

PK11W

Skriptum zu

PROGRAMMKONSTRUKTION

185.A02 Grundlagen der Programmkonstruktion

183.592 Programmierpraxis

im Wintersemester 2011/2012

Inhaltsverzeichnis

1	Maschinen und Programme	11
1.1	Ein Java-Programm	12
1.1.1	Simulierte Objekte	12
1.1.2	Programmablauf	16
1.2	Binäre Digitale Systeme	20
1.2.1	Entstehung der Digitaltechnik	20
1.2.2	Binäre Systeme	22
1.2.3	Rechnen mit Binärzahlen	24
1.2.4	Logische Schaltungen	29
1.3	Maschinen und Architekturen	31
1.3.1	Architektur üblicher Computer	31
1.3.2	Abstrakte Maschinen und Modelle	34
1.3.3	Objekte als Maschinen	36
1.3.4	Softwarearchitekturen	38
1.4	Formale Sprachen, Übersetzer und Interpreter	40
1.4.1	Syntax, Semantik und Pragmatik	40
1.4.2	Bestandteile eines Programms	42
1.4.3	Compiler und Interpreter	44
1.4.4	Übersetzung und Ausführung	47
1.5	Denkweisen	50
1.5.1	Sprachen, Gedanken und Modelle	50
1.5.2	Der Lambda-Kalkül	53
1.5.3	Eigenschaften des Lambda-Kalküls	56
1.5.4	Zusicherungen und Korrektheit	58
1.6	Softwareentwicklung	61
1.6.1	Softwarelebenszyklus	61
1.6.2	Ablauf und Werkzeuge der Programmierung	63
1.6.3	Softwarequalität	65
1.6.4	Festlegung von Softwareeigenschaften	69
1.7	Programmieren lernen	70
1.7.1	Konzeption und Empfehlungen	70

1.7.2	Kontrollfragen	72
2	Grundlegende Sprachkonzepte	77
2.1	Die Basis	77
2.1.1	Vom Algorithmus zum Programm	77
2.1.2	Variablen und Zuweisungen	82
2.1.3	Datentypen	87
2.2	Ausdrücke und Operatoren	92
2.2.1	Allgemeines	92
2.2.2	Operatoren in Java	96
2.2.3	Typumwandlungen und Literale	106
2.3	Blöcke und bedingte Anweisungen	112
2.3.1	Blöcke	112
2.3.2	Selektion mit <code>if-else</code>	113
2.3.3	Mehrfach-Selektion mit der <code>switch</code> -Anweisung	116
2.4	Funktionen	117
2.4.1	Methoden: Aufbau der Methodendefinition anhand eines Beispiels	120
2.4.2	Parameter	121
2.4.3	Die <code>return</code> -Anweisung	122
2.4.4	Gleichnamige Methoden	122
2.4.5	Beispiel einer Aufrufsequenz	124
2.4.6	Rekursive Methoden	127
2.4.7	Zusicherungen	129
2.5	Iteration, Arrays, Strings	132
2.5.1	<code>while</code> und <code>do-while</code>	132
2.5.2	Array-Typen	134
2.5.3	Mehrdimensionale Arrays	137
2.5.4	<code>for</code>	139
2.5.5	Beispiel: Pascalsches Dreieck	140
2.6	Zustände	145
2.6.1	Seiteneffekte	145
2.6.2	Funktionen und Prozeduren in der Programmier- sprache Pascal	146
2.6.3	Methoden mit Seiteneffekt in Java	149
2.6.4	Funktionaler vs. prozeduraler Programmierstil	151
2.7	Kommunikation mit der Außenwelt	153
2.7.1	Die Methode <code>main</code>	153
2.7.2	Kontrollfragen	156

3	Objektorientierte Konzepte	161
3.1	Das Objekt	161
3.1.1	Abstrakte Sichtweisen	161
3.1.2	Faktorisierung: Prozeduren und Objekte	165
3.1.3	Datenabstraktion	167
3.2	Die Klasse und ihre Instanzen	171
3.2.1	Variablen und Methoden	171
3.2.2	Sichtbarkeit	175
3.2.3	Identität und Gleichheit	179
3.2.4	Kontext und Initialisierung	183
3.2.5	Konstanten	187
3.3	Interfaces und dynamisches Binden	189
3.3.1	Interfaces zur Schnittstellenbeschreibung	189
3.3.2	Dynamisches Binden	195
3.3.3	Spezialisierung und Ersetzbarkeit	200
3.4	Vererbung	203
3.4.1	Ableitung von Klassen	204
3.4.2	Klassen versus Interfaces	208
3.4.3	Von Object abwärts	211
3.5	Quellcode als Kommunikationsmedium	216
3.5.1	Namen, Kommentare und Zusicherungen	216
3.5.2	Faktorisierung, Zusammenhalt und Kopplung	221
3.5.3	Ersetzbarkeit und Verhalten	224
3.6	Objektorientiert programmieren lernen	226
3.6.1	Konzeption und Empfehlungen	227
3.6.2	Kontrollfragen	228
4	Daten, Algorithmen und Strategien	233
4.1	Begriffsbestimmungen	233
4.1.1	Algorithmus	233
4.1.2	Datenstruktur	235
4.1.3	Lösungsstrategie	238
4.2	Rekursive Datenstrukturen und Methoden	240
4.2.1	Verkettete Liste	240
4.2.2	Rekursion versus Iteration	244
4.2.3	Binärer Baum	248
4.3	Algorithmische Kosten	253
4.3.1	Abschätzung algorithmischer Kosten	255
4.3.2	Kosten im Zusammenhang	258

4.3.3	Zufall und Wahrscheinlichkeit	261
4.4	Teile und Herrsche	264
4.4.1	Das Prinzip	264
4.4.2	Pragmatische Sichtweise	267
4.4.3	Strukturelle Ähnlichkeiten	269
4.5	Abstraktion und Generizität	272
4.5.1	Generische Datenstrukturen	272
4.5.2	Gebundene Generizität	276
4.5.3	Abstraktionen über Datenstrukturen	279
4.5.4	Iteratoren	282
4.6	Typische Lösungsstrategien	288
4.6.1	Vorgefertigte Teile	288
4.6.2	Top Down versus Bottom Up	291
4.6.3	Schrittweise Verfeinerung	294
4.7	Strukturen programmieren lernen	296
4.7.1	Konzeption und Empfehlungen	296
4.7.2	Kontrollfragen	297
5	Qualitätssicherung	301
5.1	Spezifikationen	301
5.1.1	Anforderungsspezifikation und Anwendungsfälle	302
5.1.2	Design by Contract	304
5.1.3	Abstraktion und Intuition	307
5.2	Statisches Programmverständnis	311
5.2.1	Typen und Zusicherungen	311
5.2.2	Invarianten	314
5.2.3	Termination	317
5.2.4	Beweise und deren Grenzen	320
5.3	Testen	323
5.3.1	Auswirkungen auf Softwarequalität	323
5.3.2	Testmethoden	326
5.3.3	Laufzeitmessungen	329
5.4	Nachvollziehen des Programmablaufs	333
5.4.1	Stack Traces und Debug Output	333
5.4.2	Debugger	336
5.4.3	Eingrenzung von Fehlern	338
5.5	Ausnahmebehandlung	341
5.5.1	Abfangen von Ausnahmen	342
5.5.2	Umgang mit Ausnahmefällen	345

5.5.3	Aufräumen	350
5.6	Validierung	353
5.6.1	Validierung von Daten	354
5.6.2	Validierung von Programmen	356
5.7	Qualität sichern lernen	358
5.7.1	Konzeption und Empfehlungen	358
5.7.2	Kontrollfragen	359
6	Vorsicht: Fallen!	363
6.1	Beschränkte Ressourcen	363
6.1.1	Speicherverwaltung	363
6.1.2	Dateien und Co	367
6.1.3	Antwortzeiten	373
6.2	Grenzwerte	377
6.2.1	Umgang mit ganzen Zahlen	377
6.2.2	Rundungsfehler	381
6.2.3	Null	385
6.2.4	Off-by-one-Fehler und Pufferüberläufe	389
6.3	Nebenläufigkeit	393
6.3.1	Parallelität und Nebenläufigkeit	394
6.3.2	Race Conditions und Synchronisation	398
6.3.3	Gegenseitige Behinderung	402
6.4	Einfachheit und Flexibilität	404
6.4.1	Strukturierte Programmierung	405
6.4.2	Typische Fallen objektorientierter Sprachen	409
6.4.3	Spezielle Fallen in Java	412
6.5	Vertrauen und Kontrolle	416
6.5.1	Defensive und offensive Programmierung	416
6.5.2	Programmierstil und Vertrauen	419
6.5.3	Einheitliche Regeln	422
6.6	Mythen	423
6.6.1	Paradigmen und Mythen	425
6.6.2	Mythen in Java	430
6.7	Fallen umgehen lernen	434
6.7.1	Kontrollfragen	435

Vorwort

Das Wintersemester 2011/2012 bringt mit stark überarbeiteten Studienplänen zahlreiche Neuerungen in die Informatikstudien an der TU Wien. Auch die Programmierausbildung wurde ausgeweitet und neu gestaltet: Das Modul *Programmkonstruktion* umfasst zwei eng miteinander verzahnte Lehrveranstaltungen mit den Titeln *Grundlagen der Programmkonstruktion* und *Programmierpraxis*. Wie die Namen schon vermuten lassen, wird in den Grundlagen der Programmkonstruktion umfangreiches theoretisches Basis- und Hintergrundwissen im Bereich der Programmierung im weitesten Sinne vermittelt, während in der Programmierpraxis praktische Programmierfähigkeiten entwickelt werden. Das vorliegende Skriptum ist für Studierende beider Lehrveranstaltungen vorgesehen.

Es wird dringend empfohlen, die beiden Lehrveranstaltungen zusammen zu absolvieren. So wie man in der praktischen Programmierung nicht ohne einschlägiges Wissen auskommt, so ist theoretisches Wissen alleine ohne praktische Programmiererfahrung nicht viel wert. Dennoch gibt es einen Grund für die Trennung in zwei Lehrveranstaltungen: Das theoretische Wissen kann man sich, bei entsprechender Mitarbeit, sicherlich in der vorgesehenen Zeit von einem Semester aneignen. Die Entwicklung praktischer Fähigkeiten kann dagegen, je nach Veranlagung und Vorkenntnissen, länger dauern, auch wenn man sich viel Mühe gibt. Daher darf man für Programmierpraxis bis zu zwei Semestern brauchen. Ein positiver Abschluss von Grundlagen der Programmkonstruktion kann eine ausreichende Beschäftigung mit dem theoretischen Stoff signalisieren, auch wenn ein Abschluss von Programmierpraxis möglicherweise noch aussteht.

Neu ist auch die *Studieneingangs- und Orientierungsphase (STEOP)*. Das Modul Programmkonstruktion ist Teil der STEOP. Daher ist eine positive Absolvierung dieses Moduls (zumindest für Studierende, die im Wintersemester 2011/2012 erstmals für ein Informatik-Studium inskripiert sind) eine unbedingte Voraussetzung für die Teilnahme an Lehrveranstaltungen ab dem dritten Semester. Man kann Programmkonstruktion als Hürde sehen, die übersprungen werden muss, bevor man im Studium Zugang zu den komplexeren Themengebieten bekommt. Statt als Hürde sollte man die Programmkonstruktion als Chance sehen: Jetzt stehen für die Programmierausbildung mehr Ressourcen wie Zeit und Personal zur Verfügung, und das trotz restriktiver Sparmaßnahmen an den Universitäten. Es werden hohe Erwartungen in das Modul gesetzt. Einerseits werden von Studierenden gute Programmierkenntnisse erwartet, andererseits wird ih-

nen umfangreiche Unterstützung beim Lernen geboten. Das Ziel sind keine Knock-Out-Prüfungen, sondern ein hohes Niveau an Wissen und Können.

Die besten Leistungen lassen sich nur mit viel Einsatz und Enthusiasmus erzielen. Künftige Informatikerinnen und Informatiker bringen in der Regel viel Begeisterung für das Programmieren mit. Es ist ein gutes Gefühl, wenn man eigene Gedanken und Ideen in Software real werden lassen kann. Manchmal können uns Fehler in einem Programm oder Schwierigkeiten beim Verständnis komplexer Zusammenhänge jedoch auch fast zur Verzweiflung treiben. Das gehört genauso dazu wie die Freude darüber, wenn am Ende dennoch alles funktioniert. Programmieren ist keineswegs eine so emotionslose Angelegenheit, wie man gelegentlich suggeriert bekommt. Nur wer selbst Programme schreibt, kann die Begeisterung dafür verstehen. Wie so oft steht am Anfang ein manchmal schwieriger Lernprozess mit vielen kleinen Erfolgen, aber auch so manchem Rückschlag. Wer genug Durchhaltevermögen aufbringt um den Lernprozess zu meistern und sich von der Begeisterung anstecken lässt, wird beim Programmieren sicher viel Freude erleben. Das ist die eigentliche Triebfeder, die zu einem hohen Niveau an Wissen und Können führt.

Das vorliegende Skriptum ist ganz neu und zum Teil noch unfertig. Aus diesem Grund ist es in zwei Teilen erschienen. Das Skriptum ist absichtlich umfangreich angelegt um einen breiten Überblick über die Programmierung in Java geben zu können. Es soll für Studierende ohne vorherige Programmiererfahrung geeignet sein, aber auch jenen mit viel Programmiererfahrung neue Blickwinkel aufzeigen. Die Autoren bitten um Verständnis dafür, dass das Skriptum aufgrund zeitlicher Einschränkungen und der Zerteilung noch nicht die Qualität hat, die sie sich wünschen würden. Leider war es nicht möglich, das Skriptum so zu vervollständigen, dass der gesamte Stoff bis zum Ende abgedeckt wird – ein Kapitel sowie der Ausblick fehlen noch. Dennoch sind die Autoren zuversichtlich, dass das Skriptum auch so wie es ist schon eine gute Basis für das Modul Programmkonstruktion bietet.

Lassen Sie sich von der Begeisterung für das Programmieren anstecken! Viel Freude und Erfolg beim Programmieren lernen!

Franz Puntigam
Michael Reiter
Andreas Krall

1 Maschinen und Programme

Unter *Programmieren* versteht man heute meist die Tätigkeit des Erstellens von Computerprogrammen. Genau darum geht es im Modul *Programmkonstruktion*: Wir wollen lernen, wie man Computerprogramme schreibt, also einer Maschine die nötigen Anweisungen gibt, damit sie das macht, was wir von ihr wollen.

Auf den ersten Blick ist Programmieren tatsächlich so einfach: Es reicht, die wenigen grundlegenden Anweisungen einer Maschine zu kennen und auf geeignete Weise aneinanderzureihen. Kleine Programme werden wir bald verstehen und entwickeln können. Hier ist das erste Java-Programm:

Listing 1.1: Java-Programm zur Ausgabe von „Hello World!“

```
1 public class Hello {  
2     public static void main (String[] args) {  
3         System.out.println("Hello World!");  
4     }  
5 }
```

Dieses Programm (ohne die in zur besseren Lesbarkeit vorangestellten Zeilennummern) brauchen wir nur mehr in der Datei `Hello.java` speichern, mit dem Befehl „`javac Hello.java`“ übersetzen und mit dem Befehl „`java Hello`“ ausführen. Schon haben wir eine Maschine zum Leben erweckt und dazu gebracht, mit der Welt in Kontakt zu treten.

Zahlreiche Schwierigkeiten zeigen sich erst auf den zweiten Blick: Ein typisches Programm reiht viele Tausend oder Millionen Anweisungen aneinander, und jeder noch so kleine Fehler kann schwerwiegende Folgen nach sich ziehen. Es ist nicht leicht, den Überblick über ein riesiges Netzwerk an Anweisungen zu bewahren oder herauszufinden, welchen Zweck eine einzelne Anweisung darin hat. Zudem ändern sich die Wünsche an das Programm im Laufe der Zeit ebenso wie die Maschinen, auf denen das Programm laufen soll. Auch die Personen, die das Programm erstellen und weiterentwickeln, werden gelegentlich ausgetauscht. Nur mit viel Wissen, geschultem abstraktem Denkvermögen und durchdachten Vorgehensweisen ist es möglich, die große Komplexität zu meistern.

Im Modul *Programmkonstruktion* geht es daher auch darum: Wir wollen

- wichtige Zusammenhänge zwischen Maschinen, Programmen, den in die Entwicklung und Anwendung von Programmen involvierten Personen und ihrem Umfeld betrachten,
- grundlegende Konzepte und Aussagen der Informatik im Bereich der Programmierung einführen und deren Auswirkungen auf konkrete Programmiersprachen und Programmierstile veranschaulichen,
- Techniken, Werkzeuge und Vorgehensweisen kennenlernen, die uns bei der Erstellung hochwertiger Programme unterstützen,
- durch das Lösen zahlreicher Programmieraufgaben das nötige handwerkliche Geschick und abstrakte Denkvermögen weiterentwickeln.

1.1 Ein Java-Programm

Zur Einstimmung betrachten wir ein kleines Programm in Java und typische Merkmale davon. Es ist noch nicht nötig, alle Details zu verstehen. Vorerst wollen wir nur ein Gespür für das Aussehen von Programmen sowie die Terminologie und die Konzepte der Programmierung bekommen. Erst später werden wir lernen, solche Programme selbst zu entwickeln.

1.1.1 Simulierte Objekte

Java ist eine *objektorientierte Programmiersprache*, das heißt, *Software-Objekte* oder kurz *Objekte* stehen im Mittelpunkt der Programmierung. Ein Software-Objekt *simuliert* ein Objekt aus der „realen Welt“. Alle Arten von Objekten können simuliert werden, gegenständliche Objekte wie Bäume, Autos und Häuser, aber auch abstrakte Objekte wie Zahlen und Mengen, also alles, was tatsächlich oder in unserer Vorstellung existiert.

Objekte werden in Java nicht direkt, sondern durch *Klassen* dargestellt. Eine Klasse beschreibt gleichartige Objekte. Beispielsweise beschreibt eine Klasse `Baum` die Bäume einer Allee, und jeder einzelne Alleebaum gehört zur Klasse `Baum`. Statt „ein Objekt gehört zur Klasse“ sagt man eher „ein Objekt ist *Instanz* der Klasse“, also ein bestimmter Alleebaum ist eine Instanz von `Baum`. In Java ist jedes Objekt Instanz irgendeiner Klasse.

Listing 1.2 zeigt den Programmcode der Klasse `UnbekannteZahl`. Jede Instanz davon stellt eine außerhalb dieses Objekts unbekannte Zahl dar,

Listing 1.2: Kapsel, die eine Zahl und Vergleichsmethoden enthält

```

1 import java.util.Random;
2
3 public class UnbekannteZahl {
4     // die Zahl ist nur innerhalb der Klasse bekannt
5     private int zahl;
6
7     // Initialisierung mit Zufallszahl zwischen 0 und (grenze - 1)
8     // Voraussetzung: grenze > 0
9     public UnbekannteZahl (int grenze) {
10         zahl = (new Random()).nextInt() % grenze;
11         if (zahl < 0) {
12             zahl = zahl + grenze;
13         }
14     }
15
16     // Ergebnis von "gleich(n)" ist true wenn zahl gleich n ist
17     public boolean gleich (int vergleichszahl) {
18         return (zahl == vergleichszahl);
19     }
20
21     // Ergebnis von "kleiner(n)" ist true wenn zahl kleiner n ist
22     public boolean kleiner (int vergleichszahl) {
23         return (zahl < vergleichszahl);
24     }
25 }

```

die aber innerhalb des Objekts sehr wohl bekannt ist. Man unterscheidet die Innen- von der Außenansicht. Vor einem Betrachter von außen sind viele Details versteckt. Im Programmcode findet man die Wörter `public` und `private`. Alles was `private` ist, kann man nur innerhalb der Klasse sehen, während alles was `public` ist, überall im Programm bekannt ist. Im *Rumpf* der überall sichtbaren Klasse, das ist alles was innerhalb der geschweiften Klammern nach `class UnbekannteZahl` steht, werden folgende Programmteile deklariert bzw. definiert:

- in Zeile 5 eine nur in der Klasse sichtbare *Variable* namens `zahl`,
- in den Zeilen 9 bis 14 ein überall sichtbarer *Konstruktor* mit demselben Namen wie die Klasse, also `UnbekannteZahl`,
- in den Zeilen 17 bis 19 und 22 bis 24 zwei überall sichtbare *Methoden* namens `gleich` und `kleiner`.

Jeder Deklaration und Definition in dieser Klasse ist ein *Kommentar* vorangestellt, das ist beliebiger Text zwischen „/ /“ und dem Ende einer Zeile. Kommentare helfen Menschen dabei, das Programm zu verstehen, werden aber vom Computer nicht beachtet.

Die Deklaration in Zeile 5 legt die *Sichtbarkeit* `private`, den *Typ* `int` und den *Namen* `zahl` einer Variablen fest. Eine Variable steht für einen im Laufe der Zeit änderbaren (= variablen) Wert. Man sagt auch, sie *enthält* oder *hat* einen Wert. Beispielsweise hat die Variable zu einem Zeitpunkt den Wert 87 und zu einem anderen Zeitpunkt den Wert 3. Der Wert selbst ist in der Deklaration nicht festgelegt; genau deswegen spricht man nur von einer *Deklaration* und keiner Definition. Der Typ legt die Art der Werte fest, welche die Variable haben kann. Die Variable `zahl` hat den Typ `int` (eine Abkürzung für das englischsprachige Wort *integer*), der den ganzen Zahlen entspricht. Ein anderer möglicher Typ wäre `boolean`, der den Boole'schen bzw. logischen Werten `true` (wahr) und `false` (falsch) entspricht. Aber auch der Name einer Klasse wie `UnbekannteZahl` kann als Typ verwendet werden, sodass eine mit diesem Typ deklarierte Variable Instanzen dieser Klasse enthalten kann. Generell spricht man von den *Instanzen eines Typs* wenn man alle Werte meint, die von diesem Typ sind. Ganze Zahlen sind demnach Instanzen von `int`, und `true` und `false` sind Instanzen von `boolean`.

Die Methoden `gleich` und `kleiner` werden *definiert*, nicht nur deklariert, das heißt, es werden alle Details festgelegt, und diese bleiben im Laufe der Zeit gleich. Die Methoden eines Objekts erlauben anderen Objekten, mit diesem Objekt in Kontakt zu treten. Man tritt mit einem Objekt in Kontakt, indem man ihm eine *Nachricht* schickt, und das Objekt beantwortet die Nachricht. Die Klasse beschreibt über Methoden die Nachrichten, welche die Instanzen der Klasse verstehen, sowie das *Objektverhalten*, also das, was die Objekte machen, wenn sie solche Nachrichten empfangen. Obwohl der Wert der Variablen `zahl` in anderen Klassen nicht sichtbar ist, kann über die überall sichtbaren Methoden `gleich` und `kleiner` etwas über diesen Wert herausgefunden werden. Sowohl `gleich` als auch `kleiner` liefert ein Ergebnis vom Typ `boolean` zurück, das heißt, solche Nachrichten an eine Instanz von `UnbekannteZahl` werden mit `true` oder `false` beantwortet. Eine Nachricht `gleich(3)` fragt an, ob die unbekannte Zahl gleich drei ist, und `kleiner(3)` ob die unbekannte Zahl kleiner drei ist. Das funktioniert auch mit anderen Zahlen. Werte wie die Zahl 3, die als Teil einer Nachricht an ein Objekt geschickt werden, nennt man *Argumente* oder *aktuelle Parameter* der Nachricht. Neben

der Sichtbarkeit, dem Ergebnistyp und dem Namen enthält jede Definition einer Methode eine Liste *formaler Parameter-Deklarationen* oder kurz *Parameter-Deklarationen* in runden Klammern und einen *Rumpf* in geschweiften Klammern. Die Deklaration eines *formalen Parameters* ähnelt der einer Variablen, jedoch fehlt die Sichtbarkeit. Eine Methode wird in einem Objekt *ausgeführt*, wenn das Objekt eine gleichnamige Nachricht empfängt. Während dieser Ausführung enthalten die formalen Parameter die Werte, die der Nachricht als Argumente mitgegeben wurden. Anzahl und Typen der Argumente müssen der Anzahl und den Typen der formalen Parameter entsprechen. Bei der Ausführung einer Methode werden die *Anweisungen* im Rumpf der Methode ausgeführt, eine nach der anderen von oben nach unten und von links nach rechts. Die Rümpfe der Methoden `gleich` und `kleiner` bestehen nur aus je einer *return*-Anweisung, die eine Antwort an den Sender einer Nachricht zurückgibt. Der zurückgegebene Boole'sche Wert ergibt sich aus der Anwendung eines Vergleichsoperators (`==` bzw. `<`) auf die Werte der Variablen `zahl` und des formalen Parameters `vergleichszahl`, wobei `==` dann `true` liefert wenn die verglichenen Werte identisch sind, und `<` wenn der linke Operand eine kleinere Zahl ist als der rechte.

Konstruktoren ähneln Methoden, haben aber keine Ergebnistypen, und der Name ist immer gleich dem Klassennamen. Konstruktoren werden benötigt, um mit `new` erzeugte Objekte zu *initialisieren*, wobei die Variablen des Objekts ihre ersten Werte bekommen. Man sagt auch, die Variablen des Objekts werden initialisiert. Der formale Parameter `grenze` des Konstruktors von `UnbekannteZahl` legt fest, in welchem Bereich die Zahl einer neuen Instanz liegen soll – größer oder gleich 0 und kleiner `grenze`. Entsprechend dem Kommentar soll der Wert von `grenze` größer 0 sein. Im Rumpf des Konstruktors stehen zwei Anweisungen. Die erste weist der Variablen `zahl` links vom Zuweisungsoperator `=` den Wert zu, der im Ausdruck rechts von `=` berechnet wird. Dabei wird mittels `new Random()` eine neue Instanz der Klasse `Random` erzeugt und über den parameterlosen Konstruktor `Random()` initialisiert. Diese Klasse ist in der *Bibliothek* `java.util` definiert und wird von dort mittels der Anweisung in der ersten Zeile in das Programm *importiert*, also zugreifbar gemacht. Das neue Objekt ist ein Zufallszahlengenerator. Wir schicken ihm die parameterlose Nachricht `nextInt()` und bekommen eine zufällig gewählte Zahl zurück, deren Wertebereich aber noch nicht so eingeschränkt ist, wie wir ihn haben wollen. Daher wenden wir den Operator `%` auf diese Zahl und den Wert von `grenze` an, der den Rest der Division der Zufallszahl

durch `grenze` berechnet. Wenn die Zufallszahl negativ war, ist auch der Divisionsrest negativ. Der an `zahl` zugewiesene Wert ist somit größer als $-\text{grenze}$ und kleiner als `grenze`. Die zweite Anweisung, eine *bedingte Anweisung*, sorgt dafür, dass der Wert von `zahl` nach Beendigung der Ausführung des Konstruktors nicht kleiner 0 sein kann. Nur wenn der aktuelle Wert von `zahl` kleiner 0 ist, das ist die *Bedingung* in der bedingten Anweisung, wird die Anweisung `zahl = zahl + grenze` im Rumpf der bedingten Anweisung ausgeführt. Dabei wird ein neuer Wert an `zahl` zugewiesen, der um den Wert von `grenze` größer ist als der alte Wert. Wie gewünscht ist der Wert von `zahl` danach sicher größer oder gleich 0 und kleiner `grenze`.

Die Variable `zahl` bildet zusammen mit dem Konstruktor und den Methoden von `UnbekannteZahl` eine untrennbare Einheit. Erst durch die Methoden wird die Variable sinnvoll verwendbar, und die Methoden brauchen die Variable, um ihre Aufgaben zu erfüllen. Variablen und Methoden sind quasi in eine gemeinsame Kapsel eingeschlossen. Die Eigenschaft eines Objekts, Variablen und Methoden zu einer Einheit zusammenzuführen, nennt man *Datenkapselung*. Zusammen mit *data hiding*, dem Verstecken von Details durch `private`, spricht man von *Datenabstraktion*. Wir können Objekte als abstrakte Einheiten betrachten, ohne wissen zu müssen, wie sie genau funktionieren. Auf solche abstrakte Weise haben wir eine neue Instanz von `Random` verwendet: Über eine Nachricht erhalten wir eine Zufallszahl, ohne zu wissen, wie sie ermittelt wird. Wir werden sehen: Abstraktion spielt in der Informatik eine sehr große Rolle.

1.1.2 Programmablauf

Listing 1.3 zeigt den Programmcode der Klasse `Zahlenraten`, die eine unbekannte Zahl in einem kleinen Spiel verwendet. Diese Klasse ist nicht dafür gedacht, dass Instanzen von ihr erzeugt werden. Daher enthält der Rumpf weder Variablen-Deklarationen noch Konstruktor-Definitionen. Stattdessen enthält die Klasse eine spezielle Methode namens `main`, in deren Definition das Wort `static` klarstellt, dass die Methode zur Klasse selbst, aber nicht zu Instanzen der Klasse gehört. Aufgrund des Resultstyps `void` darf man auf eine entsprechende Nachricht keine Antwort erwarten. Die Methode `main`, die bis auf den Rumpf in jedem Programm so aussehen muss wie hier, definiert den Startpunkt des Programms. Im Allgemeinen enthält der formale Parameter `args` Argumente, die einem

Listing 1.3: Spiel zum Erraten der Zahl in einer Instanz von `UnbekannteZahl`

```

1 import java.util.Scanner;
2
3 public class Zahlenraten {
4     public static void main (String[] args) {
5         UnbekannteZahl zuErraten = new UnbekannteZahl(100);
6         Scanner sc = new Scanner(System.in);
7         System.out.println("Errate eine Zahl zwischen 0 und 99:");
8         while (sc.hasNext()) {
9             if (sc.hasNextInt()) {
10                int versuch = sc.nextInt();
11                if (versuch < 0 || versuch > 99) {
12                    System.out.println("Nur Zahlen von 0 bis 99!");
13                }
14                else if (zuErraten.gleich(versuch)) {
15                    System.out.println("Gratulation! Zahl erraten!");
16                    return;
17                }
18                else if (zuErraten.kleiner(versuch)) {
19                    System.out.println("Gesuchte Zahl ist kleiner.");
20                }
21                else {
22                    System.out.println("Gesuchte Zahl ist größer.");
23                }
24            }
25            else {
26                sc.next();
27                System.out.println("Das ist keine erlaubte Zahl!");
28            }
29            System.out.println("Neuer Versuch (Ende mit ^D):");
30        }
31        System.out.println("Zahl leider nicht erraten :-(");
32    }
33 }

```

Programmaufruf mitgegeben werden können. In diesem Programm brauchen wir keine solchen Argumente. Trotzdem muss der formale Parameter vorhanden sein, weil die Sprachdefinition von Java das so vorsieht. Es ist klar, warum wir eine spezielle Methode brauchen: Wenn ein Programm gestartet wird, gibt es noch keine Objekte, sondern nur Klassen. Wir können zu Beginn also keinem Objekt eine Nachricht schicken, um die Ausführung einer Methode zu starten. Diese spezielle Methode wird ausgeführt, ohne vorher eine Nachricht an ein Objekt schicken zu müssen.

Variablen können auch im Rumpf einer Methode deklariert werden. Solche *lokalen Variablen* existieren nur während der Ausführung der Methode und sind nur in dem Bereich (innerhalb der geschweiften Klammern) sichtbar, in dem sie deklariert sind; daher brauchen wir auch keine Sichtbarkeit angeben. Die Variable `zuErraten` wird gleich in der Deklaration in Zeile 5 mit einer neuen Instanz von `UnbekannteZahl` im Wertebereich von 0 bis 99 initialisiert. In Zeile 6 wird eine lokale Variable `sc` vom Typ `Scanner`, einer aus `java.util` importierten Klasse, deklariert und mit einer neuen Instanz initialisiert. Instanzen von `Scanner` unterstützen die Umwandlung von Eingaben über Tastatur oder aus einer Datei in ein Datenformat, das wir innerhalb des Programms benötigen. Dem Konstruktor von `Scanner` übergeben wir als Argument den Wert der Variablen `System.in`. Diese von unserem System vorgegebene Variable steht für die *Standard-Eingabe*, die normalerweise mit einem *Terminal* verbunden ist, also mit den über die Tastatur erfolgten Eingaben in ein Fenster am Bildschirm. Unsere Instanz von `Scanner` erlaubt uns also das Einlesen über die Tastatur eingegebener Daten in das Programm. In Zeile 7 verwenden wir mit `System.out` eine weitere vom System vorgegebene Variable, die für die *Standard-Ausgabe* steht und normalerweise mit demselben Terminal verbunden ist. An den Wert dieser Variable schicken wir die Nachricht `println` (Abkürzung für *print line*) um die Textzeile im Argument am Bildschirm auszugeben. Die Textzeile ist vom Typ `String`, das ist eine *Zeichenkette*, also eine Folge beliebiger Zeichen eingeschlossen zwischen " am Anfang und Ende. Der ausgegebene Text fordert BenutzerInnen des Programms auf, eine Zahl einzutippen.

Eine *while-Schleife* bildet den Hauptteil der Methode (Zeilen 8 bis 30). Der Rumpf der Schleife wird so lange *iteriert*, also wiederholt ausgeführt, solange die *Schleifenbedingung* `sc.hasNext()` erfüllt ist, das heißt, sooft hintereinander wie zu Beginn jeder *Iteration* (= *Schleifendurchlauf*) auf die Nachricht `hasNext()` an `sc` die Antwort `true` zurückgeliefert wird. Die Ausführung der Methode `hasNext` wird so lange warten, bis eine nicht-leere Eingabe vorhanden ist, das heißt im Wesentlichen, bis jemand über die Tastatur etwas anderes als nur Leerzeichen in das Terminal eingegeben und die Eingabe mit „Enter“ abgeschlossen hat, oder die Standard-Eingabe (beispielsweise durch Eingabe von „Control-d“) geschlossen wird. Die Antwort ist `true` wenn eine nicht-leere Eingabe vorhanden ist und `false` wenn die Standard-Eingabe geschlossen wurde. Im ersten Fall wird der Rumpf der Schleife ausgeführt – siehe unten. Im zweiten Fall wird der Schleifenrumpf nicht mehr durchlaufen, und die erste Anweisung nach der

Schleife in Zeile 31 informiert BenutzerInnen durch eine Textausgabe darüber, dass die gesuchte Zahl nicht gefunden wurde. Die Ausführung von `main` (und damit des ganzen Programms) endet daraufhin, da es keine weiteren Anweisungen mehr gibt.

Der Schleifenrumpf beginnt in Zeile 9 mit einer bedingten Anweisung. Diese hat zwei Rümpfe, die wir *Zweige* nennen. Der erste Zweig ab Zeile 10 wird nur ausgeführt wenn die Bedingung wahr ist, der zweite Zweig ab Zeile 26, *else-Zweig* genannt, nur wenn die Bedingung falsch ist. Auf die Nachricht `hasNextInt()` an `sc` in der Bedingung bekommen wir die Antwort `true`, wenn die nächste Eingabe – von der wir schon wissen, dass sie existiert – eine Zahl ist. Falls sie keine Zahl ist, kommen wir in den *else-Zweig*. Dort wird durch die Nachricht `next()` an `sc` die nächste Eingabe gelesen und gelöscht, damit danach nur mehr die darauf folgenden Eingaben gesehen werden. Die Antwort auf diese Nachricht ignorieren wir, da wir damit nichts anfangen können. Stattdessen informieren wir BenutzerInnen über die falsche Eingabe. Falls die Eingabe eine Zahl ist, kommen wir in den ersten Zweig, wo die lokale Variable `versuch` mit der Zahl aus der Eingabe initialisiert wird. Auch die Nachricht `nextInt()` an `sc`¹ löscht die Eingabe, damit danach die darauf folgenden Eingaben sichtbar sind. In diesem Zweig machen wir mehrere *Fallunterscheidungen*, das heißt, abhängig von verschiedenen Bedingungen wird einer von mehreren möglichen Programmzweigen ausgeführt. Falls die eingelesene Zahl nicht im erlaubten Wertebereich liegt, also `versuch` einen Wert kleiner 0 oder größer 99 hat (`||` steht für „oder“), informieren wir BenutzerInnen darüber. Andernfalls, falls wir die gesuchte Zahl gefunden haben, informieren wir BenutzerInnen über diese Tatsache und brechen das Programm ab. Normalerweise bricht die `return`-Anweisung nur die Ausführung der Methode ab und bestimmt die Antwort der Methode, aber `main` liefert (wegen `void`) keine Antwort zurück und ist die spezielle Methode, mit der die Programmausführung beginnt und auch endet. Wir haben noch zwei mögliche Fälle: In einem Fall ist die zu erratende Zahl kleiner und im anderen größer als die eingegebene Zahl. BenutzerInnen werden jeweils darüber informiert. Das Programm wird mit der nächsten Anweisung nach

¹Sowohl in `Random` (siehe Klasse `UnbekannteZahl`) als auch in `Scanner` gibt es eine Methode namens `nextInt`. Diese beiden Methoden haben aber nichts miteinander zu tun. Gleiche Nachrichten an Objekte unterschiedlicher Typen können ganz unterschiedliche Effekte haben. Wir können anhand der Typen von Variablen leicht unterscheiden, ob die Nachricht `nextInt()` an eine Instanz von `Random` oder `Scanner` (oder einer anderen Klasse) gerichtet ist.

der bedingten Anweisung fortgesetzt. Egal welche Zweige der bedingten Anweisungen ausgeführt wurden (abgesehen von dem einen Fall, der zum Programmabbruch führt), landen wir immer bei der letzten Anweisung im Schleifenrumpf. Hier werden BenutzerInnen durch eine Textausgabe aufgefordert, einen neuen Versuch zu starten. Danach wird die Ausführung mit der nächsten Überprüfung der Schleifenbedingung und gegebenenfalls der nächsten Iteration fortgesetzt.

Um das Programm auszuführen, erstellen wir zunächst die Textdateien `UnbekannteZahl.java` und `Zahlenraten.java`, welche den Code in den Listings 1.2 und 1.3 (ohne Zeilennummern) enthalten. Diesen *Quellcode* müssen wir in einen *Zwischencode* übersetzen, der dann durch einen *Interpreter* oder eine *virtuelle Maschine* ausgeführt wird. Dazu brauchen wir ein Terminal-Fenster, über das wir Befehle an den Computer eingeben. Der Befehl „`javac UnbekannteZahl.java Zahlenraten.java`“ ruft den *Compiler* auf, der aus den Quellcodedateien die Zwischencode-dateien `UnbekannteZahl.class` und `Zahlenraten.class` erzeugt. Den Interpreter und das Spiel starten wir durch „`java Zahlenraten`“.

Vor allem Leser ohne oder mit nur wenig Programmiererfahrung sollten das Spiel tatsächlich ausprobieren und seine Funktionsweise zu verstehen versuchen. Manche Zusammenhänge werden klarer, wenn man kleine Änderungen vornimmt und schaut, wie sich diese auswirken. Falls das Programm nicht in allen Details verständlich ist, so ist das jetzt noch kein Problem. Wir müssen wichtige Grundlagen und Konzepte der Programmierung erst schrittweise einführen. Dabei ist es aber hilfreich, wenn man zumindest schon einmal versucht hat, ein Programm zu verstehen.

1.2 Binäre Digitale Systeme

Die Digitaltechnik bildet eine wesentliche Grundlage heutiger Computer und vieler anderer technischer Systeme. Digitale Systeme basieren überwiegend auf dem binären Zahlensystem, verwenden also die Zahl Zwei als Basis, nicht die Zahl Zehn wie im gewohnten dezimalen Zahlensystem.

1.2.1 Entstehung der Digitaltechnik

Programmierbare Maschinen sind schon seit langem bekannt. Beispielsweise stellen Falcons und Jacquards Erfindungen von über Lochkarten gesteuerten Webstühlen Mitte des 18. Jahrhundert einen Meilenstein der

Industrialisierung dar. Jede Lochkarte aus Holz oder Karton bestimmt durch das Vorhanden- oder Nichtvorhandensein eines Loches an einer bestimmten Stelle die Stellung eines Fadens – oberhalb oder unterhalb eines kreuzenden Fadens. Viele zu einer endlosen Schleife zusammengeheftete und nacheinander verwendete Lochkarten, je eine Karte pro kreuzendem Faden, erzeugen ein Muster im gewebten Stoff. Durch Austausch der Lochkarten kann das Muster leicht geändert werden, ohne den Mechanismus des Webstuhls modifizieren zu müssen. Moderne Webstühle arbeiten noch immer nach demselben Prinzip.

