergleichbar. Flüchtigkeits- und Tippfehler werden sich im Eifer des Gefechtes nicht vermeiden lassen und wahrscheinlich in nicht geringem Maße auftreten. Ich bin natürlich stets dankbar, wenn ich auf solche aufmerksam gemacht werde und diese korrigieren kann.

Die Vorlesung setzt die Kerninhalte der Vorgängervorlesung voraus[Pan03].

Der Quelltext dieses Skripts ist eine XML-Datei, die durch eine XQuery in eine LaTeX-Datei transformiert und für die schließlich eine pdf-Datei und eine postscript-Datei erzeugt wird.

Alle Beispielprogramme sind mit einem Skript aus dem Quelltext extrahiert und übersetzt worden.

Inhaltsverzeichnis

Einführung

Ziel der Vorlesung

Mit den Grundtechniken der Programmierung am Beispiel von Java wurde in der Vorgängervorlesung vertraut gemacht. Jetzt ist es an der Zeit, das Gelernte anzuwenden und gängige Techniken und Muster der Programmierung zu lernen. Im Prinzip wird kein neues Javakonstrukt kennengelernt werden[1], sondern mit vorhandenen Bibliotheken vetraut gemacht. Am Ende der Vorlesung sollte im besten Fall erreicht sein, daß jeder Teilnehmer sich in der Lage sieht, selbstständig fremde Bibliotheken zu verwenden und eigene Bibliotheken zu entwerfen.

Eine große Bibliothek, mit der insbesondere in dieser Vorlesung vertraut gemacht werden soll, ist Javas Standardbibliothek für graphische Benutzeroberflächen: Swing.

[1] Ein paar wenige Konstrukte werden neu sein, wie z.B. innere Klassen.

Kapitel 1

Frühe und späte Bindungen

1.1 Wiederholung: Das Prinzip der späten Bindung

Methoden aus Klassen können in einer Unterklasse überschrieben werden. Damit gibt es mehrere Methodendefinitionen mit gleichen Namen und gleicher Signatur. Wenn das Programm kompiliert wird, wird in den erzeugten Code nicht hineingeschrieben, welche dieser zwei gleich heißenden Methodendefinitionen ausgeführt werden soll. Erst zur Ausführungszeit wird das in Frage kommende Objekt gefragt, ob es eine eigene Definition für diese Methode hat, und dann diese genommen. Der Aufrufer muß dabei noch nicht einmal wissen, um was für eine spezielle Unterklasse es sich in diesem Fall handelt.

Wem obiger Absatz zu abstrakt klingt, betrachte folgendes simple Beispiel.

Beispiel:
Wir sehen zunächst eine Schnittstelle vor. Dieses wäre zur Verdeutlichung der späten Bindung nicht unbedingt notwendig. Objekte, die diese Schnittstelle implementieren, sollen eine Methode `getDescription` enthalten.

Descriptable.java

```
1  package name.panitz.lateBinding;
2
3  public interface Descriptable{
4    String getDescription();
5  }
```

Jetzt können wir eine Klasse schreiben, die diese Schnittstelle implementiert.

Upper.java

```
1  package name.panitz.lateBinding;
2
3  public class Upper implements Descriptable{
4    public String getDescription(){return "Upper";}
5  }
```

Von dieser Klasse schreiben wir eine Unterklasse, die eine eigene neue Implementierung der Methode `getDescription` enthält.

Lower.java

```
1  package name.panitz.lateBinding;
2
3  public class Lower extends Upper{
4    public String getDescription(){return "Lower";}
```

Jetzt erzeugen wir einmal von beiden Klassen Objekte und speichern sie zunächst in Variablen des Typs `Upper`. Obwohl wir den Variablen selbst nicht mehr ansehen können, ob in ihnen Objekte des Typs `Upper` oder `Lower` gespeichert sind, wird jeweils die eigene Methode des Objektes ausgeführt. Das gleiche läßt sich beobachten, wenn die Objekte in einer Variablen des Schnittstellentyps `Descriptable` gespeichert sind.

Lower.java

```
5    public static void main(String[] _){
6      Upper up = new Upper();
7      Upper lo = new Lower();
8      System.out.println(up.getDescription());
9      System.out.println(lo.getDescription());
10     Descriptable des = up;
11     System.out.println(des.getDescription());
12     des = lo;
13     System.out.println(des.getDescription());
14   }
15 }
```

Wir können folgende Ausgabe des Testprogramms beobachten:

```
sep@linux:~/fh/prog2/examples> java -classpath classes/ name.panitz.lateBinding.Lower
Upper
Lower
Upper
Lower
sep@linux:~/fh/prog2/examples>
```

1.2 Keine späte Bindung für Felder

Wir haben in der Vorlesung *Programmieren I* nicht unerwähnt gelassen, daß das Prinzip der späten Bindung nur für Methoden und nicht für Felder gilt. Damit sind Felder in Java in gewisser Weise nur Eigenschaften zweiter Klasse; ihnen steht ein fundamentales Konzept für Methoden nicht zur Verfügung. Diese Eigenschaft kann eine Stolperfalle für Programmierer sein und ist oft an Java kritisiert worden. Um uns die entsprechenden Effekte zu illustrieren schreiben wir eine kleine Klasse, die uns eine Ente modellieren soll:

Ente.java

```
1  package name.panitz.lateBinding;
2
3  class Ente {
4    int beine = 2;
5    int getBeine(){return beine;}
```

```
6   int getWirklicheBeine(){return beine;}
7 }
```

In der Klasse ist in einem Feld gespeichert wieviel Beine eine Ente hat. Zwei Methoden sind reine `get`-Methoden, indem sie nur den Wert des Feldes `beine` auslesen.

Als nächstes schreiben wir eine Unterklasse der Klasse `Ente`. Diese Klasse soll besondere Enten beschreiben, deren Schicksal es ist, nur noch ein Bein zu haben. Wir überschreiben das Feld `beine` mit dem neuen Wert `1`. Ebenso überschreiben wir die Methode `getWirklicheBeine`, allerdings mit der identischen Implementierung aus der Oberklasse.

LahmeEnte.java

```
1 package name.panitz.lateBinding;
2
3 class LahmeEnte extends Ente {
4   int beine = 1;
5   int getWirklicheBeine(){return beine;}
6 }
```

Jetzt können wir versuchen alle drei Eigenschaften auszuprobieren. Hierzu legen wir von beiden Klassen jeweils ein Objekt an und geben für die entsprechenden Objekte die drei Eigenschatfen auf dem Bildschirm aus. Dabei erzeugen wir ein Objekt vom Typ `LahmeEnte` und geben die Eigenschaften einmal aus, indem wir das Objekt einmal von einem Feld des Typs `Ente` und einmal von einem Feld des Typs `LahmeEnte` referenzieren.

EntenTest.java

```
1  package name.panitz.lateBinding;
2
3  class EntenTest {
4    public static void main(String [] args){
5      Ente ente1 = new Ente();
6      LahmeEnte ente2 = new LahmeEnte();
7      Ente ente3 = ente2;
8
9      System.out.println("Ente ente1 = new Ente()");
10     System.out.println("beine: "+ente1.beine);
11     System.out.println("getBeine(): "+ente1.getBeine());
12     System.out.println(
13      "getWirklicheBeine(): "+ente1.getWirklicheBeine());
14
15     System.out.println(
16      "\nLahmeEnte ente2 = new LahmeEnte()");
17     System.out.println("beine: "+ente2.beine);
18     System.out.println("getBeine(): "+ente2.getBeine());
19     System.out.println(
20      "getWirklicheBeine(): "+ente2.getWirklicheBeine());
21
22     System.out.println("\nEnte ente3 = new LahmeEnte()");
23     System.out.println("beine: "+ente3.beine);
24     System.out.println("getBeine(): "+ente3.getBeine());
```

```
25      System.out.println(
26        "getWirklicheBeine(): "+ente3.getWirklicheBeine());
27    }
28  }
```

Starten wir das Programm, so stellen wir ein paar unerwartete Effekte fest:

```
sep@swe10:~/fh/prog2/beispiele> java name.panitz.lateBinding.EntenTest
Ente ente1 = new Ente()
beine: 2
getBeine(): 2
getWirklicheBeine(): 2

LahmeEnte ente2 = new LahmeEnte()
beine: 1
getBeine(): 2
getWirklicheBeine(): 1

Ente ente3 = ente2
beine: 2
getBeine(): 2
getWirklicheBeine(): 1
sep@swe10:~/fh/prog2/beispiele>
```

Für das Objekt `ente1` vom Typ `Ente` haben wir noch die erwarteten Ergebnisse. 2 Beine, egal über welche Methode wir darauf zugreifen. Für das Objekt `ente2` haben wir zwei zunächst überraschende Ergebnisse. Während `beine` und `getWirklicheBeine()` in der Betrachtung als Feld vom Typ `LahmeEnte` jeweils die Zahl `1` als Ergebnis liefern, so liefert die Methode `getBeine()` den Wert `2`. Dieses liegt daran, daß wir diese Methode nicht überschrieben haben. Sie wird aus der Klasse `Ente` geerbt. Diese greift auf das Feld `beine` zu. Da für Felder das Prinzip der späten Bindung nicht gilt, greift sie auf das Feld `beine` der Klasse `Ente` und nicht der Klasse `LahmeEnte` zu. Daher erhalten wir den Wert aus der Klasse `Ente`.

Das zweite überraschende Ergebnis ist, wenn wir das Objekt der Klasse `LahmeEnte` von einem Feld der Klasse `Ente` betrachten. Dann bekommen wir für das Feld `beine` plötzlich für dasselbe Objekt einen anderen Wert (die `2`) ausgegeben. Für dasselbe Objekt plötzlich einen anderen Wert zu bekommen, widerspricht unserer Intuition von Objekten. Auch hier ist die fehlende späte Bindung für Felder dran ursächlich. Anders als bei Methodenaufrufen, in denen wir sagen, das Objekt soll selbst seine eigene Methode mit dem entsprechenden Namen ausführen, sagt der Aufrufer aus welcher Klasse er das Feld haben möchte. Bei Feldern ist nicht der Typ, mit dem das Objekt einst erzeugt wurde, sondern der Typ, unter dem es betrachtet, mit dem es referenziert wird, relevant.

Dieses läßt sich besonders durch eine Typzusicherung illustrieren:

EntenTest2.java

```
1  package name.panitz.lateBinding;
2  class EntenTest2 {
3    public static void main(String [] args){
4      Ente ente = new LahmeEnte();
5      System.out.println(            ente .beine);
6      System.out.println(((LahmeEnte)ente).beine);
7    }
8  }
```

Die Ausgabe dieses Tests ist:

```
sep@swe10:~/fh/prog2/beispiele> java name.panitz.lateBinding.EntenTest2
2
1
sep@swe10:~/fh/prog2/beispiele>
```

Wie man sieht sind überschriebene Felder mit Vorsicht zu genießen. Deshalb ist es zu empfehlen Felder nicht zu überschreiben. Besonders sicher fährt man, wenn man Felder privat macht und nur über get- und set-Methoden auf Felder zugreift.

1.3 Keine Späte Bindung für überladene Methoden

Es gibt noch eine zweite Art und Weise verschiedene Funktionsdefinitionen für eine Methode zu haben. Statt in verschiedenen Klassen eine Funktion zu schreiben, kann man auch in einer Klasse eine Methode für verschiedene Argumenttypen überladen. Um dieses zu Demonstrieren, schreiben wir eine kleine Klassenhierarchie aus drei Klassen:

Up.java
```
1 package name.panitz.overload;
2 public class Up{}
```

Für diese Oberklasse schreiben wir zwei Unterklassen.

Down1.java
```
1 package name.panitz.overload;
2 public class Down1 extends Up{}
```

Down2.java
```
1 package name.panitz.overload;
2 public class Down2 extends Up{}
```

Jetzt überladen wir eine Methode `getDescription` für Argumente dieser drei Klassen

GetDescription.java
```
1 package name.panitz.overload;
2 public class  GetDescription {
3   public String getDescription(Up _){return "Up";}
4   public String getDescription(Down1 _){return "Down1";}
5   public String getDescription(Down2 _){return "Down2";}
```

Zum Testen wenden wir die Methode auf Objekte dieser Klassen an. Dabei zunächst über Variablen, die jeweils genau den Typ des Objekts und nicht einer Oberklasse haben. Dann speichern wir die Objekte in der Variablen des gemeinsamen Obertyps `Up`. Und schließlich noch, indem wir zuvor wieder eine Typzusicherung auf den eigentlichen Objekttyp vornehmen.

GetDescription.java

```
6   public static void main(String [] _){
7     GetDescription g = new GetDescription();
8     Up up = new Up();
9     Down1 down1 = new Down1();
10    Down2 down2 = new Down2();
11    System.out.println(g.getDescription(up));
12    System.out.println(g.getDescription(down1));
13    System.out.println(g.getDescription(down2));
14    up = down1;
15    System.out.println(g.getDescription(up));
16    up = down2;
17    System.out.println(g.getDescription(up));
18    System.out.println(g.getDescription((Down2)up));
19  }
20 }
```

An der Ausgabe kann man sich davon überzeugen, daß das Objekt nicht gefragt wird, von welchen Typ es eigentlich ist. Es wird jeweils die Methodendefinition genommen, die auf den Typ der Variablen passt.

```
sep@linux:~/fh/prog2/examples> java -classpath classes/ name.panitz.overload.GetDescription
Up
Down1
Down2
Up
Up
Down2
sep@linux:~/fh/prog2/examples>
```

Späte Bindung findet also für überladene Methoden nicht statt. Der Übersetzer generiert fest den Methodenaufruf, für genau den Typ, den der statische Typchecker für das Argument festgestellt hat.

Kapitel 2

Graphische Benutzeroberflächen und Graphiken

Die meisten Programme auf heutigen Rechnersystemen haben eine graphische Benutzeroberfläche (GUI)[1].

Java stellt Klassen zur Verfügung, mit denen graphische Objekte erzeugt und in ihrem Verhalten instrumentalisiert werden können. Leider gibt es historisch gewachsen zwei Pakete in Java mit Klassen zur GUI-Programmierung. Die beiden Pakete überschneiden sich in der Funktionalität.

- `java.awt`: Dieses ist das ältere Paket zur GUI-Programmierung. Es enthält Klassen für viele graphische Objekte (z.B. eine Klasse `Button`) und Unterpakete, zur Programmierung der Funktionalität der einzelnen Komponenten.
- `javax.swing`: Dieses neuere Paket ist noch universeller und platformunabhängiger als das `java.awt`-Paket. Auch hier finden sich Klassen für unterschiedliche GUI-Komponenten. Sie entsprechen den Klassen aus dem Paket `java.awt`. Die Klassen haben oft den gleichen Klassennamen wie in `java.awt` jedoch mit einem `J` vorangestellt. So gib es z.B. eine Klasse `JButton`.

Ein fundamentaler Unterschied dieser beiden Pakete besteht in ihrer Behandlung von Steuerfäden. Während das `awt`-Paket sicher in Bezug auf Steuerfäden ist, können im `swing`-Paket Steuerfäden nicht ohne zusätzlihe Absicherung benutzt werden.

Zu allen Unglück sind die beiden Pakete nicht ganz unabhängig. Man ist zwar angehalten, sofern man sich für eine Implementierung seines Guis mit den Paket `javax.swing` entschieden hat, nur die graphischen Komponenten aus diesem Paket zu benutzen; die Klassen leiten aber von Klassen des Pakets `java.awt` ab. Hinzu kommt, daß die Ereignissklassen, die die Funktionalität graphischer Objekte bestimmen, nur in `java.awt` existieren und nicht noch einmal für `javax.swing` extra umgesetzt wurden.

Sind Sie verwirrt. Sie werden es hoffentlich nicht mehr sein, nachdem Sie die Beispiele dieses Kapitels durchgespielt haben.

[1] GUI ist die Abkürzung für *graphical user interface*. Entsprechend wäre GRABO eine Abkürzung für das deutsche *graphische Benutzeroberfläche*. Im Skript von Herrn Grude wird dieser sehr schöne Ausdruck für GUI verwendet.

2.1 Graphische Benutzeroberflächen

Zur Programmierung eines GUIs bedarf es graphischer Objekte, einer Möglichkeit, graphische Objekte zu gruppieren, und schließlich eine Möglichkeit zu bestimmen, wie die Objekte auf bestimmte Ereignisse reagieren (z.B. was passiert, wenn ein Knopf gedrückt wird). Java kennt für diese Aufgaben jeweils eigene Klassen:

- Komponenten
- Ereignisbehandlung
- Layoutstuerung

In den folgenden Abschnitten, werden wir an ausgewählten Beispielen Klassen für diese drei wichtigen Schritte der GUI-Programmierung kennenlernen.

2.1.1 Graphische Komponenten

Javas `swing`-Paket kennt drei Arten von Komponenten.

- Top-Level Komponenten
- Zwischenkomponenten
- Atomare Komponenten

Leider spiegelt sich diese Unterscheidung nicht in der Ableitungshierarchie wider. Alle Komponenten leiten schließlich von der Klasse `java.awt.Component` ab. Es gibt keine Schnittmengen, die Beschreiben, daß bestimmte Komponenten atomar oder top-level sind.

Komponenten können Unterkomponenten enthalten; ein Fenster kann z.B. verschiedene Knöpfe und Textflächen als Unterkomponenten enthalten.

Top-Level Komponenten

Eine top-level Komponenten ist ein GUI-Objekt, das weitere graphische Objekte enthalten kann, selbst aber kein graphisches Objekt hat, in dem es enthalten ist. Somit sind top-level Komponenten in der Regel Fenster, die weitere Komponenten als Fensterinhalt haben. Swing top-level Komponenten sind Fenster und Dialogfenster. Hierfür stellt Swing entsprechende Klassen zur Verfügung: `JFrame, JDialog`

Eine weitere top-level Komponente steht für *Applets* zur Verfügung: `JApplet`.

Graphische Komponenten haben Konstruktoren, mit denen sie erzeugt werden. Für die Klasse `JFrame` existiert ein parameterloser Konstruktor.

> **Beispiel:**
> Das minimalste GUI-Programm ist wahrscheinlich folgendes Programm, das ein Fensterobjekt erzeugt und dieses sichtbar macht.

Window1.java

```
1 package name.panitz.simpleGui;
2 import javax.swing.*;
3
4 public class Window1{
5   public static void main(String [] _){
6     new JFrame("erstes Fenster").setVisible(true);
7   }
8 }
```

Dieses Programm erzeugt ein leeres Fenster und gibt das auf dem Bildschirm aus. Der Fenstertitel kann als Argument dem Konstruktor der Klasse `JFrame` als `String` übergeben werden. Die optische Ausprägung des Fensters kann vom Betriebssystem abhängen. In Abbildung 2.1 ist dieses Fenster beispielhaft in einer Suse Linux Umgebung mit KDE zu bewundern.

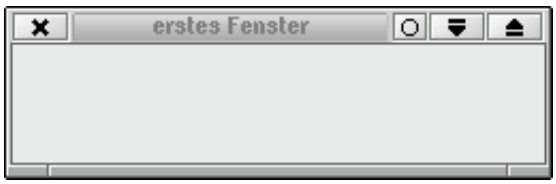


Abbildung 2.1: Ein erstes Fenster.

Zwischenkomponenten

Graphische Zwischenkomponenten haben primär die Aufgabe andere Komponenten als Unterkomponenten zu haben und diese in einer bestimmten Weise anzuordnen. Zwischenkomponenten haben oft keine eigene visuelle Ausprägung. Sie sind dann unsichtbare Komponenten, die als Behälter weiterer Komponenten dienen.

Die gebräuchlichste Zwischenkomponenten ist von der Klasse `JPanel`. Weitere Zwischenkomponenten sind`JScrollPane` und `JTabbedPane`. Diese haben auch eine eigene visuelle Ausprägung.

Die Objekte der Klasse `JFrame` enthalten bereits eine Zwischenkomponente. Diese kann mit der Methode `getContentPane` erhalten werden, um dieser Zwischenkomponenten schließlich weitere Unterkomponenten, den Inhalt des Fensters, hinzufügen zu können.

Atomare Komponenten

Die atomaren Komponenten sind schließlich Komponenten, die ein konkretes graphisches Objekt darstellen, das keine weiteren Unterkomponenten enthalten kann. Solche Komponenten sind z.B.
`JButton, JTextField, JTable` und `JComBox`.

Diese Komponenten lassen sich über ihre Konstruktoren instanziieren, um sie dann einer Zwischenkomponenten über deren Methode `add` als Unterkomponente hinzuzufügen. Sind alle gewünschten graphischen Objekte einer Zwischenkomponente hinzugefügt worden, so

kann auf der zugehörigen top-level Komponenten die Methode `pack` aufgerufen wurden. Diese berechnet die notwendige Größe und das Layout des Fensters, welches schließlich sichtbar gemacht wird.

Beispiel:
Das folgende Programm erzeugt ein Fenster mit einer Textfläche.

JT.java

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
3
4  class JT {
5    public static void main(String [] _){
6      JFrame frame = new JFrame();
7      JTextArea textArea = new JTextArea();
8      textArea.setText("hallo da draußen");
9      frame.getContentPane().add(textArea);
10     frame.pack();
11     frame.setVisible(true);
12   }
13 }
```

Das Programm erzeugt das folgende Fenster:

Abbildung 2.2: Fenster mit Textfläche

In der Regel wird man nicht die Fenster des Programms nacheinander in der Hauptmethode definieren, sondern die gebrauchten Fenster werden als eigene Objekte definiert. Hierzu leitet man eine spezifische Fensterklasse von der Klasse `JFrame` ab und fügt bereits im Konstruktor die entsprechenden Unterkomponenten hinzu.

Beispiel:
Die folgende Klasse definiert ein Fenster, das einen Knopf und eine Textfläche enthält. In der Hauptmethode wird das Objekt instanziiert:

JTB.java

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
3
4  class JTB extends JFrame {
5    JPanel pane = new JPanel();
6    JTextArea textArea = new JTextArea();
7    JButton button = new JButton("ein Knopf");
8
9    public JTB(){
10     textArea.setText("hallo da draußen");
```

```
11      pane.add(textArea);
12      pane.add(button);
13      getContentPane().add(pane);
14      pack();
15      setVisible(true);
16    }
17
18    public static void main(String [] _){
19      new JTB();
20    }
21  }
```

Das Programm erzeugt folgendes Fenster:

Abbildung 2.3: Fenster mit Textfläche und Knopf

2.1.2 Ereignisbehandlung

Wir haben es bisher schon geschafft ein Fenster zu definieren, in dem mehrere graphische Komponenten dargestellt werden. Für ein funktionierendes GUI ist es nun notwendig, für einzelne Komponenten eine Funktionalität zu definieren. Hierzu kennt Swing das Konzept der Ereignisbehandlung. Graphischen Komponenten kann ein Ereignisbehandlungsobjekt angehänt werden. Objekte zur Ereignisbehandlung müssen dabei bestimmte Schnittstellen implementieren, die sogenannten `Listener`.

Ereignisarten gibt es eine ganze Reihe auf den unterschiedlichsten graphischen Komponenten: ein Knöpf kann gedrückt werden, ein Fenster minimiert, mit der Maus auf eine Komponente gezeigt werden, ein Menupunkt gewählt werden etc. Für die unterschiedlichen Ereignisarten gibt es unterschiedliche Schnittstellen, die beschreiben, wie auf solche Ereignisse reagiert werden soll. Die Schnittstellen für Ereignisse befinden sich im Paket: `java.awt.event.*`.

Beispiel:
Als erstes Beispiel für eine Ereignisbehandlung betrachten wir die vielleicht elementarste Art eines Ereignisses: ein Knopf wird gedrückt. Hierzu gibt es die elementare Ereignisbehandlungsschnittstelle: `java.awt.event.ActionListener`. Diese Schnittstelle enthält nur eine Methode: `actionPerformed`

Wir implementieren diese Schnittstelle. Unsere Klasse zur Ereignisbehandlung bekommt dabei ein Textfeld übergeben. Intern enthalte die Klasse einen Zähler. Wann immer das Ereignis eintritt, soll der Zähler um eins erhöht werden und das neue Ergebnis des Zählers im Textfeld angezeigt werden.

CounterListener.java

```
1  package name.panitz.simpleGui;
2  import java.awt.event.*;
```

```
3  import javax.swing.*;
4
5  public class CounterListener implements ActionListener{
6    private final JTextArea textArea;
7    private int counter = 0;
8    public CounterListener(JTextArea ta){textArea=ta;}
9
10   public void actionPerformed(ActionEvent _){
11     counter = counter+1;
12     textArea.setText(counter+"");
13   }
14 }
```

Jetzt können wir eine Unterklasse unserer letzten kleinen Gui-Klasse `JTB` schreiben, in der diese Ereignisbehandlung dem Knopf hinzugefügt wird. Hierzu gibt es für Knöpfe die Methode `addActionListener`.

Counter.java

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
3
4  class Counter extends JTB {
5    public Counter(){
6      super();
7      button.addActionListener(new CounterListener(textArea));
8    }
9
10   public static void main(String [] _){new Counter();}
11 }
```

Und tatsächlich, wenn wir dieses Programm starten, so bewirkt, daß ein Drücken des Knopfes mit der Maus bewirkt, daß im Textfeld angezeigt wird, wie oft bisher der Knopf gedrückt wurde.

Aufgabe 1 Die Methode `actionPerformed` der Schnittstelle `ActionListener` bekommt ein Objekt der Klasse `ActionEvent` übergeben. In dieser gibt es eine Methode `getWhen`, die den Zeitpunkt, zu dem das Ereignis aufgetreten ist als eine Zahl kodiert zurückgibt. Mit dieser Zahl können Sie die Klasse `java.util.Date` instanziieren, um ein Objekt zu bekommen, das Datum und Uhrzeit enthält.

Schreiben Sie jetzt eine Unterklasse von `JTB`, so daß beim Drücken des Knopfes, jetzt die aktuelle Uhrzeit im Textfeld erscheint.

Aufgabe 2 Schreiben Sie eine kleine Guianwendung mit einer Textfläche und drei Knöpfen, die einen erweiterten Zähler darstellt:

- einen Knopf zum Erhöhen eines Zählers.
- einen Knopf zum Verringern des Zählers um 1.
- einen Knopf zum Zurücksetzen des Zählers auf 0.

2.1.3 Setzen des Layouts

Bisher haben wir Unterkomponenten weitere Komponenten mit der Methode `add` hinzugefügt, ohne uns Gedanken über die Plazierung der Komponenten zu machen. Wir haben einfach auf das Standardverhalten zur Plazierung von Komponenten vertraut. Ob die Komponenten schließlich nebeneinander, übereinander oder irgendwie anders Gruppiert im Fenster erschienen, haben wir nicht spezifiziert.

Um das Layout von graphischen Komponenten zu steuern, steht das Konzept der sogenannten *Layout-Manager* zur Verfügung. Ein Layout-Manager ist ein Objekt, das einer Komponente hinzugefügt wird. Der Layout-Manager steuert dann, in welcher Weise die Unterkomponenten gruppiert werden.

`LayoutManager` ist eine Schnittstelle. Es gibt mehrere Implementierungen dieser Schnittstelle. Wir werden in den nächsten Abschnitten drei davon kennenlernen. Es steht einem natürlich frei, eigene Layout-Manager durch Implementierung dieser Schnittstelle zu schreiben. Es wird aber davon abgeraten, weil dieses notorisch schwierig ist und die in Java bereits vorhandenen Layout-Manager bereits sehr mächtig und ausdrucksstark sind.

Zum Hinzufügen eines Layout-Manager gibt es die Methode `setLayout`.

Flow Layout

Der vielleicht einfachste Layout-Manager nennt sich `FlowLayout`. Hier werden die Unterkomponenten einfach der Reihe nach in einer Zeile angeordnet. Erst wenn das Fenster zu schmal hierzu ist, werden weitere Komponenten in eine neue Zeile gruppiert.

Beispiel:
Die folgende Klasse definiert ein Fenster, dessen Layout über ein Objekt der Klasse `FlowLayout` gesteuert wird. Dem Fenster werden fünf Knöpfe hinzugefügt:

FlowLayoutTest.java

```
1  package name.panitz.gui.layoutTest;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  class FlowLayoutTest extends JFrame {
7
8    public FlowLayoutTest(){
9      Container pane = getContentPane();
10     pane.setLayout(new FlowLayout());
11     pane.add(new JButton("eins"));
12     pane.add(new JButton("zwei"));
13     pane.add(new JButton("drei (ein langer Knopf)"));
14     pane.add(new JButton("vier"));
15     pane.add(new JButton("fuenf"));
16     pack();
17     setVisible(true);
18   }
19
```

```
20    public static void main(String [] _){new FlowLayoutTest();}
21  }
```

Das Fenster hat folgende optische Ausprägung.

Verändert man mit der Maus die Fenstergröße, macht es z.B. schmal und hoch, so werden die Knöpfe nicht mehr nebeneinander sonder übereinander angeordnet.

Border Layout

Die Klasse `BorderLayout` definiert einen Layout Manager, der fünf feste Positionen kennt: eine Zentralposition, und jeweils links/rechts und oberhalb/unterhalb der Zentralposition eine Position für Unterkomponenten. Die Methode `add` kann in diesem Layout auch noch mit einem zweitem Argument aufgerufen werden, das eine dieser fünf Positionen angibt. Hierzu bedient man sich der konstanten Felder der Klasse `BorderLayout`.

Beispiel:
In dieser Klasse wird die Klasse `BorderLayout` zur Steuerung des Layoout benutzt. Die fünf Knöpfe werden an jeweils eine der fünf Positionen hinzugefügt:

BorderLayoutTest.java

```
1   package name.panitz.gui.layoutTest;
2
3   import java.awt.*;
4   import javax.swing.*;
5
6   class BorderLayoutTest extends JFrame {
7
8     public BorderLayoutTest(){
9       Container pane = getContentPane();
10      pane.setLayout(new BorderLayout());
11      pane.add(new JButton("eins"),BorderLayout.NORTH);
12      pane.add(new JButton("zwei"),BorderLayout.SOUTH);
13      pane.add(new JButton("drei (ein langer Knopf)")
14                                   ,BorderLayout.CENTER);
15      pane.add(new JButton("vier"),BorderLayout.WEST);
16      pane.add(new JButton("fuenf"),BorderLayout.EAST);
17      pack();
18      setVisible(true);
19    }
20
21    public static void main(String [] _){new BorderLayoutTest();}
22  }
```

Die Klasse erzeugt das folgende Fenster.

Das Layout ändert sich nicht, wenn man mit der Maus die Größe und das Format des Fensters verändert.

Grid Layout

Die Klasse `GridLayout` ordnet die Unterkomponenten tabellarisch an. Jede Komponente wird dabei gleich groß ausgerichtet. Die Größe richtet sich also nach dem größten Element.

Beispiel:
Folgende Klasse benutzt ein Grid-Layout mit zwei Zeilen zu je drei Spalten.

GridLayoutTest.java

```
1  package name.panitz.gui.layoutTest;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  class GridLayoutTest extends JFrame {
7    public GridLayoutTest(){
8      Container pane = getContentPane();
9      pane.setLayout(new GridLayout(2,3));
10     pane.add(new JButton("eins"));
11     pane.add(new JButton("zwei"));
12     pane.add(new JButton("drei (ein langer Knopf)"));
13     pane.add(new JButton("vier"));
14     pane.add(new JButton("fünf"));
15     pack();
16     setVisible(true);
17   }
18
19   public static void main(String [] _){new GridLayoutTest();}
20 }
```

Folgendes Fensters wird durch dieses Programm geöffnet.

Auch hier ändert sich das Layout nicht, wenn man mit der Maus die Größe und das Format des Fensters verändert.

2.2 Exkurs: verschachtelte Klassen

Bisher haben wir gelernt, daß jede Java Quelltextdatei genau eine Klasse enthält und Klassen Eigenschaften in Form von Feldern und Methoden enthalten. In den ersten GUI-Beispielen und Aufgaben haben wir schon gesehen, daß es insbesondere für die Ereignisbehandlung oft zu sehr kleinen Klassen kommt, die lediglich für eine ganz spezielle Funktionalität innerhalb einer GUI-Klasse benötigt werden. Um in solchen Situationen den Code etwas struktuierter zu schreiben, so daß ein Ereignisbehandler direkt im Quelltext bei dem GUI-Objekt zu finden ist, gibt es in Java die Möglichkeit, innerhalb einer Klasse ein interne Klasse zu schreiben. Hierbei spricht man von verschachtelten Klassen. Verschachtelte Klassen können statisch sein, oder als nicht-statische Eigenschaft an konkrete Objekte gebunden sein. In diesem Fall heißen sie: *innere Klassen*. Tatsächlich können verschachtelte überall im Code auftauchen: sie können auf obere Ebene parallel zu Feldern und Methoden in einer Klasse definiert werden, aber auch an beliebiger Stelle innerhalb eines Methodenrumpfes.

2.2.1 Innere Klassen

Eine innere Klasse wird gechrieben wie jede andere Klasse auch, nur daß sie eben im Rumpf einer äußeren Klasse auftauchen kann. Die innere Klasse hat das Privilig auf die Eigenschaften der äußeren Klasse zuzugreifen, sogar auf die als privat markierten Eigenschaften. Das Attribut privat soll lediglich verhindern, daß eine Eigenschaft von außerhalb der Klasse benutzt wird. Innere Klasse befinden sich aber innerhalb der Klasse.

Beispiel:
Unser erstes GUI mit einer Funktionalität läßt sich jetzt mit Hilfe einer inneren Klasse in einer Datei schreiben. Das Feld `counter`, das wir in der vorherigen Implementierung als privates Feld der Klass `CounterListener` definiert hatten, haben wir hier als Feld der GUI-Klasse modelliert. Trotzdem kann die Klasse `CounterListener` weiterhin darauf zugreifen. Ebenso brauch die Textfläche nicht der Klasse `CounterListener` im Konstruktor übergeben werden. Als innere Klasse kann in `CounterListener` auf dieses Feld der äußeren Klasse zugegriffen werden.

InnerCounter.java

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
3  import java.awt.event.*;
4
5  class InnerCounter extends JTB {
6    private int counter = 0;
7
8    class CounterListener implements ActionListener{
9      public void actionPerformed(ActionEvent _){
```

```
10        counter = counter+1;
11        textArea.setText(counter+"");
12      }
13    }
14
15    public InnerCounter(){
16      button.addActionListener(new CounterListener());
17    }
18
19    public static void main(String [] _){new InnerCounter();}
20  }
```

Tatsächlich ist die Implementierung kürzer und etwas übersichtlicher geworden.

Beim Übersetzen einer Klasse mit inneren Klassen, erzeugt der Javaübersetzer für jede innere Klasse eine eigene Klassendatei:

```
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui> ll *.class
-rw-r--r--    1 sep      users        1082 2004-03-29 11:36 InnerCounter$CounterListener.class
-rw-r--r--    1 sep      users         892 2004-03-29 11:36 InnerCounter.class
```

Der Javaübersetzer schreibt intern den Code um in eine Menge von Klassen ohne innere Klassendefinition und erzeugt für diese den entsprechenden Code. Für die innere Klasse generiert der Javaübersetzer einen Namen, der sich aus äußeren und inneren Klassenamen durch ein Dollarzeichen getrennt zusammensetzt.

Aufgabe 3 Schreiben Sie Ihr Programm eines Zählers mit drei Knöpfen jetzt so, daß sie innere Klassen benutzen.

2.2.2 Anonyme Klassen

Im letzten Abschnitt hatten wir bereits das Beispiel einer inneren Klasse, für die wir genau einmal ein Objekt erzeugen. In diesem Fall wäre es eigentlich unnötig für eine solche Klasse einen Namen zu erfinden, wenn man an genau dieser einen Stelle, an der das Objekt erzeugt wird, die entsprechende Klasse spezifizieren könnte. Genau hierzu dienen anonyme Klassen in Java. Sie ermöglichen, Klassen ohne Namen zu instanziieren. Hierzu ist nach dem Schlüsselwort `new` anzugeben, von welcher Oberklasse namenlose Klasse ableiten soll, oder welche Schnittstelle mit der namenlosen Klasse implementiert werden soll. Dann folgt nach dem leeren Klammerpaar für dem Konstruktoraufruf in geschweiften Klammern der Rumpf der namenlosen Klasse.

Beispiel:
Wir schreiben ein drittes Mal die Klasse `Counter`. Diesmal wird statt der nur einmal instanziierten inneren Klasse eine anonyme Implementierung der Schnittstelle `ActionListener` Instanziiert.

AnonymousCounter.java

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
```

```
3  import java.awt.event.*;
4
5  class AnonymousCounter extends JTB {
6    private int counter = 0;
7
8    public AnonymousCounter(){
9      button.addActionListener(
10       new ActionListener(){
11         public void actionPerformed(ActionEvent _){
12           counter = counter+1;
13           textArea.setText(counter+"");
14         }
15       });
16   }
17
18   public static void main(String [] _){new AnonymousCounter();}
19 }
```

Auch für anonyme Klassen generiert der Javaübersetzer eigene Klassendateien. Mangels eines Names, numeriert der Javaübersetzer hierbei die inneren Klassen einfach durch.

