Programmieren II

SS 03

Prof. Dr. Sven Eric Panitz

TFH Berlin

Version 21. April 2004

Dieses Skript entstand begleitend zur Vorlesung des SS 03 vollkommen neu. Es stellt somit eine Mitschrift der Vorlesung dar. 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. Der XML-Quelltext verweist direkt auf ein XSLT-Skript, das eine HTML-Darstellung erzeugt, so daß ein entsprechender Browser mit XSLT-Prozessor die XML-Datei direkt als HTML-Seite darstellen kann.

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, 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.

Aufbau der Vorlesung

Die Vorlesung wird anhand ausgewählter Kernkapitel der Informatik beispielhaft deren Umsetzung in Java vorführen. Besonderes Gewicht wird dabei auf Baumstrukturen und formale Sprachen gelegt.

Kapitel 1

Klassenpfad und Java-Archive

In der Vorgängervorlesung haben wir Javaklassen einzeln übersetzt und jeweils pro Klasse eine Datei `.class` erhalten.[1] Ein Javaprogramm als solches läßt sich damit gar nicht genau eingrenzen. Ein Javaprogramm ist eine Klasse mit einer Hauptmethode zusammen mit der Menge aller Klassen, die von der Hauptmethode zur Ausführung benötigt werden. Selbst Klassen, die zwar im gleichen Paket wie die Hauptklasse mit der Hauptmethode liegen, werden nicht unbedingt vom Programm benötigt und daher auch nicht geladen. Ein Programm liegt also in vielen verschiedenen Dateien verteilt. Das kann unhandlich werden, wenn wir unser Programm anderen Benutzern bereitstellen wollen. Wir wollen wohl kaum mehrere hundert Dateien auf einen Server legen, die ein potentieller Kunde dann herunterlädt.

Aus diesem Grunde bietet Java eine Möglichkeit an, die Klassen eines Programms oder einer Bibliothek zusammen in einer Datei zu bündeln. Hierzu gibt es `.jar`-Dateien. `Jar` steht für *Java archive*. Die Struktur und Benutzung der `.jar`-Dateien leitet sich von den seit alters her in Unix bekannten `.tar`-Dateien ab, wobei `tar` für *tape achive* steht und ursprünglich dazu gedacht war, um Dateien gemeinsam auf einen Tonband-Datenträger abzuspeichern.

In einem Javaarchiv können nicht nur Klassendateien gespeichert werden, sondern auch Bilder und Sounddateien, die das Programm benötigt und zusätzlich Versionsinformationen.

1.1 Benutzung von jar-Dateien

`jar` ist zunächst einmal ein Programm zum Verpacken von Dateien und Ordnern in eine gemeinsame Datei, ähnlich wie es auch das Programm `zip` macht. Das Programm `jar` wird in einer Javainstellation mitgeliefert und kann von der Kommandozeile gestartet werden. Ein Aufruf ohne Argumente führt zu einer Hilfemeldung:

```
sep@swe10:~/fh/prog2> jar
Syntax: jar {ctxu}[vfmOMi] [JAR-Datei] [Manifest-Datei] [-C dir] Dateien ...
Optionen:
    -c  neues Archiv erstellen
    -t  Inhaltsverzeichnis für Archiv auflisten
    -x  benannte (oder alle) Dateien aus dem Archiv extrahieren
```

[1] Bei Benutzung anonymer und innerer Klassen konnte es hierbei auch zu mehreren `.class` Dateien kommen.

```
    -u  vorhandenes Archiv aktualisieren
    -v  ausführliche Ausgabe für Standardausgabe generieren
    -f  Namen der Archivdatei angeben
    -m  Manifestinformationen aus angegebener Manifest-Datei einbeziehen
    -0  nur speichern; keine ZIP-Komprimierung verwenden
    -M  keine Manifest-Datei für die Einträge erstellen
    -i  Indexinformationen für die angegebenen JAR-Dateien generieren
    -C  ins angegebene Verzeichnis wechseln und folgende Datei einbeziehen
Falls eine Datei ein Verzeichnis ist, wird sie rekursiv verarbeitet.
Der Name der Manifest-Datei und der Name der Archivdatei müssen
in der gleichen Reihenfolge wie die Flags ''m'' und ''f'' angegeben werden.

Beispiel 1: Archivieren von zwei Klassendateien in einem Archiv
mit dem Namen classes.jar:
       jar cvf classes.jar Foo.class Bar.class
Beispiel 2: Verwenden der vorhandenen Manifest-Datei ''meinmanifest''
und Archivieren aller
           Dateien im Verzeichnis foo/ in ''classes.jar'':
       jar cvfm classes.jar meinmanifest -C foo/ .

sep@swe10:~/fh/prog2>
```

Wie man sieht, gibt es eine Reihe von Optionen, um mit `jar` die Dateien zu erzeugen, ihren Inhalt anzuzeigen oder sie wieder auszupacken.

1.1.1 Erzeugen von Jar-Dateien

Das Kommando zum Erzeugen einer Jar-Datei hat folgende Form:

```
jar cf jar-file input-file(s)
```

Die Befehlsoptionen im einzelnen:

- Das `c` steht dafür, das eine Jar-Datei erzeugt werden soll.
- Das `f` steht dafür, daß der Dateiname der zu erzeugenden Jar-Datei folgt.
- `jar-file` ist der Name den die zu erzeugende Datei haben soll. Hier nimmt man üblerweise die Erweiterung `.jar` für den Dateinamen.
- `input-file(s)` ist eine Liste beliebiger Dateien und Ordner. Hier kann auch die bekannte `*`-Notation aus der Kommandozeile benutzt werden.

  Beispiel:
  Der Befehl

  ```
  jar cf myProg.jar *.class
  ```

  verpackt alle Dateien mit der Erweiterung `.class` in eine `Jar`-Datei namens `myProg.jar`.

1.1.2 Anzeige des Inhalts einer Jar-Datei

Das Kommando zur Anzeige der in einer Jar-Datei gespeicherten Dateien hat folgende Form:

```
jar tf jar-file
```

Die Befehlsoptionen im einzelnen:

- Das `t` steht dafür, das eine Auflistung des enthaltenen Dateien angezeigt werden[2] soll.
- Das `f` steht dafür, daß der Dateiname der benötigten Jar-Datei folgt.
- `jar-file` ist der Name existierenden Jar-Datei.

1.1.3 Extrahieren des Inhalts der Jar-Datei

Der Befehl zum Extrahieren des Inhalts einer JAR-Datei hat folgende schematische Form:

```
jar xf jar-file [archived-file(s)]
```

Die Befehlsoptionen im einzelnen:

- Die `x` Option gibt an, daß Dateien aus einer JAR-Datei extrahiert werden sollen.
- Das `f` gibt wieder an, daß der Name eine JAR-Datei folgt.
- `jar-file` ist der Name einer JAR-Datei, aus der die entsprechenden Dateien zu extrahieren sind.
- Optional kann noch eine Liste von Dateinamen folgen, die angibt, welche Dateien extrahiert werden sollen. Wird diese Liste weggelassen, so werden alle Dateien extrahiert.

Bei der Extraktion der Dateien werden die Datein in dem aktuellen Verzeichnis der Kommandozeile gespeichert. Hierbei wird die original Ordnerstruktur der Dateien verwendet, sprich Unterordner, wie sie in der JAR-Datei verpackt wurden werden auch als solche Unterordner wieder ausgepackt.

1.1.4 Ändern einer Jar Datei

Schließlich gibt es eine Option, die es erlaubt, eine bestehende JAR-Datei in ihrem Inhalt zu verändern. Der entsprechende Befehl hat folgendes Format:

```
jar uf jar-file input-file(s)
```

Die Befehlsoptionen im einzelnen:

- Die Option `u` gibt an, daß eine bestehende JAR-Datei geändert werden soll.
- Das `f` gibt wie üblich an, daß der Name eine JAR-Datei folgt.
- Eine Liste von Dateinamen gibt an, daß diese zur JAR-Datei hinzuzufügen sind.

Datein, die bereits in der JAR-Datei enthalten sind, werden durch diesen Befehl mit der neuen Version überschrieben.

[2] `t` steht dabei für *table*.

Aufgabe 1 Nehmen Sie eines Ihrer Javaprojekte des letzen Semesters (z.B. Eliza oder VierGewinnt) und verpacken die `.class` Dateien des Projektes in eine JAR-Datei. Berücksichtigen Sie dabei die Paketstruktur, die sich in der Ordnerhierarchie der `.class` Dateien wiederspiegelt.

1.2 Der Klassenpfad

JAR-Dateien sind nicht nur dazu gedacht, daß damit Javaanwendungen einfacher ausgetauscht werden können, sondern der Javainterpreter kann Klassen direkt aus der JAR-Datei starten. Eine JAR-Datei braucht also nicht ausgepackt zu werden, um eine Klasse darin auszuführen. Es ist dem Javainterpreter lediglich mitzuteilen, daß er auch in einer JAR-Datei nach den entsprechenden Klassen suchen soll.

1.2.1 Angabe des Klassenpfades als Option

Der Javainterpreter hat eine Option, mit der ihm angegeben werden kann, wo er die Klassen suchen soll, die er zur Ausführung der Anwendung benötigt. Die entsprechende Option des Javainterpreters heißt `-cp` oder auch `-classpath`, wie die Hilfmeldung des Javainterpreters auch angibt:

```
sep@swe10:~/fh/prog2> java -help -cp
Usage: java [-options] class [args...]
           (to execute a class)
   or  java -jar [-options] jarfile [args...]
           (to execute a jar file)

where options include:

    -cp -classpath <directories and zip/jar files separated by :>
                  set search path for application classes and resources
```

Es läßt sich nicht nur eine JAR-Datei für den Klassenpfad angeben, sondern mehrere und auch Ordner im Dateisystem, von denen aus die Paketstruktur der Javaanwendung bis hin zu dn Klassendateien befindet. Die verschiedenen Einträge des Klassenpfades werden bei der Suche einer Klasse von vorne nach hinten benutzt, bis die Klasse im Dateisystem oder in einer Jar-Datei gefunden wurde. Die verschiedenen Einträge des Klassenpfades werden durch einen Doppelpunkt getrennt.

Aufgabe 2 Starten Sie die Anwendung, die Sie in der letzten Aufgabe als JAR-Datei verpackt haben, mit Hilfe der `java`-Option: `-cp`.

1.2.2 Angabe des Klassenpfades als Umgebungsvariable

Implizit existiert immer ein Klassenpfad in ihrem Betriebssystem als eine sogenannte Umgebungsvariable. Sie können den Wert dieser Umgebungsvariable abfragen und ändern. Die Umgebungsvariable, die Java benutzt, um den Klassenpfad zu speichern heißt `CLASSPATH`. Deren Wert benutzt Java, wenn kein Klassenpfad per Option angegeben wird.

Windows und Unix unterscheiden sich leicht in der Benutzung von Umgebungsvariablen. In Unix wird der Wert einer Umgebungsvariable durch ein vorangestelltes Dollarzeichen bezeichnet, also `$CLASSPATH` in Windows wird sie durch Prozentzeichen eingeschlossen also `%CLASSPATH%`.

Abfrage des Klassenpfades

In der Kommandozeile kann man sich über den aktuellen Wert einer Umgebungsvariablen informieren.

Unix In Unix geschieht dieses leicht mit Hilfe des Befehls `echo`, dem die Variable in der Dollarnotation folgt:

```
sep@swe10:~/fh/prog2> echo $CLASSPATH
.:/home/sep/jarfiles/log4j-1.2.8.jar:/home/sep/jarfiles:
sep@swe10:~/fh/prog2>
```

Windows In Windows hingegen benutzt man den Konsolenbefehl `set`, dem der Name der Umgebungsvariablen folgt.

```
set CLASSPATH
```

Setzen des Klassenpfades

Innerhalb einer Eingabeaufforderung kann für diese Eingabeaufforderung der Wert einer Umgebungsvariablen geändert werden. Auch hierin unterscheiden sich Unix und Windows marginal:

Unix

Beispiel:
Wir fügen dem Klassenpfad eine weitere Jar-Datei an ihren Beginn an:

```
sep@swe10:~/fh/prog2> export CLASSPATH=~/jarfiles/jugs.jar:$CLASSPATH
sep@swe10:~/fh/prog2> echo $CLASSPATH
/home/sep/jarfiles/jugs.jar:.:/home/sep/jarfiles/log4j-1.2.8.jar:/home/sep/jarfiles:
```

Windows In Windows werden Umgebungsvariablen auch mit dem `set` Befehl geändert. Hiebei folgt dem Umgebungsvariablennamen mit einem Gleichheitszeichen getrennt der neue Wert.

Beispiel:
Der entsprechende Befehl des letzten Beispiels ist in Windows:
`set CLASSPATH=~\jarfiles\jugs.jar:%CLASSPATH%`

Aufgabe 3 Laden Sie die JAR-Datei:
jugs.jar (`http://www.tfh-berlin.de/~panitz//prog2/load/jugs.jar`) .
In diesem Archive liegt eine Javaanwendung, die eine interaktive Javaumgebung bereitstellt. Javaausdrücke und Befehle können eingegeben und direkt ausgeführt werden. Das Archiv enthält zwei Klassen mit einer Hauptmethode:

- `Jugs`: ein Kommandozeilen basierter Javainterpreter.
- `JugsGui`: eine graphische interaktive Javaumgebung.

Um diese Anwendung laufen zu lassen, wird ein zweites Javaarchive benötigt: die JAR-Datei `tools.jar`. Diese befindet sich in der von Sun gelieferten Entwicklungsumgebung.

Setzen Sie den Klassenpfad (einmal per Javaoption, einmal durch neues Setzen der Umgebungsvariablen `CLASSPATH`) auf die beiden benötigten JAR-Dateien und starten Sie eine der zwei Hauptklassen. Lassen Sie folgende Ausdrücke in *Jugs* auswerten.

- `2*21`
- `"hello world".toUpperCase().substring(2,5)`
- `System.getProperties()`
- `System.getProperty("user.name")`

Kapitel 2

Felder sind Eigenschaften zweiter Klasse

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 class Ente {
2   int beine = 2;
3   int getBeine(){return beine;}
4   int getWirklicheBeine(){return beine;}
5 }
```

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 class LahmeEnte extends Ente {
2   int beine = 1;
3   int getWirklicheBeine(){return beine;}
4 }
```

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  class EntenTest {
2    public static void main(String [] args){
3      Ente ente1 = new Ente();
4      LahmeEnte ente2 = new LahmeEnte();
5      Ente ente3 = ente2;
6
7      System.out.println("Ente ente1 = new Ente()");
8      System.out.println("beine: "+ente1.beine);
9      System.out.println("getBeine(): "+ente1.getBeine());
10     System.out.println(
11      "getWirklicheBeine(): "+ente1.getWirklicheBeine());
12
13     System.out.println(
14      "\nLahmeEnte ente2 = new LahmeEnte()");
15     System.out.println("beine: "+ente2.beine);
16     System.out.println("getBeine(): "+ente2.getBeine());
17     System.out.println(
18      "getWirklicheBeine(): "+ente2.getWirklicheBeine());
19
20     System.out.println("\nEnte ente3 = new LahmeEnte()");
21     System.out.println("beine: "+ente3.beine);
22     System.out.println("getBeine(): "+ente3.getBeine());
23     System.out.println(
24      "getWirklicheBeine(): "+ente3.getWirklicheBeine());
25   }
26 }
```

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

```
sep@swe10:~/fh/prog2/beispiele> java 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 class EntenTest2 {
2   public static void main(String [] args){
3     Ente ente = new LahmeEnte();
4     System.out.println(            ente .beine);
5     System.out.println(((LahmeEnte)ente).beine);
6   }
7 }
```

Die Ausgabe dieses Tests ist:

```
sep@swe10:~/fh/prog2/beispiele> java 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.

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 Zug.

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.

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: (**Object**,**List{Tree}**) →**Tree**

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.

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

- children: **Tree→List{Tree}**
- mark: **Tree→Object**

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

mark(Node(x,xs)) = x
children(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 Klassehierarchie

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

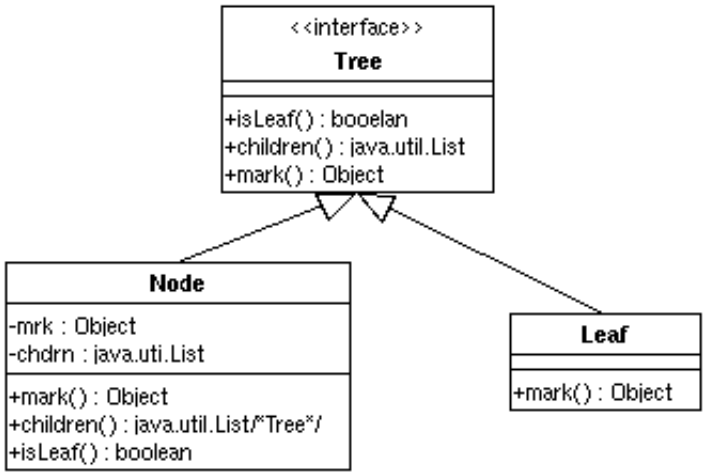


Abbildung 3.2: Modellierung von Bäumen mittels einer Klassenhierarchie.

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(Object mark)`: für die Kinder wird eine leere Liste erzeugt.

Wir erhalten folgende einfache Klasse:

```
1  import java.util.List;
2  public class Tree {
3
4    private Object mrk;
5    private List/*Tree*/ chldrn;
6
7    public Tree(Object mark,List children){
8      this.mrk=mark;
9      this.chldrn=children;
10   }
11
12   public Tree(List children){
13     this(null,children);
14   }
15
16   public Tree(Object mark){
17     this(mark,new java.util.ArrayList());
18   }
19
20   public boolean isLeaf(){
21     return chldrn.isEmpty();
22   }
23
24   public List/*Tree*/ children(){return chldrn;}
25   public Object mark(){return mrk;}
26
```

Aufgabe 4 Erzeugen Sie ein Baumobjekt, das einen Stammbaum Ihrer Familie darstellt. Die Knoten und Blätter sind mit Namen markiert. Die Wurzel sind Sie. 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.

Zum Iterieren über die Kinder des Knoten, wird der für die Javalisten bereitgestellte Iterator benutzt.

```
1  public int count(){
2    int result = 1; // Startwert 1 für diesen Knoten
3
4    //für jedes Kind
5    for (Iterator it=children().iterator();it.hasNext();)
6      //addiere die Knotenanzahl des Kindes zum Ergebnis
7      result=result+((Tree)it.next()).count();
8
9    return result;
10 }
```

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 (children().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 (children().size()==0) return result;
7
8    //ansonsten nimm implizit durch Operator + das
9    //toString() der Kinder
10   return result+children();
11 }
```

Auch wenn wir es nicht sehen, ist diese Methode rekursiv. Der Ausdruck `result+children()` 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.

```
1  public List flatten(){
2    //leere Liste für das Ergebnis
3    List result = new ArrayList();
4
5    //füge die Markierung dieses Knotens zur
6    //Ergebnisliste hinzu
7    result.add(mark());
8
9    //für jedes Kind
10   for (Iterator it=children().iterator();it.hasNext();)
11     //erzeuge dessen Knotenliste und füge alle deren
12     //Elemente zum Ergebnis hinzu
13     result.addAll(((Tree)it.next()).flatten());
14
15
16   return result;
17 }
```

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

Aufgabe 6 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(Object 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 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.

f) `java.util.Iterator 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) `boolean sameStructure(Tree 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 Modifizierende Methoden

Im letzen Abschnitt haben wir eine rein funktionale Umsetzung von Bäumen erarbeitet. Einmal erzeugte Bäume sind unveränderbar. 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.

addLeaf

Wir schreiben eine Methode, die an einen Knoten als letztes Kind ein neues Blatt einfügt:

```
1 public void addLeaf(Object o){
2   List chldrn = children();
3   chs.add(new Tree(o));
4 }
```

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

deleteChild

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

```
1 public void deleteChild(int n){
2    children().remove(n);
3 }
```

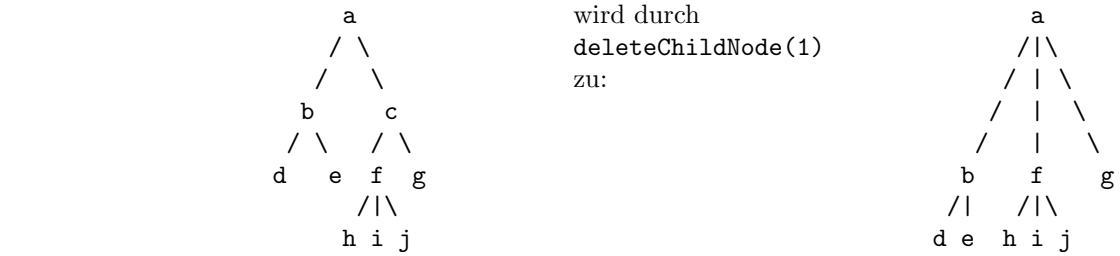
In dieser Implementierung wird die modifizierte Liste durch die Methode `children()` ermittelt, anstatt auf das Feld `chldrn` direkt zuzugreifen.

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:

```
1 public void setMark(Object mark){mrk=mark;}
```

Aufgabe 7 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:



3.2.5 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.