Diese Webstühle nutzten wahrscheinlich zum ersten Mal die Digitaltechnik in industriellem Maßstab. Dass Webmuster in *digitaler* Form – also als diskrete Werte im Gegensatz zu *analogen* bzw. kontinuierlichen Daten – aufgezeichnet wurden, liegt an der Verwendung der Daten: Es ist nur entscheidend, ob ein Faden oberhalb oder unterhalb eines ihn kreuzenden Fadens liegt, wie weit ober oder unter ihm ist egal. Die nächste Lochkarte kommt erst zum Einsatz, wenn ein Faden vollständig durchgefädelt ist. Anders verhält es sich mit Drehorgeln und Spieluhren, wo die Daten, ähnlich wie beim Webstuhl, in Form von Löchern oder Stiften auf Kartonbändern, Blechtrommeln und vergleichbaren Datenträgern vorliegen: Die Längen von sowie Abstände zwischen Löchern oder Stiften sind von Bedeutung. Damit sind Pausen und die Längen der Töne oder Stärken der Anschläge auf analoge, nicht digitale Weise bestimmt.

Seit Ende des 19. Jahrhunderts wird die Lochkartentechnik auch verbreitet für die Steuerung damals noch rein mechanischer Rechenmaschinen und Registrierkassen eingesetzt. Bekannt wurde die Technik vor allem durch die US-amerikanische Volkszählung 1890, die durch den Einsatz eines Systems von Lochkarten, Stanzmaschinen und Zählmaschinen in nur einem Jahr abgeschlossen werden konnte. Jede Person wurde durch eine Lochkarte aus Karton identifiziert, die personenbezogene Daten durch Löcher an bestimmten Stelle festhielt. Solche Lochkarten wurden weltweit bis etwa 1980 in der Verwaltung eingesetzt, dann aber durch zuverlässigere magnetische Datenträger (z.B. Magnetstreifenkarten) und in jüngster Zeit mehr oder weniger intelligente Speicherchips (z.B. Chipkarten) ersetzt.

Ein Trend ist trotz aller technologischen Änderungen seit Beginn der automationsunterstützten Datenverarbeitung gleich geblieben: Daten werden zunehmend in digitaler Form aufgezeichnet und fast ausschließlich in digitaler Form weiterverarbeitet. Wenn Daten auf natürliche Weise in analoger Form vorliegen, werden sie durch digitale Daten angenähert. Beispielsweise wird die Körpergröße nur auf ganze Zentimeter genau angege-

ben, ein Fingerabdruck nur durch dessen wichtigste Merkmale auf normierte Weise beschrieben, und eine Schallwelle durch eine Folge von in gleichmäßig kurzen Abständen abgetasteten ungefähren Amplitudenwerten dargestellt. Dabei entstehen immer Digitalisierungsfehler (beispielsweise nur eine Körpergröße von 180 cm angegeben, obwohl tatsächlich 180,3 cm gemessen wurden), die aber in Kauf genommen werden. Der Vorteil der Digitalisierung liegt auf der Hand: Digitale Daten können ohne weiteren Qualitätsverlust leicht gespeichert und verarbeitet werden, gleichgültig ob auf Lochkarten, magnetischen Speichermedien oder Speicherchips. Bei der Speicherung und Verarbeitung analoger Daten tritt dagegen durch Einflüsse von außen (Temperatur, Feuchtigkeit, Magnetfelder, Stöße, etc.) praktisch immer ein Qualitätsverlust auf, das heißt, die Daten werden unkontrollierbar verändert. Außerdem kann der Digitalisierungsfehler immer so klein wie nötig gehalten werden. Man könnte die Körpergröße auch in Millimeter statt Zentimeter angeben, sodass der Messfehler vermutlich größer ist als der Digitalisierungsfehler.

Vor gar nicht so langer Zeit übliche analoge Rechenschieber findet man nur mehr im Museum. Viele Taschenrechner sind solchen Präzisionsinstrumenten preislich und leistungsmäßig klar überlegen. Auch Bild- und Tonmaterial wird fast nur mehr digital aufgezeichnet. Enthusiasten schwören zwar immer noch auf die unvergleichliche Qualität alter Schallplatten ohne Digitalisierungsrauschen, aber kaum jemand kann sich einen Plattenspieler leisten, bei dem der Qualitätsverlust durch die analoge Verarbeitung nicht deutlich größer ist als der durch kleine Digitalisierungsfehler.

1.2.2 Binäre Systeme

Auch ein weiteres Prinzip ist seit dem ersten programmierbaren Webstuhl unverändert geblieben: Daten werden überwiegend in *binärer* Form dargestellt. Dabei wird nur zwischen zwei möglichen Werten unterschieden, die für 0 und 1, Ja und Nein, Wahr und Falsch, Ein- und Ausgeschaltet, Faden kreuzt einen anderen unterhalb oder oberhalb (Webstuhl), etc. stehen. Speichermedien, Datenübertragungsleitungen und Maschinen verwenden verschiedene physikalische Ausprägungen zur Unterscheidung dieser Werte – Löcher, Änderungen der Magnetisierungsrichtung, elektrische Spannungen oder Ströme über einem Schwellenwert, Breite von hellen und dunklen Streifen in Barcodes, etc. Wie die Werte unterschieden werden, spielt keine Rolle. Wichtig sind hingegen einige vereinfachende mathematische Eigenschaften binärer Systeme. Vor allem ist es einfach, mit Wahr-

0000 = 0 (0)	0100 = 4 (4)	1000 = 8 (-8)	1100 = 12 (-4)
0001 = 1 (1)	0101 = 5 (5)	1001 = 9 (-7)	1101 = 13 (-3)
0010 = 2 (2)	0110 = 6 (6)	1010 = 10 (-6)	1110 = 14 (-2)
0011 = 3 (3)	0111 = 7 (7)	1011 = 11 (-5)	1111 = 15 (-1)

Abbildung 1.4: Binärdarstellung der Zahlen 0...15 (bzw. -8...7)

heitswerten (Wahr und Falsch) umzugehen, und darauf aufbauende binäre Logik-Systeme werden intensiv genutzt. Mit Binärzahlen können Maschinen einfacher rechnen als mit Dezimalzahlen oder in anderen Zahlensystemen. Man könnte statt auf binäre Systeme beispielsweise auf dreiwertige Systeme aufbauen und zwischen Null, Eins und Zwei oder Wahr, Falsch und Unbekannt, etc. unterscheiden. Davon macht man kaum Gebrauch, da man die mathematische Einfachheit binärer Systeme verlieren würde.

Eine einstellige binäre Zahl (also 0 oder 1) nennt man *Bit*. Mit einem Bit alleine fängt man nicht viel an, mit mehreren Bits zusammen aber schon. Man verwendet eine fixe Anzahl von Bits als grundlegende Einheit für die Darstellung von Daten, so wie jede Lochkarte eine fixe Anzahl von Bits enthält. Ein solches *Wort* (auch *Maschinenwort* oder *Binärwort* genannt) enthält in modernen Computern häufig 32 oder 64 Bits, aber es gibt auch kleinere mit nur 4, 8 oder 16 Bits und größere mit 256, 512 oder 1024 Bits. Obwohl Worte (oder Wörter) mit 2^k Bits bevorzugt werden, kommen auch solche mit beispielsweise 12, 24 oder 28 Bits vor.

In n Bits lassen sich 2^n unterschiedliche Werte ausdrücken, beispielsweise 16 Werte in 4 Bits. Welchem Wert ein bestimmtes Bitmuster entspricht, ist jedoch nicht fix vorgegeben, sondern hängt von der *Codierung* der Werte ab. Abbildung 1.4 zeigt zwei übliche Codierungen für Zahlen in 4 Bits, wobei eine nur nicht-negative Zahlen und die andere (in Klammern, *Zweierkomplement* genannt) negative wie positive Zahlen darstellen kann. In Java bestimmt der Typ die Codierung. So legt der Typ `int` fest, dass Werte als Zweierkomplement mit 32 Bit codiert werden. Dabei bestimmt das am weitesten links stehende Bit das Vorzeichen, und einen Wert negiert man, indem man ihn um eins verkleinert und dann alle Bits umkehrt (also das Komplement bildet). Beispielsweise erhält man die Binärdarstellung 1011 von -5 als Komplement der Binärdarstellung 0100 von 4.

Das ganz links stehende Bit eines Wortes ist das *Most Significant Bit (MSB)*, das ganz rechts stehende das *Least Significant Bit (LSB)*. Diese

1 Kilobyte (kB oder KB)	= 2^{10} Byte =	1.024 Byte $\approx 10^3$ Byte
1 Megabyte (MB)	= 2^{20} Byte =	1.048.576 Byte $\approx 10^6$ Byte
1 Gigabyte (GB)	= 2^{30} Byte =	1.073.741.824 Byte $\approx 10^9$ Byte
1 Terabyte (TB)	= 2^{40} Byte =	1.099.511.627.776 Byte $\approx 10^{12}$ Byte

Abbildung 1.5: Am häufigsten verwendete Größeneinheiten für Daten

Begriffe sind auch dann klar, wenn „links“ und „rechts“ von der Blickrichtung abhängen, beispielsweise auf Lochkarten oder Schaltplänen.

Aus historischen Gründen nennt man je 8 nebeneinander liegende Bits in einem Wort ein *Byte*. Ein Wort mit 32 Bits enthält also 4 Bytes. Früher wurde jedes Zeichen (Buchstabe, Ziffer, Sonderzeichen, etc.) in einem Byte dargestellt. Da sich die vielen länderspezifischen Zeichen in den $2^8 = 256$ möglichen Werten eines Bytes nicht ausgeben, stellt man seltener benutzte Zeichen in mehreren Bytes dar. Manchmal nimmt man auch 16- oder 32-Bit-Worte. Dessen ungeachtet ist es üblich, die Größe von Daten, die in einem Block aneinandergereihten Worte abgelegt sind, in Bytes anzugeben. Für Texte entspricht diese Angabe grob genähert der Anzahl der Zeichen.

Byte ist eine praktische Größenangabe für kleine Datenmengen. Für große verwenden wir eher Kilobyte, Megabyte, Gigabyte und so weiter, siehe Abbildung 1.5. Da diese Einheiten hauptsächlich in binären Systemen vorkommen, definiert man sie über Zweierpotenzen (statt wie sonst üblich über Zehnerpotenzen). So hat ein Kilobyte 2^{10} Byte, ein Megabyte 2^{20} Kilobyte und so weiter. Der Multiplikationsfaktor $2^{10} = 1.024$ kommt nahe genug an $10^3 = 1.000$ heran, sodass zumindest die Größenordnung stimmt, wenn wir in Zehnerpotenzen statt in Zweierpotenzen denken.

Wie wir im zweiten Teil des Skriptums sehen werden, ist *bit* eine Einheit für Information. Obwohl ein bit (klein geschrieben) viel mit einem Bit (groß geschrieben) gemeinsam hat, gibt es doch wichtige Unterschiede. Beispielsweise kommen Bits nur ganzzahlig vor, während 3,2 bit ein sinnvolles Maß ist.

1.2.3 Rechnen mit Binärzahlen

Ein Vorteil des Rechnens mit Binärzahlen liegt darin, dass es nur wenige Möglichkeiten gibt, zwei einstellige Binärzahlen zu kombinieren. Das

AND:	00 \rightarrow 0	01 \rightarrow 0	10 \rightarrow 0	11 \rightarrow 1
OR:	00 \rightarrow 0	01 \rightarrow 1	10 \rightarrow 1	11 \rightarrow 1
XOR:	00 \rightarrow 0	01 \rightarrow 1	10 \rightarrow 1	11 \rightarrow 0
NOT:	0 \rightarrow 1	1 \rightarrow 0		

Abbildung 1.6: Einfache Bit-Operationen

„kleine Einmaleins“ ist schnell aufgelistet:

$$0 \times 0 = 0 \quad 0 \times 1 = 0 \quad 1 \times 0 = 0 \quad 1 \times 1 = 1$$

Diese Operation ist ein AND, bei der das Ergebnis genau dann 1 ist wenn beide Argumente 1 sind, siehe Abbildung 1.6. Bei OR ist das Ergebnis genau dann 1 wenn mindestens ein Argument 1 ist, und bei XOR (exklusives Oder) wenn genau ein Argument 1 ist. NOT negiert ein Bit. Komplexere Operationen lassen sich auf diese einfachen Operationen zurückführen.

Manchmal haben Ergebnisse mehr Stellen als die Argumente. Betrachten wir die Addition zweier einstelliger Binärzahlen:

$$0 + 0 = 00 \quad 0 + 1 = 01 \quad 1 + 0 = 01 \quad 1 + 1 = 10$$

Das zusätzliche Bit des Ergebnisses heißt *Übertrag*. Das eigentliche Additionsergebnis wird durch XOR gebildet, der Übertrag durch AND. Wenn mehrstellige Zahlen addiert werden, ist der Übertrag zum nächsthöheren Bit zu addieren. Beispielsweise wird $11 + 01 = 100$ nach demselben Schema, nach dem wir Dezimalzahlen addieren, so berechnet: Wir beginnen mit dem LSB und bekommen $1 + 1 = 10$. Das letzte Bit des Ergebnisses ist daher 0. Dann addieren wir die nächsten Bits zu $1 + 0 = 01$ und zu dieser Zahl den Übertrag aus der Berechnung des vorigen Bits, also $01 + 1 = 10$. Das sind die vorderen Stellen des Ergebnisses. Ein Schema zum Berechnen eines Ergebnisses nennt man ganz allgemein *Algorithmus*.

Multiplikationen mehrstelliger Binärzahlen können doppelt so viele Stellen haben wie die multiplizierten Zahlen, siehe Abbildung 1.7. Wir können dafür denselben Algorithmus verwenden wie für Dezimalzahlen.

Folgende Programmstücke sollen inhaltliche Aussagen verdeutlichen und einen Vorgeschmack auf die Programmierung in Java geben. Lesern ohne Programmiererfahrung wird empfohlen, die Programmstücke aufmerksam durchzulesen und darin nach Anhaltspunkten zu suchen, die einen Sinn

00 × 00 = 0000	01 × 00 = 0000	10 × 00 = 0000	11 × 00 = 0000
00 × 01 = 0000	01 × 01 = 0001	10 × 01 = 0010	11 × 01 = 0011
00 × 10 = 0000	01 × 10 = 0010	10 × 10 = 0100	11 × 10 = 0110
00 × 11 = 0000	01 × 11 = 0011	10 × 11 = 0110	11 × 11 = 1001

Abbildung 1.7: Multiplikation von zwei zweistelligen Binärzahlen

ergeben. Keine Sorge: Diese Beispiele müssen noch nicht im Detail verstanden werden. Es geht eher darum, ein Gespür für das Aussehen von Programmen und den „Klang“ der Beschreibungen zu entwickeln.

Die Klasse `Bit` in Listing 1.8 deklariert eine Boole'sche Variable namens `value` und versteckt sie vor direkten Zugriffen von außen. Nur die überall sichtbaren Methoden `and`, `or`, `xor`, `not`, `add` und `intValue` können in Instanzen von `Bit` darauf zugreifen. Diese Methoden (abgesehen von `intValue`) setzen `this.value` auf das Ergebnis der jeweiligen Operation, lassen die über die Parameter `x` und `y` zugreifbaren Variablen aber unverändert und geben keine Antwort zurück. Statt `this.value` könnten wir auch nur `value` schreiben, da `this` den Empfänger der Nachricht, die gerade bearbeitet wird, bezeichnet. Die Methode `add` ändert auch die Variable im Parameter `carry`, der sowohl den Übertrag von der vorigen auf die aktuelle Bit-Addition (vor Ausführung der Methode) als auch den auf die nächste Addition (nach Ausführung) enthält. Für eine Bit-Addition werden zwei *Halbaddierer* verwendet, einer für die Addition von `x` und `y` und einer für die Addition des Ergebnisses mit dem Übertrag. Das Ergebnisbit des zweiten Halbaddierers wird als Ergebnis der gesamten Addition in `this` abgelegt, die beiden von den Halbaddierern berechneten Überträge durch OR verknüpft in `carry`. Um zu zeigen, dass die interne und externe Darstellung nicht übereinstimmen muss, wandelt `intValue` das Bit in eine Zahl um und gibt sie als Antwort zurück.

Die Klasse `Word` in Listing 1.9 deklariert eine Variable `value`, die ein *Array* von 32 Bits enthält. Ein Array kann man als eine Aneinanderreihung gleichartiger Werte verstehen, wobei man über einen *Index*, also eine Zahl in eckigen Klammern, angibt, der wievielte Wert in der Reihe gemeint ist. Die Variable `value` soll die 32 Bits eines Wortes enthalten. Statt der Zahl 32 verwenden wir genaugenommen die *Konstante* `SIZE`, die für die Zahl 32 steht. Die Definition einer Konstante ähnelt der Deklaration einer

Listing 1.8: Die Klasse `Bit` welche einfache Bit-Operationen implementiert

```

1 public class Bit {
2     private boolean value;           // Wert des Bits
3     public Bit() {                   // Konstruktor
4         this.value = false;
5     }
6     public void and (Bit x, Bit y) {
7         this.value = (x.value && y.value);
8     }
9     public void or (Bit x, Bit y) {
10        this.value = (x.value || y.value);
11    }
12    public void xor (Bit x, Bit y) {
13        this.value = (x.value != y.value);
14    }
15    public void not (Bit x) {
16        this.value = !x.value;
17    }
18    public void add (Bit x, Bit y, Bit carry) {
19        Bit half1bit = new Bit();     // lokale Variablen
20        Bit half1carry = new Bit();
21        Bit half2carry = new Bit();
22        half1bit.xor (x, y);          // 1. Halbaddierer
23        half1carry.and (x, y);
24        this.xor (half1bit, carry);   // 2. Halbaddierer
25        half2carry.and (half1bit, carry);
26        carry.or (half1carry, half2carry); // verknüpfe Überträge
27    }
28    public int intValue () {
29        if (value) { return 1; }
30        else      { return 0; }
31    }
32 }

```

Variablen, jedoch deuten die Wörter `static` und `final` darauf hin, dass die Initialisierung in der Definition vorgenommen werden muss und der Wert unveränderlich ist. Nach außen sichtbar sind nur ein Konstruktor, der das Array mit Bits initialisiert, und die Methoden `and`, `add` und `mul` (Multiplikation). Diese operieren auf ganzen 32-Bit-Worten und iterieren in je einer `for`-Schleife über die einzelnen Bits, vom LSB zum MSB. Eine Iteration ist der einmalige Durchlauf durch den Rumpf einer Schleife, und hier haben wir je eine Iteration für jedes Bit, das heißt, wir iterieren über die Bits. Bei Iterationen über einem Array erlaubt eine `for`-Schleife eine

Listing 1.9: Die Klasse Word implementiert Operationen auf 32-Bit-Worten

```

1 public class Word {
2     public static final int SIZE = 32;    // Konstante
3     private Bit[] value = new Bit[SIZE]; // Array von 32 Bits
4     public Word () {                      // Konstruktor
5         for (int i = 0; i < SIZE; i++) {
6             this.value[i] = new Bit();
7         }
8     }
9     public void and (Word x, Word y) {
10        for (int i = 0; i < SIZE; i++) {
11            this.value[i].and (x.value[i], y.value[i]);
12        }
13    }
14    public void add (Word x, Word y) {
15        Bit carry = new Bit();
16        for (int i = 0; i < SIZE; i++) {
17            this.value[i].add (x.value[i], y.value[i], carry);
18        }
19    }
20    public void mul (Word x, Word y) {
21        Word xshift = new Word();
22        this.and (xshift, xshift);          // this.value  <- 0..0
23        xshift.and (x, x);                  // xshift.value <- x.value
24        for (int i = 0; i < SIZE; i++) {
25            if (y.value[i].intValue() > 0) {
26                this.add (this, xshift);
27            }
28            xshift.add (xshift, xshift);    // Verschiebung um 1 Bit
29        }
30    }
31 }

```

etwas kompaktere Darstellung des Codes als eine while-Schleife:

```
for (int i = 0; i < SIZE; i++) { ... }
```

Zuerst wird ein *Schleifenzähler* *i* als lokale Variable vor der ersten Iteration deklariert und mit 0 initialisiert, dann kommt die Schleifenbedingung, die wie in einer while-Schleife vor jeder Iteration überprüft wird, danach steht die Anweisung *i++* gleichbedeutend mit *i = i + 1*, die am Ende jeder Iteration ausgeführt wird und den Schleifenzähler erhöht, und schließlich kommt noch der Schleifenrumpf, der in jeder Iteration (vor der Anweisung *i++*) ausgeführt wird.

Wie die Methoden von *Bit* ändern auch die Methoden in *Word* nur *this* und lassen die Parameter *x* und *y* unverändert. Die Methode *add* verwendet eine Variable *carry* für Überträge zwischen Bit-Additionen, verwirft deren Wert aber nach Beendigung der Wort-Addition. Dadurch wird genau genommen nicht $x+y$ berechnet, sondern $(x+y)$ modulo 2^{32} . In der Praxis werden solche Modulo-Rechnungen häufig verwendet, beispielsweise fast überall in Java. Die Methode *mul* berechnet $(x \times y)$ modulo 2^{32} , schneidet also alles ab, was über die darstellbaren 32 Bits hinausgeht.

Die Anweisung *this.and(xshift, xshift)* in *mul* ist trickreich. Sie kopiert den Inhalt des Wortes *xshift* nach *this*. Direkt nach der Erzeugung sind alle Bits von *xshift* mit 0 bzw. *false* gefüllt, und nach dieser Anweisung auch die von *this*. Auf gleiche Weise kopiert *xshift.and(x, x)* den Inhalt von *x* nach *xshift*. Der Wert dieser Variablen wird am Ende jeden Schleifendurchlaufs durch einen Aufruf von *xshift.add(xshift, xshift)* um ein Bit nach links verschoben: Eine Addition mit sich selbst (was dasselbe wie eine Multiplikation mit 2 ist) hängt im Binärsystem einfach nur 0 hinten an, beispielsweise $1 + 1 = 10$, genauso wie im Dezimalsystem eine Multiplikation mit 10 nur 0 an die Zahl hängt, beispielsweise $2 \times 10 = 20$. Das bisherige MSB in *xshift* geht dabei verloren, da Überläufe der Addition ignoriert werden. Falls das in einem Schleifendurchlauf gerade behandelte Bit von *y* gleich 1 bzw. *true* ist, wird der Inhalt von *xshift* zum Ergebnis in *this* addiert.

Wir haben angenommen, dass Worte nicht-negative Werte codieren. Die Multiplikation vorzeichenbehafteter Zahlen ist aufwendiger. Die Addition von Zahlen in Zweierkomplementdarstellung entspricht jedoch der von nicht-negativen Zahlen. Es ergeben sich jedoch eigenartige Effekte, wenn Ergebnisse mehr Binärstellen benötigen als vorhanden sind und daher abgeschnitten werden. Berechnungsergebnisse sind dann gänzlich falsch. Beispielsweise kann durch Abschneiden des Vorzeichenbits die Addition zweier positiver Zahlen ein negatives Ergebnis liefern.

1.2.4 Logische Schaltungen

Heutige digitale Systeme sind fast ausschließlich als elektrische Schaltkreise auf der Basis von Halbleitern, also Transistoren und Dioden realisiert. Informatiker interessieren sich dafür, wie ein System aus vorgegebenen *logischen Bausteinen* zusammengesetzt ist. Wie diese Bausteine intern durch Halbleiter realisiert sind, ist für ein Verständnis des Systems nicht entscheidend. Die wichtigsten logischen Bausteine sind sogenannte *Gat-*

ter, welche die oben beschriebenen einfachen Bit-Operationen ausführen. Beispielsweise hat ein AND-Gatter zwei Eingänge und einen Ausgang und sorgt dafür, dass am Ausgang eine dem Wert 1 entsprechende elektrische Spannung liegt wenn an beiden Eingängen ebenfalls eine 1 entsprechende Spannung liegt, sonst liegt am Ausgang eine 0 entsprechende Spannung. OR-, XOR- und NOT-Gatter erfüllen ihre Funktionen auf ähnliche Weise. Oft sind mehrere logische Funktionen in ein Gatter integriert, beispielsweise AND und NOT in ein NAND-Gatter. Komplexere Schaltungen entstehen durch Kombination mehrerer Gatter, beispielsweise ein Halbaddierer durch die Kombination eines XOR- und eines AND-Gatters. Aus zwei NAND-Gattern lässt sich ein *Flip-Flop* bauen, das als Speicher für ein Bit dient. Gatter reichen aus, um daraus Computer zu bauen.

Gatter brauchen einige Zeit, bevor der Ausgang den Spannungslevel erreicht, den er aufgrund der Spannungen an den Eingängen haben sollte. Solche *Gatterlaufzeiten* bewegen sich im Bereich von weniger als 0,1 ns bis über 100 ns (Nanosekunden, $1 \text{ ns} = 10^{-9}$ Sekunden). Wenn für eine Berechnung mehrere Gatter hintereinander durchlaufen werden müssen, addieren sich die Gatterlaufzeiten. Meist werden viele Berechnungen durchgeführt, wobei Zwischenergebnisse in folgende Berechnungen einfließen. Dabei ist es wichtig zu wissen, ab wann die Ergebnisse einer Berechnung vorliegen, die Ausgangssignale also stabil sind. Erst dann kann die nächste Berechnung gestartet werden. Meist gibt man einen fixen Takt vor. Mit jedem Takt beginnt eine neue Berechnung. Die Taktfrequenz wird so gewählt, dass alle Ergebnisse zu Beginn des nächsten Taktes sicher vorliegen.

Es ist leicht vorstellbar, wie ein Webstuhl programmiert wird: Man entwirft ein Webmuster und stanzt händisch für bestimmte Überkreuzungen der Fäden an entsprechenden Stellen Löcher in die Lochkarten. Fast alle logischen Schaltungen werden dagegen durch Programme spezifiziert. Die verbreitetste *Hardwarebeschreibungssprache* ist *VHDL*, siehe Abbildung 1.10. Sowohl kleine als auch große Schaltungen bis hin zu ganzen Prozessoren sind damit beschreibbar. Derart beschriebene Schaltungen können auf Computern simuliert werden um Fehler zu finden, und es ist möglich, die Beschreibungen „in Hardware zu gießen“, also daraus echte integrierte Schaltkreise (Chips) zu erzeugen. Eine nachträgliche Änderung einmal erzeugter integrierter Schaltkreise ist aber nicht mehr möglich.

VHDL Programme beschreiben die Struktur der logischen Schaltungen, nicht den dynamischen Ablauf, also die Hintereinanderreihung einzelner Berechnungsschritte. Die übliche Programmierung von Computern kon-

```
library IEEE;                -- aus der Bibliothek IEEE:
use IEEE.std_logic_1164.all; -- Form der Bit-Darstellung

entity ANDGATE is            -- ein ANDGATE hat
    port (IN1 : in std_logic; -- zwei Eingänge (IN1, IN2)
          IN2 : in std_logic; -- und einen Ausgang (OUT1)
          OUT1: out std_logic); -- als einfache Bits
end ANDGATE;

architecture RTL of ANDGATE is
begin
    OUT1 <= IN1 and IN2;      -- Berechnung des Ergebnisses
end RTL;
```

Abbildung 1.10: VHDL Programm zur Spezifikation eines AND-Gatters

zentriert sich aber genau auf den dynamischen Ablauf und nimmt die Struktur der logischen Schaltungen als fix vorgegeben an. Ab jetzt werden wir uns verstärkt mit dem dynamischen Ablauf beschäftigen.

1.3 Maschinen und Architekturen

In der Regel wollen wir Computer programmieren. Den „Computer“ an sich gibt es aber nicht, sondern zahlreiche Arten von Maschinen, die auf unterschiedliche Weise programmierbar sind. Nicht jede Maschine, für die wir Programme schreiben, existiert real. In der Informatik beschäftigen wir uns überwiegend mit Maschinen, die nur in unserer Vorstellung existieren. Programme für solche *abstrakte Maschinen* können dennoch auf realen Maschinen zur Ausführung kommen. Die Grenzen zwischen realen und abstrakten Maschinen sind fließend. Ein und dieselbe Maschine bietet mehrere Betrachtungsebenen, wobei höhere Ebenen abstrakter sind. Für die Programmierung sind gerade die höheren Ebenen interessant.

1.3.1 Architektur üblicher Computer

Viele Computer sind nach der *Von Neumann-Architektur* aufgebaut, die John von Neumann 1945 vorgeschlagen hat. Das sind ihre Komponenten:

Arithmetic Logic Unit (ALU): Dieser Teil führt Berechnungen und logische Verknüpfungen durch. Typische Operationen sind Grundrechenarten sowie Verknüpfungen mittels AND, OR, NOT, etc.

Control Unit: Hier wird der Programmablauf gesteuert. Ein Befehl nach dem anderen wird in Steuersignale umgewandelt, die z.B. den Speicher anweisen, Daten zu liefern, die ALU, darauf eine Operation auszuführen, und wieder den Speicher, die Ergebnisse abzulegen.

Memory: Der Speicher enthält die Daten und macht sie bei Bedarf den anderen Komponenten (vor allem der ALU) zugänglich.

I/O Unit: Die Ein-/Ausgabeeinheit stellt die Verbindung zur *Peripherie*, also zu Geräten wie Festplatte, Tastatur, Grafikkarte und Bildschirm und damit auch zur Außenwelt des Computers her.

Alle Komponenten sind über ein gemeinsames *Bus-System* miteinander verbunden, das ist ein System elektrischer Leitungen zusammen mit Protokollen, die (ähnlich einer Straße zusammen mit Verkehrsregeln) dafür sorgen, dass der Nachrichtentransport zwischen den Komponenten mit möglichst wenig gegenseitiger Behinderung vonstatten geht.

Der Speicher ist als *Random Access Memory (RAM)* aufgebaut, das heißt, die im Speicher abgelegten Worte sind einzeln ansprechbar. Programme werden, wie auch alle anderen Daten, im Speicher abgelegt. An einer speziellen Adresse liegt der *Befehlszeiger (Program Counter = PC)*, der die Adresse des nächsten auszuführenden Programmbefehls enthält.

Folgende Schritte werden endlos (bis zum Abschalten) wiederholt:

- Ein Befehl wird von der Adresse, auf die der PC zeigt, aus dem Speicher in die Control Unit gelesen.
- Der PC wird erhöht, sodass er auf den nächsten Befehl zeigt.
- Der Befehl wird von der Control Unit interpretiert und ausgeführt.

Die Ausführung eines Befehls kann den PC verändern, sodass der nächste Befehl nicht unmittelbar nach dem vorigen stehen muss. Eine solche Änderung im Programmablauf nennt man *Programmsprung* oder kurz *Sprung*. Über Sprünge werden Schleifen und bedingte Verzweigungen realisiert.

Die *Harvard-Architektur* ändert die Von Neumann-Architektur dahingehend ab, dass Befehle und Daten in getrennten Speichern liegen. Das hat Vorteile hinsichtlich der Effizienz und Betriebssicherheit, da Befehle nicht mehr (unabsichtlich) durch Daten überschrieben werden können. Aktuelle Computer verwenden Elemente beider Architekturen.

Oft gibt es eine ganze Hierarchie von Speicherebenen, je höher die Ebene, desto effizienter und kleiner. Ganz oben stehen *Register* für die gerade

```
section .text                ; Beginn des Abschnitts mit Programmcode
global _start               ; Ausfuehrung beginnt bei _start
_start:
    mov ecx, hello          ; Kopiere Adresse des Textes nach Register ecx
    mov edx, length         ; Kopiere Laenge des Textes nach Register edx
    mov ebx, 1              ; Dateinummer der Standard-Ausgabe (= 1) in ebx
    mov eax, 4              ; Funktionsnummer 4 des Systemaufrufs = Ausgabe
    int 80h                 ; System-Interrupt; Linux fuehrt Funktion aus
    mov ebx, 0              ; Lege Ergebniswert des Programms in ebx ab
    mov eax, 1              ; Funktionsnummer 1 in eax = Programm beenden
    int 80h                 ; System-Interrupt; Linux beendet Programm

section .data                ; Beginn des Abschnitts mit Daten
hello db 'Hello World!'     ; der auszugebende Text
length equ $ - hello        ; aktuelle Position - Beginn von hello
```

Abbildung 1.11: Einfaches Intel i386 Assembler-Programm (Hello World!)

verarbeiteten Daten. Für den PC und Ähnliches gibt es Spezialregister, die nur von speziellen Befehlen, z.B. Sprungbefehlen, verwendet werden. Die nächsten Ebenen bilden *Caches*, meist getrennt in Daten- und Befehls-Caches. Diese enthalten eine Auswahl der Daten aus der untersten Ebene, dem Hauptspeicher, um raschere Zugriffe darauf zu erlauben.

Die am häufigsten verwendeten Daten kommen automatisch in Caches, sodass wir uns beim Programmieren nicht darum kümmern brauchen. Caches sind technische Details zur Leistungssteigerung, die in der *Architektur* der Maschine nicht sichtbar sein müssen. Wir unterscheiden die Architektur von der *Implementierung der Architektur*, das ist eine konkret realisierte Maschine. Die Architektur beschreibt nur eine Grobstruktur und die ausführbaren Befehle, also alles, was man zum Programmieren über die Maschine wissen muss. Alle Implementierungen einer Architektur verstehen die Programme, die für diese Architektur geschrieben werden.

Befehle werden durch Maschinenworte dargestellt, also durch Bitmuster, mit denen Maschinen einfach umgehen können, Menschen aber nicht. Für die hardware-nahe Programmierung verwendet man daher *Assembler-Sprachen* und nicht direkt Maschinen-Sprachen. Eine Assembler-Sprache stellt Befehle, Adressen und Daten durch für Menschen besser lesbare Symbole dar, und ein Assembler übersetzt die Symbole in die eigentlichen Befehle. Abbildung 1.11 zeigt ein Assembler-Programm für die Intel i386 Architektur unter dem Betriebssystem Linux. Dieses Programm

ist nur mit viel Wissen über die Maschine und das Betriebssystem verständlich. Wegen ungünstiger Eigenschaften meidet man die Assembler-Programmierung weitgehend und programmiert in höheren Sprachen.

Ein *Prozessor* fasst die zentralen Teile eines Computers auf einem Chip zusammen, das sind ALU, Register, Control Unit und (Teile der) Speicher-verwaltung. Meist zählen auch Caches dazu, aber nicht der Hauptspeicher. Viele Prozessoren haben mehrere *Prozessor-Kerne*, die weitgehend unabhängig voneinander arbeiten, so als ob wir mehrere Prozessoren hätten. Damit sind mehrere Programme gleichzeitig ausführbar. Manchmal werden Programme in Teile aufgespaltet, die gleichzeitig (*nebenläufig* oder *parallel*) ausführbar sind und damit die Ressourcen mehrerer Prozessoren und Prozessor-Kerne nutzen können. Möglicherweise gleichzeitige Ausführungen innerhalb eines Programms heißen *Threads*. Die Verwaltung von Threads kann sehr kompliziert sein.

1.3.2 Abstrakte Maschinen und Modelle

Genaugenommen bezieht sich der Begriff „Architektur“, wie wir ihn in Abschnitt 1.3.1 eingeführt haben, auf eine *abstrakte Maschine*, also eine abstrakte Beschreibung der Implementierungen. Im Falle der Computer-Architektur ist der *Abstraktionsgrad* gering, das heißt, die Architektur kommt nahe an die reale Maschine heran. Häufig verwenden und programmieren wir abstrakte Maschinen mit einem höheren Abstraktionsgrad. Die genaue Beziehung zwischen der abstrakten Maschine und deren Implementierung auf einer realen Maschine ist dabei kaum erkennbar.

Die *Java Virtual Machine (JVM)* ist eine abstrakte Maschine. Auf den ersten Blick unterscheiden sich JVM-Programme (siehe Abbildung 1.12) nur unwesentlich von Assembler-Programmen. Allerdings wurden die Befehle einer Hardware-Architektur dafür optimiert, dass sie effizient auf bestimmten realen Maschinen ausführbar sind, während die Befehle der JVM hinsichtlich der Unabhängigkeit von realen Maschinen optimiert wurden. Programme für eine Hardware-Architektur laufen nur auf den Computern, die diese Architektur unter einem bestimmten Betriebssystem verwenden. Der höhere Abstraktionsgrad entkoppelt die JVM von Hardware- und Betriebssystem-Details, sodass JVM-Programme fast überall laufen können. Sie sind *portabel*, also von einer Maschine auf eine andere übertragbar. Damit können Computer in Netzwerken einfacher miteinander kooperieren. Allerdings hat der höhere Abstraktionsgrad auch einen Preis:

```
public class Hello extends java.lang.Object{
public Hello();           //Konstruktor (nicht verwendet)
Code:
    0: aload_0
    1: invokespecial #1; //Method java/lang/Object."<init>"
    4: return

public static void main(java.lang.String[]);
Code:
    0: getstatic #2;          //Field java/lang/System.out
    3: ldc #3;                //String Hello World!
    5: invokevirtual #4; //Method java/io/PrintStream.println
    8: return
}
```

Abbildung 1.12: JVM-Code entsprechend der Klasse Hello (Listing 1.1)

Programme sind nicht so effizient ausführbar und in Betriebssysteme eingebunden als wenn sie für ein bestimmtes System entwickelt worden wären.

Abstrakte Maschinen sind einfacher programmierbar: Man braucht sich nicht um Details der Hardware und des Betriebssystems kümmern. Die JVM ist für die Programmierung bei weitem noch nicht abstrakt genug. Programme werden auf einer viel höheren Abstraktionsebene entwickelt.

Abstrakte Maschinen auf sehr hohem Abstraktionsniveau werden als *Berechnungsmodelle* bezeichnet. Eines der einflussreichsten Berechnungsmodelle ist der *Lambda-Kalkül*, der den Begriff der mathematischen Funktion definiert und alles berechnen kann, was über Funktionen ausdrückbar ist, siehe Abschnitte 1.5.2 und 1.5.3. Die *Turing-Maschine* ist ein historisch bedeutendes Berechnungsmodell, das Berechnungen durch Beschreiben und Lesen eines gedanklich unendlich langen Bandes durchführt und nichts mit dem Lambda-Kalkül zu tun hat. Es spielt keine Rolle, ob eine solche Maschine tatsächlich realisierbar ist. Vielmehr handelt es sich um Maschinen, die nur in unserer Vorstellung existieren und sich daher nicht an technische Grenzen halten müssen. Das gibt uns große Freiheit. Wie wir aus Untersuchungen einer Vielzahl solcher Berechnungsmodelle wissen, ist die Freiheit aber nicht unendlich groß. Auch Berechnungsmodelle unterliegen gewissen Gesetzmäßigkeiten und Grenzen, genauso wie die Physik Gesetzmäßigkeiten und Grenzen in der Natur aufzeigt. Nicht alles, was wir gerne berechnen würden, ist tatsächlich berechenbar. Was berechenbar ist, hängt (ab einer bestimmten Mächtigkeit des Modells) nicht vom Berechnungsmodell ab. So können Lambda-Kalkül und Turing-Maschine

wie viele weitere Maschinen (*Turing-vollständige Maschinen*) trotz ihrer Unterschiedlichkeit genau dasselbe berechnen. Das Erkennen von Gesetzmäßigkeiten in Berechnungsmodellen macht einen Großteil der Informatik aus. Daneben geht es natürlich auch darum, Aufgaben so zu lösen, dass wir Gesetzmäßigkeiten ausnützen und nicht an die Grenzen stoßen.

Hinter jeder Programmiersprache steckt ein Berechnungsmodell. Wir programmieren diese abstrakte Maschine, die wir dabei (oft unbewusst) im Kopf haben. Losgelöst vom Berechnungsmodell ergibt eine Sprache keinen Sinn. Bevorzugt verwenden wir Berechnungsmodelle, die für Menschen relativ einfach zu verstehen sind und die Intuition unterstützen. Das, was berechenbar ist, hängt ja nicht vom Berechnungsmodell ab, solange die Sprache bzw. das Modell Turing-vollständig ist. Vom Modell hängt aber sehr wohl ab, wie einfach etwas ausdrückbar ist, wie viel Text dafür nötig ist, wie einfach Programme lesbar und änderbar sind, und so weiter.

Viele Berechnungsmodelle sind mathematische Modelle als Grundlagen für theoretische Analysen auf äußerst abstraktem Niveau. Programmiersprachen haben einen etwas anderen Fokus: Manchmal möchte man auf einer hardware-näheren Ebene programmieren und Eigenschaften realer Maschinen direkt beeinflussen können. Berechnungsmodelle hinter *höheren Programmiersprachen* gehen von mathematischen Modellen aus, erweitern diese aber oft um Konzepte auf hardware-näheren Ebenen. Es gibt Unterschiede im Abstraktionsgrad: Hardware-nahe höhere Programmiersprachen haben eine gute Unterstützung für die Programmierung auf niedrigerem Abstraktionsniveau (neben dem auf hohem Niveau), während reine Hochsprachen kaum auf niedrigerem Niveau programmierbar sind, aber die Programmierung auf hohem Niveau meist besser unterstützen.