```
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui> ll *.class
-rw-r--r--    1 sep      users        1106 2004-03-29 11:59 AnonymousCounter$1.class
-rw-r--r--    1 sep      users         887 2004-03-29 11:59 AnonymousCounter.class
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui>
```

2.2.3 Statische verschachtelte Klassen

Selten benutzt gibt es auch statische verschachtelte Klassen. Diese können entsprechend auch nur auf statische Eigenschaften der äußeren Klasse zugreifen.

2.3 Selbstdefinierte graphische Komponenten

Bisher graphische Komponenten unter Verwendung der fertigen GUI-Komponente aus der Swing-Bibliothek zusammengesetzt. Oft will man graphische Komponenten schreiben, für die es keine fertige GUI-Komponente in der Swing-Bibliothek gibt. Wir müssen wir eine entsprechende Komponente selbst schreiben.

2.3.1 Graphics Objekte

Um eine eigene GUI-Komponente zu schreiben, schreibt man eine Klasse, die von der GUI-Klasse ableitet. Dieses haben wir bereits in den letzten Beispielen getan, indem wir von der Klasse `JFrame` abgelitten haben. Die dort geschriebenen Unterklassen der Klasse `JFrame` zeichneten sich dadurch aus, daß sie eine Menge von graphischen Objekten (Knöpfe, Textfelder...) in einer Komponente zusammengefasst haben. In diesem Abschnitt werden wir eine neue Komponente definieren, die keine der bestehenden fertigen Komponenten benutzt, sondern selbst alles zeichnet, was zu ihrer Darstellung notwendig ist.

Hierzu betrachten wir eine der entscheidenen Methoden der Klasse `JComponent`, die Methode `paintComponent`. In dieser Methode wird festgelegt, was zu zeichnen ist, wenn die graphische Komponente darzustellen ist. Die Methode `paintComponent` hat folgende Signatur:

```
1 public void paintComponent(java.awt.Graphics g)
```

Java ruft diese Methode immer auf, wenn die graphische Komponente aus irgendeinen Grund zu zeichnen ist. Dabei bekommt die Methode das Objekt übergeben, auf dem gezeichnet wird. Dieses Objekt ist vom Typ `java.awt.Graphics`. Es stellt ein zweidimensionales Koordinatensystem dar, in dem zweidimensionale Graphiken gezeichnet werden können. Der Nullpunkt dieses Koordinatensystems ist oben links und nicht unten links, wie wir es vielleicht aus der Mathematik erwartet hätten.

In der Klasse `Graphics` sind eine Reihe von Methoden definiert, die es erlauben graphische Objekte zu zeichnen. Es gibt Methoden zum Zeichnen von Geraden, Vierecken, Ovalen, beliebigen Polygonzügen, Texten etc.

Wollen wir eine eigene graphische Komponente definieren, so können wir die Methode `paintComponent` überschreiben und auf dem übergebenen Objekt des Typs `Graphics` entsprechende Methoden zum Zeichnen aufrufen. Um eine eigene graphische Komponente zu definieren, wird empfohlen die Klasse `JPanel` zu erweitern und in ihr die Methode `paintComponent` zu überschreiben.

Beispiel:
Folgende Klasse definiert eine neue graphische Komponente, die zwei Linien, einen Text, ein Rechteck, ein Oval und ein gefülltes Kreissegment enthält.

SimpleGraphics.java

```
1  package name.panitz.gui.graphicsTest;
2
3  import javax.swing.JPanel;
4  import javax.swing.JFrame;
5  import java.awt.Graphics;
6
7  class SimpleGraphics extends JPanel{
8    public void paintComponent(Graphics g){
9      g.drawLine(0,0,100,200);
10     g.drawLine(0,50,100,50);
11     g.drawString("hallo",10,20);
12     g.drawRect(10, 10, 60,130);
13     g.drawOval( 50,  100, 30, 80);
14     g.fillArc(-20, 150, 80, 80, 0, 50);
15   }
16 }
```

Diese Komponente können wir wie jede andere Komponente auch einem Fenster hinzufügen, so daß sie auf dem Bildschirm angezeigt werden kann.

UseSimpleGraphics.java

```
1  package name.panitz.gui.graphicsTest;
2
3  import javax.swing.JFrame;
```

```
4
5  class UseSimpleGraphics {
6    public static void main(String [] args){
7      JFrame frame = new JFrame();
8      frame.getContentPane().add(new SimpleGraphics());
9
10     frame.pack();
11     frame.setVisible(true);
12   }
13 }
```

Wird dieses Programm gestartet, so öfnet Java das Fenster aus Abbildung 2.4 auf dem Bildschirm.

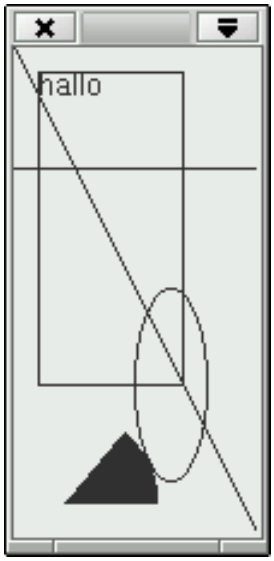


Abbildung 2.4: Einfache graphische Komponente.

Erst wenn wir das Fenster mit der Maus größer ziehen, können wir das ganze Bild sehen.

2.3.2 Dimensionen

Ärgerlich in unserem letzten Beispiel war, daß Java zunächst ein zu kleines Fenster für unsere Komponente geöffnet hat, und wir dieses Fenster mit Maus erst größer ziehen mußten. Die Klasse `JComponent` enthält Methoden, in denen die Objekte angeben können, welches ihre bevorzugte Größe bei ihrere Darstellung ist. Wenn wir diese Methode überschreiben, so daß sie eine Dimension zurückgibt, in der das ganze zu zeichnende Bild passt, so wird von Java auch ein entsprechend großes Fenster geöffnet. Wir fügen der Klasse `SimpleGraphics` folgende zusätzliche Methode hinzu.

```
1   public java.awt.Dimension getPreferredSize() {
```

```
2     return new java.awt.Dimension(100,200);
3   }
```

Jetzt öffnet Java ein Fenster, in dem das ganze Bild dargestellt werden kann.

Aufgabe 4 (Punkteaufgabe 2 Punkte) Schreiben Sie eine GUI-Komponente `StrichKreis`, die mit geraden Linien einen Kreis entsprechend untenstehender Abbildung malt. Der Radius des Kreises soll dabei dem Konstruktor der Klasse `StrichKreis` als `int`-Wert übergeben werden.

Testen Sie Ihre Klasse mit folgender Hauptmethode:

```
1  public static void main(String [] args) {
2    StrichKreis k = new StrichKreis(120);
3    JFrame f = new JFrame();
4    JPanel p = new JPanel();
5    p.add(new StrichKreis(120));
6    p.add(new StrichKreis(10));
7    p.add(new StrichKreis(60));
8    p.add(new StrichKreis(50));
9    p.add(new StrichKreis(70));
10   f.getContentPane().add(p);
11
12   f.pack();
13   f.setVisible(true);
14 }
```

Hinweis: In der Klasse `java.lang.Math` finden Sie Methoden trigeometrischer Funktionen. Insbesondere `toRadians` zum Umrechnen von Gardwinkel in Bogenmaß sowie `sin` und `cos`.

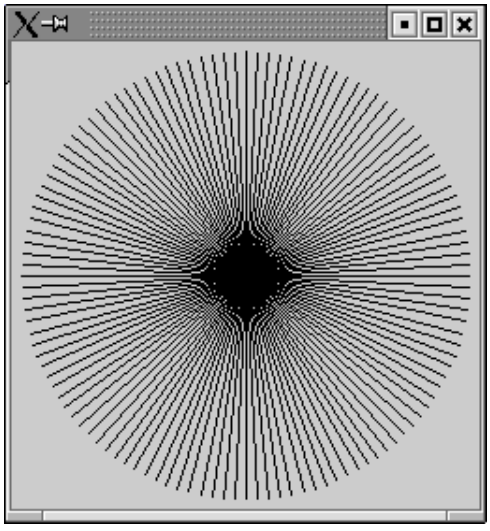


Abbildung 2.5: Kreis mit Durchmesserlinien.

2.3.3 Farben

In der graphische Komponente aus dem letzten Abschnitt haben wir uns noch keine Gedanken über die Farbe der gezeichneten Grafik gemacht. Die Klasse `Graphics` stellt eine Methode bereit, die es erlaubt die Farbe des Stiftes für die Grafik auf einen bestimmten Wert zu setzen. Alle anschließenden Zeichenbefehle auf das Objekt des Typs `Graphics` werden dann mit dieser Farbe vorgenommen, solange bis die Methode zum Setzen der Farbe ein weiteres Mal aufgerufen wird. Die Methode zum Setzen der Farbe auf einem Objekt der Klasse `Graphics` hat folgende Signatur:

```
1 public abstract void setColor(java.awt.Color c)
```

Zusätzlich gibt es für Komponenten noch eine Methode, die die Hintergrundfarbe der Komponente beschreibt.

Farben werden mit der Klasse `java.awt.Color` beschrieben. Hier gibt es mehrere Konstruktoren, um eine Farbe zu spezifizieren, z.B. einen Konstruktor, der Intensität der drei Grundfarben in einen Wert zwischen 0 und 255 festlegt.

Für bestimmte gängige Farben stellt die Klasse `Color` Konstanten zur Verfügung, die wir benutzen können, wie z.B.: `Color.RED` oder `Color.GREEN`.

Beispiel:
Wir können eine kleine Klasse schreiben, in der wir ein wenig mit Farben spielen. Wir setzen die Hintergrundfarbe des Objekts auf einen dunklen Grauwert, zeichnen darin ein dunkelgrünes Rechteck, ein rotes Oval und ein gelbes Oval.

ColorTest.java

```
1  package name.panitz.gui.graphicsTest;
2
3  import javax.swing.JPanel;
4  import javax.swing.JFrame;
5
6  public class ColorTest extends JPanel{
7    public ColorTest(){
8      //Hintergrundfarbe auf einen dunklen Grauton setzen
9      setBackground(new java.awt.Color(100,100,100));
10   }
11
12   public void paintComponent(java.awt.Graphics g){
13     //Farbe auf einen dunklen Grünton setzen
14     g.setColor(new java.awt.Color(22,178,100));
15     //Rechteck zeichnen
16     g.fillRect(25,50,50,100);
17     //Farbe auf rot setzen
18     g.setColor(java.awt.Color.RED);
19     g.fillOval(25,50,50,100);
20     //Farbe auf gelb setzen
21     g.setColor(java.awt.Color.YELLOW);
22     g.fillOval(37,75,25,50);
23   }
```

```
24
25    public java.awt.Dimension getPreferredSize() {
26      return new java.awt.Dimension(100,200);
27    }
28
29    public static void main(String [] _){
30      javax.swing.JFrame f = new JFrame("Farben");
31      f.getContentPane().add(new ColorTest());
32      f.pack();f.setVisible(true);
33    }
34  }
```

Das Programm ergibt das Bild aus Abbildung 2.6.

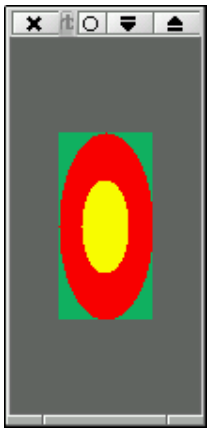


Abbildung 2.6: Erste farbige Graphik.

Aufgabe 5 Erweitern Sie die Strichkreisklasse aus der letzten Aufgabe, so daß jeder Strich in einer anderen Farbe gezeichnet wird.

Fraktale

Um noch ein wenig mit Farben zu spielen, zeichnen wir in diesem Abschnitt die berühmten Apfelmännchen. Apfelmännchen werden definiert über eine Funktion auf komplexen Zahlen. Die aus der Mathematik bekannten komplexen Zahlen sind Zahlen mit zwei reellen Zahlen als Bestandteil, den sogenannten Imaginärteil und den sogenannten Realteil. Wir schreiben zunächst eine rudimentäre Klasse zur Darstellung von komplexen Zahlen:

Complex.java

```
1  package name.panitz.crempel.tool.apfel;
2
3  public class Complex{
```

Diese Klasse braucht zwei Felder um Real- und Imaginärteil zu speichern:

Complex.java
```
4   public double re;
5   public double im;
```

Ein naheliegender Konstruktor für komplexe Zahlen füllt diese beiden Felder.

Complex.java
```
6  public Complex(double re,double im){
7   this.re=re;this.im=im;
8  }
```

Im Mathematikbuch schauen wir nach, wie Addition und Multiplikation für komplexe Zahlen definiert sind, und schreiben entsprechende Methoden:

Complex.java
```
9   public Complex add(Complex other){
10    return new Complex(re+other.re,im+other.im);
11  }
12
13  public Complex mult(Complex other){
14    return new Complex
15      (re*other.re-im*other.im,re*other.im+im*other.re);
16  }
```

Zusätzlich finden wir in Mathematik noch die Definition der Norm einer komplexen Zahl und setzen auch diese Definition in eine Methode um. Zum Quadrat des Realteils wird das Quadrat des Imaginärteils addiert.

Complex.java
```
17  public double norm(){return re*re+im*im;}
18 }
```

Soweit komplexe Zahlen, wie wir sie für Apfelmännchen brauchen.

Grundlage zum Zeichnen von Apfelmännchen ist folgende Iterationsgleichung auf komplexen Zahlen: $z_{n+1} = z_n^2 + c$. Wobei z_0 die komplexe Zahl $0 + 0i$ mit dem Real- und Imaginärteil 0 ist.

Zum Zeichnen der Apfelmännchen wird ein Koordinatensystem so interpretiert, daß die Achsen jeweils Real- und Imaginärteil von komplexen Zahlen darstellen. Jeder Punkt in diesem Koordinatensystem steht jetzt für die Konstante c in obiger Gleichung. Nun wir geprüft ob und für welches n die Norm von z_n größer eines bestimmten Schwellwertes ist. Je nach der Größe von $n$wird der Punkt im Koordinatensystem mit einer anderen Farbe eingefärbt.

Mit diesem Wissen können wir nun versuchen die Apfelmännchen zu zeichnen. Wir müssen nur geeignete Werte für die einzelnen Parameter finden. Wir schreiben eine eigene Klasse für das graphische Objekt, in dem ein Apfelmännchen gezeichnet wird. Wir deklarieren die Imports der benötigten Klassen:

Apfelmaennchen.java

```
1  package name.panitz.crempel.tool.apfel;
2
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import java.awt.Dimension;
6  import javax.swing.JFrame;
7  import javax.swing.JPanel;
8
9  public class Apfelmaennchen extends JPanel {
```

Als erstes deklarieren wir Konstanten für die Größe des Apfelmännchens.

Apfelmaennchen.java

```
10   final int width = 480;
11   final int height = 430;
```

Eine weitere wichtige Konstante ist der Faktor, der angibt, welcher reellen Zahl ein Pixel entspricht:

Apfelmaennchen.java

```
12   double zelle=0.00625;
```

Eine weitere Konstanten legt die Farbe fest, mit der die Punkte, die nicht über einen bestimmten Schwellwert konvergieren, eingefärbt werden sollen:

Apfelmaennchen.java

```
13   final Color colAppleman = new Color(0,129,190);
```

Weitere Konstanten legen fest welche komplexe Zahl der Nullpunkt unseres `Graphics`-Objekts darstellt.

Apfelmaennchen.java

```
14   double startX = -2;
15   double startY = -1.35;
```

Weitere Konstanten sind der Schwellwert und die maximale Rekusrionstiefe n, für die wir jeweils z_n berechnen:

Apfelmaennchen.java

```
16   final int recDepth = 50;
17   final int schwellwert = 4;
```

Die wichtigste Methode berechnet die Werte für die Gleichung $z_{n+1} = z_n^2 + c$. Der Eingabeparameter ist die komplexe Zahl c. Das Ergebnis dieser Methode ist das n, für das z_n größer als der Schwellwert ist:

Apfelmaennchen.java

```
18  //C-Werte checken nach zn+1 = zn*zn + c,
19  public int checkC(Complex c) {
20    Complex zn = new Complex(0,0);
21
22    for (int n=0;n<recDepth;n=n+1) {
23      final Complex znp1 = zn.mult(zn).add(c);
24      if (znp1.norm() > schwellwert) return n;
25      zn=znp1;
26    }
27    return recDepth;
28  }
```

Jetzt gehen wir zum Zeichnen jedes Pixel unseres `Graphics`-Objekts durch, berechnen welche komplexe Zahl an dieser Stelle steht und benutzen dann die Methode *checkC*, um zu berechnen ob und nach wieviel Iterationen die Norm von z_n größer als der Schwellwert wird. Abhängig von dieser Zahl, färben wir den Punkt mit einer Farbe ein.

Apfelmaennchen.java

```
29  public void paint(Graphics g) {
30    for (int y=0;y<height;y=y+1) {
31      for (int x=0;x<width;x=x+1) {
32
33        final Complex current
34          =new Complex(startX+x*zelle,startY+y*zelle);
35
36        final int iterationenC = checkC(current);
37
38        paintColorPoint(x,y,iterationenC,g);
39      }
40    }
41  }
```

Zur Auswahl der Farbe benutzen wir folgende kleine Methode, die Abhängig von ihrem Parameter `it` an der Stelle `(x,y)` einen Punkt in einer bestimmten Farbe zeichnet.

Apfelmaennchen.java

```
42  private void paintColorPoint
43                 (int x,int y,int it,Graphics g){
44    final Color col
45      = it==recDepth
46        ?colAppleman
47        :new Color(255-5*it%1,255-it%5*30,255-it%5* 50);
48    g.setColor(col);
49    g.drawLine(x,y,x,y);
50  }
```

Schließlich können wir noch die Größe festlegen und das Ganze in einer Hauptmethode starten:

Apfelmaennchen.java

```
51    public Dimension getPreferredSize(){
52      return new Dimension(width,height);
53    }
54
55    public static void main(String [] args){
56      JFrame f = new JFrame();
57      f.getContentPane().add(new Apfelmaennchen());
58      f.pack();
59      f.setVisible(true);
60    }
61  }
```

Das Programm ergibt das Bild aus Abbildung 2.7.

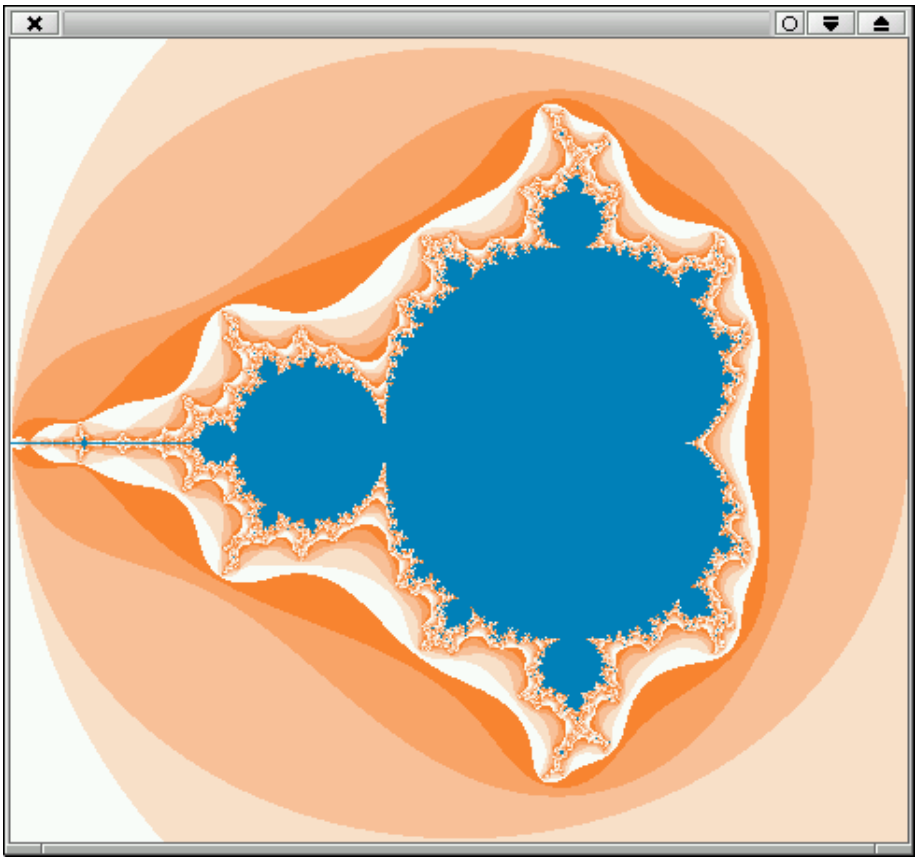


Abbildung 2.7: Apfelmännchen.

2.3.4 Fonts und ihre Metrik

Ähnlich wie für Farben hat ein Objekt des Typs `Graphics` auch einen aktuellen Font. Dieser ist vom Typ `java.awt.Font`. Ebenso wie es die Methoden `getColor` und `setColor` in der

Klasse `Graphics` gibt, gibt es dort auch Methoden `getFont` und `setFont`.

Eine wichtige Information zu einem Font ist beim Zeichnen, wieviel Platz ein bestimmter Text für in diesem Font benötigt. Hierzu existiert eine Klasse `java.awt.FontMetrics`. Entsprechend gibt es Methoden, die für ein `Graphics`-Objekt ein Objekt mit den aktuellen `FontMetrics`-Objekt zurückgibt.

Beispiel:
Folgende Kleine Klasse vergrößert nach und nach den Font, und malt in verschiedenen Größen einen Text in ein Fenster.

FontTest.java

```
1  package name.panitz.gui.graphicsTest;
2
3  import java.awt.Font;
4  import java.awt.FontMetrics;
5  import java.awt.Graphics;
6  import java.awt.Dimension;
7  import javax.swing.JPanel;
8  import javax.swing.JFrame;
9
10
11 class FontTest extends JPanel {
12
13   public Dimension getPreferredSize(){
14     return new Dimension(250,650);
15   }
16
17   public void paintComponent(Graphics g){
18     //die y-Koordinate für einen Text
19     int where = 0;
20
21     //25 mal
22     for (int i=1; i<=25;i=i+1){
23       //nimm aktuellen Font und seine Maße
24       final Font f = g.getFont();
25       final FontMetrics fm = g.getFontMetrics(f);
26
27       //gehe die Höhe des Fonts weiter in Richtung x
28       where = where + (f.getSize()) ;
29
30       //male "hallo"
31       g.drawString("hallo",0,where);
32
33       //berechne wo der gerade gemalte String endet
34       //und male dort "welt"
35       g.drawString
36         ("welt!",fm.stringWidth("hallo "),where);
37
38       //verändere die Fontgröße. Erhöhe sie um 1.
39       g.setFont(getFont().deriveFont(f.getSize2D()+1));
40     }
```

```
41    }
42
43    public static void main(String [] _){
44      JFrame f = new JFrame("Fonts");
45      f.getContentPane().add(new FontTest());
46      f.pack();
47      f.setVisible(true);
48    }
49  }
```

Das Programm ergibt das Bild aus Abbildung 2.8.

2.3.5 Erzeugen graphischer Dateien

Bisher haben wir alles auf dem Bildschirm in irgendeinen Fenster gezeichnet. Das Schöne ist, daß wir die gleichen graphischen Objekte jetzt benutzen können, um Graphiken in eine Datei zu schreiben.

Hierzu stellt Java die Klasse `java.awt.image.RenderedImage` zur Verfügung. Für Objekte dieser Klasse gibt es ein Objekt des Typs `Graphics`. Man kann also ein `RenderedImage`-Objekt erzeugen und mit beliebige `paint`-Methoden auf dessen `Graphics`-Objekt anwenden.

In der Klasse `javax.imageio.ImageIO` gibt es schließlich Methoden, die es erlauben ein `RenderedImage`in eine Datei zu schreiben.

Beispiel:
Wir schreiben eine kleine Klasse mit Methoden, um das Bild einer graphischen Komponente in eine Datei zu speichern.

Zunächst brauchen wir eine ganze Reihe von Klassen, die wir importieren:

ComponentToFile.java

```
1  package name.panitz.gui.graphicsFile;
2
3  import name.panitz.crempel.tool.apfel.Apfelmaennchen;
4  import java.awt.image.BufferedImage;
5  import java.awt.image.RenderedImage;
6  import java.awt.Dimension;
7  import java.awt.Graphics;
8
9  import javax.imageio.ImageIO;
10 import javax.swing.JComponent;
11
12 import java.io.File;
13 import java.io.IOException;
```

Wir schreiben eine Hauptmethode, die als Kommandozeilenparameter den Namen der zu schreibenen Bilddatei übergeben bekommt:

ComponentToFile.java

```
14 public class ComponentToFile {
15   public static void main(String [] args){
```

Abbildung 2.8: Verschiedene Fontgrößen.

Zu Testzwecken schreiben wir je eine `png`- und eine `jpg`-Datei, in die wir die Komponente zeichnen wollen:

ComponentToFile.java

```
16    try {
17      final JComponent component = new Apfelmaennchen();
18      final RenderedImage image = createComponentImage(component);
19      ImageIO.write(image,"png", new File(args[0]+".png"));
```

```
20      ImageIO.write(image,"jpg", new File(args[0]+".jpg"));
21    }catch (IOException e) {System.out.println(e);}
22  }
```

Die folgende Methode erzeugt das eigentliche Bild der Komponente:

ComponentToFile.java

```
23  static public RenderedImage createComponentImage
24                                  (JComponent component) {
```

Wir erzeugen ein Bildobjekt in der für die Komponente benötigten Größe.

ComponentToFile.java

```
25      Dimension d = component.getPreferredSize();
26      BufferedImage bufferedImage
27       = new BufferedImage((int)d.getWidth(),(int)d.getHeight()
28                           ,BufferedImage.TYPE_INT_RGB);
```

Nun besorgen wir dessen `Graphics`-Umgebung:

ComponentToFile.java

```
29    Graphics g = bufferedImage.createGraphics();
```

Darin läßt sich unsere Komponente zeichnen:

ComponentToFile.java

```
30    component.paint(g);
```

Die Methode kann das fertig gezeichnete Bild als Ergebnis zurückgeben:

ComponentToFile.java

```
31      return bufferedImage;
32    }
33  }
```

2.3.6 Graphics2D

Seit Java 1.2 existiert eine Unterklasse der Klasse `Graphics` mit wesentlich mächtigeren Mehoden zum Zeichnen von 2-Dimenionalen Graphiken, die Klasse `java.awt.Graphics2D`. Leider konnten nicht die Signaturen der entsprechenden Methoden `paint` in den Komponentenklassen geändert werden, so daß dem übergebene `Graphics`-Objekt zunächst zugesichert werden muß, daß es sich um ein `Graphics2D`-Objekt handelt:

```
1  public void paint(Graphics g) {
2      Graphics2D g2 = (Graphics2D) g;
3      ...
4  }
```

`Graphics2D` enthält Methoden zum Darstellen verschiedener Formen, wahlweise umrandet oder gefüllt. Auf geometrische Formen können Transformationen angewendet werden, einzelne Formen können kombiniert werden zu komplexeren Figuren, Farbverläufe können definiert werden. Auch Text stellt sich hierbei als geometrische Form dar.

Beispiel:
Folgende Klasse benutzt willkürlich ein paar Eigenschaften der `Graphics2D`-Klasse, um einen Eindruck über die Arbeitsweise dieser Klasse zu geben. Eine systematische Behandlung der graphischen ist nicht Gegenstand dieser Vorlesung.

G2DTest.java

```
1  package name.panitz.gui.g2d;
2  import javax.swing.JPanel;
3  import javax.swing.JFrame;
4
5  import java.awt.font.*;
6  import java.awt.*;
7  import java.awt.geom.*;
8
9  public class G2DTest extends JPanel{
10   int w = 550;
11   int h = 650;
12
13   public Dimension getPreferredSize(){
14     return new Dimension(w,h);}
15
16   public void paintComponent(Graphics g){
17     Graphics2D g2 = (Graphics2D) g;
18     TextLayout textTl
19      = new TextLayout
20              ("Kommunismus", new Font("Helvetica", 1, 96)
21              , new FontRenderContext(null, false, false));
22
23     AffineTransform textAt = new AffineTransform();
24     textAt.translate(0,(float)textTl.getBounds().getHeight());
25     Shape shape = textTl.getOutline(textAt);
26
27     AffineTransform at = new AffineTransform();
28     at.rotate(Math.toRadians(45));
29     at.shear(1.5, 0.0);
30
31     g2.transform(at);
32
33     float dash[] = {10.0f};
34     g2.setStroke(
35       new BasicStroke
36            (3.0f, BasicStroke.CAP_BUTT
37            ,BasicStroke.JOIN_MITER, 10.0f, dash, 0.0f)
38     );
39
40     g2.setPaint(
41       new GradientPaint(0,0,Color.black,w,h
42                         ,new Color(255,100,100),false)
43     );
44
```

```
45      g2.fill(shape);
46      g2.setColor(Color.darkGray);
47      g2.draw(shape);
48    }
49
50    public static void main(String [] _){
51      JFrame f = new JFrame("G2");
52      f.getContentPane().add(new G2DTest());
53      f.pack();
54      f.setVisible(true);
55    }
56  }
```

Das Programm öffnet das Fenster aus Abbildung 2.9 auf dem Bildschirm.

2.4 Weitere Komponente und Ereignisse

Im Prinzip haben wir bisher alles kennengelernt um graphische Benutzerflächen und Anwendungen zu beschreiben. Die Schwierigkeit bei graphischen Anwendungen liegt hauptsächlich in der schieren Menge der unterschiedlichen Komponenten und ihrem Zusammenspiel. Die Anzahl der vordefinierten Komponenten der Swing-Bibliothek ist immens. Diese Komponenten können auf eine Vielzahl von unterschiedlichen Ereignisarten reagieren. Bisher haben wir nur die SChnittstelle `ActionListener` kennengelernt, die lediglich ganz allgemein ein Ereignis behandelt.

In diesem Abschnitt schauen wir uns ein paar wenige weitere Komponenten und Ereignistypen an.

2.4.1 Mausereignisse

Ein in modernen graphischen Oberflächen häufigst benutztes Eingabemedium ist die Maus. Zwei verschiedene Ereignisarten sind für die Maus relevant:

- Mausereignisse, die sich auf das Drücken, Freilassen oder Klicken auf einen der Mausknöpfe bezieht. Hierfür gibt es eine Schnittstelle zur Behandlung solcher Ereignisse: `MouseListener`.
- Mausereignisse, die sich auf das Bewegen der Maus beziehen. Die Behandlung solcher Ereignisse kann über eine Implementierung der Schnittstelle `MouseMotionListener` spezifiziert werden.

Entsprechend gibt es für graphische Komponenten Methoden, um solche Mausereignisbehandler der Komponente hinzuzufügen: `addMouseListener` und `addMouseMotionListener`.

Um die Arbeit mit Ereignisbehandlern zu vereinfachen, gibt es für die entsprechnden Schnittstellen im Paket `java.awt.event` prototypische Implementierungen, in denen die Methoden der Schnittstelle so implementiert sind, daß ohne Aktion auf die entsprechenden Ereignisse reagiert wird. Diese prototypischen Implementierung sind Klassen, deren Namen mit *Adapter* enden. So gibt es zur Schnittstelle `MouseListener` die implementierende Klasse

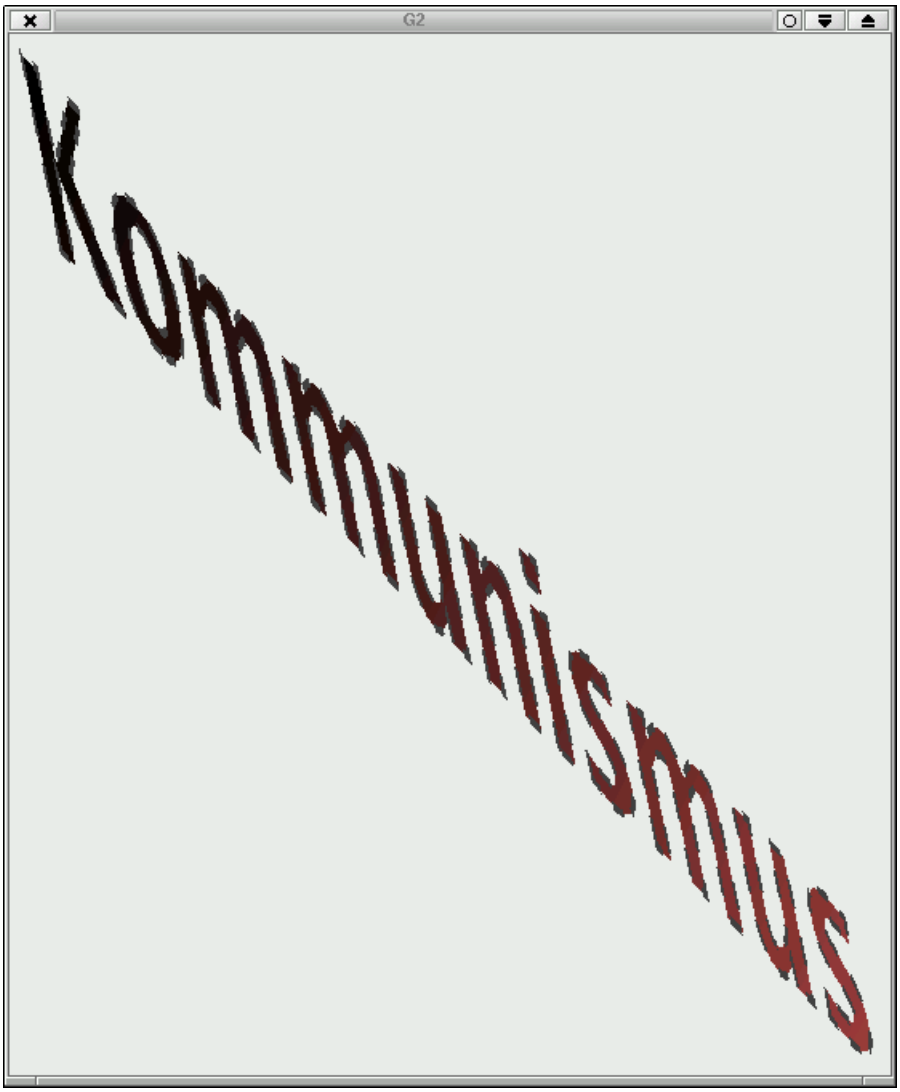


Abbildung 2.9: Transformationen auf Text in Graphics2D.

`MouseAdapter`. Will man eine bestimmte Mausbehandlung programmieren, reicht es aus, diesen Adapter zu erweitern und nur die Methoden zu überschreiben, für die bestimmte Aktionen vorgesehen sind. Es erübrigt sich dann für alle sechs Methoden der Schnittstelle `MouseListener` Implementierungen vorzusehen.

Beispiel:
Wir erweitern die Klasse `Apfelmaennchen` um eine Mausbehandlung. Der mit gedrückter Maus markierte Bereich soll vergrößert in dem Fenster dargestellt werden.

ApfelWithMouse.java

```
1 package name.panitz.crempel.tool.apfel;
2
```

```
3  import java.awt.Graphics;
4  import java.awt.event.*;
5  import javax.swing.JFrame;
6
7  public class ApfelWithMouse extends Apfelmaennchen{
8    public ApfelWithMouse(){
```

Im Konstruktor fügen wir der Komponente eine Mausbehandlung hinzu. Der Mausbehandler merkt sich die Koordinaten, an denen die Maus gedrückt wird und berechnet beim Loslassen des Mausknopfes den neuen darzustellenden Zahlenbereich:

ApfelWithMouse.java

```
9      addMouseListener(new MouseAdapter(){
10       int mouseStartX=0;
11       int mouseStartY=0;
12
13       public void mousePressed(MouseEvent e) {
14         mouseStartX=e.getX();
15         mouseStartY=e.getY();
16       }
17
18       public void mouseReleased(MouseEvent e) {
19         int endX = e.getX();
20         int endY = e.getY();
21         startX = startX+(mouseStartX*zelle);
22         startY = startY+(mouseStartY*zelle);
23         zelle  = zelle*(endX-mouseStartX)/width;
24         repaint();
25       }
26     });
27   }
```

Auch für diese Klasse sehen wir eine kleine Startmethode vor:

ApfelWithMouse.java