```
1 import java.util.List;
2
3 public class Tree {
4
5   private Object mrk;
```

```
6   private List/*Tree*/ chldrn;
7
8   public Tree parent=null;
9 ...
```

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:

```
1   public Tree(Object mark,List children){
2     //setze die entsprechenden Felder des Knotens
3     this.mrk=mark;
4     this.chldrn=children;
5
6     //es gibt noch keinen Elternknoten
7     this.parent=null;
8
9     //setze den neuen Knoten als Elternknoten der Kinder
10    respectParents();
11  }
12
13 public void respectParents(){
14   //für jedes Kind
15   for (Iterator it=children().iterator();it.hasNext();){
16    // ist dieser Knoten jetzt der Elternknoten
17    ((Tree)it.next()).parent=this;
18   }
19 }
```

Hier geraten wir mit den modifizierenden Methoden in einen doppelten Konflikt.

- Zum einen modifiziert der Konstruktor mit seinem Aufruf der Methode `respectParents` seine Kinder. Bäume können damit nur einmal als Unterbäume benutzt werden.
- Wir müssen die Implementierung der modifizierenden Methoden verändern, so daß sie den Elternknoten nach einer Modifikation für alle beteiligten Knoten korrekt setzen.

Aufgabe 8 Schreiben Sie eine Methode `List/*Tree*/ siblings()`, die die Geschwister eines Knotens als Liste ausgibt. In dieser Liste der Geschwister soll der Knoten selbst nicht auftauchen. Passen Sie auf, daß Sie keine modifizierende Methode schreiben.

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  import java.util.List;
2  import java.util.ArrayList;
3
4  public class BinTree extends Tree{
5
6    //private Felder für rechtes und linkes Kind
7    private BinTree rght;
8    private BinTree lft;
```

Wir sehen zwei naheliegende Konstuktoren vor:

BinTree.java

```
9    public BinTree(BinTree left,Object mark,BinTree right){
10     super(mark);
11     lft=left; rght=right;
12   }
13
14   public BinTree(Object mark){
15     this(null,mark,null);
16   }
```

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

BinTree.java

```
17   public BinTree left(){return lft;}
18   public BinTree right(){return rght;}
```

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

```
19   public java.util.List children(){
20     //erzeuge Ergebnisliste
21     java.util.List result = new java.util.ArrayList();
22
23     //wenn linkes bzw. rechtes Kind existieren,
24     //füge sie zum Ergebnis
25     if (left()!=null)  result.add(left());
26     if (right()!=null) result.add(right());
27
28     return result;
29   }
30 }
```

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 modzifizierenden 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 `children()` 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:

```
1 public void addLeaf(Object o){
2     throw new UnsupportedOperationException();
3 }
```

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

```
1 public void deleteChild(int n){
2   if (n==0) lft=null;
3   else if (n==1) rght=null;
4   else throw new IndexOutOfBoundsException ();
5 }
```

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

BinTree.java

```
1   void setLeft(BinTree l){ lft=l;}
2   void setRight(BinTree r){ rght=r;}
```

Aufgabe 9 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 10 Schreiben Sie analog zur Methode `flatten` in der Klasse `Tree` eine Methode `List 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

```
1  public List inorder(){
2    //Ergebnisliste
3    List result = new ArrayList();
4
5    //gibt es ein linkes Kind, füge dessen
6    //Zinearisierung hinzu
7    if (left()!=null) result.addAll(left().inorder());
8
9    //dann den Knoten selbst
10   result.add(mark());
11
12   //und gegebenenfalls dann das recte Kind
13   if (right()!=null) result.addAll(right().inorder());
14
15   return result;
16 }}
```

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 3.3 veranschaulicht diese Ordnung:

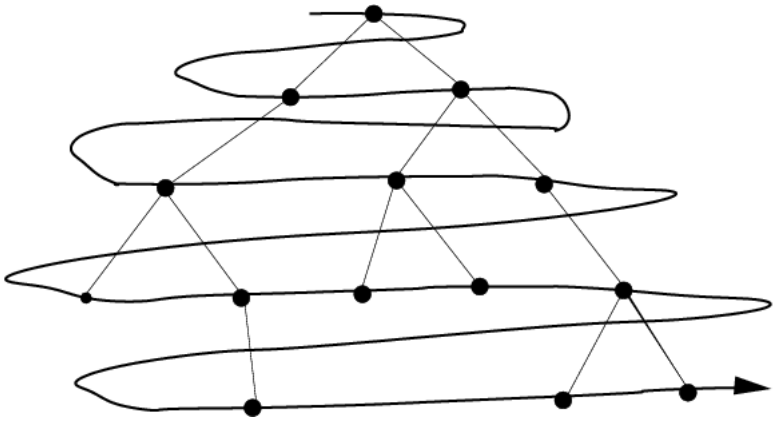


Abbildung 3.3: Schematische Darstellung der Militärordnung.

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.

```
1  public List military(){
2    //Ergebnisliste enthält erst diesen Knoten
3    List result=new ArrayList();
4    result.add(mark());
5
6    //Wir speichern uns die Knoten einer
7    //aktuellen Generation
8    List currentGeneration = children();
9
10   //Solange es in der aktuellen Generation Knoten gibt
11   while (!currentGeneration.isEmpty()){
12     //erzeuge die nächste Generation
13     List nextGeneration=new ArrayList();
14
15     for (Iterator it=currentGeneration.iterator()
16            ;it.hasNext();){
17       //indem für jedes Kind dessen Kinder darin
18       //aufgenommen werden
19       Tree nextChild = (Tree)it.next();
20       nextGeneration.addAll(nextChild.children());
21
```

```
22        //Füge das Elemente der aktuellen Generation
23        //zum Ergebnis
24        result.add(nextChild.mark());
25      }
26      //schalte eine Generation tiefer
27      currentGeneration=nextGeneration;
28    }
29    return result;
30  }
```

Aufgabe 11 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  public class SearchTree extends BinTree{
```

Wir sehen nur einen Konstruktor vor, der ein Blass erzeugt

SearchTree.java

```
2    SearchTree(Comparable o){
3      super(o);
4    }
```

Suchbäume wachsen durch eine modifizierende Einfügeoperation:

SearchTree.java

```
5   public void insert(Comparable o){
6     if (o.compareTo(mark())<=0){
7       //neuer Knoten ist kneiner, also in den
8       //linken Teilbaum
9       if (left()==null){
10        setLeft(new SearchTree(o));
11      }else
12        ((SearchTree)left()).insert(o);
13    }else{
14      //wenn der neue Knoten also größer, dann dasselbe
15      //im rechten Teilbaum
16      if (right()==null){
17         setRight(new SearchTree(o));
18      }else
19        ((SearchTree)right()).insert(o);
20    }
21  }
22
```

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

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

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  public class TestSearchTree {
2
3    public static void main(String [] args){
4      SearchTree t = new SearchTree("otto");
```

```
5       t.insert("sven");
6       t.insert("eric");
7       t.insert("lars");
8       t.insert("uwe");
9       t.insert("theo");
10      t.insert("otto");
11      t.insert("kai");
12      t.insert("henrik");
13      t.insert("august");
14      t.insert("berthold");
15      t.insert("arthur");
16      t.insert("arno");
17      t.insert("william");
18      t.insert("tibor");
19      t.insert("hassan");
20      t.insert("erwin");
21      t.insert("anna");
22      System.out.println(t.inorder());
23    }
24  }
```

Das Programm führt zu folgender Ausgabe.

```
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 12 Zeichnen Sie 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 ist sortiert:

[2] Die im philosophischen Sinne keine Induktion sondern eine Deduktion 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 $count(k') < n$.

  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

```
1  public boolean contains(Object o){
2    //bist du es schon selbst
3    if (mark().equals(o)) return true;
4
5    //links oder rechts suchen:
6    final int compRes = ((Comparable)mark()).compareTo(o);
7
8    if (compRes>0){
9      //keine Knoten mehr. Dann ist er nicht enthalten
10     if (left()==null) return false;
11     //Sonst such weiter unten im Baum
12     else return ((SearchTree)left()).contains(o);
13   }
14
15   if (compRes<0){
16     if (right()==null) return false;
17     else return ((SearchTree)right()).contains(o);
18   }
19   return false;
20 }
```

Wie man sucht, 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 13 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 3.4

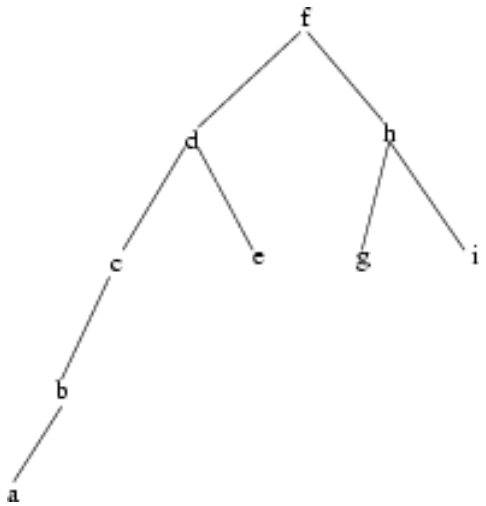


Abbildung 3.4: Nicht balanzierter Baum.

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 3.5

Das linke Kind dieses Baumes ist auch nicht ausbalanziert, wir können den Baum noch einmal verändern und erhalten den Baum in Abbildung 3.6:

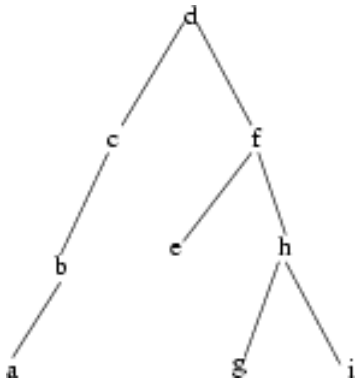


Abbildung 3.5: Baum nach Routierungsschritt.

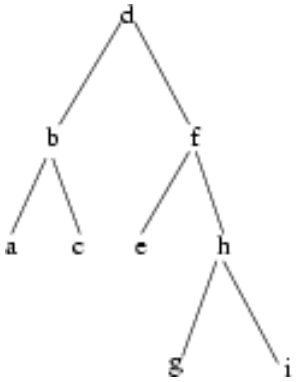
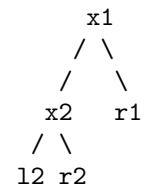


Abbildung 3.6: Baum nach weiteren Routierungsschritt.

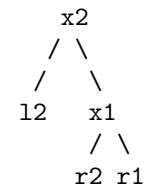
Spezifikation Zum Ausbalanzieren wird ein Baum gedreht. Einer seiner Kinderknoten wird die neue Wurzel und die alte Wurzel dessen Kind. Es gibt zwei Richtungen, in die gedreht werden kann: mit und gegen den Urhzeigersinn. Schematisch lassen sich diese beiden Operationen mit folgenden Gleichungen spezifizieren:

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

```
1  public static SearchTree rotateRight(SearchTree t){
2    try{
3      //speichere die einzelnen Teile in die Bezeichner,
4      //die wir in der Spezifikation benutzt haben
5      final SearchTree l2 = (SearchTree)t.left().left();
6      final SearchTree r2 = (SearchTree)t.left().right();
7      final SearchTree r1 = (SearchTree)t.right();
8      final Comparable x1 = (Comparable)t.mark();
9      final Comparable x2 = (Comparable)t.left().mark();
10
11     return new SearchTree
12                  (l2,x2,new SearchTree(r2,x1,r1));
13   }catch (NullPointerException _){
14       return t;
15   }
16 }
```

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

```
1  void rotateRight(){
2    try{
3      BinTree l1 = left();
4      BinTree l2 = l1.left();
5      BinTree r2 = l1.right();
6      BinTree r1 = right();
7      Object  x1 = mark();
8      Object  x2 = l1.mark();
9
10     setMark(x2);
```

```
11      setLeft(l2);
12      setRight(l1);
13      l1.setMark(x1);
14      l1.setLeft(r2);
15      l1.setRight(r1);
16    }catch (NullPointerException _){
17    }
18  }}
```

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 14

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.

3.5 XML-Dokumente als Bäume

XML Dokumente sind Bäume. Es gibt zwei Arten von Knoten[3]:

- **Textknoten**: Diese Knoten sind immer Blätter und mit einem Text markiert.
- **Elementknoten**: Diese Knoten können eine beliebige Anzahl von Kindern haben, oder auch Blätter sein. Elementknoten haben auch einen String als Markierung, den sogenannten `tag`.

3.5.1 Modellierung/Implementierung von XML-Bäumen

Im Gegensatz zu den allgemeinen Bäumen aus den letzten Abschnitten, modellieren wir die XML-Dokumente mit mehreren Klassen. Wir definieren eine allgemeine Schnittstelle für `XML` und definieren Klassen, die die verschiedenen Knotentypen eines XML-Dokuments ausdrücken können. Somit können wir auch leicht weitere Knotentypen z.B. für *processing instructions* hinzufügen.

Die Schnittstelle sei zunächst eine leere Schnittstelle.

XML.java

```
1  public interface XML {}
```

[3]Es gibt in XML noch weitere Knotentypen (Attribute, Processing-Instructions, Kommentare) die wir vorerst außer acht lassen wollen.

Elemente

Wir implementieren die Schnittstelle mit einer Klasse, die Elementknoten darstellt. Elementknoten können weitere XML-Dokumente als Kinder haben und haben einen Tagnamen.

Element.java

```
1  import java.util.List;
2  import java.util.ArrayList;
3  public class Element implements XML{
4    public String tagName;
5    public List/*XML*/ children;
6
7    public Element(String tagName,List/*XML*/ children){
8      this.tagName=tagName;
9      this.children=children;
10   }
11
12   public Element(String tagName){
13     this(tagName,new ArrayList());
14   }
15 }
```

Textknoten

Der zweite Knotentyp eines XML-Dokuments ist ein Textknoten. Sie können keine Kinder haben.

TextNode.java

```
1  public class TextNode implements XML{
2    public String text;
3    public TextNode(String text){this.text=text;}
4  }
```

3.5.2 Serialisierung von XML

Oben haben wir die logische Struktur von XML-Dokumneten in Form von Bäumen spezifiziert. XML-Dokumente haben eine textuelle Form. Man spricht auch von der Serialisierung, denn die logische Baumstruktur wird hier in eine serielle Folge von Zeichen ausgedrückt. Die beiden Knotentypen eines XML-Dokuments werden wie folgt serialisiert:

- **Elementknoten:** Die Serialisierung geht nach den Schema:

  <*tagname*>*Serialisierung der Kinder*</*tagname*>

  Dabei wird kein Trennzeichen zwischen den Serialisierungen der Kinder geschrieben. Sie stehen direkt hintereinander.

- **Textknoten:** Hier besteht die Serialisierung lediglich aus dem Text, mit dem der Knoten markiert ist.

Aufgabe 15 Erzeugen Sie ein Objekt des Typs `XML`, das folgende Serialisierung hat:

```
<drama>
 <title>Macbeth</title>
 <author>William Shakespear</author>
 <personae>
  <persona>Macbeth</persona>
  <persona>Lady Macbeth</persona>
  <persona>Duncan</persona>
  <persona>Banquo</persona>
  <persona>Macduff</persona>
  <persona>Lady Macduff</persona>
  <persona>Ross</persona>
 </personae>
</drama>
```

Aufgabe 16 Überschreiben Sie in den Klassen `Element` und `TextNode` die Methode `toString` so, daß sie die Serialisierung von XML als Ergebnis hat.

Schreiben Sie Tests für ihre neue Methode `toString`

3.6 Keller

Eine häufig in der Informatik auftretende Datenstruktur ist der Keller, auch als Stapel bekannt, englisch *stack*.[4]

Keller finden sich überall in der Informatik. Insbesondere haben wir schon den Aufrufkeller von Java kennengelernt.

Im Prinzip ist ein Keller nicht viel anderes als unsere funktionalen Liste aus der ersten Vorlesung. In unseren Listen haben wir auch immer das als letztes eingefügte Element als erstes gesehen. Und genau dieses ist das Prinzip eines Kellers: wenn ein Keller ausgeräumt wird, so kommt als erstes das zum Vorschein, was wir als letztes hineingetan haben. Das was als erstes in einen Keller getan wurde, kann erst wieder aus dem Keller hervorgeholt werden, wenn alles andere zuvor entfernt wurde.

3.6.1 formale Spezifikation

push: (Keller,Object) → Keller
pop: Keller → (Keller,Object)

pop(push(keller,x)) = (keller,x)

[4]Die Bezeichnung *Keller* ist seit Jahrzehnten in der Informatik eingeführt und keine Erfindung von mir. In der französischen Literatur findet sich übrigens der Ausdruck *pile*.

3.6.2 Implementierung

Keller.java

```
1  public class Keller {
2    public static final int MAX_ELEMENTS = 10000;
3    private Object [] keller = new Object[MAX_ELEMENTS];
4    private int stackPointer = 0;
5
6    public void push(Object x){
7      keller[stackPointer] = x;
8      stackPointer = stackPointer+1;
9    }
10
11   public Object pop(){
12     stackPointer = stackPointer -1;
13     return keller[stackPointer];
14   }
15 }
```

Kapitel 4

Guis und graphische Objekte

Grundtechniken der GUI-Programmierung mit dem Paket `javax.swing` haben wir bereits im Skript des ersten Semesters beschrieben. In diesem Kapitel soll anhand einiger Beispiele der Umgang mit graphischen Objekten geübt werden.

4.1 Fenster und Fensterereignisse

Als kleine Wiederholung aus dem letzten Semester betrachten wir die Handhabung von Fenstern in Javas Swing Paket. Hierzu steht die Klasse `javax.swing.JFrame` zur Verfügung. Diese Klasse stellt eine graphische Komponente dar, die im Betriebssystem als ein Fenster dargestellt wird.

4.1.1 Erzeugen und Öffnen von Fenstern

Ein Objekt dieses Typs kann sichtbar gemacht werden mit der Methode `setVisible`. Dem Fenster kann ein graphisches Objekte `o` als Fensterinhalt hinzugefügt werden durch den Aufruf von `getContentPane().add(o)`. Durch den Aufruf der Methode `pack` wird dafür gesorgt, daß das Fenster sich entsprechend der Größe seines Inhalts in der Größe ausrichtet.

Folgendes kleine Programm fügt einem Fenster einen Knopf zu und zeigt dieses Fenster auf dem Bildschirm.

OpenFrame.java

```
1  import javax.swing.*;
2  class OpenFrame{
3    public static void main(String [] args){
4      //erzeuge ein neues Fensterobjekt
5      JFrame f = new JFrame("Fenster testen");
6
7      //füge einen neu erzeugten Knopf als
8      //Fensterinhalt hinzu
9      f.getContentPane().add(new JButton("ein knopf"));
10
11     //lass das Fenster sich der Größe seines Inhalts
```

```
12      //entsprechend in der Größe dimensionieren
13      f.pack();
14
15      //mach das Fenster auf dem Bildschirm sichtbar
16      f.setVisible(true);
17    }
18  }
```

4.1.2 Reagieren auf Fensterereignisse

Graphische Objekte sollen in der Regel auf bestimmte Eingabeereignisse auf bestimmte Weise reagieren. Hierzu gibt es für verschiedene Ereignisse *Listener*-Klassen. In solchen Klassen kann spezifiziert werden, auf welche Weise ein Programm auf unterschiedliche Ereignisse reagiert. Es gibt *Listener*-Klassen für alle möglichen Ereignisarten z.B.:

- Mausereignisse (Mausknopf gedrückt, Maus bewegt sich etc.)
- Tastaturereignisse (bestimmte Tasten werden gedrückt)
- Fensterereigniss (Fenster wir geöffnet, geschlossen, minimiert)

Die Klasssen der graphischen Komponenten halten Methoden bereit, die es erlauben, den graphischen Objekten spezifische *Listener*-Objekte hinzuzufügen.

Eine sehr häufig benutzte Ereignisart für Fenster ist, daß das Programm komplett beendet wird, sobald das Fenster geschlossen wird. Hierzu können wir eine Unterklasse der Klasse `JFrame` schreiben, die auf Schließen des Fensters mit einem Programmabbruch reagiert:

ClosingFrame.java

```
1  package de.tfhberlin.panitz.gui;
2  import javax.swing.JFrame;
3  import java.awt.event.*;
4
5  public class ClosingFrame extends JFrame{
6    public ClosingFrame(String titel){
7      this();
8      setTitle(titel);
9    }
10
11   public ClosingFrame(){
12     //Füge als Listener hinzu
13     addWindowListener(
14       //Ein Objekt einer anonymen FensterListener Klasse
15       new WindowAdapter(){
16         //in dem auf das Schließen des Fensters mit einem
17         //Programmabbruch reagiert wird
18         public void windowClosing(WindowEvent e) {
19           System.exit(0);
20       }
21     });
```

```
22    }
23  }
```

Im Konstruktor wird dem neu konstruierten Fensterobjekt hier gleich ein Ereignisbehandlungsobjekt hinzugefügt, das beim Schließen des Fensters das Programm beendet. Dieses *Listener*-Objekt wird dabei als anonyme innere Unterklasse der Klasse `WindowAdapter` definiert.

Unsere neue eigene Fensterklasse können wir in gleicher Weise benutzen wie die Klasse `JFrame`:

OpenClosingFrame.java

```
1  package de.tfhberlin.panitz.gui;
2  import javax.swing.*;
3  import de.tfhberlin.panitz.gui.ClosingFrame;
4
5  class OpenClosingFrame{
6    public static void main(String [] args){
7      JFrame f = new ClosingFrame("Fenster testen");
8      f.getContentPane().add(new JButton("ein knopf"));
9      f.pack();
10     f.setVisible(true);
11   }
12 }
```

Schließen wir hier das Fenster, dann wird das Programm beendet. Wir erkennen das daran, daß auf der Kommandozeile wieder Eingaben getätigt werden können.

4.2 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 import java.util.List;
2 import java.util.ArrayList;
3 import java.util.Iterator;
4 import java.util.Enumeration;
5 import javax.swing.tree.TreeNode;
6
7 public class Tree implements TreeNode{
```

Probleme bereitet die Methode `children`. Sie hat in unserer Klasse `Tree` einen anderen Rückgabetyp als in der Schnittstelle. Hier sind wir gezwungen tatsächlich unsere Methode `children` zu ändern. Hierzu ändern wir den Namen der Methode `children` in der Klasse `Tree` zu: `theChildren`.

```
1     public List/*Tree*/ theChildren(){return chldrn;}
```

Entsprechend ist jeder Aufruf der Methode `children` in einen Aufruf der Methode `theChildren` zu ändern.

Jetzt können wir die Methode `children` mit der in der Schnittstelle verlangten Signatur implementieren. 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.

```
1   public java.util.Enumeration/*Tree*/ children() {
2     final Iterator it = theChildren().iterator();
3
4     return new Enumeration(){
5       public boolean hasMoreElements() {
6         return it.hasNext();
7       }
8       public Object nextElement() {return it.next();}
9     };
10  }
```

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

```
  public TreeNode getChildAt(int childIndex) {
    return (TreeNode)theChildren().get(childIndex);
  }

  public int getChildCount() {
   return theChildren().size();
  }

  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.

```

  public boolean getAllowsChildren(){
    throw new UnsupportedOperationException();
  }

  public int getIndex(TreeNode node) {
    throw new UnsupportedOperationException();
  }
```

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

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

Somit haben wir unsere Klasse `Tree` soweit massiert[1], 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. Zunächst erzeugen wir einen darzustellenden Baum, z.B. in Form eines Suchbaumes:

ExampleSearchTree.java

```
class ExampleSearchTree{
  public static SearchTree getExampleTree(){
    SearchTree t = new SearchTree("uwe");
    t.insert("sven");
```

[1] Die komplette Klasse ist im Anhang noch einmal abgedruckt.

```
5     t.insert("eric");
6     t.insert("lars");
7     t.insert("uwe");
8     t.insert("theo");
9     t.insert("otto");
10    t.insert("kai");
11    t.insert("tibor");
12    t.insert("hassan");
13    t.insert("erwin");
14    t.insert("anna");
15    t.insert("august");
16    t.insert("william");
17    t.insert("oscar");
18    t.insert("dario");
19    t.insert("erni");
20    t.insert("bert");
21    t.insert("krümel");
22    t.insert("grobi");
23    t.insert("bibo");
24    return t;
25  }
26 }
```

Dieser kann jetzt einem Objekt des Typs `JTree` übergeben werden. Das so erzeugte Objekt des Typs `JTree` kann in einem Fenster dargestellt werden.