1.3.3 Objekte als Maschinen

In Abschnitt 1.1 haben wir gesehen, dass Objekte in Java-Programmen eine essentielle Rolle spielen. Software-Objekte simulieren Objekte der realen Welt, sodass wir Wissen aus der realen Welt direkt in die Software übertragen können. Wir können jedes einzelne Objekt aber auch als abstrakte Maschine sehen, welche die durch die Sprache vorgegebene Maschine erweitert. Dabei helfen folgende drei Eigenschaften von Objekten:

Objektzustand: Die Werte aller Variablen eines Objekts ergeben zusammen den Objektzustand. Dieser ist durch Zuweisungen anderer Werte an die Variablen änderbar. Man kann den Objektzustand mit dem

Zustand eines Computers (vor allem des Speicherinhalts) vergleichen, jedoch ist ein Objekt kleiner als der Computer und der Zustand eines Objekts damit leichter überschaubar als der des ganzen Computers.

Objektverhalten: Das Objektverhalten beschreibt, wie das Objekt auf den Empfang von Nachrichten reagiert, das entspricht dem Code in den Methodendefinitionen. Ähnlich wie ein Computer einen Befehl nach dem anderen ausführt und dabei den Speicherzustand verändert, so wird auch im Objekt (entsprechend den empfangenen Nachrichten) eine Methode nach der anderen ausgeführt und dabei der Objektzustand verändert.²

Objektidentität: Jedes Objekt hat eine eindeutige Identität, die das Objekt unverwechselbar macht. Diese ist nötig, um ein Objekt von anderen ähnlichen Objekten zu unterscheiden (auch wenn sich der Objektzustand verändert) und Nachrichten an ganz bestimmte Objekte senden zu können. Computer verwenden häufig Internetadressen zur eindeutigen Identifizierung und zum Senden von Nachrichten.

Hinter allen Instanzen einer Klasse steckt dieselbe abstrakte Maschine. Wenn wir eine Klasse entwickeln, denken wir oft an diese abstrakte Maschine. Durch diese Denkweise können wir uns auf die wesentlichen Aspekte eines ganz bestimmten kleinen Teils der Software konzentrieren. Wir vermeiden damit, dass wir stets die viel zahlreicheren und undurchschaubareren Abhängigkeiten zwischen den Zuständen des ganzen Programms im Kopf haben müssen. Beispielsweise beschreiben die Klassen `UnbekannteZahl`, `Bit` und `Word` in den Listings 1.2, 1.8 und 1.9 abstrakte Maschinen, die für Leser mit Programmiererfahrung durchaus verständlich sind. Diese Klassen sind aber keine vollständigen Programme, weil wesentliche Teile fehlen. Erst die Klasse `Zahlenraten` in Listing 1.3 stellt über Ein- und Ausgaben eine Verbindung zwischen einer Instanz von `UnbekannteZahl` und der Außenwelt her.

Man könnte meinen, `UnbekannteZahl` sei gar nicht nötig, weil die zu erratende Zahl gleich als Instanz von `int` in `Zahlenraten` dargestellt

²Genaugenommen werden die Methoden eines Objekts sehr oft teilweise überlappt ausgeführt, während die Hardware eines Computers normalerweise wirklich nur einen Befehl nach dem anderen ausführen kann. Vor allem in gut durchdachten Programmen spielt dieser Unterschied aber keine wesentliche Rolle. Ein Grund dafür liegt darin, dass wir konzeptuell voneinander möglichst unabhängige Methoden haben und komplizierte Überlappungen vermeiden wollen.

werden könnte. Eine solche Sichtweise ist verständlich. Allerdings würde man dabei viel verlieren: `UnbekannteZahl` wurde als in sich logisch konsistente Einheit unabhängig von `Zahlenraten` entwickelt und kann auch in anderen Programmen verwendet werden. Jede der beiden Klassen ist einfacher verständlich als eine kombinierte Klasse. Einfachheit und Verständlichkeit sind in der Softwareentwicklung sehr wichtig.

Um ein Objekt zu verwenden, brauchen wir nur dessen *Schnittstelle* verstehen. Die Schnittstelle beschreibt die Nachrichten, die vom Objekt verstanden werden. Die *Implementierung des Objekts* bzw. die *Implementierungen der Methoden* der entsprechenden Klasse, also den Code, der beim Empfang von Nachrichten ausgeführt wird, brauchen wir dagegen nicht kennen. Wir müssen beispielsweise nur wissen, dass eine Nachricht `gleich(3)` an eine Instanz von `UnbekannteZahl` genau dann `true` als Antwort liefert, wenn die gekapselte Zahl gleich 3 ist. Dagegen spielt es keine Rolle, wie die gekapselte Zahl intern dargestellt ist und wie der Vergleich funktioniert. Wir können die Implementierung so verändern, dass die Schnittstelle und damit die Verwendungsmöglichkeiten gleich bleiben.

1.3.4 Softwarearchitekturen

Der Begriff *Architektur* bezieht sich nicht nur auf die Baukunst oder Hardware, sondern auch auf Software. Unter einer *Softwarearchitektur* verstehen wir eine Beschreibung der wichtigsten Teile der Software und der Beziehungen zwischen diesen Teilen. Man bezeichnet diese Teile als *Komponenten* oder in kleinerem Maßstab als *Module*. Komponenten haben viele Eigenschaften von Objekten, können aber deutlich größer sein. Eine wichtige Rolle kommt ihren *Schnittstellen* zu, die bestimmen, wie die Komponenten zusammenspielen und miteinander kombinierbar sind.

In anderen Bereichen der Technik wie im Maschinenbau, in der Elektrotechnik und im Bauwesen ist eine auf Komponenten ausgelegte Entwicklung schon seit langem nicht mehr wegzudenken. Beispielsweise besteht ein Auto aus Komponenten wie Motor, Getriebe und Radaufhängung. Eine kaputte Komponente ist gegen eine neue austauschbar. Komponenten können unabhängig voneinander entwickelt werden, solange die Schnittstellen zusammenpassen. Auf das Auto bezogen ist das der Fall, wenn Schrauben, Bohrungen und Flansche an der richtigen Stelle sitzen und die übertragene Kräfte innerhalb der erwarteten Grenzen liegen. Ganz unterschiedliche Experten können sich um die Entwicklung der einzelnen Komponenten und den Zusammenbau des Ganzen kümmern. Ein Motorenbauer weiß

nur wenig über Radaufhängungen, der Experte für Radaufhängungen nur wenig über Motoren, und der Autobauer kennt alle Komponenten oberflächlich, kann sie aber nicht selbst entwickeln. So betrachtet jeder nur seine Maschine auf dem jeweils notwendigen Detaillierungsgrad.

In der Softwareentwicklung ist es ähnlich. Häufig konstruieren wir Software, indem wir (wie ein Autobauer) fertige Komponenten miteinander kombinieren. Oft erstellen wir (wie ein Motorenbauer) eine Komponente, die von anderen genutzt wird. Dabei profitieren wir von unserem Spezialwissen in einem engen Bereich und achten darauf, dass die Schnittstellen unserer Komponente mit denen anderer zusammenpassen. Anders als Komponenten am Auto geht Software nicht durch Abnutzung kaputt, aber auch Software altert: Erwartungen und Einsatzbereiche ändern sich, und daher müssen auch Softwarekomponenten gelegentlich erneuert werden.

Es gibt vielfältige Gestaltungsmöglichkeiten für Softwarearchitekturen. Beispielsweise hat sich in vielen Bereichen eine aus *Schichten* aufgebaute Architektur bewährt, die wie die Schalen einer Zwiebel übereinander liegen. Die Komponenten einer Schicht haben nur Schnittstellen zu Komponenten der direkt darunter bzw. darüber liegenden Schicht. Das entkoppelt die Komponenten voneinander und macht sie leichter austauschbar, wirkt sich aber negativ auf die Laufzeiteffizienz aus. Zahlreiche neuere Systeme sind in drei Schichten aufgebaut: Die oberste Schicht dient der Präsentation und Eingabe von Daten und wird in der Regel durch einen Web-Browser realisiert. Die unterste Schicht verwaltet die Daten meist in einer Datenbank. Die mittlere Schicht enthält die eigentliche Anwendungslogik, verknüpft Eingaben aus der obersten Schicht mit Daten aus der untersten und sorgt dafür, dass alle Daten in sich konsistent und geschützt bleiben. Klare Trennungen zwischen Schichten erlauben auch die Ausführung unterschiedlicher Schichten auf unterschiedlichen Rechnern. So spiegelt die Softwarearchitektur manchmal auch die Hardwarearchitektur wider.

Bei ausreichend detaillierter Betrachtung kann jede Komponente wiederum aus Komponenten in mehreren Schichten aufgebaut sein. Softwarearchitekturen sorgen durch Abstraktion über Details dafür, dass die Gesamtstruktur eines großen Systems verständlich bleibt. Wir können das System auf jedem Detaillierungsgrad betrachten, und auf jeder Betrachtungsebene sehen wir eine andere abstrakte Maschine, die unser Denken auf signifikante Weise beeinflusst. Gerade wegen diesem starken Einfluss auf unser Denken ist die Softwarearchitektur so wichtig: Eine gute Architektur macht vieles einfacher, eine schlechte lässt sich durch noch so ausgefeilte Programmieretechniken kaum in den Griff bekommen.

1.4 Formale Sprachen, Übersetzer und Interpreter

Programmiersprachen unterscheiden sich wesentlich von natürlichen Sprachen wie Deutsch oder Englisch. Der Grund dafür liegt in ganz anderen Zielsetzungen: In einem Programm gibt man einer dummen, interesselosen Maschine präzise Anweisungen, die ohne Überlegung befolgt werden können. Menschen kommunizieren auf annähernd gleichem Niveau, sodass der Austausch emotionaler und unpräziser Informationen viel effektiver ist. Programmiersprachen ähneln eher den in der Mathematik verwendeten formalen Modellen und Sprachen, da es auf Präzision ankommt. Wir wollen einige Grundlagen von Sprachen betrachten und untersuchen, wie Programme auf einer Maschine zur Ausführung kommen.

1.4.1 Syntax, Semantik und Pragmatik

Beschreibungen natürlicher wie formaler Sprachen umfassen drei Aspekte:

Syntax: Die Syntax regelt den *Aufbau* der Sätze bzw. eines Programms. Alle Satz- oder Programmteile stehen zueinander in Beziehungen, die durch ein Regelsystem, die *Grammatik* der Sprache, beschrieben sind. In natürlichen Sprachen gibt es beispielsweise Regeln für Satzformen, die Beistrichsetzung, die Großschreibung, etc. In formalen Sprachen ist es ähnlich. Regeln schreiben beispielsweise vor, wie Deklarationen, Definitionen und Anweisungen aufgebaut sind, dass einfache Anweisungen mit Strichpunkt enden und Rümpfe von Methoden in geschwungene Klammern eingeschlossen sind. Anhand der Syntax eines Sprachelements können wir erkennen, ob es sich um eine Deklaration, eine Anweisung oder etwas anderes handelt.

Semantik: Die Semantik legt die *Bedeutung* von Begriffen, Sätzen bzw. Programmen fest. Zumindest für formale Sprachen ist auch die Semantik über Regeln festgelegt. Diese Regeln bestimmen den genauen Ablauf eines Programms und sind oft viel komplexer als syntaktische Regeln. Im Gegensatz zu natürlichen Sprachen wird für formale Sprachen viel Aufwand betrieben, um die Semantik genau zu beschreiben.

Pragmatik: Die Pragmatik untersucht das Verhältnis der Sprache zum Sprecher und zum Angesprochenen. Beispielsweise macht es in manchen Situationen einen großen Unterschied, ob man jemanden mit „Sie“ oder „Du“ anspricht, auch wenn die Sätze die gleiche Semantik

```
Statement = { {Statement} }
           | if ( Expression ) Statement [else Statement]
           | for ( ForInit ; [Expression] ; [Expression] ) Statement
           | while ( Expression ) Statement
           | return [Expression] ;
           | [Expression] ;
           | ...
```

Abbildung 1.13: Ausschnitt aus der Grammatik von Java (in EBNF)

haben. Computer und formale Systeme nehmen solche Unterschiede nicht wahr. Für die Programmierung ist die Pragmatik dennoch wichtig: Man versteht darunter alle praktischen Aspekte der Sprachverwendung, beispielsweise den Einsatz bestimmter Sprachlemente zur Verbesserung der Lesbarkeit von Programmen.

Die Syntax einer Programmiersprache wird in dessen Grammatik festgelegt. Abbildung 1.13 zeigt einen kleinen, vereinfachten Ausschnitt aus der Grammatik von Java, genauer gesagt einer Anweisung (englisch *statement*). Wir verwenden die *BNF* oder *EBNF* (*Extended Backus Naur Form*) als *Meta-Sprache*, eine Sprache über der Sprache, zur Festlegung der Grammatik. Wörter, die als sogenannte *Schlüsselwörter* im Programm genau gleich vorkommen wie in der Grammatik, sind fett geschrieben. Schlüsselwörter dürfen in keiner anderen Bedeutung, z.B. als Variablennamen, vorkommen. Der senkrechte Strich trennt Alternativen voneinander, $A|B$ bedeutet also, dass das beschriebene Sprachelement so wie A oder so wie B aussieht. In nicht fett geschriebenen geschwungenen Klammern vorkommende Elemente können beliebig oft wiederholt oder gar nicht vorkommen, $\{ \{A\} \}$ steht also für $\{ \}$ oder $\{A\}$ oder $\{AA\}$ und so weiter. Elemente in eckigen Klammern sind optional, sie können vorkommen, müssen aber nicht; $[A]$; steht also für A ; oder nur $;$. Sogenannter *white space* bestehend aus Leerzeichen, Tabulatorzeichen und Zeilenumbrüchen bleibt in dieser Syntaxbeschreibung unberücksichtigt, abgesehen davon, dass white space nebeneinander stehende Wörter voneinander trennt; `int zahl` ist nur ein Wort, während `int zahl` zwei ganz andere Wörter sind.

Wie wir anhand der Grammatikregel in Abbildung 1.13 erkennen können, ist sowohl `if (zahl < 0) { zahl = zahl + grenze ; }` als auch, durch Weglassung von Klammern, `if (zahl < 0) zahl = zahl + grenze ;` eine An-

weisung (vorausgesetzt, dass `zahl < 0` und `zahl = zahl + grenze` Ausdrücke, englisch *expressions*, sind). Die Bedeutungen dieser Anweisungen gehen aus der Grammatik nicht hervor. Mit viel Aufwand oder Erfahrung können wir aus der umfangreichen Beschreibung von Java³ ableiten, dass diese beiden Anweisungen dieselbe Bedeutung, also dieselbe Semantik haben. Für die Ausführung des Programms ist es in diesem Fall egal, ob wir die geschwungenen Klammern hinschreiben oder nicht. Aber hinsichtlich der Pragmatik gibt es Unterschiede: Die Klammern können wir nur weglassen, wenn im Rumpf der bedingten Anweisung nur eine Anweisung steht, nicht bei mehreren. Aufgrund praktischer Erfahrungen schreiben wir die Klammern hin, da es dadurch später einfacher und weniger fehleranfällig ist, den Rumpf um weitere Anweisungen zu ergänzen. Außerdem werden wir die Anweisung eher in dieser Form hinschreiben:

```
if (zahl < 0) {
    zahl = zahl + grenze;
}
```

Zusätzlicher white space erhöht die Lesbarkeit, und die tiefere Einrückung des Rumpfes lässt die Struktur auf den ersten Blick erkennen.

Gewisse Kenntnisse der Syntax und Semantik einer Programmiersprache sind die Grundvoraussetzung, um überhaupt Programme schreiben zu können. Der Einsatz pragmatischen Wissens macht dagegen den Unterschied zwischen guten und nicht so guten Programmen aus. Oft haben Programmieranfänger sehr viel Wissen über die Syntax und Semantik einer Sprache, können dieses Wissen beim Programmieren aber kaum umsetzen. Ihnen fehlt noch bestimmtes pragmatisches Wissen, das sich erst im Laufe der Zeit mit der Programmiererfahrung langsam entwickelt.

1.4.2 Bestandteile eines Programms

Programme und Programmteile, egal in welcher Programmiersprache, setzen sich aus Beschreibungen folgender Aspekte zusammen:

Datenstrukturen: Wir legen Daten in Variablen ab und beschreiben die Daten durch die Namen und Typen der Variablen. Daten stehen in Beziehungen zueinander und bilden dabei *Datenstrukturen*. Beispielsweise sind in Listing 1.9 die Bits eines Wortes zu einem Array,

das ist eine bestimmte Datenstruktur, zusammengefasst. Programmiersprachen geben meist einfache Datenstrukturen vor und erlauben uns, daraus beliebige komplexe Datenstrukturen aufzubauen.

Algorithmen: Ein Algorithmus ist eine aus endlich vielen Schritten bestehende eindeutige Handlungsvorschrift zur Lösung eines Problems, quasi ein „Kochrezept“, nach dem sogar eine Maschine ohne jegliche Intelligenz eine Lösung zustandebringt. Unter einem *Problem* (im mathematischen Sinn) verstehen wir dabei eine Aufgabe, die wir lösen sollen. Die Lösung selbst besteht aus Daten. Es ist klar, dass wir eine Lösung in endlich vielen Schritten finden müssen, da wir eine Lösung in unendlich vielen Schritten niemals erreichen werden. In Java beschreiben wir Algorithmen durch Methoden und Konstruktoren.⁴

Programmorganisation: Es muss möglich sein, Programme in voneinander möglichst unabhängige Teile (Klassen, Module und Komponenten) zu zerlegen. Andernfalls wäre die Komplexität größerer Programme nicht beherrschbar. Im Programm müssen Zusammenhänge zwischen den einzelnen Teilen beschrieben sein. In Java werden Variablen und Methoden zu Objekten zusammengefasst, und entsprechende Klassen, die im selben Ordner stehen, bilden *Pakete*.

Umgebung: Die Umgebung einer Stelle im Programm bestimmt die Namen, die an dieser Stelle sichtbar sind. Beispielsweise sind in einer Methode die lokalen Variablen sowie die in der Klasse deklarierten bzw. definierten Variablen und Methoden sichtbar. Wir sehen von einer Java-Klasse aus andere Klassen, die im selben Ordner stehen. Damit wirkt sich auch die Programmorganisation auf die sichtbaren Namen aus. Mittels `import`-Deklarationen wie in den Listings 1.2 und 1.3 fügen wir `public` Klassen, die irgendwo anders definiert sind, zur Menge der sichtbaren Klassen hinzu und nehmen damit Einfluss auf die Umgebung und Programmorganisation.

⁴Genaugenommen beschreibt nicht jede Methode *einen* Algorithmus: Manche Methoden sind dafür ausgelegt, dass sie niemals terminieren, also zu keinem Ende kommen, und daher auch keine Lösung (im engeren Sinn) liefern. Trotzdem berechnen diese Methoden ständig wiederholt immer wieder etwas Sinnvolles. Man kann solche Methoden gedanklich in mehrere Teile aufspalten und jeden einzelnen Teil als Beschreibung eines Algorithmus betrachten. Wie wir noch sehen werden, kann man im Allgemeinen gar nicht entscheiden, ob eine Berechnung zu einem Ende kommt. Dennoch verwenden wir den Begriff Algorithmus, wenn die Berechnung zumindest manchmal eine Lösung liefern kann.

³Siehe <http://java.sun.com/docs/books/jls/index.html>

Datenstrukturen, Algorithmen, etc. sind großteils *statisch* definiert, das heißt, sie werden in Programmen fix festgelegt und bleiben während der Programmausführung (also *zur Laufzeit*) unverändert. Ausführungen der Algorithmen zur Laufzeit ändern die Daten in den Datenstrukturen jedoch *dynamisch*. Um ein Programm zu verstehen, müssen wir es sowohl auf der statischen als auch dynamischen Ebene verstehen.

Die Grenze zwischen dem statischen und dynamischen Teil hängt von der Sprache und vom *Programmierstil* (das ist die Art und Weise des Programmierens) ab. Java ist eine eher statische Sprache, das heißt, man versucht möglichst viel in den statischen Strukturen abzubilden, sodass man ein Programm alleine aus der statischen Betrachtung schon recht gut kennt. Dynamische Sprachen wie beispielsweise Ruby halten den statischen Anteil dagegen klein, sodass man zur Laufzeit viele Freiheiten hat, aber das Programm auf der statischen Ebene nicht immer so einfach verstehen kann. Obwohl es schwieriger ist als in Ruby, kann man auch in Java recht dynamisch programmieren und vieles zur Laufzeit hin verschieben. Dagegen kann man auch in Ruby einen eher statischen Programmierstil pflegen, indem man auf die Freiheiten der Sprache verzichtet.

In statischen Sprachen spielen *deklarierte Typen* eine große Rolle. Die Werte von Variablen, formalen Parametern und Methodenergebnissen werden in Deklarationen und Definitionen durch Typen statisch sofort sichtbar eingeschränkt, was oft einen guten Anhaltspunkt für ein Verständnis des Zwecks dieser Werte ergibt. Das erhöht die Lesbarkeit der Programme. Weiters ermöglichen diese Typen *statische Typüberprüfungen*, die bestimmte Fehler im Programm ausschließen, indem sie zusichern, dass Operatoren und Operanden immer miteinander kompatibel sind. Beispielsweise ist die Zuweisung `zahl=zahl+grenze` sinnvoll und korrekt, wenn `zahl` und `grenze` beide vom Typ `int` sind, aber nicht, wenn beide Variablen vom Typ `boolean` sind, oder eine vom Typ `int` und die andere vom Typ `boolean`. In dynamischen Sprachen gibt es keine deklarierten Typen; das erspart Arbeit beim Programmieren. Allerdings sind inkompatible (also nicht zusammenpassende) Typen erst zur Laufzeit erkennbar. Programme in statischen Sprachen sind also leichter zu lesen und besser überprüft, die in dynamischen Sprachen leichter zu schreiben und flexibler.

1.4.3 Compiler und Interpreter

Java-Programme sind in der Form, in der sie geschrieben werden, nicht direkt ausführbar. Um das Programm ausführbar zu machen, müssen wir

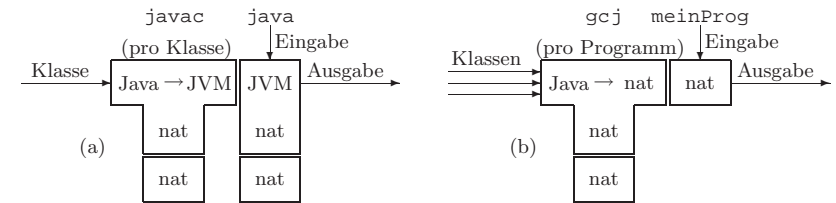


Abbildung 1.14: Vom Code zur Ausführung (a) mit und (b) ohne Zwischencode

darauf einen *Compiler* (auch *Übersetzer* genannt) anwenden, der das Java-Programm in ein JVM-Programm, also ein auf der JVM (Java Virtual Machine, siehe Abschnitt 1.3.2) lauffähiges Programm übersetzt. Für den Java-Compiler ist Java die *Quellsprache* und die Sprache der JVM die *Zielsprache*; entsprechend wird Java-Code als *Quellcode* oder *Source Code* und JVM-Code als *Zielcode* betrachtet. Beim Aufruf des Compilers `javac` kann eine einzelne Klasse oder mehrere Klassen auf einmal angegeben werden. Jedenfalls wird aus jeder Java-Klasse genau eine JVM-Klasse erzeugt, die Klassen werden also getrennt voneinander übersetzt.⁵ Eine JVM-Klasse, die aus einer Java-Klasse mit der speziellen Methode `main` (siehe Abschnitt 1.1.2) erzeugt wurde, ist durch einen *JVM-Interpreter* (z.B. `java`, vereinfacht auch *Java-Interpreter* genannt) ausführbar. JVM-Interpreter implementieren die JVM. Wenn während der Ausführung einer JVM-Klasse auf eine andere JVM-Klasse zugegriffen wird, so wird diese andere Klasse *dynamisch geladen*, also zum ausgeführten Programm hinzugefügt. Auf diese Weise müssen nur solche Klassen im Speicher des Computers stehen, die mindestens einmal im Programm verwendet werden. Diese übliche Vorgehensweise ist in Abbildung 1.14(a) veranschaulicht.

Obige Vorgehensweise verwendet die Sprache der JVM als *Zwischensprache* zwischen der Quellsprache (Java) und der eigentlichen Zielsprache, das ist die natürliche Sprache der realen Maschine (*native language*, hier kurz durch *nat* bezeichnet). Compiler und Interpreter laufen auf dieser Maschine, dargestellt als kleines Kästchen. Ohne Zwischencode kann

⁵Getrennte Übersetzung bedeutet nicht, dass die Klassen unabhängig voneinander übersetzt werden. Wenn eine Klasse auf eine andere zugreift, muss bei der Übersetzung der einen Klasse der Code der anderen vorliegen, damit das Zusammenpassen der Klassen überprüft werden kann. Beispielsweise muss `UnbekannteZahl.java` vor oder gleichzeitig mit `Zahlenraten.java` übersetzt werden.

man ein ganzes Java-Programm, also alle Klassen auf einmal, durch einen Compiler (beispielsweise `gcj`) übersetzen lassen, der eine einzige, direkt ausführbare Datei (beispielsweise `meinProg` genannt) in der natürlichen Sprache der Maschine erstellt. Das ist in Abbildung 1.14(b) veranschaulicht. Zur Ausführung des übersetzten Programms ist kein Interpreter nötig.

In T-Diagrammen wie in Abbildung 1.14 werden Compiler durch T-förmige und Interpreter durch I-förmige Symbole dargestellt, wobei im oberen Teil jeden Symbols Quell- und Zielsprache des Compilers bzw. die vom Interpreter interpretierte Sprache steht. Im unteren Teil steht die Sprache, in welcher der Compiler oder Interpreter implementiert ist. Kleine Kästchen repräsentieren Ausführungen auf realen Maschinen. Symbole unterhalb eines Symbols bedeuten, dass der Compiler oder Interpreter auf einer entsprechenden realen oder abstrakten Maschine ausgeführt wird. Nebeneinander stehende Symbole bedeuten, dass die links erzeugten Programme oder Programmteile rechts verwendet werden.

Die meisten Quellsprachen wurden entwickelt, damit Menschen damit für Menschen gut verständliche Programme schreiben können. Maschinensprachen haben ihren Schwerpunkt in der effizienten Ausführbarkeit durch reale Maschinen, und abstrakte Zwischensprachen suchen einen Kompromiss aus effizienter Ausführung und Portabilität. Compiler sorgen hauptsächlich dafür, dass für Menschen optimierte Programme in für Maschinen optimierte Programme übersetzt werden. Interpreter führen dagegen Programme auf einer abstrakten Maschine aus und entkoppeln die Sprache des Programms von der natürlichen Sprache der realen Maschine.

Früher galt interpretierter Code im Vergleich zu übersetztem als sehr langsam; die Interpretation hatte einen starken negativen Einfluss auf die Ausführungsgeschwindigkeit. Mit heutigen Techniken ist die Interpretation nicht mehr so teuer, und Interpretations- und Compilationstechniken sind zusammengewachsen: Viele aktuelle Interpreter (einschließlich fast aller JVM-Interpreter) haben intern einen für den Anwender unsichtbaren Compiler eingebaut, der häufig ausgeführte Programmteile zur Laufzeit in Maschinencode übersetzt, selten ausgeführte Teile aber interpretiert. Dieser Ansatz heißt *JIT (Just In Time) Compilation*. Er vereint die Vorteile eines Interpreters mit einer beinahe so effizienten Ausführung der Programme wie bei einer Übersetzung im Vorhinein. Von außen gesehen handelt es sich noch immer um einen Interpreter, der portablen Zwischencode ausführen kann. Speziell JVM-Code ist auf fast jeder Maschine ausführbar. Auch bei Verwendung der JIT-Technik werden Klassen dynamisch

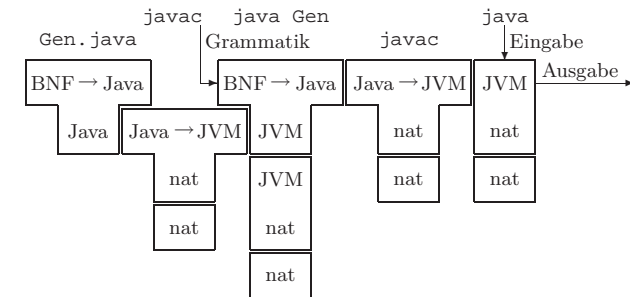


Abbildung 1.15: Beispiel zur kombinierten Verwendung mehrerer Compiler

geladen, und man braucht nicht alle Klassen auf einmal zu übersetzen. Daher werden Zwischencode-Interpreter häufig verwendet.

Oft kommen für spezielle Zwecke spezielle Sprachen zum Einsatz, die für eine Sache sehr gut geeignet ist, für andere aber nicht. Ein Beispiel ist die BNF zur Festlegung einer Grammatik, siehe Abschnitt 1.4.1. Um die Vorteile verschiedener Sprachen miteinander zu kombinieren, geht man manchmal so vor wie in Abbildung 1.15 gezeigt: Wir schreiben in Java ein Programm, das aus einer BNF-Grammatik eine entsprechende Java-Klasse generiert. Es handelt sich dabei um einen Compiler. Zuerst müssen wir ihn durch Anwendung von `javac` auf einem JVM-Interpreter ausführbar machen. Dann erzeugen wir durch Ausführung des BNF-Compilers aus einer Grammatik eine Java-Klasse, die wir wie üblich in JVM-Code übersetzen und ausführen können. Wie dieses Beispiel zeigt, ergeben sich in der Praxis schnell recht viele Übersetzungs- und Interpretationsschritte, die nötig sind, bevor ein Programm laufen kann.

1.4.4 Übersetzung und Ausführung

Ein Interpreter funktioniert im Prinzip genauso wie eine reale Maschine, vergleiche mit Abschnitt 1.3.1: Er verwendet einen PC (program counter), der den nächsten auszuführenden Befehl angibt, und führt in einer Schleife immer wieder folgende Aktionen aus:

- Hole den nächsten auszuführenden Befehl entsprechend dem PC.
- Sorge dafür, dass der PC den danach auszuführenden Befehl angibt.

- Stelle fest, was zur Ausführung des gerade geholten Befehls zu tun ist, und mache das.

In der Klasse `Zahlenraten` in Listing 1.3 sind wir ähnlich vorgegangen: Im Rumpf einer Schleife

- holen wir über `nextInt` die nächste Zahl von der Eingabe,
- wobei gleichzeitig die darauffolgende Zahl zur nächsten Eingabe wird,
- stellen fest, ob die geholte Zahl kleiner oder gleich der zu erratenden Zahl ist, und geben schließlich eine entsprechende Meldung aus.

Auch in einem weiteren Punkt ähnelt die Klasse `Zahlenraten` einem Interpreter oder einer realen Maschine: Es kann vorkommen, dass ein Befehl aus irgendwelchen Gründen nicht ausführbar ist, so wie beim Zahlenraten vorkommen kann, dass eine Eingabe keine gültige Zahl darstellt. Beim Zahlenraten können wir eine falsche Eingabe einfach ignorieren und mit der nächsten Eingabe weitermachen. Aber ein Interpreter darf solche *Laufzeitfehler* nicht so einfach ignorieren. Die folgenden Befehle können ja davon abhängen, was vorher passiert ist. Eine Möglichkeit besteht darin, die Programmausführung abubrechen und, sofern der Fehler das noch erlaubt, eine Fehlermeldung auszugeben. Einen solchen unvorhergesehenen Programmabbruch nennt man *Programmabsturz*. Weitere Möglichkeiten werden wir in Kapitel 5 kennenlernen.

Fehler in einem Programm können auch während der Übersetzung durch einen Compiler erkannt werden. Der Compiler wird in diesem Fall statt des übersetzten Codes nur eine Liste von *Fehlermeldungen* (engl. *error messages*) ausgeben. Es ist wünschenswert, dass bereits der Compiler möglichst viele Fehler in einem Programm erkennt, damit diese Fehler nicht erst zur Laufzeit auftreten und das Programm zum Absturz bringen können. In Sprachen wie Java macht der Compiler zahlreiche statische Typüberprüfungen, wobei das Wort *statisch* sich darauf bezieht, dass etwas *vor der Ausführung* passiert, während *dynamisch* sich auf etwas *zur Laufzeit* bezieht. Die meisten Typfehler werden statisch abgefangen, siehe Abbildung 1.16. Sprachen, die nur interpretiert und nicht übersetzt werden, bieten keine Möglichkeit zur statischen Typüberprüfung.

Manchmal erkennt der Compiler eine Situation im Programm, die unüblich ist und auf einen Fehler hindeuten könnte, aber vielleicht tatsächlich doch keinen Fehler darstellt. In diesen Fällen wird der Compiler das

```
UnbekannteZahl.java:17: incompatible types
found   : boolean
required: int
        return (zahl == vergleichszahl);
                ^
1 error
```

Abbildung 1.16: Fehlermeldung die entsteht, wenn als Ergebnistyp von `gleich` in `UnbekannteZahl` (Listing 1.2) `int` statt `boolean` verwendet wird

Programm zwar richtig übersetzen, aber dennoch eine *Warnung* (engl. *warning*) ausgeben. Warnungen sind bei der Fehlersuche hilfreich.

Kein Compiler kann alle Fehler erkennen. Auch muss nicht jeder Fehler zur Laufzeit erkannt werden und zum Absturz führen. Viele Fehler bewirken nur, dass ein Programm sich nicht so verhält, wie wir es uns wünschen. Diese Fehler sind nur durch das *Testen* des Programms herauszufinden.

Die Hauptaufgabe eines Compilers besteht darin, Quellcode in semantisch äquivalenten Zielcode zu übersetzen. Die Semantik muss erhalten bleiben, obwohl sich die Syntax ändert. Auch die Pragmatik ändert sich, da die Pragmatik des Quellcodes auf der Ebene des Zielcodes meist keine Rolle spielt. Wir unterscheiden zwischen der *statischen Semantik* und der *dynamischen Semantik*: Die statische Semantik beschreibt, welche Fälle zu einem übersetzten Programm und welche nur zu Fehlermeldungen führen. Die dynamische Semantik beschreibt dagegen, wie sich das übersetzte Programm zur Laufzeit verhält. Quell- und Zielcode müssen nur hinsichtlich der dynamischen Semantik äquivalent sein, da sich ausschließlich der Compiler um die statische Semantik kümmert.

Eine weitere Aufgabe des Compilers besteht in *Optimierungen* des Programms. Der Zielcode entspricht semantisch dem Quellcode, aber wenn es mehrere Möglichkeiten gibt, etwas semantisch äquivalent auszudrücken, dann wählt der Compiler die Variante, die effizienter (also mit weniger Aufwand) ausführbar ist. Beispielsweise wird statt dem Ausdruck `3+4` der einfachere, semantisch äquivalente Ausdruck `7` verwendet. Man spricht von Programmoptimierungen, obwohl der Zielcode nur selten optimal ist. Es würde viel zu lange dauern, einen auf gewisse Weise optimalen Code zu ermitteln, selbst wenn man weiß, welche Aspekte zu optimieren sind. Man möchte beispielsweise kurze Laufzeiten, geringen Speicherbedarf, kurze Antwortzeiten, geringe Netzwerkbelastungen, etc. haben. Alles auf ein-

mal geht jedoch nicht. Beispielsweise kann man kurze Laufzeiten häufig auf Kosten des Speicherbedarfs erhalten und umgekehrt, aber nicht beides gleichzeitig. Compiler versuchen daher, einen guten Kompromiss zu finden, und optimierter Code ist in vielen Aspekten meist recht gut. Oft bieten Compiler Konfigurationsmöglichkeiten, über die man die wichtigsten Optimierungsziele beeinflussen kann. Man kann Optimierungen als eine Form der Pragmatik auf dem Zielcode betrachten.

Optimierungen durch den Compiler steigern die Effizienz eines Programms unter Beibehaltung der Semantik meist nur geringfügig. Wesentlich stärker wird die Effizienz durch die Wahl geeigneter Algorithmen bei der Erstellung des Programms beeinflusst.

1.5 Denkweisen

Es ist eine alte Weisheit, dass unser Denken unsere Sprache beeinflusst, aber umgekehrt auch unsere Sprache unser Denken. Auf gewisse Weise trifft das auch auf formale Sprachen und insbesondere Programmiersprachen zu. Viele Konzepte in Programmiersprachen haben ihren Ursprung in reinen Gedankenmodellen. Andererseits bestimmen diese Konzepte zu einem großen Teil, wie wir die Welt betrachten und in Software umsetzen. Auch losgelöst von einer realen Maschine unterwerfen sich unsere Gedanken gewissen Strukturen.

1.5.1 Sprachen, Gedanken und Modelle

Sprache dient dem Austausch von Gedanken. Der Sprecher teilt dem Angesprochenen seine Gedanken mit. Auf das Programmieren übertragen bedeutet das, dass wir uns überlegen, wie unsere Maschine eine gestellte Aufgabe lösen kann, und die Ergebnisse dieser Überlegungen der Maschine in Form eines Programms mitteilen. Im Gegensatz zur menschlichen Kommunikation ist die Kommunikation mit der Maschine sehr einseitig, da die Maschine keine eigenen Überlegungen anstellen kann. Als Antwort bekommen wir höchstens Fehlermeldungen oder Ergebnisse irgendwelcher Berechnungen zurück, die jedoch nur auf unseren eigenen Gedanken beruhen. Menschen können durch den Austausch von Gedanken gemeinsame Lösungen entwickeln. Maschinen fehlt dazu etwas Entscheidendes. Sie entwickeln keine eigenen Gedanken, sondern führen nur die von uns vorgegebenen Programme aus.

Tatsächlich ist das Programmieren keine so einseitige Form der Kommunikation: Programme halten vor allem auch Gedanken fest, die zwischen Menschen ausgetauscht werden. Nur selten wird Software von einer Person alleine entwickelt und gepflegt. Fast immer sind mehrere Personen beteiligt, die nicht nur die Software und die Maschine genau kennen, sondern jeweils auch die Gedanken der anderen verstehen müssen. Diese Gedanken sind oft auf der Ebene winzigster Details angesiedelt. Programme können Details recht präzise ausdrücken und sind darin Texten in natürlicher Sprache überlegen. Sogar wenn man alleine Software entwickelt, kann man die eigenen Gedanken als Notizen in Form eines Programms festhalten und später wieder in Erinnerung rufen.

Wie bereits in Abschnitt 1.3.2 erwähnt, steckt hinter jeder Programmiersprache ein Berechnungsmodell. Gedanken, die wir in der Sprache ausdrücken, beziehen sich auf dieses Modell. Das Modell hat natürlich Auswirkungen auf unser Denken. Wir organisieren unsere Gedanken so, dass sie mit dem Modell in Einklang sind. Praktisch durchgesetzt haben sich nur Modelle, die sowohl *einfach* als auch *vollständig* sind. Das Modell selbst soll einfach verständlich sein, das Wichtigste soll sich einfach ausdrücken lassen, aber auch alles noch so Komplexes soll irgendwie ausgedrückt werden können. Wer Programmieren lernt, lernt vor allem, im Modell hinter der Sprache zu denken. Es gibt zwar Tausende von Programmiersprachen, aber die Modelle dahinter ähneln einander stark.

InformatikerInnen zeichnen sich dadurch aus, dass sie in abstrakten Modellen denken können. Erst durch *Abstraktion über Details*, das heißt, durch Vernachlässigung unwichtiger Nebensächlichkeiten und Konzentration auf die wichtigsten Aspekte, können wir schwierige und umfangreiche Aufgaben lösen. Wir verwenden also Sprachen und denken in Modellen, die uns die Konzentration auf das Wesentliche erlauben. Diese Sprachen können, müssen aber keine Programmiersprachen sein. Sie sollen dennoch bis zu einem gewissen Grad formal sein, um Missverständnissen vorzubeugen. Einige Sprachen sind grafisch, sie drücken also etwas in Form von Zeichnungen oder Diagrammen aus, andere sind wie die meisten Programmiersprachen textuell, und wieder andere haben sowohl grafische als auch textuelle Darstellungsformen. Im Laufe des Informatikstudiums lernt man zahlreiche solche Modelle kennen. Beispiele dafür sind verschiedene Arten von Grammatiken und Automaten (aus der Automatentheorie) und das Entity Relationship Modell zur Modellierung von Beziehungen zwischen Daten. Bekannt ist vor allem *UML (Unified Modelling Language)*, eine standardisierte grafische Sprache für viele, vor allem in der objektorien-

tierten Softwareentwicklung häufig verwendete Modelle.

Programmiersprachen sind dafür ausgelegt, sowohl Details präzise auszudrücken als auch über Details zu abstrahieren. Bereits in Abschnitt 1.1.1 haben wir den Begriff der Datenabstraktion (als Datenkapselung zusammen mit data hiding) eingeführt. Datenabstraktion erlaubt uns, Objekte als abstrakte Einheiten zu sehen, die Objekte aus der realen Welt simulieren und daher intuitiv gut verständlich sind. Durch genau beschriebene Objektschnittstellen bleibt die Präzision trotz Abstraktion erhalten. Abstraktion bildet also keinen Gegensatz zur Präzision.