```
28   public static void main(String [] _){
29     JFrame f = new JFrame();
30     f.getContentPane().add(new ApfelWithMouse());
31     f.pack();
32     f.setVisible(true);
33   }
34 }
```

2.4.2 Fensterereignisse

Auch für Fenster in einer graphischen Benutzeroberfläche existieren eine Reihe von Ereignissen. Das Fenster kann minimiert oder maximiert werden, es kann das aktive Fenster oder im Hintergrund sein und es kann schließlich auch geschlossen werden. Um die Reaktion auf solche Ereignisse zu spezifizieren existiert die Schnittstelle `WindowListener` mit entsprechender

prototypischer Adapterklasse `WindowAdapter`. Die Objekte der Fensterereignisbehandlung können mit der Methode `addWindowListener` Fensterkomponenten hinzugefügt werden.

Beispiel:
In den bisher vorgestellten Programmen wird Java nicht beendet, wenn das einzige Fenster der Anwendung geschlossen wurde. Man kann an der Konsole sehen, daß der Javainterpreter weiterhin aktiv ist. Das liegt daran, daß wir bisher noch nicht spezifiziert haben, wie die Fensterkomponenten auf das Ereignis des Schließens des Fensters reagieren sollen. Dieses kann mit einem Objekt, das `WindowListener` implementiert in der Methode `windowClosing` spezifiziert werden. Wir schreiben hier eine Version des Apfelmännchenprogramms, in dem das Schließen des Fensters den Abbruch des gesammten Programms bewirkt.

ClosingApfelFrame.java

```
1  package name.panitz.crempel.tool.apfel;
2
3  import javax.swing.JFrame;
4  import java.awt.event.*;
5
6  public class ClosingApfelFrame {
7    public static void main(String [] args){
8      JFrame f = new JFrame();
9      f.getContentPane().add(new ApfelWithMouse());
10     f.addWindowListener(new WindowAdapter(){
11       public void windowClosing(WindowEvent e) {
12         System.exit(0);
13       }
14     });
15     f.pack();
16     f.setVisible(true);
17   }
18 }
```

2.4.3 Weitere Komponenten

Um einen kleinen Überblick der vorhandenen Swing Komponenten zu bekommen, können wir ein kleines Programm schreiben, daß möglichst viele Komponenten einmal instanziiert. Hierzu schreiben wir eine kleine Testklasse:

ComponentOverview.java

```
1  package name.panitz.gui.example;
2
3  import javax.swing.*;
4  import java.awt.*;
5  import java.util.*;
6
7  public class ComponentOverview  {
8    public ComponentOverview(){
```

Darin definieren wir eine Reihung von einfachen Swing-Komponenten:

ComponentOverview.java
```
9    JComponent [] cs1 =
10    {new JButton("knopf")
11    ,new JCheckBox("check mich")
12    ,new JRadioButton("drück mich")
13    ,new JMenuItem("ins Menue mit mir")
14    ,new JComboBox(combos)
15    ,new JList(combos)
16    ,new JSlider(0,350,79)
17    ,new JSpinner(new SpinnerNumberModel(18,0.0,42.0,2.0))
18    ,new JTextField(12)
19    ,new JFormattedTextField("hallo")
20    ,new JLabel("einfach nur ein Label")
21    ,new JProgressBar(0,42)
22    };
```

Sowie eine zweite Reihung von komplexeren Swing-Komponenten:

ComponentOverview.java
```
23    JComponent [] cs2 =
24    {new JColorChooser(Color.RED)
25    ,new JFileChooser()
26    ,new JTable(13,5)
27    ,new JTree()
28    };
```

Diese beiden Reihungen zeigen sollen mit einer Hilfsmethode in einem Fenster angezeigt werden:

ComponentOverview.java
```
29    displayComponents(cs1,3);
30    displayComponents(cs2,2);
31   }
```

Für die Listen- und Auswahlkomponenten oben haben wir eine Reihung von Strings benutzt:

ComponentOverview.java
```
32   String [] combos = {"friends","romans","contrymen"};
```

Bleibt die Methode zu schreiben, die die Reihungen von Komponenten anzeigen kann. Als zweites Argument bekommt diese Methode übergeben, in wieviel Spalten die Komponenten angezeigt werden sollen.

ComponentOverview.java
```
33   public void displayComponents(JComponent [] cs,int col){
```

Ein Fenster wird definiert, für das eine Gridlayout-Zwischenkomponente mit genügend Zeilen erzeugt wird:

ComponentOverview.java
```
34      JFrame f = new JFrame();
35      JPanel panel = new JPanel();
36      panel.setLayout(
37       new GridLayout(cs.length/col+(cs.length%col==0?0:1),col));
```

Für jede Komponente wird ein `Panel` mit Rahmen und den Klassennamen der Komponente als Titel erzeugt und der Zwischenkomponente hinzugefügt:

ComponentOverview.java
```
38      for (JComponent c:cs){
39        JPanel p = new JPanel();
40        p.add(c);
41        p.setBorder(BorderFactory
42                    .createTitledBorder(c.getClass().getName()));
43        panel.add(p);
44      }
```

Schließlich wird noch das Hauptfester zusammengepackt:

ComponentOverview.java
```
45      f.getContentPane().add(panel);
46      f.pack();
47      f.setVisible(true);
48    }
```

Und um alles zu starten, noch eine kleine Hauptmethode:

ComponentOverview.java
```
49    public static void main(String [] _){
50      new  ComponentOverview();
51    }
52  }
```

Wir erhalten einmal die Übersicht von Komponenten wie in Abbildung 2.10 und einmal wie in Abbildung 2.11 dargestellt.

Womit wir uns noch nicht im einzelnen beschäftigt haben, ist das Datenmodell, das hinter den einzelnen komplexeren Komponenten steht. Im nächsten Hauptkapitel werden insbesondere Bäume als Datenstruktur, die auch in einer graphischen Komponente darstellbar ist, untersucht werden.

2.5 Swing und Steuerfäden

In der GUI-Programmierung kommt es oft vor, daß Dinge nebenläufig verlaufen sollen. Hierzu kennen wir das Prinzip von Steuerfäden. Und tatsächlich scheinen viele Dinge in unseren GUIs bisher schon nebenläufig zu funktionieren. Die verschiedenen Objekte zur Ereignisbehandlung fragen nebenläufig ab, ob ein Ereignis aufgetreten ist, auf das sie reagieren

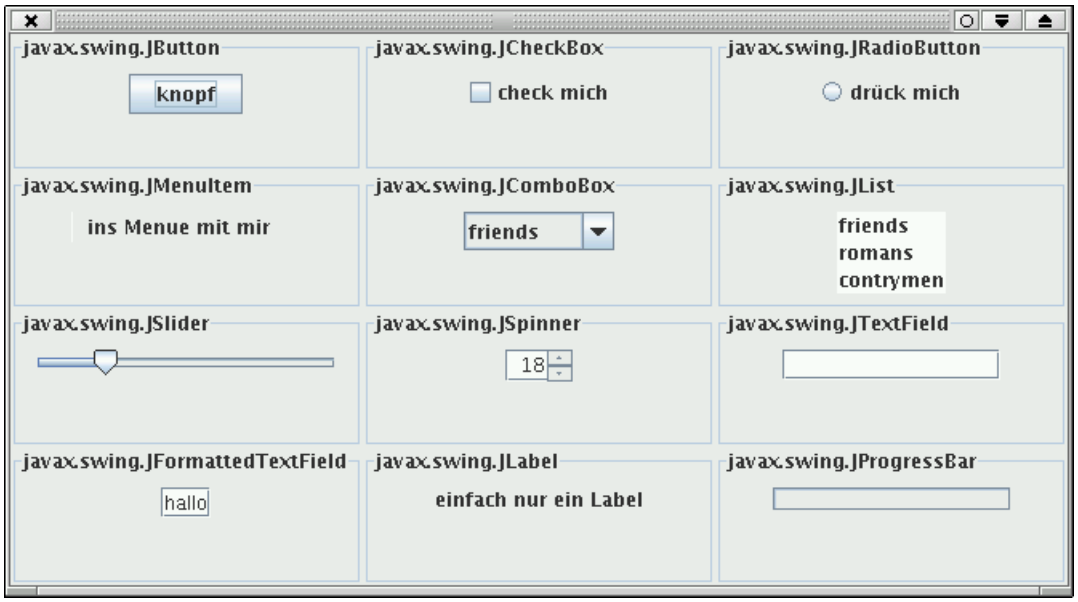


Abbildung 2.10: Überblick über einfache Komponenten.

sollen oder nicht. Solange man nur diese Nebenläufigkeit nutzt, die in Swing quasi für die Ereignisbehandlung fest eingebaut ist, so bekommt man keine problematischen Fälle. Im Java Handbuch von Sun[CWH00] gibt es daher eine erste große Empfehlung im bezug auf Steuerfäden:

> The first rule of using threads is this: avoid them when you can.

Oft braucht man aber Nebenläufigkeit. Wobei man zwei große Anwendungsfälle identifizieren kann, in denen in einem GUI Nebenläufigkeit wünschenswert ist.

- Bestimmte Operationen können sehr lange dauern und damit das GUI blockieren. Das Laden oder Bearbeiten von Multimediadateien kann unter Umständen sehr lange dauern. Während dieser Zeit, werden Fensterinhalte nicht aktualisiert.
- Für Animationen soll nach bestimmten Zeitintervallen die Graphik eines GUIs verändert und neu gezeichnet werden. Hierzu benötigt man so etwas wie ein `sleep` der Steuerfäden.

Beispiel:
Wir können uns zunächst davon überzeugen, daß die einzelnen Komponenten in einer Swinganwendung nicht in einzelnen Steuerfäden laufen. Hierzu schreiben wir ein Programm mit zwei Knopfkomponenten, von denen der eine eine Ereignisbehandlung hat, die nicht terminiert.

GuiHangs.java

```
1 package name.panitz.swing.threadTest;
2
3 import javax.swing.JFrame;
4 import javax.swing.JButton;
```

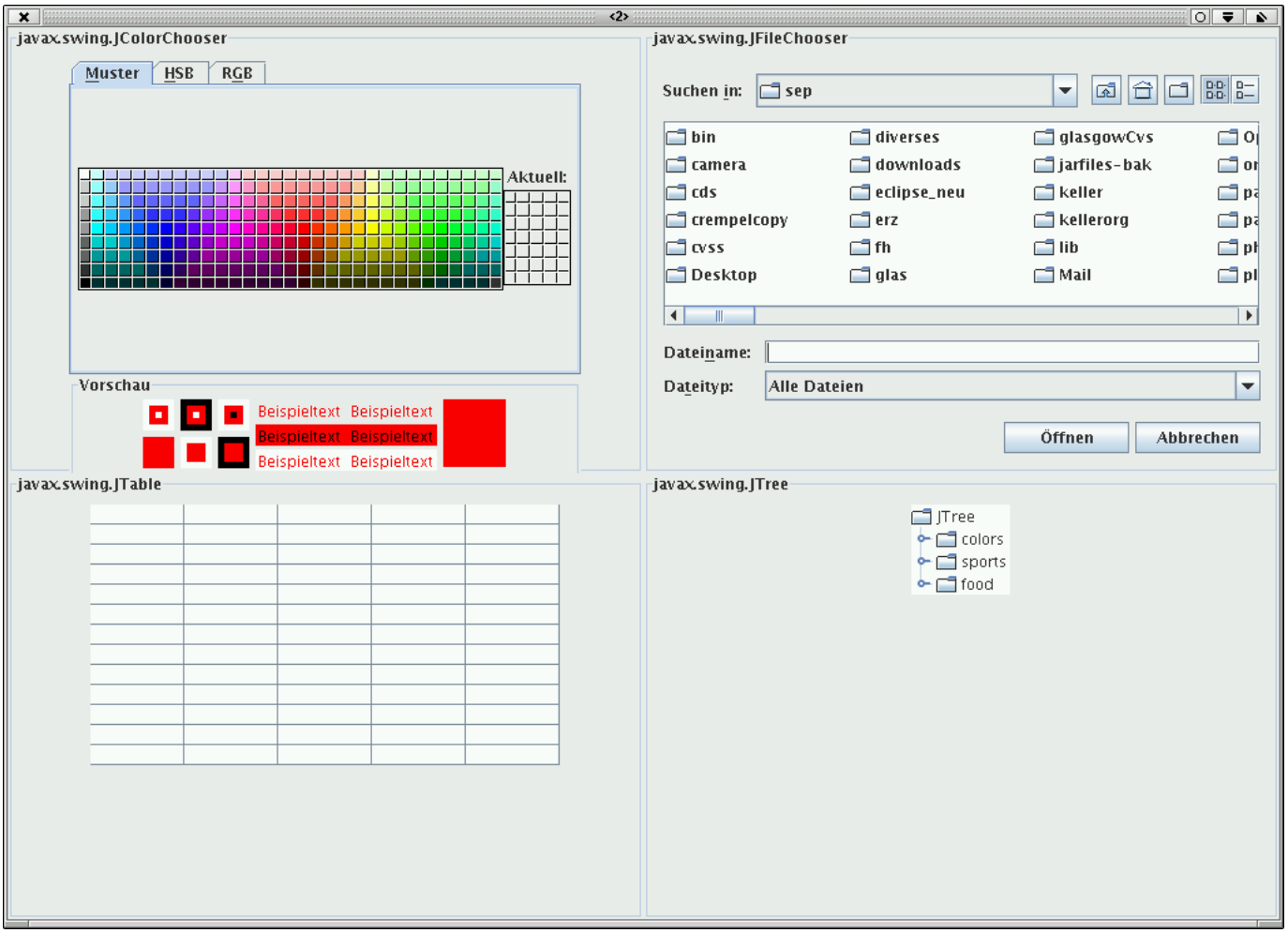


Abbildung 2.11: Überblick über komplexere Komponenten.

```
5  import java.awt.event.*;
6
7  public class GuiHangs {
8    public static void main(String [] _){
9      JFrame f1 = new JFrame("f1");
10     JFrame f2 = new JFrame("f2");
11     JButton b1 = new JButton("b1");
12     JButton b2 = new JButton("b2");
13
14     b1.addActionListener(new ActionListener(){
15       public void actionPerformed(ActionEvent _){
16         System.out.println("b1 action");
17       }
18     });
19
```

```
20      b2.addActionListener(new ActionListener(){
21        public void actionPerformed(ActionEvent _){
22          System.out.println("b2 action");
23          while (true){}
24        }
25      });
26
27      f1.getContentPane().add(b1);
28      f2.getContentPane().add(b2);
29      f1.pack();
30      f2.pack();
31      f1.setVisible(true);
32      f2.setVisible(true);
33    }
34  }
```

Starten wir dieses Programm, so werden zunächst zwei Fenster mit Knöpfen geöffnet. Sobald der Knopf `b2` das erste Mal gedrückt wird, hängt die komplette Anwendung. Es können keine Aktionen mehr durchgeführt werden und die Fenster werden auch nicht mehr neu gezeichnet.

Leider ist Swing so konzipiert, daß man nicht auf naive Weise eigene Objekte vom Typ `Thread` erzeugen und nebenläufig starten kann. Man spricht davon, daß Swing nicht *thread safe* ist. Insbesondere in den Zeichenmethoden der Komponenten, wie `paintComponent` darf man nicht eigene Steuerfäden benutzen. Das GUI könnte sich sonst mit den Steuerfäden zur Ereignisbehandlung in den Zeichenmethoden zur Darstellung der Komponenten auf unkontrollierte Weise verwurschteln.

Um einen sicheren Umgang mit Steuerfäden zu gewährleisten, stellt Swing ein paar Hilfsklassen zur Verfügung, die die wichtigen Anwendungsfälle für Steuerfäden in GUIs abdecken. Diese Hilfsklasse sorgen dafür, daß weitere Steuerfäden immer mit den Steuerfäden zur Ereignisbehandlung konform sind.

2.5.1 Timer in Swing

Um zeitlich immer wiederkehrende Ereignisse in GUIs zu prgrammieren gibt in Swing eine Hilfsklasse `Timer`. Objekte dieser Klasse können so instanziiert werden, daß sie in bestimmten Zeitabständen Ereignisse auslösen. Der `Timer` ist also so etwas wie ein Ereignisgenerator. Zusätzlich gibt man einem `Timer`-Objekt auch einen `ActionListener` mit, der spezifiziert, wie auf diese in zeitintervallen auftretenen Ereignisse reagiert werden soll.

Beispiel:
Folgende Klasse implementiert eine simple Uhr. In einem `JLabel` wird die aktuelle Zeit angegeben. Die Komponente wird einem `Timer` übergeben, der jede Sekunde eine neues Ereigniserzeugt. Diese Ereignisse sorgen dafür, daß die Zeit im Label aktualisiert wird.

Uhr.java

```
1  package name.panitz.swing.threads;
2
```

```
3  import javax.swing.*;
4  import java.util.Date;
5  import java.awt.event.*;
```

Die Klasse Uhr ist nicht nur ein `JPanel`, in dem ein `JLabel` benutzt wird, Datum und Uhrzeit anzuzeigen, sondern implementiert gleichfalls auch einen `ActionListener`.

Uhr.java

```
6  public class Uhr extends JPanel implements ActionListener{
```

Zunächst sehen wir das Datumsfeld für diese Komponente vor:

Uhr.java

```
7    JLabel l = new JLabel(new Date()+"");
```

Im Konstruktor erzeugen wir ein Objekt vom Typ `Timer`. Dieses Objekt soll alle Sekunde (alle 1000 Millisekunden) ein Ereignis erzeugen. Dem `Timer` wird das gerade im Konstruktor erzeute Objekt vom Typ `Uhr` übergeben, das, da es ja einen `ActionListener` implementiert, auf diese Ereignisse reagieren soll.

Uhr.java

```
8    public Uhr (){
9      new Timer(1000,this).start();
10     add(l);
11   }
```

Um die Schnittstelle `ActionListener` korrekt zu implementieren, muß die Methode `actionPerformed` implementiert werden. In dieser setzen wir jeweils Datum und Uhrzeit mit dem aktuellen Wert neu ins Label.

Uhr.java

```
12   public void actionPerformed(ActionEvent _){
13     l.setText(""+new Date());
14   }
```

Und natürlich sehen wir zum Testen eine kleine Hauptmethode vor, die die Uhr in einem Fensterrahmen anzeigt.

Uhr.java

```
15   public static void main(String [] _){
16     JFrame f = new JFrame();
17     f.getContentPane().add(new Uhr());
18     f.pack();
19     f.setVisible(true);
20   }
21 }
```

Das so erzeugte Fenster mit einer laufenden Uhr findet sich in Abbildung 2.12 dargestellt.

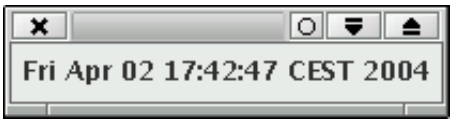


Abbildung 2.12: Eine einfache Digitaluhr.

Aufgabe 6 (2 Punkte) Implementieren Sie eine Analoguhr. Das Ziffernblatt dieser Uhr können Sie dabei mit den Methoden auf einem `Graphics`-Objekt zeichnen, oder Sie können versuchen eine Bilddatei zu laden und das Ziffernblatt als Bild bereitstellen. Als Anregung für ein Ziffernblatt können Sie sich das Bild des Ziffernblatts der handgefertigten Sekundenuhr (`http://www.panitz.name/prog2/examples/images/wiebking.jpg`) des Uhrmachers G.Wiebking herunterladen.

Animationen mit Timern

Mit dem Prinzip des Timers können wir jetzt auf einfache Weise Animationen realisieren. In einer Animation bewegt sich etwas. Dieses drücken wir durch eine entsprechende Schnittstelle aus:

Animation.java

```
1 package name.panitz.animation;
2
3 public interface Animation {
4   public void move();
5 }
```

Wir wollen einen besonderen `JPanel` realisieren, in dem sich etwas bewegen kann. Damit soll ein solcher `JPanel` auch eine Animation sein. Es bietet sich an, eine abstrakte Klasse zu schreiben, in der die Methode `move` noch nicht implementiert ist:

AnimatedJPanel.java

```
1 package name.panitz.animation;
2
3 import javax.swing.JPanel;
4 import javax.swing.Timer;
5 import java.awt.event.*;
6
7 public abstract class AnimatedJPanel
8   extends JPanel implements Animation {
```

Um zeitgesteuert das Szenario der Animation zu verändern, brauchen wir einen `Timer`.

AnimatedJPanel.java

```
9   Timer t;
```

Im Konstruktor wird dieser initialisiert. Als Ereignisbehandlung wird ein Ereignisbehandlungsobjekt erzeugt, das die Methode `move` aufruft, also dafür sorgt, daß die Szenerie sich weiterbewegt und das dafür sorgt, daß die Szenerie neu gezeichnet wird. Wir starten diesen Timer gleich.

AnimatedJPanel.java

```
10   public AnimatedJPanel(){
11     super(true);
12     t = new Timer(29,new ActionListener(){
13       public void actionPerformed(ActionEvent _){
14         move();
15         repaint();
16       }
17     });
18     t.start();
19   }
20 }
```

Jetzt können wir durch implementieren der Methode `move` und Überschreiben der Methode `paintComponent` beliebige Animationen erzeugen. Als erstes schreiben wir eine Klasse in der ein Kreis sich auf und ab bewegt:

BouncingBall.java

```
1  package name.panitz.animation;
2
3  import java.awt.Graphics;
4  import java.awt.Dimension;
5  import java.awt.Color;
6  import javax.swing.JFrame;
7
8  public class BouncingBall extends AnimatedJPanel {
```

Die Größe des Kreises und des Spielfeldes setzen wir in Konstanten fest:

BouncingBall.java

```
9    final int width = 100;
10   final int height = 200;
11   final int ballSize = 20;
```

Der Ball soll sich entlang der y-Achse bewegen, und zwar pro Bild um 4 Pixel:

BouncingBall.java

```
12   int yDir = 4;
```

Anfangs soll der Ball auf der Hälfte der x-Achse liegen und ganz oben im Bild liegen:

BouncingBall.java

```
13   int ballX = width/2-ballSize/2;
14   int ballY = 0;
```

Wir bewegen den Ball. Wenn er oben oder unten am Spielfeldrand anstößt, so ändert er seine Richtung:

BouncingBall.java

```
15   public void move(){
16     if (ballY>height-ballSize || ballY<0) yDir=-yDir;
17     ballY=ballY+yDir;
18   }
```

Zum Zeichnen, wird ein roter Hintergrund gezeichnet und der Kreis an seiner aktuellen Position.

BouncingBall.java

```
19   public void paintComponent(Graphics g){
20     g.setColor(Color.RED);
21     g.fillRect(0,0,width,height);
22     g.setColor(Color.YELLOW);
23     g.fillOval(ballX,ballY,ballSize,ballSize);
24   }
```

Unsere Größe wird verwendet als bevorzugte Größe der Komponente:

BouncingBall.java

```
25   public Dimension getPreferredSize(){
26     return new Dimension(width,height);
27   }
```

Und schließlich folgt eine kleine Hauptmethode zum Starten der Animation.

BouncingBall.java

```
28   public static void main(String [] _){
29     JFrame f = new JFrame("");
30     f.getContentPane().add(new BouncingBall());
31     f.pack();
32     f.setVisible(true);
33   }
34 }
```

Aufgabe 7 Schreiben Sie eine Animation, in der zwei Kreise sich bewegen. Die Kreise sollen an den Rändern der Spielfläche abprallen und sie sollen ihre Richtung ändern, wenn sie sich berühren.

Aufgabe 8 Schreiben Sie eine Animation, deren Verhalten über Mausklicks beeinflussen können.

2.5.2 SwingWorker für Steuerfäden in der Ereignisbehandlung

Sun bietet eine Beispielimplementierung einer Hilfsklasse an, die es erlaubt, Code in einen eigenen Steuerfaden zu verpacken. Die Klasse `SwingWorker`. `SwingWorker` ist nicht Bestandteil

des offiziellen Swing APIs, sondern lediglich im Tutorial angegeben. Im Anhang des Skripts ist die Klasse `SwingWorker` abgedruckt.

Die Klasse `SwingWorker` ist dazu gedacht, Code innerhalb einer Ereignisbehandlungsmethode, dessen Ausführung lange dauern kann, in einen Steuerfaden zu kapseln. Statt den Code direkt in der Ereignisbehandlungsmethode zu schreiben:

```
1  public void actionPerformed(ActionEvent e) {
2      ...
3      //...Code, der sehr lange dauert ...
4      ...
5  }
```

Ist er in die Methode `construct` einer `SwingWorker`-Instanz zu kapseln:

```
1  public void actionPerformed(ActionEvent e) {
2      ...
3      final SwingWorker worker = new SwingWorker() {
4          public Object construct() {
5              //...Code, der sehr lange dauert, ist jetzt hier...
6              return someValue;
7          }
8      };
9      worker.start();
10     ...
11 }
12
13
```

Beispiel:
Jetzt können wir unser Eingangsbeispiel mit der `SwingWorker`-Klasse so umschreiben, daß es nicht mehr zum Hängen des gesamten Programms kommt.

GuiHangsNoLonger.java

```
1  package name.panitz.swing.threadTest;
2
3  import javax.swing.JFrame;
4  import javax.swing.JButton;
5  import java.awt.event.*;
6
7  public class GuiHangsNoLonger {
8    public static void main(String [] _){
9      JFrame f1 = new JFrame("f1");
10     JFrame f2 = new JFrame("f2");
11     JButton b1 = new JButton("b1");
12     JButton b2 = new JButton("b2");
13
14     b1.addActionListener(new ActionListener(){
15       public void actionPerformed(ActionEvent _){
16         System.out.println("b1 action");
```

```
17        }
18      });
19
20      b2.addActionListener(new ActionListener(){
21        public void actionPerformed(ActionEvent _){
22          final SwingWorker worker = new SwingWorker() {
23            public Object construct() {
24              System.out.println("b2 action");
25              int x = 1; while (true){if (x==0) break;}
26              return null;
27            }
28          };
29          worker.start();
30        }
31      });
32
33      f1.getContentPane().add(b1);
34      f2.getContentPane().add(b2);
35      f1.pack();
36      f2.pack();
37      f1.setVisible(true);
38      f2.setVisible(true);
39    }
40  }
```

Nach Drücken des Knopfes `b2` bleibt das Gui weiterhin voll funktionsfähig. Allerdings stellt man fest, indem man sich z.B. mit `top` die laufenden Prozesse in einem Unixsystem anschaut, daß jedes Drücken des Knopfes einen neuen Unterprozess von Java erzeugt.

SwingWorkerbeispiel

Im folgenden ist ein weiteres Beispiel angegeben, in dem die Klasse `SwingWorker` benutzt wird. Diesmal soll eine Bilddatei geladen werden. Hierzu schreiben wir eine Klasse, die von `JLabel` ableitet. Das Label soll eine vorgegebene feste Größe haben. Dem Konstruktor wird eine Bilddatei übergeben.

ImageIcon.java

```
1  package name.panitz.gui;
2
3  import java.io.File;
4  import java.net.MalformedURLException;
5
6  import java.awt.*;
7  import java.awt.event.*;
8  import javax.swing.*;
9
10 public class ImageIcon extends JLabel {
11   private int height = 60;
12   private int width = 80;
```

```
13    final Toolkit toolkit = Toolkit.getDefaultToolkit();
14    private Image image;
15
16    public ImageIcon(String fileName){this(new File(fileName));}
17    public ImageIcon(final File file){
18      setImage(file);
19    }
```

Zum Setzen des Bildes, wird die Bilddatei gelesen, um es dann auf die vorgegebene Größe zu skalieren:

ImageIcon.java

```
20    public void setImage(File jpgFile){
21      try{
22        image=toolkit.getImage(jpgFile.toURL());
23        final MediaTracker tracker = new MediaTracker(this);
24        tracker.addImage(image, 0,width,height);
25        try {tracker.waitForAll();}
26        catch (Exception e) {e.printStackTrace();}
27        fitImage();
28        setIcon(new javax.swing.ImageIcon(image));
29      }catch (MalformedURLException e){}
30    }
```

Zum Skalieren ist zunächst die Originalgröße zu lesen, die skalierte Größe zu berechnen und schließlich das Bild auf die skalierte Größe zu setzen:

ImageIcon.java

```
31    private void fitImage(){
32      final int w = image.getWidth(this);
33      final int h = image.getHeight(this);
34
35      final int w1 = width;        final int h1 = h*width/w;
36      final int w2 = w*height/h;   final int h2 = height;
37      final int w3 = w1<w2?w1:w2;  final int h3 = h1<h2?h1:h2;
38
39      image=image.getScaledInstance
40                    (w3<w?w3:w,h3<h?h3:h,Image.SCALE_FAST);
41    }
42  }
```

Damit haben wir eine Komponente, die Bilder beliebiger Größe aus Dateien in einer vorgegebenen Größe anzeigen kann. Diese Komponente können wir benutzen, um eine kleine Anwendung zu starten. Die Anwendung soll zwei Fenster Öffnen: im erstem Fenster sollen alle über die Kommandozeile mitgegebenen Biler als Label angezeigt werden, im zweitem Fenster soll eine Uhr laufen:

UseImageIcon.java

```
1   package name.panitz.gui;
2
```

```
3  import name.panitz.swing.threads.Uhr;
4
5  import java.io.File;
6
7  import java.awt.*;
8  import java.awt.event.*;
9  import javax.swing.*;
10
11 class UseImageIcon {
12   public static void main(String [] args){
13     JFrame f= new JFrame();
14     final JPanel p = new JPanel();
15     f.getContentPane().add(p);
16     final int c = 10;
17     final int n = args.length;
18
19     p.setLayout(new GridLayout(n/c+((n%c)==0?0:1),c));
20     for (final String arg:args)p.add(new ImageIcon(arg));
21
22     f.pack();
23     f.setVisible(true);
24
25     JFrame f2 = new JFrame();
26     f2.getContentPane().add(new Uhr());
27     f2.pack();
28     f2.setVisible(true);
29   }
30 }
```

Wenn wir dieses Programm mit allen unseren Urlaubsbildern starten, dann stellen wir fest, daß es sehr lange dauert, bis sich die beiden Fenster öffnen. Erst wenn alle Bilder geladen und skaliert sind, werden die beiden Fenster geöffnet. Auch dieses Programm läßt sich mit Hilfe der Klasse `SwingWorker` verbessern. Das Erzeugen der Objekte vom Typ `ImageIcon` wird hierzu in einem `ActionListener` durchgeführt. Es wird ein `Timer` benutzt, der genau ei Ereignis erzeugt, so daß der Ereignisbehandler genau einmal auf diesen Timer reagieren kann.

BetterUseOfImageIcon.java

```
1  package name.panitz.gui;
2
3  import name.panitz.swing.threads.Uhr;
4  import name.panitz.swing.threadTest.SwingWorker;
5
6  import java.io.File;
7
8  import java.awt.*;
9  import java.awt.event.*;
10 import javax.swing.*;
11
12 public class BetterUseOfImageIcon{
```

```
13    public static void main(String [] args){
14      final JFrame f= new JFrame();
15      final JPanel p = new JPanel();
16      f.getContentPane().add(p);
17      final int c = 10;
18      final int n = args.length;
19
20      p.setLayout(new GridLayout(n/c+((n%c)==0?0:1),c));
21      for (final String arg:args){
22        ActionListener listen = new ActionListener(){
23          public void actionPerformed(ActionEvent e) {
24           final SwingWorker worker = new SwingWorker() {
25             public Object construct() {
26                p.add(new ImageIcon(arg)); f.pack();
27                f.repaint();
28                return null;
29             }
30           };
31           worker.start();
32          }
33        };
34
35        Timer t = new Timer(10,listen);
36        t.setRepeats(false);
37
38        t.start();
39      }
40      f.pack();
41      f.setVisible(true);
42
43      JFrame f2 = new JFrame();
44      f2.getContentPane().add(new Uhr());
45      f2.pack();
46      f2.setVisible(true);
47    }
48  }
```

Und tatsächlich: starten wir dieses Programm jetzt mit vielen Bilddateien, so öffnen sich gleich beide Bilder und die Uhr läuft durch. Nach und nach erscheinen die Bilder in ihrem Fenster, jedes solbald es erfolgreich geladen und skaliert wurde.

Kapitel 3

Abstrakte Datentypen

3.1 Bäume

Bäume sind ein gängiges Konzept um hierarchische Strukturen zu modellieren. Sie sind bekannt aus jeder Art von Baumdiagramme, wie Stammbäumen oder Firmenhierarchien. In der Informatik sind Bäume allgegenwärtig. Fast alle komplexen Daten stellen auf die eine oder andere Art einen Baum dar. Beispiele für Bäume sind mannigfach:

- **Dateisystem**: Ein gängiges Dateisystem, wie es aus Unix, MacOS und Windows bekannt ist, stellt eine Baumstruktur dar. Es gibt einen ausgezeichneten Wurzelordner, von dem aus zu jeder Datei einen Pfad existiert.
- **Klassenhierarchie**: Die Klassen in Java stellen mit ihrer Ableitungsrelation eine Baumstruktur dar. Die Klasse `Object` ist die Wurzel dieses Baumes, von der aus alle anderen Klassen über einen Pfad entlang der Ableitungsrelation erreicht werden können.
- **XML**: Die logische Struktur eines XML-Dokuments ist ein Baum. Die Kinder eines Elements sind jeweils die Elemente, die durch das Element eigeschlossen sind.
- **Parserergebnisse**: Ein Parser, der gemäß einer Grammatik prüft, ob ein bestimmter Satz zu einer Sprache gehört, erzeugt im Erfolgsfall eine Baumstruktur. Im nächsten Kapitel werden wir dieses im Detail kennenlernen.
- **Listen**: Auch Listen sind Bäume, allerdings eine besondere Art, in denen jeder Knoten nur maximal ein Kind hat.
- **Berechnungsbäume**: Zur statischen Analyse von Programmen, stellt man Bäume auf, in denen die Alternativen eines bedingten Ausdrucks Verzweigungen im Baum darstellen.
- **Tableaukalkül**: Der Tableaukalkül ist ein Verfahren zum Beweis logischer Formeln. Die dabei verwendeten Tableaux sind Bäume.
- **Spielbäume**: Alle möglichen Spielverläufe eines Spiels können als Baum dargestellt werden. Die Kanten entsprechen dabei einem Spielzug.

- **Prozesse**: Auch die Prozesse eines Betriebssystems stellen eine Baumstruktur dar. Die Kinder eines Prozesses sind genau die Prozesse, die von ihm gestartet wurden.

Wie man sieht, lohnt es sich, sich intensiv mit Bäumen vertraut zu machen, und man kann davon ausgehen, was immer in der Zukunft neues in der Informatik entwickelt werden wird, Bäume werden darin in irgendeiner Weise eine Rolle spielen.

Ein Baum besteht aus einer Menge von Knoten die durch gerichtete Kanten verbunden sind. Die Kanten sind eine Relation auf den Knoten des Baumes. Die Kanten verbinden jeweils einen Elternknoten mit einem Kinderknoten. Ein Baum hat einen eindeutigen Wurzelknoten. Dieses ist der einzige Knoten, der keinen Elternknoten hat, d.h. es gibt keine Kante, die zu diesen Knoten führt. Knoten, die keinen Kinderknoten haben, d.h. von denen keine Kante ausgeht, heißen Blätter.

Die Kinder eines Knotens sind geordnet, d.h. sie stehen in einer definierten Reihenfolge.

Eine Folge von Kanten, in der der Endknoten einer Vorgängerkante der Ausgangsknoten der nächsten Kanten ist, heißt Pfad.

In einem Baum darf es keine Zyklus geben, das heißt, es darf keinen Pfad geben, auf dem ein Knoten zweimal liegt.

Knoten können in Bäumen markiert sein, z.B. einen Namen haben. Mitunter können auch Kanten eine Markierung tragen.

Bäume, in denen Jeder Knoten maximal zwei Kinderknoten hat, nennt man Binärbäume. Bäume, in denen jeder Knoten maximal einen Kinderknoten hat, heißen Listen.

3.1.1 Formale Spezifikation

Allgemeine Bäume

Wir wollen in diesem Abschnitt Bäume als einen abstrakten Datentypen spezifizieren.

Konstruktoren Abstrakte Datentypen lassen sich durch ihre Konstruktoren spezifizieren. Die Konstruktoren geben an, wie Daten des entsprechenden Typs konstruiert werden können.

In dem Fall von Bäumen bedarf es nach den obigen Überlegungen nur eines Konstruktors, den für Knoten. Er konstruiert aus der Liste der Kinderbäume einen neuen Baumknoten. Als zusätzliches Argument bekommt er ein Objekt, das eine Markierung für den Baumknoten darstellt. Den Typ dieses Markierungsobjektes können wir erst einmal offen lassen. Hier darf ein beliebiger Typ gewählt werden.

Wir benutzen in der Spezifikation eine mathematische Notation der Typen von Konstruktoren.[1] Dem Namen des Konstruktors folgt dabei mit einem Doppelpunkt abgetrennt der Typ. Der Ergebnistyp wird von den Parametertypen mit einem Pfeil getrennt.

Somit läßt sich der Typ des Konstruktors für Bäume wie folgt spezifizieren:

- Node: (α,**List<Tree<**α**>>**) $\rightarrow$**Tree<**α**>**

[1] Entgegen der Notation in Java, in der der Rückgabetyp kurioser Weise vor den Namen der Methode geschrieben wird.

Selektoren Die Selektoren können wieder auf die einzelnen Bestandteile der Konstruktion zurückgreifen. Der Konstruktor **Node** hat zwei Parameter. Für Bäume werden zwei Selektoren spezifiziert, die jeweils einen dieser beiden Parameter wieder aus dem Baum selektieren.

- theChildren: **Tree<**α**>**→**List<Tree<**α**>>**
- mark: **Tree<**α**>**→α

Der funktionale Zusammenhang von den Selektoren und Konstruktoren läßt sich durch folgende Gleichungen spezifizieren:

$$mark(Node(x,xs)) = x$$
$$theChildren(Node(x,xs)) = xs$$

Testmethoden Da wir nur einen Konstruktor vorgesehen haben, brauchen wir keine Testmethode, die unterscheidet, mit welchem Konstruktor ein Baum konstruiert wurde.

Die Unterscheidung, ob es sich um ein Blatt oder um einen Knoten mit Kindern handelt, läßt sich über die Abfrage, ob die Liste der Kinderknoten leer ist, erfahren. Wir können eine entsprechende Funktion spezifizieren:

$$isLeaf(Node(x,xs)) = isEmpty(xs)$$

3.1.2 Modellierung

Wir haben wieder verschiedene Möglichkeiten eine Baumstruktur mit Klassen zu modellieren.

mit einer Klasse

Die einfachste Modellierung ist mit einer Klasse. Diese Klasse stellt einen Knoten dar. Sie enthält ein Feld für die Knotenmarkierung und ein Feld für die Kinder des Knotens. Nach unserer Spezifikation sind die Kinder eine Liste weiterer Knoten. Ein entsprechendes UML Diagramm befindet sich in Abbildung 3.1

Abbildung 3.1: Modellierung von Bäumen mit einer Klasse.

Mit einer Klassenhierarchie

Ebenso, wie wir es bei Listen vorgeschlagen haben, können wir auch wieder eine Modellierung mit mehreren Klassen vornehmen. Hierzu beschreibt eine Schnittstelle die allgemeine Funktionalität von Bäumen. Zwei Unterklassen, jeweils eine für innere Knoten und eine für Blätter implementieren diese Schnittstelle. Ein entsprechendes UML Diagramm befindet sich in Abbildung 3.2

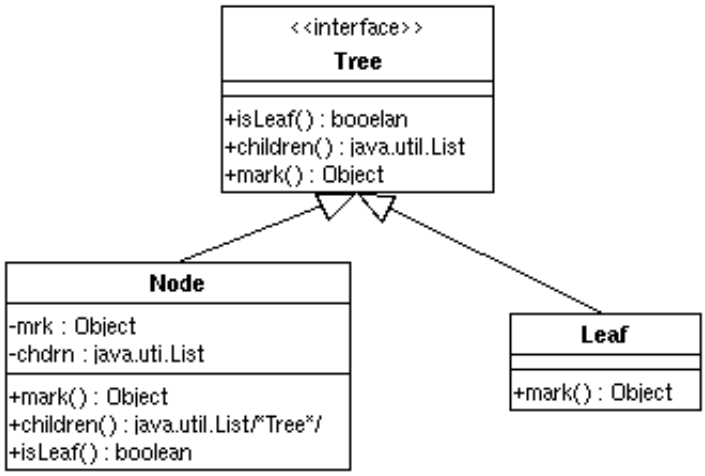


Abbildung 3.2: Modellierung von Bäumen mittels einer Klassenhierarchie.

Welche dieser beiden Modellierungen jeweils vorzuziehen ist, hängt sehr davon ab, für was die Bäume in der Gesammtanwendung benutzt werden sollen. Eine grobe Faustregel für abstrakte Datentypen ist:

- Wenn zu erwarten ist, daß im Laufe der Softwareentwicklung noch weitere Konstruktoren zum abstrakten Datentyp hinzudefiniert werden, dann ist eine Modellierung als Hierarchie vorteilhaft. Der neue Konstruktor führt zu einer neuen Klasse, in der lokale Version der für die implementierte Schnittstelle benötigten Methoden implementiert werden.
- Wenn zu erwarten ist, daß spezialisierte Versionen des abstrakten Datentyps benötigt werden, so ist eine Modellierung als eine Klasse vorteilhaft, weil dann für dieser einen Klassen eine Unterklasse implementiert werden kann.

Wir werden im Folgenden für unsere Bäume die Modellierung mit einer Klasse bevorzugen und anschließend spezialisierte Baumtypen von dieser einen Klasse ableiten.

3.1.3 Implementierung

Für unsere Umsetzung von Bäumen benutzen wir die Modellierung in einer Klasse. Wir schreiben die Klasse `Tree` entsprechend unserer Spezifikation. Zusätzlich sehen wir zwei weitere Konstruktoren vor, die jeweils einen Standardwert für die beiden Felder setzen:

- `Tree(List children)`: als Knotenmarkierung wird `null` genommen.
- `Tree(a mark)`: für die Kinder wird eine leere Liste erzeugt.

Wir erhalten folgende einfache Klasse:

Tree.java
```
1  package name.panitz.data.tree;
2
3  import java.util.List;
4  import java.util.Iterator;
5  import java.util.Enumeration;
6  import java.util.ArrayList;
7  import javax.swing.tree.TreeNode;
8
9  public class Tree<a>  implements TreeNode{
10
11   private a mrk;
12   private List<Tree<a>> chldrn;
```

Die beiden Felder werden durch den Konstruktor gesetzt:

```
1    public Tree(a mark,List<Tree<a>> children){
2      this.mrk=mark;
3      this.chldrn=children;
4    }
```

Zwei weitere Konstruktoren setzen beziehen sich auf diesen Konstruktor:

Tree.java
```
5    public Tree(List<Tree<a>> children){
6      this(null,children);
7    }
8
9    public Tree(a mark){
10     this(mark,new ArrayList<Tree<a>>());
11   }
12
13
14   public List<Tree<a>> theChildren(){return chldrn;}
15   public a mark(){return mrk;}
16
17   public boolean isLeaf(){return theChildren().isEmpty();}
18
```

Aufgabe 9 Erzeugen Sie ein Baumobjekt, das einen Stammbaum Ihrer Familie darstellt. Die Knoten und Blätter sind mit Namen markiert. Die Wurzel ist mit Ihren Namen markiert. Kinderknoten sind mit den leiblichen Eltern der Person eines Knotens markiert.

3.2 Algorithmen auf Bäumen

Die Selektormethoden für Bäume und Listen können wir jetzt gemeinsam nutzen, um einfache Algorithmen auf Bäumen umzusetzen.

3.2.1 Knotenanzahl

Entsprechend der Methode `length`, wie wir sie für Listen geschrieben haben, lassen sich die Knoten eines Baumes zählen.

Tree.java

```
19 public int count(){
20   int result = 1; // Startwert 1 für diesen Knoten
21
22   //für jedes Kind
23   for (Tree<a> child: theChildren())
24     //addiere die Knotenanzahl des Kindes zum Ergebnis
25     result=result+child.count();
26
27   return result;
28 }
```

Diese Methode ist rekursiv geschrieben. Der terminierende Fall versteckt sich dieses mal in der `for-Schleife`. Wenn die Liste der Kinder leer ist, so wird die `for`-Schleife nicht durchlaufen. Nur im Rumpf der Schleife steht ein rekursiver Aufruf. Daher gibt es keine Anweisung der Form:
`if (theChildren().isEmpty()) return 1`

3.2.2 toString

Unter Benutzung der Methode `toString` aus der Listenimplementierung, läßt sich relativ einfach eine `toString`-Methode für unsere Baumimplementierung umsetzen.

```
1  public String toString(){
2    //Nimm erst die Markierung an diesem Knoten
3    String result = mark().toString();
4
5    //Gibt es keine Kinder, dann bist du fertig
6    if (isLeaf()) return result;
7
8    //ansonsten nimm implizit durch Operator + das
9    //toString() der Kinder
10   return result+theChildren();
11 }
```

Auch wenn wir es nicht sehen, ist diese Methode rekursiv. Der Ausdruck `result+theChildren()` führt dazu, daß auf einer Liste von Bäumen die Methode

`toString()` ausgeführt wird. Dieses führt wiederum zur Ausführung der `toString`-Methode auf jedes Element und dieses sind wiederum Bäume. Der Aufruf von `toString` der Klasse `Tree` führt über den Umweg der Methode `toString` der Schnittstelle `List` abermals zur Ausführung der `ToString`-Methode aus `Tree`. Man spricht dabei auch von *verschränkter* Rekursion.

3.2.3 Linearisieren

Die Methode `flatten` erzeugt eine Liste, die die Markierung jedes Baumknotens genau einmal enthält. Aus einem Baum wird also eine Liste erzeugt. Die logische Struktur des Baumes wird flachgeklopft.

Tree.java

```
12 public List<a> flatten(){
13   //leere Liste für das Ergebnis
14   List<a> result = new ArrayList<a>();
15
16   //füge die Markierung dieses Knotens zur
17   //Ergebnisliste hinzu
18   result.add(mark());
19
20   //für jedes Kind
21   for (Tree<a> child : theChildren())
22     //erzeuge dessen Knotenliste und füge alle deren
23     //Elemente zum Ergebnis hinzu
24     result.addAll(child.flatten());
25
26   return result;
27 }
```

Als Ergebnisliste wird ein Objekt der Klasse `ArrayList` angelegt. Diesem Objekt wird die Markierung des aktuellen Knotens zugefügt und dann die Linearisierung jedes der Kinder. Die Methode `addAll` aus der Schnittstelle `List` sorgt dafür, daß alle Elemente der Parameterliste hinzugefügt werden.

Aufgabe 10 Testen Sie die in diesem Abschnitt entwickelten Methoden auf Bäumen mit dem in der letzten Aufgabe erzeugten Baumobjekt.

Aufgabe 11 Punkteaufgabe (4 Punkte):
Für diese Aufgabe gibt es maximal 4 auf die Klausur anzurechnende Punkte. Abgabetermin: wird noch bekanntgegeben.

Schreiben Sie die folgenden weiteren Methoden für die Klasse `Tree`. Schreiben Sie Tests für diese Methoden.

a) `List leaves()`: erzeugt eine Liste der Markierungen an den Blättern des Baumes.

b) `String show()`: erzeugt eine textuelle Darstellung des Baumes. Jeder Knoten soll eine eigene Zeile haben. Kinderknoten sollen gegenüber einem Elternknoten eingerückt sein.

Beispiel:

```
william
  charles
    elizabeth
    phillip
  diana
    spencer
```

Tipp: Benutzen Sie eine Hilfsmethode
`String showAux(String prefix)`,
die den String `prefix` vor den erzeugten Zeichenketten anhängt.

c) `boolean contains(a o)`: ist wahr, wenn eine Knotenmarkierung gleich dem Objekt `o` ist.

d) `int maxDepth()`: gibt die Länge des längsten Pfades von der Wurzel zu einem Blatt an.

e) `List<a> getPath(a o)`: gibt die Markierungen auf dem Pfad von der Wurzel zu dem Knoten, der mit dem Objekt `o` markiert ist, zurück. Ergebnis ist eine leere Liste, wenn ein Objekt gleich `o` nicht existiert.

f) `java.util.Iterator<a> iterator()`: erzeugt ein Iterator über alle Knotenmarkierungen des Baumes.
Tipp: Benutzen Sie bei der Umsetzung die Methode `flatten`.

g) `public boolean equals(Object other)`: soll genau dann wahr sein, wenn `other` ein Baum ist, der die gleiche Struktur und die gleichen Markierungen hat.

h) `<b>boolean sameStructure(Tree<a> other)`: soll genau dann wahr sein, wenn der Baum `other` die gleiche Struktur hat. Die Markierungen der Knoten können hingegen unterschiedlich sein.

3.2.4 Eltern und Geschwister

Wir können in unserer Implementierung von einem Knoten nicht mehr seinen Eltern- und seine Geschwisterknoten angeben. Wir können nur einen Baum von der Wurzel an zu den Blätter durchlaufen, aber nicht umgekehrt, von einem beliebiegen Knoten zurück zur Wurzel laufen. Dieses liegt daran, daß wir keinerlei Verbindung von einem Kind zu seinem Elternknoten haben. Wenn wir eine solche haben wollen, müssen wir hierfür ein Feld vorsehen. Wir ergänzen die Klasse `Tree` somit um ein weiteres Feld.

Tree.java

```
28
29  public Tree<a> parent=null;
30
```

Für einen neuen Baumknoten, der neu erzeugt wird, ist das Feld auf den Elternknoten zunächst auf `null` gesetzt. Erst wenn ein Baumknoten als Kind in einen Baum eingehängt wird, ist sein Feld für den Elternknoten auf den entsprechenden Baum zu setzen:

Tree.java

```
31   public Tree(a mark,List<Tree<a>> children){
32     //setze die entsprechenden Felder des Knotens
33     this.mrk=mark;
34     this.chldrn=children;
35
36     //es gibt noch keinen Elternknoten
37     this.parent=null;
38
39     //setze den neuen Knoten als Elternknoten der Kinder
40     respectParents();
41   }
42
43  public void respectParents(){
44    //für jedes Kind
45    for (Tree<a> child:theChildren()){
46     // ist dieser Knoten jetzt der Elternknoten
47     child.parent=this;
48    }
49  }
```

Jetzt haben wir allerdings eine neue Situation. Bisher konnten wir Baumknoten mehrfach in einen Baum einhängen oder einen Knoten in mehrere Mäume einhängen. Das geht jetzt nicht mehr. Zwei Bäume können sich nicht mehr einen Knoten teilen, denn der Knoten kann ja nur einen Elternknoten speichern. Wir modifizieren jedesmal das Feld `parent`, wenn wir einen Knoten als Kinderknoten einen neuen Baumknoten übergeben.

Aufgabe 12 Schreiben Sie eine Methode `List<Tree<a>> siblings()`, die die Geschwister eines Knotens als Liste ausgibt. In dieser Liste der Geschwister soll der Knoten selbst nicht auftauchen.

3.2.5 Modifizierende Methoden

Bisher haben wir eine rein funktionale Umsetzung von Bäumen erarbeitet. Einmal erzeugte Bäume sind unveränderbar. Die einzige Ausnahme war hierbei das Feld `parent`, das geändert wurde, sobald ein Knoten in einen Baum eingehängt wurde.

Die Felder der Klasse `Tree` sind privat. Es gibt keine Methoden, die sie verändern. Ebenso war auch unsere Listenimplementierung gehalten. Die Java Standardsammlumgsklassen haben im Gegensatz dazu Methoden, die ein Listenobjekt verändern.

Wir können für unsere Klasse `Tree` auch solche modifizierenden Methoden implementieren. Solche Methoden fügen typischer Weise einen neuen Knoten in einen Baum ein oder löschen Knoten. Da es modifizierende Methoden sind, empfiehlt es sich, sie als `void`-Methoden zu schreiben.

addChild

Wir schreiben eine Methode, die an einen Knoten als letztes Kind einen neuen Knoten einfügt:

Tree.java

```
50 public void addChild(Tree<a> newChild){
51   List<Tree<a>> chs = theChildren();
52   newChild.parent=this;
53   chs.add(newChild);
54 }
```

Es wird direkt auf dem Feld mit der Liste der Kinder operiert und die modifizierende methode `add` aus der Schnitstelle `List` benutzt.

addLeaf

Eine weitere Methode fügt dem Baum ein neues Blatt ein:

Tree.java

```
55 public void addLeaf(a o){addChild(new Tree<a>(o));}
```

deleteChild

Folgende Methode löscht das n-te Kind aus einem Knoten:

Tree.java

```
56 public void deleteChild(int n){
57    theChildren().remove(n);
58 }
```

In dieser Implementierung wird die modifizierte Liste durch die Methode `theChildren()` ermittelt, anstatt auf das Feld `chldrn` direkt zuzugreifen.

setMark

Das Feld `mrk` hat das Attribut `private`. Wollen wir seinen Wert aus einem anderen Kontext ändern, so brauchen wir eine Methode, die dieses macht. Wir können eine typische `Set`-Methode schreiben:

Tree.java

```
59 public void setMark(a mark){mrk=mark;}
```

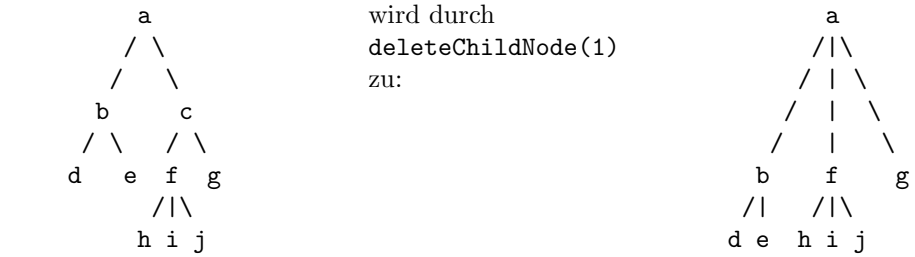
Beispiel: Baumerzeugung mit modifizierenden Methoden

SkriptTree.java

```
1  package name.panitz.data.tree.example;
2
3  import  name.panitz.data.tree.Tree;
4  public class SkriptTree{
5    public static Tree<String> getSkript(){
6      Tree<String> skript = new Tree<String>("Programmieren 2");
7
8      Tree<String> kap1=new Tree<String>("Frühe und späte Bindung");
9      Tree<String> kap2
10      = new Tree<String>
11         ("Graphische Benutzeroberflächen und Graphiken");
12     Tree<String> kap3
13      = new Tree<String>("Abstrakte Datentypen");
14
15     skript.addChild(kap1);
16     skript.addChild(kap2);
17     skript.addChild(kap3);
18
19     kap1.addLeaf("Wiederholung: Das Prinzip der späten Bindung");
20     kap1.addLeaf("Keine späte Bindung für Felder");
21     kap1.addLeaf("Keine Späte Bindung für überladene Methoden");
22
23     Tree<String> sec2_1
24      = new Tree<String>("Graphische Benutzeroberflächen");
25     Tree<String> sec2_2
26      = new Tree<String>("Exkurs: verschachtelte Klassen");
27     Tree<String> sec2_3
28      =new Tree<String>("Selbstdefinierte graphische Komponenten");
29     Tree<String> sec2_4
30      = new Tree<String>("Weitere Komponente und Ereignisse");
31     Tree<String> sec2_5
32      = new Tree<String>("Swing und Steuerfäden");
33
34     kap2.addChild(sec2_1);kap2.addChild(sec2_2);
35     kap2.addChild(sec2_3);kap2.addChild(sec2_4);
36     kap2.addChild(sec2_5);
37
38     Tree<String> sec3_1
39      = new Tree<String>("Bäume");
40     Tree<String> sec3_2
41      = new Tree<String>("Algorithmen auf Bäumen");
42     Tree<String> sec3_3
43      = new Tree<String>("Binärbäume");
44     Tree<String> sec3_4
45      = new Tree<String>("Binäre Suchbäume");
46     Tree<String> sec3_5
47      = new Tree<String>("XML-Dokumente als Bäume");
```

```
48      Tree<String> sec3_6
49       = new Tree<String>("Keller");
50
51      kap3.addChild(sec3_1);kap3.addChild(sec3_2);
52      kap3.addChild(sec3_3);kap3.addChild(sec3_4);
53      kap3.addChild(sec3_5);kap3.addChild(sec3_6);
54
55      sec2_1.addLeaf("Graphische Komponenten");
56      sec2_1.addLeaf("Ereignisbehandlung");
57      sec2_1.addLeaf("Setzen des Layouts");
58
59      sec2_2.addLeaf("Innere Klassen");
60      sec2_2.addLeaf("Anonyme Klassen");
61      sec2_2.addLeaf("Statisch verschachtelte Klassen");
62
63      sec2_3.addLeaf("Graphics Objekte");
64      sec2_3.addLeaf("Dimensionen");
65      sec2_3.addLeaf("Farben");
66      sec2_3.addLeaf("Fonts und ihre Metrik");
67      sec2_3.addLeaf("Erzeugen graphischer Dateien");
68      sec2_3.addLeaf("Graphics2D");
69
70      sec2_4.addLeaf("Mausereignisse");
71      sec2_4.addLeaf("Fenstereignisse");
72      sec2_4.addLeaf("Weitere Komponenten");
73
74      sec2_5.addLeaf("Timer in Swing");
75      sec2_5.addLeaf
76       ("SwingWorker für Steuerfäden in der Ereignisbehandlung");
77      return skript;
78    }
79  }
```

Aufgabe 13 Schreiben Sie eine modifizierende Methode
`void deleteChildNode(int n)`,
die den n-ten Kindknoten löscht, und stattdessen die Kinder des n-ten Kindknotens als neue Kinder mit einhängt.
Beispiel:



3.2.6 Bäume mit JTree graphisch darstellen

Wir haben uns in dieser Vorlesung ausgiebig mit Bäumen beschäftigt, sie aber bisher nicht graphisch sondern lediglich textuell dargestellt. In diesem Kapitel sollen zwei Möglichkeiten zur graphischen Darstellung von Bäumen in Java untersucht werden. Zunächst benutzen wir eine bereits in der `swing`-Bibliothek angebotene Komponente zur Baumdarstellung, anschließend definieren wir eine eigene neue Komponente zur Baumdarstellung.

Im Paket `javax.swing` gibt es eine Klasse, die es erlaubt beliebige Baumstrukturen graphisch darzustellen, die Klasse `JTree`. Diese Klasse übernimmt die graphische Darstellung eines Baumes. Der darzustellende Baum ist dieser Klasse zu übergeben. Hierzu hat sie einen Konstruktor, der einen Baum erhält: `JTree(TreeNode root)`.

Um entsprechend die Klasse `JTree` benutzen zu können, so daß sie uns einen Baum darstellt, müssen wir einen Baum, der die Schnittstelle `javax.swing.tree.TreeNode` implementiert, zur Verfügung stellen. Diese Schnittstelle hat 7 relativ naheliegende Methoden, die sich in ähnlicher Form fast alle in unserer Klasse `Tree` auch schon befinden:

```
1  package javax.swing.tree;
2  interface TreeNode{
3    Enumeration children();
4    int getChildCount();
5    TreeNode getParent();
6    boolean isLeaf();
7    TreeNode getChildAt(int childIndex);
8
9    boolean getAllowsChildren();
10   int getIndex(TreeNode node);
11 }
```

Wir können unsere Klasse `Tree` entsprechend modifizieren, damit sie diese Schnittstelle implementiert. Hierzu fügen wir Implementierungen der in der Schnittstelle verlangten Methoden der Klasse `Tree` hinzu und sorgen so dafür, daß unsere Bäume alle Eigenschaften haben, so daß die Klasse `JTree` sie graphisch darstellen können.

```
1  public class Tree<a>implements TreeNode{
```

Entscheidend ist natürlich die Methode `children`, die inhaltlich unserer Methode `theChildren` entspricht. Leider benutzt die Schnittstelle als Rückgabetyp eine weitere Klasse aus dem Paket `java.util`. Dieses ist die Schnittstelle `Enumeration`. Logisch ist sie fast funktionsgleich mit der Schnittstelle `Iterator`. Die Existenz dieser zwei sehr ähnlichen Schnittstellen hat historische Gründe. Mit einer anonymen inneren Klassen können wir recht einfach ein Rückgabeobjekt der Schnittstelle erzeugen.

Tree.java

```
2    public Enumeration<TreeNode> children() {
3      final Iterator<Tree<a>> it = theChildren().iterator();
4
5      return new Enumeration<TreeNode>(){
6        public boolean hasMoreElements(){return it.hasNext();}
```

```
7       public Tree<a> nextElement() {return it.next();}
8     };
9   }
```

Die Methoden `getChildCount`, `getParent`, `isLeaf` `getChildAt` existieren bereits oder sind einfach aus der bestehenden Klasse ableitbar.

Tree.java

```
10  public TreeNode getChildAt(int childIndex) {
11    return theChildren().get(childIndex);
12  }
13
14  public int getChildCount(){return theChildren().size();}
15
16  public TreeNode getParent(){return parent;}
```

Die übrigen zwei Methoden aus der Schnittstelle `TreeNode` sind für unsere Zwecke unerheblich. Wir implementieren sie so, daß sie eine Ausnahme werfen.

Tree.java

```
17
18  public boolean getAllowsChildren(){
19    throw new UnsupportedOperationException();
20  }
21
22  public int getIndex(TreeNode node) {
23    throw new UnsupportedOperationException();
24  }
```

Schließlich ändern wir die Methode `toString` so ab, daß sie nicht mehr eine `String`-Repräsentation des ganzen Baumes ausgibt, sondern nur des aktuellen Knotens:

Tree.java

```
25  public String toString(){return mark().toString();}
```

Somit haben wir unsere Klasse `Tree` soweit massiert, daß sie von der Klasse `JTree` als Baummodell genutzt werden kann.

Beispiel:
Wir können jetzt ein beliebiges Objekt der Klasse `Tree` der Klasse `JTree` übergeben, um es graphisch darzustellen. Der Code hierzu ist denkbar einfach.

JTreeTest.java

```
1   package name.panitz.data.tree.example;
2
3   import javax.swing.JFrame;
4   import javax.swing.JTree;
5
6   public class JTreeTest {
7     public static void main(String [] args){
```

```
8       JFrame frame = new JFrame("Baumtest Fenster");
9       frame.getContentPane()
10            .add(new JTree(SkriptTree.getSkript()));
11
12      frame.pack();
13      frame.setVisible(true);
14    }
15  }
```

Diese Methode erzeugt die graphische Baumdarstellung aus Abbildung 3.3.

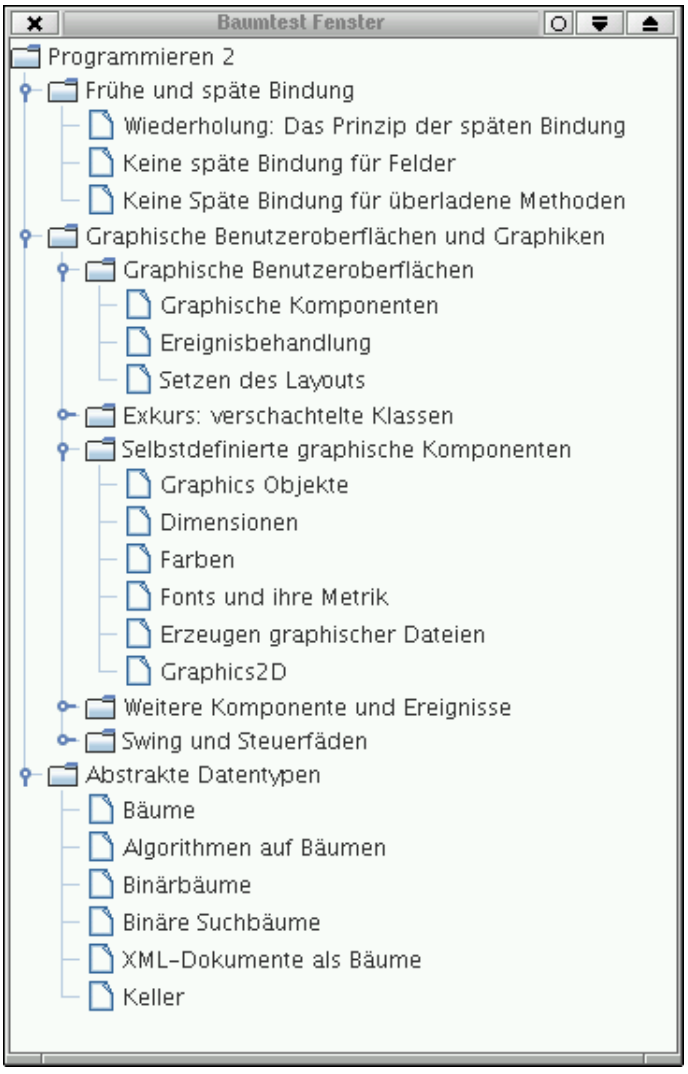


Abbildung 3.3: Baumdarstellung mit JTree.

Dieser Baum reagiert dabei sogar auf Mausaktionen, in dem durch Klicken auf Baumknoten deren Kinder ein- und ausgeblendet werden können.

Die Klasse `JTree` verfügt noch über eine große Anzahl weiterer Funktionalitäten, die wir hier nicht im Detail betrachten wollen.

3.2.7 Bäume zeichnen

Im letzem Unterkapitel haben wir eine fertige Swingkomponente benutzt um Bäume darzustellen. In diesem Abschnitt werden wir eine eigene Komponente schreiben, die Bäume graphisch darstellt. Dabei soll ein Baum in der Weise gezeichnet werden, wir wir es handschriftlich gerne machen: oben zentriert liegt die Wurzel und Linien führen zu den weiter unten hingeschriebenen Kindern.

Die Hauptschwierigkeit hierbei wird sein, die einzelnen Positionen der Baumknoten zu berechnen.

DisplayTree.java

```
1  package name.panitz.data.tree;
2
3  import java.util.Enumeration;
4
5  import javax.swing.JComponent;
6  import javax.swing.JPanel;
7  import javax.swing.tree.TreeNode;
8
9  import java.awt.Font;
10 import java.awt.FontMetrics;
11 import java.awt.Color;
12 import java.awt.Graphics;
13 import java.awt.Dimension;
14
15 public class DisplayTree extends JPanel{
```

Ähnlich wie in der Klasse `JTree` enthält das Objekt, das einen Baum graphisch darstellt, ein Baummodell:

DisplayTree.java

```
16   private TreeNode treeModell;
```

In zwei Konstanten sei spezifiziert, wieviel Zwischenraum horizontal zwischen Geschwistern und vertikal zwischen Eltern und Kindern liegen soll.

DisplayTree.java

```
17   static final public int VERTICAL_SPACE = 50;
18   static final public int HORIZONTAL_SPACE = 20;
```

In einem Feld werden wir uns die Maße des benutzten Fonts vermerken. Dieses ist wichtig, um die Größe eines Baumknotens zu errechnen.

DisplayTree.java

```
19     private FontMetrics fontMetrics = null;
```

Die Felder für Höhe und Breite werden die Maße der Baumzeichnung wiederspiegeln.

DisplayTree.java

```
20  private int height=0;
21  private int width=0;
```

Diese Maße sind zu berechnen, abhängig vom Font der benutzt wird. Eine bool'scher Wert wird uns jeweils angeben, ob die Maße bereits berechnet wurden.

DisplayTree.java

```
22  private boolean dimensionCalculated = false;
```

Es folgen zwei naheliegende Konstruktoren.

DisplayTree.java

```
23  public DisplayTree(TreeNode tree){treeModell=tree;}
24
25  public DisplayTree(TreeNode t,FontMetrics fm){
26    treeModell=t;fontMetrics=fm;
27  }
```

Die schwierigste Methode berechnet, wie groß der zu zeichnende Baum wird:

DisplayTree.java

```
28  private void calculateDimension(){
```

Zunächst stellen wir sicher, daß die Fontmaße bekannt sind:

DisplayTree.java

```
29
30    if (fontMetrics==null){
31      final Font font = getFont();
32      fontMetrics = getFontMetrics(font);
33    }
```

Die Breite der Knotenmarkierung wird berechnet. Dieses ist die minimale Breite des Gesamtbaums. Zunächst berechnen wir die Breite des Wurzelknotens:

DisplayTree.java

```
34    final int x
35      = fontMetrics.stringWidth(treeModell.toString());
```

Für jedes Kind berechnen wir nun die Breite und Höhe. Die maximale Höhe der Kinder bestimmt die Höhe des Gesamtbaums, die Summe aller Breiten der Kinder, bestimmt die Breite des Gesamtbaums.

DisplayTree.java

```
36    final Dimension childrenDim = childrenSize();
37    final int childrenX = (int)childrenDim.getWidth();
38    final int childrenY = (int)childrenDim.getHeight();
```

Mit der Höhe und Breite der Kinder, läßt sich die Höhe und Breite des Gesamtbaums berechnen:

DisplayTree.java

```
39    width=x>childrenX?x:childrenX;
40
41    height
42      =  childrenY == 0
43        ?fontMetrics.getHeight()
44        :VERTICAL_SPACE+childrenY;
45
46    dimensionCalculated = true;
47  }
```

Zum Berechnen der Größe aller Kinder haben wir oben eine Methode angenommen, die wir hier implementieren.

DisplayTree.java

```
48  Dimension childrenSize(){
49    int x = 0;
50    int y = 0;
51
52    final Enumeration<TreeNode> it=treeModell.children();
53    while (it.hasMoreElements()){
54      final DisplayTree t
55       = new DisplayTree(it.nextElement(),fontMetrics);
56
57      y =  y > t.getHeight()?y:t.getHeight();
58
59      x=x+t.getWidth();
60
61      if (it.hasMoreElements())  x=x+HORIZONTAL_SPACE;
62    }
63    return new Dimension(x,y);
64  }
```

Mehrere Methoden geben die entsprechende Größe aus, nachdem sie gegebenenfalls vorher ihre Berechnung angestoßen haben:

DisplayTree.java

```
65  public int getHeight(){
66    if (!dimensionCalculated) calculateDimension();
67    return height;
68  }
69
70  public int getWidth(){
71    if (!dimensionCalculated) calculateDimension();
72    return width;
73  }
74
```

```
75  public Dimension getSize(){
76    if (!dimensionCalculated) calculateDimension();
77    return new Dimension(width,height);
78  }
79
80  public Dimension getMinimumSize(){
81    return getSize();
82  }
83
84  public Dimension getPreferredSize(){
85    return getSize();
86  }
```

Schließlich die eigentliche Methode zum Zeichnen des Baumes. Hierbei benutzen wir eine Hilfsmethode, die den Baum an eine bestimmte Position auf der Leinwand plaziert:

DisplayTree.java

```
87  public void paintComponent(Graphics g){paintAt(g,0,0);}
88
89  public void paintAt(Graphics g,int x,int y){
90    fontMetrics = g.getFontMetrics();
91    final String marks =treeModell.toString();
```

Zunächst zeichne die Baummarkierung für die Wurzel des Knotens. Zentriere diese in bezug auf die Breite des zu zeichnenden Baumes:

DisplayTree.java

```
92    g.drawString
93     (marks
94     ,x+(getWidth()/2-fontMetrics.stringWidth(marks)/2)
95     ,y+10);
```

Jetzt sind die Kinder zu zeichenen. Hierzu ist zu berechnen, wo ein Kind zu positionieren ist. Zusätzlich ist die Kante von Elternknoten zu Kind zu zeichnen. Hierzu merken wir uns den Startpunkt der Kanten beim Elternknoten. Auch dieser ist auf der Breite zentriert:

DisplayTree.java

```
96    final int startLineX = x+getWidth()/2;
97    final int startLineY = y+10;
```

Für jedes Kind ist jetzt die x-Position zu berechnen, das Kind zu zeichnen, und die Kante zum Kind zu zeichenen:

DisplayTree.java

```
98    final Enumeration it=treeModell.children();
99
100   final int childrenWidth = (int)childrenSize().getWidth();
101
102   //wieviel nach rechts zu rücken ist
103   int newX
```

```
104       = getWidth()>childrenWidth?(getWidth()-childrenWidth)/2:0;
105
106     //die y-Koordinate der Kinder
107     final int nextY = y+VERTICAL_SPACE;
108
109     while (it.hasMoreElements()){
110       DisplayTree t
111        = new DisplayTree((TreeNode)it.nextElement());
112
113       //x-Positionen für das nächste Kind
114       final int nextX = x+newX;
115
116       //zeichne das Kind
117       t.paintAt(g,nextX,nextY);
118
119       //zeichne Kante
120       g.drawLine(startLineX,startLineY
121                  ,nextX+t.getWidth()/2,nextY);
122
123       newX = newX+t.getWidth()+HORIZONTAL_SPACE;
124     }
125   }
126 }
```

Damit ist die Klasse zur graphischen Darstellung von Bäumen komplett spezifiziert. Folgendes kleine Programm benutzt unsere Klasse `DisplayTree` in der gleichen Weise, wie wir auch die Klasse `JTree` benutzt haben. Das optische Ergebnis ist in Abbildung 3.4 zu bewundern.

DisplayTreeTest.java

```
1  package name.panitz.data.tree.example;
2  import name.panitz.data.tree.DisplayTree;
3  import javax.swing.JFrame;
4  import javax.swing.JScrollPane;
5
6  class DisplayTreeTest {
7    public static void main(String [] args){
8      JFrame frame = new JFrame("Baum Fenster");
9      frame.getContentPane()
10          .add(new JScrollPane(
11                 new DisplayTree(SkriptTree.getSkript())));
12
13     frame.pack();
14     frame.setVisible(true);
15   }
16 }
```

Wir schreiben eine kleine Klasse mit Methoden, um das Bild eines Baumes in eine Datei zu speichern.

Zunächst brauchen wir eine ganze Reihe von Klassen, die wir importieren:

Abbildung 3.4: Baum graphisch dargestellt mit DisplayTree.

TreeToFile.java