```
1  import javax.swing.JFrame;
2  import javax.swing.JTree;
3
4  public class JTreeTest {
5
6    public static void main(String [] args){
7      JFrame frame = new JFrame("Baumtest Fenster");
8      frame
9        .getContentPane()
10       .add(new JTree(SearchTree.getExampleTree()))
11     ;
12
13     frame.pack();
14     frame.setVisible(true);
15   }
16
17 }
```

Diese Methode erzeugt die graphische Baumdarstellung aus Abbildung 4.1.

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.

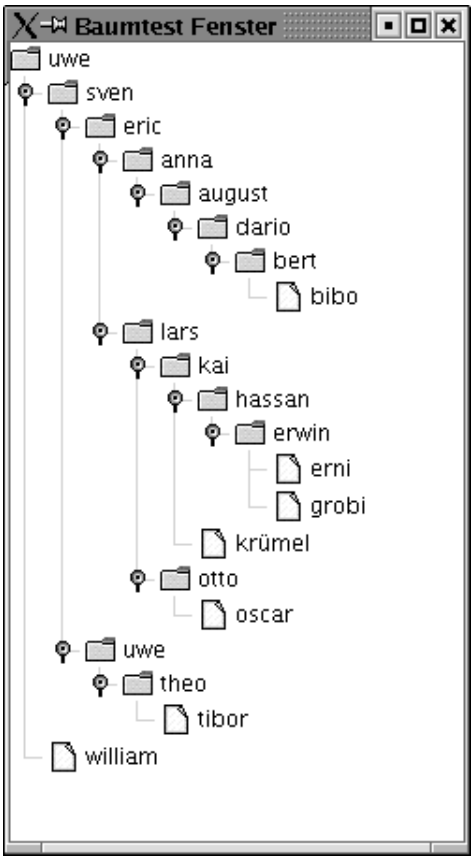


Abbildung 4.1: Baumdarstellung mit JTree.

4.3 Selbstdefinierte graphische Komponenten

Bisher haben wir Bäume unter Verwendung der fertigen GUI-Komponente `JTree` aus der Swing-Bibliothek graphisch dargestellt. Dabei wurde die aus einem Datei-Browser bekannte Darstellung benutzt. An der Tafel sind wir eine andere graphische Darstellung von Bäumen gewohnt. Da es hierfür keine fertige GUI-Komponente in der Swing-Bibliothek gibt, müssen wir eine entsprechende Komponente selbst schreiben[2].

Bevor wir unsere eigene GUI-Komponente für Bäume schreiben, schauen wir uns die Klasse `JComponent` einmal genauer an.

4.3.1 Graphics Objekte

Um eine eigene GUI-Komponente zu schreiben, schreibt man eine Klasse, die von der Klasse `JComponent` ableitet. Dieses haben wir bereits in der letzten Vorlesung getan. Die dort geschriebenen Unterklassen der Klasse `JComponent` zeichneten sich dadurch aus, daß sie eine

[2]Wir könnten natürlich auch nach einer entsprechenden Bibliothek im Internet suchen. Aber wir wollen ja lernen, wie eine solche Komponente geschrieben werden kann.

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 `paint`.[3] In dieser Methode wird festgelegt, was zu zeichnen ist, wenn die graphische Komponente darzustellen ist. Die Methode `paint` hat folgende Signatur:

```
1 public void paint(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 `paint` überschreiben und auf dem übergebenen Objekt des Typs `Graphics` entsprechende Methoden zum Zeichnen aufrufen.

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

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

[3] Ich muß gestehen, hier macht sich bei mir jetzt auch etwas Verwirrung breit. In den Swing Komponenten soll man laut Suns Tutorial nicht die Methode `paint` sondern die Methode `paintComponent` überschreiben. Es gibt aber auch wiederum Swing Komponenten, die gar nicht von `JComponent` ableiten und somit auch nicht die Methode `paintComponent` haben, nämlich die Klasse `JApplet`. Das Durcheinander mit den `java.awt`-Paket und dem `javax.swing`-Paket kann leider immer wieder zur Verwirrung führen.

UseSimpleGraphics.java

```
1  import javax.swing.JFrame;
2
3  class UseSimpleGraphics {
4    public static void main(String [] args){
5      JFrame frame = new JFrame();
6      frame
7        .getContentPane()
8        .add(new SimpleGraphics());
9      ;
10
11     frame.pack();
12     frame.setVisible(true);
13   }
14 }
```

Wird dieses Programm gestartet, so öfnet Java das Fenster aus Abbildung 4.2 auf dem Bildschirm.

Abbildung 4.2: Einfache graphische Komponente.

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

4.3.2 Dimensionen

Ärgerlich in unserem letzen 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; wie in Abbildung 4.3 zu sehen.

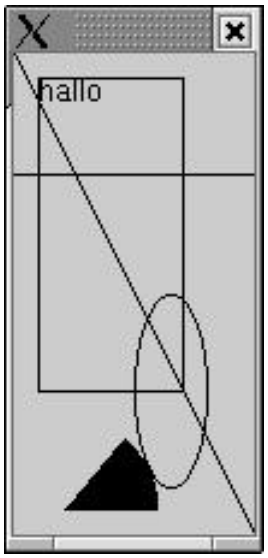


Abbildung 4.3: Graphische Komponente unter Berücksichtigung ihrer Größe.

4.3.3 JComponent, JPanel, Canvas, JApplet

Wir haben im letzten Abschnitt immer eine Unterklasse der Klasse `JComponent` geschrieben und die Methode `paint` überschrieben. In Suns Swing Handbuch steht, man solle für diese Zwecke eine Unterklasse der Klasse `JPanel` schreiben, wobei `JPanel` selbst wieder eine Unterklasse von `JComponent` ist. Bei manchen der nachfolgenden Beispiele hat dieses allerdings seltsame Effekte zur Folge gehabt. Eine Alternative ist, die Klasse `JApplet` zu erweitern. Dann haben alle Beispiele gut funktioniert. Suns Beispiele zu eigenen graphischen Objekten sind auch über die Erweiterung der Klasse `JApplet` geschrieben. Dies scheint also ein gutes Mittel der Wahl zu sein. Im AWT-Paket gibt noch die Klasse `Canvas`, die eine geeignete Klasse zum Erweitern war. Allerdings gibt es im Swing-Paket keine analoge Klasse `JCanvas`.

Das alles ist sehr verwirrend allein wegen der Vielzahl der Klassen, die in einer graphischen Bibliothek auftreten. Wenn mich in diesem Fall insbesondere jemand aufklären kann, wäre ich sehr dankbar. In den nachfolgenden Beispielen werden wir für eigene graphische Objekte die Klasse `JApplet` erweitern, auch wenn wir gar kein Applet schreiben wollen.

Aufgabe 17 (Punkteaufgabe 3 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 ClosingFrame();
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`, `sin` und `cos`.

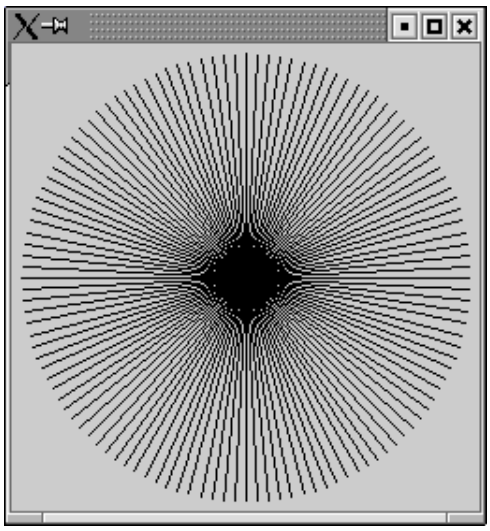


Abbildung 4.4: Kreis mit Durchmesserlinien.

4.3.4 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  import javax.swing.JApplet;
2  import javax.swing.JFrame;
3
4  public class ColorTest extends JApplet{
5    public ColorTest(){
6      //Hintergrundfarbe auf einen dunklen Grauton setzen
7      setBackground(new java.awt.Color(100,100,100));
8    }
9
10   public void paint(java.awt.Graphics g){
11     //Farbe auf einen dunklen Grünton setzen
12     g.setColor(new java.awt.Color(22,178,100));
13     //Rechteck zeichnen
14     g.fillRect(25,50,50,100);
15     //Farbe auf rot setzen
16     g.setColor(java.awt.Color.RED);
17     g.fillOval(25,50,50,100);
18     //Farbe auf gelb setzen
19     g.setColor(java.awt.Color.YELLOW);
20     g.fillOval(37,75,25,50);
21   }
22
23   public java.awt.Dimension getPreferredSize() {
24     return new java.awt.Dimension(100,200);
25   }
26
27   public static void main(String [] _){
28     javax.swing.JFrame f = new JFrame("Farben");
29     f.getContentPane().add(new ColorTest());
30     f.pack();f.setVisible(true);
31   }
32 }
```

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 de.tfhberlin.panitz.gui.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 de.tfhberlin.panitz.gui.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.JApplet;
8  import de.tfhberlin.panitz.gui.ClosingFrame;
9
10 public class Apfelmaennchen extends JApplet {
```

Als erstes deklarieren wir Konstanten für die Größe des Apfelmännchens.

Apfelmaennchen.java

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

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

Apfelmaennchen.java

```
13   final 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

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

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

Apfelmaennchen.java

```
15   final double startX = -2;
16   final double startY = -1.35;
```

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

Apfelmaennchen.java

```
17  final int recDepth = 50;
18  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

```
19  //C-Werte checken nach zn+1 = zn*zn + c,
20  public int checkC(Complex c) {
21    Complex zn = new Complex(0,0);
22
23    for (int n=0;n<recDepth;n=n+1) {
24      final Complex znp1 = zn.mult(zn).add(c);
25      if (znp1.norm() > schwellwert) return n;
26      zn=znp1;
27    }
28    return recDepth;
29  }
```

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

```
30  public void paint(Graphics g) {
31    for (int y=0;y<height;y=y+1) {
32      for (int x=0;x<width;x=x+1) {
33
34        final Complex current
35          =new Complex(startX+x*zelle,startY+y*zelle);
36
37        final int iterationenC = checkC(current);
38
39        paintColorPoint(x,y,iterationenC,g);
40      }
41    }
42  }
```

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

```
43  private void paintColorPoint
44                (int x,int y,int it,Graphics g){
45    final Color col
46      = it==recDepth
47        ?colAppleman
```

```
48          :new Color(255-5*it%1,255-it%5*30,255-it%5* 50);
49      g.setColor(col);
50      g.drawLine(x,y,x,y);
51    }
```

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

Apfelmaennchen.java

```
52    public Dimension getPreferredSize(){
53      return new Dimension(width,height);
54    }
55
56    public static void main(String [] args){
57      JFrame f = new ClosingFrame();
58      f.getContentPane().add(new Apfelmaennchen());
59      f.pack();
60      f.setVisible(true);
61    }
62  }
```

Das Programm ergibt das Bild aus Abbildung 4.5.

4.3.5 Fonts und ihre Metrik

Bevor wir nun endlich damit anfangen können unsere Bäume zu zeichnen, wie wir es gewohnt sind, müssen wir uns noch Gedanken über Fonts machen. Die Knoten des Baumes werden wir mit ihrer Markierung beschriften. Die Größe der Beschriftung jedes einzelnen Knotens bestimmt die Ausrichtung des gesammten Baumes; die Beschriftungen von Geschwisterknoten sollen sich ja nicht überlappen.

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  import java.awt.Font;
2  import java.awt.FontMetrics;
3  import java.awt.Graphics;
4  import java.awt.Dimension;
5  import javax.swing.JApplet;
6  import de.tfhberlin.panitz.gui.ClosingFrame;
```

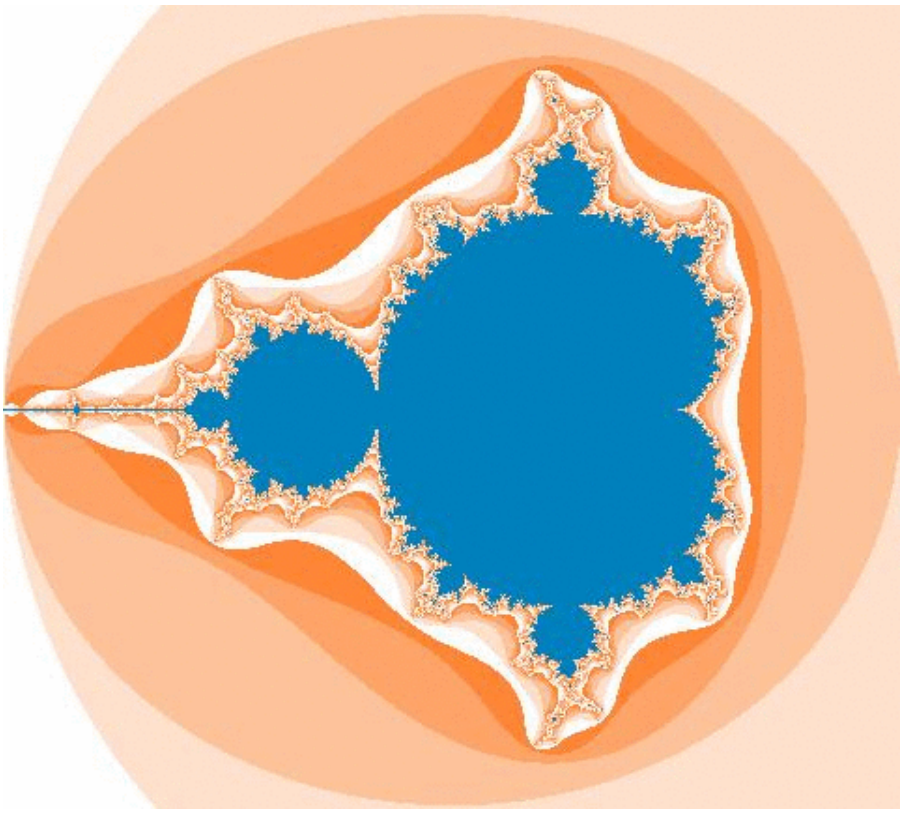


Abbildung 4.5: Apfelmännchen.

```
7
8   class FontTest extends JApplet {
9
10    public Dimension getPreferredSize(){
11      return new Dimension(250,650);
12    }
13
14    public void paint(Graphics g){
15      //die y-Koordinate für einen Text
16      int where = 0;
17
18      //25 mal
19      for (int i=1; i<=25;i=i+1){
20        //nimm aktuellen Font und seine Maße
21        final Font f = g.getFont();
22        final FontMetrics fm = g.getFontMetrics(f);
23
24        //gehe die Höhe des Fonts weiter in Richtung x
25        where = where + (f.getSize()) ;
26
27        //male "hallo"
```

```
28        g.drawString("hallo",0,where);
29
30        //berechne wo der gerade gemalte String endet
31        //und male dort "welt"
32        g.drawString
33          ("welt!",fm.stringWidth("hallo "),where);
34
35        //verändere die Fontgröße. Erhöhe sie um 1.
36        g.setFont(getFont().deriveFont(f.getSize2D()+1));
37      }
38    }
39
40    public static void main(String [] _){
41      ClosingFrame f = new ClosingFrame("Fonts");
42      f.getContentPane().add(new FontTest());
43      f.pack();
44      f.setVisible(true);
45    }
46  }
```

4.3.6 Bäume zeichnen

Jetzt kennen wir tatsächlich alle Klassen, die wir brauchen um Bäume zu zeichnen.

DisplayTree.java

```
47  import java.util.Enumeration;
48
49  import javax.swing.JComponent;
50  import javax.swing.tree.TreeNode;
51
52  import java.awt.Font;
53  import java.awt.FontMetrics;
54  import java.awt.Color;
55  import java.awt.Graphics;
56  import java.awt.Dimension;
57
58  public class DisplayTree extends JComponent{
```

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

DisplayTree.java

```
59    private TreeNode treeModell;
```

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

DisplayTree.java

```
60    static final public int VERTICAL_SPACE = 50;
61    static final public int HORIZONTAL_SPACE = 20;
```

In einem Feld werden wir uns die Maße des benutzten Fonts vermerken.

DisplayTree.java

```
62    private FontMetrics fontMetrics = null;
```

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

DisplayTree.java

```
63  private int height=0;
64  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

```
65  private boolean dimensionCalculated = false;
```

Es folgen zwei naheliegende Konstruktoren.

DisplayTree.java

```
66  public DisplayTree(TreeNode tree){treeModell=tree;}
67
68  public DisplayTree(TreeNode t,FontMetrics fm){
69    treeModell=t;fontMetrics=fm;
70  }
```

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

DisplayTree.java

```
71  private void calculateDimension(){
```

Zunächst stellen wir sicher, daß die Fontmaße bekannt sind:

DisplayTree.java

```
72
73    if (fontMetrics==null){
74      final Font font = getFont();
75      fontMetrics = getFontMetrics(font);
76    }
```

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

DisplayTree.java

```
77    final int x
78      = 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

```
79      final Dimension childrenDim = childrenSize();
80      final int childrenX = (int)childrenDim.getWidth();
81      final int childrenY = (int)childrenDim.getHeight();
```

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

DisplayTree.java

```
82      width=x>childrenX?x:childrenX;
83
84      height
85        =  childrenY == 0
86          ?fontMetrics.getHeight()
87          :VERTICAL_SPACE+childrenY;
88
89      dimensionCalculated = true;
90    }
```

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

DisplayTree.java

```
91    Dimension childrenSize(){
92      int x = 0;
93      int y = 0;
94
95      final Enumeration it=treeModell.children();
96      while (it.hasMoreElements()){
97        final DisplayTree t
98         = new DisplayTree((TreeNode)it.nextElement(),fontMetrics);
99
100       y =  y > t.getHeight()?y:t.getHeight();
101
102       x=x+t.getWidth();
103
104       if (it.hasMoreElements())  x=x+HORIZONTAL_SPACE;
105     }
106     return new Dimension(x,y);
107   }
```

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

DisplayTree.java

```
108   public int getHeight(){
109     if (!dimensionCalculated) calculateDimension();
110     return height;
111   }
112
```

```
113   public int getWidth(){
114     if (!dimensionCalculated) calculateDimension();
115     return width;
116   }
117
118   public Dimension getSize(){
119     if (!dimensionCalculated) calculateDimension();
120     return new Dimension(width,height);
121   }
122
123   public Dimension getMinimumSize(){
124     return getSize();
125   }
126
127   public Dimension getPreferredSize(){
128     return getSize();
129   }
```

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

```
130   public void paintComponent(Graphics g){paintAt(g,0,0);}
131
132   public void paintAt(Graphics g,int x,int y){
133     fontMetrics = g.getFontMetrics();
134     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

```
135     g.drawString
136      (marks
137      ,x+(getWidth()/2-fontMetrics.stringWidth(marks)/2)
138      ,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

```
139     final int startLineX = x+getWidth()/2;
140     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

```
141     final Enumeration it=treeModell.children();
142
143     final int childrenWidth = (int)childrenSize().getWidth();
144
145     //wieviel nach rechts zu rücken ist
146     int newX
147       = getWidth()>childrenWidth?(getWidth()-childrenWidth)/2:0;
148
149     //die y-Koordinate der Kinder
150     final int nextY = y+VERTICAL_SPACE;
151
152     while (it.hasMoreElements()){
153       DisplayTree t
154        = new DisplayTree((TreeNode)it.nextElement());
155
156       //x-Positionen für das nächste Kind
157       final int nextX = x+newX;
158
159       //zeichne das Kind
160       t.paintAt(g,nextX,nextY);
161
162       //zeichne Kante
163       g.drawLine(startLineX,startLineY
164                  ,nextX+t.getWidth()/2,nextY);
165
166       newX = newX+t.getWidth()+HORIZONTAL_SPACE;
167     }
168   }
169 }
```

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 4.6 zu bewundern.

DisplayTreeTest.java

```
1  import javax.swing.JFrame;
2
3  class DisplayTreeTest {
4
5    public static void main(String [] args){
6      JFrame frame = new JFrame("Baum Fenster");
7      frame
8        .getContentPane()
9        .add(new DisplayTree
10                 (ExampleSearchTree.getExampleTree()))
11     ;
12
13     frame.pack();
14     frame.setVisible(true);
```

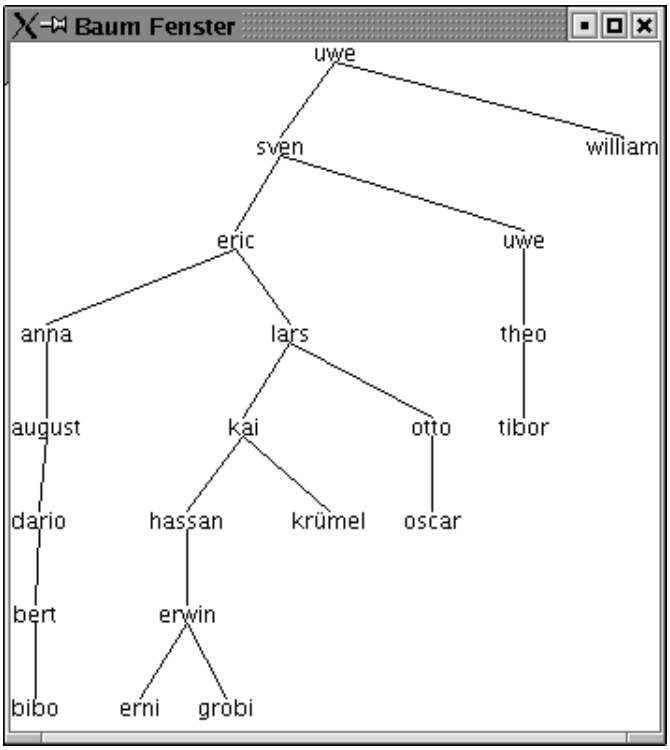


Abbildung 4.6: Baum graphisch Dargestellt mit DisplayTree.

```
15    }
16 }
```

4.3.7 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 eines Baumes in eine Datei zu speichern.