Der Begriff der *Funktion* hat in fast allen Programmiersprachen eine zentrale Bedeutung. Darunter verstehen wir im Wesentlichen eine mathematische Funktion, also eine Abbildung von einer Wertemenge auf eine andere Wertemenge. Funktionen in Programmiersprachen sind *intensional* spezifiziert, das heißt, sie beschreiben einen Algorithmus zur Berechnung des Funktionsergebnisses. Dieses wird durch *Ausführung* der Funktion und damit des entsprechenden Algorithmus berechnet. In Java übernehmen Methoden die Rolle von Funktionen, obwohl sich Methoden in einigen Aspekten von Funktionen unterscheiden. Auch Begriffe wie *Prozedur* oder *Routine* werden für funktionsähnliche Sprachelemente verwendet. Eine Gemeinsamkeit besteht darin, dass bei Ausführung (gleichbedeutend: bei einer Anwendung, nach einem Aufruf, nach dem Senden und Empfangen einer Nachricht) die spezifizierten Berechnungsschritte durchgeführt werden. Jedoch müssen Methoden, Prozeduren und Routinen nicht notwendigerweise Ergebnisse zurückliefern, wie beispielsweise die Methode `main` in Java. Ergebnisse können, anders als bei mathematischen Funktionen, nicht nur von den Parametern abhängen, sondern auch von Eingaben und Werten in den Variablen von Objekten. Außerdem kann es *Seiteneffekte* geben, das heißt, die Berechnungsschritte können Ausgaben machen und neue Werte an Variablen zuweisen. Wenn man Funktionen ohne Seiteneffekte meint, deren Ergebnisse nur von den Parametern abhängen, dann spricht man von *reinen Funktionen* (engl. *pure functions*).

Nach der wichtigsten Form der Abstraktion kann man Programmiersprachen in mehrere Kategorien einteilen:

Imperative Sprachen: Der Name kommt von der Befehlsform, in der man einer Maschine Anweisungen gibt. An oberster Stelle steht die Möglichkeit, neue Werte an Variablen zuzuweisen. Durch Befehle und Variablen ist die Rechnerarchitektur hinter dem Berechnungsmodell deutlich erkennbar. Man unterscheidet folgende Varianten:

Prozedurale Sprachen verwenden zur Abstraktion im Wesentlichen Prozeduren mit Seiteneffekten. Reine Funktionen reichen nicht aus, da Berechnungsfortschritte nur über Zuweisungen erfolgen.

Objektorientierte Sprachen stellen eine Erweiterung prozeduraler Sprachen dar. Objekte und Datenabstraktionen werden wichtiger als die Abstraktion über Prozeduren.

Deklarative Sprachen: Berechnungsmodelle sind nahe an mathematische Modelle angelehnt. Elemente von Rechnerarchitekturen sind kaum erkennbar. Es gibt keine Zuweisungen, die den alten Wert einer Variablen überschreiben würden.

Funktionale Sprachen verwenden zur Abstraktion reine Funktionen. Das Fehlen von Seiteneffekten erleichtert das Verständnis eines Programms in gewisser Weise. Andererseits erfordern Ein- und Ausgaben eine bestimmte Strukturierung des Programms.

Logikorientierte Sprachen haben ihre Wurzeln in der mathematischen Logik, und die Programmausführung ist gleichbedeutend mit dem Beweis einer logischen Aussage. Trotzdem ähnelt die Programmausführung der eines prozeduralen Programms, jedoch werden Werte nur an freie Variablen zugewiesen, nicht an Variablen, die bereits Werte haben.

Wegen des starken Einflusses der Abstraktionen auf Denkweise und Programmierstil entsprechen diese Kategorien den wichtigsten *Programmierparadigmen*. Es ist leicht zu sehen, dass Funktionen und funktionsähnliche Sprachelemente, direkt oder indirekt, in jedem Paradigma vorkommen.

1.5.2 Der Lambda-Kalkül

Der in den 1930er Jahren von Church und Kleen entwickelte Lambda-Kalkül, benannt nach dem griechischen Buchstaben λ , bildet eine formale Basis für Funktionen und daher für praktisch alle Programmiersprachen. Wegen seiner Wichtigkeit wollen wir diesen Kalkül näher betrachten.

Zunächst definieren wir die Menge der λ -Ausdrücke *Exp*, die im Kalkül eine Bedeutung haben – die syntaktisch richtigen Ausdrücke. Als Basis dafür verwenden wir eine unendliche Menge *V* von *Variablen*. Anders als in Java können wir keine Werte an diese Variablen zuweisen. Sie haben eher die Bedeutung von Variablen in der Mathematik, stehen also für

unbekannte Werte. Folgende Ausdrücke kommen in Exp vor:

Variable:	$v \in Exp$ wenn $v \in V$
Funktionsanwendung:	$f e \in Exp$ wenn $f \in Exp$ und $e \in Exp$
Funktionsabstraktion:	$\lambda v.f \in Exp$ wenn $v \in V$ und $f \in Exp$

Im λ -Ausdruck $f e$ wird f als Funktion aufgefasst, die auf das Argument e angewandt wird. Der Ausdruck $\lambda v.f$ definiert eine Funktion, wobei v dem formalen Parameter (ohne Typ) und f dem Ergebnis der Funktion entspricht. Beispielsweise ist $\lambda v.v$ die Identitätsfunktion, die nur das Argument zurückgibt. Runde Klammern wie in $(\lambda v.f) e$ sind nicht Teil der Syntax, sondern werden beliebig verwendet, um die Struktur der λ -Ausdrücke zu verdeutlichen. Üblicherweise werden wir Elemente der Menge V durch u, v, w, \dots bezeichnen und Elemente von Exp durch e, f, g, \dots .

Man sagt, eine Variable in einem Ausdruck ist *gebunden*, wenn sie nur in Funktionen vorkommt, die diese Variable als formale Parameter verwenden. Gebundene Variablen stehen für die Argumente, auf welche die Funktionen angewandt werden. Eine Variable kommt in einem Ausdruck *frei* vor, wenn sie nicht gebunden ist. Freie Variablen stehen für beliebige Ausdrücke. Beispielsweise ist v in $\lambda v.(u v)$ gebunden und u ist frei. Mit $FV(e)$ bezeichnen wir die Menge aller in einem λ -Ausdruck e frei vorkommenden Variablen:

$$\begin{aligned} FV(v) &= \{v\} & \text{wobei } v \in V \\ FV(f e) &= FV(f) \cup FV(e) \\ FV(\lambda v.f) &= FV(f) \setminus \{v\} & (v \text{ ist nicht in der Menge}) \end{aligned}$$

Zur Definition der Semantik des Kalküls benötigen wir den Begriff der *Ersetzung*: Ein Ausdruck $[e/u]f$ (gesprochen: „ e ersetzt u in f “) steht für den λ -Ausdruck, der entsteht, wenn man im λ -Ausdruck f jedes freie Vorkommen der Variablen u durch den λ -Ausdruck e ersetzt:

$$\begin{aligned} [e/u]v &= \begin{cases} e & \text{wenn } u = v \\ v & \text{wenn } u \neq v \text{ und } v \in V \end{cases} \\ [e/u](f g) &= ([e/u]f) ([e/u]g) \\ [e/u](\lambda v.f) &= \begin{cases} \lambda v.f & \text{wenn } u = v \\ \lambda v.[e/u]f & \text{wenn } u \neq v \text{ und } v \notin FV(e) \\ \lambda w.[e/u][w/v]f & \text{sonst, wobei } u \neq w \neq v \text{ und } w \notin FV(f e) \end{cases} \end{aligned}$$

Die Ersetzung auf Funktionsabstraktionen (dritte Regel) bedarf einiger Erklärungen: Wenn die Variable u , die ersetzt werden soll, gleich dem

formalen Parameter v ist, brauchen wir nichts machen, da diese Variable in der Funktion ja nicht frei, sondern nur gebunden vorkommt. Wenn sich die Variablen unterscheiden ($u \neq v$), müssen wir die Ersetzung auch im Ergebnis vornehmen. Dabei darf aber kein Konflikt zwischen dem formalen Parameter v und frei in e vorkommenden Variablen auftreten, es muss also $v \notin FV(e)$ gelten. Wenn es doch einen Konflikt gibt (letzte Alternative), müssen wir den formalen Parameter umbenennen. Das heißt, wir müssen eine andere Variable w als formalen Parameter verwenden, die sich sowohl von u als auch v unterscheidet, und die weder in f noch in e frei vorkommt. Beispielsweise ergibt $[v/u]\lambda v.((v u) u)$ einen Ausdruck $\lambda w.((w v) v)$.

Drei einfache Regeln bestimmen die Semantik des Lambda-Kalküls:

$$\begin{aligned} \alpha\text{-Konversion:} & \quad \lambda v.f \leftrightarrow \lambda u.[u/v]f \quad \text{wobei } u \notin FV(\lambda v.f) \\ \beta\text{-Konversion:} & \quad (\lambda v.f) e \leftrightarrow [e/v]f \\ \eta\text{-Konversion:} & \quad \lambda v.(f v) \leftrightarrow f \quad \text{wobei } v \notin FV(f) \end{aligned}$$

Über diese Regeln wird eine Äquivalenz zwischen λ -Ausdrücken definiert. Zwei λ -Ausdrücke e_0 und e_n sind äquivalent (also quasi gleich) wenn es möglich ist, durch beliebig oft wiederholte Anwendungen der Regeln e_0 in e_n umzuformen, das heißt, wenn es λ -Ausdrücke e_1, \dots, e_{n-1} gibt, sodass nach obigen Regeln $e_i \leftrightarrow e_{i+1}$ oder $e_{i+1} \leftrightarrow e_i$ für alle $0 \leq i < n$ gilt. Die Konversions-Regeln sind nicht gerichtet. Sie sind von links nach rechts genauso anwendbar wie von rechts nach links. Äquivalenz verwenden wir als Basis für Berechnungen: Wir suchen nach einem Ergebnis, das äquivalent zur gestellten Aufgabe, aber so stark vereinfacht wie möglich ist. Der Ausdruck rechts vom Pfeil ist für β - und η -Konversionen einfacher als der links vom Pfeil. Wenn wir einen λ -Ausdruck *reduzieren*, also vereinfachen wollen, wenden wir diese Regeln nur von links nach rechts an:

$$\begin{aligned} \beta\text{-Reduktion:} & \quad (\lambda v.f) e \mapsto [e/v]f \\ \eta\text{-Reduktion:} & \quad \lambda v.(f v) \mapsto f \quad \text{wobei } v \notin FV(f) \end{aligned}$$

Die α -Konversion (ausgesprochen: „alpha-Konversion“) heißt auch *Umbenennung*. Formale Parameter dürfen beliebig umbenannt werden (wobei die Umbenennungen im ganzen Ausdruck auf gleiche Weise erfolgen müssen), solange es dabei zu keinen Namenskonflikten kommt. Beispielsweise gilt $\lambda u.u \leftrightarrow \lambda v.v$. Aufgrund der α -Konversion können wir Umbenennungen, die in der letzten Alternative in der Definition von Ersetzungen notwendig sind, jederzeit auch ohne zwingenden Grund durchführen. Umbenennungen haben keine Richtung und vereinfachen nichts. Deshalb gibt es auch keine α -Reduktion, sondern nur eine α -Konversion.

Die β -Reduktion („beta-Reduktion“) bzw. *Funktionsanwendung* ist die wichtigste Regel: Das Ergebnis einer Funktionsanwendung ist der Ausdruck rechts vom Punkt, wobei jedes freie Vorkommen des formalen Parameters durch den aktuellen Parameter ersetzt ist. Zum Beispiel wird $(\lambda v.v) e$ zu e reduziert.

Die η -Reduktion („eta-Reduktion“) oder *Erweiterungs-Regel* spielt nur eine untergeordnete Rolle. Beispielsweise kann $(\lambda v.(fv)) e$ mit $v \notin FV(f)$ sowohl durch β -Reduktion als auch durch η -Reduktion zum Ausdruck fe reduziert werden. Die η -Reduktion verlangt einen Funktionsrumpf in einer Form, in der das Ergebnis nicht vom Argument abhängt, und braucht daher kein Argument, während die β -Reduktion immer ein Argument haben muss. Wenn beide Regeln anwendbar sind, liefern sie dasselbe Ergebnis.

Ein λ -Ausdruck ist in *Normalform*, wenn darauf weder eine β - noch η -Reduktion anwendbar ist. Beispiele sind $\lambda v.v$ und uv . Ausdrücke in Normalform entsprechen den Endergebnissen von Berechnungen.

1.5.3 Eigenschaften des Lambda-Kalküls

Der Lambda-Kalkül hat einige Eigenschaften, die ihn als Grundlage für Programmiersprachen sehr wertvoll machen. Einer davon ist die Einfachheit. Wir brauchen tatsächlich nicht mehr als eine Konversions-Regel und zwei Reduktions-Regeln, um die Semantik einer Programmiersprache zu beschreiben. Hier sind einige weitere wichtige Eigenschaften:

Vollständigkeit: Alles, was berechenbar ist, ist auch mittels Lambda-Kalkül berechenbar. Der Lambda-Kalkül ist Turing-vollständig. Leider sind gerade viele einfache Operationen, beispielsweise Additionen von ganzen Zahlen, in diesem Kalkül nur sehr umständlich ausdrückbar. Daher verwendet man häufig kombinierte Systeme, in denen einfache Berechnungen über andere Formalismen erfolgen und kompliziertere Fälle den λ -Ausdrücken vorbehalten sind.

Endlos-Reduktionen: Nicht zu jedem λ -Ausdruck gibt es einen äquivalenten Ausdruck in Normalform. Das bedeutet, manchmal sind β -Reduktionen endlos wiederholt anwendbar. Beispielsweise führt eine β -Reduktion von $(\lambda v.(vv)) (\lambda v.(vv))$ wieder zu genau demselben Ausdruck. Solche endlosen Reduktionen sind in Turing-vollständigen Systemen, nicht nur im Lambda-Kalkül, prinzipiell nicht vermeidbar.

Reihenfolge von Reduktionen: Es spielt fast keine Rolle, in welcher Reihenfolge wir die Regeln anwenden. Beispielsweise können wir zuerst

Argumente reduzieren, oder zuerst die äußerste Funktion. Das Ergebnis, das heißt, die berechnete Normalform hängt nicht davon ab, abgesehen von möglichen Umbenennungen formaler Parameter. Falls eine Berechnung zu einem Ergebnis führt, dann führt jede einigermaßen gerechte Reihenfolge der Anwendung von Reduktions-Regeln zum selben Ergebnis. Eine Ausnahme bilden Ausdrücke, in denen ein Teilausdruck, der im Ergebnis gar nicht vorkommt, endlos reduzierbar ist, beispielsweise $(\lambda u.w) ((\lambda v.(vv)) (\lambda v.(vv)))$. Hier führt eine einzige Anwendung der β -Reduktion auf der ersten Funktionsabstraktion zur Normalform w , während wir kein Ergebnis erhalten, wenn wir stets nur das Argument $(\lambda v.(vv)) (\lambda v.(vv))$ reduzieren.

Funktionen erster Ordnung: Funktionen werden wie Daten behandelt und sind dadurch *Elemente erster Ordnung* (engl. *first class entities*). Funktionen können als Argumente verwendet und als Ergebnisse zurückgegeben werden, so wie im λ -Ausdruck $(\lambda v.v) (\lambda u.e)$.

Keine Kontrollstrukturen: In allen imperativen Sprachen spielen *Kontrollstrukturen* wie bedingte Anweisungen und Schleifen eine große Rolle. Der Lambda-Kalkül kommt ohne Kontrollstrukturen aus, da sie durch Funktionen erster Ordnung ersetzt werden können.

Currying: Eine Funktion wird im Lambda-Kalkül immer nur auf ein einziges Argument angewandt. Durch Funktionen erster Ordnung ist das keine Einschränkung: Zum Beispiel ist $((\lambda u.(\lambda v.(vu))) e) f$ durch eine β -Reduktion zu $(\lambda v.(ve)) f$ und durch eine weitere zu fe reduzierbar. Wir können $\lambda u.(\lambda v.(vu))$ als Funktion mit zwei Parametern betrachten (die wir auf die beiden Argumente e und f angewendet haben), oder gleichbedeutend als eine Funktion, die durch Anwendung auf ein Argument eine (auf ein weiteres Argument anwendbare) Funktion zurückgibt. Diese Technik nennt man *Currying*.

Der Lambda-Kalkül erfüllt seine Aufgabe als Berechnungsmodell hervorragend. Aber λ -Ausdrücke sind kaum lesbar, und ohne Erweiterungen gibt es keine Möglichkeit zur Beschreibung der Datenstrukturen und Programmorganisation. In der Praxis wünschen wir uns zumindest einfache Kontrollstrukturen, mehr Kontrolle (z.B. indem wir die Reihenfolge der Reduktionen bestimmen können), benannte Funktionen statt namenloser Funktionsabstraktionen und vielleicht auch ein statisches Typsystem. Der Kalkül lässt sich entsprechend erweitern. Leider verlieren wir dadurch et-

was, nämlich die einfache Analysierbarkeit. Praxistaugliche Programmiersprachen sind immer viel komplexer als einfache formale Modelle.

Formale Modelle verraten uns viel über Sprachen, z.B. Folgendes:

Entscheidbarkeit: Man kann formal nachweisen, dass nicht alle Probleme entscheidbar, also lösbar sind. Andererseits ist es gar nicht schwer, Turing-vollständige Systeme wie den Lambda-Kalkül zu entwickeln, in denen alle bisher als entscheidbar bekannten Probleme lösbar sind. In jedem solchen System sind aber auch unentscheidbare Probleme ausdrückbar. Im Lambda-Kalkül äußert sich das durch Endlosreduktionen, die niemals zu einer Normalform führen.

Unentscheidbarkeit des Halteproblems: Ein bekanntes unentscheidbares Problem ist das Halteproblem: Im Allgemeinen ist nicht entscheidbar, ob wiederholte β -Reduktionen jemals zu einer Normalform führen oder nicht. Genau aus diesem Grund können wir weder den Lambda-Kalkül noch irgendein anderes Turing-vollständiges System oder eine Programmiersprache so einschränken, dass nur entscheidbare Probleme ausdrückbar sind. Wenn wir Endlosreduktionen vermeiden, geht die Vollständigkeit verloren.

Genaugenommen ist das Halteproblem, wie auch viele anderen Probleme, *halbentscheidbar*, das heißt, in manchen Fällen ist das Problem entscheidbar, in anderen nicht. Manchmal wissen wir, dass eine Berechnung terminiert (also eine Normalform nach endlich vielen Reduktionen gefunden wird), manchmal wissen wir, dass eine Berechnung niemals terminiert, und manchmal können wir weder das eine noch das andere feststellen.

In der Programmierpraxis ist die Unentscheidbarkeit des Halteproblems nur selten von Bedeutung. Bei der Programmierung bemühen wir uns ja, nur Algorithmen einzusetzen, die nach relativ kurzer Zeit terminieren. Dabei stoßen wir nur selten an die Grenzen der Entscheidbarkeit. Scheinbar endlose Berechnungen (ob sie tatsächlich endlos sind, wissen wir ja oft nicht) deuten eher darauf hin, dass das Programm fehlerhaft ist als dass wir ein unentscheidbares Problem lösen wollen.

1.5.4 Zusicherungen und Korrektheit

Beim Programmieren passieren leicht Fehler. Ursachen dafür sind vielfältig: Fehler im Verständnis der logischen Zusammenhänge, Kommunikationsfehler zwischen Personen, aus Unachtsamkeit oder Zeitdruck nur un-

Listing 1.17: Zusicherungen als Kommentare (Code aus Listing 1.2)

```
zahl = (new Random()).nextInt() % grenze;
// -grenze < zahl < grenze    (wegen ... % grenze)
if (zahl < 0) {
    // -grenze < zahl < 0    (wegen Bedingung zahl < 0)
    zahl = zahl + grenze;
    // 0 < zahl < grenze    (wegen Addition von grenze)
}
// 0 < zahl < grenze    (wenn Bedingung wahr war)
// oder 0 <= zahl < grenze    (wenn Bedingung falsch war)
// ergibt: 0 <= zahl < grenze
```

vollständig durchgeführte oder vergessene Änderungen, Unübersichtlichkeit großer Systeme, und so weiter. Wir müssen etwas gegen die wichtigsten Fehlerursachen tun. Gegen die Unübersichtlichkeit können wir vorgehen, indem wir ein großes System in übersichtlichere Teile zerlegen. Kommunikationsfehler und Fehler im logischen Verständnis können wir vermindern, indem wir unsere Intentionen im Programm klar machen. Dabei helfen uns Typen und Kommentare. Sie unterstützen uns dabei, das schwer fassbare dynamische Verhalten des Programms auf die einfacher verständliche statische Ebene zu bringen. Gut gewählte Namen und Analogien zur realen Welt verbessern ebenfalls die statische Verständlichkeit.

Zusicherungen beschreiben relevante Ausschnitte aus dem erwarteten Zustand eines Objekts oder Systems an der richtigen Stelle im Programm auf systematische Weise. Im einfachsten Fall verwenden wir Kommentare als Zusicherungen zwischen Anweisungen. Im Beispiel in Listing 1.17 steht `<=` für „kleiner oder gleich“. Auf diese Weise können wir einfach verstehen, in welchem Bereich der Wert von `zahl` liegt, ohne den dynamischen Programmablauf nachvollziehen zu müssen. Leider wissen wir nicht, ob der Inhalt der Kommentare stimmt. Er wird ja nirgends überprüft.

Wenn wir Überprüfungen haben möchten, verwenden wir statt der Kommentare `assert`-Anweisungen wie in Listing 1.18. Hier steht `&&` für die logische UND-Verknüpfung. Die Bedingungen in den `assert`-Anweisungen werden bei entsprechenden Interpretereinstellungen zur Laufzeit jedes Mal überprüft, wenn diese Stellen im Programm ausgeführt werden. Falls eine Zusicherung nicht erfüllt ist, tritt ein Laufzeitfehler auf. Allerdings ist nicht garantiert, dass eine falsche Zusicherung gleich erkannt wird. Wenn

Listing 1.18: Zusicherungen als `assert`-Anweisungen

```

zahl = (new Random()).nextInt() % grenze;
assert((-grenze < zahl) && (zahl < grenze));
if (zahl < 0) {
    assert((-grenze < zahl) && (zahl < 0));
    zahl = zahl + grenze;
    assert((0 < zahl) && (zahl < grenze));
}
assert((0 <= zahl) && (zahl < grenze));

```

wir beispielsweise statt `0<=zahl` in der letzten Zeile die zu strenge Zusicherung `0<zahl` machen würden, müssten wir dieses Programmstück oft wiederholt ausführen, bis `zahl` zufällig einmal den Wert 0 bekommt und der Fehler auffällt. Mit einigen wenigen Testdurchläufen ist dieser Fehler nicht zu entdecken. Wenn man aufmerksam ist, können solche Fehler schon beim Hinschreiben der Zusicherungen – gleichgültig ob als `assert`-Anweisung oder Kommentar – auffallen. Also auch ohne Überprüfung tragen Zusicherungen zur Fehlervermeidung bei. Der wichtigste Beitrag von Zusicherungen zur Fehlervermeidung besteht darin, dass wir uns beim Programmieren überlegen müssen, ob die Zusicherungen halten können. Ohne schriftlich festgehaltene Zusicherungen vergessen wir leicht darauf.

Besonders wichtig sind Zusicherungen dort, wo man den Programmablauf nicht anhand weniger Anweisungen nachvollziehen kann. Das trifft auf Schnittstellen zwischen Programmteilen zu, besonders auf Schnittstellen von Funktionen und Ähnlichem. Beispielsweise können wir auf dem Konstruktor von `UnbekannteZahl` folgende Zusicherungen haben:

Listing 1.19: Kommentare als Zusicherungen auf Konstruktor

```

// Initialisierung mit Zufallszahl x; 0 <= x < grenze
// Voraussetzung: grenze > 0
public UnbekannteZahl (int grenze) { ... }

```

Die Zusicherungen sollen den Konstruktor so beschreiben, dass man den Code im Rumpf gar nicht kennen muss um zu verstehen, was er macht. Wir verlassen uns eher auf Kommentare als auf den Code. Zum Teil könnten wir auch für Schnittstellenbeschreibungen `assert`-Anweisungen (innerhalb des Rumpfes) verwenden, aber nicht für alles. Es wäre kaum möglich, in einer `assert`-Anweisung festzulegen, dass die Initialisierung mit einer Zufallszahl erfolgt. Mit einem Kommentar ist das einfach.

Es gibt zumindest zwei Arten von Zusicherungen auf Schnittstellen:

Vorbedingung: Diese Bedingung muss erfüllt sein, bevor die Funktion, Methode, etc. ausgeführt werden kann. Das betrifft vor allem Einschränkungen auf formalen Parametern. Im Beispiel wäre das die Bedingung `grenze>0`. Die Vorbedingung muss bereits bei der Anwendung einer Funktion bzw. beim Senden einer Nachricht erfüllt sein. Die Funktion, Methode, etc. hat selbst keine Möglichkeit, dafür zu sorgen, dass die Vorbedingung erfüllt ist.

Nachbedingung: Eine Nachbedingung muss während der Ausführung der Funktion, Methode, etc. erfüllt werden. Die erste Kommentarzeile im Beispiel ist eine Nachbedingung. Nach Ausführung muss die Initialisierung mit einer Zufallszahl innerhalb der Grenzen erfolgt sein.

Gerade in der objektorientierten Programmierung ist der Rumpf der ausgeführten Methoden sehr oft unbekannt, sodass solche Zusicherungen die einzige Möglichkeit darstellen, um das Verhalten zu beschreiben.

Zusicherungen kann man auch verwenden, um die Semantik einzelner Elemente in Programmiersprachen zu definieren. Bekannt ist der nach C.A.R. Hoare benannte *Hoare-Kalkül*. Ausdrücke in diesem Kalkül haben die Form $\{P\} S \{Q\}$, wobei die Vorbedingung P und die Nachbedingung Q Ausdrücke aus der Prädikatenlogik sind und S eine Anweisung (Statement) ist. Durch sorgfältige Wahl von P und Q wird eine manchmal recht genaue Spezifikation der Semantik von S über mathematisch einfach handhabbare Mittel erreicht. Es gibt auch einige andere Techniken zur Spezifikation der Semantik, die alle Vor- und Nachteile haben.

1.6 Softwareentwicklung

Die Programmierung ist ein wichtiger Teil der Softwareentwicklung. Wir wollen nun das Umfeld beschreiben, in dem die Programmierung zum Einsatz kommt, sowie einige Ziele, die wir in der Programmierung anstreben.

1.6.1 Softwarelebenszyklus

Jede Software hat einen *Lebenszyklus*, der bei der ersten Idee beginnt und mit der letzten Anwendung endet. Dazwischen liegt die *Entwicklung* (*development*), *Wartung* (*maintenance*) und *Anwendung* (*use*) der Software. Folgende Entwicklungsschritte bzw. -phasen werden unterschieden:

Analyse (analysis): In dieser Phase wird die Aufgabe, die durch die zu entwickelnde Software gelöst werden soll, analysiert. Meist ist die Aufgabe anfangs nur grob umrissen, und es ist erst herauszufinden, was die Software tun soll. Das Ergebnis der Analyse ist eine *Anforderungsdokumentation*, in der klare Anforderungen festgelegt sind.

Entwurf (design): Ausgehend von den Anforderungen wird die Struktur bzw. Architektur der Software in der *Entwurfsdokumentation* festgelegt. Der Entwurf umfasst alle Betrachtungsebenen, von der obersten Architekturebene bis hinunter zu Details im gewünschten Verhalten einzelner Objekte. Vereinfachend kann man sagen, dass die abstrakten Maschinen in der zu entwickelnden Software beschrieben werden.

Implementierung (implementation): Die Implementierung ist die Tätigkeit der Umsetzung des Entwurfs in ein Programm. Auch das Ergebnis dieser Tätigkeit nennt man Implementierung. Man implementiert also die in der Entwurfsdokumentation beschriebenen abstrakten Maschinen. Unter Programmierung im engeren Sinn versteht man das, was in der Implementierungsphase gemacht wird.

Verifikation (verification): Durch die Verifikation wird überprüft, ob bereits implementierte Software der Anforderungsdokumentation entspricht. Hier wird der Kreis zum Ergebnis der Analyse geschlossen, um Fehler im Entwurf und in der Implementierung zu finden. Unter Verifikation im engeren Sinn versteht man formale Überprüfungen, während im weiteren Sinn jede Form der Überprüfung zulässig ist. Eine wichtige Form der nicht-formalen Überprüfung ist ein *Code Review*, bei dem man den Programmcode liest und auf Übereinstimmung mit den Anforderungen hin analysiert.

Testen (testing): Beim Testen wendet man die Software systematisch auf sorgfältig gewählte Testfälle an, um Fehler aufzudecken. Im Gegensatz zur formalen Verifikation können Fehler durch Testen nur mit einer bestimmten Wahrscheinlichkeit entdeckt werden, abhängig vom Umfang und der Qualität der Testfälle. Durch Testen ist es jedoch auch möglich, Fehler in der Analyse selbst oder in einem Bereich zu finden, der durch die formale Verifikation nicht abgedeckt ist.

Validierung (validation): Unter Validierung versteht man die Überprüfung der Software hinsichtlich einer breiten Palette von Zielen. Man möchte beispielsweise feststellen, ob und wie gut die Software die

tatsächlichen Aufgaben bestimmter Anwender erfüllen kann, oder ob die Qualität und Praxisrelevanz der Software deren Anschaffung oder Weiterentwicklung rechtfertigt. Die Validierung schließt alle Entwicklungsphasen ein, auch die Analyse und Verifikation bzw. das Testen.

Diese Entwicklungsschritte werden meist nicht nur hintereinander, einer nach dem anderen durchgeführt, sondern überlappend. Das heißt, man analysiert, entwirft, implementiert, etc. zuerst nur einen kleinen Teil der Software und wiederholt diese Schritte für andere Teile, noch bevor alle Schritte für die ersten Teile durchgeführt sind. Damit will man gesammelte Erfahrungen so rasch wie möglich nutzen. Man spricht von *zyklischen Softwareentwicklungsprozessen*, da die einzelnen Schritte zyklisch wiederholt werden, auch wenn die Zyklen nur selten klar voneinander abgegrenzt sind. Die *schrittweise Verfeinerung* bezieht sich darauf, dass die Software zuerst nur in groben Zügen vorliegt, aber stetig verfeinert wird.

An die Entwicklungsphase schließt die Wartungsphase an. Dabei wird die sich schon im praktischen Einsatz befindliche Software gepflegt, also im laufenden Betrieb festgestellte Fehler korrigiert und die Software im notwendigen Ausmaß an sich ändernde Bedingungen und Anforderungen angepasst. Änderungen in der Wartungsphase umfassen alle Entwicklungsschritte von der Analyse bis zu Verifikation, Test und Validierung. Jedoch erfordern Änderungen in der Wartungsphase äußerste Vorsicht, um die Ziele der Anwender nicht zu gefährden. Häufig befinden sich unterschiedliche Versionen der Software gleichzeitig in der Entwicklungs- und Wartungsphase. So kann man die größeren Freiheiten in der Entwicklungsphase nutzen und gleichzeitig die Anwender unterstützen, die aus irgendwelchen Gründen noch mit älterer Software arbeiten müssen.

Die Anwendungsphase der Software deckt sich im Großen und Ganzen mit der Wartungsphase. Software, die nicht mehr gewartet wird, wird nach wenigen Jahren kaum mehr effizient nutzbar sein, da sich die Einsatzbedingungen meist rasch ändern.

1.6.2 Ablauf und Werkzeuge der Programmierung

Programmierung im weiteren Sinn umfasst neben der Implementierung große Teile des Entwurfs, der Verifikation und des Testens. Zur Klarstellung, dass wir es mit der Programmierung im weiteren Sinn zu tun haben, verwenden wir auch den Begriff *Programmkonstruktion*. Beim Programmieren wiederholen wir zyklisch immer wieder folgende Schritte:

Planen: Zuerst legen wir uns einen Plan zurecht, was im aktuellen Durchlauf erreicht werden soll.

Editieren: Darunter verstehen wir das Schreiben oder Ändern von Programmcode mittels eines Editors.

Übersetzen: Wenn erforderlich verwenden wir den Compiler und weitere Werkzeuge zur Erzeugung ausführbaren Codes. Falls der Compiler Fehlermeldungen liefert, gehen wir zurück zum Planen und Editieren.

Testen: In jedem Zyklus müssen Testfälle durchlaufen werden, um festzustellen, ob und inwieweit die Ziele erreicht wurden und welche Fehler noch vorhanden sind. Es ist durchaus möglich, dass von den Änderungen auch Programmteile betroffen sind, von denen wir das nicht erwartet haben. Wenn wir nach ausgiebigem Testen keine Fehler finden und die Software vollständig ist, sind wir fertig. Bei größeren Programmen tritt dieser Fall aber so gut wie nie ein.

Debuggen: Ein *Bug* (auf deutsch Käfer, Wanze, Laus) ist eine umgangssprachliche Bezeichnung für einen Programmierfehler. Beim Debuggen versuchen wir die Ursache für ein unerwünschtes Verhalten des Programms, also einen Fehler zu finden. Dabei gewonnenes Wissen benötigen wir zur Planung des weiteren Vorgehens.

Zahlreiche *Entwicklungswerkzeuge* unterstützen uns bei der Konstruktion von Programmen. Unter einem solchen Werkzeug verstehen wir spezielle Software, die entweder (ähnlich einem Hammer oder einer Zange) einen weiten Anwendungsbereich im Bereich der Softwareentwicklung hat oder nur für ganz spezifische Aufgaben einsetzbar ist. Mit diesen Werkzeugen kommen wir sicher in Berührung:

Editor: Der Editor ist ein universell einsetzbares Werkzeug zum Lesen und Editieren (Schreiben oder Ändern) beliebiger Texte. Zum Editieren von Programmen verwenden wir überwiegend spezielle Editoren, welche die Syntax unserer Programmiersprache kennen und beispielsweise Zeilen automatisch entsprechend einem für die Sprache typischen Stil einrücken, auf noch offene Klammern hinweisen und über Farben oder Schriftarten syntaktische Sprachelemente hervorheben (Syntax Highlighting). Diese Fähigkeiten sind beim Programmieren und Lesen von Programmen oft sehr hilfreich. Wenn der vom Editor erwartete Programmierstil jedoch nicht mit dem tatsächlichen

Programmierstil übereinstimmt, kann die vom Editor stammende (falsche) Zusatzinformation sehr irritierend sein.

Compiler und Interpreter: Diese wichtigen Werkzeuge haben wir bereits in Abschnitt 1.4.3 kennengelernt.

Debugger: Mit Hilfe eines Debuggers können wir die Ausführung eines Programms an ausgewählten Stellen unterbrechen, den Zustand des Systems (vor allem die aktuellen Werte der Variablen) analysieren, und das Programm Anweisung für Anweisung schrittweise ausführen. Das gibt uns einen genauen Einblick in den dynamischen Programmablauf. Allerdings ist der Umgang mit einem Debugger sehr arbeitsaufwendig, da in üblichen Programmen gigantisch viele Anweisungen ausgeführt werden. Daher versucht man das Programm statisch zu verstehen und nur dann einen Debugger einzusetzen, wenn dies zum Finden einer Fehlerursache nötig ist.

Integrierte Entwicklungsumgebung: Ein solches Werkzeug integriert eine ganze Reihe zusammenpassender Entwicklungswerkzeuge in einer gemeinsamen Umgebung, meist unter einer grafischen Benutzeroberfläche. Die wichtigsten Werkzeuge wie die oben genannten sind durch wenige Mausklicks anwendbar. In der Regel werden benötigte Dateien automatisch verwaltet. Daten in diesen Dateien werden dazu verwendet, das Programmieren zu erleichtern. Beispielsweise wird man rasch auf falsch geschriebene Namen hingewiesen, oder Namen werden automatisch ergänzt, sobald deren Anfang eindeutig ist.

Nicht nur Werkzeuge, sondern auch *Bibliotheken* unterstützen uns bei der Programmierung ganz wesentlich. Eine Bibliothek ist eine Sammlung vorgefertigter Programmteile. In der Java-Programmierung verwenden wir hauptsächlich Klassen-Bibliotheken, also Sammlungen von Klassen. Wir müssen nicht alles neu programmieren, sondern haben für die häufigsten Aufgaben schon bewährte Lösungen zur Verfügung. Die wichtigsten Bibliotheken bekommen wir als Einheit zusammen mit den wichtigsten von der Sprache abhängigen Werkzeugen (Compiler und Interpreter) geliefert.

1.6.3 Softwarequalität

Software ist nicht gleich Software. Auch bei der Entwicklung von Software müssen wir auf Qualität achten. Generell können wir zwischen zwei Arten von Qualitätskriterien entscheiden – solche, die uns die Programmierung

erleichtern, und solche, die Anwender von uns verlangen. Nicht immer sind diese Arten klar voneinander zu trennen. Das sind die wichtigsten Qualitätskriterien, die in fast jeder Art von Software von Bedeutung sind:

Brauchbarkeit: Die Softwarequalität richtet sich hauptsächlich nach den Bedürfnissen der Anwender, die auf die Software angewiesen sind. Um brauchbar zu sein, muss die Software mehrere Kriterien erfüllen:

Zweckerfüllung: Die Software erfüllt nur dann ihren Zweck, wenn sie genau die Aufgaben, für die die Software tatsächlich eingesetzt wird, zufriedenstellend lösen kann. Das gilt für alle Anwendungsfälle, nicht nur die häufigsten. Umgekehrt liefern unnötige Eigenschaften der Software keinen Beitrag zur Zweckerfüllung, können jedoch die Kosten erhöhen und die Brauchbarkeit durch schlechtere Bedienbarkeit und größeren Ressourcenbedarf negativ beeinflussen. Daher sollen wir bei der Softwareentwicklung den tatsächlichen Bedarf der Anwender genau analysieren und alle benötigten, aber keine unnötigen Eigenschaften einbauen.

Bedienbarkeit: Die Bedienbarkeit hängt davon ab, wie einfach Aufgaben mithilfe der Software lösbar sind und wie hoch der Einarbeitungsaufwand ist. Vor allem häufig zu lösende Aufgaben sollen möglichst wenige Arbeitsschritte benötigen. Außerdem sollen keine unerwartet langen Wartezeiten entstehen, und die Bedienung soll intuitiv, ohne aufwendige Schulung möglich sein. Die Bedienbarkeit hängt von den Gewohnheiten und Erfahrungen der Anwender ab. Daher kann auch die Bedienbarkeit durch genaue Analyse des Anwenderverhaltens verbessert werden.

Effizienz: Man benötigt Ressourcen wie Rechenzeit, Hauptspeicher, Massenspeicher und Netzwerkbandbreite. Software, die mit Ressourcen sparsamer umgeht, ist effizienter und von höherer Qualität. Sie bietet mehr Potential für künftige Erweiterungen. Auch die Effizienz der Softwareentwicklung ist für Anwender von Bedeutung: Wenn die Entwicklung weniger Ressourcen benötigt, ist die Software billiger und rascher verfügbar.

Zuverlässigkeit: Falsche Ergebnisse, Programmabstürze, fehlende oder zu späte Reaktionen auf Ereignisse, etc. sollen in hochwertiger Software nicht vorkommen. Man muss sich auf die Software verlassen können. Die geforderte Zuverlässigkeit ist ein wesentlicher Kostenfaktor in der Softwareentwicklung, und deshalb strebt man nicht in jeder Art

von Software denselben hohen Zuverlässigkeitsgrad an. Ein Editor ist zwar wichtig, braucht aber nicht so zuverlässig sein wie die Steuerungssoftware in einem Kernkraftwerk, Flugzeug oder Auto, wo Fehler Leben kosten können. Absolute Zuverlässigkeit kann nie garantiert werden. Zur Erhöhung der Zuverlässigkeit setzt man auf Folgendes:

Bewährtheit: Software, die sich über einen langen Zeitraum praktisch bewährt hat, ist zuverlässiger als neue, nur wenig getestete Software. Wir können die Qualität erhöhen, indem wir unter realistischen Bedingungen ausgiebig testen. Allerdings können sich auch in bewährter Software in außergewöhnlichen Situationen immer wieder neue Fehler zeigen. Beispielsweise funktioniert eine Steuerungssoftware jahrelang problemlos, aber versagt beim ersten Auftreten eines ungewöhnlichen Störfalls völlig. Auch in der Softwareentwicklung kommt es auf Bewährtheit an. Man setzt Techniken und Methoden ein, die sich hinsichtlich der Zuverlässigkeit bewährt haben, und man nutzt die einschlägige Erfahrung von Entwicklerteams.