```
1  package name.panitz.data.tree;
2  import name.panitz.gui.graphicsFile.ComponentToFile;
3  import static name.panitz.gui.graphicsFile.ComponentToFile.*;
4
5  import java.io.File;
6  import java.io.IOException;
7
8  import javax.imageio.ImageIO;
9  import javax.swing.JComponent;
10 import java.awt.image.RenderedImage;
11
12 public class TreeToFile{
13   public static void treeToFile(Tree tree,String fileName){
14     try {
15      final JComponent component = new DisplayTree(tree);
16      final RenderedImage image = createComponentImage(component);
17      ImageIO.write(image,"png", new File(fileName+".png"));
18      ImageIO.write(image,"jpg", new File(fileName+".jpg"));
19     }catch (IOException e) {System.out.println(e);}
20   }
21 }
```

3.3 Binärbäume

Listen sind Bäume, in denen jeder Knoten maximal ein Kind hat. Binärbäume sind Bäume, in denen jeder Knoten maximal zwei Kinder hat. Wir können eine Unterklasse von `Tree` schreiben, in der ein Konstruktor sicherstellt, daß ein Knoten maximal zwei Kinder hat.

Wir wollen zusätzlich zulassen, daß es ein rechtes aber kein linkes Kind gibt. Dafür stellen wir zwei Felder zur Verfügung, die die entsprechenden Kinder enthalten. Existiert ein Kind nicht, so sei das entsprechende Feld mit `null` belegt.

Damit wir die geerbten Methoden fehlerfrei benutzen können, müssen wir sicherstellen, daß die Methode `children` auch für diese Unterklasse gemäß der Spezifikation funktioniert.

BinTree.java
```
1  package name.panitz.data.tree;
2
3  import java.util.List;
4  import java.util.ArrayList;
5
6  public class BinTree<a> extends Tree<a>{
7
8    //private Felder für rechtes und linkes Kind
9    private BinTree<a> rght;
10   private BinTree<a> lft;
```

Wir sehen zwei naheliegende Konstuktoren vor:

BinTree.java
```
11   public BinTree(BinTree<a> left,a mark,BinTree<a> right){
12     super(mark);lft=left; rght=right;}
13
14   public BinTree(a mark){this(null,mark,null);}
```

Zwei Selektormethoden greifen auf das linken bzw. rechte Kind zu:

BinTree.java
```
15   public BinTree<a> left(){return lft;}
16   public BinTree<a> right(){return rght;}
```

Wir müssen die Methode `theChildren` überschreiben. Die Kinder befinden sich nicht mehr in einer Liste gespeichert sondern in den zwei speziellen hierfür angelegten Feldern:

BinTree.java
```
17   public List<Tree<a>> theChildren(){
18     List<Tree<a>> result = new ArrayList<Tree<a>>();
19     if (left()!=null)  result.add(left());
20     if (right()!=null) result.add(right());
21     return result;
22   }
```

3.3.1 Probleme der Klasse BinTree

Als Unterklasse der Klasse `Tree` erben wir sämtliche Methoden der Klasse `Tree`. Das ist zunächst einmal positiv, weil wir automatisch Methoden wie `count` oder `flatten` erben und nicht neu zu implementieren brauchen.

Leider spielen uns die modifizierenden Methoden der Klasse `Tree` dabei einen Streich. Kurz gesagt: sie funktionieren nicht für die Klasse `BinTree`. Sie modifizieren die Liste der Kinder, wie wir sie von der Methode `theChildren()` erhalten. Diese Liste ist in der Klasse `BinTree` transient, d.h. sie wird nicht im Objekt direkt gespeichert, sondern jeweils neu aus den beiden Feldern `rght` und `lft` erzeugt. Eine Modifikation an dieser Liste hat also keinen bleibenden (persistenten) Effekt.

Andererseits ist dieses eine gute Eigenschaft, denn damit wird verhindet, daß eine modifizierende Methode die Binäreigenschaft zerstört. Die Methode `addChild` könnte sonst einen binären Baum an einem Knoten ein drittes Kind einfügen.

Es empfiehlt sich trotzdem, dieses Problem nicht stillschweigend hinzunehmen, und eine entsprechende Ausnahme zu werfen. Daher fügen wir der Klasse `BinTree` folgende Variante der Methode `addLeaf` hinzu:

BinTree.java

```
23 public void addLeaf(a o){
24     throw new UnsupportedOperationException();
25 }
```

Für die Methode `deleteChild` können wir eine Variante anbieten, die auf binären Bäumen funktioniert.

BinTree.java

```
26 public void deleteChild(int n){
27   if (n==0) lft=null;
28   else if (n==1) rght=null;
29   else throw new IndexOutOfBoundsException ();
30 }
```

Modifizierende Methoden, die ungefährlich für die Binärstrukrur der Bäume sind, setzen die beiden Kinder eines Baumes neu:

BinTree.java

```
31   void setLeft(BinTree<a> l){ lft=l;}
32   void setRight(BinTree<a> r){ rght=r;}
```

Aufgabe 14 Überschreiben Sie die Methoden `addLeaf` und `deleteChildNode` in der Klasse `BinTree`, so daß sie nur eine Ausnahme werfen, wenn die Durchführung der Modifikation dazu führen würde, daß das Objekt, auf dem die Methode angewendet wird, anschließend kein Binärbaum mehr wäre.

3.3.2 Linearisieren binärer Bäume

In der Klasse `Tree` haben wir eine Methode definiert, die die Knoten eines Baumes in einer linearisierten Form in eine Liste speichert. Wir haben uns bisher keine Gedanken gemacht, in welcher Reihenfolge die Baumknoten in der Ergebnisliste stehen. Hierbei sind mehrere fundamentale Ordnungen der Knoten des Baumes vorstellbar:

Preordnung

Unsere bisherige Implementierung der Methode `flatten` benutzt die Reihenfolge, in der ein Knoten in der Ergebnisliste immer vor seinen Kindern steht und Geschwisterknoten stets erst nach den Knoten folgen. Diese Ordnung nennt man Preordnung. In der Preordnung kommt die Wurzel eines Baumes als erstes Element der Ergebnisliste.

Postordnung

Wie der Name vermuten läßt, ist die Postordnung gerade der umgekehrte Weg zur Preordnung. Jeder Knoten steht in der Liste nach allen Kinderknoten. Die Wurzel wird damit das letzte Element der Ergebnisliste.

Aufgabe 15 Schreiben Sie analog zur Methode `flatten` in der Klasse `Tree` eine Methode `List<a> postorder()`, die die Knoten eines Baumes in Postordnung linearisiert.

Inordnung

Pre- und Postordnung kann man nicht nur auf Binärbäumen definieren, sondern allgemein für alle Bäume. Die dritte Variante hingegen, die Inordnung, kann nur für Binärbäume sinnvoll definiert werden. Hier steht jeder Knoten nach allen Knoten seines linken Kindes und vor allen Knoten seines rechten Kindes.

BinTree.java

```
33 public List<a> inorder(){
34   //Ergebnisliste
35   List<a> result = new ArrayList<a>();
36
37   //gibt es ein linkes Kind, füge dessen
38   //Linearisierung hinzu
39   if (left()!=null) result.addAll(left().inorder());
40
41   //dann den Knoten selbst
42   result.add(mark());
43
44   //und gegebenenfalls dann das recte Kind
45   if (right()!=null) result.addAll(right().inorder());
46
47   return result;
48 }
```

Pre-, Post- und Inordnung entsprechen der Pre-, Post- und Infixschreibweise von Operatorausdrücken. Betrachtet man einen Operatorausdruck als Baum mit dem Operator als Wurzel und den beiden Operanden als Kindern, dann ergibt sich die Analogie sofort.

Militärordnung

Eine Ordnung, die der Baumstruktur sehr entgegen steht, ist die sogenannte Militärordnung. Diese Ordnung geht ebenenweise vor. Geschwisterknoten stehen in der Liste direkt nebeneinander. Abbildung 3.5 veranschaulicht diese Ordnung:

Zunächst wird die Wurzel genommen, dann der Reihe nach alle Kinder der Wurzel, dann die Kinder der Kinder usw. Es werden also alle Knoten einer Generation hintereinander geschrieben. Diese Ordnung entspricht nicht der rekursiven Definition einer Baumstruktur. Daher ist ihre Implementierung auch nicht sehr elegant. Hierfür sind ein paar Hilfslisten zu verwalten. Eine Liste für die aktuelle Generation und eine in der die Knoten der nächsten Generation gesammelt werden.

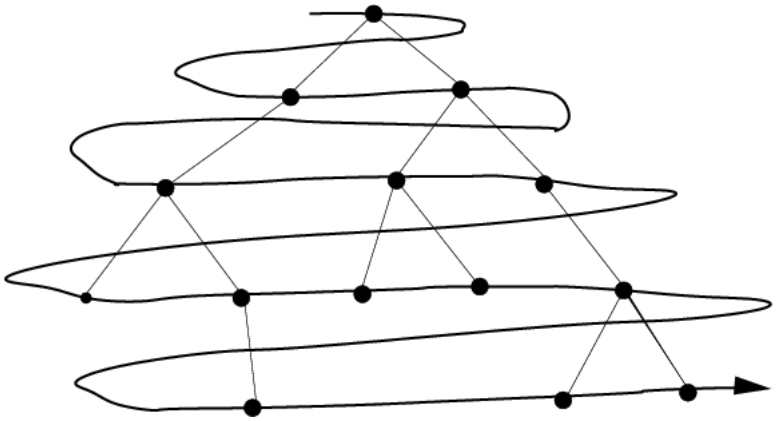


Abbildung 3.5: Schematische Darstellung der Militärordnung.

BinTree.java

```
49 public List military(){
50   //Ergebnisliste enthält erst diesen Knoten
51   List<a> result=new ArrayList<a>();
52   result.add(mark());
53
54   //Wir speichern uns die Knoten einer
55   //aktuellen Generation
56   List<Tree<a>> currentGeneration = theChildren();
57
58   //Solange es in der aktuellen Generation Knoten gibt
59   while (!currentGeneration.isEmpty()){
60     //erzeuge die nächste Generation
61     List<Tree<a>> nextGeneration=new ArrayList<Tree<a>>();
62
63     for (Tree<a> nextChild:currentGeneration){
64       nextGeneration.addAll(nextChild.theChildren());
65
66       //Füge das Elemente der aktuellen Generation
67       //zum Ergebnis
68       result.add(nextChild.mark());
69     }
70     //schalte eine Generation tiefer
71     currentGeneration=nextGeneration;
72   }
73   return result;
74 }}
```

Aufgabe 16 Rechnen Sie auf dem Papier ein Beispiel für die Arbeitsweise der Methode `military()`.

3.4 Binäre Suchbäume

Binärbäume können dazu genutzt werden, um Objekte effizient gemäß einer Ordnung zu speichern und effizient nach ihnen unter Benutzung dieser Ordnung wieder zu suchen. Die Knotenmarkierungen sollen hierzu nicht mehr beliebige Objekte sein, sondern Objekte, die die Schnittstelle `Comparable` implementieren. Solche Objekte können mit beliebigen Objekten verglichen werden, so daß sie eine Ordnungsrelation haben.

Binäre Bäume sollen diese Eigenschaft so ausnutzen, daß für einen Knoten gilt:

- jede Knotenmarkierung eines linken Teilbaums ist kleiner oder gleich als die Knotenmarkierung.
- jede Knotenmarkierung eines rechten Teilbaums ist größer als die Knotenmarkierung.

Man erkennt, daß die Konstruktion solcher Bäume nicht mehr lokal geschehen kann, indem der linke und der rechte Teilbaum in einem neuen Knoten zusammengefasst werden. Bei der Baumkonstruktion ist ein neuer Knoten entsprechend der Ordnung an der richtige Stelle als Blatt einzuhängen.

Wir schreiben eine Unterklasse `SearchTree` der Klasse `BinTree`. Diese hat nur einen Konstruktor, um einen Baum mit nur einem Wurzelknoten zu konstruieren. Weitere Knoten können mit der Methode `insert` in den Baum eingefügt werden. Die Methode `insert` stellt sicher, daß der Baum die größer Relation der Elemente berücksichtigt. Die Elemente des Baumes müssen die Schnittstelle `Comparable` implementieren.

SearchTree.java

```
1  package name.panitz.data.tree;
2
3  public class SearchTree<a extends Comparable<a>>
4                                     extends BinTree<a>{
```

Wir sehen nur einen Konstruktor vor, der ein Blatt erzeugt

SearchTree.java

```
5    SearchTree(a o){super(o);}
```

Suchbäume wachsen durch eine modifizierende Einfügeoperation:

SearchTree.java

```
6    public void insert(a o){
7      if (o.compareTo(mark())<=0){
8        //neuer Knoten ist kneiner, also in den
9        //linken Teilbaum
10       if (left()==null){
11         setLeft(new SearchTree<a>(o));
12       }else
13         ((SearchTree<a>)left()).insert(o);
14     }else{
15       //wenn der neue Knoten also größer, dann dasselbe
16       //im rechten Teilbaum
```

```
17      if (right()==null){
18         setRight(new SearchTree<a>(o));
19      }else
20        ((SearchTree<a>)right()).insert(o);
21    }
22  }
23
```

Wir können für interne Zwecke einen Konstruktor vorsehen, der einen Suchbaum direkt aus den Kindern konstruiert. Hierbei wollen wir uns aber nicht darauf verlassen, daß dem Konstruktor solche Teilbäume und eine solche Knotenmarkierung übergeben werden, daß der resultierende Baum ein korrekter Suchbaum ist.

SearchTree.java

```
24 private SearchTree
25             (SearchTree<a> l,a o,SearchTree<a> r){
26   super(l,o,r);
27   try {
28    if (!(   o.compareTo(l.mark())>=0
29          && o.compareTo(r.mark())<=0))
30      throw new IllegalArgumentException
31      ("ordering violation in search tree construction");
32   }catch(NullPointerException _){
33   }
34 }
```

Die `if`-Bedingung mit der Prüfung auf eine Eigenschaft der Parameter ist ein typischer Fall für eine Zusicherung, wie sie seit Javas Version 1.4 ein fester Bestandteil von Java ist.

Die Linearisierung eines binären Suchbaums in Inordnung ergibt eine sortierte Liste der Baumknoten, wie man sich mit folgenden Test vergegenwärtigen kann:

TestSearchTree.java

```
1  package name.panitz.data.tree;
2  import javax.swing.*;
3
4  public class TestSearchTree {
5    public static void main(String [] args){
6      SearchTree<String> t = new SearchTree<String>("otto");
7      t.insert("sven");
8      t.insert("eric");
9      t.insert("lars");
10     t.insert("uwe");
11     t.insert("theo");
12     t.insert("otto");
13     t.insert("kai");
14     t.insert("henrik");
15     t.insert("august");
16     t.insert("berthold");
17     t.insert("arthur");
```

```
18      t.insert("arno");
19      t.insert("william");
20      t.insert("tibor");
21      t.insert("hassan");
22      t.insert("erwin");
23      t.insert("anna");
24      System.out.println(t.inorder());
25
26      JFrame frame = new JFrame("Baum Fenster");
27      frame.getContentPane().add(new DisplayTree(t));
28      frame.pack();
29      frame.setVisible(true);
30    }
31  }
```

Das Programm führt zu folgender Ausgabe auf der Kommandozeile.

```
sep@swe10:~/fh/prog2/beispiele/Tree> java TestSearchTree
[anna, arno, arthur, august, berthold, eric, erwin, hassan, henrik,
kai, lars, otto, otto, sven, theo, tibor, uwe, william]
sep@swe10:~/fh/prog2/beispiele/Tree>
```

Aufgabe 17 Zeichnen Sie schrittweise die Baumstruktur, die im Programm `TestSearchTree` aufgebaut wird.

Beweis der Sortiereigenschaft

Wir haben oben behauptet und experiementell an einem Beispiel ausprobiert, daß die Inordnung eines binären Suchbaums eine sortierte Liste als Ergebnis hat. Wir wollen jetzt versuchen diese Aussage zu beweisen.

Zum Beweis einer Aussage über eine rekursiv definierten Datenstruktur (wie Listen und Bäume) bedient man sich der vollständigen Induktion.[2] Hierbei geht die Induktion meist über die Größe der Datenobjekte. Als Induktionsanfang dient das kleinste denkbare Datenobjekt, bei Listen also leere Listen, bei Bäumen Blätter. Der Induktionsschluß setzt dabei voraus, daß die Aussage für alle kleineren Objekte bereits bewiesen ist, also für alle Teilobjekte (Teilbäume und Teillisten) bereits gilt.

Beweisen wir entsprechend die Aussage, daß die Liste der Elemente eines binären Suchbaums in Inordnung sortiert ist:

- **Anfang:** Der Induktionsanfang behauptet, die Inordung eines Blattes ist sortiert. Für ein Blatt k gilt: $left(k) = null$ und $right(k) = null$. Es folgt, daß die Inordung nur ein Element, nämlich $mark(k)$, enthält. Einelementige Listen sind immer sortiert.
- **Schritt:** Wir wollen die Aussage beweisen für alle Bäume k mit: $count(k) = n$.

  Annahme: die Aussage ist wahr für alle Bäume k' mit $count(k') < n$.

[2] Die im philosophischen Sinne keine Induktion sondern eine Deduktion ist.

Die Definition der Inordnung ist:
$inorder(k) = (inorder(left(k)), mark(k), inorder(right(k)))$.

Für endliche Bäume gilt:
$count(left(k)) < n$ und $count(right(k)) < n$
Nach der Induktionsvoraussetzung gilt damit, daß
$inorder(left(k))$ und $inorder(right(k))$
sortierte Listen sind.

Über die Definition der binären Suchbäume gilt:
für alle $x \in inorder(left(k))$ gilt $x \leq mark(k)$
und für alle $x \in inorder(right(k))$ gilt $x > mark(k)$

Damit ist insbesondere die Liste $inorder(k)$ sortiert.

3.4.1 Suchen in Binärbäumen

Die Sortiereigenschaft der Suchbäume erlaubt es jetzt, die Methode `contains` so zu schreiben, daß sie maximal einen Pfad durchlaufen muß. Es muß nicht mehr der ganze Baum betrachtet werden.

SearchTree.java

```
32 public boolean contains(a o){
33   //bist du es schon selbst
34   if (mark().equals(o)) return true;
35
36   //links oder rechts suchen:
37   final int compRes = mark().compareTo(o);
38
39   if (compRes>0){
40     //keine Knoten mehr. Dann ist er nicht enthalten
41     if (left()==null) return false;
42     //Sonst such weiter unten im Baum
43     else return ((SearchTree<a>)left()).contains(o);
44   }
45
46   if (compRes<0){
47     if (right()==null) return false;
48     else return ((SearchTree<a>)right()).contains(o);
49   }
50   return false;
51 }
```

Wie man sieht, ist die Methode in ihrer Struktur analog zur Methode `insert`.

3.4.2 Entartete Bäume

Bäume, die keine eigentliche Baumstruktur mehr haben, sondern nur noch Listen sind, werden manchmal auch als entartete Bäume bezeichnet. Insbesondere bei binären Suchbäumen verlieren wir die schönen Eigenschaften, nämlich daß wir maximal in der maximalen Pfadlänge im Baum zu suchen brauchen.

Aufgabe 18 Erzeugen Sie ein Objekt des typs `SearchTree` und fügen Sie nacheinander die folgenden Elemente ein:
`"anna","berta","carla","dieter","erwin","florian","gustav"`
Lassen Sie anschließend die maximale Tiefe des Baumes ausgeben.

Balanzieren von Bäumen

Wie wir in der letzten Aufgabe sehen konnnten, liegt es an der Reihenfolge, in der die Elemente in einem binären Suchbaum eingefügt werden, wie gut ein Baum ausbalanziert ist. Wir sprechen von einem ausbalanzierten Baum, wenn sein linker und rechter Teilbaum die gleiche Tiefe haben.

Man kann einen Baum, der nicht ausbalanziert ist, so verändern, daß er weiterhin ein korrekter binärer Suchbaum in Bezug auf die Ordnung ist, aber die Tiefen der linken und rechten Kinder ausgewogen sind, d.h. sich nicht um mehr als eins unterscheiden.

Beispiel:
Betrachten Sie den Baum aus Abbildung 3.6

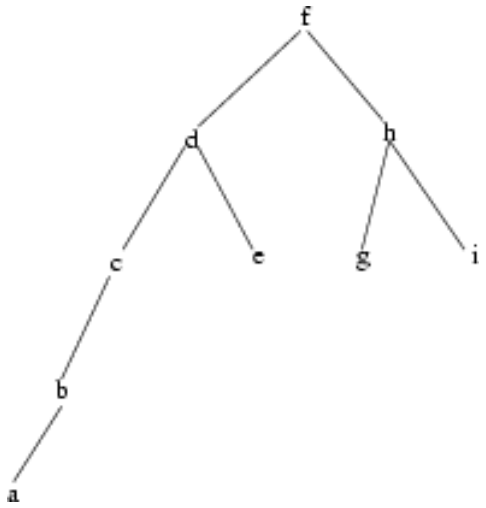


Abbildung 3.6: Nicht balanzierter Baum.

Der linke Teilbaum hat eine Tiefe von 4, der rechte eine Tiefe von 2. Wir können den Baum so verändern, daß beide Kinder eine Tiefe von 3 haben. Wir bekommen den Baum aus Abbildung 3.7

Das linke Kind dieses Baumes ist auch nicht ausbalanziert, wir können den Baum noch einmal verändern und erhalten den Baum in Abbildung 3.8:

Spezifikation Zum Ausbalanzieren wird ein Baum gedreht. Einer seiner Kinderknoten wird die neue Wurzel und die alte Wurzel dessen Kind. Es gibt zwei Richtungen, in die gedreht werden kann: mit und gegen den Urhzeigersinn. Schematisch lassen sich diese beiden Operationen mit folgenden Gleichungen spezifizieren:

- rotateRight(Node(Node(l2,x2,r2),x1,r1)) = Node(l2,x2,Node(r2,x1,r1))

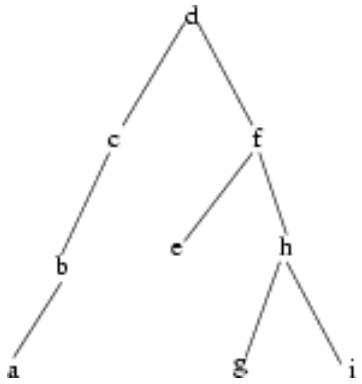


Abbildung 3.7: Baum nach Rotierungsschritt.

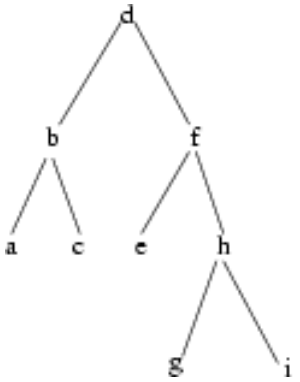
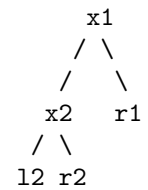


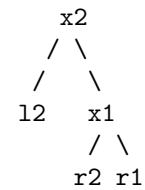
Abbildung 3.8: Baum nach weiteren Rotierungsschritt.

- rotateLeft(Node(l1,x1,Node(l2,x2,r2))) = Node(Node(l1,x1,l2),x2,r2)

In graphischer Darstellung wird die erste Gleichung zu:



wird mit `rotateRight` zu:



Die Operationen erhalten die Eigenschaft eines Suchbaumes: linke Kinder sind kleiner, rechte Kinder größer als die Wurzel. Die Linearisierung in der Inordnung bleibt gleich. Die maximale Tiefe der Kinder verändert sich um 1. Die Anzahl der Knoten im Baum ändert sich nicht.

Implementierung Wir können generell zwei Umsetzungen des oben spezifizierten Algorithmus wählen: eine die einen neues Baumobjekt erzeugt und keinde der bestehenden Bäume modifiziert und eine, die die Felder der bestehenden Baumknoten so umändert, daß das Objekt, auf dem die Methode aufgerufen wird, modifiziert wird.

Funktionale Umsetzung Für die funktionale Umsetzung können wir eine statische Methode schreiben. Sie bekommt einen Baum und soll einen neuen Baum, der durch Rotation aus den Eingabebaum entstanden ist, als Ergebnis liefern.

Die Umsetzung leitet sich direkt aus der Spezifikation über eine Gleichung ab. Wir benutzen dieselben Bezeichner wie in der Spezifikation.

SearchTree.java

```
52 public static <a extends Comparable<a>> SearchTree<a>
53                          rotateRight(SearchTree<a> t){
54   try{
55     //speichere die einzelnen Teile in die Bezeichner,
56     //die wir in der Spezifikation benutzt haben
57     final SearchTree<a> l2 = (SearchTree<a>)t.left().left();
58     final SearchTree<a> r2 = (SearchTree<a>)t.left().right();
59     final SearchTree<a> r1 = (SearchTree<a>)t.right();
60     final a x1 = t.mark();
61     final a x2 = t.left().mark();
62
63     return new SearchTree<a>
64                  (l2,x2,new SearchTree<a>(r2,x1,r1));
65   }catch (NullPointerException _){
66       return t;
67   }
68 }
```

Die Ausnahme tritt auf, wenn das linke Kind nicht existiert, also hier der Wert `null` steht. Dann kann nicht nach rechts rotiert werden (da links nichts zum Rotieren steht). In dem Fall, das nicht rotiert werden kann, wird der Eingabebaum unverändert ausgegeben.

Modifizierende Umsetzung Wir können die modifizierende Methode zum Rotieren eines Baumes in der Klasse `BinTree` implementieren. Wir benutzen wieder die Bezeichner aus unserer Spezifikation. Anstatt aber wie eben neue Knoten zu erzeugen, setzen die entsprechenden Referenzen um.

SearchTree.java

```
69 void rotateRight(){
70   try{
71     BinTree<a> l1 = left();
72     BinTree<a> l2 = l1.left();
73     BinTree<a> r2 = l1.right();
74     BinTree<a> r1 = right();
75     a  x1 = mark();
76     a  x2 = l1.mark();
77
78     setMark(x2);
79     setLeft(l2);
80     setRight(l1);
81     l1.setMark(x1);
82     l1.setLeft(r2);
83     l1.setRight(r1);
```

```
84    }catch (NullPointerException _){
85    }
86  }}
```

Die Ausnahme wird schon in der zweiten Zeile geworfen, bevor Referenzen verändert wurden. Das Objekt bleibt unverändert. Ansonsten werden die Knoten entsprechend der Spezifikation umgehängt.

Aufgabe 19

a) Schreiben sie entsprechend die Methode:
static public SearchTree rotateLeft(SearchTree t)

b) Schreiben Sie in der Klasse `BinTree` die entsprechende modifizierende Methode `rotateLeft()`

c) Testen Sie die beiden Rotierungsmethoden. Testen Sie, ob die Tiefe der Kinder sich verändert und ob die Inordnung gleich bleibt. Testen Sie insbesondere auch einen Fall, in dem die `NullPointerException` abgefangen wird.

Kapitel 4

Formale Sprachen, Grammatiken, Parser

Sprachen sind ein fundamentales Konzept nicht nur der Informatik. In der Informatik begegnen uns als auffälligste Form der Sprache*Programmiersprachen*. Es gibt gewisse Regeln, nach denen die Sätze einen Sprache aus einer Menge von Wörtern geformt werden können. Die Regeln nennen wir im allgemeinen eine *Grammatik*. Ein Satz einer Programmiersprache nennen wir Programm. Die Wörter sind, Schlüsselwörter, Bezeichner, Konstanten und Sonderzeichen.

Ausführlich beschäftigt sich die Vorlesung *Compilerbau*[GKS01] mit formalen Sprachen. Wir werden in diesem Kapitel die wichtigsten Grundkenntnisse hierzu betrachten, wie sie zum Handwerkszeug eines jeden Informatikers gehören.

4.1 formale Sprachen

Eine der bahnbrechenden Erfindungen des 20. Jahrhunderts geht auf den Sprachwissenschaftler Noam Chomsky[Cho56][1] zurück. Er präsentierte als erster ein formales Regelsystem, mit dem die Grammatik einer Sprache beschrieben werden kann. Dieses Regelsystem ist in seiner Idee verblüffend einfach. Es bietet Regeln an, mit denen mechanisch die Sätze einer Sprache generiert werden können.

Systematisch wurden Chomsky Ideen zum erstenmal für die Beschreibung der Syntax der Programmiersprache Algol angewendet[NB60].

4.1.1 kontextfreie Grammatik

Eine kontextfreie Grammatik besteht aus

- einer Menge $\mathcal{T}$ von Wörteren, den *Terminalsymbole*.

[1] Chomsky gilt als der am häufigsten zitierte Wissenschaftler des 20. Jahrhunderts. Heutzutage tritt Chomsky weniger durch seine wissenschaftlichen Arbeiten als vielmehr durch seinen Einsatz für Menschenrechte und bedrohte Völker in Erscheinung.

- einer Menge $\mathcal{N}$ von Nichtterminalsymbolen.
- ein ausgezeichnetes Startsymbol $S \in \mathcal{N}$.
- einer endlichen Menge $\mathcal{R}$ von Regeln der Form:
  $nt ::= t_1 \ldots t_n$, wobei $nt \in \mathcal{N}$, $t_i \in \mathcal{N} \cup \mathcal{T}$.

Mit den Regeln einer kontextfreien Grammatik werden Sätze gebildet, indem ausgehend vom Startsymbol Regel angewendet werden. Bei einer Regelanwendung wird ein Nichtterminalzeichen t durch die Rechte Seite einer Regel, die t auf der linken Seite hat, ersetzt.

Beispiel:
Wir geben eine Grammatik an, die einfache Sätze über unser Sonnensystem auf Englisch bilden kann:

- $\mathcal{T}$ = {mars,mercury,deimos,phoebus,orbits,is,a,moon,planet}
- $\mathcal{N}$ = {*start,noun-phrase,verb-phrase,noun,verb,article*}
- S =*start*
- *start* ::= *noun-phrase verb-phrase*
  noun-phrase ::= *noun*
  noun-phrase ::= *article noun*
  verb-phrase ::= *verb noun-phrase*
  noun ::= planet
  noun ::= moon
  noun ::= mars
  noun ::= deimos
  noun ::= phoebus
  verb ::= orbits
  verb ::= is
  article ::= a

Wir können mit dieser Grammatik Sätze in der folgenden Art bilden:

- *start*
  →*noun-phrase verb-phrase*
  →*article noun verb-phrase*
  →*article noun verb noun-phrase*
  → a *noun verb noun-phrase*
  → a moon *verb noun-phrase*
  → a moon orbits *noun-phrase*
  → a moon orbits *noun*
  → a moon orbits mars

- *start*
  →*noun-phrase verb-phrase*
  →*noun verb-phrase*
  → mercury *verb-phrase*
  → mercury *verb noun-phrase*
  → mercury is *noun-phrase*

→ mercury is *article noun*
→ mercury is a *noun*
→ mercury is a planet

- Mit dieser einfachen Grammatik lassen sich auch Sätze bilden, die weder korrektes Englisch sind, noch eine vernünftige inhaltliche Aussage machen:
  start
  →*noun-phrase verb-phrase*
  →*noun verb-phrase*
  → planet *verb-phrase*
  → planet *verb noun-phrase*
  → planet orbits *noun-phrase*
  → planet orbits *article noun*
  → planet orbits a *noun*
  → planet orbits a phoebus

Eine Grammatik beschreibt die Syntax einer Sprache im Gegensatz zur Semantik, der Bedeutung, einer Sprache.

Rekursive Grammatiken

Die Grammatik aus dem letzen Beispiel kann nur endlich viele Sätze generieren. Will man mit einer Grammatik unendlich viele Sätze beschreiben, so wie eine Programmiersprache unendlich viele Programme hat, so kann man sich dem Trick der Rekursion bedienen. Eine Grammatik kann rekursive Regeln enthalten; das sind Regeln, in denen auf der rechten Seite das Nichtterminalsymbol der linken Seite wieder auftaucht.

Beispiel:
Die folgende Grammatik erlaubt es arithmetische Ausdrücke zu generieren:

- $\mathcal{T} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$
- $\mathcal{N} = \{start, expr, op, integer, digit,\}$
- $S =$ *start*
- *start* ::= *expr*
  expr ::= *integer*
  expr ::= *integer op expr*
  integer ::= *digit*
  integer ::= *digit integer*
  op ::= +
  op ::= -
  op ::= *
  op ::= /
  digit ::= 0
  digit ::= 1
  digit ::= 2
  digit ::= 3

$digit ::= 4$
$digit ::= 5$
$digit ::= 6$
$digit ::= 7$
$digit ::= 8$
$digit ::= 9$

Diese Grammatik hat zwei rekursive Regeln: eine für das Nichtterminal *expr* und eines für das Nichtterminal *integer*.

Folgende Abeleitung generiert einen arithmetischen Ausdrucke mit dieser Grammatik:

- $start$
  $\rightarrow expr$
  $\rightarrow integer\ op\ expr$
  $\rightarrow integer\ op\ integer\ op\ expr$
  $\rightarrow integer\ op\ integer\ op\ integer\ op\ expr$
  $\rightarrow integer\ op\ integer\ op\ integer\ op\ integer$
  $\rightarrow integer + integer\ op\ integer\ op\ integer$
  $\rightarrow integer + integer * integer\ op\ integer$
  $\rightarrow integer + integer * integer - integer$
  $\rightarrow digit\ integer + integer * integer - integer$
  $\rightarrow 1\ integer + integer * integer - integer$
  $\rightarrow 1\ digit\ integer + integer * integer - integer$
  $\rightarrow 12\ integer + integer * integer - integer$
  $\rightarrow 12\ digit + integer * integer - integer$
  $\rightarrow 129 + integer * integer - integer$
  $\rightarrow 129 + digit * integer - integer$
  $\rightarrow 129 + 4 * integer - integer$
  $\rightarrow 129 + 4 * digit\ integer - integer$
  $\rightarrow 129 + 4 * 5 integer - integer$
  $\rightarrow 129 + 4 * 5 digit - integer$
  $\rightarrow 129 + 4 * 53 - integer$
  $\rightarrow 129 + 4 * 53 - digit\ integer$
  $\rightarrow 129 + 4 * 53 - 8\ integer$
  $\rightarrow 129 + 4 * 53 - 8\ digit$
  $\rightarrow 129 + 4 * 53 - 87$

Aufgabe 20 Erweitern Sie die obige Grammatik so, daß sie mit ihr auch geklammerte arithmetische Ausdrücke ableiten können. Hierfür gibt es zwei neue Terminalsymbolde: `(` und `)`.

Schreiben Sie eine Ableitung für den Ausdruck: `1+(2*20)+1`

Grenzen kontextfreier Grammatiken

Kontextfreie Grammatiken sind ein einfaches und dennoch mächtiges Beschreibungsmittel für Sprachen. Dennoch gibt es viele Sprachen, die nicht durch eine kontextfreie Grammatik beschrieben werden können.

syntaktische Grenzen Es gibt syntaktisch recht einfache Sprachen, die sich nicht durch eine kontextfreie Grammatik beschreiben lassen. Eine sehr einfache solche Sprache besteht aus drei Wörtern: $\mathcal{T}$= {a,b,c}. Die Sätze dieser Sprache sollen so gebildet sein, daß für eine Zahl n eine Folge von n mal dem Zeichen a, n mal das Zeichen b und schließlich n mal das Zeichen c folgt, also

$$\{a^n b^n c^n | n \in I\!N\}.$$

Die Sätze dieser Sprache lassen sich aufzählen:

abc
aabbcc
aaabbbccc
aaaabbbbcccc
aaaaabbbbbccccc
aaaaaabbbbbbcccccc
...

Es gibt formale Beweise, daß derartige Sprachen sich nicht mit kontextfreie Grammatiken bilden lassen. Versuchen Sie einmal das Unmögliche: eine Grammatik aufzustellen, die diese Sprache erzeugt.

Eine weitere einfache Sprache, die nicht durch eine kontextfreie auszudrücken ist, hat zwei Terminalsymbole und verlangt, daß in jedem Satz die beiden Symbole gleich oft vorkommen, die Reihenfolge jedoch beliebig sein kann.

semantische Grenzen Über die Syntax hinaus, haben Sprachen noch weitere Einschränkungen, die sich nicht in der Grammatik ausdrücken lassen. Die meisten syntaktisch korrekten Javaprogramme werden trotzdem vom Javaübersetzer als inkorrekt zurückgewiesen. Diese Programme verstoßen gegen semantische Beschränkungen, wie z.B. gegen die Zuweisungskompatibilität. Das Programm:

```
1 class SemanticalIncorrect{int i = "1";}
```

ist syntaktisch nach den Regeln der Javagrammatik korrekt gebildet, verletzt aber die Beschränkung, daß einem Feld vom Typ `int` kein Objekt des Typs `String` zugewiesen werden darf.

Aus diesen Grund besteht ein Übersetzer aus zwei großen Teilen. Der syntaktischen Analyse, die prüft, ob der Satz mit den Regeln der Grammatik erzeugt werden kann und der semantischen Analyse, die anschließend zusätzliche semantische Bedingungen prüft.

Das leere Wort

Manchmal will man in einer Grammatik ausdrücken, daß in Nichtterminalsymbol auch zu einem leeren Folge von Symbolen reduzieren soll. Hierzu könnte man die Regel

$t ::=$

mit leerer rechter Seite schreiben. Es ist eine Konvention ein spezielles Zeichen für das leere Wort zu benutzen. Hierzu bedient man sich des griechischen Buchtstabens ϵ. Obige Regel würde man also schreiben als:

$t ::= \epsilon$

Lexikalische Struktur

Bisher haben wir uns keine Gedanken gemacht, woher die Wörter unserer Sprache kommen. Wir haben bisher immer eine gegebene Menge angenommen. Die Wörter einer Sprache bestimmen ihre lexikalische Struktur. In unseren obigen Beispielen haben wir sehr unterschiedliche Arten von Wörtern: einmal Wörter der englischen Sprache und einmal Ziffernsymbole und arithmetische Operatorsymbole. Im Kontext von Programmiersprachen spricht man von Token.

Bevor wir testen können, ob ein Satz mit einer Grammatik erzeugt werden kann, sind die einzelnen Wörter in diesem Satz zu identifizieren. Dieses geschieht in einer lexikalischen Analyse. Man spricht auch vom *Lexer* und *Tokenizer*. Um zu beschreiben, wie die einzelnen lexikalischen Einheiten einer Sprache aussehen, bedient man sich eines weiteren Formalismus, den regulären Ausdrücken.

Andere Grammatiken

Die in diesem Kapitel vorgestellten Grammatiken heißen *kontextfrei*, weil eine Regel für ein Nichtterminalzeichen angewendet wird, ohne dabei zu betrachten, was vor oder nach dem Zeichen für ein weiteres Zeichen steht, der Kontext also nicht betrachtet wird. Läßt man auch Regeln zu, die auf der linken Seite nicht ein Nichtterminalzeichen stehen haben, so kann man mächtigere Sprachen beschreiben, als mit einer kontextfreien Grammatik.

Beispiel:
Wir können mit der folgenden nicht-kontextfreien Grammatik die Sprache beschreiben, in der jeder Satz gleich oft die beiden Terminalsymbole, eber in beliebiger Reihenfolge enthält.

- $\mathcal{T} = \{\text{a,b}\}$
- $\mathcal{N} = \{start, A, B\}$
- $S = start$
- $start ::= ABstart$
  start $::= \epsilon$
  $AB ::= BA$
  $BA ::= AB$
  $A ::= \text{a}$
  $B ::= \text{b}$

  $start$
  $\rightarrow ABstart$
  $\rightarrow ABABstart$
  $\rightarrow ABABABstart$

$\rightarrow ABABABABstart$
$\rightarrow ABABABABABstart$
$\rightarrow ABABABABAB$
$\rightarrow AABBABABAB$
$\rightarrow AABBBAABAB$
$\rightarrow AABBBABAAB$
$\rightarrow AABBBABABA$
$\rightarrow AABBBABBAA$
$\rightarrow AABBBBABAA$
$\rightarrow AABBBBBAAA$
$\rightarrow$ a$ABBBBBAAA$
$\rightarrow$ aa$BBBBBAAA$
$\rightarrow$ aab$BBBBAAA$
$\rightarrow$ aabb$BBBAAA$
$\rightarrow$ aabbb$BBAAA$
$\rightarrow$ aabbbb$BAAA$
$\rightarrow$ aabbbbbAAA
$\rightarrow$ aabbbbbaAA
$\rightarrow$ aabbbbbaa$A$
$\rightarrow$ aabbbbbaaa

Beispiel:
Auch die nicht durch eine kontextfreie Grammatik darstellbare Sprache:

$\{a^n b^n c^n | n \in I\!N\}$.

läßt sich mit einer solchen Grammatik generieren:

S::=ab$T$c
T::=A bTc$|\epsilon$
bA::=Ab
aA::=aa

Eine Ableitung mit dieser Grammatik sieht wie folgt aus:

S
$\rightarrow$ ab$T$c
$\rightarrow$ ab$A$bTcc
$\rightarrow$ ab$A$bAbTccc
$\rightarrow$ ab$A$bAbAbTcccc
$\rightarrow$ ab$A$bAbAbAbTccccc
$\rightarrow$ ab$A$bAbAbAbccccc
$\rightarrow$ abAAbb$A$bAbccccc
$\rightarrow$ abAAbAbb$A$bccccc
$\rightarrow$ abAAAbbb$A$bccccc
$\rightarrow$ abAAAbb$A$bbccccc
$\rightarrow$ abAAAbAbbbccccc
$\rightarrow$ ab$AAAA$bbbbccccc
$\rightarrow$ aAbAAAbbbbccccc
$\rightarrow$ aAAbAAbbbbccccc

$\rightarrow$ aAAAbAbbbbccccc
$\rightarrow$ a$AAAA$bbbbbccccc
$\rightarrow$ aaAAAbbbbbccccc
$\rightarrow$ aaaAAbbbbbccccc
$\rightarrow$ aaaa$A$bbbbbccccc
$\rightarrow$ aaaaabbbbbccccc

Als Preis dafür, daß man sich nicht auf kontextfreie Grammatiken beschränkt, kann nicht immer leicht und eindeutig erkannt werden, ob ein bestimmter vorgegebener Satz mit dieser Grammatik erzeugt werden kann.

Grammatiken und Bäume

Im letzten Kapitel haben wir uns ausführlich mit Bäumen beschäftigt. Eine Grammatik stellt in naheliegender Weise nicht nur ein Beschreibung einer Sprache dar, sondern jede Generierung eines Satzes dieser Sprache entspricht einem Baum, dem Ableitungsbaum. Die Knoten des Baumes sind mit Terminal- und Nichtterminalzeichen markiert, wobei Blätter mit Terminalzeichen markiert sind. Die Kinder eines Knotens sind die Knoten, die mit der rechten Seite einer Regelanwendung markiert sind.

Liest man die Blätter eines solchen Baumes von links nach rechts, so ergibt sich der generierte Satz.

Beispiel:
Die Ableitungen der Sätze unserer ersten Grammatik haben folgende Baumdarstellung:

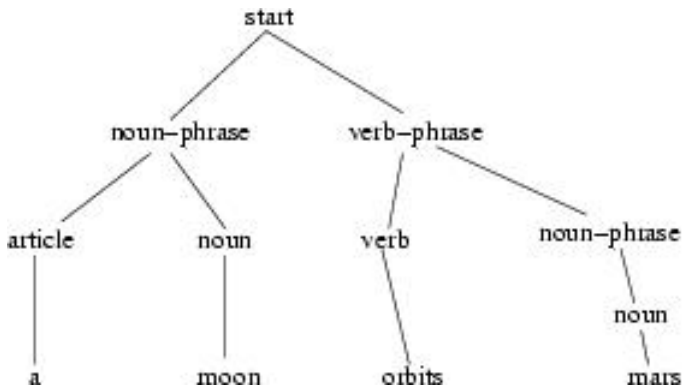


Abbildung 4.1: Ableitungsbaum für a moon orbits mars.

Gegenüber unserer bisherigen Darstellung der Ableitung eines Wortes mit den Regeln einer Grammatik, ist die Reihenfolge, in der die Regeln angewendet werden in der Baumdarstellung nicht mehr ersichtlich. Daher spricht man häufiger auch vom Syntaxbaum des Satzes.

Aufgabe 21 Betrachten Sie die einfache Grammatik für arithmetische Ausdrücke aus dem letzen Abschnitt. Zeichnen Sie einen Syntaxbaum für den Audruck `1+1+2*20`.

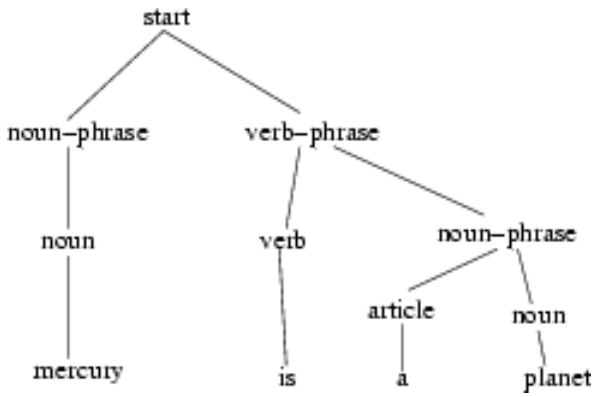


Abbildung 4.2: Ableitungsbaum für mercury is a planet.

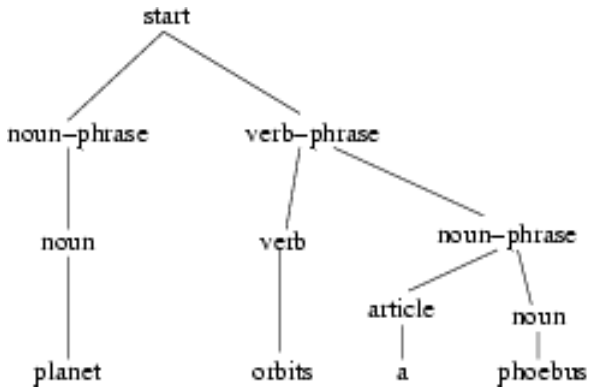


Abbildung 4.3: Ableitungsbaum für planet orbits a phoebus.

4.1.2 Erweiterte Backus-Naur-Form

Die Ausdrucksmöglichkeit einer kontextfreien Grammatik ist auf wenige Konstrukte beschränkt. Das macht das Konzept einfach. Im Kontext von Programmiersprachen gibt es häufig sprachliche Konstrukte, wie die mehrfache Wiederholung oder eine Liste von bestimmten Teilen, die zwar mit einer kontextfreien Grammatik darstellbar ist, für die aber spezielles zusätzliche Ausdrucksmittel in der Grammatik eingeführt werden. In den nächsten Abschnitten werden wir diese Erweiterungen kennenlernen, allerdings werden wir in unseren Algorithmen für Sprachen und Grammatiken diese Erweiterungen nicht berücksichtigen.

Alternativen

Wenn es für ein Nichtterminalzeichen mehrere Regeln gibt, so werden diese Regeln zu einer Regel umgestaltet. Die unterschiedlichen rechten Seiten werden dann durch einen vertikalen Strich | gestrennt.

Durch diese Darstellung verliert man leider den direkten Zusammenhang zwischen den Regeln und einen Ableitungsbaum.

Beispiel:
Die Regeln unserer ersten Grammatik können damit wie folgt geschrieben werden:

start ::= *noun-phrase verb-phrase*
noun-phrase ::= *noun*|*article noun*
verb-phrase ::= *verb noun-phrase*
noun ::= planet|moon|mars|deimos|phoebus
verb ::= orbits|is
article ::= a

Gruppierung

Bestimmte Teile der rechten Seite einer Grammatik können durch Klammern gruppiert werden. In diesen Klammern können wieder durch einen vertikalen Strich getrennte Alternativen stehen.

Beispiel:
Die einfache Grammatik für arithmetische Ausdrücke läßt sich damit ohne das Nichtterminalzeichen *op* schreiben:

start ::= *expr*
expr ::= *integer*|*integer* (+|−|∗|/) *expr*
integer ::= *digit*|*digit integer*
digit ::= 0|1|2|3|4|5|6|7|8|9

Wiederholungen

Ein typische Konstrukt in Programmiersprachen ist, daß bestimmte Kontrukte wiederholt werdn können. So stehen z.B. in Java im Rumpf einer Methode mehrere Anweisungen. Solche Sprachkonstrukte lassen sich mit einer kontextfreien Grammatik ausdrücken.

Beispiel:
Eine Zahl besteht aus einer Folge von n Ziffern ($n > 0$). Dieses läßt sich durch folgende Regel ausdrücken:

Zahl ::= *Ziffer Zahl* | *Ziffer*

1 bis n-fach Im obigen Beispiel handelt es sich um eine 1 bis n-fache Wiederholung des Zeichens *Ziffer*. Hierzu gibt es eine abkürzende Schreibweise. Dem zu wiederholenden Teil wird das Zeichen + nachgestellt.

Beispiel:
Obige Regel für das Nichtterminal *Zahl* läßt sich mit dieser abkürzenden Schreibweise schreiben als:

Zahl ::= *Ziffer*+

0 bis n-fach Soll ein Teil in einer Wiederholung auch keinmal vorkommen, so wird statt des Zeichens + das Zeichen * genommen.

> **Beispiel:**
> Folgende Regel drückt aus, daß ein Ausdruck eine durch Operatoren getrennte Liste von Zahlen ist.
>
> $$expr ::= Zahl \mid (Op\ Zahl)^*$$

Option Ein weiterer Spezialfall der Wiederholung ist die, in der der entsprechende Teil keinmal oder einmal vorkommen darf, d.h. der Teil ist optional. Optionale Teile werden in der erweiterten Form in eckige Klammern gesetzt.

4.2 Parser

Grammatiken geben an, wie Sätze einer Sprache gebildet werden können. In der Informatik interessiert der umgekehrte Fall. Ein Satz ist vorgegeben und es soll geprüft werden, ob dieser Satz mit einer bestimmten Grammatik erzeugt werden kann. Ein Javaübersetzer prüft z.B. ob ein vorgegebenes Programm syntaktisch zur durch die Javagrammatik beschriebenen Sprache gehört. Hierzu ist ein Prüfalgorithmus anzugeben, der testet, ob die Grammatik einen bestimmten Satz generieren kann. Ein solches Programm wird *Parser* genannt. Ein Parser[2] zerteilt einen Satz in seine syntaktischen Bestandteile.

4.2.1 Parsstrategien

Man kann unterschiedliche Algorithmen benutzen, um zu testen, daß ein zu der Sprache einer Grammatik gehört. Eine Strategie geht primär von der Grammatik aus, die andere betrachten eher den zu prüfenden Satz.

Rekursiv absteigend

Eine einfache Idee zum Schreiben eines Parsers ist es, einfach nacheinander auszuprobieren, ob die Regelanwendung nach und nach einen Satz bilden kann. Hierzu startet man mit dem Startsymbol und wendet nacheinander die Regeln an. Dabei wendet man immer eine Regel auf das linkeste Nichtterminalzeichen an, so lange, bis der Satzanfang abgelitten wurde, oder aber ein falscher Satzanfang abgelitten wurde. Im letzteren Fall hat man offensichtlich bei einer Regel die falsche Alternative gewählt. Man ist in eine Sackgasse geraten. Nun geht man in seiner Ableitung zurück bis zu der letzten Regelanwendung, an der eine andere Alternative hätte gewählt werden können. Man spricht dann von *backtracking*; auf deutsch kann man treffend von Zurückverfolgen sprechen.

Auf diese Weise gelangt man entweder zu einer Ableitung des Satzes, oder stellt irgendwann fest, daß man alle Alternativen ausprobiert hat und immer in eine Sackgasse geraten ist. Dann ist der Satz nicht in der Sprache.

Diese Strategie ist eine Suche. Es wird systematisch aus allen möglichen Ableitungen in der Grammatik nach der Ableitung gesucht, die den gewünschten Satz findet.

[2] Lateinisch Pars bedeutet Teil.

Beispiel:
Wir suchen mit dieser Strategie die Ableitung des Satzes *a moon orbits mars* in dem Sonnensystembeispiel. Sackgassen sind durch einen Punkt • markiert.

(0)		*start*	
(1) aus 0	→	*noun-phrase verb-phrase*	
(2a) aus 1	→	*noun verb-phrase*	
(2a3a) aus 2a	→	planet *verb-phrase*	•
(2a3b) aus 2a	→	moon *verb-phrase*	•
(2a3c) aus 2a	→	mars *verb-phrase*	•
(2a3d) aus 2a	→	deimos *verb-phrase*	•
(2a3e) aus 2a	→	phobus *verb-phrase*	•
(2b) aus 1	→	*article noun verb-phrase*	
(3) aus 2b	→	a *noun verb-phrase*	
(4a) aus 3	→	a planet *verb-phrase*	•
(4b) aus 3	→	a moon *verb-phrase*	
(5) aus 4b	→	a moon *verb noun-phrase*	
(6) aus 5	→	a moon orbits *noun-phrase*	
(7) aus 6	→	a moon orbits *noun*	
(8a) aus 7	→	a moon orbits planet	•
(8b) aus 7	→	a moon orbits moon	•
(8c) aus 7	→	a moon orbits mars	

Alternativ könnte man diese Strategie auch symetrisch nicht von links sondern von der rechten Seite an ausführen, also immer eine Regel für das rechtestes Nichtterminalsymbol anwenden.

Linksrekursion Die Strategie des rekursiven Abstiegs auf der linken Seite funktioniert für eine bestimmte Art von Regeln nicht. Dieses sind Regeln, die in einer Alternative als erstes Symbol wieder das Nichtterminalsymbol stehen haben, das auch auf der linken Seite steht. Solche Regeln heißen linksrekursiv. Unsere Strategie terminiert in diesen Fall nicht.

Beispiel:
Gegeben sei eine Grammatik mit einer linksrekursiven Regel:

expr::=*expr*+*zahl*|*zahl*

Der Versuch den Satz

zahl+zahl

mit einem links rekursiv absteigenden Parser abzuleiten, führt zu einem nicht terminierenden rekursiven Abstieg. Wir gelangen nie in eine Sackgasse:

$$
\begin{array}{ll}
 & expr \\
\rightarrow & expr{+}zahl \\
\rightarrow & expr{+}zahl{+}zahl \\
\rightarrow & expr{+}zahl{+}zahl{+}zahl \\
\rightarrow & expr{+}zahl{+}zahl{+}zahl{+}zahl \\
 & \ldots
\end{array}
$$

Es läßt sich also nicht für alle Grammatiken mit dem Verfahren des rekursiven Abstiegs entscheiden, ob ein Satz mit der Grammatik erzeugt werden kann; aber es ist möglich, eine Grammatik mit linksrekursiven Regeln so umzuschreiben, daß sie die gleiche Sprache generiert, jedoch nicht mehr linksrekursiv ist. Hierzu gibt es ein einfaches Schema:

Eine Regel nach dem Schema:

$A ::= A\ rest|alt2$

ist zu ersetzen durch die zwei Regeln:

$A ::= alt2\ R$
$R ::= rest\ R|\epsilon$

wobei R ein neues Nichtterminalsymbol ist.

Linkseindeutigkeit Die rekursiv absteigende Strategie hat einen weiteren Nachteil. Wenn sie streng schematisch angewendet wird, führt sie dazu, daß bestimmte Prüfungen mehrfach durchgeführt werden. Diese mehrfache Ausführung kann sich bei wiederholter Regelanwendung multipizieren und zu einem sehr ineffizienten Parser führen. Grund dafür sind Regeln, die zwei Alternativen mit gleichem Anfang haben:

$S ::= AB|A$

Beide Alternativen für das Nichtterminalzeichen S starten mit dem Zeichen A. Für unseren Parser bedeutet das soweit: versuche nach der ersten Alternative zu parsen. Hierzu parse erst nach dem Symbol A. Das kann eine sehr komplexe Berechnung sein. Wenn sie gelingt, dann versuche anschließend weiter nach dem Symbol B zu parsen. Wenn das fehlschlägt, dann verwerfe die Regelalternative und versuche nach der zweiten Regelalternative zu parsen. Jetzt ist wieder nach dem Symbol A zu parsen, was wir bereits gemacht haben.

Regeln der obigen Art wirken sich auf unsere Parsstrategie ungünstig aus. Wir können aber ein Schema angegeben, wie man solche Regeln aus der Grammatik eleminiert, ohne die erzeugte Sprache zu ändern:

Regeln der Form

$S ::= AB|A$

sind zu ersetzen durch

$S ::= AT$
$T ::= B|\epsilon$

wobei T ein neues Nichtterminalzeichen ist.

Vorausschau Im letzten Abschnitt haben wir gesehen, wie wir die Grammatik umschreiben können, so daß nach dem Zurücksetzen in unserem Algorithmus es nicht vorkommt, bereits ausgeführte Regeln ein weiteres Mal zu durchlaufen. Schöner noch wäre es, wenn wir auf das Zurücksetzen ganz verzichten könnten. Dieses läßt sich allgemein nicht erreichen, aber es gibt Grammatiken, in denen man durch Betrachtung des nächsten zu parsenden Zeichens erkennen kann, welche der Regelalternativen als einzige Alternative in betracht kommt. Hierzu kann man für eine Grammatik für jedes Nichtterminalzeichen in jeder Regelalternative berechnen, welche Terminalzeichen als linkestes Zeichen in einem mit dieser Regel abgelittenen Satz auftreten kann. Wenn die Regelalternativen disjunkte solche Menge des ersten Zeichens haben, so ist eindeutig bei Betrachtung des ersten Zeichens eines zu parsenden Satzes erkennbar, ob und mit welcher Alternative dieser Satz nur parsbar sein kann.

Die gängigsten Parser benutzen diese Entscheidung, nach dem erstem Zeichen. Diese Parser sind darauf angewiesen, daß die Menge der ersten Zeichen der verschiedenen Regelalternativen disjunkt sind.

Der Vorteil an diesem Verfahren ist, daß die Token nach und nach von links nach rechts stückweise konsumiert werden. Sie können durch einen Datenstro, relisiert werden. Wurden sie einmal konsumiert, so werden sie nicht mehr zum Parsen benötigt, weil es kein Zurücksetzen gibt.

Schieben und Reduzieren

Der rekursiv absteigende Parser geht vom Startsymbol aus und versucht durch Regelanwendung den in Frage stehenden Satz abzuleiten. Eine andere Strategie ist die des Schiebens-und-Reduzierens (shift-reduce). Diese geht vom im Frage stehenden Satz aus und versucht die Regeln rückwärts anzuwenden, bis das Startsymbol erreicht wurde. Hierzu wird der Satz von links nach rechts (oder symmetrisch von rechts nach links) betrachtet und versucht, rechte Seiten von Regeln zu finden und durch ihre linke Seite zu ersetzen. Dazu benötigt man einen Marker, der angibt, bis zu welchen Teil man den Satz betrachtet. Wenn links des Markers keine linke Seite einer Regel steht, so wird der Marker ein Zeichen weiter nach rechts verschoben.

Beispiel:
Wir leiten im folgenden unserer allseits bekannten Beispielsatz aus dem Sonnensystem durch Schieben und Reduzieren ab. Als Marker benutzen wir einen Punkt.

		. a moon orbits mars
(shift)	→	a . moon orbits mars
(reduce)	→	*article* . moon orbits mars
(shift)	→	*article* moon . orbits mars
(reduce)	→	*article noun* . orbits mars
(reduce)	→	*noun-phrase* . orbits mars
(shift)	→	*noun-phrase* orbits . mars
(reduce)	→	*noun-phrase verb* . mars
(shift)	→	*noun-phrase verb* mars .
(reduce)	→	*noun-phrase verb noun* .
(reduce)	→	*noun-phrase verb noun-phrase* .
(reduce)	→	*noun-phrase verb-phrase* .
(reduce)	→	*start*.

Wir werden im Laufe dieser Vorlesung diese Parserstrategie nicht weiter verfolgen.

4.3 Handgeschriebene Parser

Genug der Theorie. Wir kennen jetzt Grammatiken und zwei Strategien, um für einen Satz zu entscheiden, ob er mit einer Grammatik generiert werden kann. In diesem Abschnitt wollen wir unsere theoretischen Erkenntnisse in ein Programm praktisch umsetzen. Wir werden hierzu einen rekursiv absteigenden Parser entwickeln.

4.3.1 Basisklassen der Parserbibliothek

Wir werden eine Bibliothek entwerfen, mit der Parser einer beliebigen Grammatik konstruiert werden können. Parser werden dabei Objekte sein, insbesondere mit einer Methode zum Parsen. Entwerfen wir also zunächst eine Schnittstelle, die einen Parser beschreibt. Die Schnittstelle für Parser soll genau eine Methode haben, die Methode `parse`. Sie bekommt ein Argument, die Liste der Token, die zu parsen sind. Ein Parser erzeugt ein Ergebnis. Das Ergebnis kann von beliebigen Typen sein: so gibt es Parser, die einen Baum als Ergebnis produzieren, oder Parser, die nur als einen bool'schen Wert angeben, ob der Parsevorgang erfolgreich war. Um diesem Rechnung zu tragen, lassen wir in der Schnittstelle `Parser` den Ergebnistyp zunächst variabel. Wir erhalten folgende generische Schnittstelle:

Parser.java

```
1 package name.panitz.parser;
2 import java.util.List;
3 public interface Parser<a> {
4   ParseResult<a> parse(List<Token> ts);
5 }
```

Für einen beliebigen aber festen Typen `a` können wir Parser implementieren: so ist der Typ `Parser<String>` ein Parser, der ein Stringobjekt als Ergebnis liefert.

Wir haben in der obigen Definition der Parserschnittstelle einen Typ `Token` vorgesehen. Auch für diesen sehen wir eine Schnittstelle vor:

Token.java

```
1 package name.panitz.parser;
2 public interface Token {}
```

Die Schnittstelle ist leer. Sie hat keine Eigenschaften. Der Einfachheit haben wir als einen Strom von Token eine Liste von Token verwenden.

Schließlich haben wir einen Typ `ParseResult<a>` in der Parserschnittstelle vorgesehen. Das Ergebnis des Parsevorgangs eines Parser des Typs `Parser<a>` ist vom Typ `a`. Es reicht aber nicht aus, allein das Parseergebnis zurückzugeben. Wir müssen zusätzlich angeben, welche Token durch den Parsvorgang konsumiert wurden. Daher haben wir einen Typ `ParseResult<a>` vorgesehen, der aus zwei Teilen besteht:

- dem eigentlichen Ergebnis vom Typ `a`.

- der Liste der durch den Parsvorgang nicht konsumierten restlichen Token.

So läßt sich folgende Klasse für Parsergebnisse definieren:

ParseResult.java

```
1  package name.panitz.parser;
2  import java.util.List;
3  public class ParseResult<a> {
4    final private a result;
5    final private List<Token> remainingToken;
6    public ParseResult(a r,List<Token> toks){
7      result=r;remainingToken=toks;
8    }
9    public a getResult(){return result;}
10   public List<Token> getRemainingToken(){return remainingToken;}
11   public boolean failed(){return result==null;}
12 }
```

Wir haben für die zwei Teile eines Parsergebnisse jeweils eine `get`-Methode definiert. Die Methode `failed` gibt an ob es sich um einen erfolgreichen Parsvorgang gehandelt hat oder nicht.

Der Epsilon Parser

Wir können schon einen allerersten Parser schreiben, den Epsilonparser, der kein Token der Eingabe konsumiert. Er endet immer erfolgreich. Wir lassen ihm daher immer den Wert `true` als Ergebnis zurückgeben. Kein Token der Eingabeliste wird konsumiert.

Epsilon.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class Epsilon implements Parser<Boolean>{
5    public ParseResult<Boolean> parse(List<Token> ts){
6       return new ParseResult<Boolean>(true,ts);
7    }
8  }
```

4.3.2 Parser zum Erkennen von Token

Der allgemeine Rahmen für Parser ist nun abgesteckt. Jetzt können wir schrittweise für komplexe Grammatiken Parser bauen. Der elementarste Parser ist der, der prüft, ob ein bestimmtes Token als nächstes in der Eingabeliste folgt. Hierzu ist eine Klasse, die `Parser` implementiert, zu definieren, in der die Methode `parse` entsprechend implementiert ist. In der Methode `parse` ist zunächst zu prüfen, ob die Tokenliste nicht leer ist. Dann ist zu prüfen, ob das nächste Token ein bestimmtes Token, nach dem der Parser sucht, gleicht. Im Erfolgsfall wird ein erfolgreiches Ergebnis mit dem gefundenen Token erzeugt.

ParseToken.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class ParseToken<a extends Token> implements Parser<a>{
5    final private a t;
6    public ParseToken(a _t){t=_t;}
7    public ParseResult<a> parse(List<Token> ts){
8      if (ts.isEmpty()) return new ParseResult<a>(null,ts);
9      if (ts.get(0).equals(t))
10       return new ParseResult<a>((a)ts.get(0),ts.subList(1,ts.size()));
11     return new ParseResult<a>(null,ts);
12   }
13 }
```

4.3.3 Parser zum Bilden der Sequenz zweier Parser

In einfachen Chomsky-Grammatiken gibt es nur zwei Arten wie für ein Nichtterminalsymbol Regeln gebildet werden:

- durch die Sequenz mehrer Symbole auf der rechten Seite einer Regel.
- durch mehrere Regelalternativen.

Für diese beiden Bildungsprinzipien schreiben wir Parserobjekte, die aus jeweils zwei Parser einen neuen Parser bilden.

In diesen Abschnitt machen wir das zunächst für die Sequenz zweier Parser. Haben wir in der Grammatik eine Regel der Form:

$$A ::= B \;\; C$$

So können wir sie beim Schreiben eines Parser lesen als: Um den Parser für das Nichtterminalsymbol `A` zu schreiben, schreibe die Parser für die Symbole B und C und konstruiere mit diesen beiden den Parser für das Symbol A durch den Sequenzparser aus den beiden Teilparsern.

Das Ergebnis einer Sequenz von zwei Parsern muß die beiden Parsergebnisse irgendwie sinnvoll kombinieren. Hierzu sehen wir eine Klasse vor, die Paare von zwei Objekten repräsentieren kann:

Pair.java

```
1  package name.panitz.parser;
2
3  public class Pair<a,b>{
4    final private a e1;
5    final private b e2;
6    public Pair(a x1,b x2){e1=x1;e2=x2;}
7    public a getE1(){return e1;}
8    public b getE2(){return e2;}
9  }
```

Die Sequenz zweier Parser kann jetzt als Parsergebnis ein Paar aus den beiden Teilergebnissen haben. Wir implementieren den Parser, der die Sequenz zweier Parser darstellt. Die zwei Teilparser werden dem Konstruktor übergeben:

Seq.java
```
1  package name.panitz.parser;
2
3  import java.util.List;
4
5  public class Seq<a,b> implements Parser<Pair<a,b>>{
6    final private Parser<a> p1;
7    final private Parser<b> p2;
8    public Seq(Parser< a> _p1,Parser< b> _p2){p1=_p1;p2=_p2;}
```

Die Methode `parse` wendet zunächst den ersten Parser auf die Tokenliste an:

Seq.java
```
9    public ParseResult<Pair<a,b>> parse(List<Token> ts){
10     ParseResult<a> r1 = p1.parse(ts);
```

Sollte dieses schon mißglücken, so ist das Gesamtergebnis auch ein mißglückter Parsversuch:

Seq.java
```
11     if (r1.failed()) return new ParseResult<Pair<a,b>>(null,ts);
```

Im erfolgreichen Fall sind die übrigen noch nicht konsumierten Token des ersten Parses zu nehmen und der zweite Parser auf diese anzuwenden:

Seq.java
```
12     ParseResult<b> r2 = p2.parse(r1.getRemainingToken());
```

Sollte dieses schon mißglücken, so ist das Gesamtergebnis auch ein mißglückter Parsversuch:

Seq.java
```
13     if (r2.failed()) return new ParseResult<Pair<a,b>>(null,ts);
```

Sind schließlich beide Parser erfolgreich gewesen ist ein erfolgreiches Ergebnis als Paar der beiden Teilergebnisse zu bilden.

Seq.java
```
14     return new ParseResult<Pair<a,b>>
15                   (new Pair<a,b>(r1.getResult(),r2.getResult())
16                   ,r2.getRemainingToken());
17   }
18 }
```

4.3.4 Parser zum Bilden der Alternative zweier Parser

Das zweite Bildungsprinzip, um für eine Grammatik einen Parser zu bauen, ist die alternative Anwendung zweier bestehender Parser. In einer einfachen Chomskygrammatik stellt sich dieses durch zwei Regelalternativen dar.

Haben wir in der Grammatik zwei Regeln der Form:

$A ::= B$
$A ::= C$

So können wir diese lesen als: um das Nichtterminal `A` zu parsen, versuche den Parser für B, sollte dieses mißglücken versuche mit dem Parser für C zu Parsen.

Entsprechend der Klasse `Seq` für die Sequenz von zwei Parsern definieren wir die Klasse `Alt` für die Alternative aus zwei Parsern:

Alt.java

```
1  package name.panitz.parser;
2
3  import java.util.List;
4  import java.util.ArrayList;
5
6  public class Alt<a> implements Parser<a>{
7    final private Parser<a> p1;
8    final private Parser<a> p2;
9    public Alt(Parser<a> _p1,Parser<a> _p2){p1=_p1;p2=_p2;}
```

Zum Parsen der Alternative zweier Parser wird erst der erste der beiden ausprobiert:

Alt.java

```
10   public ParseResult<a> parse(List<Token> ts){
11     ParseResult<a> r1 = p1.parse(ts);
```

Sollte dieses mißglücken, so ist der zweite Parser auf derselben Tokenliste anzuwenden, andernfalls ist das erfolgreiche Ergebnis des ersten Parses zurückzugeben.

Alt.java

```
12     if (r1.failed()) return p2.parse(ts);
13     return r1;
14   }
15 }
```

4.3.5 Parser zum Verändern des Ergebnisses

Unsere kleine Parserbibliothek kann bisher drei Dinge:

- elementare Parser definieren, die bestimmte Token erkennen
- aus zwei Parsern die Sequenz bilden, die als Ergebnis das Paar der zwei Tailparses hat.

- aus zwei Parsern einen neuen Parser bilden, der einen der beiden Parser anzuwendet.

Oft will man etwas mit dem Ergebnis anstellen, nachdem ein Parser es erfolgreich erkannt hat. Man möchte also auf das Ergebnis eine Funktion anwenden. Hierzu definieren wir uns eine Schnittstelle, die zunächst einmal beschreibt, wie Funktionsobjekte definiert sind.

Function.java

```
1  package name.panitz.parser;
2  public interface Function<a,b>{
3      b apply(a x);
4  }
```

Unsere letzte Klasse zum Bilden von Parsern, beschreibt einen Parser, der zunächst einen vorgegebenen Parser auf die Tokenliste anwendet und anschließend auf das erhaltene Ergebnis eine gegebene Funktion anwendet.

Map.java

```
1  package name.panitz.parser;
2  import java.util.List;
3  import java.util.ArrayList;
4
5  public class Map<a,b> implements Parser<b>{
6    final private Parser<a> p1;
7    final private Function<a,b> f;
8    public Map(Parser<a> _p1,Function<a,b> _f){p1=_p1;f=_f;}
```

Zum Parsen wird zunächst der gegebene Parser auf die Tokenliste angewendet:

Map.java

```
9    public ParseResult<b> parse(List<Token> ts){
10     ParseResult<a> r1 = p1.parse(ts);
```

Im Erfolgsfall wird ein neues Ergebnis gebaut, indem auf die Ergebiskomponente die mitgegebene Funktion angewendet wird.

Map.java

```
11     if (r1.failed()) return new ParseResult<b>(null,ts);
12     return new ParseResult<b>
13       (f.apply(r1.getResult()),r1.getRemainingToken());
14   }
15 }
```

Unsere Parserbibliothek ist vollständig. In den nächsten Abschnitten sehen wir Beispiele, wie sie zum Schreiben von Parsern benutzt werden kann.

4.3.6 Beispiel: arithmetische Ausdrücke

Als Beispiel betrachten wir eine Grammatik, die einfache arithmetische Ausdrücke generiert.

Grammatik:

start ::= *addExpr*
addExpr ::= *multExpraddOp addExpr*|*multExpr*
multExpr ::= zahl *multOp multExpr*| zahl
multOp ::= *|/
addOp ::= +|-

Token

In unserer obigen Grammatik gibt es 5 verschiedene Token. Die vier Operatoren und Zahlen. Für Zahlen können wir eine einfache Tokenklasse definieren, die diese beschreibt.

Zahl.java

```
1 package name.panitz.parser;
2 public class Zahl implements Token{
3   final public int i;
4   public Zahl(int _i){i=_i;}
5   public boolean equals(Object o){return o instanceof Zahl;}
6 }
```

Die vier Operatoren können wir als Aufzählung realisieren:

OpToken.java

```
1 package name.panitz.parser;
2 public enum OpToken implements Token{
3  add,sub,mult,div;
```

Jeder Operator entspricht einer zweistelligen Funktion. Auch dieses können wir mit Hilfe einer Schnittstelle ausdrücken:

BinFunction.java

```
1 package name.panitz.parser;
2
3 public interface BinFunction<a,b,c>{
4   c apply(a x,b y);
5 }
```

Die Operatoren sollen spezielle zweistellige Funktionen auf ganzen Zahlen sein:

IntOp.java

```
1 package name.panitz.parser;
2
3 public interface IntOp
4          extends BinFunction<Integer,Integer,Integer>{}
```

Nun können wir in der Aufzählungsklasse der Operatortoken noch eine Methode vorsehen, die zu dem entsprechenden Operator die entsprechende Funktion als Ergebnis zurückgibt:

OpToken.java