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

TreeToFile.java

```
1 import java.awt.Image;
2 import java.awt.image.BufferedImage;
```

```
3  import java.awt.image.RenderedImage;
4
5  import javax.imageio.ImageIO;
6
7  import java.awt.Graphics;
8
9  import javax.swing.tree.TreeNode;
10
11 import java.io.File;
12 import java.io.IOException;
```

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

TreeToFile.java

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

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

TreeToFile.java

```
16 try {
17       ImageIO.write(
18          createTreeImage
19             (ExampleSearchTree.getExampleTree())
20          ,"png"
21          , new File(args[0]+".png")
22       );
23       ImageIO.write(
24          createTreeImage
25             (ExampleSearchTree.getExampleTree())
26          ,"jpg"
27          , new File(args[0]+".jpg")
28       );
29     }catch (IOException e) {
30       System.out.println(e);
31     }
32   }
```

Die folgende Methode erzeugt das eigentliche Bild des Baumes:

TreeToFile.java

```
33 static public RenderedImage createTreeImage
34                                  (TreeNode tree) {
```

Wir erzeugen ein Bildobjekt. Allerdings wissen wir noch nicht, wie groß das Bild des Baumes sein wird. Daher nehmen wir ersteinmal eine beliebige Größe an:

TreeToFile.java
```
35      BufferedImage bufferedImage
36        = new BufferedImage
37            (10,10,BufferedImage.TYPE_INT_RGB);
```

Besorgen dessen `Graphics`-Umgebung:

TreeToFile.java
```
38      Graphics g = bufferedImage.createGraphics();
```

Jetzt erzeugen wir ein Objekt, daß die `paint`-Methode für unseren Baum hat:

TreeToFile.java
```
39      final DisplayTree treeDisplay
40        = new DisplayTree(tree,g.getFontMetrics());
```

Erst jetzt können wir die eigentliche Größe des Bildes ermessen:

TreeToFile.java
```
41      final int width = treeDisplay.getWidth();
42      final int height = treeDisplay.getHeight();
```

Mit dieser Größe können wir jetzt ein Bildobjekt mit den korrekten Dimensionmen erzeugen:

TreeToFile.java
```
43      bufferedImage
44       = new BufferedImage
45          (width,height,BufferedImage.TYPE_INT_RGB);
46      g = bufferedImage.createGraphics();
```

Und darin endlich unseren Baum zeichnen:

TreeToFile.java
```
47      // Draw graphics
48      g.setColor(java.awt.Color.WHITE);
49      g.fillRect(0,0,width,height);
50      g.setColor(java.awt.Color.BLACK);
51      treeDisplay.paintComponent(g);
```

Die Methode kann das fertig gezeichnete Bild als Ergebnis zurückgeben:

TreeToFile.java
```
52      return bufferedImage;
53    }
54  }
```

4.4 Ein Telespiel

In diesem Abschnitt wollen wir ein einfaches Telespiel programmieren. Telespiele waren Ende der Siebziger Anfang der Achtziger kleine Hardwareboxen, die an den Fernseher angeschlossen wurden. Die Software war fest im ROM eingebrannt und bestand aus einem einzigen Spiel, das gerne als Tennis bezeichnet wurde. Wir wollen es *Ping* nennen. Jugendlichen konnten damals nachmittage damit verbringen, dieses Spiel zu spielen. Das Spiel war übrigens relativ teuer, wenn man vergleicht, wie eingschränkt es war.[4]

Das Spiel Ping wird auf einem schwarzen Bildhintergrund gespielt. Eine weiße Kugel flitzt als der Ball vor diesem Hintergrund hin und her. Am oberen und unteren Bildschirmrand prallt der Ball ab, wobei der Ausfallswinkel gleich dem Einfallswinkel ist. Am rechten und linken Bildschirmrand haben die Spieler ihre Schläger. Diese bestehen aus weißen Rechtecken. Die Spieler können diese nach oben und unten bewegen. Der Ball prallt an diesen Schläger ab. Stößt er am rechten oder linken Bildchirmrand nicht auf einen Schläger, so geht er ins Aus und der gegenüberliegende Spieler bekommt einen Punkt.

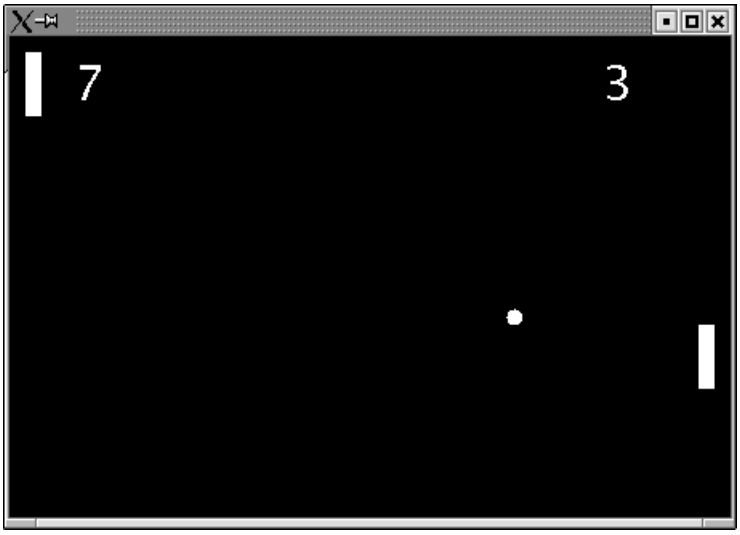


Abbildung 4.7: Bildschirmfoto des Spiels Ping.

4.4.1 Modellierung

Wir können folgende Objekte, die wir für das Spiel Ping benötigen, identifizieren.

- **Ball**: Der Ball ist ein eigenes Objekt, es hat einen Durchmesser, eine Position auf dem Spielfeld und eine Steigung und Richtung in der er sich bewegt.
- **Schläger**: Die Schläger haben eine Größe, eine Position auf dem Spielfeld und auch eine Richtung, in der sie sich bewegen (rauf oder runter).

[4] In meinem Elternhaus gab es kein Telespiel; aber ich habe es bei Freunden gespielt. Würde mich interessieren, ob noch welche von diesen komischen Kisten auf irgendwelchen Speichern existieren.

- **Spielfeld**: Das Spielfeld enthält zwei Schläger und einen Ball. Es hat natürlich auch eine eigene Größe.
- **Spiel**: Das Spiel enthält ein Spielfeld und den Spielstand.

Die informelle Beschreibung der Objekte oben, zeigt, daß die Schläger und der Ball viele gemeinsame Eigenschaften haben. Beides sind Objekte, die sich bewegten können. Lediglich die Art der Bewegung und die Gestalt der Objekte, wenn sie gezeichnet werden, unterscheiden sich. Daher modellieren wir eine gemeinsame Oberklasse für bewegliche Objekte.

Bewegte Objekte

Movable Wir werden für die Klasse `Movable` ein paar Klassen aus dem `awt`-Paket benötigen:

Movable.java
```
55 package de.tfhberlin.panitz.gui;
56
57 import java.awt.Graphics;
58 import java.awt.Color;
59 import javax.swing.JComponent;
```

Wir schreiben eine abstrakte Klasse. Nicht alle Eigenschaften können allgemein festgelegt werden; manche können erst durch konkrete Unterklasse beschrieben werden:

Movable.java
```
60 abstract public class Movable extends JComponent{
```

Ein bewegliches Objekt hat eine derzeitge Position in Form ihrer x- und y-Koordinate:

Movable.java
```
61   int x = 0;
62   int y = 0;
```

Weiterhin hat ein bewegtes Objekt eine Höhe und Weite. Wir betrachten solche Objekte in ihrer Größe immer als das sie minimal umschließende Rechteck:

Movable.java
```
63   int height;
64   int width;
```

Eine weitere wichtige Information ist die Größe des Fläche, auf der sich das Objekt bewegt:

Movable.java
```
65   public int boardHeight;
66   public int boardWidth;
```

Pro Bewegungsschritt eines beweglichen Objektes verändern sich der x- und y-Wert um einen bestimmten Betrag. Hierfür sehen wir zwei Felder vor. Standardmäßig soll der x-Wert zehn Pixel betragen und der y-Wert (damit die Steigung der Geraden) durch eine Zufallsfunktion ermittelt:

Movable.java
```
67   int yStep  = 2+((int)(Math.random()*20));
68   int xStep  = 10;
```

Wir sehen einen Konstruktor vor, der die Größe der Fläche, auf dem sich das Objekt bewegt, übergeben bekommt.

Movable.java
```
69   protected Movable(int w,int h){
70     boardWidth =w;
71     boardHeight =h;
72   }
```

Schließlich sehen wir ein Feld vor, in dem die Farbe des Objektes notiert ist:

Movable.java
```
73
74   Color foregroundColor = Color.WHITE;
```

Ein bewegliches Objekt hat eine Methode, die festlegt, wie das Objekt gezeichnet wird. Dieses hängt von der Form des konkreten Objektes ab. Da die Klasse `Movable` von `JComponent` ableitet, gibt es eine Methode `paintComponent`, die in Unterklasse überschrieben werden kann. Wir stellen in der Klasse `Movable` lediglich fest, daß die Farbe auf die Farbe des bewegten Objektes gesetzt wird.

Movable.java
```
75   public void paintComponent(Graphics g){
76     g.setColor(foregroundColor);
77   }
```

Die spannenste Eigenschaft eines beweglichen Objektes ist, wie es sich schrittweise bewegt. Auch dieses hängt von dem konkreten Objekt ab und ist in Unterklassen zu implementieren. Wir sehen hierzu eine Methode `move` vor. Diese wird jeweils die Werte in den Feldern `x` und `y` verändern.

Movable.java
```
78   abstract public void move();
```

Schließlich folgen noch eine Reihe von Methoden, die testen, ob das Objekt komplett über, unter, rechts oder links neben bestimmten Koordinaten liegt.

Movable.java
```
79   public boolean isLeftOf(int x)    {
80     return x>this.x+width;}
81   public boolean isRightOf(int x)   {return x<this.x;}
82   public boolean isAbove(int y)     {return y<this.y;}
83   public boolean isUnderneath(int y){
84     return y>this.y+height;
85   }
```

Eine weitere Interessante Eigenschaft ist, ob bewegliches Objekt ein anderes in irgendeinen Punkt berührt. Dieses ist z.B. wichtig, wenn wir testen wollen, ob der Ball einen Schläger berührt. Ein Objekt berührt ein anderes, wenn sie sich horizontal oder vertikal in irgendeiner Weise überschneiden.

Movable.java

```
86  public boolean touches(Movable other){
87    return (   touchesHorizontal(other)
88            && touchesVertical(other));
89  }
```

Wir definieren entsprechend, was es heißt horizontal oder vertikal ein anderes Objekt zu berühren:

Movable.java

```
90  private boolean touchesHorizontal(Movable other){
91    final int otherRight = other.x+other.width;
92    final int thisRight  = x+width;
93
94    return (x<=other.x    && other.x<=x+width)
95        || (x<=otherRight && otherRight<=x+width)
96        || (x>=other.x    && otherRight>=thisRight);
97
98  }
99
100 private boolean touchesVertical(Movable other){
101   final int otherUpper = other.y+other.height;
102   final int thisUpper  = y+height;
103
104   return   (y<=other.y    && other.y<=thisUpper)
105         || (y<=otherUpper && otherUpper<=thisUpper)
106         || (y>=other.y    && otherUpper>=thisUpper);
107 }
```

Damit ist die Klasse `Movable` vollständig geschrieben.

Movable.java

```
108 public void loadImages(java.applet.Applet a){}
109 }
```

Der Ball Um jetzt konkrete bewegliche Objekte zu implementieren, können wir jeweils Unterklassen der Klasse `Movable` schreiben. Es ist dabei zu implementieren, wie sich das Objekt bewegt und wie es zu zeichnen ist.

Ball.java

```
1  package de.tfhberlin.panitz.gui;
2
3  import java.awt.Graphics;
4
5  public class Ball extends Movable{
```

Als erstes implementieren wir die Methode `move`, die festlegt, wie sich der Ball bewegt. Er bewegt sich auf einer geraden Bahn. Wir nutzen die Werte `xStep` und `yStep` zum Bewegen des Balls. Die x-Koordinate wird um den entsprechenden Schritt erhöht (bzw. bei negativen Zahlen verringert). Dieses führt dazu, daß der Ball irgendwann aus dem Feld herausfliegt.

Ball.java

```
6  public void move(){
7   x = x+xStep;
```

Am oberen und unteren Rand soll der Ball abprallen. Das bedeutet, daß er nicht außerhalb des Feldes fliegen kann und daß er, wenn er auf die Feldbegrenzung trifft, seine y-Richtung ändert:

Ball.java

```
8   final int maxY = boardHeight-height;
9   y = Math.max(Math.min(y+yStep,maxY),0);
10
11  if (y==0 || y==maxY) {
12     yStep =-yStep;
13   }
14 }
```

Der Ball ist nach alter Fußballweisheit rund. Deshalb zeichnen wir ihn rund:

Ball.java

```
15 public void paintComponent(Graphics g){
16   super.paintComponent(g);
17   g.fillOval(x,y,width,height);
18 }
```

Schließlich sehen wir mehrere Konstruktoren vor, die Anfangswerte für einen Ball setzen. Das setzen bestimmter Werte wird dabei in eine Methode `init` ausgelagert. Diese kann dann dazu genutzt werden einen bestehenden Ball neu zu initialisieren.

Zunächst einen Konstruktor, der nur die Größe des Spielfelds übergeben bekommt und sonst die Standardwerte übernimmt

Ball.java

```
19
20 Ball(int boardW,int boardH){
21   super(boardW,boardH);
22   init();
23 }
```

Ein zweiter Konstruktor erlaubt die Position des Balles zu übergeben:

Ball.java

```
24 Ball(int xPos,int yPos, int boardW,int boardH){
25   this(boardW,boardH);
26   this.x=xPos;
27   this.y=yPos;
28 }
```

Ein dritter Konstruktor erlaubt es die horizontale Richtung des Balles mitzugeben. Der Parameter `xDir` soll dabei zwischen `-1` und `1` liegen.

Ball.java

```
29   Ball(int xDir,int xPos,int yPos
30        ,int boardW,int boardH){
31     this(xPos,yPos,boardW,boardH);
32     if (xDir<-1 || xDir >1)
33       throw new IllegalArgumentException
34                        ("wrong xDir value: "+xDir);
35     xStep=xDir*xStep;
36   }
```

Die benutzen Methoden `init` setzen dabei lediglich die Felder mit bestimmten Werten:

Ball.java

```
37   public void init(){
38     width=10;
39     height=width;
40     yStep  = 1+((int)(Math.random()*10));
41     xStep  = 5;
42   }
43
44   public void init(int x,int y){
45     init();
46     this.x=x;
47     this.y=y;
48   }
49
50   public void init(int dir,int x,int y){
51     init(x,y);
52     xStep=dir*xStep;
53   }
54 }
```

AnimatedCanvas Wir haben jetzt die Möglichkeit, bewegliche Bildschirmobjekte zu beschreiben. Jetzt brauchen wir eine Fläche, auf der sich diese Objekte bewegen. Hierui schreiben wir die Klasse `AnimatedCanvas`. Sie soll das Spielfeld beschreiben, auf dem sich bewegliche Objekte nach ihrer Methode `move` bewegen.

Wir importieren die notwendigen GUI-Klassen:

AnimatedCanvas.java

```
1  package de.tfhberlin.panitz.gui;
2
3  import java.awt.Dimension;
4  import javax.swing.JApplet;
5  import java.awt.Graphics;
6  import javax.swing.JFrame;
```

Wir erweitern die Klasse `JApplet`. Damit können wir das Spielfeld sowohl in einem Applet verwenden, als auch in einer eigenständigen Gui-Anwendung.

AnimatedCanvas.java
```
7 public class AnimatedCanvas extends JApplet
```

Desweiteren soll ein Spiel einen eigenen Steuerfaden implementieren:

AnimatedCanvas.java
```
8                              implements Runnable{
```

Ein Spielfeld braucht eine Größe:

AnimatedCanvas.java
```
9   final protected int height;
10  final protected int width;
```

Eine weitere Variable beschreibt, alle wieviel Millisekunden das Spielfeld neu aufgebaut wird:

AnimatedCanvas.java
```
11  final protected int sleepingTime = 30;
```

Die Größe wird als bevorzugte Bildgröße angegeben:

AnimatedCanvas.java
```
12  public Dimension getPreferredSize(){
13    return new Dimension(width,height);
14  }
```

Die folgenden Methoden sind für die Implementiering des Steuerfadens notwendig. Es sind Methoden aus der Schnittstelle `Runnable`.

AnimatedCanvas.java
```
15  public void start (){
16    Thread th = new Thread (this);
17    th.start ();
18  }
19
20  public void stop(){}
21  public void destroy(){}
```

Die Methode `run` des Steuerfadens stellt schließlich den Verlauf der Objekte auf dem Spielfeld dar. In einer endlos laufenden Schleife gibt es vier Phasen:

- die beweglichen Objekten werden weiterbewegt.
- das Spielfeld wird neu gezeichnet.
- bestimmte Checks bezüglich des Spielstandes werden durchgeführt. Hierzu gehört z.B., ob ein Objekt auf ein anderes Objekt getroffen ist (im Spiel `Ping`, ob ein Schläger den Ball berührt).

- eine kurze Pause, bis es zum nächsten Schritt geht.

AnimatedCanvas.java

```
22  public void run (){
23    while (true){
24      move();
25      repaint();
26
27      doChecks();
28
29      try {Thread.sleep(sleepingTime);}
30      catch (InterruptedException e) {}
31    }
32  }
```

Die Methoden, die die Checks und Bewegungsschritte enthalten, werden in den entsprechenden konkreten Spielen in Unterklassen überschrieben.

AnimatedCanvas.java

```
33  public void move(){}
34  public void doChecks(){}
35
36  public AnimatedCanvas
37           (final int width,final int height){
38    this.width= width;
39    this.height= height;
40  }
```

Ein Objekt vom Typ `AnimatedCanvas` ist ein Applet. Wenn wir es in einer eigenständigen Anwendung benutzen wollen, müssen wir es einen entsprechenden Fensterobjekt hinzufügen, welches dann geöffnet wird. Hierzu sehen wir eine Methode vor, die einen neuen Fensterrahmen öffnet, das Spiel hinzufügt und dann das Fenster sichtbar macht. Zusätzlich wird noch verhindert, daß der Benutzer das Fenster in der Größe ändern kann.

AnimatedCanvas.java

```
41  public void showInFrame(){
42    JFrame frame = new ClosingFrame();
43    frame.getContentPane().add(this);
44    frame.pack();
45    frame.setResizable(false);
46    frame.setVisible(true);
47    start();
48  }
49 }
```

AnimatedMovable Das erste einfache Spiel, das wir mit diesen Klassen schreiben können, läßt unseren Ball einmal über ein Spielfeld fliegen. Hierui können wir eine Klasse schreiben, die ein Spielfeld darstellt, auf dem genau ein bewegliches Objekt sich bewegt. Wir erweitern die Klasse `AnimatedCanvas`:

AnimatedMovable.java

```
1  package de.tfhberlin.panitz.gui;
2  import java.awt.Graphics;
3
4  public class AnimatedMovable extends AnimatedCanvas {
```

In einem Feld wird das bewegliche Objekt gespeichert:

AnimatedMovable.java

```
5
6    Movable mov;
```

Ein Konstruktor sieht vor, das bewegliche Objekt und die Größe des Spielfeldes festzulegen:

AnimatedMovable.java

```
7    public AnimatedMovable(Movable m,int x,int y){
8      super(x,y);
9      setBackground (java.awt.Color.BLACK);
10     mov=m;
11   }
```

Für das konkrete Verhalten wird die Methode `paint` so überschrieben, daß sie das bewegliche Objekt neu zeichnet:

AnimatedMovable.java

```
12   public void paint(Graphics g){
13     mov.paintComponent(g);
14   }
```

Die Methode `move` wird so überschrieben, daß sich das bewegliche Objekt einen Schritt weiterbewegt:

AnimatedMovable.java

```
15   public void move(){
16     mov.move();
17   }
18 }
```

TestBall Jetzt können wir einen Ball über das entsprechende Spielfeld fliegen lassen:

TestBall.java

```
1  package de.tfhberlin.panitz.gui;
2  class TestBall {
3    public static void main(String [] args){
4      final int width=450;
5      final int height=300;
6      Ball b = new Ball(width,height);
7
8      AnimatedMovable spiel
```

```
9       = new AnimatedMovable(b,width,height);
10    spiel.setBackground (java.awt.Color.BLACK);
11    spiel.showInFrame();
12  }
13 }
```

Das Flackern beenden Obiges Programm mit der einfachen weißen Kugel, die über den Bildschirm läuft, funktioniert noch nicht hindertprozentig wie erwartet. Der Hintergrund wird nicht bei jedem `repaint` vollständig neu gezeichnet. Dieses entspricht auch dem so dokumentierten Verhalten der entsprechende Javadokumentation:

> *You can assume that the background is not cleared.*

Wir könnten dieses Problem beheben, indem wir dafür sorgen, daß bei jedem Aufruf von `repaint` das Spielfeld wieder vollkommen neu mit der Hintergrundfarbe übermalt wird, bevor der Ball an die neue Position gezeichnet wird. Wenn wir aber jedesmal alles mit der Hintergrundfarbe überzeichnen und dann die Objekte neu zeichnen, dann wird ein unschönes Flackern des Bildes entstehen; für einen kurzen Moment ist ja kein Objekt auf dem Bild zu sehen, sondern alles nur mit der Hintergrundfarbe bemalt.

Die Standardtechnik, um dieses Flackern zu verhindern, ist das Bild zu puffern. Es wird ein internes unsichtbares Bildobjekt erzeugt. Auf diesem Bild wird nach und nach der eigentliche Bildaufbau gezeichnet, erst den Hintergrund und dann nach und nach die Objekte. Wenn das Bild komplett in diesen unsichtbaren Puffer gezeichnet wurde wird die so entstandene Graphik auf die eigentliche Bildausgabe der Komponente gezeichnet.

Wir wenden diese Technik der Pufferung an, indem wir die Klasse `AnimatedCanvas` erweitern .