Formale Korrektheit: Wenn es auf hohe Zuverlässigkeit ankommt, können formale Korrektheitsbeweise das Vertrauen steigern. Der Einsatz komplexer formaler Methoden (abseits der üblichen, vom Compiler und ähnlichen Werkzeugen durchgeführten Überprüfungen) ist jedoch aufwendig und teuer. Beweisbar sind nur klar bestimmte formale Aussagen, die auf einer Reihe von Annahmen beruhen. Es kommt vor, dass eine Annahme in einer unerwarteten Situation verletzt ist, oder ein Fehler in einem Bereich auftritt, der durch die formale Aussage nicht abgedeckt ist. Fehler können also auch auftreten, wenn alle Beweise korrekt durchgeführt wurden.

Fehlerresistenz: Fehler kann man nicht gänzlich ausschließen, aber man kann deren Auswirkungen mildern. Ein entdeckter Fehler ist bei weitem nicht so schlimm wie ein verborgener. Beispielsweise erkennt man verletzte Annahmen durch `assert`-Anweisungen wie in Abschnitt 1.5.4. Auf einen entdeckten Fehler kann man reagieren, indem man Anwender darauf hinweist oder die Aufgabe auf andere Weise löst. In sicherheitskritischen Systemen berechnet man wichtige Werte manchmal mehrfach auf mehreren Rechnern mit unterschiedlichen Algorithmen. Man betrachtet nur übereinstimmende Werte als zuverlässig.

Wartbarkeit: Die Wartung von Software ist oft viel teurer als deren Entwicklung und leichte Wartbarkeit damit von großer Bedeutung. Gut wartbare Software ist auch für Anwender von höherer Qualität, da in der Wartungsphase notwendige Änderungen rascher und zuverlässiger erfolgen können. Folgende Faktoren spielen eine Rolle:

Einfachheit: Ein einfaches Programm ist natürlich leichter und zuverlässiger änderbar als ein kompliziertes. Wir versuchen daher, alle Programme so einfach wie möglich zu halten. In der Praxis werden Programme rasch kompliziert, wenn wir nachträglich Code zur Behandlung irgendwelcher Sonderfälle hinzufügen müssen. Am einfachsten bleiben Programmteile, in denen wir bereits in der ersten Version alle Eventualitäten berücksichtigt und klar und übersichtlich ausgedrückt haben. Deshalb braucht es viel Erfahrung und Voraussicht, um einfache Programme zu schreiben.

Lesbarkeit: Es soll einfach sein, durch Lesen des Programmcodes die Logik im Programm zu verstehen. Die Lesbarkeit hängt vom Programmierstil ab, dieser wiederum von der Erfahrung.

Lokalität: Der Effekt jeder Programmänderung soll auf einen kleinen Programmteil beschränkt bleiben. Nicht-lokale bzw. globale Effekte sind nur schwer erkennbar und führen daher leicht zu Fehlern. Objektorientierte Sprachen bieten einige Möglichkeiten, die uns dabei unterstützen, Änderungen lokal zu halten.

Faktorisierung: Die Zerlegung eines Programms in kleinere Einheiten mit zusammengehörigen Eigenschaften nennt man *Faktorisierung* (*factoring*). Wenn es mehrere gleiche Programmteile gibt, soll man diese zu einer Einheit zusammenführen. Darin enthaltene Fehler brauchen danach nur mehr an einer Stelle ausgebessert zu werden, nicht an mehreren schwer zu findenden Stellen. Viele Formen der Abstraktion (wie Objekte, Funktionen und Ähnliches) helfen dabei, die Faktorisierung zu verbessern. Eine gute Faktorisierung hat positive Auswirkungen auf die Einfachheit (überschaubare abstrakte Maschinen), Lesbarkeit (verständliche Namen von Klassen und Methoden) und Lokalität (durch Kapselung zusammengehöriger Variablen und Methoden in einem Objekt).

Natürlich wollen wir stets qualitativ hochwertige Software produzieren. Allerdings hat Qualität auch einen Preis und kann die Softwarekosten un-

ter Umständen explodieren lassen. Wir müssen darauf achten, in welchem Bereich es sich auszahlt, wieviel in welche Art von Qualität zu investieren, um insgesamt den größten Nutzen daraus zu ziehen.

1.6.4 Festlegung von Softwareeigenschaften

Wir müssen festlegen, welche Eigenschaften wir von unserer Software erwarten. Die Form der Festlegung ist von Bedeutung. Sie bestimmt, wie die Überprüfung der Software erfolgen kann.

Informelle Beschreibung: Beschreibungen in Form eines informellen Textes erfordern keine Spezialkenntnisse. Alle gewünschten Eigenschaften sind ausdrückbar. Die Präzision ist jedoch problematisch. Eine zu vage Beschreibung lässt unerwünschte Interpretationen zu, während eine präzise Beschreibung sehr umständlich und nur schwer lesbar ist. Formale Verifikationen sind auf dieser Basis nicht möglich. Auch beim Testen ergibt sich gelegentlich die Schwierigkeit, dass nicht klar ist, welches Ergebnis in einem bestimmten Fall erwartet wird.

Anwendungsfälle: Eine spezielle Form der informellen Beschreibung legt eine Reihe konkreter Anwendungsfälle (*use cases*) fest. Für jeden Anwendungsfall beschreibt man genau, welche Eingaben Anwender machen und welche Ergebnisse sie erwarten. Anwendungsfälle ergeben sich aus der Beobachtung künftiger Anwender. Die Anwendungsfälle können leicht in Testfälle abgebildet werden. Wenn das Programmverhalten in manchen Situationen durch keinen Anwendungsfall beschrieben ist, trifft man sinnvolle Annahmen. Die Beschreibung ist überwiegend informell, mit allen Vor- und Nachteilen.

Testfälle: Man kann das gewünschte Verhalten eines Programms auch direkt über Testfälle spezifizieren. Eigentlich stellt man dabei nur Anwendungsfälle in Form von Testfällen dar. Diese Darstellung ist in gewissem Sinne formal. Häufig gibt man, wo dies sinnvoll erscheint, keine genauen Testwerte vor, sondern nur Wertebereiche, aus denen beim tatsächlichen Testen zufällig Werte gewählt werden. Damit kann man das Problem reduzieren, dass man niemals alle möglichen Fälle testen oder über Testfälle spezifizieren kann.

Formale Spezifikation: Nur formale Spezifikationen erlauben formale Verifikationen. Leider braucht man spezielles Expertenwissen sowohl

für die Erstellung als auch Verwendung. Der Umgang mit formalen Spezifikationen ist meist viel aufwendiger als der mit informellen Spezifikationen. Daher sind formale Spezifikationen nur in jenen Bereichen sinnvoll, in denen formale Verifikationen durchgeführt werden.

Vor allem über Testfälle und formale Spezifikationen, aber auch in Programmcode kann man nicht alles ausdrücken, was man gerne spezifizieren möchte. Wir unterscheiden zwei Arten von Eigenschaften:

Funktionale Eigenschaften (functional properties)

lassen sich im Großen und Ganzen in jeder Form von Spezifikation ausdrücken. Diese Eigenschaften beziehen sich darauf, welche Ergebnisse von Berechnungen für bestimmte Daten erwartet werden. Das entspricht den Ergebnissen der Funktionsanwendung auf Daten.

Nichtfunktionale Eigenschaften (non-functional properties)

sind dagegen nur sehr schwer oder gar nicht formal zu fassen. Beispiele sind eine bestimmte geforderte Zuverlässigkeit oder Wartbarkeit der Software, oder eine einfache Bedienbarkeit, ohne genaue Vorgaben, wie diese erfolgen soll.

Das gilt auch für Zusicherungen. Über `assert`-Anweisungen lassen sich nur Eigenschaften ausdrücken, die auch über Testfälle ausdrückbar sind. Für alles, was darüber hinausgeht, können Zusicherungen nur als Kommentare formuliert werden.

1.7 Programmieren lernen

Ein kurzer Abschnitt am Ende jeden Kapitels gibt Hinweise und Empfehlungen zum Erlernen und Verstehen des Kapitelinhalts. Den Schwerpunkt bildet eine Liste von Fragen, deren Beantwortung wichtige Begriffe und Zusammenhänge in Erinnerung ruft und Querverbindungen herstellt.

1.7.1 Konzeption und Empfehlungen

Es gibt unzählige Ansätze, um Programmieren zu lernen. Wir geben uns hier nicht mit einfachen Programmierfähigkeiten zufrieden, sondern beschäftigen uns von Anfang an mit Hintergrundwissen und Zusammenhängen. Wir wollen später nicht nur kleine Programme schreiben können,

sondern auch in der Lage sein, die Programmkonstruktion im größeren Kontext zu verstehen und Programmiertechniken gezielt einzusetzen.

Im ersten Kapitel haben wir eine große Zahl an Begriffen eingeführt und zueinander in Beziehung gesetzt. Die meisten Begriffe werden wir in den nächsten Kapiteln in anderen Zusammenhängen wiederfinden. Viele Nuancen in der Semantik der Begriffe werden sich erst im Laufe der Zeit entwickeln. Beim späteren wiederholten Lesen des ersten Kapitels werden sich wahrscheinlich neue Perspektiven ergeben, die beim ersten Lesen nicht zu erkennen waren.

Um gute Programme schreiben zu können, braucht es neben einigem Wissen viel praktische Übung und Erfahrung. Vieles muss man ausprobieren. Dann werden sich erste Erfolge beim Programmierenlernen bald einstellen. Durch praktisches Programmieren schult man das abstrakte Denkvermögen, das in der Programmkonstruktion so wichtig ist. Mit ausreichend Erfahrung entstehen Bilder abstrakter Strukturen quasi automatisch im Kopf, ohne dass man alle Details kennen und alle dynamischen Abläufe durchspielen muss. Daher steigt mit dem abstrakten Denkvermögen auch die Geschwindigkeit beim Programmieren. Man lernt, hochkomplexe Zusammenhänge unter Zeitdruck zu verstehen und in Programme umzusetzen. Neben abstraktem Denkvermögen braucht und entwickelt man dafür auch eine hohe Konzentrationsfähigkeit.

Wir müssen in Teams zusammenarbeiten. Auch das muss man lernen. Konkret muss man eine fachspezifische Sprache entwickeln und fast täglich in der Kommunikation innerhalb des Teams anwenden, um Gedanken im Zusammenhang mit der Programmierung rationell auszutauschen. Man lernt, die Gedanken anderer Teammitglieder zu antizipieren. Erst dann kann man zusammen effizienter Programme konstruieren als alleine. Innerhalb jedes Teams wird sich eine andere Aufgabenverteilung ergeben, abhängig davon, wo die Stärken der einzelnen Teammitglieder liegen.

Unterschiedliche Stärken führen zu individuellen Herangehensweisen an das Programmierenlernen. Manchmal sieht man rasch Fortschritte. Gelegentlich scheint nichts weiterzugehen, bevor man plötzlich einen großen Schritt vorwärts macht. Lernfortschritte sind kaum planbar.

Die wichtigste Empfehlung lautet, sich intensiv mit der Programmierung auseinanderzusetzen. Das betrifft sowohl die theoretische Auseinandersetzung mit den angesprochenen Themenbereichen als auch das praktische Ausprobieren und Experimentieren mit Beispielprogrammen. Beispielsweise sollte man versuchen, auch solche Querverbindungen zwischen den in diesem Kapitel eingeführten Begriffen herzustellen, die nicht explizit im

Text angeführt sind. Es empfiehlt sich auch, Beispielsprogramme abzuändern und zu schauen, was die Änderungen bewirken. Konkret könnte man versuchen, den Wertebereich für das Zahlenratespiel auf 0 bis 9 einzuschränken bzw. auf -50 bis 150 zu erweitern, oder statt einer Zahl ein einzelnes Bit bzw. ein Bitmuster in einem Wort zu erraten.

1.7.2 Kontrollfragen

- Was ist eine *Deklaration* bzw. *Definition*?
- Wozu braucht man *Variablen* und was haben Variablen mit dem Speicher eines Computers gemeinsam?
- Was haben *Nachrichten* mit *Methoden* und *Funktionen* zu tun, und wie hängen diese Begriffe mit den Befehlen eines Prozessors zusammen?
- Wodurch unterscheidet sich ein *formaler Parameter* von einem *aktuellen Parameter* bzw. *Argument*? Bestehen diese Unterschiede auf syntaktischer oder semantischer Ebene?
- Wozu braucht und woran erkennt man *Kommentare*?
- Was sind *Klassen* und *Objekte* und welche Beziehung gibt es zwischen diesen Begriffen?
- Was macht ein *Konstruktor* und wie unterscheidet er sich von einer Methode?
- Was versteht man unter dem *Initialisieren* eines Objekts oder einer Variablen?
- Welche Beziehungen bestehen zwischen den Begriffen *Datenkapselung*, *data hiding* und *Datenabstraktion*?
- Was ist eine *Schleifenbedingung*, ein *Schleifenzähler*, ein *Schleifenrumpf* und eine *Iteration*?
- Wie hängen *bedingte Anweisungen* mit *Programmzweigen* und *Fallunterscheidungen* zusammen?
- Was ist ein *Bit*, ein *Byte* und ein *Wort*, und warum hat ein Kilobyte 1024 Byte?

- Was ist ein *Algorithmus*, und wodurch unterscheidet sich ein Algorithmus von einem Programm bzw. einer Methode?
- Aus welchen Teilen setzt sich die *Von Neumann-Architektur* zusammen? Wodurch unterscheidet sie sich von der *Harward-Architektur*?
- Welche Schritte werden von einer Von Neumann-Architektur ständig wiederholt ausgeführt, welche von einem Interpreter?
- Was ist eine *Assembler-Sprache*?
- Warum verwenden wir *abstrakte Maschinen* wie die JVM?
- Was sind *Berechnungsmodelle* und wie hängen sie mit Programmiersprachen zusammen?
- Welche Eigenschaften von Objekten helfen dabei, Objekte als abstrakte Maschinen zu betrachten?
- Was ist eine *Komponente*?
- Wie stehen die Begriffe *Architektur*, *Implementierung* und *Schnittstelle* sowohl in der Hardware als auch in der Software zueinander?
- Können Sie Beziehungen zwischen den Begriffen *Objektzustand*, *-verhalten* und *-identität* einerseits und *Daten*, *Algorithmen* und *Umggebung* andererseits erkennen?
- Was bedeuten *Syntax*, *Semantik* und *Pragmatik*, und wozu verwendet man eine *Grammatik*?
- Wozu dienen *deklarierte Typen*?
- Was unterscheidet einen *Compiler* von einem *Interpreter*?
- Was versteht man unter *Quell*-, *Ziel*- und *Zwischencode*?
- Wie kommen Java-Programme üblicherweise zur Ausführung (Übersetzung und/oder Interpretation)?
- Wofür steht die Abkürzung *JIT*? In welchem Zusammenhang ist sie von Bedeutung?
- Wie wirken sich *Laufzeitfehler* im Vergleich zu *Fehlermeldungen* und *Warnungen* aus?

- Wann ist etwas *statisch* und wann *dynamisch*?
- Wann spricht man von *statischer* und wann von *dynamischer Semantik*?
- Was ist eine *Programmoptimierung* und wer macht diese?
- Was unterscheidet formale von natürlichen Sprachen?
- Was versteht man unter *Abstraktion*, und welche Arten davon haben wir schon kennengelernt?
- Wodurch unterscheiden sich *reine Funktionen* von *Methoden*, *Prozeduren* und *Routinen*?
- Welche *Programmierparadigmen* haben wir unterschieden? Wodurch zeichnen sie sich jeweils aus?
- Wozu dient der *Lambda-Kalkül*? Welche Eigenschaften hat er, und warum ist er in der Informatik so bedeutend?
- Was ist ein λ -*Ausdruck*? Welche Arten von λ -Ausdrücken kann man unterscheiden?
- Was versteht man unter *freien* bzw. *gebundenen Variablen* und was unter *Ersetzung*?
- Was bewirkt eine α -*Konversion* bzw. *beta-* und η -*Reduktion*?
- Wann ist ein λ -Ausdruck in *Normalform*?
- Was besagt die *Unentscheidbarkeit des Halteproblems*?
- Was sind *Zusicherungen*? Wie können wir sie ausdrücken?
- Wodurch unterscheiden sich *Vor-* von *Nachbedingungen*?
- Was versteht man unter der *Entwicklung*, *Wartung* und *Anwendung* von Software?
- Welche *Entwicklungsschritte* werden unterschieden?
- Was versteht man unter den Begriffen *zyklischer Softwareentwicklungsprozess* und *schrittweise Verfeinerung*?

- Welche Schritte führt man beim Programmieren zyklisch wiederholt aus?
- Welche *Softwareentwicklungswerkzeuge* kennen Sie?
- Was ist eine *Klassen-Bibliothek*?
- Wovon hängt die *Qualität* von Software ab?
- Wie kann man erwartete *Eigenschaften* von Software festlegen?
- Was unterscheidet *funktionale* von *nichtfunktionalen Eigenschaften*?

2 Grundlegende Sprachkonzepte

In diesem Kapitel wird die Programmierung ganz konkret. Es wird ein Detailverständnis der wichtigsten Sprachkonzepte in Java vermittelt. Das Wissen aus diesem Kapitel sollte ausreichen um selbst kleine Programme zu schreiben.

2.1 Die Basis

Zunächst geben wir ein Beispiel für einen Algorithmus und beschreiben, wie wir intuitiv vom Algorithmus zu einem ganz einfachen Programm kommen können. Dann betrachten wir die wichtigsten Elemente dieses Programms etwas genauer: Variablen, Zuweisungen, Typen, etc.

2.1.1 Vom Algorithmus zum Programm

Ein *Algorithmus* ist eine aus endlich vielen Schritten bestehende eindeutige Handlungsvorschrift¹ zur Lösung eines Problems. Im Alltag begegnet man Algorithmen in Form von Gebrauchs- oder Bauanleitungen, oder in Form von Berechnungsschritten (z.B. beim schriftlichen Summieren durch den Kellner im Restaurant) oder in Form von Kochrezepten. Während bei einem Kochrezept aber viele Schritte nicht explizit erwähnt werden, sondern vom Koch implizit auf Grund seiner Erfahrung befolgt werden (z.B. hole die Pizza aus dem Ofen bevor sie anbrennt) soll ein Algorithmus so formuliert sein, dass sogar eine Maschine ohne jegliche Intelligenz eine Lösung zustandebringt. In Abschnitt 4.1.1 werden wir uns noch ausführlicher mit dem Begriff des Algorithmus beschäftigen.

Eine Folge von Anweisungen, die eine Maschine verstehen und ausführen kann, nennt man Programm. Algorithmen werden also durch Programme in einer für die Maschine verständlichen Form beschrieben. Programme setzen sich aus Anweisungen zusammen, die – ganz allgemein formuliert – auf bestimmten Objekten operieren, d.h. diese verarbeiten und verändern.

¹Handlungsvorschrift = Beschreibung von Handlungen und Ihrer Abfolge

Die Objekte werden durch entsprechende Daten im Rechner abgebildet. Sie setzen sich aus Werten² zusammen, also den im Programm auftretenden Berechnungsgrößen. Die Lösung des Problems liegt dann ebenso in Form von veränderten oder neuen Objekten vor.

Für unseren Zweck reicht es, wenn wir die Begriffe „Objekte“, „Daten“ und „Werte“ als gleichbedeutend betrachten, obwohl es zwischen ihnen doch feine Unterschiede gibt. So sprechen wir tendenziell eher von

- Objekten, wenn wir eine abstrakte Sichtweise hervorheben wollen,
- Werten, wenn wir konkrete einzelne Werte wie beispielsweise Zahlen oder Variableninhalte (siehe unten) meinen,
- Daten, wenn wir größere oder unbestimmte Ansammlungen von Werten meinen.

Daneben verwenden wir viele weitere Begriffe für Werte bestimmter Arten, beispielsweise Wahrheitswerte und (natürliche, ganze, etc.) Zahlen.

Der Algorithmus von Euklid. Betrachten wir ein typisches Beispiel für einen Algorithmus: Der *Euklidische Algorithmus* zur Berechnung des größten gemeinsamen Teilers (GGT) zweier natürlichen Zahlen wird vom griechischen Mathematiker Euklid (ca. 360 v. Chr. bis ca. 280 v. Chr.) im Buch 7 seiner Abhandlung „Die Elemente“ beschrieben. In Lehrbüchern über Programmierung taucht er oft als Beispiel in unterschiedlichen Formulierungen auf.

Der GGT wird beim Kürzen von Brüchen benötigt: Ein Bruch lässt sich kürzen, indem man Zähler und Nenner durch deren GGT dividiert. Der Algorithmus von Euklid lautet (adaptiert nach einer deutschen Übersetzung von Buch 7):

„... Wenn N Teiler von M ist, ist N , weil auch Teiler von sich selbst, gemeinsamer Teiler. N ist dann auch der größte Teiler, denn größer als N kann ein Teiler von N nicht sein.

Wenn N nicht Teiler von M ist, subtrahiert man, von den beiden Zahlen M und N ausgehend, immer die kleinere von der größeren bis die entstandene Zahl Teiler der ihr vorhergehenden ist, der dann der größte gemeinsame Teiler von M und N ist. ...“

²„Datum“ als Einzahl von „Daten“ wird wegen der allgemeinen Verwendung im Sinn von „Kalenderdatum“ nur selten benutzt. Stattdessen sprechen wir eher von einem „Wert“ oder manchmal auch „Datenwert“.

In diesem Text erkennt man zwei Variablen M und N , die für Objekte stehen (oder anders gesagt, die Objekte enthalten), mit welchen der Algorithmus arbeitet. Jedes dieser Objekte ist eine natürliche Zahl. Zu Beginn enthalten M und N die beiden Zahlen, deren GGT berechnet werden soll. Durch bestimmte Handlungen (wir nennen sie Operationen) werden die Variablen M und N verändert (z.B. „subtrahiert man ... die kleinere von der größeren“). Genaugenommen werden die Objekte, die in den Variablen enthalten sind, verändert. Am Ende enthalten beide Variablen dasselbe gesuchte Objekt, den größten gemeinsamen Teiler.

Struktur des Algorithmus. Die obige Formulierung des Algorithmus lässt sich so umformen, dass die einzelnen Schritte klarer ersichtlich sind:

Solange M ungleich N ist, wiederhole:

Wenn M größer als N ist, dann:

Ziehe N von M ab und weise das Ergebnis M zu.

Sonst (das heißt, N ist größer als M):

Ziehe M von N ab und weise das Ergebnis N zu.

(Nun ist M gleich N)

M bzw. N ist der größte gemeinsame Teiler.

In dieser Beschreibung des Algorithmus sieht man einen zeitlichen Ablauf. Man beginnt oben und führt die Schritte nacheinander in der gegebenen Reihenfolge durch, bis man am Ende das Ergebnis erhält. Zwei verschiedene Arten von Anweisungen sind deutlich unterscheidbar:

Operationen mit Objekten: Die beiden Variablen M und N sind nicht konstant, sondern stehen zu unterschiedlichen Zeitpunkten für unterschiedliche Objekte. Die Anweisungen, die die Inhalte von M und N verändern („weise das Ergebnis ... zu“), heißen *Zuweisungen*. Durch eine Zuweisung wird der alte Inhalt von M bzw. N durch einen neuen Inhalt ersetzt. Im Falle der Berechnung des GGT ist der neue Inhalt die Differenz der bisherigen Inhalte von M und N . Man spricht von *Variablen*, weil die Inhalte sich im Laufe der Zeit ändern können, also variabel sind. Variablen betrachten wir in Abschnitt 2.1.2 genauer.

Steuerung des Programmflusses: Die obige Beschreibung des Algorithmus geht davon aus, dass eine Handlung nach der anderen ausgeführt wird und nicht mehrere gleichzeitig. Diese Hintereinanderreihung von Handlungen nennt man auch *Sequenz*. Zusätzlich gibt es Anweisungen, die diesen sequentiellen Ablauf steuern, nämlich die Auswahl auf

Grund einer Fallunterscheidung zwischen zwei alternativen Handlungen, auch *Selektion* genannt: „Wenn . . . , dann: . . . Sonst . . . “. Die beiden alternativen Anweisungen sind gegenüber der Selektionsanweisung eingerückt dargestellt, um deutlich zu machen, dass sie nur unter der darüber angegebenen Bedingung durchgeführt werden.

Weiters deuten die Worte „Solange M ungleich N ist, wiederhole:“ darauf hin, dass die darunterstehenden Anweisungen wiederholt ausgeführt werden sollen. Diese Anweisung zur Wiederholung heißt auch *Iteration*. Auch hier werden alle zu wiederholenden Anweisungen eingerückt dargestellt.

In unserem Beispiel ist auch erläuternder Text (in Klammern) in der Formulierung des Algorithmus vorhanden. Solche *Kommentare* gehören eigentlich gar nicht wirklich zum Algorithmus, sondern unterstützen nur die Lesbarkeit. Man könnte sie ersatzlos streichen. So ist durch das Wort „Sonst“ schon klar, in welchen Fällen die danach folgende Anweisung durchzuführen ist. Die Kommentare sorgen nur dafür, dass wir beim Lesen die entsprechenden Bedingungen gleich im Kopf haben.

Es gibt auch Algorithmen, bei denen ein Problem durch parallel ablaufende Handlungen gelöst wird. Auf oft recht komplizierte parallele Algorithmen wollen wir hier jedoch nicht näher eingehen.

Die obige umgangssprachliche Beschreibung des Euklidischen Algorithmus ist ausreichend um die Schritte für beliebige Startwerte durchzuführen. Probieren Sie den Algorithmus für die Startwerte $M = 9$ und $N = 6$ aus. Sie benötigen zwei Zuweisungen, bis M und N die gleiche Zahl 3 enthalten, die dem GGT von 9 und 6 entspricht.

Umsetzung in Java. Der Algorithmus lässt sich ziemlich direkt in Java formulieren. In Listing 2.1 finden wir ein Java Programm, das den GGT von 1027 und 395 berechnet und auf den Bildschirm ausgibt.

Generell besteht ein Programm in einer imperativen Programmiersprache aus einer Folge von *Anweisungen*. Die Anweisungen können von einer (virtuellen) Maschine ausgeführt werden. So stehen in den Zeilen 7, 8, 12 und 14 spezielle Anweisungen, die Inhalte von Variablen neu setzen; das sind Zuweisungen.

In fast allen Zeilen sind auch *Ausdrücke* als Teil einer Anweisung enthalten. Ausdrücke sind alle Konstrukte, die ein Objekt als Ergebnis liefern.

Listing 2.1: Der Euklidische Algorithmus in Form eines Java Programms

```

1 public class Euklid1 {
2     public static void main(String[] args) {
3
4         int m;
5         int n;
6
7         m = 1027;
8         n = 395;
9
10        while (m != n)
11            if (m > n)
12                m = m - n;
13            else // es gilt n > m
14                n = n - m;
15
16        // nun gilt m == n
17        System.out.println(m);
18
19    }
20 }
```

Beispielsweise ist „ $m - n$ “ in Zeile 12 ein Ausdruck, der die Differenz zwischen den Zahlen m und n liefert. Die Zahl 1027 in Zeile 7 ist ebenfalls ein einfacher Ausdruck, nämlich ein *Literal*. Literale beschreiben konkrete Werte direkt – im Gegensatz zu Ausdrücken, die Ergebnisse von Berechnungen liefern.

Der eigentliche Euklidische Algorithmus wird durch die Zeilen 10 bis 14 dargestellt. Man kann erkennen, wie die Anweisungen, die wir oben umgangssprachlich beschrieben haben, in Java formuliert werden. Zeile 17 enthält eine Anweisung, die das Ergebnis m am Bildschirm ausgibt. Es sind auch zwei Kommentare vorhanden (zwischen „//“ und dem Zeilenende), die nur dazu dienen, die Lesbarkeit des Programms zu verbessern. Ein Kommentar führt keine Anweisung aus.

Die Zeilen 4 bis 8 enthalten die Deklarationen der benötigten Variablen (eine Erklärung folgt in Abschnitt 2.1.2) und die Zuweisungen der Anfangszahlen. Die Zeilen 1 und 2 gehören zusammen mit den Zeilen 19 und 20 zur Programmorganisation, die benötigt wird, damit sich das Programm entsprechend der Java-Spezifikation übersetzen und ausführen lässt. Die Bedeutung dieser Zeilen wird in den folgenden Kapiteln erklärt.

Der Name des Programms steht nach dem Schlüsselwort `class` in Zeile 1 und kann vom Programmierer selbst festgelegt werden. Um das Programm auszuführen (dringend empfohlen) müssen wir darauf achten, dass der Programmtext in einer Datei namens `Euklid1.java` gespeichert ist. Der Dateiname muss mit dem Klassennamen übereinstimmen.

2.1.2 Variablen und Zuweisungen

Eine Variable steht für einen im Lauf der Zeit änderbaren Wert. Sie enthält zu jedem Zeitpunkt einen Wert, den *Wert der Variablen*. Man kann die Variable als benannten Speicherbereich oder Datenbehälter verstehen, der Platz für genau einen Wert hat. Dieser Wert kann durch eine Zuweisung mit einem neuen Wert überschrieben werden. In Abbildung 2.2 hat die Variable `m` den Wert 1027 und die Variable `n` den Wert 395. Der Name einer Variable wird stellvertretend für deren Wert benutzt. Wenn wir zum Beispiel sagen: „berechne die Differenz von `m` und `n`“, dann meinen wir eigentlich: „berechne die Differenz der Werte von `m` und `n`“, also $1027 - 395$.



Abbildung 2.2: Zwei Variablen `m` und `n`

Eine Variable in einer Programmiersprache wie Java unterscheidet sich von einer Variable in der Mathematik, wo es keine zeitliche Dimension gibt. So beschreibt die mathematische Gleichung $U = 2r\pi$ den funktionalen Zusammenhang zwischen zwei Variablen, das heißt, wie man U bestimmen kann, sobald r feststeht (oder umgekehrt). Die beiden mathematischen Variablen haben jedoch zu keinem Zeitpunkt einen bestimmten Wert, sondern sind Platzhalter für Elemente von Definitionsmenge bzw. Bildmenge der Funktion. Auch in rein funktionalen Programmiersprachen sowie dem in Abschnitt 1.5.2 vorgestellten Lambda-Kalkül sind Variablen reine Bezeichner für Werte, die sich im Lauf der Berechnung nicht ändern.

In Java und anderen imperativen Programmiersprachen ist eine Variable dagegen ein benannter Speicherbereich, der eine bestimmte Anzahl von Bytes belegt, um darin einen Wert zu speichern – siehe Abbildung 2.3.

Eigenschaften. Variablen haben unter anderem folgende Eigenschaften:

Wert: Manchmal spricht man auch vom *R-Wert*, eine Bezeichnung, die daher rührt, dass die Variable diesen Wert repräsentiert, wenn Sie auf der rechten Seite einer Zuweisung steht. Zuweisungen werden in Kürze besprochen. Der (R-)Wert ist also das, was wir bekommen, wenn eine Variable als Teil eines Ausdruck ausgewertet wird. In Abbildung 2.2 ist der (R-)Wert von `m` die Zahl 1027.

Name: Er wird manchmal auch *Bezeichner* (*identifier*) genannt.

Speicheradresse: An dieser Adresse wird der Wert der Variablen gespeichert. Manchmal wird die Adresse auch als *L-Wert* bezeichnet. Beispielsweise wird durch eine Zuweisung `m = n`; der (R-)Wert der Variablen `n` auf der rechten Seite der Zuweisung an die Speicheradresse der Variablen `m` geschrieben, die links steht – daher die Bezeichnung L-Wert für die Adresse. Der alte (R-)Wert von `m` wird zur Durchführung der Zuweisung nicht gebraucht und geht durch die Zuweisung verloren. In einigen Programmiersprachen kann der Programmierer eine Variable explizit über Ihren L-Wert ansprechen („Liefere die Adresse der Variablen `m`“), was in Java jedoch nicht möglich ist. In Java bleibt uns die Adresse der Variablen weitgehend verborgen, sodass sich die Adresse ändern (also innerhalb des Speichers verschieben) kann, ohne die Konsistenz zu zerstören.

Datentyp oder kurz **Typ:** Er schränkt die zulässige Wertemenge ein und bestimmt, wie das Bitmuster in den Speicherzellen der Variablen zu interpretieren ist (siehe Abschnitt 2.1.3).

Lebensdauer: Eine Variable existiert nur im Zeitraum zwischen dem Reservieren ihres Speicherbereichs (man spricht von *Memory Allocation*) bis zu dessen Freigabe (*Memory Deallocation*). Beispielsweise entspricht die Lebensdauer einer lokalen Variable einer Methode etwa der Zeit, in der die Methode ausgeführt wird.

Gültigkeitsbereich: Variablen sind im Allgemeinen nicht im gesamten Programm gültig. Der Gültigkeitsbereich bestimmt, von wo aus im Programm der Zugriff auf die Variablen ermöglicht werden kann. Beispielsweise sind lokale Variablen einer Methode nur innerhalb der Methode gültig. Dadurch wird garantiert, dass Variablen nur während ihrer Lebensdauer zugreifbar sind.

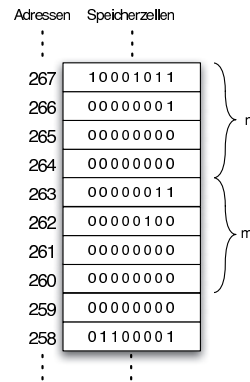


Abbildung 2.3: Schematische Darstellung eines Speicherbereichs, in dem zwei Variablen angelegt sind. Der Typ einer Variablen legt die Anzahl der benötigten Bytes fest und bestimmt gleichzeitig, wie die Bitmuster an der entsprechenden Stelle im Speicher zu interpretieren sind. Für die hier dargestellten Variablen m bzw. n vom Typ `int` mit den Adresse 260 bzw. 264 werden jeweils 4 Bytes benötigt. Das Bitmuster in der Variablen m entspricht dem Wert 1027, das in n dem Wert 395. Wäre n eine Fließkommazahl vom Typ `float`, hätte sie bei demselben Bitmuster den Wert $1.439 \cdot 10^{-42}$. Java stellt durch starke Typisierung sicher, dass es zu keinen Verwechslungen der Typen kommen kann.

Sichtbarkeitsbereich: Möglicherweise ist eine Variable an einer bestimmten Programmstelle zwar gültig, aber es darf trotzdem nicht darauf zugegriffen werden. Der Sichtbarkeitsbereich gibt an, von welchen Programmstellen aus ein Zugriff auf die Variable möglich ist. Der Sichtbarkeitsbereich kann im Vergleich zum Gültigkeitsbereich stärker eingeschränkt sein, etwa durch Deklaration als `private`.

Deklaration. Bevor eine Variable benutzt werden kann, muss sie *deklariert* (oder *vereinbart*) werden. Bei der Deklaration wird der Name der Variablen und deren Datentyp vereinbart. Die Ausführung der Deklarationsanweisung bewirkt auch, dass der für die Variable benötigte Speicherplatz reserviert wird. Die Zeilen 4 und 5 in Listing 2.1 sind Variablendeklarationen.

Der Typ `int` in der Deklaration `int m;` bedeutet, dass die Variable m ganzzahlige Werte speichern kann. Es wird also ausreichend Speicherplatz für eine ganze Zahl reserviert. Die Speicheradresse der Variablen wird durch die Deklaration zwar festgelegt, bleibt uns aber verborgen. In Java bekommen manche Variablen gleich bei der Deklaration einen Wert, der vom Datentyp abhängt. Im Falle von `int` wäre das der Anfangswert 0. Das gilt jedoch nicht für sogenannte lokale Variablen wie m und n in Listing 2.1, die in einer Methode wie `main` deklariert wurden. Solchen Variablen muss man einen Wert zuweisen, bevor man sie verwenden kann.

Die genaue Stelle einer Deklaration im Programm bestimmt den Gültigkeitsbereich, also jene Programmteile, von wo aus ein solcher Zugriff ermöglicht werden kann. Nach der Deklaration (und Zuweisung eines Anfangswertes) kann auf die Variable über ihren Namen zugegriffen werden.

Wir werden bei den Beschreibungen der jeweiligen Sprachkonstrukte erklären, wo dort deklarierte Variablen gültig sind. Oft entspricht der Sichtbarkeitsbereich (also der Bereich, in dem der Zugriff tatsächlich möglich ist) dem Gültigkeitsbereich. Wir werden erst in Kapitel 3 sehen, wie wir den Sichtbarkeitsbereich beeinflussen können.

In der Syntax der Variablendeklaration wird als erstes Wort der Name des Datentyps angegeben und nach einem Leerraum folgt der Name der Variablen. Die Deklarationsanweisung wird durch ein Semicolon beendet. Es ist aber auch möglich, mehrere Variablen desselben Typs in einer einzigen Anweisung zu deklarieren, indem man die Variablennamen durch Kommata trennt. Folgende Deklaration ist zwar syntaktisch verschieden, aber semantisch äquivalent zu den Zeilen 4 und 5 im Listing 2.1:

```
int m, n;
```

Zuweisung. Eine Zuweisung gibt einer Variablen einen neuen Wert. Der Zuweisungsoperator in Java ist „`=`“. Durch die Anweisung

```
n = 395;
```

wird der Variablen n der Wert 395 zugewiesen. Allgemein steht auf der linken Seite des Zuweisungssymbols der Name der Variablen, die einen neuen Wert bekommen soll, und rechts ein Ausdruck, der den Wert liefert. Der Typ des Ausdrucks muss mit dem Typ der Variablen *zuweisungskompatibel* sein. Wenn die Typen gleich sind, dann sind sie sicher zuweisungskompatibel. Wir werden später noch Fälle kennenlernen, in denen die Typen nicht gleich, aber trotzdem zuweisungskompatibel sind.

Verarbeitungsschritt	Werte von	
	m	n
m = 1027; n = 395;	1027	395
m = m - n;	632	395
m = m - n;	237	395
n = n - m;	237	158
m = m - n;	79	158
n = n - m;	79	79

Tabelle 2.4: Variablenzustände nach Verarbeitungsschritten nach Listing 2.1

Die Zeilen 7, 8, 12 und 14 in Listing 2.1 enthalten Zuweisungen. Dabei legen die Zuweisungen in den Zeilen 7 und 8 die Anfangswerte (auch *Initialwerte* genannt) fest, für die der GGT berechnet werden soll. Diese beiden Zuweisungen zusammen *initialisieren* das Programm. Die Zeilen 12 und 14 befinden sich im Rumpf einer Schleife und werden im Regelfall mehrmals durchlaufen. Sie sind Teil einer Verzweigung (*if-else*), die bewirkt, dass immer der kleinere vom größeren Wert abgezogen wird. Wenn die Anfangswerte wie hier 1027 und 395 sind, wird die Anweisung `m = m - n` drei mal durchgeführt. Wir können die Zustände der Variablen beim Programmablauf mit Hilfe einer Tabelle verfolgen: Tabelle 2.4 zeigt den Zustand der Variablen nach jedem Verarbeitungsschritt für die Startwerte 1027 und 395, die den Variablen im ersten Verarbeitungsschritt – der Initialisierung – zugewiesen werden.

Initialisierung. Zur syntaktischen Vereinfachung ist es in Java möglich, die Deklaration von Variablen mit deren Initialisierung zu verbinden. Beispielsweise haben die Deklarationen

```
int m = 1027;
int n = 395;
```

dieselbe Semantik (also dieselbe Bedeutung) wie die Zeilen 4 bis 8 in Listing 2.1 zusammengefasst. Noch kürzer ist die Syntax bei gleicher Semantik, wenn wir beide Variablen in nur einer Anweisung deklarieren:

```
int m = 1027, n = 395;
```

Zwecks einfacherer Lesbarkeit sollten wir jedoch vermeiden, zu viel in eine einzige Deklarationsanweisung zu schreiben.

Einer Variablendeklaration können sogenannte *Modifier* vorangestellt sein, welche die Eigenschaften der deklarierten Variablen genauer festlegen. Wie wir noch sehen werden, kann man damit beispielsweise die Sichtbarkeit der Variablen beeinflussen. Gelegentlich verwendet man den Modifier `final`, der verhindert, dass der Wert der deklarierten Variablen nach der ersten Zuweisung (Initialisierung) verändert wird:

```
final int vier = 4;
```

Es ist nicht möglich, den Wert 4 von `vier` (absichtlich oder unabsichtlich) durch eine weitere Zuweisung zu ändern. Sinnvollerweise ist eine solche Variable gleich bei der Deklaration zu initialisieren, da eine spätere Zuweisung ja nicht mehr möglich ist.