```
5
6   public IntOp getFunction(){
7     switch (this){
8       case div : return new IntOp(){
9            public Integer apply(Integer x,Integer y){return x/y;}};
10      case mult : return new IntOp(){
11           public Integer apply(Integer x,Integer y){return x*y;}};
12      case add : return new IntOp(){
13           public Integer apply(Integer x,Integer y){return x+y;}};
14      default : return new  IntOp(){
15           public Integer apply(Integer x,Integer y){return x-y;}};
16    }
17   }
18 }
```

AuswertungsParser

Wir schreiben einen ersten Parser, der für einen arithmetischen Ausdruck seinen ausgewerteten Wert als Ergenis hat. Für jedes Symbol der Klasse schreiben wir einen Parser, der eine ganze Zahl als Ergebnis hat. Wir fangen mit den Terminalsymbolen an. Zunächst der Parser für Zahlen. Hier handelt ich sich um den Tokenparser für das Token `Zahl`, auf den die Funktion, die die eigentliche Zahl extrahiert angewendet wird.

ZahlEval.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class ZahlEval implements Parser<Integer>{
5    public ParseResult<Integer> parse(List<Token> ts){
6      return new Map<Zahl,Integer>
7          (new ParseToken<Zahl>(new Zahl(0))
8          ,new Function<Zahl,Integer>(){
9             public Integer apply(Zahl z){return z.i;}}
10         ).parse(ts);
11   }
12 }
```

Als nächstes kümmern wir uns um die Operatoren. In einem Funktionsobjekt kapseln wir die Funktion, die für ein Operatortoken die entsprechende zweistellige Funktion extrahiert:

GetIntOp.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class GetIntOp implements Function<OpToken,IntOp>{
5    public IntOp apply(OpToken t){
6      return t.getFunction();
7    }
8  }
```

Der Parser für Punktrechnungsoperatoren besteht aus der Alternative der beiden Tokenparser mit anschließender Extraktion der binären Funktion:

MultOpEval.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class MultOpEval implements Parser<IntOp>{
5    public ParseResult<IntOp> parse(List<Token> ts){
6      return new Map<OpToken,IntOp>
7        (new Alt<OpToken>
8           (new ParseToken<OpToken>(OpToken.mult)
9           ,new ParseToken<OpToken>(OpToken.div))
10       ,new GetIntOp()).parse(ts);
11   }
12 }
```

Analog funktioniert der Parser für die Strichrechnung:

AddOpEval.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class AddOpEval implements Parser<IntOp>{
5    public ParseResult<IntOp> parse(List<Token> ts){
6      return new Map<OpToken,IntOp>
7        (new Alt<OpToken>
8           (new ParseToken<OpToken>(OpToken.add)
9           ,new ParseToken<OpToken>(OpToken.sub))
10       ,new GetIntOp()).parse(ts);
11   }
12 }
```

Als nächstes folgt der Parser für das Nichtterminalsymbol *multExpr*.

multExpr ::= zahl *multOp multExpr*| zahl

Dieses ist entweder eine Sequenz aus `Zahl`*multOp multExpr* oder nur eine `Zahl`. Die Sequenz hat als Ergebnistyp doppelt verschachtelte Paare des Typs: `Pair<Integer,Pair<IntOp,Integer>>`. Aus diesen Typ ist eine ganze Zahl zu bilden. Hierfür schreiben wir die Klasse `DoIntOp`:

DoIntOp.java

```
1  package name.panitz.parser;
2
3  public class DoIntOp
4     implements Function<Pair<Integer,Pair<IntOp,Integer>>,Integer>{
5    public Integer apply(Pair<Integer,Pair<IntOp,Integer>> x){
6      return x.getE2().getE1().apply(x.getE1(),x.getE2().getE2());
7    }
8  }
```

Jetzt können wir entsprechend der Grammatik einen Parser für das Nichtterminalsymbol *multExpr* schreiben:

MultExprEval.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class MultExprEval implements Parser<Integer>{
5    public ParseResult<Integer> parse(List<Token> ts){
6      return
7        new Alt<Integer>
8         (new Map<Pair<Integer,Pair<IntOp,Integer>>,Integer>
9           (new Seq<Integer,Pair<IntOp,Integer>>
10            (new ZahlEval(),new Seq<IntOp,Integer>
11                            (new MultOpEval(),new MultExprEval()))
12          ,new DoIntOp()
13          )
14         ,new ZahlEval()
15         ).parse(ts);
16   }
17 }
```

Analog läßt sich der Parser für das Nichtterminalsymbol *addExpr* schreiben:

$$addExpr ::= multExpr addOp\ addExpr | multExpr$$

AddExprEval.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class AddExprEval implements Parser<Integer>{
5    public ParseResult<Integer> parse(List<Token> ts){
6      return
7        new Alt<Integer>
8         (new Map<Pair<Integer,Pair<IntOp,Integer>>,Integer>
9           (new Seq<Integer,Pair<IntOp,Integer>>
10            (new MultExprEval(),new Seq<IntOp,Integer>
11                            (new AddOpEval(),new AddExprEval()))
12          ,new DoIntOp()
13          )
14         ,new MultExprEval()
15         ).parse(ts);
16   }
17 }
```

Damit sind wir fertig. *addExpr* ist das Startsymbol der Grammatik.

EvalArith.java

```
1  package name.panitz.parser;
2  import java.util.List;
```

```
3
4  public class EvalArith implements Parser<Integer>{
5    Parser<Integer> startExpr = new AddExprEval();
6
7    public ParseResult<Integer> parse(List<Token> ts){
8      return startExpr.parse(ts);
9    }
10 }
```

Im folgenden eine kleine Testklasse für unseren ersten handgeschriebenen Parser:

TestEvalArith.java

```
1  package name.panitz.parser;
2  import java.util.List;
3  import java.util.ArrayList;
4
5  public class TestEvalArith {
6
7    public static void main(String[] _){
8      List<Token> ts = new ArrayList<Token>();
9      ts.add(new Zahl(17));
10     ts.add(OpToken.mult);
11     ts.add(new Zahl(2));
12     ts.add(OpToken.add);
13     ts.add(new Zahl(8));
14
15     ParseResult<Integer> res = new EvalArith().parse(ts);
16     System.out.println(res.getResult());
17     System.out.println(res.getRemainingToken());
18   }
19 }
```

Aufbau eines Parsbaumes

In diesen Abschnitt schreiben wir einen zweiten Parser für die Grammatik der arithmetischen Ausdrücke. Dieses mal liefert der Parser einen Baum als Ergebnis.

Für jedes Symbol der Sprache schreiben wir wiesder genau eine Klasse. Die Klasse unterscheiden sich von den Parserklassen des vorherigen Parser nur in den Funktionsobjekten, die das Ergebnis berechnen.

ZahlTree.java

```
1  package name.panitz.parser;
2  import name.panitz.data.tree.*;
3  import java.util.List;
4
5  public class ZahlTree implements Parser<Tree<String>>{
6    public ParseResult<Tree<String>> parse(List<Token> ts){
7      return new Map<Zahl,Tree<String>>
8          (new ParseToken<Zahl>(new Zahl(0))
```

```
        ,new Function<Zahl,Tree<String>>(){
           public Tree<String> apply(Zahl z){
             return new Tree<String>(""+z.i);}}
        ).parse(ts);
  }
}
```

OpTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

public interface OpTree
    extends BinFunction<Tree<String>,Tree<String>,Tree<String>>{}
```

GetOpTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

import java.util.List;
import java.util.ArrayList;

public class GetOpTree implements Function<OpToken,OpTree>{
  public OpTree apply(final OpToken t){
    return
      new OpTree(){
        public Tree<String> apply(Tree<String>op1,Tree<String> op2){
          List<Tree<String>> cs = new ArrayList<Tree<String>>();
          cs.add(op1);
          cs.add(op2);
          return new Tree<String>(t.toString(),cs);
        }
      };
  }
}
```

MultOpTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

import java.util.List;

public class MultOpTree implements Parser<OpTree>{
  public ParseResult<OpTree> parse(List<Token> ts){
    return new Map<OpToken,OpTree>
      (new Alt<OpToken>
         (new ParseToken<OpToken>(OpToken.mult)
         ,new ParseToken<OpToken>(OpToken.div))
      ,new GetOpTree()).parse(ts);
```

```
  }
}
```

AddOpTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

import java.util.List;

public class AddOpTree implements Parser<OpTree>{
  public ParseResult<OpTree> parse(List<Token> ts){
    return new Map<OpToken,OpTree>
      (new Alt<OpToken>
         (new ParseToken<OpToken>(OpToken.add)
         ,new ParseToken<OpToken>(OpToken.sub))
      ,new GetOpTree()).parse(ts);
  }
}
```

DoOpTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

public class DoOpTree
 implements  Function<Pair<Tree<String>,Pair<OpTree,Tree<String>>>
                      ,Tree<String>> {
  public Tree<String> apply
                    (Pair<Tree<String>,Pair<OpTree,Tree<String>>> x){
    return x.getE2().getE1().apply(x.getE1(),x.getE2().getE2());
  }
}
```

MultExprTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

import java.util.List;

public class MultExprTree implements Parser<Tree<String>>{
  public ParseResult<Tree<String>> parse(List<Token> ts){
    return
      new Alt<Tree<String>>
       (new Map<Pair<Tree<String>,Pair<OpTree,Tree<String>>>,Tree<String>>
         (new Seq<Tree<String>,Pair<OpTree,Tree<String>>>
           (new ZahlTree(),new Seq<OpTree,Tree<String>>
                           (new MultOpTree(),new MultExprTree()))
         ,new DoOpTree()
         )
```

```
16        ,new ZahlTree()
17        ).parse(ts);
18    }
19  }
```

AddExprTree.java

```
1  package name.panitz.parser;
2  import name.panitz.data.tree.*;
3
4  import java.util.List;
5
6  public class AddExprTree implements Parser<Tree<String>>{
7    public ParseResult<Tree<String>> parse(List<Token> ts){
8      return
9        new Alt<Tree<String>>
10       (new Map<Pair<Tree<String>,Pair<OpTree,Tree<String>>>,Tree<String>>
11         (new Seq<Tree<String>,Pair<OpTree,Tree<String>>>
12           (new MultExprTree(),new Seq<OpTree,Tree<String>>
13                               (new AddOpTree(),new AddExprTree()))
14         ,new DoOpTree()
15         )
16       ,new MultExprTree()
17       ).parse(ts);
18   }
19 }
```

TreeArith.java

```
1  package name.panitz.parser;
2  import java.util.List;
3  import name.panitz.data.tree.*;
4
5  public class TreeArith implements Parser<Tree<String>>{
6    Parser<Tree<String>> startExpr = new AddExprTree();
7
8    public ParseResult<Tree<String>> parse(List<Token> ts){
9      return startExpr.parse(ts);
10   }
11 }
```

TestTreeArith.java

```
1  package name.panitz.parser;
2  import name.panitz.data.tree.*;
3  import java.util.List;
4  import java.util.ArrayList;
5  import javax.swing.*;
6
7  public class TestTreeArith {
8
```

```
9   public static void main(String[] _){
10    List<Token> ts = new ArrayList<Token>();
11    ts.add(new Zahl(17));
12    ts.add(OpToken.mult);
13    ts.add(new Zahl(2));
14    ts.add(OpToken.add);
15    ts.add(new Zahl(8));
16    ts.add(OpToken.mult);
17    ts.add(new Zahl(2));
18    ts.add(OpToken.div);
19    ts.add(new Zahl(1));
20    ts.add(OpToken.add);
21    ts.add(new Zahl(1));
22    ts.add(OpToken.sub);
23    ts.add(new Zahl(1));
24    ParseResult<Tree<String>> res = new TreeArith().parse(ts);
25    JFrame f = new JFrame();
26    f.getContentPane().add(new DisplayTree(res.getResult()));
27    f.pack();
28    f.setVisible(true);
29    System.out.println(res.getRemainingToken());
30  }
31 }
```

4.3.7 Tokenizer

Wir wollen wenigstens für die Grammatik der arithmetischen Ausdrücke einen Tokenizer schreiben, so daß wir eine Anwendung schreiben können, die eine Datei liest, deren Inhalt als arithmetischen Ausdruck parst und in einer Bilddatei den Ableitungsbaum darstellt.

Wir sehen eine Klasse vor, die Tokenizerobjekte für die Token in arithmethischen Ausdrücken darstellt:

ArithTokenizer.java

```
1 package name.panitz.parser;
2 import java.util.List;
3 import java.util.ArrayList;
4 import java.io.Reader;
5 import java.io.StringReader;
6 import java.io.IOException;
7
8 class ArithTokenizer {
```

Ein Tokenizer soll auf einen Eingabestrom des Typs `Reader` operieren. Ein Feld soll das aktuelle aus diesem Strom untersuchte Zeichen enthalten, und eine Liste die bisher gefundenen Token:

ArithTokenizer.java

```
9   Reader reader;
10  int next=0;
11  List<Token> result = new ArrayList<Token>();
```

Wir sehen einen Konstruktor vor, der den Strom, aus dem gelesen wird übergeben bekommt, und das erste Zeichen aus diesem Strom einliest:

ArithTokenizer.java

```
12  ArithTokenizer(Reader reader){
13    this.reader=reader;
14    try {
15      next= reader.read();
16    }catch (IOException _){}
17  }
```

Die entscheidene Methode `tokenize` liest nach und nach die Zeichen aus dem Eingabestrom und erzeugt ja nach gelesenen Zeichen ein Token für die Ergebnistokenliste:

ArithTokenizer.java

```
18  List<Token> tokenize() throws Exception{
19    try {
20      while (next>=0){
21        char c = (char)next;
22        switch (c){
```

Für Leerzeichen wird kein Token erzeugt:

ArithTokenizer.java

```
23          case '\u0020'  : break;
24          case '\n' : break;
25          case '\t' : break;
```

Token, die aus einen Zeichen bestehen, erzeugen sobald das Zeichen erkannt wurde ein Tokenobjekt für die Ausgabeliste:

ArithTokenizer.java

```
26          case '+'  : result.add(OpToken.add) ; break;
27          case '-'  : result.add(OpToken.sub) ; break;
28          case '*'  : result.add(OpToken.mult); break;
29          case '/'  : result.add(OpToken.div) ; break;
```

Wenn keines der obigen Zeichen gefunden wurde, kann es sich nur noch um den Anfang einer Zahl, oder um eine fehlerhafte Eingabe handeln. Wir unterscheiden, ob das Zeichen eine Ziffer darstellt, und versuchen eine Zahl zu lesen, oder werfen eine Ausnahme:

ArithTokenizer.java

```
30          default   : if (Character.isDigit(c)) {
31                        result.add(getInt());
32                        continue;
33                      }else throw new Exception
34                            ("unexpected Token found: '"+c+"'");
35        }
```

Schließlich können wir den Strom um ein neues Zeichen bitten:

ArithTokenizer.java

```
36
37        next = reader.read();
38      }
39    }catch (IOException _){}
40    return result;
41  }
42
```

Zum Lesen von Zahlen holen wir solange Zeichen, wie noch Ziffern im Eingabestrom zu finden sind und addieren diese auf.

ArithTokenizer.java

```
43
44  Token getInt() throws IOException {
45    int res=0;
46    while (next>=0 && Character.isDigit((char)next)){
47      res=res*10+next-48;
48        next = reader.read();
49    }
50
51    return new Zahl(res);
52  }
53
```

Schließlich schreiben wir uns zwei statische Methoden zur einfachen Benutzung des Tokenizers.

ArithTokenizer.java

```
54
55  static List<Token> tokenize(String inp) throws Exception{
56    return tokenize(new StringReader(inp));
57  }
58
59  static List<Token> tokenize(Reader inp) throws Exception{
60      return new ArithTokenizer(inp).tokenize();
61  }
62 }
```

So lassen sich einfache Test in Jugs schnell durchführen:

```
sep@swe10:~/fh/prog2/examples> java Jugs
     -- --  --  ----   ---      ----------------------------------------
     || ||  || ||  || ||__      Jugs: the interactive Java interpreter
     || ||__|| ||__||  __||     Copyright (c) 2003 Sven Eric Panitz
     ||        ___||            World Wide Web:
||   ||                                 http://www.tfh-berlin.de/~panitz
\\__//  Version: February 2003  ----------------------------------------


>ArithTokenizer.tokenize("888+0009   *+-/   9")
```

```
[888, add, 9, mult, add, sub, div, 9]
>ArithTokenizer.tokenize("888+0009   *uiu**+- 09")
java.lang.Exception: unexpected Token found: 'u'
>
```

Wir können die unterschiedlichsten Klassen aus diesem Skript für eine kleine Anwendung zusammensetzen. Wir lesen eine Datei mit der Endung `.arith`, in der ein arithmetischer Ausdruck steht. Die Datei wird gelesen, die Tokenliste erzeugt und diese mit dem Parser geparst. Der erzeugte Ableitungsbaum wird in eine`png`-Bilddatei geschrieben. Wir sehen ein paar praktische statische Methoden zum Testen in Jugs und eine Hauptmethode vor:

ArithParsTreeBuilder.java

```
1  package name.panitz.parser;
2  import java.io.*;
3  import javax.imageio.ImageIO;
4  import javax.swing.*;
5  import name.panitz.data.tree.Tree;
6  import name.panitz.data.tree.DisplayTree;
7  import static name.panitz.data.tree.TreeToFile.treeToFile;
8
9  class ArithParsTreeBuilder{
10
11   public static void main(String [] args) throws Exception{
12     final String fileName = args[0];
13     writeParsTreeFile(new FileReader(fileName+".arith"),fileName);
14   }
15
16   static void writeParsTreeFile(Reader reader,String fileName)
17                                                    throws Exception{
18     treeToFile(parsArith(reader),fileName);
19   }
20
21   public static Tree<String> parsArith(Reader reader)
22                                                    throws Exception{
23     final ParseResult<Tree<String>> res
24      = new TreeArith().parse(ArithTokenizer.tokenize(reader));
25     return res.getResult();
26   }
27
28   public static DisplayTree parseAndShow(String inp)
29                                          throws Exception{
30     return new DisplayTree(parsArith(new StringReader(inp)));
31   }
32
33   public static void parseAndShowInFrame(String inp)
34                                          throws Exception{
35     JComponent c = ArithParsTreeBuilder.parseAndShow(inp);
36     JFrame f = new JFrame();
37     f.getContentPane().add(c);
38     f.pack();
39     f.setVisible(true);
```

```
40    }
41  }
```

Aufgabe 22 (2 Punkte) In dieser Aufgabe sollen Sie den Parser für arithmetische Ausdrücke um geklammerte Ausdrücke erweitern. Hierzu sei die Grammatik für die Regel *multExpr* wie folgt geändert:

multExpr ::= *atomExpr multOp multExpr*|*atomExpr*
atomExpr ::= `(`*addExpr*`)`|*zahl*

Hierzu sei die entsprechende zusätzliche Tokenaufzählung für die zwei Klammersymbole gegeben:

Parentheses.java

```
1  package name.panitz.parser;
2  public enum Parentheses implements Token{lpar,rpar;}
```

Zusätzlich gegeben sei die die folgende Funktion:

DoParentheses.java

```
1  package name.panitz.parser;
2
3  public class DoParentheses
4    implements
5      Function<Pair<Parentheses,Pair<Integer,Parentheses>>,Integer>{
6    public Integer
7                apply(Pair<Parentheses,Pair<Integer,Parentheses>> x){
8      return x.getE2().getE1();
9    }
10 }
```

a) Schreiben Sie eine Klasse `AtomExprEval`, die `Parser<Integer>` implementiert und der Regel für *atomExpr* entspricht.

b) Ändern Sie die Klasse *MultExprEval*, so daß sie der geänderten Grammatikregel entspricht.

c) Erweitern Sie die Klasse `ArithTokenizer`, so daß er auch die beiden Klammersymbole erkennt.

4.4 Parsergeneratoren

Im letzem Abschnitt haben wir gesehen, daß die Aufgabe einen Parser für eine kontextfreie Grammatik zu schreiben eine mechanische Arbeit ist. Nach bestimmten Regeln läßt sich eine Grammatik direkt in einen Parser umsetzen. In unserem in Java geschriebenen Parser führte jede Regel zu genau einer Methode.

Aus der Tatsache, daß einen Parser für eine Grammatik zu schreiben eine mechanische Tätigkeit ist, folgt sehr schnell, daß man diese gerne automatisiert durchführen möchte

und in der Tat schon sehr bald gab es Werkzeuge, die diese Aufgabe übernehmen konnten, sogenante Parsergeneratoren. Heutzutage schreibt kaum ein Programmierer in einer Sprache wie Java einen Parser von Hand, sondern läßt den Parser aus der Definition der Grammatik generieren.

Der wohl am weitesten verbreitete Parsergenerator ist Yacc[Joh75]. Wie man seinem Namen, der für *yet another compiler compiler* steht, entnehmen kann, war dieses bei weiten nicht der erste Parsergenerator. Yacc erzeugt traditionell C-Code; es gibt aber mitlerweile auch Versionen, die Java-Code generieren. Ein weitgehendst funktionsgleiches Programm zu Yacc heißt *Bison*. Die lexikalische Analyse steht im im Zusammenspiel mit Yacc und Bison jeweils auch ein Generatorprogramm zur Verfügung, daß einen Tokenizer generiert. Dieses Programm heißt `lex`.

Weitere modernere und auf Java zugeschnittene Parsergeneratoren sind: `yavacc`, `antlr` und `gentle`[Gru01].

4.4.1 javacc

Wir wollen uns die Mühe machen, einen Parser mit einem Parsergenerator zu beschreiben. Hierzu nehmen wir den Parsergenerator `javacc`. Dieser Parsergenerator kommt mit zwei Progammen:

- `jjtree`: Dieses Programm generiert Klassen, mit denen der Ableitungsbaum eines Parses dargestellt werden kann. Es nimmt als Eingabedatei eine Beschreibung der Grammatik. Diese Datei hat die Endung `.jjt`. Sie generiert Klassen für die Baumknoten. Zusätzlich generiert es eine Datei mit der Endung `.jj`. die im nächsten Schritt Eingabe für den eigentlichen Parsergenerator ist.
- `javacc`: Dieses Programm generiert eine Klasse für den eigentlichen Parser. Seine Eingabe ist eine Beschreibung der Grammatik in einer Datei mit Endung `.jj`. Sie generiert eine Klasse für den Parser und eine für die Token, die in der Grammatik spezifiziert werden. Zusätzlich werden Klassen zur Fehlerbehandlung generiert.

Eine wichtige Frage bei der Benutzung eines Parsergenerators ist, nach welcher Strategie der generierte Parser vorgeht. Wenn es ein rekurisv absteigender Parser ist, so dürfen wir keine Linksrekursion in der Grammatik haben. Wenn der generierte Parser kein *backtracking* hat, dann muß er mit Betrachtung des aktuellen Tokens wissen, welche Regelalternative zu wählen ist. Die Alternativen brauchen entsprechend disjunkte Startmengen. Bei generierten Parsern mit der Strategie des Schieben und Reduzierens, kann es sein, daß es zu sogenannten *shift-reduce* Konflikten kommt, wenn nämlich bereits reduziert werden kann, aber auch nach einem *shift*-Schritt reduziert werden kann.

Desweiteren ist wichtig, was für Konstrukte die Eingabegrammtik zuläßt. Konfortable moderne Parsergeneratoren lassen heute die erweiterte Backus-Naur-Form zur Beschreibung der Grammatik zu.

`javacc` generiert einen rekursiv absteigenden Parser ohne `backtracking`. Es darf also keine linksrekursive Regel in der Grammatik sein.

`javacc` integriert einen Tokenizer, d.h. mit der Grammatik werden auch die Token definiert.

Beispiel:
Als erstes Beispiel wollen wir für unsere Sprache astronomischer Aussagen einen

Parser generieren lassen. Hierzu schreiben wir eine Eingabegrammatikdatei mit den Namen `sms.jjt`.[3].

Die Datei `sms.jjt` beginnt mit einer Definition der Klasse, die den Parser enthalten soll. In unserem Beispiel soll die generierte Klasse `SMS` heißen:

sms.jj

```
1  PARSER_BEGIN(SMS)
2  public class SMS {
3  }
4  PARSER_END(SMS)
```

Anschließend lassen sich die Token der Sprache definieren. In unserem Beispiel gibt es 8 Wörter. Wir können für `javacc` definieren, daß Groß-und-Kleinschreibung für unsere Wörter irrelevant ist.

sms.jj

```
5  TOKEN [IGNORE_CASE] :
6  { <MOON: "moon">
7  | <MARS: "mars">
8  | <MERCURY: "mercury">
9  | <PHOEBUS: "phoebus">
10 | <DEIMOS: "deimos">
11 | <ORBITS: "orbits">
12 | <IS: "is">
13 | <A: "a">
14 }
```

Zusätzlich gibt es in `javacc` die Möglichkeit anzugeben, was für Zwischenraum zwischen den Token stehen darf. In unserem Beispiel wollen wir Leerzeichen, Tabularturzeichen und Zeilenendezeichen zwischen den Wörtern als Trennung zulassen:

sms.jj

```
15 SKIP :
16 { " "
17 | "\t"
18 | "\n"
19 | "\r"
20 }
```

Und schließlich und endlich folgen die Regeln für die Grammatik:

sms.jjt

```
21 void start() : {}
22 {
23   nounPhrase() verbPhrase()
24 }
25
26 void nounPhrase() : {}
27 {
28   noun()|article() noun()
```

[3]SMS stehe hier für *Sonne Mond und Sterne*.

```
29 }
30
31 void noun() : {}
32 { <MOON>|<MARS>|<MERCURY>|<PHOEBUS>|<DEIMOS>
33 }
34
35 void article() : {}
36 {<A>}
37
38 void verbPhrase() : {}
39 {
40 verb() nounPhrase()
41 }
42
43 void verb() : {}
44 {
45  <IS>|<ORBITS>
46 }
```

Aus dieser Datei `sms.jj` lassen sich jetzt mit dem Programm `jjtree` Klassen für die Darstellung des Ableitungsbaums generieren:

```
sep@swe10:~/fh/prog2/beispiele/javacc/sms> ls
sms.jjt
sep@swe10:~/fh/prog2/beispiele/javacc/sms> jjtree sms.jjt
Java Compiler Compiler Version 2.1 (Tree Builder)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "jjtree" with no arguments for help)
Reading from file sms.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
Annotated grammar generated successfully in sms.jj
sep@swe10:~/fh/prog2/beispiele/javacc/sms> ls
JJTSMSState.java  SMSTreeConstants.java  sms.jj
Node.java         SimpleNode.java        sms.jjt
sep@swe10:~/fh/prog2/beispiele/javacc/sms>
```

Wie zu sehen ist, werden Klassen für Baumknoten generiert (`SimpleNode.java`) und eine Eingabedatei für den eigentlichen Parsergenerator (`sms.jj`). Jetzt können wir diesen Parser generieren lassen:

```
sep@swe10:~/fh/prog2/beispiele/javacc/sms> javacc sms.jj
Java Compiler Compiler Version 2.1 (Parser Generator)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "javacc" with no arguments for help)
Reading from file sms.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
sep@swe10:~/fh/prog2/beispiele/javacc/sms> ls
```

```
JJTSMSState.java     SMSConstants.java      SimpleNode.java     sms.jjt
Node.java            SMSTokenManager.java   Token.java
ParseException.java  SMSTreeConstants.java  TokenMgrError.java
SMS.java             SimpleCharStream.java  sms.jj
sep@swe10:~/fh/prog2/beispiele/javacc/sms>
```

Die generierten Javaklassen lassen sich übersetzen:

```
sep@swe10:~/fh/prog2/beispiele/javacc/sms> javac *.java
sep@swe10:~/fh/prog2/beispiele/javacc/sms> ls
JJTSMSState.class     SMSConstants.class      SimpleNode.class
JJTSMSState.java      SMSConstants.java       SimpleNode.java
Node.class            SMSTokenManager.class   Token.class
Node.java             SMSTokenManager.java    Token.java
ParseException.class  SMSTreeConstants.class  TokenMgrError.class
ParseException.java   SMSTreeConstants.java   TokenMgrError.java
SMS.class             SimpleCharStream.class  sms.jj
SMS.java              SimpleCharStream.java   sms.jjt
sep@swe10:~/fh/prog2/beispiele/javacc/sms>
```

Bevor wir den Parser benutzen können, schauen wir uns die generierter Parserdatei `SMS.java` in Auszügen einmal an:

```
1  /* Generated By:JJTree&JavaCC: Do not edit this line. SMS.java */
2  public class SMS/*bgen(jjtree)*/
3  implements SMSTreeConstants, SMSConstants {/*bgen(jjtree)*/
4
5    protected static JJTSMSState jjtree = new JJTSMSState();
6
7    static final public void start() throws ParseException {
8                 /* bgen(jjtree) start */
9    SimpleNode jjtn000 = new SimpleNode(JJTSTART);
10   boolean jjtc000 = true;
11   jjtree.openNodeScope(jjtn000);
12     try {
13       nounPhrase();
14       verbPhrase();
15     } catch (Throwable jjte000) {
16
17 ...
18
19
20
21   static final public void nounPhrase() throws ParseException {
22                     /* bgen(jjtree) nounPhrase */
23   SimpleNode jjtn000 = new SimpleNode(JJTNOUNPHRASE);
24   boolean jjtc000 = true;
25   jjtree.openNodeScope(jjtn000);
26     try {
27       switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
28       case MOON:
29       case MARS:
```

```
30        case MERCURY:
31        case PHOEBUS:
32        kcase DEIMOS:
33
34
35    public SMS(java.io.InputStream stream) {
36      if (jj_initialized_once) {
37
38
39  ....
40
```

Als erstes ist festzustellen, daß der generierte Parser kein gutes objektorientiertes Design hat. Es gibt für jede Regel der Grammatik eine Methode, allerdings sind alle Methoden statisch.

Es gibt einen Konstruktor dem ein Objekt des Typs `java.io.InputStream`, in dem die zu parsende Eingabe übergeben wird.

Alle Methoden können im Fehlerfall eine `ParseException` werfen, die angibt, wieso eine Eingabe nicht geparst werden konnte.

Es gib ein Feld `jjtree`des Typs `JJTSMSState`, in welchem das Ergebnis des Parses gespeichert ist. Diese Klasse enthält eine Methode `Node rootNode()`, mit der die Wurzel des Ableitungsbaums erhalten werden kann.

Damit können wir eine Hauptmethode schreiben, in der wir den generierten Parser aufrufen.

```
1  public class MainSMS {
2    public static void main(String args[]) throws ParseException {
3      SMS parser = new SMS(System.in);
4      parser.start();
5      ((SimpleNode)parser.jjtree.rootNode()).dump("");
6    }
7  }
```

Als Eingabestrom definieren wir die Tastatureingabe. Wir benutzen die Methode `dump`, um den Ableitungsbaum auf dem Bilschirm auszugeben. Wir können nun das Hauptprgramm starten und mit der Tastatur Sätze eingeben:

```
sep@swe10:~/fh/prog2/beispiele/javacc/sms> javac *.java
sep@swe10:~/fh/prog2/beispiele/javacc/sms> java MainSMS
a moon orbits mars
start
 nounPhrase
  article
  noun
 verbPhrase
  verb
  nounPhrase
   noun
sep@swe10:~/fh/prog2/beispiele/javacc/sms> java MainSMS
mars is a red planet
Exception in thread "main" TokenMgrError:
```

```
Lexical error at line 1, column 11.  Encountered: "r" (114), after : ""
        at SMSTokenManager.getNextToken(SMSTokenManager.java:447)
        at SMS.jj_ntk(SMS.java:298)
        at SMS.noun(SMS.java:84)
        at SMS.nounPhrase(SMS.java:50)
        at SMS.verbPhrase(SMS.java:133)
        at SMS.start(SMS.java:12)
        at MainSMS.main(MainSMS.java:4)
sep@swe10:~/fh/prog2/beispiele/javacc/sms>
```

Aufgabe 23 Fügen Sie der Datei `sms.jjt` folgende Zeilen am Anfang ein:

```
1 options {
2    MULTI=true;
3    STATIC=false;
4 }
```

Generieren Sie mit `jjtree` und `javac` den Parser neu. Was hat sich geändert?

4.4.2 Parsen und moderne Programmierparadigmen

Wie an den Jahreszahlen der Literaturangaben zu sehen, sind Parser ein sehr altes Geschäft. Die Grundlagen sind zu einer Zeit entwickelt worden, in der es weder objektorientierte noch moderne funktionale Programmiersprachen gab. Daher sind Parser- und vergleichbare Algorithmen häufig in einer sehr prozeduralen Weise formuliert. Modernere Entwicklungen der Progammiertechnik machen aber auch vor Parsern nicht halt. Ein Kurs, der stärker das objektorientierte Paradigma unterstützt findet sich in [ACL02].

In der funktionalen Progammierung haben sich sogenannte Parserkombinatoren durchgesetzt. Dieses sind Operatoren die zwei Parser als die Sequenz der nacheinanderausführung kombinieren oder als Alternativen kombinieren. Ein Parserkombinator stellt also eine Funktion dar, die aus zwei Parsern einen neuen Parser als Ergebnis hat. Da Parser selbst Funktionen sind, die einen Tokenstrom in einen Parsbaum zerteilen, sind Parserkombinatoren Funktionen höherer Ordnung: sie kombinieren zwei Funktionen zu einer neuen. Ein frühes beeindruckendes Beispiel für Parserkombinatoren findet sich in [FL89], in dem ausführlich eine Grammatik für englischsprachige Sätze über das Sonnensystem modelliert und in der Sprache Miranda[Tur85] implementiert ist.

Anhang A

Lösungen ausgewählter Aufgaben

A.1 Lösung der Punkteaufgabe zu Bäumen

In diesem Abschnitt werden einige Lösungen der dritten Punkteaufgabe vorgestellt.

A.1.1 leaves

`List leaves()`: erzeugt eine Liste der Markierungen an den Blättern des Baumes.

Wir können die Methode `leaves` aus der Methode `flatten` ableiten. Beide Methoden nehmen Elemente eines Baumes und fügen diese in eine Ergebnisliste ein.

Tree.java

```
5  public List<a> leaves(){
6    //leere Liste für das Ergebnis
7    List<a> result = new ArrayList<a>();
8
9    //füge die Markierung dieses Knotens zur Ergebnisliste
10   //nur hinzu, wenn es ein Blatt ist
11   if (isLeaf()) result.add(mark());
12
13   //für jedes Kind
14   for (Tree<a> child:theChildren())
15     //erzeuge dessen Knotenliste und füge alle deren Blätter
16     //zum Ergebnis hinzu
17     result.addAll(child.leaves());
18
19   return result;
20 }
```

Alles, was zu ändern ist, ist eine Bedingung aufzustellen, wann ein Knoten der Ergebnisliste hinzugefügt werden soll. Bei `flatten` war das immer der Fall, bei `leaves` nur, wenn es sich bei dem Baumknoten um ein Blatt handelt.

Akkumulative Lösung

Wir haben uns bisher noch keine Gedanken um die Performanz unserer Methoden gemacht. Wenn wir die die Methoden `flatten` und `leaves` einmal genau betrachten, so stellen wir fest, daß jeder Aufruf der Methode eine neue Ergebnisliste erzeugt. Das bedeutet, daß die Methoden `flatten` und `leaves` durch die rekursiven Aufrufe, soviel Listen erzeugt, wie Knoten im Baum sind. Mit dem Aufruf der Methode `addAll` auf Listen, werden die Elemente einer Liste in eine neue Liste eingetragen. Wir sind aber nur an einer einzigen Liste interessiert, der endgültigen Ergebnisliste.

Es wäre also schön, wenn wir eine Methode `flatten` hätten, die nicht eine neue Liste erzeugt, sondern eine bestehende Liste benutzt, um dort weitere Elemente einzufügen. Wir können hierzu der Methode einen Listenparameter geben, in dem die Ergebnisse zu sammeln sind.

Tree.java

```
21 public List<a> flatten(List<a> result){
22   result.add(mark());
23
24   for (Tree<a> child:theChildren())
25     child.flatten(result);
26
27   return result;
28 }
```

Die eigentliche Methode `flatten` kann jetzt einmal das Objekt für die Ergebnisliste erzeugen, und dann diese jeweils weitereichen, um das Erzeugen neuer Listen und das umkopieren der Elemente von einer Liste in eine andere zu verhindern.

Tree.java

```
29 public List<a> flattenAkk(){
30   return flatten(new ArrayList<a>());
31 }
```

Und entsprechend können wir die Methode `leaves` auch umschreiben, so daß sie nicht für jeden rekursiven Aufruf ein neues Listenobjekt erzeugt:

Tree.java