BufferedAnimatedCanvas.java

```
1 package de.tfhberlin.panitz.gui;
```

Wir importieren die notwendigen Bild- und Grafikklassen und erweitern die Klasse `AnimatedCanvas`:

BufferedAnimatedCanvas.java

```
2
3 import java.awt.Graphics;
4 import java.awt.Canvas;
5 import java.awt.Image;
6
7 public class BufferedAnimatedCanvas
8                    extends AnimatedCanvas{
```

Es wird ein Konstruktor entsprechend der Oberklasse vorgesehen:

BufferedAnimatedCanvas.java

```
9   public BufferedAnimatedCanvas(final int x,final int y){
10    super(x,y);
11  }
```

Felder für den unsichtbaren Bildpuffer und sein Grafikobjekt werden angelegt:

BufferedAnimatedCanvas.java

```
12  private Image dbImage;
13  private Graphics dbg;
```

Die Methode `update`, die durch das `repaint` aufgerufen wird, legt zunächst falls noch nicht geschehen das Pufferobjekt an:

BufferedAnimatedCanvas.java

```
14  public void update (Graphics g){
15    if (dbImage == null){
16      dbImage = createImage (getWidth(), getHeight());
17      dbg = dbImage.getGraphics ();
18    }
```

Dann überzeichnet es den Bildpuffer mit der Hintergrundfarbe:

BufferedAnimatedCanvas.java

```
19    dbg.setColor (getBackground ());
20    dbg.fillRect (0, 0, getWidth(), getHeight());
```

Setzt die Vordergrundfarbe und benutzt die `paint`-Methode, um das Bild in den Bildpuffer zu malen:

BufferedAnimatedCanvas.java

```
21    dbg.setColor (getForeground());
22    paint(dbg);
```

Und zeichnet schließlich das gepufferte Bild auf das eigentlich Grafikobjekt für die Komponente:

BufferedAnimatedCanvas.java

```
23    g.drawImage (dbImage, 0, 0, this);
24  }
25  }
```

BufferedAnimatedMovable Es läßt sich jetzt analog wie oben schon, ein Spielfeld, auf dem sich ein bewegliches Objekt befindet, definieren. Diesmal aber unter Benutzung des gepufferten Bildes:

BufferedAnimatedMovable.java

```
1  package de.tfhberlin.panitz.gui;
2  import java.awt.Graphics;
3
4  public class BufferedAnimatedMovable
5                       extends BufferedAnimatedCanvas {
6    Movable mov;
7
8    public BufferedAnimatedMovable(Movable m,int x,int y){
```

```
9     super(x,y);
10    setBackground (java.awt.Color.BLACK);
11    mov=m;
12  }
13
14  public void paint(Graphics g){
15    mov.paintComponent(g);
16  }
17
18  public void move(){
19    mov.move();
20  }
21 }
```

Und auch dieses läßt sich mit unserem Ball testen; wobei wir diesmal einen besonderen Ball kreieren, der nicht am linken und rechten Spielfeldrand das Spielfeld verläßt, sondern auch an den Seiten abprallt:

TestBufferedBall.java

```
1  package de.tfhberlin.panitz.gui;
2
3  class TestBufferedBall {
4    public static void main(String [] args){
5      final int width=450;
6      final int height=300;
7
8      new BufferedAnimatedMovable
9              (new Ball(width,height){
10                public void move(){
11                  super.move();
12                  if (x<=0 || x>=boardWidth-width)
13                    xStep =-xStep;
14                }
15              },width,height)
16       .showInFrame();
17   }
18 }
```

Der Schläger Das zweite bewegliche Objekt für das Spiel `Ping` sind die Schläger. Hierzu erweitern wir wieder die Klasse `Mobable`:

Paddle.java

```
1  package de.tfhberlin.panitz.gui;
2  import java.awt.Graphics;
3  import java.awt.Component;
4  import java.awt.Color;
5
6  class Paddle extends Movable{
```

Ein Konstruktor setzt die Größe eines Schlägers, die Größe des Spielfelds und seine Bewegung in x- und y-Richtung:

Paddle.java

```
7   Paddle(int where,int boardW,int boardH){
8     super(boardW,boardH);
9     height=40;
10    width =10;
11    yStep = -10;
12    xStep = 0;
13    this.x=where;
14  }
```

Der Schläger bewegt sich entlang der y-Achse. Er fällt nicht aus dem Spielfeld:

Paddle.java

```
15  public void move(){
16    y = y+yStep;
17    if (y+height>boardHeight){
18      y=boardHeight-height;
19    }
20    if (y<0){
21      y=0;
22    }
23  }
```

Zum Zeichnen des Schlägers wird ein einfaches Rechteck gemalt:

Paddle.java

```
24  public void paintComponent( Graphics g){
25    super.paintComponent(g);
26    g.fillRect(x,y,width,height);
27  }
28}
```

Da ein `Paddle` ein bewegliches Objekt ist, können wir ihn genauso mit der Klasse `BufferedAnimatedMovable` testen:

TestPaddle.java

```
1 package de.tfhberlin.panitz.gui;
2 class TestPaddle {
3   public static void main(String [] args){
4     final int width=450;
5     final int height=300;
6
7     Paddle p = new Paddle(80,width,height);
8     p.y=70;
9
10    new BufferedAnimatedMovable(p,width,height)
11     .showInFrame();
12  }
13}
```

4.4.2 Das Spiel

Die wichtigsten Teile des Spiels `Ping` haben wir jetzt beieinander: Objekte für die Schläger und den Ball, sowie eine allgemeine Klasse für Felder, auf denen sich graphische Objekte bewegen. Was noch fehlt ist die Benutzerinteraktion über die Tastatur und eine Klasse, die das Spiel definiert.

Tastatureingabe

Per Tastatureingabe soll sich die Bewegung der Schläger steuern, sowie ein neuer Ball aufschlagen lassen. Wir schreiben eine Klasse des Typs `java.awt.event.KeyListener`.

PingKeyListener

PingKeyListener.java

```
1  package de.tfhberlin.panitz.gui;
2
3  import java.awt.event.KeyListener;
4  import java.awt.event.KeyEvent;
5
6  public class PingKeyListener implements KeyListener{
```

Konstanten legen fest, mit welcher Taste, was gesteuert werden soll. Wir sehen je eine Taste für linken und rechten Schläger vor und die Leertaste[5], um einen neuen Ball aufzuschlagen:

PingKeyListener.java

```
7    final char rightPlayerKey = 'l';
8    final char leftPlayerKey  = 'a';
9    final char nextBallKey    = '\u0020';
```

Eine weitere Konstante gibt an, um wieviel Pixel sich ein Schläger in einem Schritt bewegen soll. Sozusagen das Tempo des Schlägers:

PingKeyListener.java

```
10   //final int  step = 10;
```

Da über diese Klasse zwei Schläger gesteuert werden, werden für diese zwei Felder vorgesehen:

PingKeyListener.java

```
11   final Movable leftPaddle;
12   final Movable rightPaddle;
```

Ein weiteres Feld vermerkt, ob ein neuer Ball ins Spiel gekommen ist:

PingKeyListener.java

```
13   boolean newBall = false;
```

[5]Daß hier im Skript nicht einfach die Zeichenkonstante für das Leerzeichen steht, liegt an einem Bug des XQuery-Prozessor, der benutzt wird, um die Klassen aus dem XML-Quelltext zu extrahieren.

Der Konstruktor bekommt die zu steuernden Schläger übergeben:

PingKeyListener.java

```
14  public PingKeyListener(Movable l, Movable r){
15    leftPaddle=l;
16    rightPaddle=r;
17  }
```

In den Methoden `keyPressed`, `keyReleased` und `keyTyped` kann spezifiziert werden, was passieren soll, wenn eine Taste heruntergedrückt, losgelassen oder einfach nur einmal kurz gedrückt wird.

Das Drücken der tasten für linken und rechten Schläger soll bewirken, daß der Schläger nach unten wandert:

PingKeyListener.java

```
18  public void keyPressed(KeyEvent e) {
19    if (e.getKeyChar() == leftPlayerKey){
20      leftPaddle.yStep = Math.abs(leftPaddle.yStep);
21    }
22
23    if (e.getKeyChar() == rightPlayerKey){
24      rightPaddle.yStep = Math.abs(rightPaddle.yStep);
25    }
26  }
```

Beim Loslassen der entsprechen Tasten soll der Schläger wieder die Richtung nach oben einschlagen:

PingKeyListener.java

```
27  public void keyReleased(KeyEvent e){
28    if (e.getKeyChar() == leftPlayerKey){
29      leftPaddle.yStep = -Math.abs(leftPaddle.yStep);
30    }
31    if (e.getKeyChar() == rightPlayerKey){
32       rightPaddle.yStep = -Math.abs(rightPaddle.yStep);
33    }
34  }
```

Wird die Leertaste gedrückt, so wird vermerkt, daß ein neuer Ball ins Spiel kommen soll:

PingKeyListener.java

```
35  public void keyTyped(KeyEvent e){
36    if (e.getKeyChar() == nextBallKey){
37      newBall = true;
38    }
39  }
40 }
```

Das Spielfeld

Jetzt lassen sich die Komponente alle zum eigentlichen Spiel zusammenfügen. Das Spielfeld leitet dabei von der Klasse `BufferedAnimatedCanvas` ab:

Ping.java
```
1  package de.tfhberlin.panitz.gui;
2
3  import java.awt.Color;
4  import java.awt.Graphics;
5
6  public class Ping extends BufferedAnimatedCanvas{
```

Zwei Konstanten geben an, wieweit ein Schläger vom Rand entfernt ist und mit wieviel Punkten ein Spieler gewonnen hat.

Ping.java
```
7    final int highestPoints = 10;
8    final int paddleOffset = 10;
```

Es gibt im Spiel `Ping` drei bewegliche Objekte auf dem Spielfeld: die Schläger und der Ball.

Ping.java
```
9    Ball b ;
10   Movable leftPaddle ;
11   Movable rightPaddle;
```

Für die Steuerung über die Tastatur wird ein `KeyListener`benötigt.

Ping.java
```
12   PingKeyListener keyListener;
```

Zwei Felder geben an, wieviel Punkte die Spieler gerade haben.

Ping.java
```
13   int pointsPlayer1 = 0;
14   int pointsPlayer2 = 0;
```

Wir sehen zwei Konstruktoren vor. Einer besetzt alle Fälle mit Standardwerten, einer bekommt entsprechende Werte übergeben:

Ping.java
```
15   public Ping
16     (Movable l,Movable r,Ball ball,int width,int height){
17     super(width,height);
18     leftPaddle = l;
19     rightPaddle = r;
20     b=ball;
21     initKeyListener();
22   }
```

```
23
24  public Ping(){
25    super(450,300);
26    b= new Ball(-10,-10,width,height);
27    leftPaddle = new Paddle(paddleOffset,width,height);
28    rightPaddle
29      =new Paddle
30          (width-paddleOffset-leftPaddle.width
31          ,width,height);
32    initKeyListener();
33  }
```

Beide Konstruktoren setzen den `KeyListener` mit folgender Methode:

Ping.java

```
34  private void initKeyListener(){
35    keyListener
36      = new PingKeyListener(leftPaddle,rightPaddle);
37    addKeyListener(keyListener);
38    setBackground (Color.BLACK);
39  }
```

Zum Zeichnen des Spielfelds, werden Ball und Schläger gezeichnet, sowie der Spielstand in weißer Schrift:

Ping.java

```
40  public void paint(Graphics g){
41    b.paintComponent(g);
42    rightPaddle.paintComponent(g);
43    leftPaddle.paintComponent(g);
44
45    g.setFont(g.getFont().deriveFont((float) 30));
46    g.setColor(Color.WHITE);
47    g.drawString(""+pointsPlayer1,40,40);
48    g.drawString(""+pointsPlayer2,getWidth()-80,40);
49  }
```

Beim Start des Spiels, wird ein Ball auf höhe des linken Schlägers kreiert. Anschließend wird die `run`-Methode der Oberklasse aufgerufen. In dieser befindet sich die nichtendende Schleife.

Ping.java

```
50  public void run(){
51    repaint();
52    getNewBall(leftPaddle);
53    super.run();
54  }
```

Die eigentliche Spiellogik befindet sich in der Methode `doChecks`, die bei jeden Schleifenlauf einmal durchlaufen wird.

Ping.java
```
55  public void doChecks (){
```

Berührt ein Schläger den Ball, so ändert der Ball seine Richtung.

Ping.java
```
56    if (leftPaddle.touches(b)||rightPaddle.touches(b)){
57      b.xStep = -b.xStep;
58    }
```

Ist der Ball links vom linken Schläger, so hat der rechte Spieler (Spieler 2) einen Punkt gemacht. Ein neuer Ball ist aufzuschlagen.

Ping.java
```
59    if (b.isLeftOf(paddleOffset)){
60      pointsPlayer2++;
61      repaint();
62      getNewBall(rightPaddle);
63    }
```

Entsprechend, wenn der Ball rechts vom rechten Schläger ist.

Ping.java
```
64    if (b.isRightOf(getWidth()-paddleOffset)){
65      pointsPlayer1++;
66      repaint();
67      getNewBall(leftPaddle);
68    }
69
70
```

Wenn ein Spieler gewonnen hat, kann ein neues Spiel beginnen. Der Spielstand wird zurückgesetzt.

Ping.java
```
71   if (   pointsPlayer2==highestPoints
72        || pointsPlayer1==highestPoints){
73      pointsPlayer2=0;
74      pointsPlayer1=0;
75      getNewBall(leftPaddle);
76    }
77  }
```

Wenn das Spiel sich einen Schritt weiterbewegt, dann ist jedes bewegliche Objekt einen Schritt weiterzusetzen.

Ping.java
```
78  public void move(){
79    b.move();
80    rightPaddle.move();
81    leftPaddle.move();
82  }
```

Wenn ein neuer Ball geholt werden soll, so wird auf eine Eingabe in der Tastatur gewartet und dann der Ball neu initialisiert.

Ping.java

```
83   void getNewBall(Movable p){
84     while (!keyListener.newBall){
85       try {Thread.sleep(sleepingTime);}
86       catch (InterruptedException e) {}
87     }
88
89     final int dir = p.x>100?-1:1;
90     b.init(dir,p.x+50*dir,p.y);
91     keyListener.newBall = false;
92   }
```

Mit folgender Hauptmethode kann das Spiel gestartet werden.

Ping.java

```
93   public static void main(String [] args){
94     new Ping().showInFrame();
95   }
96 }
```

4.4.3 Bilder und Geräusche

SmileBall.java

```
1  package de.tfhberlin.panitz.gui;
2  import java.awt.Image;
3  import java.awt.Graphics;
4  import java.awt.Toolkit;
5  import java.awt.Component;
6  import java.awt.image.ImageObserver;
7
8  public class SmileBall extends Ball {
9    final Toolkit toolkit = Toolkit.getDefaultToolkit();
10   Image smile;
11
12   public void paintComponent(Graphics g){
13     g.drawImage(smile, x, y, width, height, this);
14   }
15
16   SmileBall
17     (int xPos,int yPos,int boardWidth,int boardHeight){
18     super(xPos,yPos,boardWidth,boardHeight);
19     try {
20       smile = toolkit.getImage("images/smile.gif");
21     }catch(Exception _){}
22
23     init();
24   }
```

```
25
26    public void init(){
27      super.init();
28      width=20;
29      height=width;
30      yStep  = 1+((int)(Math.random()*40));
31      xStep  = 20;
32    }
33
34    public void loadImages(java.applet.Applet a){
35      try {
36        smile
37         = a.getImage(a.getCodeBase(),"images/smile.gif");
38      }catch(Exception _){}
39    }
40  }
```

ImagePaddle.java

```
1   package de.tfhberlin.panitz.gui;
2   import java.awt.Image;
3   import java.awt.Graphics;
4   import java.awt.Color;
5   import java.awt.image.ImageObserver;
6   import java.awt.Toolkit;
7   import java.applet.Applet;
8
9   public class ImagePaddle extends Paddle {
10    final Toolkit toolkit = Toolkit.getDefaultToolkit();
11    Image img;
12
13    public void paintComponent(Graphics g){
14      g.drawImage(img, x, y ,width, height, this);
15    }
16
17    ImagePaddle(int where,int boardW,int boardH){
18      super(where,boardW,boardH);
19      width=11;
20      height=100;
21      yStep=-30;
22      try {
23        img = toolkit.getImage("images/schlaeger.gif");
24      }catch(Exception _){}
25    }
26
27    public void loadImages(Applet a){
28      try {
29        img = a.getImage
30                (a.getCodeBase(),"images/schlaeger.gif");
31      }catch(Exception _){}
```

```
32    }
33
34  }
```

PingSmile.java

```
1   package de.tfhberlin.panitz.gui;
2   import java.applet.AudioClip;
3   import java.net.URL;
4   import java.applet.Applet;
5
6   public class PingSmile extends Ping {
7     static int width = 900;
8     static int height= 700;
9
10    AudioClip bounce;
11
12    public PingSmile(){
13      super(new ImagePaddle(10,width,height)
14           ,new ImagePaddle(width-10-11,width,height)
15           ,new SmileBall(-20,-20,width,height)
16           ,width,height);
17      setBackground(java.awt.Color.GREEN);
18      try{
19        bounce=Applet.newAudioClip
20                    (new URL("file://./sounds/bounce.au"));
21      }catch (Exception e){System.out.println(e);}
22    }
23
24    public  void init(){
25      leftPaddle.loadImages(this);
26      rightPaddle.loadImages(this);
27      b.loadImages(this);
28      try{
29        bounce
30         = getAudioClip(getCodeBase(),"sounds/bounce.au");
31      }catch (Exception e){System.out.println(e);}
32    }
33
34    public void doChecks(){
35      if (b.touches(leftPaddle) || b.touches(rightPaddle))
36        bounce.play();
37      super.doChecks();
38
39    }
40
41    public static void main(String [] args){
42      new PingSmile().showInFrame();
43    }
44  }
```

4.4.4 Beliebige zweidimensionale Spiele

Schnee auf Photo

Snow.java

```
1  package de.tfhberlin.panitz.gui;
2  import java.awt.Color;
3  import java.awt.Graphics;
4  import java.awt.Component;
5
6  public class Snow extends Movable{
7    int yStep=1;
8
9    public void move(){
10     final int maxY = boardHeight-height;
11     y = y+yStep;
12   }
13
14   public void paintComponent(Graphics g){
15     super.paintComponent(g);
16     g.fillOval(x,y,width,height);
17   }
18
19   Snow(int boardW,int boardH){
20     super(boardW,boardH);
21     init();
22   }
23
24   public void init(){
25     width=1+(int)(Math.random()*5);
26     height=width;
27     y  = -height;
28     x  = (int)(Math.random()*boardWidth);
29     yStep = 1+(int)(Math.random()*4);
30   }
31
32   public Snow getNew(){
33      return new Snow(boardWidth,boardHeight);
34   }
35 }
```

Snowing.java

```
1  package de.tfhberlin.panitz.gui;
2  import java.awt.Color;
3  import java.awt.Graphics;
4  import java.util.List;
5  import java.util.ArrayList;
6  import java.util.Iterator;
7  import java.awt.Image;
8  import java.awt.Toolkit;
```

```
9
10 public class Snowing extends BufferedAnimatedCanvas{
11   List/*Movable*/ mvs = new ArrayList();
12   Snow s;
13
14   Image img;
15
16   public Snowing(Snow s){
17     this();
18     this.s=s;
19   }
20
21   public Snowing(){
22     super(745,412);
23     setBackground (Color.BLACK);
24     s=new Snow(754,412);
25     try{
26       img = Toolkit.getDefaultToolkit()
27                    .getImage("images/oldenburg.jpg");
28     }catch (Exception _){}
29   }
30
31   public  void init(){
32     try {
33       img = getImage(getCodeBase()
34                     ,"images/oldenburg.jpg");
35     }catch(Exception _){}
36   }
37
38
39   public void paint(Graphics g){
40     g.drawImage(img,0,0,this);
41     final Iterator it=mvs.iterator();
42     while (it.hasNext()){
43       ((Movable)it.next()).paintComponent(g);
44     }
45   }
46
47   public void run(){
48     repaint();
49     super.run();
50   }
51
52   public void doChecks (){
53     for(int i=1;i<=18;i=1+i){
54       mvs.add(s.getNew());
55     }
56
57     final Iterator it=mvs.iterator();
58     while (it.hasNext()){
```

```
59        Movable m = ((Movable)it.next());
60        if (m.y>height) it.remove();
61      }
62    }
63
64    public void move(){
65      final Iterator it=mvs.iterator();
66      while (it.hasNext()){
67        ((Movable)it.next()).move();
68      }
69    }
70
71    public static void main(String [] args){
72      new Snowing().showInFrame();
73    }
74  }
```

Ampelmännchen

Ampelmaennchen.java

```
1   package de.tfhberlin.panitz.gui;
2   import java.awt.Image;
3   import java.awt.Graphics;
4   import java.awt.Toolkit;
5
6   public class Ampelmaennchen extends Movable {
7     final Toolkit toolkit = Toolkit.getDefaultToolkit();
8     static final int standing=8;
9
10    final Image [] ampel = new Image[standing+1];
11
12
13    static final int left=4;
14    static final int right=0;
15
16    int jumpHeight = 120;
17
18    public Ampelmaennchen(int boardW,int boardH){
19      super(boardW,boardH);
20      width=110;
21      height=width;
22      x=(boardWidth-width)/2;
23      y=boardHeight-height;
24      xStep=0;
25      try
26        {ampel[0] = toolkit.getImage("images/ampelr3o.gif")
27        ;ampel[1] = toolkit.getImage("images/ampelr2o.gif")
28        ;ampel[2] = toolkit.getImage("images/ampelr1o.gif")
29        ;ampel[3] = toolkit.getImage("images/ampelr2o.gif")
```

```
30      ;ampel[4] = toolkit.getImage("images/ampell3o.gif")
31      ;ampel[5] = toolkit.getImage("images/ampell2o.gif")
32      ;ampel[6] = toolkit.getImage("images/ampell1o.gif")
33      ;ampel[7] = toolkit.getImage("images/ampell2o.gif")
34      ;ampel[8] =toolkit.getImage("images/ampelroto.gif")
35      ;}
36    catch(Exception _){}
37  }
38
39  public void loadImages(java.applet.Applet a){
40    try
41      {final java.net.URL b = a.getCodeBase();
42      ;ampel[0] = a.getImage(b,"images/ampelr3o.gif")
43      ;ampel[1] = a.getImage(b,"images/ampelr2o.gif")
44      ;ampel[2] = a.getImage(b,"images/ampelr1o.gif")
45      ;ampel[3] = a.getImage(b,"images/ampelr2o.gif")
46      ;ampel[4] = a.getImage(b,"images/ampell3o.gif")
47      ;ampel[5] = a.getImage(b,"images/ampell2o.gif")
48      ;ampel[6] = a.getImage(b,"images/ampell1o.gif")
49      ;ampel[7] = a.getImage(b,"images/ampell2o.gif")
50      ;ampel[8] = a.getImage(b,"images/ampelroto.gif")
51      ;}
52    catch(Exception _){}
53  }
54
55
56  int leftRight = standing;
57  int current   = 0;
58  int solong    = 1;
59  boolean jump  = false;
60
61  public void stand(){
62    leftRight = standing;
63    xStep=0;
64  }
65
66  public void jump(){
67    if (yStep<=0) yStep=-3;
68  }
69
70  public void move(){
71    x=x+xStep;
72    y=y+yStep;
73
74    final int rightPoint = boardWidth-width;
75    if (x>rightPoint) {x = rightPoint;stand();}
76    if (x<0)          {x=0;          ;stand();}
77
78    final int maximalY = boardHeight-height;
79    if (y<maximalY-jumpHeight) yStep = 3;
```

```
80      if (y>maximalY) yStep = 0;
81
82      solong = (solong + 1)%10;
83
84      if (solong==0 && leftRight!=standing)
85         current = (current + 1)%4;
86    }
87
88    public void paintComponent(Graphics g){
89      final int w
90       = leftRight==standing?standing:current+leftRight;
91      g.drawImage(ampel[w],x,y,width,height,this);
92    }
93  }
```