2.1.3 Datentypen

Ein Datentyp oder kurz Typ bestimmt eine Wertemenge und legt die Operationen fest, die auf den Elementen dieser Menge durchgeführt werden können. Die Elemente der Wertemenge heißen auch *Instanzen* des Typs. Zum Beispiel ist die Fließkommazahl `37.6991123` eine Instanz des Typs `double` und die ganze Zahl `12` eine Instanz des Datentyps `int`.

Jede Variable braucht einen Typ, damit klar ist, wieviel Platz die Variable im Speicher benötigt, also wieviele Bytes sie belegt. Weiters bestimmt der Typ auch, wie das Bitmuster in den Speicherzellen, die die Variable belegt, interpretiert werden soll. Beispielsweise liefert ein Ganzzahl-Bitmuster als Fließkommazahl interpretiert im allgemeinen einen falschen Wert – siehe auch Abbildung 2.3.

Generell stellt ein Datentyp einen Bauplan für eine Variable dar: Alle Variablen eines bestimmten Typs haben dieselbe Darstellung im Arbeitsspeicher, das heißt, sie belegen dieselbe Anzahl von Speicherzellen und haben dieselbe Interpretation der in ihnen enthaltenen Bits.

Elementare Typen. Datentypen, deren Instanzen einfache Werte sind, heißen *elementare Typen*. Man spricht auch von einfachen oder primitiven Typen. Elementar bedeutet in diesem Zusammenhang, dass sich Werte des Typs nicht aus Werten einfacherer Typen zusammensetzen lassen, also nicht weiter zerlegbar sind. Instanzen eines elementaren Typs werden in

Java direkt in einer Variablen eines entsprechenden Typs gespeichert – im Unterschied zu Instanzen der weiter unten beschriebenen Referenztypen.

Zu den elementaren Typen zählen mehrere Arten ganzzahliger Typen, zwei Typen von Fließkommazahlen sowie Typen von Zeichen und Wahrheitswerten. Tabelle 2.5 fasst die in Java zur Verfügung stehenden elementaren Datentypen zusammen. Die vielen Typen für Zahlen unterscheiden sich in ihrem Speicherverbrauch und dem in den zur Verfügung stehenden Bits darstellbaren Wertebereichen. Für ganze Zahlen verwendet man in der Praxis meist `int` oder `long`. Fließkommazahlen haben bei gleicher Anzahl an Bytes einen viel größeren Wertebereich als ganze Zahlen, weil diese Zahlen gerundet werden, sodass sie (unabhängig vom Wertebereich) nur auf wenige Stellen genau sind. Ganze Zahlen werden dagegen nicht gerundet. Details dazu folgen im zweiten Teil des Skriptums. In der Praxis verwendet man für Fließkommazahlen meist `double`. Die Operationen, die auf Werten elementarer Typen durchgeführt werden können, sind arithmetische Operationen, Vergleichsoperationen (Boolesche Operationen) und Bitoperationen, die in Abschnitt 2.2 besprochen werden.

Referenztypen. Die zweite Art von Datentypen sind sogenannte *Referenztypen*. Diese Typen heißen so, weil jede Instanz davon – das ist ein Objekt – in einem separaten Speicherbereich abgelegt wird und eine Variable nur eine Referenz auf den Speicherbereich des Objekts enthält. Eine *Referenz* (oder ein *Verweis*) ist eine Verknüpfung der Variable mit dem Speicherbereich, die das Objekt enthält. Über diese Referenz lässt sich das Objekt ansprechen.

In Lehrbüchern wird dieses Konzept manchmal mit einer Fernbedienung eines Fernsehers verglichen: Während die Tastatur eines Mobiltelefons sich direkt auf dem Telefon befindet, ist ein Fernseher zu groß und sperrig um ihn im Wohnzimmer dorthin zu verschieben, wo er bedient werden soll. Daher befindet sich die Tastatur nicht direkt auf dem Fernseher, sondern auf der Fernbedienung. Dem Mobiltelefon entspricht in diesem Vergleich der Wert eines elementaren Typs. Dem Fernseher entspricht ein Objekt eines Referenztyps, wobei die Fernbedienung die Referenz darstellt. Es ist denkbar, mehrere gleiche Fernbedienungen für denselben Fernseher zu haben. Alle Fernbedienungen müssen mit dem Fernsehertyp kompatibel sein, das heißt, jede Fernbedienung muss die Funktionen des Fernsehers kennen und ansteuern können. So kann es auch mehrere Variablen geben, die dasselbe Objekt referenzieren. Jedes Mobiltelefon hat dagegen nur eine

Typbezeichnung	Bytes	Wertebereich	Literale (Bsp.)
Integer-Typen:			
<code>byte</code>	1	-2^7 bis $2^7 - 1$	wie <code>int</code>
<code>short</code>	2	-2^{15} bis $2^{15} - 1$	wie <code>int</code>
<code>int</code>	4	-2^{31} bis $2^{31} - 1$	0, 1, -1
<code>long</code>	8	-2^{63} bis $2^{63} - 1$	0L, 1L, -1L
Fließkomma-Typen:			
<code>float</code>	4	ca. $-3.4 \cdot 10^{38}$ bis $3.4 \cdot 10^{38}$	1.2F, -1.2E8F
<code>double</code>	8	ca. $-1.8 \cdot 10^{308}$ bis $1.8 \cdot 10^{308}$	1.2, -1.2E8
Zeichen:			
<code>char</code>	2	alle 16-Bit Unicode Zeichen	'A', 'a', '3', '<'
Wahrheitswerte:			
<code>boolean</code>	1	true, false	true, false

Tabelle 2.5: Elementare Datentypen in Java

Tastatur, und jede Instanz eines elementaren Typs liegt dementsprechend nur in einer Variablen.

Variablen, deren Typ ein Referenztyp ist, nennen wir der Einfachheit halber *Referenzvariablen*. In Abbildung 2.6 sind zwei Referenzvariablen schematisch dargestellt.

Ein Beispiel für einen Referenztyp ist `String`, der Typ von *Zeichenketten*. Eine Zeichenkette ist, wie der Name schon sagt, eine Aneinanderreihung mehrerer Zeichen vom Typ `char`, die gemeinsam in einem Objekt des Typs `String` abgespeichert werden. In einer Variable des Typs `String` wird nur die Referenz auf das Objekt gespeichert. Mit der Anweisung

```
String s;
```

wird eine neue Referenzvariable vom Typ `String` erzeugt. Wenn wir eine neue Zeichenkette mit

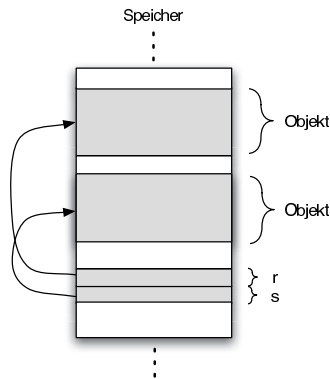


Abbildung 2.6: Schematische Darstellung zweier Referenzvariablen mit den Namen *r* und *s* sowie der referenzierten Objekte. Belegte Speicherbereiche sind grau hinterlegt. Pfeile repräsentieren Verweise von den Variablen zu den Objekten. Instanzen von Referenztypen belegen typischerweise einen größeren Speicherbereich als Instanzen primitiver Typen, da Sie meist mehrere Werte gemeinsam speichern. Ein Vorteil von Referenztypen liegt auf der Hand: Um die Inhalte der beiden Variablen zu tauschen, müssen nur die Referenzen getauscht werden und nicht die Objekte selbst. Vor allem können mehrere Variablen auf ein und das selbe Objekt verweisen. Das ist in vielen Situationen sinnvoll, beispielsweise wenn auf das Objekt von mehreren Stellen im Programm aus über verschiedene Variablen zugegriffen werden soll. Wenn dagegen zwei Variablen eines primitiven Typs denselben Wert speichern, liegen die Werte zweimal vor.

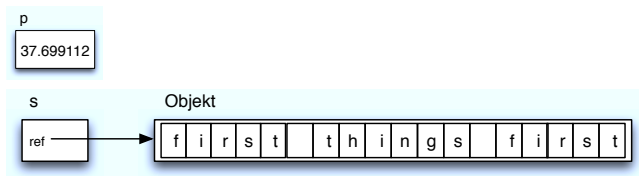


Abbildung 2.7: Die Variable *p* hat den elementaren Typ `float`. Die Variable *s* hingegen ist eine Variable vom Referenztyp `String`. Sie speichert eine Referenz auf ein Objekt vom Typ `String`.

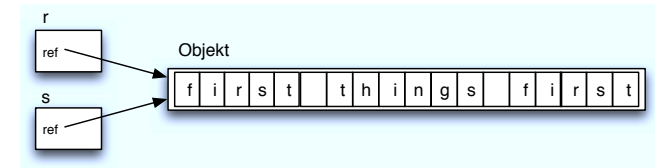


Abbildung 2.8: Zwei Referenzen auf dasselbe Objekt.

```
s = "first things first";
```

zuweisen, wird zunächst vom Ausdruck auf der rechten Seite des Zuweisungsoperators (in diesem Fall ein *String-Literal*) als Wert eine Referenz auf ein Objekt mit der Zeichenkette "first things first" geliefert, und diese Referenz wird dann in der Variablen *s* gespeichert. Das Ergebnis ist in Abbildung 2.7 veranschaulicht. In folgendem Beispiel werden die Auswirkungen der Verwendung von Referenztypen deutlicher:

```
String r, s; // Deklaration von zwei Variablen
s = "first things first";
r = s;       // Referenzen auf dasselbe Objekt
```

Die beiden Variablen *r* und *s* enthalten nach den Zuweisungen Referenzen auf dasselbe Objekt. Die Zeichenkette "first things first" existiert nur einmal (siehe Abbildung 2.8).

In Java enthalten alle Variablen gleich nach der Deklaration einen Wert, meist 0 oder etwas Vergleichbares – siehe Tabelle 2.5. Auch Referenzvariablen enthalten gleich nach der Deklaration einen Wert, nämlich `null`. Diesen speziellen Wert kann man als eine Referenz auf nichts betrachten. Direkt nach der Deklaration referenzieren *r* und *s* also kein Objekt. Jede Variable jeden Referenztyps kann statt einer Referenz auf ein Objekt `null` enthalten. In der Java-Programmierung wird `null` häufig verwendet.

Wir können beim Programmieren durch die Definition von Klassen neue Typen erstellen – siehe Kapitel 3. Alle auf diese Weise von uns selbst eingeführten Typen sind Referenztypen. Es ist in Java nicht möglich, selbst neue elementare Typ zu definieren.

2.2 Ausdrücke und Operatoren

Wir betrachten zunächst eine Reihe allgemeiner Grundlagen für das Verständnis von Ausdrücken. Danach folgt ein Überblick über die in Java vorhandenen Operatoren, teilweise mit detaillierten Beschreibungen. Schließlich betrachten wir einige Feinheiten im Zusammenhang mit Literalen und Typumwandlungen.

2.2.1 Allgemeines

Ganz allgemein ist ein *Ausdruck* in einer Programmiersprache ein Konstrukt, das einen Wert liefert. Im einfachsten Fall ist ein Ausdruck der Name einer Variablen oder ein Literal. Beides liefert einen Wert. Ein Wert kann aber auch von einer aufgerufenen Methode zurückgegeben werden, und der Aufruf stellt einen Ausdruck dar. Der *Rückgabewert* des Ausdrucks ist das Ergebnis der *Auswertung* des Ausdrucks.

Ausdrücke können durch Anwendung eines *Operators* zu einem komplexeren Ausdruck verknüpft werden. Dabei übernehmen die zu verknüpfenden Ausdrücke die Rolle von *Operanden* für diesen Operator. Jeder Ausdruck kann so als Operand wieder zu einem Bestandteil eines komplexeren Ausdrucks werden.

In Java unterscheidet man *einstellige (unäre)* von *zweistelligen (binären)* Operatoren. Zusätzlich gibt es einen *dreistelligen (ternären)* Operator, den Bedingungsoperator. Ein Beispiel für einen einstelligen Operator ist der Vorzeichenoperator „-“ (Minusoperator), der das Vorzeichen des nachfolgenden Operanden negiert, so wie in diesem Ausdruck:

$$- \ x$$

Es spielt keine Rolle, ob zwischen „-“ und „x“ Leerzeichen stehen oder nicht. Der Rückgabewert von $-x$ hat den negativen Wert der Zahl in der Variablen x . Ein Beispiel für einen zweistelligen Operator ist der Additionsoperator, der als Ergebnis die Summe der beiden Operanden liefert. So ist im Ausdruck

$$5 + 2$$

das Symbol $+$ der Operator, und die beiden Zahlen sind Operanden, die in diesem Fall Literale sind. Syntaktisch werden binäre Operatoren *infix* hingeschrieben, was bedeutet, dass der Operator zwischen den beiden Operanden stehen muss. Auch der Zuweisungsoperator ist ein Infix-Operator.

Unäre Operatoren können vor oder hinter dem Operanden stehen. Man spricht von *Präfix*- bzw. *Postfix*-Operatoren. Ein Beispiel ist der Ausdruck

$$x++$$

in dem der Operator „++“ als Postfix-Operator hinter dem Operanden x steht. Der oben erwähnte unäre Vorzeichenoperator „-“ ist ein Beispiel für einen Präfix-Operator. Im Ausdruck

$$++x$$

wird „++“ ebenso als Präfix-Operator verwendet. Die semantische Bedeutung des Symbol „++“ ist als Präfix-Operator verschieden von der als Postfix-Operator. Es handelt sich trotz Verwendung desselben Symbols also um zwei verschiedene Operatoren. Beide Operatoren inkrementieren den Wert ihres Operanden, das heißt, sie erhöhen den Wert um eins. Die genauen Bedeutungen und diffizilen Unterschiede werden später erklärt.

L-Werte und R-Werte. Wie bereits in Abschnitt 2.1.2 erwähnt, hat in einer Zuweisung der Ausdruck links vom Zuweisungsoperator eine andere Bedeutung als der rechts davon. Ein Ausdruck, der die Adresse einer im Speicher angelegte Variable bezeichnet, wird *L-Wert* genannt. Das „L“ steht dabei einfach nur für „links vom Zuweisungsoperator“. Der Begriff wird hauptsächlich in der Programmiersprache C benutzt, wo es vielfältige Möglichkeiten für Berechnungen mit Adressen gibt. In Java ist die Verwendung von L-Werten dagegen stark eingeschränkt. Außer als Operanden einiger Operatoren (einschließlich des linken Operanden des Zuweisungsoperators) können sie nirgends vorkommen.

Als L-Wert ist in Java nur ein Variablenname, einschließlich des Namens einer Variablen in einem Objekt, oder ein Array-Element verwendbar (Arrays sowie Variablen in Objekten werden wir später behandeln). Beispielsweise kann der Ausdruck $x+1$ nicht als L-Wert verwendet werden, da er keine Speicheradresse bezeichnet.

Steht ein Variablenname rechts vom Zuweisungsoperator, so entspricht der Name dem Wert bzw. *R-Wert* der Variablen. R-Werte kommen nicht nur rechts von Zuweisungsoperatoren vor, sondern jeder Ausdruck, der kein L-Wert ist, ist ein R-Wert.

Wird eine Variable mit dem Modifier `final` deklariert, darf ihr L-Wert nicht verwendet werden. Auf der linken Seite einer Zuweisung darf eine solche Variable genauso wenig stehen wie als irgendein anderer Operand,

der ein L-Wert sein muss. Beispielsweise ist `(x+1)++` keinesfalls erlaubt, da der Postfix-Operator `++` einen L-Wert als Operanden verlangt. Dagegen ist `x++` erlaubt, falls `x` nicht als `final` deklariert wurde, bei Deklaration als `final` jedoch verboten.

Ausdrucksanweisungen. Anweisungen sind Konstrukte, die den Zustand eines Programms dynamisch ändern oder den Programmfluss steuern. Im Gegensatz zu Ausdrücken liefern Anweisungen im Allgemeinen keine Werte. Beispielsweise sind `while`-Schleifen und `if`-Anweisungen keine Ausdrücke.

Es gibt aber auch Ausdrücke, die zugleich Anweisungen sind. Das bedeutet, sie liefern einen Wert und haben zusätzlich einen *Seiteneffekt*, nämlich das Ändern des Wertes einer Variablen. In vielen Fällen wird der Wert des Ausdrucks gar nicht weiter benötigt, sondern der Ausdruck wird nur wegen seines Seiteneffekts eingesetzt. Diese *Ausdrucksanweisungen* ermöglichen eine kurze Schreibweise. Allerdings wird das Ausnutzen von Seiteneffekten in komplexeren Ausdrücken oft als schlechter Programmierstil angesehen, weil die Programme dadurch unübersichtlich werden können. Folgendes Beispiel zeigt eine Ausdrucksanweisung in der zweiten Zeile:

```
int x = 5;
int y = x++;
```

Die erste Anweisung ist eine Variablendeklaration, wobei `x` der Wert 5 zugewiesen wird. In der zweiten Zeile wird eine weitere Variable `y` deklariert und ebenfalls ein Wert zugewiesen, der sich folgendermaßen ergibt: Die Ausdrucksanweisung `x++` liefert den Wert von `x`, also 5 zurück und erhöht danach als Seiteneffekt den Wert der Variablen `x` um eins, sodass nach Durchführung dieser beiden Programmzeilen `x` den Wert 6 und `y` den Wert 5 hat. Ein weiteres Beispiel für eine Ausdrucksanweisung ist die Zuweisung selbst – siehe Abschnitt 2.2.2.

Auswertung von Ausdrücken. Ein Operand eines Operators kann irgendein Ausdruck sein, wodurch man Ausdrücke zu komplexeren Ausdrücken verschachteln kann. Bei der Auswertung eines verschachtelten Ausdrucks spielt die Auswertungsreihenfolge eine wichtige Rolle. Beispielsweise wird beim Ausdruck

```
3 - 4 * 2
```

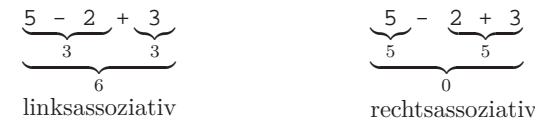


Abbildung 2.9: Es gibt zwei mögliche Reihenfolgen, in denen die Operatoren im Ausdruck `5 - 2 + 3` verknüpft werden könnten. Java verwendet für zweistellige arithmetische Operatoren die *linksassoziative* Reihenfolge.

zunächst der Teilausdruck mit dem Multiplikationsoperator `4*2` ausgewertet, bevor das Ergebnis der Subtraktion berechnet wird. Das Ergebnis ist also `-5` und nicht `-2`. Wie für die Mathematik in der Schule gilt: „Punktrechnung geht vor Strichrechnung.“ Die zweistelligen Operatoren `*` (Multiplikation) und `/` (Division) haben eine höhere *Priorität* (Vorrangstufe) als die zweistelligen Operatoren `+` und `-`.

Die *Assoziativität* eines Operators bestimmt die Reihenfolge, in der Operatoren und Operanden verknüpft werden, wenn ein Ausdruck aus mehreren Operatoren gleicher Priorität zusammengesetzt ist. Man unterscheidet links- rechts- und nicht-assoziative Verknüpfungen. Nichtassoziativ bedeutet einfach nur, dass eine Verknüpfung nicht erlaubt ist, falls mehrere Operatoren die gleiche Priorität haben. Im linksassoziativen Fall werden Ausdrücke von links nach rechts aufgelöst, im rechtsassoziativen Fall von rechts nach links. Abbildung 2.9 zeigt eine links- und rechtsassoziative Verknüpfung im Vergleich. Verknüpfungen arithmetischer Operatoren sind in Java immer linksassoziativ, da dies eher der üblichen Intuition entspricht.

Bleibt noch eine Regel, die bei der Auswertung von Ausdrücken zu beachten ist, nämlich die *Auswertungsreihenfolge der Operanden* bei zweistelligen Operatoren. Die Reihenfolge der Auswertung der Operanden ist generell von links nach rechts. Der Unterschied zur Assoziativität wird an folgendem Beispiel klar: Der Ausdruck

```
x++ - x
```

wendet den zweistelligen Operator `-` auf die Operanden `x++` und `x` an, ein und dieselbe Variable kommt also im linken und rechten Operanden vor. Links steht eine Ausdrucksanweisung, die den Wert von `x` liefert und den Seiteneffekt hat, dass `x` um eins erhöht wird. Der Wert des linken Operanden ist also der Wert von `x` vor der Erhöhung. Bevor die Subtrak-

tion durchgeführt werden kann, müssen die beiden Operanden ausgewertet werden. Dabei wird immer zuerst der linke Operand ausgewertet, das heißt, dessen Seiteneffekte müssen bereits passiert sein, bevor der rechte Operand ausgewertet wird. Die Auswertung des rechten Operanden liefert also den Wert von `x` nach der Erhöhung. Aufgrund der Auswertungsreihenfolge wird der gesamte Ausdruck immer zu `-1` ausgewertet.

Obwohl Priorität, Assoziativität und Auswertungsreihenfolge in Java für alle Operatoren ganz klar festgelegt sind, ist es für Menschen oft gar nicht leicht, die Auswertung komplexerer Ausdrücke richtig nachzuvollziehen. Im Sinne einfacher Lesbarkeit sollten wir daher alle Ausdrücke vermeiden, die schwer nachvollziehbar sind. Eine einfache und empfehlenswerte Technik zur Verbesserung der Lesbarkeit ist die Verwendung von Klammern. Beispielsweise bietet der Ausdruck

```
3 - (4 * 2)
```

keinerlei Interpretationsspielraum, auch wenn man die Operatorprioritäten nicht im Kopf hat. Allerdings können Klammern die Lesbarkeit hinsichtlich von Seiteneffekten nicht verbessern. In diesem Zusammenhang ist es hilfreich, statt einer Anweisung, die große und komplexe Ausdrücke enthält, mehrere Anweisungen mit einfacheren Ausdrücken zu verwenden und sparsam mit Seiteneffekten umzugehen.

2.2.2 Operatoren in Java

Im Folgenden geben wir einen Überblick über die gängigsten Operatoren in Java. Wir werden nicht auf alle Operatoren eingehen, sondern vor allem die einfachsten Operatoren überspringen. Detaillierte Beschreibungen aller Operatoren finden sie zum Beispiel in *The Java Language Specification*³ und in zahlreichen Java-Büchern aus einer Buchhandlung oder Bibliothek.

Zweistellige arithmetische Operatoren. Dazu zählen *Addition* „+“, *Subtraktion* „-“, *Multiplikation* „*“, *Division* „/“ und der *Restwertoperator* „%“, der auch Modulusoperator genannt wird. Alle diese Operatoren sind Infix-Operatoren. Die Operanden können beliebige numerische Typen haben (sowohl ganze Zahlen als auch Fließkommazahlen).

In Java ist es nicht notwendig, dass beide Operanden den gleichen numerischen Datentyp haben. Im Fall von unterschiedlichen Typen wandelt der

Compiler selbstständig vor der Anwendung der Operation alle Operanden in den umfassenderen gemeinsamen Datentyp um. Wenn der Ausdruck zum Beispiel einen `int`- und einen `double`-Operanden hat, wird der `int`-Operand in einen `double`-Wert umgewandelt, bevor der Operator angewendet wird. Nach welchen allgemeinen Regeln diese Typumwandlungen erfolgen, wird in Abschnitt 2.2.3 besprochen.

Der *Divisionsoperator* „/“ ist für ganze Zahlen und Fließkommazahlen unterschiedlich definiert. Bei der Division von ganzen Zahlen (das heißt, beide Operanden haben ganzzahlige Typen) wird zum Wert 0 hin ab- oder aufgerundet, sprich die Nachkommastellen werden verworfen, und das Ergebnis ist wieder eine ganze Zahl. Bei Operanden, die Fließkommazahlen sind, ist das Ergebnis wieder eine Fließkommazahl:

```
3.0 / 2 ergibt 1.5 (2 wird zu Fließkommazahl)
3 / 2.0 ergibt 1.5 (3 wird zu Fließkommazahl)
3 / 2 ergibt 1 (ganzzahlig, gegen 0 gerundet)
```

Die Division durch null ist in der Mathematik undefiniert. Es gibt keinen vernünftigen Wert, den man dem Ergebnis einer solchen Division geben könnte. Wenn wir in Java eine Ganzzahldivision mit dem Divisor 0 durchzuführen versuchen, so wird zur Laufzeit des Programmes eine `ArithmeticException` geworfen – siehe Abschnitte 5.4.1 und 5.5 – was ohne besondere Vorkehrungen zum Programmabbruch führt.

Bei Fließkommazahlen liefert die Division durch 0.0 jedoch ein spezielles Ergebnis, nämlich `Infinity` (Unendlich) oder `-Infinity` (negativ Unendlich) – siehe zweiten Teil des Skriptums. Als Spezialfall liefert der Ausdruck `0.0/0.0` den Wert `NaN` (*Not A Number*), der eigentlich *keinen Wert* repräsentiert. Die speziellen Werte `Infinity`, `-Infinity` und `NaN` sind zwar Instanzen von `double` und `float`, aber keine Zahlen im engeren Sinn. Tritt einer dieser Werte als Operand in einem Ausdruck auf, so ist auch das Ergebnis einer dieser Werte. Man kann mit diesen speziellen Werten also nicht mehr normal weiterrechnen.

Der *Restwertoperator* „%“ liefert den Rest der Ganzzahldivision. Beispielsweise liefert der Ausdruck `5%2` den Wert 1, weil bei der Ganzzahldivision `5/2` als Rest 1 übrig bleibt. Der Rest kann in Java auch negativ werden, nämlich genau dann, wenn der Dividend negativ ist. Daher ergibt `(-5)%2` den Wert `-1`.

In der Praxis wird der Restwertoperator häufig auf ganzen Zahlen eingesetzt, wie wir bereits an einem Beispiel in Abschnitt 1.1 gesehen haben, jedoch kaum auf Fließkommazahlen. Dennoch ist der Operator auch auf

³Der Inhalt dieses Buches von den Java-Entwicklern kommt einer offiziellen Definition der Sprache am nächsten – siehe <http://java.sun.com/docs/books/jls/index.html>.

Fließkommazahlen definiert. Wenn einer der Operanden eine Fließkommazahl ist, wird als Ergebnis wieder eine Fließkommazahl geliefert. Ein Beispiel:

```
(-5) % 2.25 ergibt -0.5
```

Dieses Ergebnis erhält man, da $5 = 2 * 2.25 + 0.5$ gilt und der Dividend negativ ist. Der Wert 2 ergibt sich daraus, dass 2.25 in 5 zwei mal vollständig enthalten ist, und 0.5 bleibt übrig, wenn man 2.25 zwei mal von 5 abzieht.

Bezüglich der Division durch null verhält sich der Restwertoperator ähnlich wie der Divisionsoperator: Bei Anwendung auf ganze Zahlen wird eine `ArithmeticException` geworfen, und bei Fließkommazahlen ist das Ergebnis NaN.

Verkettungs-Operator. Das Operatorsymbol „+“ wird nicht nur zur Addition, sondern ebenso als zweistelliger Infix-Operator zur Verkettung von Zeichenketten benutzt. Ein Beispiel:

```
String s1 = "Hallo ";
String s2 = s1 + "Karl";
System.out.println(s2); //gibt "Hallo Karl" aus
```

Man sagt, der Infix-Operator „+“ ist *überladen*. In einem Ausdruck `x+y` wird „+“ als

- Additions-Operator aufgefasst, wenn sowohl `x` als auch `y` Zahlen sind, also einen numerischen Typ haben,
- Verkettungs-Operator aufgefasst, wenn `x` oder `y` vom Typ `String` ist (oder beide vom Typ `String` sind).

In allen anderen Fällen liefert der Compiler eine Fehlermeldung. Wenn nur ein Operand vom Typ `String` und der andere von irgendeinem anderen Typ ist, dann wird der andere Operand in eine Zeichenkette umgewandelt, bevor die beiden Zeichenketten verkettet werden. Beispielsweise gibt

```
System.out.println("3 + 4 = " + (3 + 4));
```

die Zeichenkette `"3 + 4 = 7"` aus: Der linke Operand von „+“ außerhalb der Klammern ist eine Zeichenkette (die zufällig das Zeichen `'+'` enthält). Daher ist dieser Operator der Verkettungs-Operator. Im rechten Operanden wird der Additions-Operator auf 3 und 4 angewandt, und

das Ergebnis 7 wird in die Zeichenkette `"7"` umgewandelt. Versuchen Sie als Übungsaufgabe zu bestimmen, welche Ausgabe die Anweisung (ohne Klammerung des arithmetischen Ausdrucks)

```
System.out.println("3 + 4 = " + 3 + 4);
```

erzeugen würde.

Einstellige arithmetische Operatoren. Zu den unären arithmetischen Operatoren zählen der *positive* „+“ und *negative* „-“ *Vorzeichenoperator*. Der positive Vorzeichenoperator wird selten verwendet, da er einfach den Wert seines Operanden liefert. So liefert `+x` einfach den Wert von `x`. Der negative Vorzeichenoperator kehrt das Vorzeichen seines Operanden um. So liefert beispielsweise `-x` den negierten Wert von `x` oder

```
- -5
```

den Wert 5. Das Leerzeichen zwischen den beiden Vorzeichenoperatoren ist hier wichtig, um sie vom Präfix-Dekrementoperator `--` zu unterscheiden; `--5` wäre kein gültiger Ausdruck.

Der *Präfix-* bzw. *Postfix-Inkrementoperator* „++“ und der *Präfix-* bzw. *Postfix-Dekrementoperator* „--“ gehören ebenfalls zu den unären arithmetischen Operatoren. Im Gegensatz zu den anderen arithmetischen Operatoren haben diese Operatoren einen Seiteneffekt: Sie liefern nicht nur einen Wert zurück, sondern verändern den Operanden auch. Daher müssen Operanden dieser Operatoren L-Werte sein.

Die Inkrementoperatoren erhöhen den Wert um eins, die Dekrementoperatoren verringern ihn um eins. Als Präfixoperatoren ändern sie zuerst den Wert und liefern dann den veränderten Wert zurück. Wenn die Variable `x` beispielsweise den Wert 5 enthält, liefert `--x` als Ergebnis 4 und ändert den Wert von `x` auf 4:

```
int x = 5;
int y = --x; // x hat den Wert 4, y hat den Wert 4
```

Analog liefert, wenn `x` den Wert 5 enthält, `++x` als Ergebnis 6 und ändert den Wert von `x` auf 6. Als Postfixoperatoren liefern `++` und `--` zuerst den unveränderten Wert zurück und ändern ihn erst dann:

```
int x = 5;
int y = x--; // x hat den Wert 4, y hat den Wert 5
```

Zuweisungsoperatoren. Zuweisungs-Operatoren sind binäre Operatoren mit Seiteneffekten. Der linke Operand muss bei allen Zuweisungsoperatoren ein L-Wert sein, der rechte Operand jedoch nicht. Zu den Zuweisungsoperatoren gehören der *einfache Zuweisungsoperator* „=“ sowie die *kombinierten Zuweisungsoperatoren*, deren Symbol sich aus einem entsprechenden binären arithmetischen oder Bit-Operatorsymbol mit nachfolgendem = zusammensetzt. Beispielsweise ist „+=“ der Additions-Zuweisungsoperator.

Der *einfache Zuweisungsoperator* liefert als Rückgabewert den Wert des rechten Operanden. Da eine Zuweisung auch ein Ausdruck ist, ist beispielsweise folgende Schreibweise möglich:

```
int x, y, z;
x = y = z = 1;
```

Zuerst wird der Variablen *z* der Wert 1 zugewiesen. Die Zuweisung *z*=1 liefert als Ausdruck den zugewiesenen Wert. Damit erhält *y* ebenfalls den Wert 1. Zuletzt wird „=“ ganz links ausgewertet, und damit bekommt auch *x* den Wert 1. Der Ausdruck wird von rechts nach links wie in *x*=(*y*=(*z*=1)) abgearbeitet. Das heißt, „=“ ist *rechtsassoziativ*.

Zuweisungsoperatoren haben eine sehr niedrige Priorität. Daher ist beispielsweise die Schreibweise

```
y = x + 1;
```

möglich, ohne dass der arithmetische Ausdruck auf der rechten Seite in Klammern gesetzt werden muss. Zuerst wird der arithmetische Operator angewendet und danach das Ergebnis zugewiesen.

Der Additions-Zuweisungsoperator „+=“ kombiniert eine Zuweisung mit einer Addition: Der Wert des rechten Operanden wird zum linken Operanden hinzuaddiert. Beispiele:

```
x += 1;      gleichbedeutend mit x = x + 1;
x += y = 1;  gleichbedeutend mit x = x + (y = 1);
```

Das letzte Beispiel zeigt, dass der rechte Operand immer zuerst ausgewertet wird, trotz höherer Priorität der Addition. Die Semantik der kombinierten Zuweisungsoperatoren ist also allgemein:

```
L op= R;  gleichbedeutend mit L = L op (R);
```

x += 2	// x = x + 2
x -= 3	// x = x - 3
x *= 4	// x = x * 4
x /= 5	// x = x / 5
x %= 6	// x = x % 6
x &= 255	// x = x & 255
x = 8	// x = x 8
x ^= y	// x = x ^ y
x <<= 1	// x = x << 1
x >>= 1	// x = x >> 1
x >>= 3	// x = x >>> 3

Abbildung 2.10: Kombinierte Zuweisungsoperatoren anhand von Beispielen

wobei *op* arithmetischer Operator oder Bit-Operator und *L* bzw. *R* der linke bzw. rechte Operand ist.

Abbildung 2.10 zeigt eine Liste aller kombinierten Zuweisungsoperatoren in Beispielen. Die letzten 6 Zeilen in dieser Liste stellen Bitoperatoren dar, die weiter unten besprochen werden.

Relationale Operatoren. Das sind binäre Infix-Operatoren die als Operanden numerische Ausdrücke haben und einen *Wahrheitswert*, also einen Wert vom Typ `boolean` zurückliefern. Die relationalen Operatoren werden daher manchmal auch als *Boolesche Operatoren* bezeichnet. Es gibt den *Kleineroperator* „<“, *Größeroperator* „>“, den *Kleingleichoperator* „<=“ und den *Größergleichoperator* „>=“. Weiters gibt es *Gleichheitsoperatoren* für den Test auf Gleichheit „==“ und Ungleichheit „!=“. Letztere haben eine geringere Priorität als die anderen relationalen Operatoren. Die Gleichheitsoperatoren können nicht nur auf numerische Operanden angewendet werden, sondern auch auf Boolesche Werte und auf Referenzen. Dieses Thema wird später aufgegriffen. Abbildung 2.11 zeigt einige Beispiele für die Anwendung relationaler Operatoren.

Logische Operatoren. Logische Operatoren verknüpfen Boolesche Werte (vom Typ `boolean`) zu einem Booleschen Ergebniswert. Es gibt zweistellige Infix-Operatoren für das *logische UND* „&&“, das *logische ODER* „||“ und das *logische Exklusiv-ODER* „^“ sowie einen einstelligen Präfix-


```
2 == 2           // true
2 == 3           // false
3 < 2            // false
1 < 2            // true
3 < 5            // true
3 > 3            // false
3 >= 3           // true
3 <= 3           // true
2 <= 3           // true
2 != 3           // true
2 != 2           // false
true == false    // false
true == 2 <= 3   // true
```

Abbildung 2.11: Relationale Operatoren anhand von Beispielen

Ausdruck	Wert	Ausdruck	Wert
false && false	false	false false	false
false && true	false	false true	true
true && false	false	true false	true
true && true	true	true true	true

Ausdruck	Wert	Ausdruck	Wert
false ^ false	false	!false	true
false ^ true	true	!true	false
true ^ false	true		
true ^ true	false		

Abbildung 2.12: Definition der logischen Operatoren

Operator, den *Negationsoperator* „!“: Die Funktionalität der Operatoren ist anhand der Verknüpfungen in Abbildung 2.12 vollständig definiert.

Unter den logischen Operatoren hat „!“ die höchste Priorität, gefolgt von „^“. Danach kommt „&&“ mit einer geringeren Priorität und schließlich „||“ mit der geringsten Priorität. Die Priorität aller logischer Operatoren mit Ausnahme der Negation „!“ ist geringer als die der Booleschen Operatoren. Der Ausdruck „`x < 5 && y == 0 || y == x`“ ist daher

gleichbedeutend mit „ $((x < 5) \ \&\& \ (y == 0)) \ || \ (y == x)$ “.

Bei der Verwendung der Operatoren „&&“ und „|“ ist zu beachten, dass der rechte Ausdruck nicht immer ausgewertet wird. Er wird nur dann ausgewertet, wenn er das Ergebnis noch beeinflussen kann. Ist bei „&&“ der linke Ausdruck falsch, kann die Verknüpfung bereits nicht mehr wahr werden, daher wird kein weiterer Operand ausgewertet, sondern gleich `false` zurückgegeben. Ist bei „|“ der linke Ausdruck wahr, kann der Ausdruck nicht mehr falsch werden, daher wird auch hier kein weiterer Operand ausgewertet, sondern gleich `true` zurückgegeben. Auf diese Weise kann der Compiler bzw. die Laufzeitumgebung den Programmfluss abkürzen. Man nennt diese Operatoren daher auch *Kurzschlussoperatoren*. Diese Optimierungen müssen speziell dann beachtet werden, wenn die Operanden Ausdrucksanweisungen mit Seiteneffekten sind.

Sollen bei UND- und ODER-Operationen in Ausnahmefällen beide Operanden ausgewertet werden, kann man statt der Kurzschlussoperatoren die Operatoren „&“ und „|“ verwenden. Normalerweise führt man mit diesen Operatoren jedoch Bit-Operationen aus.

Bit-Operatoren. Diese führen Manipulationen auf dem Bitmuster eines Wertes durch. Es gibt in Java vier *logische Bit-Operatoren* und drei *Shift-Operatoren*. Die logischen Bit-Operatoren führen eine bitweise logische Operation durch. Dabei entspricht ein Bit im Zustand 1 einem `true` und im Zustand 0 einem `false`.

In den folgenden Beispielen verwenden wir als Operanden die `int`-Variablen `m` und `n` aus Abbildung 2.3. Die entsprechenden Darstellungen der Werte als Bitmuster im Binärzahlensystem werden rechts als Kommentar angegeben. Ein Beispiel für den bitweisen *UND*-Operator „&“:

[illegible]

In diesem Beispiel wird `r` als Ergebnis der bitweisen UND-Verknüpfung von `m` und `n` der Wert 3 zugewiesen. Der Wert ergibt sich, weil die beiden niedrigstwertigen Bits die einzigen sind, die im Bitmuster von `m` und `n` auf 1 gesetzt sind. Das `int`-Bitmuster, in dem diese beiden Bits als einzige auf 1 gesetzt sind, entspricht dem Wert 3.

Ein Beispiel für den *bitweisen ODER-Operator* „|“:

Operatoren	Priorität	Assoziativität	Bedeutung
[]	1	links	Arrayzugriff
()	1	links	Methodenaufruf
.	1	links	Komponentenzugriff
++, --	1	links	Postinkrement, Postdekrement
++, --	2	rechts	Präinkrement, Prädekrement
+, -	2	rechts	unäres Plus und Minus
~	2	rechts	bitweises Komplement
!	2	rechts	logisches Komplement
(Typ)	3	rechts	Cast
new	3	rechts	Erzeugung (siehe Kapitel 3)
*, /, %	4	links	Multiplikation, Division, Rest
+, -	5	links	Addition und Subtraktion
+	5	links	Stringverkettung
<<	6	links	Linksshift
>>	6	links	Rechtsshift mit Vorzeichenerweiterung
>>>	6	links	Rechtsshift ohne Vorzeichenerw.
<, <=, >, >=	7	links	numerische Vergleiche
instanceof	7	links	Typvergleich (siehe Kapitel 3)
==, !=	8	links	Gleich-/Ungleichheit
&	9	links	bitweises/logisches UND
^	10	links	bitweises/logisches exklusives ODER
	11	links	bitweises/logisches ODER
&&	12	links	logisches konditionales UND
	13	links	logisches konditionales ODER
?:	14	rechts	Bedingungsoperator
=	15	rechts	Zuweisung
*=, /=, %=, +=, -=, <=, >=, >>=, &=, ^=, =	16	rechts	kombinierter Zuweisungsoperator

Tabelle 2.13: Übersicht über Operatoren

2.2.3 Typumwandlungen und Literale

Obwohl Java eine Programmiersprache mit strenger Typisierung ist, wurde darauf geachtet, dass die Typisierung nicht all zu einschränkend wirkt. So erlauben Typumwandlungen das Verknüpfen von Operanden unterschiedlichen Typs. Zum Beispiel werden die folgenden Zeilen vom Compiler akzeptiert, obwohl darin Werte mehrerer elementarer Typen gemischt vorkommen:

```
int m = 14;
```

```
long n = m;
long r = 10;
```

In der zweiten Zeile wird implizit der `int`-Wert von `m` in einen `long`-Wert umgewandelt, bevor der Wert der Variablen `n` zugewiesen wird. Somit ist der Wert der Zuweisung vom Typ `long`. Genauso wird in der dritten Zeile das `int`-Literal `10` implizit in eine Zahl vom Typ `long` umgewandelt. Das entsprechende `long`-Literal wäre `10L`.