```
32 public List<a> leaves(List<a> result){
33   if (isLeaf()) result.add(mark());
34
35   for (Tree<a> child:theChildren())
36     child.leaves(result);
37
38   return result;
39 }
40
41 public List<a> leavesAkk(){
42   return leaves(new ArrayList<a>());
43 }
```

Dieses Prinzip, ein Ergebnis in einem Parameter nach und nach zusammenzusammeln nennt man *Akkumulation*. Der Parameter, in dem das Ergebnis jeweils hinzugefügt wird, ist der Akkumulator. Mit Hilfe des Befehls `time` auf der Unix-Shell, läßt sich sogar ungefähr messen, daß die akkumulierende Methode schneller abgearbeitet wird.

Programmierung höherer Ordnung

Wir haben im letztem Abschnitt bereits beobachten können, daß die Methoden `flatten` und `leaves` fast identischen Code haben. Beide selektieren aus einem Baum bestimmte Baummarkierungen heraus. Das einzige was sie unterscheidet, ist die Bedingung, die steuert, ob eine Knotenmarkierung zu selektieren ist oder nicht. Ansonsten haben wir Code verdoppelt.

Unser Bestreben ist stets eine Codeverdoppelung zu vermeiden, denn verdoppelter Code ist auch doppelt zu pflegen und zu warten. Das haben wir sogar schon festgestellt, als wir auf die akkumulative Lösung der beiden Methoden gekommen sind. Da haben wir beide Methoden geändert, um für sie die akkumulative Lösung einzuführen.

Wir können eine Methode schreiben, die den gemeinsamen Code der beiden Methoden enthält. Das einzige, was die beiden Methoden unterscheidet ist eine Bedingung. Diese Bedingung ist eine Prüfung auf die Baumknoten. Die gemeinsame Methode muß diese Bedingung als Parameter übergeben bekommen. Eine Prüfung ist eine Methode. Wir können in Java keine Methoden direkt als Parameter übergeben, sondern nur Objekte. Wir definieren daher eine Schnittstelle, die beschreibt, daß ein Objekt eine Prüfmethoden enthält.

TreeCondition.java

```
1  package name.panitz.data.tree;
2
3  interface TreeCondition<a>{
4    boolean takeThis(Tree<a> o);
5  }
```

Mit dieser Schnittstelle sind wir jetzt in der Lage eine Bedingung auf Baumknoten auszudrücken. Wir können damit eine allgemeine Filtermethode schreiben, die Knoten mit einer bestimmten Bedingung aus einem Baum selektiert. Die Methode `filter` ist die Verallgemeinerung die Methoden `leaves` und `flatten`. Wir verallgemeinern die akkumulative Lösung.

Tree.java

```
6  public List<a> filter(List<a> result,TreeCondition<a> c){
7    //wenn die Bedingung für den Knoten wahr ist, füge
8    //ihm dem Ergebnis hinzu
9    if (c.takeThis(this)) result.add(mark());
10
11   //für alle Kinder
12   for (Tree<a> child:theChildren())
13     //rufe den Filter mit gleicher Bedingung auf
14     child.filter(result,c);
15
16   return result;
17 }
```

Die akkumulative Methode kann noch wie gewohnt gekapselt werden:

Tree.java
```
18 public List<a> filter(TreeCondition<a> c){
19   return filter(new ArrayList<a>(),c);
20 }
```

Jetzt haben wir eine Methode geschrieben, die einmal durch einen Baum läuft und Knoten aufgrund einer Bedingung in einer Ergebnisliste sammelt. Wir können jetzt die Methoden `leaves` und `flatten` über diese Methode ausdrücken:`leaves` ist ein Filter mit der Bedingung, daß die Baumknoten ein Blatt ist. Wir erzeugen das Bedingungsobjekt als anonyme innere Klasse:

Tree.java
```
21 public List leavesFilter(){
22   return filter(
23      new TreeCondition<a>(){
24         public boolean takeThis(Tree<a> t){
25            return t.isLeaf();
26         }
27      });
28 }
```

`flatten` ist ein Filter mit der Bedingung die immer wahr ist. Also erzeugen wir eine Baumbedingung, die konstant `true` als Ergebnis liefert:

Tree.java
```
29 public List<a> flattenFilter(){
30   return filter(new TreeCondition<a>(){
31                   public boolean takeThis(Tree<a> _){ return true;}
32                 });
33 }
```

Entsprechend können wir jetzt beliebige Bedingungen benutzen, um Knoten aus dem Baum zu selektieren; z.B. alle Knoten mit genau zwei Kindern:

Tree.java
```
34 public List<a> nodesWithTwoChildren(){
35   return filter(new TreeCondition<a>(){
36                   public boolean takeThis(Tree<a> t){
37                      return t.theChildren().size()==2;
38                   }
39                 });
40 }
```

Oder Knoten, die mit einem bestimmten Objekt markiert sind:

Tree.java
```
41 public List<a> nodesMarkedWith(final a o){
42   return filter(new TreeCondition<a>(){
43                   public boolean takeThis(Tree<a> t){
```

```
44                          return t.mark().equals(o);
45                      }
46                  });
47  }
```

Letztere Methode kann sogar genutzt werden, um zu entscheiden, ob ein Objekt in einem Baum enthalten ist.

Tree.java

```
48  public boolean containsFilter(a o){
49    return !nodesMarkedWith(o).isEmpty();
50  }
```

Das in dieser Lösung vorgestellte Prinzip, daß quasi eine Methode als Parameter an eine andere Methode übergeben wird, heißt *Programmierung höherer Ordnung*. Java unterstützt dieses Konzept nur rudimentär, weil zum Übergeben einer Methode als Parameter erst ein Objekt erzeugt werden muß. In Sprachen wie C kann dieses Programmierprinzip über Funktionszeiger realisiert werden. In Sprachen der `Lisp`-Tradition ist die Übergabe von Methoden als Parameter ein fundamentaler Bestandteil. Mit Hilfe der Programmierung höherer Ordnung hat man ein sehr mächtiges Ausdrucksmittel.

A.1.2 show

`String show()`: erzeugt eine textuelle Darstellung des Baumes. Jeder Knoten soll eine eigene Zeile haben. Kinderknoten sollen gegenüber einem Elternknoten eingerückt sein.

Wir schreiben als erstes eine Methode, die einen Baum mit einer als Parameter übergebenen Einrückung zeigt. Diese Einrückung wird für die Kinder um zwei Zeichen verlängert:

Tree.java

```
51    public String show(String indent){
52      //zeige diesen Knoten mit Einrückung
53      String result = indent+mark();
54
55      //erhöhe die Einrückung
56      indent=indent+"  ";
57
58      //für jedes Kind
59      for (Tree<a> child:theChildren())
60       //erzeuge neue Zeile und zeige Kind mit neuer Einrückung
61        result=result+"\n"+child.show(indent);
62
63      return result;
64    }
```

Die Wurzel eines Baumes zeigen wir ohne Einrückung. Wir kapseln die obige Methode entsprechend:

Tree.java

```
65    public String show(){
66        return show("");
67    }
```

Akkumulative Lösung

Auch für die Methode `show` können wir eine akkumulative Lösung schreiben. Der Grund, weshalb dieses sinnvoll ist, ist in der Technik von der Stringumsetzung in Java versteckt. Der Übeltäter ist hierbei der Operator `+` auf `String`. Dieser Operator hängt nicht einen `String` an einen anderen `String` an, sondern kopiert beide in einen neuen `String`. Jede Anwendung des Operators `+` kopiert also `String`-Objekte; und wir wenden den Operator oft an, um unser Ergebnis zu berechnen.

Java stellt eine Klasse zur Verfügung mit der `String`-Objekte akkumuliert werden können, und Kopiervorgänge weitgehendst vermieden werden: die Klasse `StringBuffer`.

Objekte der Klasse `StringBuffer` können mit der Methode `append` weitere `String`-Objekte angehängt werden. Sie wird benutzt, wenn ein `String` erst nach und nach zusammengebaut wird.

Ähnlich wie bei der Akkumulation eines Listenergebnisses können wir mit `StringBuffer` eine Zeichenkette akkumulieren. Hierzu geben wir in einem Parameter einen `StringBuffer` als Akkumulator mit. Wie im Listenfall mit der Methode `add` benutzen wir die Methode `append` zum Aufsammeln der Ergebnisteile:

Tree.java

```
68  public StringBuffer show(String indent,StringBuffer result){
69    //zeige diesen Knoten mit Einrückung
70    result.append(indent);
71    result.append(mark().toString());
72
73    //erhöhe die Einrückung
74    indent=indent+"  ";
75
76    //für jedes Kind
77    for (Tree<a> child:theChildren()){
78     //erzeuge neue Zeile und zeige Kind mit neuer Einrückung
79      result.append("\n");
80      child.show(indent,result);
81    }
82    return result;
83  }
```

Die kapselnde Methode, erzeugt einen neuen `StringBuffer`, läßt diesen akkumulieren und gibt schließlich den in ihm enthaltenen `String` als Ergebnis aus.

Tree.java

```
84  public String showAkk(){
85      return show("",new StringBuffer()).toString();
86  }
```

A.1.3 contains

`boolean contains(a o)`: ist wahr, wenn eine Knotenmarkierung gleich dem Objekt `o` ist.

Wir haben im Zuge der Methode `filter` schon eine Lösung für die Methode `contains` quasi umsonst bekommen. Diese Lösung hat alle Knoten die das fragliche Objekt enthalten in einer Liste gesammelt und gekuckt, ob die Liste leer ist oder nicht. Dieses Prinzip wird manchmal auch unter dem Motto: *representing failure by a list of successes* vorgestellt. Die Ergebnisliste zeigt die Treffer an. `contains` falliert, wenn diese Liste leer ist.

Der Nachteil an dieser Lösung war, daß der gesammte Baum durchlaufen wurde, auch wenn z.B. die Wurzel bereits der fragliche Knoten war. Wir entwickeln nun eine Lösung, die einen Baum nur bis zum ersten Auftreten des fraglichen Objektes durchläuft. Nur wenn das Objekt nicht im Baum enthalten ist, wird der komplette Baum durchlaufen.

Tree.java

```
87  public boolean contains(a o){
88    //bin ich schon der gesuchte Knoten? Wenn ja ende mit erfolg
89    if (mark().equals(o)) return true;
90
91    //für jedes Kind
92   for (Tree<a> child:theChildren()){
93      //wenn es den Knoten enthält, beende mit erfolg
94      if (child.contains(o)) return true;
95    }
96    //keiner der Knoten enthielt das Objekt
97    return false;
98  }
```

A.1.4 maxDepth

`int maxDepth()`: gibt die Länge des längsten Pfades von der Wurzel zu einem Blatt an.

Die maximale Tiefe läßt sich relativ einfach finden. Jedes Kind wird nach seiner maximalen Tiefe befragt und aus diesen die maximale Zahl genommen. Schließlich wird diese um eins erhöht, für die Wurzel selbst.

Tree.java

```
99  public int maxDepth(){
100   //anfangstiefe
101   int result = 0;
102
103   //für jedes Kind
104  for (Tree<a> child:theChildren()){
105     //berechne seine maximale Tiefe
106     int n = child.maxDepth();
107
108     //ist sie höher als die bisher gefundene, dann nimm diese
109     if (n>result) result=n;
110   }
111
112   //erhöhe Ergebnis um eins für die Wurzel
113   return result+1;
114 }
```

A.1.5 getPath

`List getPath(Object o)`: gibt die Markierungen auf dem Pfad von der Wurzel zu dem Knoten, der mit dem Objekt `o` markiert ist, zurück. Ergebnis ist eine leere Liste, wenn ein Objekt gleich `o` nicht existiert.

naive Lösung

Fast alle Studenten haben eine Lösung entwickelt, in der die Aufgabe mit Hilfe der Mehode `contains` realisiert wird. Jeder Knoten, der auf dem Pfad zu dem fraglichen Objekt liegt, enthält diesen Knoten. Es wird eine Ergebnisliste aufgebaut, die genau die Knoten enthält, deren Bäume den fraglichen Knoten enthalten:

Tree.java

```
115   public List<a> getPathBad(a o){
116     List<a> result = new ArrayList<a>();
117
118   //wenn Du irgendwo das Objekt enthältst, liegst Du auf den Pfad
119     if (contains(o)) { result.add(mark());}
120
121     //für jedes Kind
122    for (Tree<a> child:theChildren()){
123       //füge den Pfad zum fraglichen Objekt hinzu
124       result.addAll(child.getPathBad(o));
125     }
126     return result;
127   }
```

Lösung höherer Ordnung mit Hilfe von filter

Die obige Lösung ist wieder von der Art: selektiere alle Knoten mit einer bestimmten Eigenschaft aus den Baum. Die Eigenschaft ist keine lokal den Knoten betreffende Eigenschaft, sondern die Tatsache, daß der Teilbaum das Objekt enthält. Wir können also auch für diese Aufgabe die Methode `filter` anwenden.

Tree.java

```
128 public List<a> getPathFilter(final a o){
129    return filter(
130       new TreeCondition<a>(){
131         public boolean takeThis(Tree<a> e){
132           return e.contains(o);
133         }
134       }
135    );
136 }
```

Spätestens jetzt merkt man, was für eine ausdrucksstarke Methode die Methode `filter` durch ihren Parameter höherer Ordnung ist.

performantere Lösung

Beide obigen Lösungen waren schlecht, wenn wir ihr Laufzeitverhalten betrachten. Stellen wir uns der Einfachheit halber einen zur Liste entartetetn Baum mit n Knoten vor. Das einzige Blatt ist der Knoten, zu dem der Pfad gesucht wird. Für jeden Knoten des Baumes fragen wir, ob er das fragliche Objekt in seinem Teilbaum enthält, und jedesmal durchlaufen wir dabei den Baum bis zu seinem Blatt, weil wir die Methode `contains` aufrufen. Wir rufen also n-Mal die Methode `contains` auf, die jedesmal den Baum bis zum Blatt durchläuft. Wir durchlaufen also $n + (n-1) + (n-2) + \ldots + (n-n)$
Baumknoten, was nach der Summenformel $\frac{n*(n+1)}{2}$ ist.

Damit haben wir einen quadratischen Aufwand in Abhängigkeit von der Tiefe des gesuchten Pfades. Quadratische Aufwände sind unbedingt zu vermeiden, weil mit zunehmender Datengröße die Laufzeit eines Algorithmus enorm schnell zunimmt.

Wir können mit einem kleinen Trick, auf die Benutzung der Methode `contains` verzichten. Hierzu erinnern wir uns an das Prinzip, einen Fehlschlag durch eine leere Liste zu charakterisieren. Wir fragen einfach jedes Kind nach seinem Pfad zu dem bewußten Objekt. Wenn diese Pfad leer ist, so enthält das Kind diesen Knoten nicht. Sobald ein Kind uns einen nichtleeren Pfad gibt, benutzen wir die überladene Methode `add` der Listenschnittstelle, die uns erlaubt, ein Element an eine beliebige Stelle einer Liste einzufügen.

Tree.java

```
137  public List<a> getPath(a o,List<a> result){
138    //bist Du selbst der Knoten,
139    //dann ende mit der einelementigen Liste
140    if (mark().equals(o)) { result.add(mark()); return result;}
141
142   for (Tree<a> child:theChildren()){
143      //berechne Pfad vom Kind zum Objekt
144      final List<a> path = child.getPath(o,result);
145
146      //Kind hat einen solchen Pfad
147      if (!path.isEmpty()){
148        //hänge Dich vorne dran
149        result.add(0,mark());
150        //und fertig
151        return result;
152      }
153    }
154    return result;
155  }
```

Da wir oben die akkumulative Methode geschrieben haben, benötigen wir noch eine kapselnde Methode, in der die Ergebnisliste erzeugt wird:

```
1  public List<a> getPath(a o){
2      return getPath(o,new ArrayList<a>());
3  }
```

A.1.6 iterator

`java.util.Iterator iterator()`: erzeugt ein Iterator über alle Knotenmarkierungen des Baumes.

Wir brauchen einen Iterator, der alle Markierungen des Baumes durchläuft. Wir können es und hier sehr einfach machen. Anstatt eine eigene Iteratorklasse zu implementieren, die über die Knoten eines Baumes läuft, können wir einfach unsere schon bestehende Methode `flatten` benutzen, die die Elemente eines Baumes in eine Liste steckt, um dann einfach den Iterator dieser Liste zu benutzen:

Tree.java

```
4  public Iterator<a> iterator(){
5      return flatten().iterator();
6  }
```

A.1.7 sameStructure

`boolean sameStructure(Tree other)`: soll genau dann wahr sein, wenn der Baum `other` die gleiche Struktur hat.

Zur Lösung durchlaufen wir die Struktur beider Bäume in gleicher Weise, so lange bis, wir einen Unterschied in ihrer Struktur gefunden haben. Falls wir, ohne auf eine Widerlegung der Strukturgleichheit zu stoßen, beide Bäume komplett durchlaufen haben, dann haben beide Bäume die gleiche Struktur.

Tree.java

```
7  public <b> boolean sameStructure(Tree<b> other){
8    //haben beide Bäume gleich viel Kinder.
9    if(this.theChildren().size()!=other.theChildren().size())
10       //wenn nein, dann sind sie nicht strukturgleich
11       return false;
12
13   //Iteriere über die Kinder beider Bäume
14   Iterator<Tree<a>> it1=this.theChildren().iterator();
15   Iterator<Tree<b>> it2=other.theChildren().iterator();
16
17   //solange es noch Kinder gibt
18   while (it1.hasNext()){
19     //hohle das nächste Kind beider Bäume
20     final Tree<a> thisChild = it1.next();
21     final Tree<b> otherChild =it2.next();
22
23     //sind diese nicht strukturgleich, dann breche ab
24     if (!thisChild.sameStructure(otherChild)) return false;
25   }
26
27   //die Strukturgleichheit wurde nie verletzt
28   return true;
29 }
```

Die einzige Schwierigkeit, die viele Studenten bei dieser Aufgabe hatten, war gleichzeitig durch zwei Kinderlisten zu iterieren. Hierzu bedarf es zweier Iteratoren, die aber in nur einer Schleife gleichzeitig immer um eins weitergeschaltet werden.

A.1.8 equals

`public boolean equals(Object other)`: soll genau dann wahr sein, wenn `other` ein Baum ist, der die gleiche Struktur und die gleichen Markierungen hat.

Wir können mit der obigen Lösung sehr schnell zu einer Lösung kommen. Entweder wir kopieren die Methode der Strukturgleichheit und fügen noch einen zusätzlichen Test ein, ob die Knotenmarkierungen gleich sind:

```
1  if (!this.mark().equals(other.mark())) return false;
```

Oder aber, wir prüfen erst auf Strukturgleichheit und vergleichen dann die Listen der Knoten beider Bäume auf Gleichheit.

Tree.java

```
2    public <b> boolean equals(Tree<b> other){
3        if (!this.sameStructure(other)) return false;
4        return this.flatten().equals(other.flatten());
5    }
```

Diese Lösung ist zwar schön kurz, aber durchläuft den Baum zweimal: einmal, um die Struktur zu testen und ein zweites Mal um die Markierungen auf Gleichheit zu testen.

In der Aufgabe war die Gleichheitsmethode mit dem Parameter des Typs `Object` verlangt. Hierzu wird erst der Parameter untersucht, ob er überhaupt ein Baum ist, und dann erst mit den eigentlichen Vergleich begonnen:

Tree.java

```
6    public boolean equals(Object other){
7        if (! (other instanceof Tree)) return false;
8        Tree<?> o = (Tree<?>) other;
9        return this.equals(o);
10   }
11 }//end der Klasse Tree
```

Dies ist eine typische Implementierung für die Gleichheit. Es soll in den meisten Fällen nur Gleiches mit Gleichen verglichen werden.

Anhang B

Hilfklasssen zu Swing

SwingWorker.java

```
1  package name.panitz.swing.threadTest;
2  import javax.swing.SwingUtilities;
3
4  /**
5   * This is the 3rd version of SwingWorker (also known as
6   * SwingWorker 3), an abstract class that you subclass to
7   * perform GUI-related work in a dedicated thread.  For
8   * instructions on using this class, see:
9   *
10  * http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html
11  *
12  * Note that the API changed slightly in the 3rd version:
13  * You must now invoke start() on the SwingWorker after
14  * creating it.
15  */
16 public abstract class SwingWorker {
17     private Object value;  // see getValue(), setValue()
18     private Thread thread;
19
20     /**
21      * Class to maintain reference to current worker thread
22      * under separate synchronization control.
23      */
24     private static class ThreadVar {
25         private Thread thread;
26         ThreadVar(Thread t) { thread = t; }
27         synchronized Thread get() { return thread; }
28         synchronized void clear() { thread = null; }
29     }
30
31     private ThreadVar threadVar;
32
33     /**
```

```
34      * Get the value produced by the worker thread, or null if it
35      * hasn't been constructed yet.
36      */
37     protected synchronized Object getValue() {
38         return value;
39     }
40 
41     /**
42      * Set the value produced by worker thread
43      */
44     private synchronized void setValue(Object x) {
45         value = x;
46     }
47 
48     /**
49      * Compute the value to be returned by the <code>get</code> method.
50      */
51     public abstract Object construct();
52 
53     /**
54      * Called on the event dispatching thread (not on the worker thread)
55      * after the <code>construct</code> method has returned.
56      */
57     public void finished() {
58     }
59 
60     /**
61      * A new method that interrupts the worker thread.  Call this method
62      * to force the worker to stop what it's doing.
63      */
64     public void interrupt() {
65         Thread t = threadVar.get();
66         if (t != null) {
67             t.interrupt();
68         }
69         threadVar.clear();
70     }
71 
72     /**
73      * Return the value created by the <code>construct</code> method.
74      * Returns null if either the constructing thread or the current
75      * thread was interrupted before a value was produced.
76      *
77      * @return the value created by the <code>construct</code> method
78      */
79     public Object get() {
80         while (true) {
81             Thread t = threadVar.get();
82             if (t == null) {
83                 return getValue();
```

```
84              }
85              try {
86                  t.join();
87              }
88              catch (InterruptedException e) {
89                  Thread.currentThread().interrupt(); // propagate
90                  return null;
91              }
92          }
93      }
94
95
96      /**
97       * Start a thread that will call the <code>construct</code> method
98       * and then exit.
99       */
100     public SwingWorker() {
101         final Runnable doFinished = new Runnable() {
102            public void run() { finished(); }
103         };
104
105         Runnable doConstruct = new Runnable() {
106             public void run() {
107                 try {
108                     setValue(construct());
109                 }
110                 finally {
111                     threadVar.clear();
112                 }
113
114                 SwingUtilities.invokeLater(doFinished);
115             }
116         };
117
118         Thread t = new Thread(doConstruct);
119         threadVar = new ThreadVar(t);
120     }
121
122     /**
123      * Start the worker thread.
124      */
125     public void start() {
126         Thread t = threadVar.get();
127         if (t != null) {
128             t.start();
129         }
130     }
131 }
```

Anhang C

Beispielaufgaben

Anhang D

Referate

Studenten wird die Gelegenheit gegeben, statt der Übungsaufgaben ein Referat auszuarbeiten und zu halten. Erwartet wird eine kurze schriftliche Ausarbeitung mit einem kleine Beispiel und ein halbstündiger Vortrag in der Übung.

Zu folgenden Themen sind im SS03 Referate gehalten worden:

D.1 JNI: Javas Schnittstelle zu C

Wie können Javaprogramme mit C-Programmen kommunizieren. Hierzu gibt es JNI (java native interface).

D.2 Java 2D Grafik

In Java gibt es ein umfangreiches Paket zum Erzeugen und zeichnen graphischer Objekte.

D.3 Java 3D Grafik

In Java gibt es ein umfangreiches Paket zum Erzeugen und zeichnen animierter dreidimensionaler graphischer Objekte.

D.4 Java und Netzwerk

Wie greift man mit Java auf Netzwerkresourcen zu.

D.5 Java und Datenbanken

Wie greift man mit Java auf eine Datenbank zu.

Anhang E

Gesammelte Aufgaben

Aufgabe 1 Die Methode `actionPerformed` der Schnittstelle `ActionListener` bekommt ein Objekt der Klasse `ActionEvent` übergeben. In dieser gibt es eine Methode `getWhen`, die den Zeitpunkt, zu dem das Ereignis aufgetreten ist als eine Zahl kodiert zurückgibt. Mit dieser Zahl können Sie die Klasse `java.util.Date` instanziieren, um ein Objekt zu bekommen, das Datum und Uhrzeit enthält.

Schreiben Sie jetzt eine Unterklasse von `JTB`, so daß beim Drücken des Knopfes, jetzt die aktuelle Uhrzeit im Textfeld erscheint.

Aufgabe 2 Schreiben Sie eine kleine Guianwendung mit einer Textfläche und drei Knöpfen, die einen erweiterten Zähler darstellt:

- einen Knopf zum Erhöhen eines Zählers.
- einen Knopf zum Verringern des Zählers um 1.
- einen Knopf zum Zurücksetzen des Zählers auf 0.

Aufgabe 3 Schreiben Sie Ihr Programm eines Zählers mit drei Knöpfen jetzt so, daß sie innere Klassen benutzen.

Aufgabe 4 (Punkteaufgabe 2 Punkte) Schreiben Sie eine GUI-Komponente `StrichKreis`, die mit geraden Linien einen Kreis entsprechend untenstehender Abbildung malt. Der Radius des Kreises soll dabei dem Konstruktor der Klasse `StrichKreis` als `int`-Wert übergeben werden.

Testen Sie Ihre Klasse mit folgender Hauptmethode:

```
1 public static void main(String [] args) {
2   StrichKreis k = new StrichKreis(120);
3   JFrame f = new JFrame();
4   JPanel p = new JPanel();
5   p.add(new StrichKreis(120));
6   p.add(new StrichKreis(10));
```

```
7     p.add(new StrichKreis(60));
8     p.add(new StrichKreis(50));
9     p.add(new StrichKreis(70));
10    f.getContentPane().add(p);
11
12    f.pack();
13    f.setVisible(true);
14  }
```

Hinweis: In der Klasse `java.lang.Math` finden Sie Methoden trigeometrischer Funktionen. Insbesondere `toRadians` zum Umrechnen von Gardwinkel in Bogenmaß sowie `sin` und `cos`.

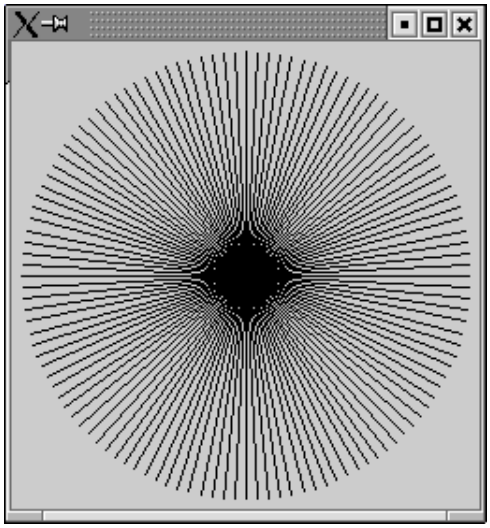


Abbildung E.1: Kreis mit Durchmesserlinien.

Aufgabe 5 Erweitern Sie die Strichkreisklasse aus der letzten Aufgabe, so daß jeder Strich in einer anderen Farbe gezeichnet wird.

Aufgabe 6 **(2 Punkte)** Implementieren Sie eine Analoguhr. Das Ziffernblatt dieser Uhr können Sie dabei mit den Methoden auf einem `Graphics`-Objekt zeichnen, oder Sie können versuchen eine Bilddatei zu laden und das Ziffernblatt als Bild bereitstellen. Als Anregung für ein Ziffernblatt können Sie sich das Bild des Ziffernblatts der handgefertigten Sekundenuhr (`http://www.panitz.name/prog2/examples/images/wiebking.jpg`) des Uhrmachers G.Wiebking herunterladen.

Aufgabe 7 Schreiben Sie eine Animation, in der zwei Kreise sich bewegen. Die Kreise sollen an den Rändern der Spielfläche abprallen und sie sollen ihre Richtung ändern, wenn sie sich berühren.

Aufgabe 8 Schreiben Sie eine Animation, deren Verhalten über Mausklicks beeinflussen können.

Aufgabe 9 Erzeugen Sie ein Baumobjekt, das einen Stammbaum Ihrer Familie darstellt. Die Knoten und Blätter sind mit Namen markiert. Die Wurzel ist mit Ihren Namen markiert. Kinderknoten sind mit den leiblichen Eltern der Person eines Knotens markiert.

Aufgabe 10 Testen Sie die in diesem Abschnitt entwickelten Methoden auf Bäumen mit dem in der letzten Aufgabe erzeugten Baumobjekt.

Aufgabe 11 Punkteaufgabe (4 Punkte):
Für diese Aufgabe gibt es maximal 4 auf die Klausur anzurechnende Punkte. Abgabetermin: wird noch bekanntgegeben.

Schreiben Sie die folgenden weiteren Methoden für die Klasse `Tree`. Schreiben Sie Tests für diese Methoden.

a) `List leaves()`: erzeugt eine Liste der Markierungen an den Blättern des Baumes.

b) `String show()`: erzeugt eine textuelle Darstellung des Baumes. Jeder Knoten soll eine eigene Zeile haben. Kinderknoten sollen gegenüber einem Elternknoten eingerückt sein.

Beispiel:

```
william
  charles
    elizabeth
    phillip
  diana
    spencer
```

Tipp: Benutzen Sie eine Hilfsmethode
`String showAux(String prefix)`,
die den String `prefix` vor den erzeugten Zeichenketten anhängt.

c) `boolean contains(a o)`: ist wahr, wenn eine Knotenmarkierung gleich dem Objekt `o` ist.

d) `int maxDepth()`: gibt die Länge des längsten Pfades von der Wurzel zu einem Blatt an.

e) `List<a> getPath(a o)`: gibt die Markierungen auf dem Pfad von der Wurzel zu dem Knoten, der mit dem Objekt `o` markiert ist, zurück. Ergebnis ist eine leere Liste, wenn ein Objekt gleich `o` nicht existiert.

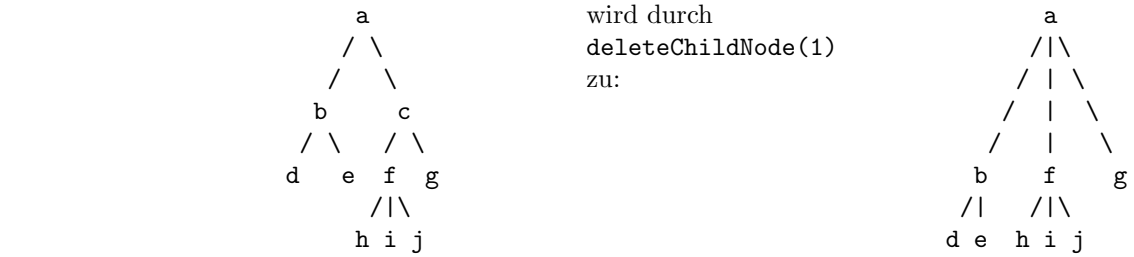
f) `java.util.Iterator<a> iterator()`: erzeugt ein Iterator über alle Knotenmarkierungen des Baumes.
Tipp: Benutzen Sie bei der Umsetzung die Methode `flatten`.

g) `public boolean equals(Object other)`: soll genau dann wahr sein, wenn `other` ein Baum ist, der die gleiche Struktur und die gleichen Markierungen hat.

h) `<b>boolean sameStructure(Tree<a> other)`: soll genau dann wahr sein, wenn der Baum `other` die gleiche Struktur hat. Die Markierungen der Knoten können hingegen unterschiedlich sein.

Aufgabe 12 Schreiben Sie eine Methode `List<Tree<a>> siblings()`, die die Geschwister eines Knotens als Liste ausgibt. In dieser Liste der Geschwister soll der Knoten selbst nicht auftauchen.

Aufgabe 13 Schreiben Sie eine modifizierende Methode
`void deleteChildNode(int n)`,
die den n-ten Kindknoten löscht, und stattdessen die Kinder des n-ten Kindknotens als neue Kinder mit einhängt.
Beispiel:



Aufgabe 14 Überschreiben Sie die Methoden `addLeaf` und `deleteChildNode` in der Klasse `BinTree`, so daß sie nur eine Ausnahme werfen, wenn die Durchführung der Modifikation dazu führen würde, daß das Objekt, auf dem die Methode angewendet wird, anschließend kein Binärbaum mehr wäre.

Aufgabe 15 Schreiben Sie analog zur Methode `flatten` in der Klasse `Tree` eine Methode `List<a> postorder()`, die die Knoten eines Baumes in Postordnung linearisiert.

Aufgabe 16 Rechnen Sie auf dem Papier ein Beispiel für die Arbeitsweise der Methode `military()`.

Aufgabe 17 Zeichnen Sie schrittweise die Baumstruktur, die im Programm `TestSearchTree` aufgebaut wird.

Aufgabe 18 Erzeugen Sie ein Objekt des typs `SearchTree` und fügen Sie nacheinander die folgenden Elemente ein:
`"anna","berta","carla","dieter","erwin","florian","gustav"`
Lassen Sie anschließend die maximale Tiefe des Baumes ausgeben.

Aufgabe 19

a) Schreiben sie entsprechend die Methode:
static public SearchTree rotateLeft(SearchTree t)

b) Schreiben Sie in der Klasse `BinTree` die entsprechende modifizierende Methode `rotateLeft()`

c) Testen Sie die beiden Rotierungsmethoden. Testen Sie, ob die Tiefe der Kinder sich verändert und ob die Inordnung gleich bleibt. Testen Sie insbesondere auch einen Fall, in dem die `NullPointerException` abgefangen wird.

Aufgabe 20 Erweitern Sie die obige Grammatik so, daß sie mit ihr auch geklammerte arithmetische Ausdrücke ableiten können. Hierfür gibt es zwei neue Terminalsymbolde: `(` und `)`.

Schreiben Sie eine Ableitung für den Ausdruck: `1+(2*20)+1`

Aufgabe 21 Betrachten Sie die einfache Grammatik für arithmetische Ausdrücke aus dem letzen Abschnitt. Zeichnen Sie einen Syntaxbaum für den Audruck `1+1+2*20`.

Aufgabe 22 **(2 Punkte)** In dieser Aufgabe sollen Sie den Parser für arithmetische Ausdrücke um geklammerte Ausdrücke erweitern. Hierzu sei die Grammatik für die Regel *multExpr* wie folgt geändert:

multExpr ::= *atomExpr multOp multExpr*|*atomExpr*
atomExpr ::= `(`*addExpr*`)`|*zahl*

Hierzu sei die entsprechende zusätzliche Tokenaufzählung für die zwei Klammersymbole gegeben:

Parentheses.java

```
1 package name.panitz.parser;
2 public enum Parentheses implements Token{lpar,rpar;}
```

Zusätzlich gegeben sei die die folgende Funktion:

DoParentheses.java

```
1  package name.panitz.parser;
2
3  public class DoParentheses
4    implements
5      Function<Pair<Parentheses,Pair<Integer,Parentheses>>,Integer>{
6    public Integer
7                apply(Pair<Parentheses,Pair<Integer,Parentheses>> x){
8      return x.getE2().getE1();
9    }
10 }
```

a) Schreiben Sie eine Klasse `AtomExprEval`, die `Parser<Integer>` implementiert und der Regel für *atomExpr* entspricht.

b) Ändern Sie die Klasse *MultExprEval*, so daß sie der geänderten Grammatikregel entspricht.

c) Erweitern Sie die Klasse `ArithTokenizer`, so daß er auch die beiden Klammersymbole erkennt.

Aufgabe 23 Fügen Sie der Datei `sms.jjt` folgende Zeilen am Anfang ein:

```
1 options {
2    MULTI=true;
3    STATIC=false;
4 }
```

Generieren Sie mit `jjtree` und `javac` den Parser neu. Was hat sich geändert?

Klassenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[ACL02] César F. Acebal, Raúl Izquierdo Castanedo, and Juan M. Cueva Lovelle. Good design principles in a compiler university course. *ACM SIGPLAN Notices*, 37(4):62–73, 2002.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions of Information Theory*, 2:113–124, 1956.

[CWH00] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial Third Edition*. Addison-Wesley Publishing Company, 2000.

[FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional languages. *The Computer Journal*, 32(2):108–121, 1989.

[GKS01] Ulrich Grude, Chritsoph Knabe, and Andreas Solymosi. Compilerbau. Skript zur Vorlesung, TFH Berlin, 2001. `www.tfh-berlin.de/~grude/SkriptCompilerbau.pdf`.

[Gru01] Ulrich Grude. Einführung in gentle. Skript, TFH Berlin, 2001. `www.tfh-berlin.de/~grude/SkriptGentle.pdf`.

[Joh75] Stephen C. Johnson. Yacc: Yet another compiler compiler. Technical Report 39, Bell Laboratories, 1975.

[NB60] Peter Naur and J. Backus. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, may 1960.

[Pan03] Sven Eric Panitz. Programmieren I. Skript zur Vorlesung, 2. revidierte Auflage, 2003. `www.panitz.name/prog1/index.html`.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer, 1985.