AmpelKeyListener.java

```
1   package de.tfhberlin.panitz.gui;
2   import java.awt.event.KeyListener;
3   import java.awt.event.KeyEvent;
4   public class AmpelKeyListener implements KeyListener {
5     private Ampelmaennchen ampel;
6     public AmpelKeyListener(Ampelmaennchen a){ampel=a;}
7
8     public void keyPressed(KeyEvent e) {
9       if (e.getKeyChar() == 'l'){
10        ampel.xStep = 3;
11        ampel.leftRight = Ampelmaennchen.right;
12      }
13      if (e.getKeyChar() == 'k'){
14        ampel.xStep = -3;
15        ampel.leftRight = Ampelmaennchen.left;
16      }
17    }
18
19    public void keyReleased(KeyEvent e){
20      if (e.getKeyChar() != " l".charAt(0))
21        ampel.stand();
22    }
23
24    public void keyTyped(KeyEvent e){
25      if (e.getKeyChar() == " l".charAt(0)){
26        ampel.jump();
27      }
28    }
29  }
```

TestAmpel.java

```
1   package de.tfhberlin.panitz.gui;
2   import java.awt.event.KeyListener;
```

```
3  import java.awt.event.KeyEvent;
4
5  class TestAmpel {
6    public static void main(String [] args){
7      final int width=850;
8      final int height=300;
9
10     final Ampelmaennchen ampel
11      = new Ampelmaennchen(width,height);
12
13     BufferedAnimatedMovable canvas
14      = new BufferedAnimatedMovable(ampel,width,height);
15     canvas.setBackground(java.awt.Color.YELLOW);
16     canvas.addKeyListener(new AmpelKeyListener(ampel));
17     canvas.showInFrame();
18   }
19 }
```

Auto.java

```
1  package de.tfhberlin.panitz.gui;
2
3  import java.awt.Graphics;
4  import java.awt.Color;
5
6  public class Auto extends Movable{
7    public void move(){
8       x=x+xStep;
9    }
10
11   public void paintComponent(Graphics g){
12     super.paintComponent(g);
13 //    g.fillRect(x,y,width,height);
14     g.setColor(Color.BLACK);
15     g.fillOval(x+5,y+height-20,20,20);
16     g.fillOval(x+width-20-5,y+height-20,20,20);
17     g.setColor(Color.BLUE);
18     g.fillRect(x,y,width,height-10);
19   }
20
21   Auto(int boardW,int boardH){
22     super(boardW,boardH);
23     init();
24   }
25
26   public void init(){
27     width=80+(int)(Math.random()*120);
28     height=60;
29     y  = boardHeight-height;
30     x = boardWidth;
```

```
31      yStep = 0;
32      xStep = -5;
33    }
34  }
```

AmpelSpiel.java

```
1   package de.tfhberlin.panitz.gui;
2   import java.util.List;
3   import java.util.ArrayList;
4   import java.util.Iterator;
5   import java.awt.Color;
6   import java.awt.Graphics;
7
8   public class AmpelSpiel extends BufferedAnimatedCanvas{
9     int runde = 0;
10    final int abstand = 150;
11
12    int minusPunkte = 0;
13
14    Ampelmaennchen ampel;
15
16    final List autos = new ArrayList();
17
18    public void doChecks(){
19      runde = (runde+1)%abstand;
20      if (runde==0)autos.add(new Auto(width,height));
21
22      final Iterator it=autos.iterator();
23      while (it.hasNext()){
24        Movable m = ((Movable)it.next());
25        if (m.touches(ampel)) {
26            minusPunkte=minusPunkte-1;
27            break;
28        }
29        if (m.x< -m.width) it.remove();
30      }
31
32
33    }
34
35    public AmpelSpiel(){
36      super(850,300);
37      setBackground (Color.YELLOW);
38      ampel = new  Ampelmaennchen(width,height);
39      addKeyListener(new AmpelKeyListener(ampel));
40    }
41
42    public  void init(){
43      ampel.loadImages(this);
```

```
44    }
45
46
47    public void paint(Graphics g){
48      ampel.paintComponent(g);
49      final Iterator it=autos.iterator();
50      while (it.hasNext()){
51        ((Movable)it.next()).paintComponent(g);
52      }
53      g.setFont(g.getFont().deriveFont((float) 30));
54      g.setColor(Color.WHITE);
55      g.drawString(""+minusPunkte,40,40);
56
57    }
58
59    public void move(){
60      final Iterator it=autos.iterator();
61      while (it.hasNext()){
62        ((Movable)it.next()).move();
63      }
64      ampel.move();
65    }
66
67    public static void main(String [] args){
68      new AmpelSpiel().showInFrame();
69    }
70  }
```

Kapitel 5

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.

5.1 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].

5.1.1 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 18 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.

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

  start
  →*ABstart*
  →*ABABstart*
  →*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$b$$Tc|\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:

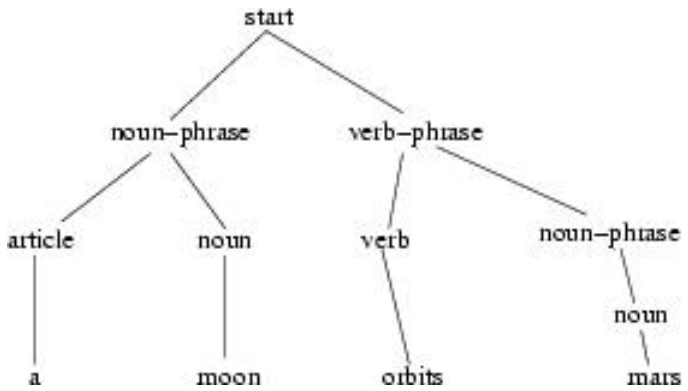


Abbildung 5.1: Ableitungsbaum für a moon orbits mars.

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

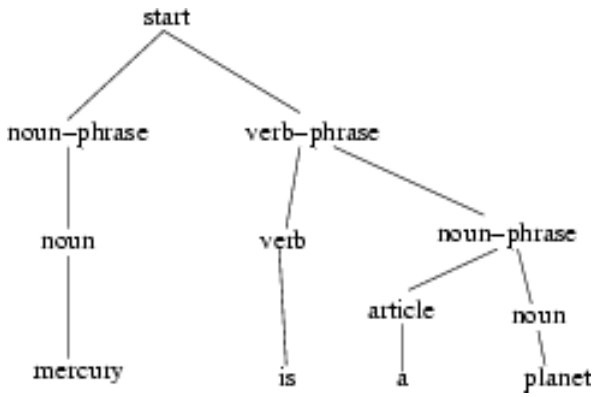


Abbildung 5.2: Ableitungsbaum für mercury is a planet.

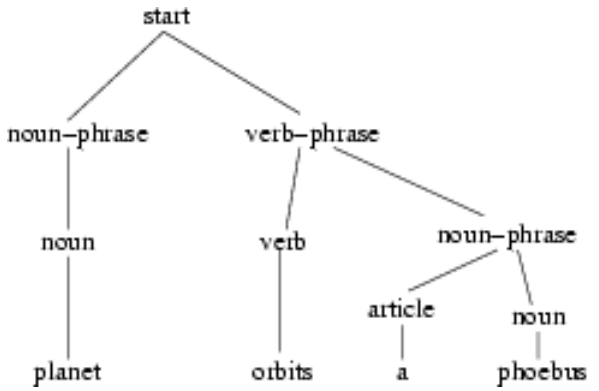


Abbildung 5.3: Ableitungsbaum für planet orbits a phoebus.

5.1.2 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.

5.2 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.

5.2.1 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.

5.2.2 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.

Test auf syntaktische Korrektheit

Wir wollen für eine Folge von Token testen, ob sie mit den Regeln einer bestimmten Grammatik abgelitten werden kann. Hierzu definieren wir zunächst eine Schnittstelle, die Token ausdrückt.

Token.java

```
1 public interface Token {}
```

Verschiedene Token werden wir einfach anhand ihrer Klasse unterscheiden.

Als nächstes wollen wir in einer Schnittstelle definieren, was wir allgemein von einem Parser für Funktionalität erwarten.

- Die wichtigste Methode soll die eigentliche Parsemethode sein. Sie soll zunächst nur entscheiden, ob ein der Satz ableitbar ist oder nicht. Ihr Ergebnis ist entsprechend ein Wert vom Typ `boolean`.
- Der einfachste Parser testet, ob ein bestimmtes Terminalsymbol vorliegt. Hierzu sehen wir eine Methode vor, die testet, ob ein bestimmtes Token erkannt werden kann.
- Schließlich wollen wir eine Methode die das leere Wort parst vorsehen.

Zusammengefasst erhalten wir folgende Schnittstelle für Parser.

Parser.java

```
1 public interface Parser{
2   public boolean parse();
3   public boolean parseToken(Class tokenClass);
4   public boolean epsilon();
5 }
```

Bevor wir ein spezielles Beispiel für einen Parser implementieren, können wir die Methoden, die wir allgemein für jeden Parser benötigen, in einer Klassen schreiben, von der wir alle weiteren Parser ableiten werden. Ein abstrakter Parser enthält:

- den Satz, den er parsen soll. Wir werden diesen als eine Reihung von Token speichern und sehen hierzu ein Feld vor.
- eine Zahl `current`, die angibt, bis zu welchem Token wir bereits unseren Satz konsumiert haben.

- die Methode `epsilon()`, die ein leeres Wort parst. Das leere Wort läßt sich immer parsen, daher gibt die Methode konstant `true` zurückgibt und den `current` Wert nicht verändert (da ja nichts konsumiert wird).
- eine Methode `boolean parseToken(Class cl)`, die testet, ob das aktuelle Token von der Klasse `cl` ist.

Diese Funktionalität läßt sich in folgender kleinen Klasse bereitstellen.

AbstractParser.java

```
1  class AbstractParser {
2
3    Token[] token ;
4    int current=0;
5
6    public boolean epsilon(){
7      return true;
8    }
9
10   public boolean parseToken(Class cl){
11     if (current<token.length){
12       Token next = token[current];
13       if (cl.isAssignableFrom(next.getClass())){
14         current=current+1;
15         return true;
16       }
17     }
18     return false;
19   }
20 }
```

Die Methode `parseToken` soll testen ob das nächste Token von einer bestimmten Art ist. Verschiedene Token haben in unserer Umsetzung unterschiedliche Klassen. Das Argument `cl` der Methode gibt die Klasse an, mit der das nächste Token zu vergleichen ist. Sofern es noch Token gibt `current<token.length` wird das aktuelle Token betrachtet. Wenn dieses von derselben Klasse oder einer Klasse der Klasse `cl` ist, so ergibt die Methode `true`. Zusätzlich ist die Position `current` auf den Index des nächsten Tokens zu setzen.

Beispiel:
Ein sehr einfaches Beispiel sei die Grammatik, die nur ein Token kennt, das eine Zahl darstellt. Hierzu implementieren wir die Schnittstelle `Token`.

Zahl.java

```
1  class Zahl implements Token{
2    int i;
3    Zahl(int i){this.i=i;}
4    public String toString(){return i+"";}
5  }
```

Der einfache Grammatik soll erzeuge die Sprache mit nur einen Satz:

start ::= Zahl

Für diese Grammatik läßt sich mit den bisherigen Mitteln schnell ein Parser schreiben.

OneNumber.java

```
1  public class OneNumber
2   extends AbstractParser implements Parser{
3    public OneNumber(Token[] token){this.token=token;}
4
5    public boolean parse(){return start();}
6
7    boolean start(){return parseToken(Zahl.class);}
8
9    public static void main(String [] _){
10     Token [] correct = {new Zahl(42)};
11     Parser parser = new OneNumber(correct);
12     System.out.println(parser.parse());
13   }
14
15 }
```

Wir sind in der Lage Terminalsymbole zu erkennen. Jetzt gilt es die Regeln einer Grammatik in ein Programm umzusetzen. Hierzu wird jede Regel zu genau einer Methode, mit dem Namen der Regel. Das Ergebnis dieser Methode ist ein bool'scher Wert, der angibtg, ob die Anwendung der Regel erfolgreich einen Teil des Eingabesatzes erzeugen kann. Der Rumpf der Methode stellt die Rechte Seite einer Regel dar.

Sequenzen Auf der rechten Seite einer Regel steht eine Sequenz von Symbolen. Wir interpretieren diese als weitere Parser. Nichtterminalsymbole werden für uns Aufrufe der Methoden für das entsprechende Symbol, Terminalsymbole werden. Operational können wir die rechte Seite einer Regel als eine Sequenz von Parsläufen, die nacheinander auszuführen und alle erfolgreich sein müssen. Hierzu können wir ihre Aufrufe mit einen logischem *und* (`&&`) verknüpfen:

Beispiel:
Wir schreiben einen Parser für eine einfache Grammatik der Sprache, die die Sequenz dreier Zahlen als einzigen Satz hat:

start ::= *a* Zahl
a ::= Zahl Zahl

Für die zwei Regeln werden wir je eine Methode schreiben. Die Sequenzen von zwei Symbolen auf der rechten Seite jeder Regel führt zur Und-Verknüpfung der jeweiligen Parser für diese Symbole.

ThreeNumbers.java

```
1  public class ThreeNumbers
2   extends AbstractParser implements Parser{
3    public ThreeNumbers(Token[] token){this.token=token;}
4
5    public boolean parse(){return start();}
```

```
6
7     boolean start(){return a() && parseToken(Zahl.class);}
8
9     boolean a(){
10      return parseToken(Zahl.class) && parseToken(Zahl.class);}
11  }
```

Wir können Instanzen dieses einfachen Parser erzeugen und das jeweilige Parsergebnis ausgeben:

TestThreeNumbers.java

```
1   class TestThreeNumbers{
2     public static void main(String [] _){
3       Token [] correct = {new Zahl(42),new Zahl(44),new Zahl(52)};
4       Parser parser = new ThreeNumbers(correct);
5       System.out.println(parser.parse());
6
7       Token [] inCorrect = {new Zahl(42),new Zahl(44)};
8       parser = new ThreeNumbers(inCorrect);
9       System.out.println(parser.parse());
10    }
11  }
```

Alternativen Schließlich gilt es noch Regeln für Nichtterminalsymbole mit mehreren Alternativen in unserer Implementierung umzusetzen. Hierbei wollen wir vom aktuellen Stand aus nach der erste Alternative versuchen zu parsen und nur wenn dieses fehlschlägt es mit der zweiten Alternative versuchen. Es wird nötig vor diesem zweiten Versuch den Wert von `current` wieder auf seinen Ursprungswert zurückzusetzen.

Eine Regel mit zwei Alternativen der Form:

$a ::= alt1|alt2$

ist im Programm umzusetzen in die Form:

```
1   boolean a(){
2     int i = current;
3     if (alt1()) return true;
4     current=i;
5     return alt2();
6   }
```

Damit können wir jetzt alle Konstrukte einer kontextfreien Grammatik direkt in Javacode umsetzen.

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

Grammatik:

start ::= *addExpr*
addExpr ::= multExpr *addOp addExpr*| multExpr
multExpr ::= *zahl multOp multExpr*|*zahl*
multOp ::= Mult|Div
addOp ::= Add|Sub

In dieser gibt es vier weitere Terminalsymbole: für jede der vier Grundrechenarten jeweils eines. Wir schreiben vier entsprechende Klassen:

Add.java
```
1 public class Add implements Token{
2   public String toString(){return "+";}
3 }
```

Sub.java
```
1 public class Sub implements Token{
2   public String toString(){return "-";}
3 }
```

Mult.java
```
1 public class Mult implements Token{
2   public String toString(){return "*";}
3 }
```

Div.java
```
1 public class Div implements Token{
2   public String toString(){return "/";}
3 }
```

Wir schreiben einen Parser für diese Grammatik. Jede Regel wird eine Methode.

ExpressionParser.java
```
1  public class ExpressionParser
2    extends AbstractParser implements Parser{
3
4    public ExpressionParser(Token[] token){this.token=token;}
5
6    public boolean parse(){return start();}
7
8    boolean start(){
9      if (addExpr()){
10       return current == token.length;
11     }
12     return false;
13   }
14
15   boolean addExpr(){
16     int i = current;
17     if (multExpr() && addOp() && addExpr())
18       return true;
19     current = i;
20     return multExpr();
21   }
22
```

```
23    boolean multExpr(){
24      int i = current;
25      if(parseToken(Zahl.class) && multOp() && multExpr())
26        return true;
27      current = i;
28      return parseToken(Zahl.class);
29    }
30
31    boolean multOp(){
32      if (parseToken(Mult.class)) return true;
33      return parseToken(Div.class);
34    }
35
36    boolean addOp(){
37      if (parseToken(Add.class)) return true;
38      return parseToken(Sub.class);
39    }
40  }
```

Wir können jetzt Instanzen dieses Parsers erzeugen und Sätze mit ihr parsen.

TestExpressionParser.java

```
1  public class TestExpressionParser{
2    public static void main(String [] _){
3      Token [] correct = {new Zahl(2),new Add(),new Zahl(42)
4                          ,new Div(),new Zahl(42)};
5      Parser parser = new ExpressionParser(correct);
6      System.out.println(parser.parse());
7      Token [] inCorrect
8       = {new Zahl(2),new Mult(),new Zahl(42),new Div()};
9      parser = new ExpressionParser(inCorrect);
10     System.out.println(parser.parse());
11   }
12 }
```

Aufgabe 20 Punkteaufgabe (3 Punkte):
Für diese Aufgabe gibt es maximal 3 auf die Klausur anzurechnende Punkte. Abgabetermin: 17. Juli 2003.

Gegeben seien die zusätzlichen Tokenklassen

LPar.java

```
1 • public class LPar implements Token{
2     public String toString(){return "(";}
3   }
```

RPar.java

```
1 • public class RPar implements Token{
2     public String toString(){return ")";}
3   }
```

Die rechte und linke Klammern repräsentieren.

Zusätzlich sei die obige Grammatik für arithmetische Ausdrücke erweitert um geklammerte Ausdrücke:Als Beispiel betrachten wir eine Grammatik, die einfache arithmetische Ausdrücke generiert.

Grammatik:

start ::= *addExpr*
addExpr ::= multExpr *addOp addExpr*| multExpr
multExpr ::= *parExpr multOp multExpr*|*parExpr*
parExpr ::= Zahl | LPar *addExpr* RPar
multOp ::= Mult|Div
addOp ::= Add|Sub

Änder Sie die Klasse `ExpresssionParser` so ab, daß sie für die erweiterte Grammatik entscheidet, ob eine Tokenreihung mit ihr abgeleitet werden kann.
Benutzen Sie folgende Klasse als kleines Testbeispiel:

TestExpressionParser2.java

```
1  public class TestExpressionParser2{
2    public static void main(String [] _){
3      Token [] correct = {new LPar(),new Zahl(2),new Add()
4                          ,new Zahl(42),new RPar(),new Div()
5                          ,new Zahl(42)};
6      Parser parser = new ExpressionParser(correct);
7      System.out.println(parser.parse());
8      Token [] inCorrect
9       = {new Zahl(2),new Mult(),new LPar()
10        ,new Zahl(42),new LPar()};
11     parser = new ExpressionParser(inCorrect);
12     System.out.println(parser.parse());
13   }
14 }
```

Aufbau des Ableitungsbaums

Die Prüfung allein, ob ein Satz aus einer Grammmatik abgelitten werden kann, ist fast nie ausreichend. Über die syntaktische Prüfung hinaus soll der Satz auch eine Semantik bekommen. Hierzu ist es nicht ausreichend zu wissen, daß der Satz abgelitten werden kann, sondern auch wie. Dieses gibt der Ableitungsbaum an. Der Ableitungsbaum ist ein Baum dessen Blätter mit Terminalsymbolen markiert sind und dessen übrigen Knoten mit einem Regelnamen markiert sind.

In diesem Abschnitt wollen wir Parser schreiben, deren Ergebnis nicht ein bool'scher ist, sondern ein Baum. Hierzu können wir unsere einfache Klasse zur Darstellung allgemeiner Bäume benutzen. Die Schnittstelle für Parser reflektiert in den Rückgabetypen jetzt, daß wir Ableitungsbäume als Ergebnis bekommen wollen:

TParser.java

```
1  public interface TParser{
2    Tree parse();
3    Tree parseToken(Class tokenClass);
4    Tree epsilon();
5  }
```

Entsprechend ändern wir auch die Klasse für abstrakte Parser. Erfolglose Parsversuche ergeben den Wert `null`.

AbstractTParser.java

```
1  class AbstractTParser {
2
3    Token[] token ;
4    int current=0;
5
6    public Tree epsilon(){
7      return new Tree("epsilon");
8    }
9
10   public Tree parseToken(Class o){
11     if (current<token.length){
12       Token next = token[current];
13       if (o.isAssignableFrom(next.getClass())){
14         current=current+1;
15         return new Tree(next);
16       }
17     }
18     return null;
19   }
20 }
```

Schließlich sind die Regeln einer Grammatik Stück für Stück so umzusetzen, daß sie einen Ableitungsbaum zurückgeben. Hierzu reicht es für die Sequenz nicht mehr aus, die Aufrufe der einzelnen Parser mit dem logischem *Und* zu verknüpfen. Die Ergebnisse sind in diesem Fall nicht mehr bool'sche Werte, sondern Bäume. Deshalb müssen wir jeden einzelnen Ergebniswert einer Parsermethode mit einer `if`-Abfrage darauf prüfen, ob sie eventuell `null` ist. Erst wenn dieses nicht der Fall ist, kann mit den restlichen Parsermethoden der Sequenz weitergemacht werden.

Beispiel:
Wie nehmen dieselbe Grammatik aus dem letzten Beispiel und ändern den Parser so um, daß er jetzt einen Ableitungsbaum als Ergebnis liefert.

ExpressionTParser.java

```
1  import java.util.List;
2  import java.util.ArrayList;
3
4  public class ExpressionTParser
5    extends AbstractTParser
```

```
6   implements TParser{
7
8   public ExpressionTParser(Token[] token){this.token=token;}
9
10  public Tree parse(){return start();}
11
12  Tree start(){
13    Tree result = addExpr();
14    if (current != token.length){
15      return null;
16    }
17    return result;
18  }
19
20  Tree addExpr(){
21    int i = current;
22    List children = new ArrayList();
23    Tree x1 = multExpr();
24    if (x1!=null){
25      Tree x2 = addOp();
26      if (x2!=null){
27        Tree x3 = addExpr();
28        if (x3!=null){
29          children.add(x1);
30          children.add(x2);
31          children.add(x3);
32          return new Tree("addExpr",children);
33        }
34      }
35    }
36    current = i;
37    x1 = multExpr();
38    if (x1!=null){
39      children.add(x1);
40      return new Tree("addExpr",children);
41    }
42    return null;
43  }
44
45
46  Tree multExpr(){
47    int i = current;
48    List children = new ArrayList();
49    Tree x1 = parseToken(Zahl.class) ;
50    if (x1!=null){
51      Tree x2 = multOp();
52      if (x2!=null){
53        Tree x3 = multExpr();
54        if (x3!=null){
55          children.add(x1);
```

```
56            children.add(x2);
57            children.add(x3);
58            return new Tree("multExpr",children);
59          }
60        }
61      }
62      current = i;
63      x1 = parseToken(Zahl.class);
64      if (x1!=null){
65        children.add(x1);
66        return new Tree("multExpr",children);
67      }
68      return null;
69    }
70
71    Tree multOp(){
72      List children = new ArrayList();
73      Tree x1 = parseToken(Mult.class);
74      if (x1!=null){
75         children.add(x1);
76         return new Tree("multOp",children);
77      }
78      x1 = parseToken(Div.class);
79      if (x1!=null){
80         children.add(x1);
81         return new Tree("multOp",children);
82      }
83      return null;
84    }
85
86    Tree addOp(){
87      List children = new ArrayList();
88      Tree x1 = parseToken(Add.class);
89      if (x1!=null){
90         children.add(x1);
91         return new Tree("addOp",children);
92      }
93      x1 = parseToken(Sub.class);
94      if (x1!=null){
95         children.add(x1);
96         return new Tree("addOp",children);
97      }
98      return null;
99    }
100 }
```

Wir können wieder Instanzen dieses Parser erzeugen, und Test durchführen. Die Ausgabe hängt davon ab, wie die Methode `toString` in der Klasse `Tree` implementiert ist.