Solche Typumwandlungen laufen nach bestimmten Regeln ab. Man unterscheidet zwei Arten von Typumwandlungen:

Erweiternde Typumwandlungen: Diese erfolgen von einem kleineren hin zu einem größeren Datentyp. Der Wert des kleineren Datentyps ist als Wert des größeren Datentyps immer darstellbar. Allerdings kann es zu Fehlern auf Grund der Darstellung der Werte kommen, z.B. zu Rundungsfehlern bei der Umwandlung von `int` nach `float`, da die Fließkommazahlen nicht beliebig dicht aufeinander folgen. Die erweiternden Typumwandlungen werden vom Compiler automatisch durchgeführt – siehe obiges Beispiel.

Einschränkende Typumwandlungen: Diese erfolgen von einem größeren hin zu einem kleineren Datentyp. Dabei kann es zu einem Genauigkeitsverlust kommen, da der Wert des größeren Typs im Allgemeinen nicht als Wert des kleineren Typs darstellbar ist. Einschränkende Typumwandlungen werden nicht automatisch durchgeführt. Stattdessen muss man dem Compiler explizit mitteilen, dass eine solche Umwandlung erwünscht ist und man einen etwaigen Genauigkeitsverlust in Kauf nimmt. Der Operator für eine explizite Typumwandlung ist der `Cast`-Operator.

Eine explizite Typumwandlung mit dem `cast`-Operator hat folgende Form:

```
float x = 13.5f;
int y = (int) x; //Rundung gegen Null: y == 13
```

In Tabelle 2.14 sind Typumwandlungen für elementare Datentypen dargestellt. Man beachte, dass – abweichend von der allgemeinen Regel – von `byte` und `short` zu `char` nur explizite Typumwandlungen erfolgen können. Ausdrücke vom Typ `boolean` können weder implizit noch explizit umgewandelt werden und kommen daher in der Tabelle nicht vor.

Der `Cast`-Operator ist auch für Referenztypen sinnvoll, hat dort aber eine andere Bedeutung, wie wir in Abschnitt 3.4.3 sehen werden.

zu von	byte	short	char	int	long	float	double
double	↘	↘	↘	↘	↘	↘	
float	↘	↘	↘	↘	↘		↗
long	↘	↘	↘	↘		↗	↗
int	↘	↘	↘		↗	↗	↗
char	↘	↘		↗	↗	↗	↗
short	↘		↘	↗	↗	↗	↗
byte		↗	↘	↗	↗	↗	↗

Tabelle 2.14: Typumwandlungen auf elementaren Typen: implizit bzw. erweitert (↗) versus explizit bzw. einschränkend (↘)

Erweiterung. Verknüpft man in einem Ausdruck Operanden der Typen `byte`, `short` oder `char` (also Typen mit weniger Bytes als `int`), so werden diese meist vor der Verknüpfung implizit in den Datentyp `int` umgewandelt. Diese implizite Umwandlung wird *Integer-Erweiterung* (*integral promotion*) genannt. Sie hat zur Folge, dass in manchen Fällen eine explizite Umwandlung erforderlich ist:

```
short x = 30;
short y = +x; // Compilerfehler: +x liefert int
short y = (short) +x; // so geht es
```

Bei zweistelligen Operatoren werden die beiden Operanden in den größten gemeinsamen Typ umgewandelt. Dieser gemeinsame Typ bestimmt auch den Ergebnistypen. Ein Beispiel:

```
1.0 + 5 * 3L;
```

Hier wird aufgrund seiner höheren Priorität zunächst der Multiplikationsoperator `*` angewendet. Dabei wird das `int`-Literal auf den Typ `long` gebracht und die Multiplikation liefert einen `long`-Wert. Dieser wird vor der Berechnung der Summe wegen des linken Operandentyps in einen `double`-Wert umgewandelt. Der Typ des Ausdrucks ist `double`.

Konstante Ausdrücke. Ein Ausdruck ist konstant, wenn bereits vom Compiler der Rückgabewert des Ausdrucks berechnet wird. Das ist nur möglich, wenn der Ausdruck keine Variablen enthält, denn Werte von Variablen,

die erst zur Laufzeit initialisiert werden und sich während der Programmausführung ändern können, kennt der Compiler ja nicht. Im Wesentlichen enthalten konstante Ausdrücke nur Literale, die möglicherweise über Operatoren miteinander verknüpft sind.

Steht in einer Zuweisung ein konstanter Ausdruck des Typs `int`, so kann dessen Wert auch einer Variablen eines kleineren Typs ohne explizite Typumwandlung zugewiesen werden, wenn der Wert ohne Informationsverlust in den Typ passt. Das wird statisch überprüft. Beispiel:

```
short n = 5 * 3; //wird übersetzt
short n = 5000 * 3; //Compilerfehler
```

Diese Regel zusammen mit Integer-Erweiterung bewirkt, dass wir keine eigenen Literale für `byte`- und `short`-Werte brauchen. Wir können dafür einfache `int`-Literale verwenden.

Die statische Überprüfung, ob konstante `int`-Werte ohne Informationsverlust in einen kleineren Typ passen, darf uns nicht zur Annahme verleiten, dass der Compiler generell auf mögliche Informationsverluste hinweist. Das ist leider nicht der Fall, wie man hier sieht:

```
long n = 2147483647 * 2L; // n == 4294967294L
long n = 2147483647 * 2; // n == -2L !Fehler!
```

Die Zahl 2147483647 ist gerade noch als `int`-Wert darstellbar, der doppelte Wert jedoch nicht mehr; der ist nur als `long`-Wert darstellbar. In der ersten dieser beiden Zeilen wird die Zahl vor der Multiplikation in einen `long`-Wert umgewandelt, weil der zweite Operand vom Typ `long` ist. Daher ist auch das Ergebnis vor der Zuweisung vom Typ `long`. In der zweiten Zeile werden dagegen zwei `int`-Werte miteinander multipliziert, und das Ergebnis ist der `int`-Wert `-2`, der zufällig durch Abschneiden nicht mehr darstellbarer Bits entsteht. Dieser falsche `int`-Wert wird schließlich zu `long` konvertiert und an `n` zugewiesen. Weder vom Compiler noch zur Laufzeit gibt es eine Fehlermeldung. Aus diesem Grund müssen wir beim Programmieren besonders darauf achten, dass alle unsere Werte in den Typen, die wir verwenden, darstellbar sind.

Literale. Fehler wie dieser entstehen leicht aus Unachtsamkeit, weil wir Literale der falschen Typen verwenden. Daher ist es wichtig, die Literale zu kennen und auf deren Typen zu achten.

Literale für `boolean` sind schnell aufgezählt: Es gibt nur die beiden Literale `false` und `true`.

Für ganze Zahlen werden dagegen mehrere Arten von Literalen unterstützt. Einerseits unterscheiden wir zwischen Literalen vom Typ `int` und solchen vom Typ `long`. Letztere sind gleich aufgebaut wie `int`-Literalen, enden aber mit `L` oder `l`; Klein- und Großschreibung spielt dafür keine Rolle. Eigene Literalen für `byte` und `short` gibt es nicht⁴. Sowohl `int`-als auch `long`-Literalen können in drei unterschiedlichen Zahlensystemen definiert werden:

- Meist verwenden wir Literalen im Dezimalsystem, also Zahlen auf der Basis von zehn. Alle (nicht durch andere Zeichen unterbrochenen) Ziffernfolgen, die *nicht* mit 0 beginnen, stellen Dezimalzahlen dar.
- Eine Ziffernfolge, die mit 0 beginnt, wird als *Oktalzahl* gesehen, also als Zahl auf der Basis von acht. In der Ziffernfolge dürfen nur die Ziffern 0 bis 7 vorkommen. Beispielsweise entspricht die Oktalzahl 024 der Dezimalzahl 20, da $2 \cdot 8^1 + 4 \cdot 8^0 = 2 \cdot 10^1 + 0 \cdot 10^0$ gilt. Aufgrund dieser Zahlendarstellung dürfen wir niemals führende Nullen vor Dezimalzahlen schreiben.
- *Hexadezimalzahlen*, das sind Zahlen auf der Basis von 16, beginnen mit 0x oder 0X gefolgt von einer Folge von Ziffern und den Buchstaben a bis f oder A bis F, wobei a und A für zehn, b und B für elf, und so weiter bis f und F für 15 stehen. Beispielsweise entspricht die Hexadezimalzahl 0x2B der Dezimalzahl 43, da $2 \cdot 16^1 + 11 \cdot 16^0 = 4 \cdot 10^1 + 3 \cdot 10^0$ gilt.

Auch für Fließkommazahlen gibt es mehrere Formen von Literalen. Literalen vom Typ `float` unterscheiden sich von denen vom Typ `double` durch ein hinten angehängtes `f` oder `F`. An ein Literal vom Typ `double` können wir hinten ein `d` oder `D` anhängen, müssen aber nicht. Weiters können wir zwei Formen von Fließkomma-Literalen unterscheiden:

- Die normale Darstellung beginnt mit einer ganzen Dezimalzahl gefolgt von „.“ und einer weiteren Dezimalzahl (möglicherweise mit führenden Nullen). Ein Beispiel ist 12.34. Abgesehen davon, dass wir, wie im englischsprachigen Raum üblich, einen Punkt statt dem im deutschsprachigen Raum üblichen Komma verwenden, entspricht diese Darstellung unserer Intuition.

<code>\b</code>	Backspace (Rückschritt)
<code>\f</code>	Formfeed (Seitenumbruch)
<code>\t</code>	horizontaler Tabulator
<code>\'</code>	einfaches Anführungszeichen
<code>\"</code>	doppeltes Anführungszeichen
<code>\\</code>	Backslash
<code>\r</code>	Cursor Return (Wagenrücklauf)
<code>\n</code>	Newline (Zeilenschaltung)
<code>\ooo</code>	ASCII-Code des Zeichens, 3-stellige Oktalzahl ooo
<code>\uxxxx</code>	Unicode des Zeichens, 4-stellige Hexadezimalzahl xxxx

Tabelle 2.15: Escape-Sequenzen für Zeichen-Literalen

- In der wissenschaftlichen Darstellung beginnt die Zahl wie in der normalen Darstellung, und danach folgt ein `e` oder `E` und der Exponent als positive oder negative ganze Dezimalzahl. Beispielsweise steht 12.34e3 für $12,34 \cdot 10^3$ und entspricht damit der Fließkommazahl 12340.0, und 12.34E-3 steht für $12,34 \cdot 10^{-3}$ und entspricht damit 0.01234.

Zeichen-Literalen sind in einfache Anführungszeichen gesetzte Zeichen wie z.B. `'x'`, `'&'` und `'3'`. Es gibt jedoch auch Zeichen, die nicht direkt über die Tastatur eingegeben und am Bildschirm dargestellt werden können. Für diese Zeichen gibt es spezielle Darstellungsformen, sogenannte *Escape-Sequenzen*, die in Tabelle 2.15 aufgelistet sind. Beispielsweise ist `'\n'` das Zeichen-Literal für die Zeilenumschaltung. Das Zeichen „\“ spielt die Rolle eines Escape-Zeichens, das heißt, spezielle Zeichen werden mit „\“ eingeleitet. Um Verwechslungen zu vermeiden ist das Zeichen-Literal für „\“ daher `'\\'` und nicht `'\'`. Jedes Zeichen entspricht auch einer Zahl. Zeichen, die Zahlen zwischen 0 und 255 (oder zwischen -128 und 127 wenn man das erste Bit als Vorzeichen interpretiert) entsprechen, können durch `'\ooo'` dargestellt werden, wobei ooo eine dreistellige Oktalzahl ist. Alle Zeichen können auch durch `'\uxxxx'` dargestellt werden, wobei xxxx eine vierstellige Hexadezimalzahl ist. Beispielsweise repräsentieren folgende Literalen dasselbe Zeichen: `'\101'`, `'\u0041'` und `'A'`.

Der einzige Referenztyp, für den es Literalen gibt, ist `String`. Literalen von Zeichenketten werden innerhalb doppelter Anführungszeichen angeschrieben, die in der gleichen Zeile enden, in der sie beginnen. Man

⁴So steht beispielsweise in der Anweisung `short n = 5;` rechts ein konstanter Ausdruck, dessen `int`-Wert implizit in einen `short`-Wert umgewandelt wird.

kann dieselben Escape-Sequenzen wie in Zeichen-Literalen verwenden. Beispielsweise ist "Hello!\nBye!" eine Zeichenkette, die sich bei der Ausgabe über zwei Zeilen erstreckt.

2.3 Blöcke und bedingte Anweisungen

Die Anweisungen eines Programmes werden grundsätzlich hintereinander in der Reihenfolge ausgeführt, in der Sie im Programm stehen. *Kontrollstrukturen* ermöglichen es, Anweisungen in logisch zusammenhängende Gruppen zusammenzufassen und die Reihenfolge der Ausführung zu ändern. Sie steuern den *Kontrollfluss* des Programms, das heißt, sie bestimmen, welche Anweisungen des Programms ausgeführt werden und in welcher Reihenfolge die Ausführung erfolgt. Blöcke und bedingte Anweisungen sind zwei Arten von Kontrollstrukturen. Weitere Kontrollstrukturen werden in den nächsten Abschnitten vorgestellt.

2.3.1 Blöcke

In bestimmten Situationen ist es notwendig, eine Sequenz von Anweisungen in einem *Anweisungsblock* (oft einfach nur *Block* genannt) zusammenzufassen. Ein Block darf (fast⁵) überall dort eingesetzt werden, wo eine einzelne Anweisung erlaubt ist. Ein Block *ist* eine Anweisung, wenn auch eine aus mehreren Anweisungen zusammengesetzte.

Die Begrenzungssymbole für einen Block sind die geschwungenen Klammern { und }. Ein Block umfasst meist mehrere Anweisungen, jede in einer eigenen Zeile, und wird in folgender Form angeschrieben:

```
{
    Anweisung1
    Anweisung2
    ⋮
    Anweisungk
}
```

Da Blöcke selbst Anweisungen sind, kann in einem Block ein weiterer Block enthalten sein. Es entstehen verschachtelte Unterblöcke.

⁵wir werden später Ausnahmen kennenlernen

Variablen, die in einem Anweisungsblock deklariert werden, nennt man *lokale* Variablen. Lokale Variablen können an jeder Stelle im Block deklariert werden, nicht nur am Anfang. Auf sie kann in allen auf die Deklaration folgenden Anweisungen innerhalb des Blocks zugegriffen werden. Die Variable hört auf zu existieren, sobald der Kontrollfluss das Ende des Blocks erreicht. In Abschnitt 2.1.2 wurden *Gültigkeitsbereich* und *Lebensdauer* einer Variablen erwähnt. Der Gültigkeitsbereich ist jener Teil des Programmes, in dem sich der Name einer Variablen auf diese Variable bezieht – für eine lokale Variable also der Programmabschnitt zwischen der Deklaration und dem Ende des Blocks. Die Lebensdauer umfasst denjenigen Zeitabschnitt der Programmausführung, in dem die Variable existiert. In diesem Zeitabschnitt ist für die Variable ein Speicherplatz reserviert. Neben dem Gültigkeitsbereich ist also auch die Lebensdauer lokaler Variablen eingeschränkt.

Nach ihrer Deklaration muss einer lokalen Variable zunächst ein Wert zugewiesen werden, bevor sie gelesen werden kann. Vergisst man darauf, wird der Compiler beim Übersetzen des Programms einen Fehler melden.

Die folgenden Programmausschnitte geben einfache Beispiele für fehlerhafte Blöcke. Die Blöcke würden sich auch dann nicht übersetzen lassen, wenn eine vollständige Programmorganisation vorhanden wäre.

Der Block in Listing 2.16 ist fehlerhaft, weil für die Auswertung eines Ausdrucks der Wert der Variablen *m* benötigt wird, aber *m* nicht initialisiert wurde. Daher gibt es diesen Wert nicht.

Listing 2.18 zeigt zwei verschachtelte Anweisungsblöcke. Die Variable *m* ist auch im inneren Block gültig, daher ist die Zuweisung von *m* an *o* erlaubt. Jedoch ist der Gültigkeitsbereich von *o* auf den inneren Block beschränkt, sodass *o* außerhalb des inneren Blocks nicht zugreifbar ist.

Listing 2.17 gibt schließlich ein Beispiel dafür, dass alle gültigen Namen lokaler Variablen in einem Block verschieden sein müssen. Im Gegensatz zur Programmiersprache C gilt das in Java auch dann, wenn eine Variable in einem inneren Block deklariert wird.

2.3.2 Selektion mit if-else

Die grundlegendste Form einer bedingten Anweisung ist die *if*-Anweisung, die auf Grund einer Bedingung entscheidet, ob die darauffolgende Anweisung ausgeführt wird, oder nicht. Die *if*-Anweisung hat folgende Syntax:

Listing 2.16: Fehlende Initialisierung

```
{
    int m;
    int n;

    n = 1;
    n = n + m; //Fehler
}
```

Listing 2.17: Namenskonflikt

```
{
    int m = 1;
    int n = 2;

    {
        int m; //Fehler
        m = n;
    }
}
```

```
if ( boolescher Ausdruck )
    Anweisung1
else
    Anweisung2
```

Liefert die Auswertung des booleschen Ausdrucks den Wert `true`, dann wird nur *Anweisung1* ausgeführt. Sonst wird nur *Anweisung2* ausgeführt. Der `else`-Zweig kann auch weggelassen werden, dann hat für den Fall, dass der boolesche Ausdruck `false` liefert, die `if`-Anweisung insgesamt keine weitere Auswirkung.

Listing 2.19 und 2.20 zeigen zwei äquivalente Programmfragmente als Beispiel für den Einsatz einer `if`-Anweisung, einmal mit und einmal ohne `else`-Zweig. Beide Programme berechnen das Gehalt für die in einer Woche geleistete Arbeitszeit unter Berücksichtigung von Überstunden, d.h. Arbeitszeitstunden, die über 40 Stunden hinausgehen. Für Überstunden wird der eineinhalbfache Stundenlohn berechnet. Wie in Listing 2.19 ge-

Listing 2.18: Zugriff auf eine Variable außerhalb ihres Gültigkeitsbereichs

```
{
    int m;
    int n;

    m = 1;
    n = 2;

    {
        int o;
        o = m;
    }

    System.out.println(o); // Fehler
}
```

zeigt schreibt man aus pragmatischen Gründen meist Klammern um die einzelnen Programmzweige, damit spätere Erweiterungen vereinfacht werden. Wie in Listing 2.20 funktionieren die Programme aber auch ohne Klammern, solange ein Zweig aus genau einer Anweisung besteht.

Listing 2.19: Beispiel für eine `if`-Anweisung mit `else`-Zweig

```
if (wochenstunden > 40) {
    gehalt = stundenlohn*40
           + 1.5*stundenlohn*(wochenstunden - 40);
} else {
    gehalt = stundenlohn*wochenstunden;
}
```

Listing 2.20: Beispiel für eine `if`-Anweisung ohne `else`-Zweig

```
bonus = 0;
gehalt = wochenstunden * stundenlohn;
if (wochenstunden > 40)
    bonus = 0.5*stundenlohn*(wochenstunden - 40);
gehalt = gehalt + bonus;
```

Jeder Zweig einer `if`-Anweisung kann wieder eine `if`-Anweisung sein. Dadurch wird eine Mehrfach-Selektion ermöglicht. Ein Beispiel ist in Listing 2.21 zu sehen. Diese Verschachtelung von `if`-Anweisungen berechnet das Maximum von 3 Werten, die in den Variablen `a`, `b` und `c` gespeichert sind. Nach der Durchführung dieser Anweisung enthält die Variable `max` das Maximum von `a`, `b` und `c`. Versuchen Sie die Anweisung für 3 beliebige Werte von `a`, `b` und `c` nachzuvollziehen. Überlegen Sie sich welche Zusicherungen für jeden der Programmzweige gelten.

Listing 2.21: Beispiel für eine verschachtelte `if`-Anweisungen

```
if(a > b) {
    if(a > c) {
        max = a;
    } else {
        max = c;
    }
} else {
    if(b > c) {
        max = b;
    } else {
        max = c;
    }
}
```

Listing 2.22: Beispiel für eine Mehrfach-Selektion mit `if`-Anweisungen

```
//Precondition: 0 <= punkte <=100

if (punkte >= 90) {
    note = "sehr gut";
} else {
    if (punkte >= 80) {
        note = "gut";
    } else {
        if (punkte >= 68) {
            note = "befriedigend";
        } else {
            if (punkte >= 50) {
                note = "genügend";
            } else {
                note = "nicht genügend";
            }
        }
    }
}
```

2.3.3 Mehrfach-Selektion mit der `switch`-Anweisung

Mit der `if`-Anweisung können alle Arten von Verzweigungen bewerkstelligt werden. Auch Mehrfach-Selektion ist durch Verschachtelung möglich (siehe Listing 2.22).

Für Mehrfach-Selektion, bei der als Bedingungen nur Vergleiche konstanter Ausdrücke auftreten, kann auch die `switch`-Anweisung als einfachere, übersichtlichere Variante verwendet werden. Die `switch`-Anweisung hat folgende Form:

```
switch ( Ausdruck ) {
    case konstanterAusdruck1:
        Anweisungen1
        break;
    case konstanterAusdruck2:
        Anweisungen2
        break;
    :
    case konstanterAusdruckk:
        Anweisungenk
        break;
    default:
        defaultAnweisungen
}
```

Der Rumpf der `switch`-Anweisung wird auch `switch-Block` genannt. Er enthält Anweisungen mit vorangestellten `case`-Sprungmarken. Diese Sprungmarken enthalten konstante Ausdrücke, die dem Typ des `switch`-Ausdrucks zuweisbar sein müssen. Ist der Wert des `switch`-Ausdrucks gleich einem Wert eines konstanten Ausdrucks, wird die Ausführung des Programms bei der ersten Anweisung nach der entsprechenden Sprungmarke weitergeführt. Wird kein passender konstanter Ausdruck gefunden, wird die Ausführung des Programms bei der ersten Anweisung nach der `default`-Sprungmarke (sofern vorhanden) fortgesetzt. Die in der `switch`-Anweisung verwendeten Ausdrücke dürfen folgenden Typ haben: *char*, *byte*, *short*, *int*, *Character*, *Byte*, *Short*, *Integer*, alle Aufzählungstypen (siehe Abschnitt 3.2.5) und *String* (ab JDK7).

Listing 2.23: Wandelt die Note als `int` in Text um (Variante1).

```
switch (note) {
    case 1:
        text = "sehr gut";
        break;
    case 2:
        text = "gut";
        break;
    case 3:
        text = "befriedigend";
        break;
    case 4:
        text = "genügend";
        break;
    case 5:
        text = "nicht genügend";
        break;
    default:
        text = "FEHLER: keine gültige Note";
}
```

2.4 Funktionen

Betrachten wir noch einmal Listing 2.1 in Abschnitt 2.1.1. Stellen wir uns vor, in einem Programm soll nicht nur einmal, sondern an mehreren Stellen der GGT von unterschiedlichen Wertepaaren berechnet werden. Um das zu bewerkstelligen, könnte man die Zeilen 7 bis 17 einfach an die Stellen im Programm kopieren, wo ein GGT berechnet werden soll und dort

Listing 2.24: Wandelt die Note als `int` in Text um (Variante2).

```
String text = "";

switch (note) {
    case 1:
        text += "mit Auszeichnung ";
    case 2:
    case 3:
    case 4:
        text += "bestanden";
        break;
    case 5: text += "nicht bestanden";
        break;
    default:
        text += "FEHLER: keine gültige Note";
}
```

den Variablen `m` und `n` die entsprechenden neuen Startwerte zuweisen. Es ist leicht einzusehen, dass diese Vorgehensweise viele Nachteile hat: Das Programm wird unnötig lange, da die dieselbe Anweisungssequenz mehrfach vorkommt und falls wir den Algorithmus zur Berechnung des GGT verändern wollen, müssen wir diese Änderung überall dort durchführen, wo die entsprechende Anweisungssequenz vorkommt.

In praktisch allen Programmiersprachen gibt es daher Sprachmittel, die Anweisungssequenzen, die mehrfach an verschiedenen Stellen im Programm, oder auch in verschiedenen Programmen gebraucht werden, wiederverwendbar machen. Durch diese soll nicht nur Codewiederholung vermieden werden, sondern auch die Möglichkeit zur Strukturierung bzw. Modularisierung des Softwareentwurfs geschaffen werden.

Man nennt solche Programmteile allgemein *Unterprogramme* oder *Routinen*. Ein Unterprogramm soll eine in sich abgeschlossene und gut beschreibbare Teilaufgabe erledigen.

Wir möchten uns zunächst mit einer speziellen Form von Unterprogrammen beschäftigen, nämlich mit Unterprogrammen, die einen oder mehrere Eingangswerte auf einen Rückgabewert abbilden und sonst keine externen Daten benutzen. Das bedeutet, sie benutzen keine Variablen, die nicht im Anweisungsblock des Unterprogramms selbst definiert wurden. Solche Unterprogramme nennt man *reine Funktionen*. Eine reine Funktion hat also, abgesehen von dem von ihr gelieferten Rückgabewert, keinen *Seiteneffekt*.

Listing 2.25: Definiert eine Methode für die GGT Funktion.

```
1 public class Euklid2 {
2
3     public static void main(String[] args) {
4
5         int ergebnis = ggt(1027,395);
6         System.out.println(ergebnis);
7
8         ergebnis = ggt(9,ggt(24,27)); //GGT von 3 Zahlen
9         System.out.println(ergebnis);
10
11         ergebnis = ggt(ggt(9,24),27); //gleiches Ergebnis wie oben
12         System.out.println(ergebnis);
13
14         int n = 4;
15         int d = 8;
16
17         System.out.println("Bruchzahl: "+n+"/"+d);
18
19         ergebnis = ggt(n,d);
20         n /= ergebnis;
21         d /= ergebnis;
22
23         System.out.println("gekürzt: "+n+"/"+d);
24
25     }
26
27     public static int ggt(int m, int n) {
28
29         while (m != n) {
30             if (m > n) {
31                 m = m - n;
32             } else {
33                 n = n - m;
34             }
35         }
36         return m;
37
38     }
39 }
```

Wir können beispielsweise den GGT Algorithmus wie folgt als Funktion definieren:

Das Programm berechnet, so wie das Programm in Listing 2.1, den

GGT. Der Unterschied ist hier, dass die Anweisungen des eigentlichen GGT Algorithmus nun in einer eigenen Methodendefinition in den Zeilen 27 bis 37 stehen. *Methoden* sind in Java Unterprogramme, die bestimmte Aufgaben lösen. Die Methode `ggt` wird dann in den Zeilen 5, 8 und 11 aufgerufen.

2.4.1 Methoden: Aufbau der Methodendefinition anhand eines Beispiels

Eine *Methode* ist ein Anweisungsblock, der unter einem Namen aufgerufen werden kann. In Java werden Funktionen als spezielle Methoden (ohne Seiteneffekte) definiert. Die Definition der Methode `ggt` befindet sich in den Zeilen 27 - 37 im Listing 2.25.

Eine *Methodendefinition* besteht aus dem *Methodenkopf*, der die *Methodendeklaration*⁶ enthält, und dem *Methodenrumpf*, dem Block mit Anweisungen. In Zeile 27 befindet sich der *Methodenkopf* und die öffnende geschwungene Klammer des Blocks. Die ersten beiden Schlüsselwörter in Zeile 27 sind *Modifier*, die die Sichtbarkeit (siehe Abschnitt 3.2.2) bzw. die Bedeutung der Methode ändern. Wir werden in Kapitel 3 auf die Bedeutung dieser beiden Modifier genauer eingehen. Hier nur soviel: Der Modifier `public` bedeutet, dass die Methode öffentlich ist (d.h. von allen Klassen aus aufgerufen werden kann), und `static` bedeutet, dass es sich um eine statische Methode handelt, d.h. eine Methode die unabhängig von Instanzen der Klasse `Euklid2` benutzt werden kann, weil Sie keine Instanz dieser Klasse benötigt, um ausgeführt werden zu können. Wir werden vorerst nur öffentliche, statische Methoden definieren, ohne näher auf die Bedeutung dieser Modifier einzugehen.

Das dritte Wort im Methodenkopf ist der Name des Datentyps der Rückgabe, es gibt also Auskunft darüber, dass die Methode einen `int`-Wert als Ergebnis zurückgeliefert. Danach folgt der Name der Methode, der - wie ein Variablenname - selbst gewählt werden kann. In den darauffolgenden runden Klammern stehen eine Liste von Deklarationen *formaler*

Parameter (siehe Abschnitt 2.4.2), die durch Beistriche getrennt werden.

Nach dem Methodenkopf folgt ein Anweisungsblock, der *Methodenrumpf* (*Body*) genannt wird. Der Methodenrumpf ist der zweite Teil der Definition und enthält die eigentlichen Anweisungen der Methode. In diesem Fall wird hier der Funktionswert berechnet. Die letzte Anweisung, die im Methodenrumpf zur Ausführung kommt, ist eine `return`-Anweisung, die den Ablauf der Methode beendet und den rechts davon stehenden Ausdruck als Ergebnis zurückliefert.

2.4.2 Parameter

Formale Parameter sind spezielle lokale Variablen einer Methode. Obwohl sie innerhalb der runden Klammern des Methodenkopfs deklariert werden, erstreckt sich ihr Gültigkeitsbereich über den Methodenrumpf. Es gibt einen wichtigen Unterschied zu gewöhnlichen lokalen Variablen: Parameter nehmen beim Aufruf der Methode Werte entgegen, die der Aufrufer der Methode zur Verfügung stellen muss. Über Parameter kann der Aufrufer der Methode also alle Informationen übermitteln, die die Methode benötigt, damit sie ihre Arbeit tun kann.

Die Deklaration hat dieselbe Syntax wie gewöhnliche Deklarationen lokaler Variablen, also Datentyp gefolgt von Bezeichner (ohne abschließenden Strichpunkt). Mehrere formale Parameter können zwischen den runden Klammern des Methodenkopfs stehen, wobei die Deklarationen durch Beistriche getrennt werden müssen.

Wenn die Methode aufgerufen wird, wird die Parameterliste mit der Argumentliste des Aufrufs - den *aktuellen Parametern* - abgeglichen. Die Werte der aktuellen Parametern werden implizit den formalen Parametern zugewiesen (siehe Abschnitt 2.4.5).

Anzahl und Typ der aktuellen Parameter müssen mit den formalen Parametern zusammenpassen, d.h. der Typ der aktuellen Parameter muss dem Typ der formalen Parameter zuweisbar sein.

Da ein formaler Parameter eine lokale Variable der Methode ist, darf im Methodenrumpf keine lokale Variable mit demselben Namen deklariert werden. Während ein formaler Parameter eine Variable ist, ist der aktuelle Parameter, der beim Aufruf angegeben wird, ganz allgemein ein Ausdruck, der einen Wert für den formalen Parameter liefert.

Es gibt auch Methoden, ohne formalen Parameter. Ein Beispiel ist die Methode in Listing 2.26, oder die Methode `public static double random()`, die in der Klasse `java.lang.Math` vorgegeben ist, und

⁶Man spricht von *Definition*, wenn die gesamte Information über die Methode (inklusive Anweisungen des Blocks) bekannt gegeben wird, im Unterschied zur *Deklaration*, bei der nur die Aspekte (Name, Rückgabotyp, formale Parameter, Modifier) bekannt gemacht werden, die zum Zeitpunkt der Übersetzung des Programms benötigt werden, damit es zulässig ist, diese in anderen Teilen des Programms zu verwenden (auch wenn der Anweisungsblock erst zur Laufzeit bekannt ist). Eine Methodendeklaration besteht also nur aus dem Methodenkopf. Analog wird bei der Variablendeklaration nur Name, Typ und Modifier aber kein Wert angegeben.

einen Zufallswert vom Typ `double` zurückliefert. Wir werden Methoden ohne Parameter vorerst nicht behandeln, aber in den späteren Kapiteln häufig definieren.

2.4.3 Die `return`-Anweisung

Mit Hilfe der `return`-Anweisung im Methodenrumpf kann ein Wert an den Aufrufer der Methode zurückgeliefert werden. Der Kontrollfluss wird an der Stelle im aufrufenden Programm fortgesetzt, an der der Aufruf stattgefunden hat.

Es gibt auch Methoden ohne Rückgabewert. Diese Methoden manipulieren im Regelfall Objekte, die außerhalb des Methodenrumpfs deklariert und verwendet werden. Sie sind also keine reinen Funktionen sondern wirken auf Objekte anderer Programmteile. Auf Methoden mit solchen Seiteneffekten werden wir in Abschnitt 2.6.3 noch genauer eingehen. Hier sei nur erwähnt, dass Methoden ohne Rückgabe den Rückgabebetyp `void` haben. Im Methodenrumpf kann eine "leere" `return`-Anweisung, ohne nachfolgendem Ausdruck, benutzt werden, um den Methodenablauf zu beenden. Wenn der Rückgabebetyp `void` ist, muss die Methode aber keine `return`-Anweisung haben.

Listing 2.26 zeigt eine Methode, die weder formale Parameter noch eine Rückgabe hat. Die Methode repräsentiert daher auch keine mathematische Funktion. Weiters gibt es einen Seiteneffekt (Nebeneffekt, Nebenwirkung), nämlich die Ausgabe eines Musters auf dem Bildschirm.

Listing 2.26: Eine Methode die ein Rautenmuster ausgibt.

```
public static void printDiamond () {
    System.out.println("  *  ");
    System.out.println(" *** ");
    System.out.println("*****");
    System.out.println(" *** ");
    System.out.println("  *  ");
}
```

2.4.4 Gleichnamige Methoden

Mehrere Methodendefinitionen können denselben Methodennamen benutzen, solange sie sich in ihren Parametern unterscheiden. Man spricht auch von *überladenen* Methoden. Wenn eine Methode aufgerufen wird, wird die Methode mit passendem Namen *und* Parameterliste aufgerufen, wobei hier

Typ, Anzahl und Reihenfolge der formalen Parameter eine Rolle spielen (nicht deren Namen). Name und Parametertypenliste machen die *Signatur* einer Methode aus, die mit dem Methodenaufruf zusammenpassen muss.

Wir können daher unser Programm in Listing 2.25 problemlos um eine Methode erweitern, die den GGT von drei Zahlen berechnen kann. Dazu müssen wir einfach folgende Zeilen anstelle der Leerzeile 38 in das Listing 2.25 einfügen. Hier wird eine Methode mit 3 formalen Parametern definiert, die wiederum auf die bereits definierte Methode mit 2 formalen Parametern zurückgreift, und durch verschachtelte Aufrufe den GGT von 3 Werten berechnet und zurückliefert.

Listing 2.27: Eine Methode für den GGT von 3 Zahlen.

```
public static int ggt(int m, int n, int o) {

    return ggt(m, ggt(n, o));

}
```

Das Ergebnis dieser Modifikation ist in Listing 2.28 dargestellt. Hier stehen uns nun 2 gleichnamige Methoden zur Verfügung, die sich jedoch in ihrer Signatur unterscheiden. Die Signatur der Methode in Zeile 24 lautet

ggt(int, int),

und die Signatur der Methode in Zeile 36 lautet

ggt(int, int, int).

Wir haben jetzt die Möglichkeit, den GGT von 2 Zahlen oder auch den GGT von 3 Zahlen berechnen zu lassen. Wir können also zum Beispiel folgende Aufrufe machen:

```
ergebnis = ggt(9,24,27); //Aufruf Zeile 24
ergebnis = ggt(24,27); //Aufruf Zeile 36
```

Obwohl der Rückgabebetyp zur Signatur gehört, ist es nicht möglich, mehrere Methoden mit gleichem Namen und gleicher Parameterliste, jedoch verschiedenem Rückgabebetyp zu definieren.

2.4.5 Beispiel einer Aufrufsequenz

In Zeile 5 bis 16 wird die Methode `ggt` mit jeweils anderen aktuellen Parametern aufgerufen. Der Kontrollfluss ist in Listing 2.28 dargestellt. Entlang der horizontalen Achse lässt sich der Zeitpunkt einer Anweisung bestimmen. Vertikal wird die Position der momentan auszuführenden Anweisung im Programmtext dargestellt. Der Kontrollfluss wird durch schwarze Strecken dargestellt.

Wenn eine Methode eine weitere Methode aufruft, wird die Ausführung der aufrufenden Methode unterbrochen und ein Befehlszeiger zwischengespeichert, damit an der richtigen Stelle der aufrufenden Methode weitergearbeitet werden kann, sobald die aufgerufene Methode fertig ist. Jede Methodenausführung wird in Listing 2.28 durch eine eigene Schattierung dargestellt.

Wird das Programm ausgeführt, beginnt dessen Ablauf mit der ersten Anweisung im Block der Methode `main`.

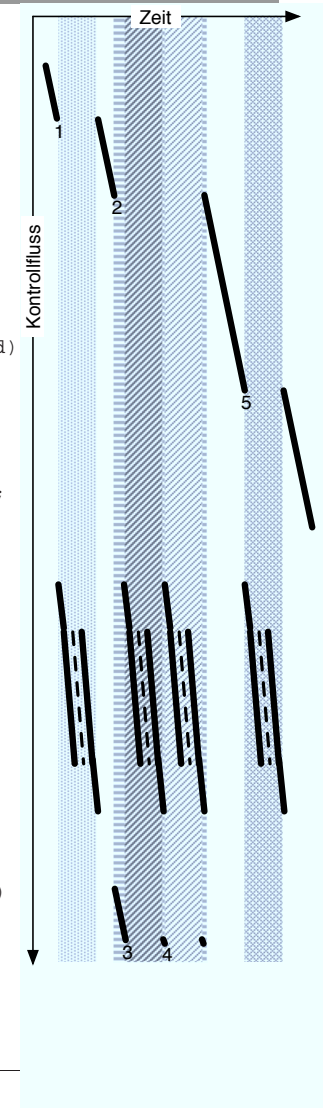
1. Zeile 5: Aufruf mit den aktuellen Parametern 1027 und 395. Damit wird die Methode `main` unterbrochen und der Kontrollfluss springt von Position 1 zur Methode mit der Signatur `ggt(int,int)`. Mit der Anweisung in Zeile 32 kehrt der Kontrollfluss zur Zeile 5 zurück und der Rückgabewert der Funktion wird an die Variable `ergebnis` zugewiesen.
2. Zeile 8: Hier wird die Methode `ggt(int,int,int)` aufgerufen (Position 2). Die Ausführung wird also in Zeile 36 fortgesetzt.
3. Zeile 38: Damit der `return`-Ausdruck berechnet werden kann muss die Ausführung der Methode hier unterbrochen werden. Zunächst wird die innere Funktion `ggt(n, o)` ausgewertet (Sprung bei Position 3) und erst wenn dieser Funktionswert geliefert wurde, kann das Gesamtergebnis durch Auswertung der äußeren Funktion `ggt(m, ggt(n, o))` mit einem erneuten Aufruf von `ggt(int, int)` berechnet werden (Position 4). Das Ergebnis wird dann von der `return`-Anweisung in Zeile 38 zurückgeliefert.
4. Zeile 16: Hier wird die Methode das letzte Mal mit (den Werten von) `n` und `d` aufgerufen (Position 5).

Listing 2.28: Beispiel für Aufrufsequenz.

```

1 public class Euklid2 {
2
3     public static void main(String[] args) {
4
5         int ergebnis = ggt(1027,395);
6         System.out.println(ergebnis);
7
8         ergebnis = ggt(9,24,27);
9         System.out.println(ergebnis);
10
11         int n = 4;
12         int d = 8;
13
14         System.out.println("Bruchzahl: "+n+"/"+d)
15
16         ergebnis = ggt(n,d);
17         n /= ergebnis;
18         d /= ergebnis;
19
20         System.out.println("gekürzt: "+n+"/"+d);
21
22     }
23
24     public static int ggt(int m, int n) {
25
26         while (m != n) {
27             if (m > n) {
28                 m = m - n;
29             } else {
30                 n = n - m;
31             }
32         }
33         return m;
34     }
35
36     public static int ggt(int m, int n, int o)
37
38         return ggt(m, ggt(n, o));
39
40     }
41
42 }

```



Wird eine Methode aufgerufen, wird für deren Ausführung ein eigener Speicherbereich benutzt um dort z.B. lokale Variablen der Methode anzulegen. Dies gilt auch dann, wenn es mehrere parallel ablaufende Ausführungen derselben Methode gibt. Wir betrachten hier nur sequentielle Abläufe, d.h. zu jedem Zeitpunkt wird nur ein Anweisungsblock ausgeführt. Grundsätzlich kann aber eine reine Funktion, die nur lokale Variablen benutzt, ohne Wechselwirkung parallel ausgeführt werden, ohne dass sich die Aufrufe beeinflussen.