TestExpressionTParser.java

```
1  public class TestExpressionTParser{
2    public static void main(String [] _){
3      Token [] correct = {new Zahl(3),new Add(),new Zahl(42)
4                          ,new Div(),new Zahl(42)};
5      TParser parser = new ExpressionTParser(correct);
6      System.out.println(parser.parse());
7      Token [] correct2 = {new Zahl(3),new Div(),new Zahl(42)
8                           ,new Add(),new Zahl(42)};
9      parser = new ExpressionTParser(correct2);
10     System.out.println(parser.parse());
11     Token [] inCorrect
12       = {new Zahl(2),new Mult(),new Zahl(42),new Div()};
13     parser = new ExpressionTParser(inCorrect);
14     System.out.println(parser.parse());
15   }
16 }
```

Dieses Programm kann z.B. folgende Ausgabe haben:

```
sep@swe10:~/fh/prog2/beispiele/buildTree> java ExpressionTParser
addExpr[multExpr[3], addOp[+], addExpr[multExpr[42, multOp[/], multExpr[42]]]]
addExpr[multExpr[3, multOp[/], multExpr[42]], addOp[+], addExpr[multExpr[42]]]
null
sep@swe10:~/fh/prog2/beispiele/buildTree>
```

Betrachten wir nun die graphische Baumdarstellung der beiden erfolgreichen obigen Tests:

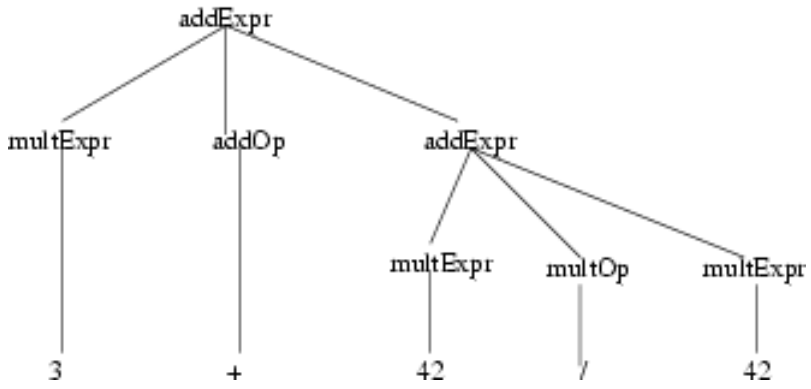


Abbildung 5.4: 3+42/42

- `3+42/42` ergibt den Baum aus Abbildung 5.4
- `3*42+42` ergibt den Baum aus Abbildung 5.5

Man erkennt, daß die Grammatik gerade so aufgebaut ist, daß die Knoten für Strichrechnung jeweils höher im Ableitungsbaum stehen aus die Knoten der Punktrechnung. Damit drückt die Grammatik bereits die Operatorprezedenz

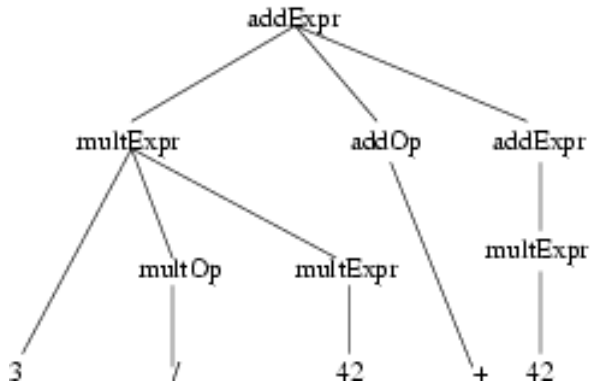


Abbildung 5.5: 3*42+42

aus. Man kann den Baum als Rechenvorschrift intgerpretieren: ein Knoten mit drei Kindern wird in der Weise ausgewertet, daß erst die Ergebnisse des ersten und dritten Kindes berechnet werden, und dann die beiden Ergebnisse mit dem Operator, der als zweites Kind angehängt ist, verrechnet werden.

Aufgabe 21 Aufgrund der Struktur der Grammatik für den Parser `ExpressionTParser`, die die Operatorpräzedenz berücksichtigt, kommt es im Ableitungsbaum zu mehr oder weniger redundanten Knoten, die jeweils nur ein Kind haben. Diese Knoten tragen für die Struktur eines Ausdrucks für uns keinerlei interessanten Informationen.

Ändern Sie die Klasse `ExpressionTParser` so ab, daß der erzeugte Ableitungsbaum keine solchen Knoten mit nur einem Kind mehr enthält.

Aufgabe 22 Gegeben seien die folgende Schnittstelle und Klassen:

```
1 • public interface Expression {
2     int eval();
3   }
```

```
1 • public class AddExpr implements Expression{
2     Expression left;
3     Expression right;
4     public AddExpr(Expression left,Expression right){
5       this.left=left;
6       this.right=right;
7     }
8     public int eval(){
9       return left.eval()+right.eval();
10    }
11  }
```

```
1 • public class IntExpr implements Expression{
2     int i;
```

```
3     pubic  IntExpr(int i){this.i=i;}
4     public eval(){return i;}
5   }
```

a) Implementieren Sie entsprechende Klassen `SubExpr`, `MultExpr`, `DivExpr`.

b) Ändern Sie die Klasse `ExpressionTParser` so ab, daß statt des Ableitungsbaumes ein Objekt des Typs `Expression` erzeugt wird.

c) Schreiben Sie eine Methode `static public int calculate(Token [] expr)`, die einen Ausdruck mit dem Parser `ExpressionTParser` parst und durch Aufruf der Methode `eval` das Ergebnis berechnet.

5.2.3 Kritik an unserer Implementierung

Wir haben eine möglichst naive und einfach Implementierung eines Parsers geschrieben. Sie ist in mehrfacher Hinsicht nicht besonders ausgereift.

- Wir haben keinen Tokenizer geschrieben.
- Der Satz, der geparst werden soll muß komplett in einer Reihung vorliegen. Geschickter wäre es sich erst nach Bedarf die Token aus einem Strom herauszuholen.
- Es gibt keine Fehlerbehandlung für Sätze, die nicht geparst werden können.
- Wir haben keine `trace`-Möglichkeit integriert.

5.2.4 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  import java.util.List;
2  import java.util.ArrayList;
3  import java.io.Reader;
4  import java.io.StringReader;
5  import java.io.IOException;
6
7  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
```
8  Reader reader;
9  int next=0;
10 List/*Token*/ result = new ArrayList();
```

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

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

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

Für Leerzeichen wird kein Token erzeugt:

ArithTokenizer.java
```
22         case '\u0020'  : break;
23         case '\n' : break;
24         case '\t' : break;
```

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

ArithTokenizer.java
```
25         case '+'  : result.add(new Add()) ; break;
26         case '-'  : result.add(new Sub()) ; break;
27         case '*'  : result.add(new Mult()); break;
28         case '/'  : result.add(new Div()) ; break;
29         case '('  : result.add(new LPar()); break;
30         case ')'  : result.add(new RPar()); 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

```
31          default   : if (Character.isDigit(c)) {
32                        result.add(getInt());
33                        continue;
34                      }else throw new Exception
35                            ("unexpected Token found: '"+c+"'");
36        }
```

Schließlich können wir den Strom um ein neues Zeichen bitten:

ArithTokenizer.java

```
37
38        next = reader.read();
39      }
40    }catch (IOException _){}
41    return result;
42  }
43
```

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

ArithTokenizer.java

```
44
45  Token getInt() throws IOException {
46    int res=0;
47    while (next>=0 && Character.isDigit((char)next)){
48      res=res*10+next-48;
49        next = reader.read();
50    }
51
52    return new Zahl(res);
53  }
54
```

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

ArithTokenizer.java

```
55
56  static List tokenize(String inp) throws Exception{
57    return tokenize(new StringReader(inp));
58  }
59
60  static List tokenize(Reader inp) throws Exception{
61      return new ArithTokenizer(inp).tokenize();
62  }
63}
```

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, +, 9, (, (, *, +, -, /, 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  import java.io.*;
2  import javax.imageio.ImageIO;
3  import javax.swing.*;
4
5  class ArithParsTreeBuilder{
6
7    public static void main(String [] args) throws Exception{
8      final String fileName = args[0];
9      writeParsTreeFile(new FileReader(fileName+".arith"),fileName);
10   }
11
12   static void writeParsTreeFile(Reader reader,String fileName)
13                                                  throws Exception{
14     ImageIO.write
15       (TreeToFile.createTreeImage(parsArith(reader))
16       ,"png"
17       , new File(fileName+".png")
18       );
19   }
20
21   public static Tree parsArith(Reader reader) throws Exception{
22     Token [] ar = {};
23     return new ExpressionTParser
24               ((Token[])ArithTokenizer
25                           .tokenize(reader)
26                           .toArray(ar))
27              .start();
28   }
29
30   public static DisplayTree parseAndShow(String inp)
31                                       throws Exception{
```

```
32      return new DisplayTree(parsArith(new StringReader(inp)));
33    }
34
35    public static void parseAndShowInFrame(String inp)
36                                          throws Exception{
37      JComponent c = ArithParsTreeBuilder.parseAndShow(inp);
38      JFrame f = new JFrame();
39      f.getContentPane().add(c);
40      f.pack();
41      f.setVisible(true);
42    }
43  }
```

5.3 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, sogennante 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].

5.3.1 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.jjt

```
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.jjt

```
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 | }
```

[3]SMS stehe hier für *Sonne Mond und Sterne*.

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.jjt

```
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()
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.jjt` 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 {
```

```
                /* bgen(jjtree) start */
  SimpleNode jjtn000 = new SimpleNode(JJTSTART);
  boolean jjtc000 = true;
  jjtree.openNodeScope(jjtn000);
    try {
      nounPhrase();
      verbPhrase();
    } catch (Throwable jjte000) {

...



  static final public void nounPhrase() throws ParseException {
                     /* bgen(jjtree) nounPhrase */
  SimpleNode jjtn000 = new SimpleNode(JJTNOUNPHRASE);
  boolean jjtc000 = true;
  jjtree.openNodeScope(jjtn000);
    try {
      switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
      case MOON:
      case MARS:
      case MERCURY:
      case PHOEBUS:
      kcase DEIMOS:


  public SMS(java.io.InputStream stream) {
    if (jj_initialized_once) {


....

```

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.

```
public class MainSMS {
  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?

5.3.2 Parsen und moderne Programmierparadigmen

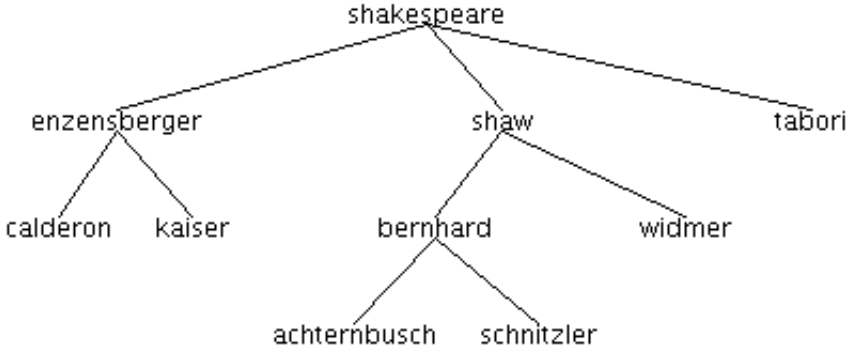
Wie an den Jahreszahlen der Literaturangaben zu sehen, sind Parser ein sehr altes Geschäft. Die Grundlagen sind zu einer Zeit entwickelt worden, in der es weder objektorientierte noch moderne funktionale Programmiersprachen gab. Daher sind Parser- und vergleichbare Algorithmen häufig in einer sehr prozeduralen 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

Beispielaufgaben

Aufgabe 1 Betrachten Sie folgenden Baum.



a) Handelt es sich bei diesen Baum um einen binären Suchbaum (mit lexikographischer Ordnung auf den Stringmarkierungen)? Begründung?

Wenn nicht, dann zeichnen Sie einen binären Suchbaum, der die gleichen Elemente enthält.

b) Wie könnten Sie vorgehen, um aus einem Binärbaum einen beliebigen Knoten zu löschen? Malen sie hierzu Beispiele und beschreiben Sie ihr Vorgehen.

Aufgabe 2 Betrachten Sie das folgende Javamethoden, die der Baumklasse `Tree` hinzugefügt wurden. Rechnen Sie die Methoden auf dem Papier für den Baum aus Aufgabe 1 durch und geben ihr Ergebnis an.

Was berechnen die Methoden?

a)

```
int foo1(){
  int result = theChildren().size();
  for (Iterator it=theChildren().iterator();it.hasNext();){
    final int n = ((Tree)it.next()).foo1();
    if (n>result) result=n;
  }
```

```
7   return result;
8 }
```

b)

```
List foo2(List result){
  result.add(mark());
  final List xs= theChildren();
  if (!xs.isEmpty()){
    try {
      ((Tree)xs.get(1)).foo2(result);
    }catch (Exception _){}
  }
  return result;
}
```

Aufgabe 3 Gegeben sei die folgende abstrakte Klasse für Bäume, gemäß unserer Spezifikation aus der Vorlesung:

AbTree.java

```
import java.util.*;

abstract class AbTree{
  abstract public AbTree tree(Object mark, List/*AbTree*/ xs);
  abstract public Object mark();
  abstract public List/*AbTree*/ theChildren();
}
```

Schreiben Sie für die Klasse folgende Methoden, die ihr hinzugefügt werden können:

a) `int howOftenContained(Object o)`: zählt, wieviel Knoten mit einem Objekt markiert sind, das gleich dem Argument `o` ist.

b) `AbTree mapToUpperCase()` ergibt einen neuen Baum, der die gleiche Struktur hat und dessen Knoten mit Strings markiert sind, die sich ergeben, indem man die Markierungen des Ausgangsbaums mit `mark().toString().toUpperCase()` umwandelt.

Aufgabe 4 Gegeben seien folgende Gui-Komponenten:

SimpleFrame.java

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

public class SimpleFrame extends JFrame{
  JLabel l = new JLabel("dies ist ein kleines Fenster");

  public SimpleFrame(){
    super("kleines Fenster");
    getContentPane().add(l);
    pack();
    setVisible(true);
```

```
12    }
13  }
```

GoodbyeFrame.java

```
1•  import javax.swing.*;
2   import java.awt.event.*;
3
4   public class GoodbyeFrame extends JFrame{
5     JLabel l  = new JLabel("Danke für Ihr Vertrauen!");
6     JButton b = new JButton("Programm beenden");
7     JPanel p  = new JPanel();
8
9     public GoodbyeFrame(){
10      super("goodbey Fenster");
11      p.add(l);
12      p.add(b);
13      getContentPane().add(p);
14      pack();
15      setVisible(true);
16    }
17  }
```

a) Erweitern Sie die Komponente `GoodbeyFrame` so, daß beim Drücken des Knopfes, das Programm mit `System.exit(0)` beendet wird.

b) Erweitern Sie die Komponente `SimpleFrame` so, daß beim Schließen des Fensters ein Objekt des Typs `GoodbyeFrame` erzeugt wird.

Aufgabe 5 Gegeben sei folgende Grammatik:

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

Das Terminalsymbol `zahl` steht dabei für eine beliebige Zahl.

a) Leiten Sie den Ausdruck `1*4+42/5-6` mit der rekursiv absteigenden Parserstrategie ab.

b) Zeichnen Sie den Ableitungsbaum für Ihre obige Ableitung.

c) Die Grammatik erlaubt kein unäres Minussymbol zum Negieren einer Zahl. Ergänzen Sie die Grammatik, so daß dieses erlaubt ist. Zeigen Sie damit die Ableitung des Audrucks `-12 * -1 -4 + 2`.

Anhang B

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 Referate gehalten worden:

B.1 JNI: Javas Schnittstelle zu C

Wie können Javaprogramme mit C-Programmen kommunizieren. Hierzu gibt es JNI (java native interface).

B.2 Java 2D Grafik

In Java gibt es ein umfangreiches Paket zum erzuegen und zeichnen graphischer Objekte.

B.3 Java 3D Grafik

In Java gibt es ein umfangreiches Paket zum erzuegen und zeichnen animierter dreidimensionaler graphischer Objekte.

B.4 Java und Netzwerk

Wie greift man mit Java auf Netzwerkresourcen zu.

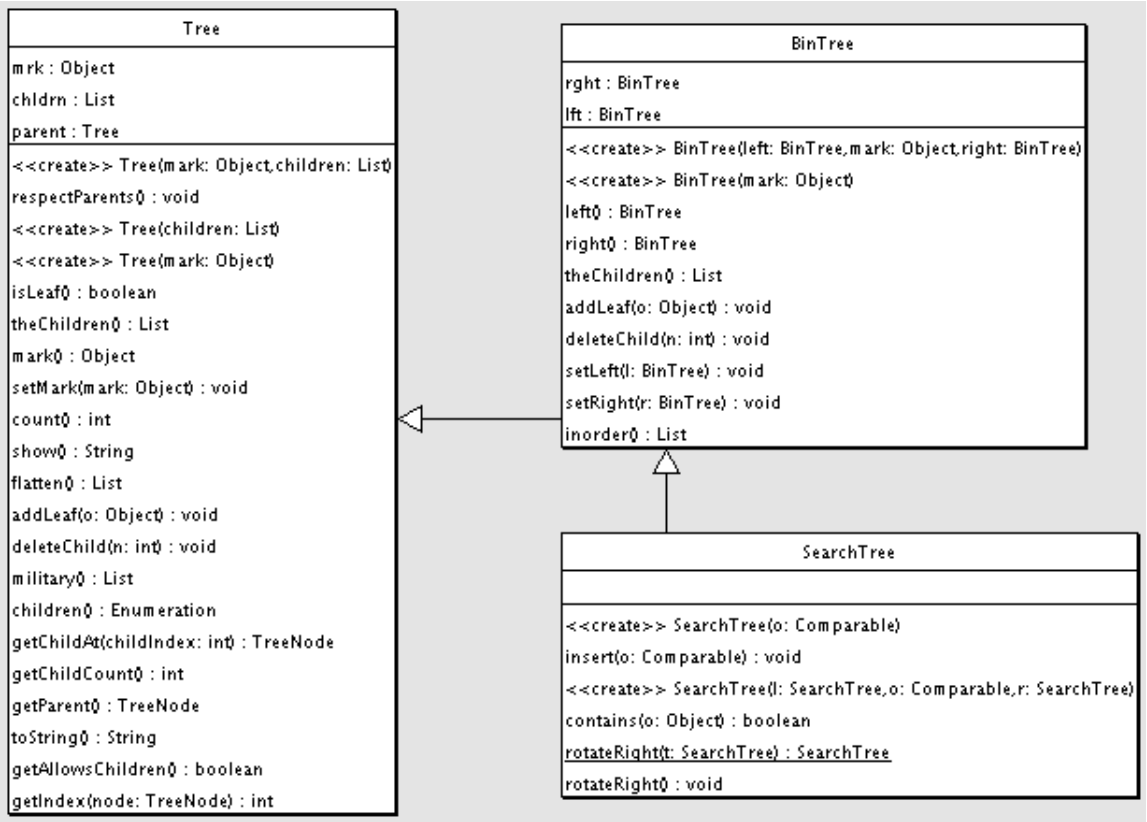
B.5 Java und Datenbanken

Wie greift man mit Java auf eine Datenbank zu.

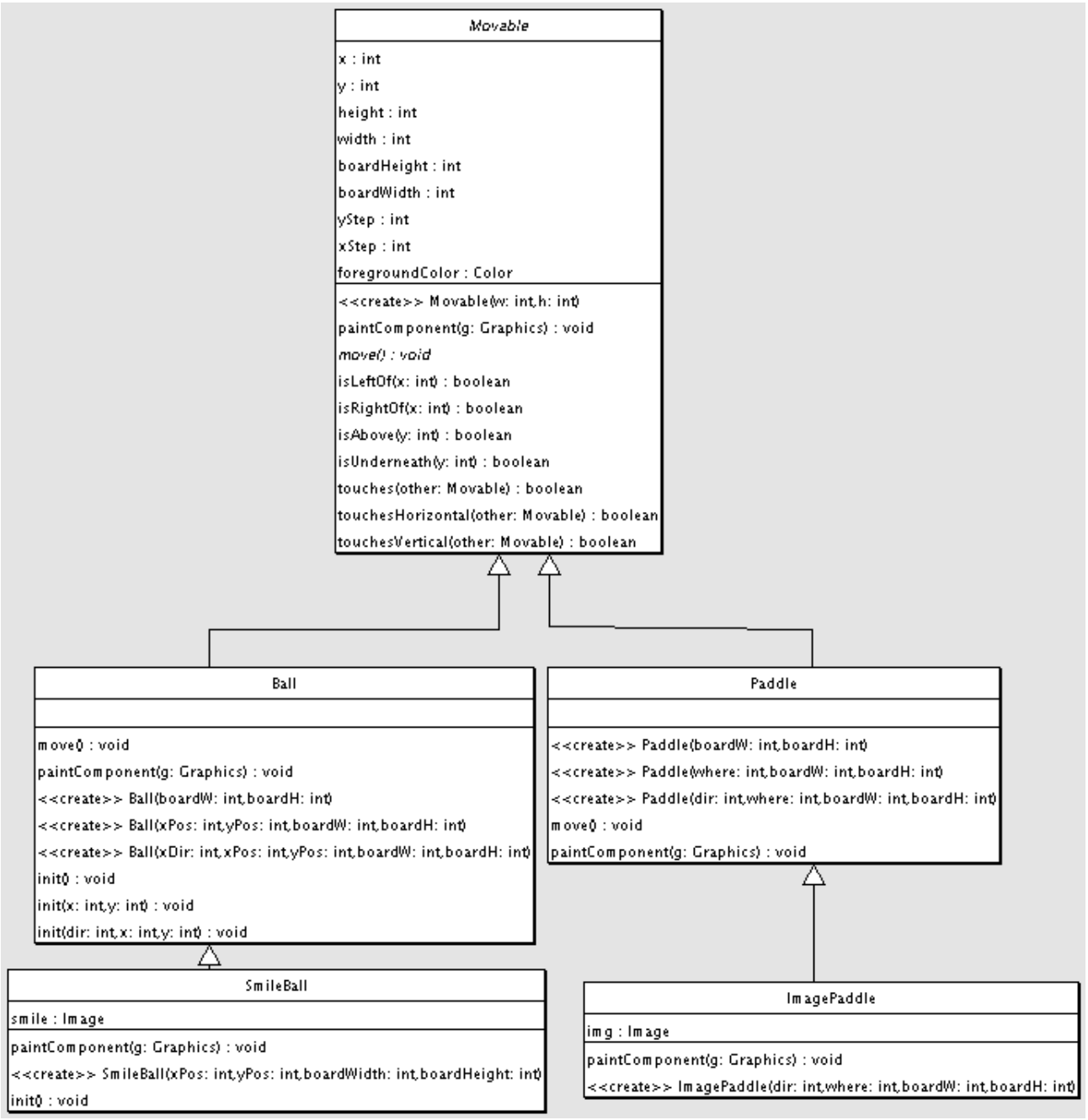
Anhang C

UML Diagramme

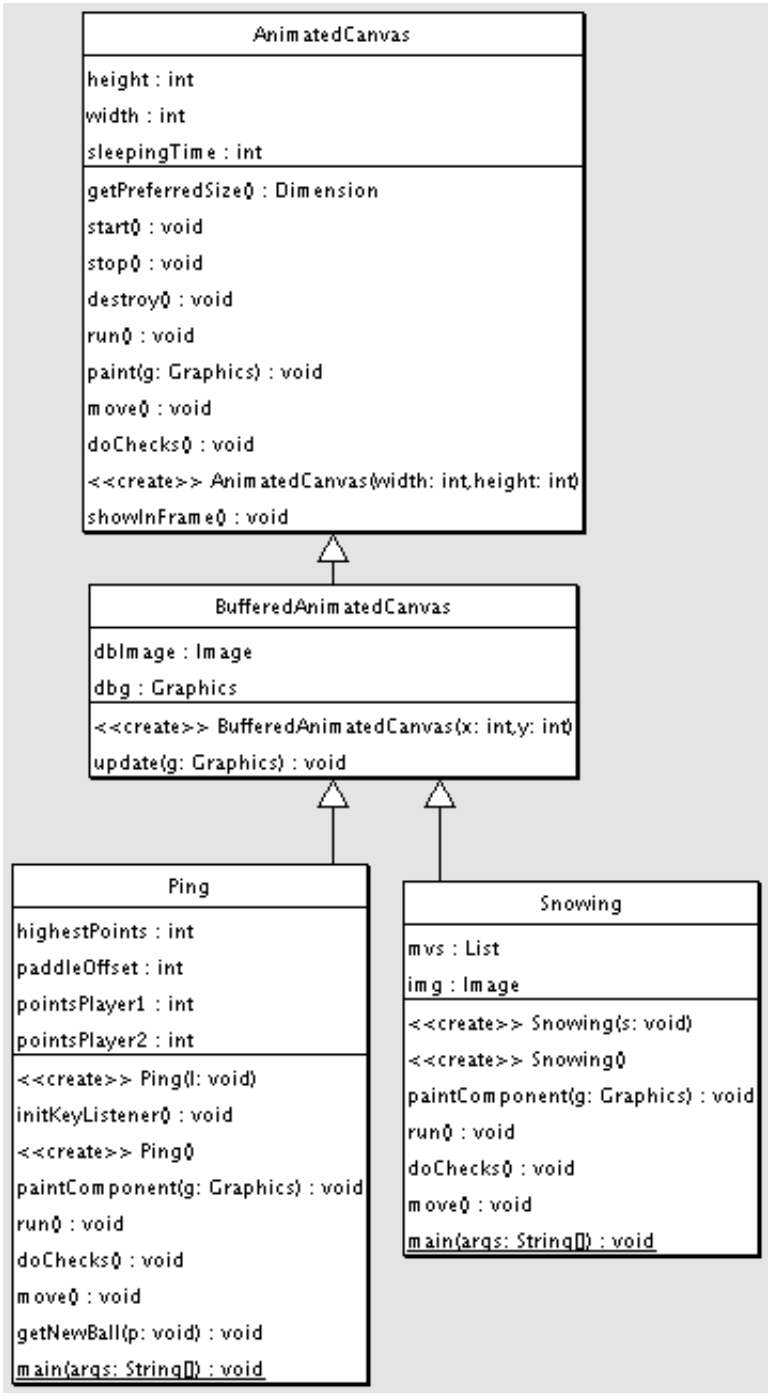
C.1 Tree



C.2 Movable


Movable
x : int
y : int
height : int
width : int
boardHeight : int
boardWidth : int
yStep : int
xStep : int
foregroundColor : Color
<<create>> Movable(w: int,h: int)
paintComponent(g: Graphics) : void
move() : void
isLeftOf(x: int) : boolean
isRightOf(x: int) : boolean
isAbove(y: int) : boolean
isUnderneath(y: int) : boolean
touches(other: Movable) : boolean
touchesHorizontal(other: Movable) : boolean
touchesVertical(other: Movable) : boolean
Ball
move() : void
paintComponent(g: Graphics) : void
<<create>> Ball(boardW: int,boardH: int)
<<create>> Ball(xPos: int,yPos: int,boardW: int,boardH: int)
<<create>> Ball(xDir: int,xPos: int,yPos: int,boardW: int,boardH: int)
init() : void
init(x: int,y: int) : void
init(dir: int,x: int,y: int) : void
Paddle
<<create>> Paddle(boardW: int,boardH: int)
<<create>> Paddle(where: int,boardW: int,boardH: int)
<<create>> Paddle(dir: int,where: int,boardW: int,boardH: int)
move() : void
paintComponent(g: Graphics) : void
SmileBall
smile : Image
paintComponent(g: Graphics) : void
<<create>> SmileBall(xPos: int,yPos: int,boardWidth: int,boardHeight: int)
init() : void
ImagePaddle
img : Image
paintComponent(g: Graphics) : void
<<create>> ImagePaddle(dir: int,where: int,boardW: int,boardH: int)

C.3 BufferedAnimatedCanvas



Anhang D

Ausgewählte Klassen

D.1 Tree

Tree.java

```
1  import java.util.List;
2  import java.util.Iterator;
3  import java.util.Enumeration;
4  import java.util.ArrayList;
5  import javax.swing.tree.TreeNode;
6
7  public class Tree implements TreeNode {
8    private Object mrk;
9    private List/*Tree*/ chldrn;
10
11   Tree parent;
12
13   public Tree(Object mark,List children){
14     this.mrk=mark;
15     this.chldrn=children;
16     this.parent=null;
17     respectParents();
18   }
19
20   public void respectParents(){
21     for (Iterator it=theChildren().iterator();it.hasNext();){
22       ((Tree)it.next()).parent=this;
23     }
24   }
25
26   public Tree(List children){
27     this(null,children);
28   }
29
30   public Tree(Object mark){
31     this(mark,new java.util.ArrayList());
```

```
32  }
33
34  public boolean isLeaf(){
35    return theChildren().isEmpty();
36  }
37
38  public List/*Tree*/ theChildren(){return chldrn;}
39
40  public Object mark(){return mrk;}
41
42  public void setMark(Object mark){mrk=mark;}
43
44  public int count(){
45   int result = 1;
46   for (Iterator it=theChildren().iterator();it.hasNext();)
47     result=result+((Tree)it.next()).count();
48   return result;
49  }
50
51  public void addLeaf(Object o){
52    List chs = theChildren();
53    chs.add(new Tree(o));
54  }
55
56  public void deleteChild(int n){
57    theChildren().remove(n);
58  }
59
60  public List military(){
61    List result=new ArrayList();
62    List toDo = theChildren();
63    result.add(mark());
64    while (!toDo.isEmpty()){
65      List nextGeneration=new ArrayList();
66      for (Iterator it=toDo.iterator();it.hasNext();){
67        Tree nextChild = (Tree)it.next();
68        result.add(nextChild.mark());
69        nextGeneration.addAll(nextChild.theChildren());
70      }
71      toDo=nextGeneration;
72    }
73    return result;
74  }
75
76  public Enumeration/*Tree*/ children() {
77    final Iterator it = theChildren().iterator();
78
79    return new Enumeration(){
80      public boolean hasMoreElements() {return it.hasNext();}
81      public Object nextElement() {return it.next();}
```

```
82      };
83    }
84
85    public TreeNode getChildAt(int childIndex) {
86      return (TreeNode)theChildren().get(childIndex);
87    }
88
89    public int getChildCount() {return theChildren().size();}
90
91    public TreeNode getParent() {
92      return parent;
93    }
94
95    public String toString(){return mark().toString();}
96
97    public boolean getAllowsChildren(){return false;}
98
99    public int getIndex(TreeNode node) {return 0;}
100
101
102 public List filter(List result,TreeCondition c){
103   //wenn die Bedingung für den Knoten wahr ist, füge
104   //ihm dem Ergebnis hinzu
105   if (c.takeThis(this)) result.add(mark());
106
107   //für alle Kinder
108   for (Iterator it=theChildren().iterator();it.hasNext();){
109     //rufe den Filter mit gleicher Bedingung auf
110     ((Tree)it.next()).filter(result,c);
111   }
112   return result;
113 }
114
115
116 public List filter(TreeCondition c){
117   return filter(new ArrayList(),c);
118 }
119
120
121 public List leaves(){
122   return filter(new TreeCondition(){
123                   public boolean takeThis(Tree t){return t.isLeaf();}
124                 });
125 }
126
127 public List flatten(){
128   return filter(new TreeCondition(){
129                   public boolean takeThis(Tree _){return true;}
130                 });
131 }
```

```
132
133 public List nodesWithTwoChildren(){
134   return filter(new TreeCondition(){
135                   public boolean takeThis(Tree t){
136                     return t.theChildren().size()==2;
137                   }
138                 });
139 }
140
141 public List nodesMarkedWith(final Object o){
142   return filter(new TreeCondition(){
143                   public boolean takeThis(Tree t){
144                     return t.mark().equals(o);
145                   }
146                 });
147 }
148
149 public boolean contains(Object o){
150   return !nodesMarkedWith(o).isEmpty();
151 }
152
153
154   public List getPathBad(Object o){
155     List result = new ArrayList();
156     if (mark().equals(o)) { result.add(mark()); return result;}
157
158     for (Iterator it=theChildren().iterator();it.hasNext();){
159       final Tree n = (Tree)it.next();
160       if (n.contains(o)) {
161           result.add(mark());
162           result.addAll(n.getPathBad(o));
163       }
164     }
165     return result;
166 }
167
168 public List getPathFilter(final Object o){
169    return filter(
170       new TreeCondition(){
171         public boolean takeThis(Tree e){
172           return e.contains(o);
173         }
174       }
175    );
176 }
177
178 public List getPath(Object o){
179       return getPath(o,new ArrayList());
180   }
181
```

```
182   public List getPath(Object o,List result){
183     if (mark().equals(o)) { result.add(mark()); return result;}
184
185     for (Iterator it=theChildren().iterator();it.hasNext();){
186       final List path = ((Tree)it.next()).getPath(o,result);
187       if (!path.isEmpty()){
188           result.add(0,mark());
189           return result;
190       }
191     }
192     return result;
193   }
194
195   public StringBuffer show(String indent,StringBuffer result){
196     //zeige diesen Knoten mit Einrückung
197     result.append(indent);
198     result.append(mark().toString());
199
200     //für jedes Kind
201     for (Iterator it=theChildren().iterator();it.hasNext();){
202      //erzeuge neue Zeile und zeige Kind mit neuer Einrückung
203       result.append("\n");
204       ((Tree)it.next()).show(indent+"  ",result);
205     }
206     return result;
207   }
208
209   public String show(){
210       return show("",new StringBuffer()).toString();
211   }
212
213 }
```

Anhang E

Gesammelte Aufgaben

Aufgabe 1 Nehmen Sie eines Ihrer Javaprojekte des letzen Semesters (z.B. Eliza oder VierGewinnt) und verpacken die `.class` Dateien des Projektes in eine JAR-Datei. Berücksichtigen Sie dabei die Paketstruktur, die sich in der Ordnerhierarchie der `.class` Dateien wiederspiegelt.

Aufgabe 2 Starten Sie die Anwendung, die Sie in der letzten Aufgabe als JAR-Datei verpackt haben, mit Hilfe der `java`-Option: `-cp`.

Aufgabe 3 Laden Sie die JAR-Datei:
jugs.jar (`http://www.tfh-berlin.de/~panitz//prog2/load/jugs.jar`) .
In diesem Archive liegt eine Javaanwendung, die eine interaktive Javaumgebung bereitstellt. Javaausdrücke und Befehle können eingegeben und direkt ausgeführt werden. Das Archiv enthält zwei Klassen mit einer Hauptmethode:

- `Jugs`: ein Kommandozeilen basierter Javainterpreter.
- `JugsGui`: eine graphische interaktive Javaumgebung.

Um diese Anwendung laufen zu lassen, wird ein zweites Javaarchive benötigt: die JAR-Datei `tools.jar`. Diese befindet sich in der von Sun gelieferten Entwicklungsumgebung.

Setzen Sie den Klassenpfad (einmal per Javaoption, einmal durch neues Setzen der Umgebungsvariablen `CLASSPATH`) auf die beiden benötigten JAR-Dateien und starten Sie eine der zwei Hauptklassen. Lassen Sie folgende Ausdrücke in *Jugs* auswerten.

- `2*21`
- `"hello world".toUpperCase().substring(2,5)`
- `System.getProperties()`
- `System.getProperty("user.name")`

Aufgabe 4 Erzeugen Sie ein Baumobjekt, das einen Stammbaum Ihrer Familie darstellt. Die Knoten und Blätter sind mit Namen markiert. Die Wurzel sind Sie. Kinderknoten sind mit den leiblichen Eltern der Person eines Knotens markiert.

Aufgabe 5 Testen Sie die in diesem Abschnitt entwickelten Methoden auf Bäumen mit dem in der letzten Aufgabe erzeugten Baumobjekt.

Aufgabe 6 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(Object 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.

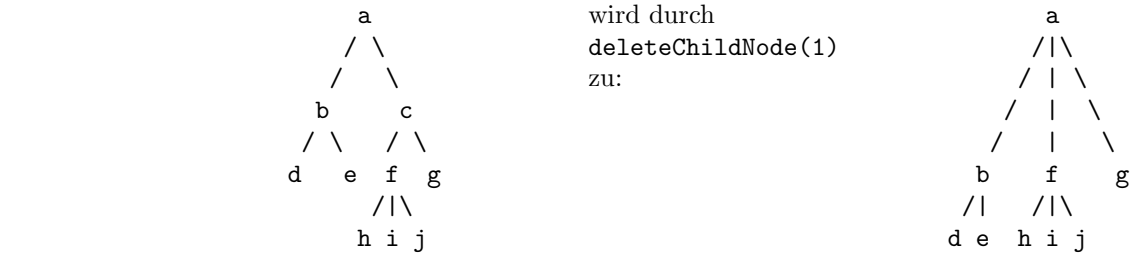
e) `List getPath(Object o)`: gibt die Markierungen auf dem Pfad von der Wurzel zu dem Knoten, der mit dem Objekt `o` markiert ist, zurück. Ergebnis ist eine leere Liste, wenn ein Objekt gleich `o` nicht existiert.

f) `java.util.Iterator 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) `boolean sameStructure(Tree other)`: soll genau dann wahr sein, wenn der Baum `other` die gleiche Struktur hat. Die Markierungen der Knoten können hingegen unterschiedlich sein.

Aufgabe 7 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 8 Schreiben Sie eine Methode `List/*Tree*/ siblings()`, die die Geschwister eines Knotens als Liste ausgibt. In dieser Liste der Geschwister soll der Knoten selbst nicht auftauchen. Passen Sie auf, daß Sie keine modifizierende Methode schreiben.

Aufgabe 9 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 10 Schreiben Sie analog zur Methode `flatten` in der Klasse `Tree` eine Methode `List postorder()`, die die Knoten eines Baumes in Postordnung linearisiert.

Aufgabe 11 Rechnen Sie auf dem Papier ein Beispiel für die Arbeitsweise der Methode `military()`.

Aufgabe 12 Zeichnen Sie die Baumstruktur, die im Programm `TestSearchTree` aufgebaut wird.

Aufgabe 13 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 14

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 15 Erzeugen Sie ein Objekt des Typs `XML`, das folgende Serialisierung hat:

```
<drama>
 <title>Macbeth</title>
 <author>William Shakespear</author>
 <personae>
  <persona>Macbeth</persona>
  <persona>Lady Macbeth</persona>
  <persona>Duncan</persona>
  <persona>Banquo</persona>
  <persona>Macduff</persona>
  <persona>Lady Macduff</persona>
  <persona>Ross</persona>
 </personae>
</drama>
```

Aufgabe 16 Überschreiben Sie in den Klassen `Element` und `TextNode` die Methode `toString` so, daß sie die Serialisierung von XML als Ergebnis hat.

Schreiben Sie Tests für ihre neue Methode `toString`

Aufgabe 17 (Punkteaufgabe 3 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:

```
public static void main(String [] args) {
  StrichKreis k = new StrichKreis(120);
  JFrame f = new ClosingFrame();
  JPanel p = new JPanel();
  p.add(new StrichKreis(120));
  p.add(new StrichKreis(10));
  p.add(new StrichKreis(60));
  p.add(new StrichKreis(50));
  p.add(new StrichKreis(70));
  f.getContentPane().add(p);

  f.pack();
  f.setVisible(true);
}
```

Hinweis: In der Klasse `java.lang.Math` finden Sie Methoden trigeometrischer Funktionen. Insbesondere `toRadians`, `sin` und `cos`.

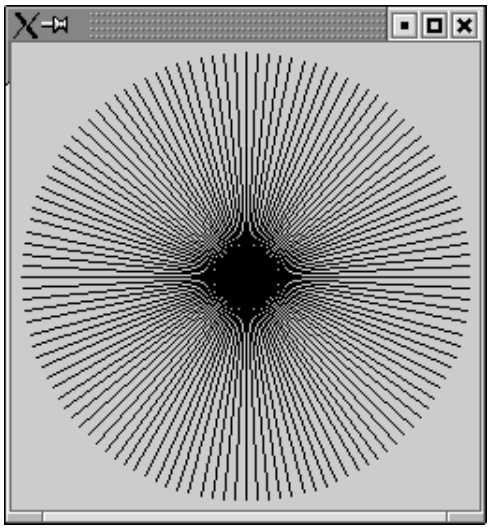


Abbildung E.1: Kreis mit Durchmesserlinien.

Aufgabe 18 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 19 Betrachten Sie die einfache Grammatik für arithmetische Ausdrücke aus dem letzen Abschnitt. Zeichnen Sie einen Syntaxbaum für den Audruck `1+1+2*20`.

Aufgabe 20 Punkteaufgabe (3 Punkte):
Für diese Aufgabe gibt es maximal 3 auf die Klausur anzurechnende Punkte. Abgabetermin: 17. Juli 2003.

Gegeben seien die zusätzlichen Tokenklassen

LPar.java

```
1 • public class LPar implements Token{
2     public String toString(){return "(";}
3   }
```

RPar.java

```
1 • public class RPar implements Token{
2     public String toString(){return ")";}
3   }
```

Die rechte und linke Klammern repräsentieren.

Zusätzlich sei die obige Grammatik für arithmetische Ausdrücke erweitert um geklammerte Ausdrücke:Als Beispiel betrachten wir eine Grammatik, die einfache arithmetische Ausdrücke generiert.

Grammatik:

start ::= *addExpr*
addExpr ::= multExpr *addOp addExpr*| multExpr
multExpr ::= *parExpr multOp multExpr*|*parExpr*
parExpr ::= Zahl | LPar *addExpr* RPar
multOp ::= Mult|Div
addOp ::= Add|Sub

Änder Sie die Klasse `ExpresssionParser` so ab, daß sie für die erweiterte Grammatik entscheidet, ob eine Tokenreihung mit ihr abgeleitet werden kann.
Benutzen Sie folgende Klasse als kleines Testbeispiel:

TestExpressionParser2.java

```
1  public class TestExpressionParser2{
2    public static void main(String [] _){
3      Token [] correct = {new LPar(),new Zahl(2),new Add()
4                          ,new Zahl(42),new RPar(),new Div()
5                          ,new Zahl(42)};
6      Parser parser = new ExpressionParser(correct);
7      System.out.println(parser.parse());
8      Token [] inCorrect
9       = {new Zahl(2),new Mult(),new LPar()
10        ,new Zahl(42),new LPar()};
11     parser = new ExpressionParser(inCorrect);
12     System.out.println(parser.parse());
13   }
14 }
```

Aufgabe 21 Aufgrund der Struktur der Grammatik für den Parser `ExpressionTParser`, die die Operatorpräzedenz berücksichtigt, kommt es im Ableitungsbaum zu mehr oder weniger redundanten Knoten, die jeweils nur ein Kind haben. Diese Knoten tragen für die Struktur eines Ausdrucks für uns keinerlei interessanten Informationen.

Ändern Sie die Klasse `ExpressionTParser` so ab, daß der erzeugte Ableitungsbaum keine solchen Knoten mit nur einem Kind mehr enthält.

Aufgabe 22 Gegeben seien die folgende Schnittstelle und Klassen:

```
1• public interface Expression {
2    int eval();
3  }
```

```
1• public class AddExpr implements Expression{
2    Expression left;
3    Expression right;
4    public AddExpr(Expression left,Expression right){
5      this.left=left;
6      this.right=right;
7    }
```

```
8     public int eval(){
9       return left.eval()+right.eval();
10    }
11  }
```

```
1 • public class IntExpr implements Expression{
2     int i;
3     pubic  IntExpr(int i){this.i=i;}
4     public eval(){return i;}
5   }
```

a) Implementieren Sie entsprechende Klassen `SubExpr`, `MultExpr`, `DivExpr`.

b) Ändern Sie die Klasse `ExpressionTParser` so ab, daß statt des Ableitungsbaumes ein Objekt des Typs `Expression` erzeugt wird.

c) Schreiben Sie eine Methode `static public int calculate(Token [] expr)`, die einen Ausdruck mit dem Parser `ExpressionTParser` parst und durch Aufruf der Methode `eval` das Ergebnis berechnet.

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.

[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. `www.tfh-berlin.de/~grude/SkriptCompilerbau.pdf`, 2001. Skript zur Vorlesung, TFH Berlin.

[Gru01] Ulrich Grude. Einführung in gentle. `www.tfh-berlin.de/~grude/SkriptGentle.pdf`, 2001. Skript, TFH Berlin.

[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. `www.panitz.name/prog1/index.html`, 2003. Skript zur Vorlesung, TFH Berlin, 2. revidierte Auflage.

[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.