In Abbildung 2.29 werden die ausgeführten Methoden mit dem Gültigkeitsbereich ihrer lokalen Variablen schematisch dargestellt. Beachten Sie, dass beispielsweise die zum Zeitpunkt 2 aufgerufene Methode zum Zeitpunkt 3 nur unterbrochen wird (vergleiche Listing 2.28), d.h. alle ihre Variablen bleiben erhalten und werden bei der Rückkehr vom Aufruf 3 weiter benutzt. Aufruf 3 benutzt einen eigenen Speicherbereich in dem wieder eigene lokale Variablen erzeugt werden. Die sich gerade in Ausführung befindende Methode kann nicht auf lokale Variablen der gerade unterbrochenen (wartenden) Methoden zugreifen.

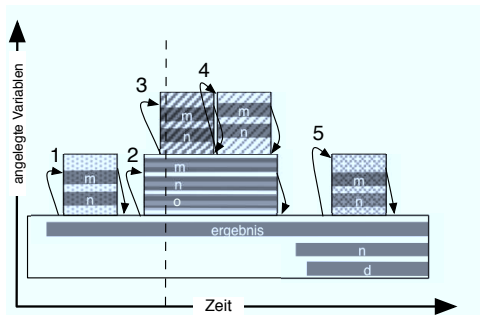


Abbildung 2.29: Zeigt, welche Variablen zu welchem Zeitpunkt beim Ablauf von Listing 2.28 angelegt sind. Jede Methodenausführung hat ihren eigenen Speicherbereich, wobei in dieser schematischen Darstellung, die Bereiche aufrufender Methoden oberhalb jener der aufrufenden Methoden liegen (siehe Aufrufpfeile). Obwohl zum Beispiel unmittelbar nach dem Aufruf 3 (senkrechte gestrichelte Linie) insgesamt sechs Variablen angelegt sind, sind nur die zwei Variablen der "obersten" Methode in Ausführung gültig. Die aufrufende Methode wurde vorübergehend unterbrochen, bis der Aufruf das Ergebnis liefert.

2.4.6 Rekursive Methoden

Eine Methode kann sich auch selbst aufrufen. Man spricht in diesem Fall von *Rekursion* bzw. von einem *rekursiven Aufruf*. Wie bei jedem Methodenaufwurf wird die Ausführung der aufrufenden Methode unterbrochen, bis die Ausführung der aufgerufenen Methode beendet wird. Die neue Methodenausführung hat, wie bereits im letzten Abschnitt gezeigt, einen eigenen Speicherbereich für ihre lokalen Variablen. Falls die Methode ausschließlich lokale Variable benutzt (dazu zählen auch ihre formalen Parameter), also keine Seiteneffekte hat, beeinflussen sich die Ausführungen nicht. Die aufrufende Methode wartet, bis die aufgerufene Methode fertig ist. Das Ergebnis des rekursiven Aufrufs kann dann weiter verwendet werden.

Listing 2.30 zeigt eine Variante des Programms aus Listing 2.28. Die Funktionalität der Methode $ggt(int, int)$ ist dieselbe geblieben. Geändert hat sich nur die Implementierung, die jetzt einen rekursiven Algorithmus darstellt.

Auf den ersten Blick wirkt es seltsam, dass bei einer Rekursion die aufrufende und aufgerufene Methode genau dieselbe ist. Es hat zur Folge, dass dieselben Programmzeilen wiederholt ausgeführt werden. In dieser Hinsicht bewirkt die Rekursion dasselbe wie die *Iteration* (Schleife): Es wird ein Anweisungsblock wiederholt ausgeführt.

Die formalen Parameter sowie alle anderen lokalen Variablen werden für jede rekursive Ausführung gesondert angelegt. Der Programmierer hat dafür zu sorgen, dass die aktuellen Parameterwerte bei jedem neuen rekursiven Aufruf andere sind. Wäre das nicht so, rief sich eine Methode mit immer wieder denselben aktuellen Parametern auf und es käme zu einer *Endlosrekursion* (das Pendant zur *Endlosschleife*).

Damit es nicht zu einer Endlosrekursion kommt, muss es einen *Rekursionsanfang* geben, also einen Fall bei dem ein Methodenaufwurf zu keinem weiteren Aufruf führt, sondern direkt ein Ergebnis liefert. In Listing 2.30 sehen wir einen solchen Fall, der dann eintritt, wenn m und n denselben Wert haben. In diesem Fall ist der GGT genau dieser Wert. Es gilt also $ggt(n, n) \rightarrow n$. Wenn wir mit Listing 2.28 vergleichen sehen wir, dass auch hier die Schleife abgebrochen wird, wenn dieser Fall eintritt. In dem Fall wird einfach der Wert zurückgeliefert.

In jedem anderen Fall gilt die Gleichung $ggt(m, n) = ggt(n - m, n)$, wenn $n - m > 0$ oder sonst die Gleichung $ggt(m, n) = ggt(m, n - m)$. Diese mathematischen Gleichungen erkennt man sowohl im iterativen Algorithmus aus Listing 2.28, wo im nächsten Schleifendurchlauf mit den

Werten $m - n$, n bzw. m , $n - m$ weitergearbeitet wird, als auch im rekursiven Algorithmus aus Listing 2.30, wo der rekursive Aufruf mit diesen Werten erfolgt.

Ein rekursiver Aufruf ist also ein Methodenaufruf derselben Methode mit neuen aktuellen Parametern. Man erkennt auch intuitiv anhand des obigen Beispiels, dass Rekursion und Iteration gleichmächtig sind. Wir können mit einer Schleife genau das berechnen, was wir rekursiv berechnen können. Tatsächlich ist die Schleife als Kontrollstruktur für die Turing-Vollständigkeit einer Programmiersprache, in der es Funktionen gibt, nicht notwendig (vergleiche *funktionale Programmierung*).

Listing 2.30: GGT mit rekursivem Algorithmus.

```

1 public class Euklid2 {
2
3     public static void main(String[] args) {
4
5         int ergebnis = ggt(1027,395);
6         System.out.println(ergebnis);
7
8         ergebnis = ggt(9,24,27);
9         System.out.println(ergebnis);
10
11         int n = 4;
12         int d = 8;
13
14         System.out.println("Bruchzahl: "+n+"/"+d);
15
16         ergebnis = ggt(n,d);
17         n /= ergebnis;
18         d /= ergebnis;
19
20         System.out.println("gekürzt: "+n+"/"+d);
21     }
22 }
23
24 public static int ggt(int m, int n) {
25
26     if (m == n) return m;
27
28     if (m > n) {
29         return ggt(m - n, n);
30     } else {
31         return ggt(m, n - m);
32     }
33 }
34

```

```

35     public static int ggt(int m, int n, int o) {
36
37         return ggt(m, ggt(n, o));
38
39     }
40
41 }

```

2.4.7 Zusicherungen

Der obige GGT Algorithmus setzt voraus, dass für Werte der aktuellen Parameter gilt: $m > 0$ und $n > 0$. Damit die Methode *ggt(int, int)* das richtige Ergebnis liefert, müssen diese Bedingungen beim Aufruf der Methode erfüllt sein. Andernfalls ist das Verhalten der Methode nicht definiert. Wenn beispielsweise einer der beiden aktuellen Parameter den Wert 0 hat, gerät das Programm in eine Endlosschleife bzw. Endlosrekursion.

Die Bedingung $m > 0$ und $n > 0$ ist eine *Zusicherung*. Eine Zusicherung kann oft als Boolescher Ausdruck, der niemals falsch werden darf, geschrieben werden. Wie bereits in Abschnitt 1.5.4 erwähnt, gibt es mehrere Möglichkeiten Zusicherungen darzustellen (z.B. durch einen Kommentar oder die *assert*-Anweisung).

Bezogen auf ein Unterprogramm oder Programmteil (z.B. Anweisungssequenz, Methode) kann man drei Arten von Zusicherungen unterscheiden:

- Vorbedingungen
- Nachbedingungen
- Invarianten

Wir betrachten nun die Zustände von Variablen und wie diese durch die Sequenz von Anweisung verändert werden. Die zu einem bestimmten Zeitpunkt aktuellen Werte aller im Programm verwendeten Variablen nennt man Zustand des Programms. Ein Unterprogramm *S* verändert den Zustand des Programms, führt das Programm also von dem Zustand *P* vor der Ausführung des Unterprogramms in einen anderen Zustand *Q* über.

Wir können nun Bedingungen für *P* festlegen, die einen korrekten Ablauf von *S* ermöglichen. Sind diese Bedingungen erfüllt, dann ist der Programmcode von *S* korrekt, wenn der Zustand *Q* der Spezifikation von *S* entspricht.

Man kann dies auf formal als *Hoare-Tripel* ausdrücken:

$$\{P\}S\{Q\}$$

Eine *Vorbedingung* P gibt an, unter welcher Bedingung das Verhalten von S definiert ist, d.h. S korrekt funktioniert. Beispielsweise ist $m > 0$ und $n > 0$ eine Vorbedingung für die Methode $ggt(int, int)$. Für die Einhaltung der Vorbedingung ist der Aufrufer (Programmteil, der die Methode aufruft) zuständig, da die Methode selbst keine Möglichkeit hat, die Einhaltung der Vorbedingung zu gewährleisten. Der Aufrufer muss also dafür sorgen, dass der Aufruf nicht in einem Zustand durchgeführt wird, der die Vorbedingung des aufgerufenen Unterprogramms nicht erfüllt.

Eine *Nachbedingung* Q stellt eine Zusicherung für den korrekten Zustand nach dem Aufruf des Unterprogramms dar. Die Nachbedingung der Methode $ggt(int, int)$ ist, dass deren Rückgabe dem GGT der beim Aufruf übergebenen aktuellen Parameter entspricht. Ist die Vorbedingung verletzt, so ist das aufgerufene Unterprogramm nicht an die Nachbedingung gebunden. Eine Verletzung der Vorbedingung kann, wie in unserem Beispiel (Endlosschleife bzw. Endlosrekursion) deutlich wird, zum Programmabsturz führen.

Die Beschreibung von Zusicherungen hilft, dass oft nur schwer fassbare dynamische Verhalten eines Anweisungsblocks auf eine übersichtliche statische Ebene zu bringen. Weiters ist die Implementierung einer Methode für den Aufrufer in vielen Fällen unbekannt, daher sind Zusicherungen oft die einzige Möglichkeit der Beschreibung dessen, was die Methode tut.

Vorbedingung und Nachbedingung stellen den Vertrag einer Methode dar, den der Aufrufer und der Aufgerufene (die Methode) einhalten müssen. Die Vorbedingung muss immer vom Aufrufer geprüft werden und nie vom Aufgerufenen. Umgekehrt muss für die Einhaltung der Nachbedingung vom Aufgerufenen gesorgt werden. Die Vorbedingung stellt eine Pflicht für den Aufrufer und einen Nutzen für den Aufgerufenen dar. Die Nachbedingung ist eine Pflicht für den Aufgerufenen und ein Nutzen für den Aufrufer.

Eine *Invariante* ist eine Bedingung, die über die Ausführung bestimmter Programmteile hinweg gilt. Die Bedingung ist vor und nach der Ausführung eines Programmteils wahr und somit unveränderlich (daher die Bezeichnung). Ein Beispiel für eine Invariante ist die Bedingung $m > 0$ und $n > 0$, die während der gesamten Ausführung der iterativen Methode $ggt(int, int)$ gilt. Würde eine Anweisung im Schleifenrumpf dazu führen, dass diese Bedingung verletzt wird, käme es zu einer Endlosschleife. Eine

Invariante kann sich nicht nur auf eine einzelne Methode beziehen, sondern auf den Zustand des Programms (bzw. auf den Zustand eines Objekts) über die Ausführungszeit einer Methode hinweg. Dies ist vor allem bei Methoden, die nicht reine Funktionen darstellen, von Bedeutung.

Ein (Unter-)Programm S kann bezüglich seiner Vor- und Nachbedingungen auf Korrektheit überprüft werden. Ist die Vorbedingung P erfüllt, dann ist S dann korrekt, wenn nach der Ausführung von S , die Nachbedingung Q erfüllt ist.

Wir können zum Beispiel für die iterative Variante von $ggt(int, int)$ einen formalen Beweis der Korrektheit durchführen. Um die Korrektheit zu beweisen sind zwei Dinge zu zeigen:

- (a) Für jede Eingabe zweier natürlicher Zahlen $m > 0$ und $n > 0$ terminiert die Methode, d.h. sie stoppt nach der Ausführung endlich vieler Schritte.
- (b) Terminiert die Methode, so ist der zurückgelieferte Wert der GGT der beim Aufruf angegebenen aktuellen Parameter m und n ; d.h. das Ergebnis ist korrekt.

Beweis von (a): Bei jeder Ausführung des Schleifenrumpfs wird entweder m oder n um mindestens 1 verkleinert. Beide Zahlen bleiben aber positiv, da immer von der größeren die kleinere Zahl abgezogen wird. Der Fall, bei dem eine Zahl 0 wird kann nur eintreten, wenn $m == n$ was zum Abbruch der Schleife führt. Folglich kann der Schleifenrumpf nur endlich oft durchlaufen werden. Weil der Schleifenrumpf stets ausgeführt wird, wenn die Bedingung $m == n$ verletzt ist, muß nach endlich vielen Iterationen $m == n$ gelten und die Methode terminieren.

Beweis von (b): Wir setzen für diesen Beweis einen Satz voraus, ohne diesen hier zu beweisen: Er besagt, dass wenn gilt $m > n > 0$ dann ist $ggt(m, n) = ggt(m - n, n)$. Außerdem gilt immer $ggt(m, n) = ggt(n, m)$ (Kommutativität). Nach diesem Satz wissen wir, dass bei Durchlauf des Schleifenrumpfs der GGT von m und n sich nicht ändert (obwohl sich m oder n ändert). Nach dem obigen Beweis von (a) stoppt der Algorithmus nach endlich vielen Schritten, und es gilt dann $m == n$. Folglich ist m der ggT von m und n und nach obigen Überlegungen auch von den ursprünglichen aktuellen Parametern des Methodenaufrufs. Ähnlich lässt sich zeigen, dass auch die rekursive Variante von $ggt(int, int)$ korrekt ist.

Im Fall der Methode $ggt(int, int)$ konnte also deren Korrektheit im Bezug auf die Vorbedingung $m > 0$ und $n > 0$ und Nachbedingung $ggt(m,$

n) \rightarrow GGT gezeigt werden. Wie in Abschnitt 1.5.3 im Zusammenhang mit dem *Halteproblem* bereits erwähnt wurde, ist jedoch eine Eigenschaft von Turing-vollständigen Programmiersprachen, dass wir formal für manche Programme auf manchen Eingaben nicht feststellen können, ob das Programm jemals terminieren bzw. ein Ergebnis liefern wird.

Man unterscheidet daher *partielle* und *totale Korrektheit*. Ein Programm ist partiell korrekt, wenn bei erfüllter Vorbedingung jedes Ergebnis die Nachbedingung erfüllt, wobei nicht gewährleistet wird, dass das Programm überhaupt ein Ergebnis liefert. Das Programm ist total korrekt, wenn es partiell korrekt ist und zusätzlich immer terminiert, wenn die Vorbedingung erfüllt ist. Im Fall von $ggt(int, int)$ konnte also auch gezeigt werden, dass die Methode bei erfüllter Vorbedingung terminiert, daher ist $ggt(int, int)$ total korrekt bezüglich der angegebenen Vor- und Nachbedingungen.

2.5 Iteration, Arrays, Strings

Eine Iteration ermöglicht das mehrfache Ausführen von Anweisungen. Rekursion und Iteration sind gleichmächtige Sprachmittel. Beides steht für die wiederholte Ausführung eines Blocks, nämlich im Fall der Rekursion des Methodenrumpfs bzw. im Fall der Iteration des iterierten Anweisungsblocks (Schleifenrumpf). Die Iteration - auch Schleife genannt - ist eine Kontrollstruktur, die in vielen Fällen eine Vereinfachung gegenüber Rekursion darstellt: Rekursive Methodenaufrufe erzeugen einen Speicher- und Zeitaufwand, da jeder Aufruf zu einer neuen unabhängigen Ausführung der Methode mit eigenem Speicherbereich für Variablen führt. Wenn die Rekursionstiefe zu groß wird führt das zur Erschöpfung der zur Verfügung stehenden Ressourcen und das Programm bricht die Rekursion ab. In den meisten Fällen ist der Absturz des Programms die Folge.

Iterative Umsetzungen sind daher oft effizienter, weil die Iteration innerhalb derselben Methodenausführung stattfindet. Das bedeutet, es werden immer dieselben Variablen benutzt (siehe Abbildung 2.31).

2.5.1 while und do-while

Die `while`-Schleife hat folgende Syntax:

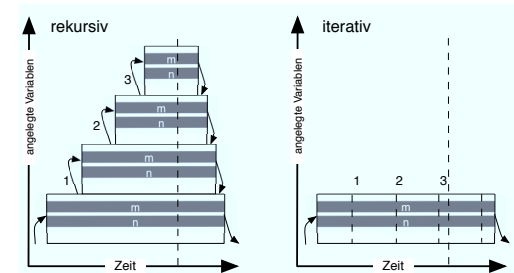


Abbildung 2.31: Zeigt die zu jedem Zeitpunkt der Methodenausführung von $ggt(int, int)$ angelegten lokalen Variablen. Links ist die rekursive Methode dargestellt und rechts die iterative. Jede Methodenausführung hat ihren eigenen Speicherbereich (auch wenn es sich um rekursive Ausführungen, d.h. Ausführungen derselben Methode, handelt). In dieser schematischen Darstellung liegen die Speicherbereiche aufgerufener Methoden oberhalb jener der aufrufenden Methoden (siehe Aufrufpfeile). Nach dem dritten rekursiven Aufruf (senkrechte gestrichelte Linie) gibt es 4 Methoden in Ausführung, wobei nur die oberste aktiv ist und die aufrufenden Methoden vorübergehend unterbrochen wurden, bis der Aufruf das Ergebnis liefert. Man erkennt, dass für jeden rekursiven Aufruf weitere Variablen angelegt werden. Die iterative Variante kommt dagegen mit einer einzigen Methodenausführung und daher weniger Ressourcen aus.

```
while ( boolescherAusdruck )
    Anweisung
```

Der Boolesche Ausdruck wird ausgewertet, und wenn er `true` liefert, wird die Anweisung ausgeführt. Sobald die Anweisung ausgeführt wurde, wird der Boolesche Ausdruck erneut ausgewertet, und wenn er noch immer `true` liefert, wird die Anweisung wieder ausgeführt. Das wird solange wiederholt bis der Boolesche Ausdruck zu `false` ausgewertet wird. Ist das der Fall, wird der Kontrollfluss nach der Anweisung fortgesetzt. Ein Beispiel für den Einsatz einer `while`-Schleife ist in Listing 2.28 in Zeile 26 zu sehen. Anstelle einer einzelnen Anweisung (in diesem Beispiel eine `if`-Anweisung) kann auch ein Block stehen. Die einzelne Anweisung bzw. der Block wird auch *Schleifenrumpf* genannt. Das Schlüsselwort `while` und der von runden Klammern begrenzte Boolesche Ausdruck bilden den *Schleifenkopf*. Um keine Endlosschleife zu erzeugen, müssen die Anweisun-

gen im Schleifenrumpf den Booleschen Ausdruck beeinflussen.

Eine `while`-Schleife führt die Anweisung mehrmals aus, jedoch kann es auch sein, dass sie gar nicht ausgeführt wird, wenn der Boolesche Ausdruck gleich bei der ersten Auswertung `false` ergibt.

Die `do-while`-Schleife ist eine Variante, bei der die Anweisung mindestens einmal ausgeführt wird. Die Syntax ist:

```
do
    Anweisung
while ( boolescherAusdruck )
```

Der Boolesche Ausdruck wird also erst ausgewertet, nachdem die Anweisung ausgeführt wurde. Solange der Ausdruck `true` liefert wird die Anweisung wiederholt ausgeführt.

2.5.2 Array-Typen

Ein Array ist ein Objekt, das aus mehreren Komponenten zusammengesetzt ist. Die Komponenten heißen *Elemente* und sind alle vom selben Datentyp (*Elementtyp*).

Eine Variable eines Array-Typs ist eine Referenzvariable, d.h. sie speichert eine Referenz auf ein Array-Objekt (siehe Abschnitt 2.1.3 über Referenztypen). Array-Typen gehören also zu den Referenztypen.

Ein Array kann erzeugt werden, wie im folgenden Beispiel gezeigt:

```
int[] ia;
ia = new int[4];
```

In der ersten Zeile wird eine Referenzvariable für ein Array-Objekt erzeugt, wobei hier durch die Klammern `[]` klar gemacht wird, dass es sich um eine Array-Variable handelt und der Elementtyp `int` ist. Die Anzahl der Elemente wird bei der Deklaration einer Array-Variable nicht angegeben.

In der zweiten Zeile geschieht Folgendes: Zunächst wird der Ausdruck `new int[4]` rechts vom Zuweisungssymbol ausgewertet. Mit Hilfe des Operators `new` wird ein neues Array-Objekt erstellt, das Werte vom Typ `int` aufnehmen kann. Die Anzahl der Elemente im Array-Objekt wird dabei mit 4 festgelegt. Beim Erstellen des Array-Objektes werden diese Elemente mit dem Default-Wert 0 initialisiert. Die allgemeine Form für den Ausdruck ist:

```
new Elementtyp[ int-Ausdruck ] ;
```

Der Elementtyp kann jeder Datentyp sein, also sowohl ein elementarer Typ als auch ein Referenztyp. Der Elementtyp kann auch selbst wieder ein Array-Typ sein, in welchem Fall ein mehrdimensionales Array entsteht (siehe Abschnitt 2.5.3). Der `new` Operator liefert eine Referenz auf das neu erzeugte Array-Objekt zurück. Diese wird im obigen Beispiel der Variable `ia` zugewiesen (siehe Abbildung 2.32). Die Elemente eines neu erzeugten `int`-Arrays werden automatisch mit dem Wert 0 initialisiert.

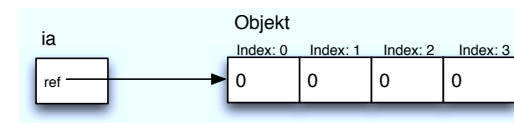


Abbildung 2.32: Ein neu erzeugtes `int`-Array. Die Referenz auf das Array-Objekt ist der Variable `ia` zugewiesen worden.

Die *Länge* (oder Größe) des Arrays entspricht der Anzahl der Elemente. Die Elemente des Arrays können über einen Index angesprochen werden, wobei der Index des ersten Elements 0 ist. Ist n die Länge des Arrays, werden die Elemente von 0 bis $n - 1$ durchgezählt.

Der Zugriff auf die Elemente des Arrays ist über die Array-Variable möglich. Beispielsweise erfolgt der Zugriff auf das dritte Element des oben erzeugten Arrays mit `ai[2]`. Wollen wir die Elemente ändern, geschieht das durch Zuweisungen:

```
ia[2] = 8; //drittes Element
ia[3] = 10; //viertes Element
ia[0] = 45; //erstes Element
```

Eine weitere Möglichkeit, ein Array-Objekt zu erzeugen, wird in folgendem Beispiel gezeigt:

```
int[] ia;
ia = new int[] {45, 0, 8, 10};
```

Auf der rechten Seite der Zuweisung wird das Array durch die Angabe einer *Initialisierungsliste* angelegt und gleichzeitig mit Werten initialisiert.

Die Länge des Arrays ergibt sich durch die Länge der *Initialisierungsliste*. Die *Initialisierungsliste* wird auch manchmal *Arrayliteral* genannt. Das Ergebnis ist das selbe wie das der obigen Erzeugung mit `new[4]` und der darauffolgenden Zuweisungen.

Bei der Erzeugung des Array-Objekts wird dessen Länge festgelegt. Sie kann später über das Datenfeld `length` abgefragt werden. So lässt sich beispielsweise die Länge des Arrays der Variable `ia` mit dem Ausdruck `ia.length` abfragen. Allgemein steht links vom Punkt-Operator `.` ein Ausdruck, der eine Referenz auf ein Array-Objekt liefert. Zum Beispiel liefert

```
new int[3].length
```

den Wert 3 (der Ausdruck ist allerdings sinnlos, da die Referenz auf das erzeugte Array-Objekt verloren geht).

Die Länge eines Arrays kann nach dessen Erzeugung nicht mehr verändert werden. Trotzdem kann der Fall eintreten, dass in einem Array mehr Platz benötigt wird. In diesem Fall muss ein neues größeres Array erzeugt werden und die Elemente müssen einzeln kopiert werden, wie in folgendem Beispiel gezeigt.

Listing 2.33: Erweiterung des Arrays `ia`.

```
int[] ia = {45, 0, 8, 10}; //ia hat Länge: 4

//...

//hier wird nun mehr Platz gebraucht und daher vergrößert:
//Schritt 1: temporäre Hilfsvariable mit neuem Array definieren
int[] iaLarger = new int[ia.length+1];

//Schritt 2: Elemente ins neue Array kopieren
int i = 0;
while(i < ia.length) {
    iaLarger[i] = ia[i];
    i++;
}

//ab nun wird das neue Array benutzt
//bisheriges Array wird nicht mehr gebraucht
//Schritt 3: Zuweisung
ia = iaLarger;

//ia hat nun Länge 5.
```

Die Auswirkungen der einzelnen Schritte in Listing 2.33 werden in Abbildung 2.34 veranschaulicht.

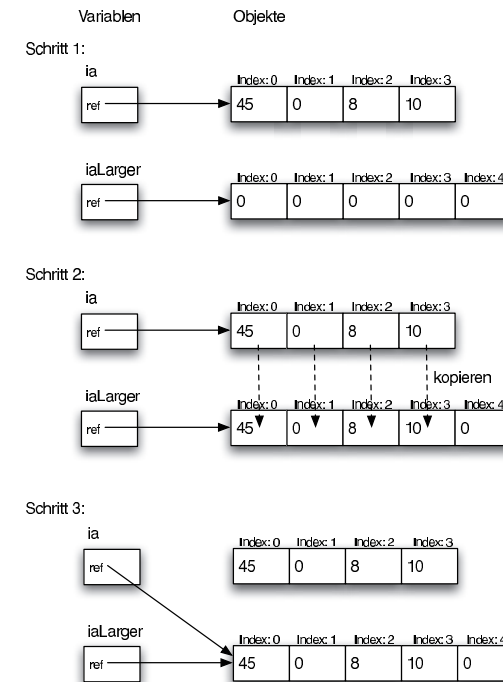


Abbildung 2.34: Die einzelnen Schritte der "Vergrößerung" eines Arrays aus Listing 2.33 graphisch veranschaulicht. Die letzte Anweisung ist eine Zuweisung, die die Referenz, die von `iaLarger` gespeichert wird, in `ia` kopiert. Nach dieser Zuweisung gibt es keine Referenz auf das ursprüngliche Array-Objekt mit Länge 4. Es wird daher verworfen, sodass dessen Speicherbereich wieder frei ist. Die Hilfsvariable `iaLarger` wird nach der Zuweisung auch nicht mehr benötigt.

2.5.3 Mehrdimensionale Arrays

Ein Array dessen Elementtyp wieder ein Array ist, wird als *mehrdimensionales Array* bezeichnet. Beispielsweise kann man eine Matrix von Zah-

lenwerten durch ein mehrdimensionales Array darstellen:

```
int[][] matrix = new int[3][4];
matrix[2][1] = 5;
```

Das hier erzeugte Array besteht aus 3 Elementen vom Typ `int[]`. Die Elemente sind also wieder Referenzvariablen. Das hier erzeugte Array wird in Abbildung 2.35 dargestellt.

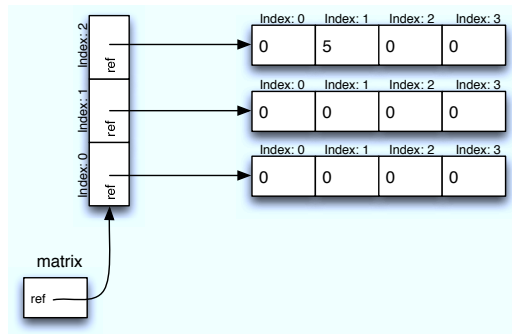


Abbildung 2.35: Ein mehrdimensionales Array.

Mehrdimensionale Arrays müssen nicht unbedingt eine rechteckige Form haben. Man kann auch die einzelnen Elemente des Arrays separat mit unterschiedlich langen Arrays initialisieren. In folgendem Beispiel wird zunächst die Länge entlang der zweiten Dimension offen gelassen (man spricht von *offenen Arrays*):

```
//zweite Länge bleibt offen
int[][] dreieck = new int[3][];

dreieck[0] = new int[3];
dreieck[1] = new int[2];
dreieck[3] = new int[1];
```

Dasselbe Array kann auch so erzeugt werden:

```
int[][] dreieck = new int[][] {{0,0,0},{0,0},{0}};
```

2.5.4 for

Die `for`-Schleife ist eine syntaktische Vereinfachung für den in der Praxis sehr häufig auftretenden Fall, dass eine Laufvariable benutzt wird. Die Syntax der `for`-Anweisung ist:

```
for (Initialisierungsklausel; BoolescherAusdruck; Aktualisierungs-Ausdrucksliste)
    Anweisung
```

Die `for`-Anweisung ist äquivalent zu:

```
{
    Initialisierungsklausel
    while ( boolescherAusdruck )
        Anweisung
        Aktualisierungs-Ausdrucksliste
}
```

Der Schritt 2 in Listing 2.33 lässt sich übersichtlicher mit einer `for`-Schleife umsetzen, wie in Listing 2.36 gezeigt.

Listing 2.36: Schritt 2 aus Listing 2.33 mit `for`-Schleife umgesetzt.

```
//Schritt 2: Elemente ins neue Array kopieren
for (int i = 0; i < ai.length; i++) {
    iaLarger[i] = ia[i];
}
```

In diesem Fall besteht sowohl die *Initialisierungsklausel* als auch die *Aktualisierungs-Ausdrucksliste* aus einem einzigen Ausdruck. Falls mehrere Initialisierungen bzw. Aktualisierungen durchgeführt werden sollen, müssen die entsprechenden Ausdrücke durch Kommata getrennt werden. In der Initialisierungsklausel darf allerdings nur eine Definition vorkommen, es ist daher nicht möglich Variablen mit unterschiedlichem Typ zu definieren:

```
for (int n = 0, k = 5; k >= 0; n++,k--) { ...
```

Eine erweiterte Variante der `for`-Anweisung steht seit JDK 5.0 zur Verfügung. Folgende Schleife gibt alle Elemente des Arrays auf dem Bildschirm aus.

Listing 2.37: Ausgabe eines Arrays mit einer `for-each`-Schleife.

```
int[] ai = {45,0,8,10};

for (int wert: ai) {
    System.out.println(wert);
}
```

Im Schleifenkopf wird zunächst eine Variable mit demselben Typ wie der Elementtyp des Arrays, das durchmustert werden soll, deklariert. Nach dem Doppelpunkt folgt ein Ausdruck, der eine Referenz auf das zu durchlaufende Array liefert. Bei jedem Schleifendurchlauf wird das nächste Element an die vor dem Doppelpunkt definierte Variable zugewiesen. Die Reihenfolge ist dabei immer aufsteigend. Es ist also nicht möglich das Array beispielsweise rückwärts zu durchlaufen oder Elemente auszulassen.

Die Schleife kann gelesen werden als "für alle Elemente von `ia`, das aus `int`-Werten besteht", daher wird manchmal auch die Bezeichnung `for-each`-Schleife benutzt.

2.5.5 Beispiel: Pascalsches Dreieck

Anhand des folgenden Beispiels sollen zwei Themen veranschaulicht werden: Der Einsatz von Schleifen als Vereinfachung gegenüber Rekursion und Arrays.

Das Pascalsche Dreieck ist eine geometrische Darstellung der Binomialkoeffizienten, die die rekursive Formel für deren Berechnung widerspiegelt. Die Formel zur Berechnung des Binomialkoeffizienten $\binom{n}{k}$ lautet:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, n > k > 0$$

und

$$\binom{0}{0} = \binom{1}{0} = \binom{n}{n} = 1.$$

Ordnet man die Binomialkoeffizienten als Einträge im Pascalschen Dreieck an, so lassen sich deren Werte leicht bestimmen: Der erste und letzte Eintrag einer Zeile ist immer 1. Alle anderen Einträge ergeben sich aus der Summe der zwei darüberstehenden Einträge. So können leicht weitere Zeilen ergänzt werden.

$$\begin{array}{ccccccc}
 & & & & \binom{0}{0} & & \\
 & & & \binom{1}{0} & \binom{1}{1} & & \\
 & & \binom{2}{0} & \binom{2}{1} & \binom{2}{2} & & \\
 & \binom{3}{0} & \binom{3}{1} & \binom{3}{2} & \binom{3}{3} & & \\
 \binom{4}{0} & \binom{4}{1} & \binom{4}{2} & \binom{4}{3} & \binom{4}{4} & &
 \end{array}
 \rightarrow
 \begin{array}{ccccccc}
 & & & & 1 & & \\
 & & & 1 & & 1 & \\
 & & 1 & & 2 & & 1 \\
 & 1 & & 3 & & 3 & & 1 \\
 1 & & 4 & & 6 & & 4 & & 1
 \end{array}$$

Die Variablen n und k können wie folgt als Zeilenindex bzw. Spaltenindex interpretiert werden.

$n \backslash k$	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

Wir wollen zunächst ein Java-Programm erstellen, das das Pascalsche Dreieck unter Verwendung einer rekursiv definierten Funktion ausgibt. Listing 2.38 zeigt den entsprechenden Programmcode.

Wird das Programm ausgeführt, wartet es zunächst in Zeile 7 auf die Eingabe einer Zahl, die die Anzahl der Zeilen des Pascalschen Dreiecks angibt. Diese Zahl wird dann der Variable `zeilen` zugewiesen. In den Zeilen 9 und 10 werden die Variablen `n` und `k` mit dem Wert 0 initialisiert. Der nun folgende Block besteht aus 2 verschachtelten Schleifen. In der äußeren Schleife wird `n` immer um eins erhöht, damit eine Zeile nach der anderen erzeugt werden kann. In jeder Zeile muss für die einzelnen Einträge der Spaltenindex `k` von 0 bis Zeilenende `n` inkrementiert werden. Zunächst wird `k = 0` gesetzt. Bei jedem Durchlauf der inneren Schleife wird ein weiterer Eintrag der Zeile ergänzt (direkt ausgegeben). Um den Eintrag an der Stelle `n, k` im Dreieck zu berechnen, wird die Funktion `binom` benutzt. Damit die Einträge unterscheidbar bleiben wird ein Leerzeichen an jeden Eintrag angehängt (Verkettung mit `+`). Der gesamte obige Vorgang wird wiederholt bis das Zeilenende erreicht wird.

Wenn `k > n` ist, wird die Schleife beendet. Die Zeile ist damit fertig. Im äußeren Schleifenrumpf wird ein Zeilenvorschub ausgegeben, damit die

Listing 2.38: Pascalsches Dreieck mit rekursivem Algorithmus.

```

1 import java.util.Scanner;
2
3 public class Pascal {
4
5     public static void main(String[] args) {
6
7         Scanner sc = new Scanner(System.in);
8
9         int zeilen = sc.nextInt();
10
11         for (int n = 0; n < zeilen; n++) {
12
13             for (int k = 0; k <= n; k++) {
14                 System.out.print(binom(n,k)+" ");
15             }
16
17             System.out.println();
18         }
19
20     }
21
22     public static int binom(int n, int k) {
23
24         if (k==0 || k==n) {
25             return 1;
26         } else {
27             return binom(n-1,k-1) + binom(n-1,k);
28         }
29     }
30
31 }
32
33 }

```

nächste Ausgabe in der nächsten Zeile erfolgt. Der Zeilenindex wird danach um 1 erhöht und es wird geprüft, ob eine weitere Zeile erzeugt werden soll.

Die Berechnung der Einträge erfolgt in Listing 2.38 rekursiv. Vergleichen Sie die Implementierung der Methode `binom` mit der oben angegebenen Formel für den Binomialkoeffizienten. Der Algorithmus ist eine direkte Umsetzung der rekursiven Formel.

Die rekursive Methode hat einige Nachteile: Wie bereits in der Einleitung zu Abschnitt 2.5 erwähnt verbrauchen rekursive Aufrufe Ressourcen.

Für den mittleren Binomialkoeffizienten in Zeile n beträgt die Anzahl der rekursiven Aufrufe $\frac{n^2}{2}$. Sie wächst also als quadratische Funktion des Zeilenindex. Zusätzlich berechnen wir für eine n -te Zeile n Binomialkoeffizienten. Es kommt also zu der Anzahl der Aufrufe noch einmal ein Faktor der den Wert von `zeilen` enthält dazu. Das Problem entsteht, weil der Algorithmus die Ergebnisse der rekursiven Aufrufe nicht zwischenspeichert und stattdessen dieselben Aufrufe mehrfach durchführt. Wenn beispielsweise die fünfte Zeile erzeugt wird, wird die dieselbe Berechnung $\text{binom}(2,1)$ vier mal durchgeführt, und zwar weil der Wert für die Berechnung des zweiten, dritten und vierten Elements der Zeile 5 gebraucht wird (siehe Abbildung 2.39).

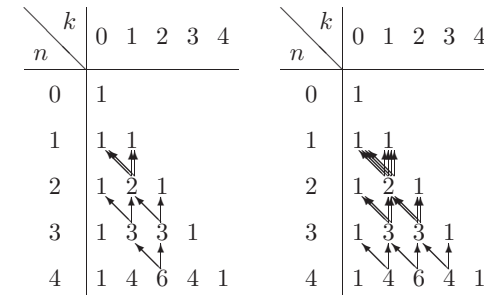


Abbildung 2.39: Links: Beim Aufruf von $\text{binom}(4,2)$ kommt es zu weiteren rekursiven Aufrufen, wobei $\text{binom}(2,1)$ und folglich $\text{binom}(1,1)$ und $\text{binom}(1,0)$ je zweimal berechnet wird. Die Anzahl der Mehrfachaufrufe wächst mit n . So kommt es beim Aufruf $\text{binom}(6,3)$ (nicht eingezeichnet) bereits je 4 mal zum Aufruf $\text{binom}(2,1)$. Rechts: Bei der Berechnung der 5. Zeile gibt es 22 rekursive Aufrufe. Viele der Aufrufe werden mehrfach durchgeführt.

In Listing 2.40 wurde das Programm `binom` aus Listing 2.38 daher so geändert, dass die Einträge des Pascalschen Dreiecks in einem mehrdimensionalen Array gespeichert werden. Auf diese Weise kann einfach auf bereits berechnete Binomialkoeffizienten zugegriffen werden, ohne sie neu zu berechnen. Der Unterschied der Effizienz der Algorithmen in Listing 2.38

Listing 2.40: Eine rein iterative Lösung zur Berechnung des Pascalschen Dreiecks.

```

1 import java.util.Scanner;
2
3 public class Pascal3 {
4
5     public static void main(String[] args) {
6
7         Scanner sc = new Scanner(System.in);
8
9         int zeilen = sc.nextInt();
10
11         int[][] binom = new int[zeilen+1][];
12
13         for (int n = 0; n < zeilen; n++) {
14
15             binom[n] = new int[n+1];
16             //Erstes und letztes Element der Zeile mit
17             //dem Wert 1 belegen:
18             binom[n][0] = 1;
19             binom[n][n] = 1;
20
21             //Alle anderen Elemente als Summe der
22             //entsprechenden Elemente der Vorgängerzeile
23             for (int k = 1; k < n; k++) {
24
25                 binom[n][k] = binom[n-1][k-1] + binom[n-1][k];
26                 System.out.print(binom[n][k]+" ");
27
28             }
29
30             System.out.println();
31         }
32
33     }
34
35 }

```

und Listing 2.40 wird deutlich, wenn die Anzahl der auszugebenden Zeilen größer wird (machen Sie einen Versuch mit 30 Zeilen).

2.6 Zustände

Zustände spielen in imperativen Programmiersprachen eine wichtige Rolle. Jede Variable hat einen Zustand, nämlich ihren Wert. Allgemeiner ausgedrückt hat jedes Objekt, beispielsweise ein Array-Objekt, einen Zustand, der sich aus der Gesamtheit der Zustände seiner Elemente ergibt. Der *Programmzustand* ist ein "Schnappschuss" aller definierten Variablen. Er umfasst also die Zustände aller vom Programm verwendeten Variablen zu einem bestimmten Zeitpunkt der Ausführung.

2.6.1 Seiteneffekte

Unter Seiteneffekt versteht man die Änderung des Programmzustands. Die Auswertung von Ausdrücken üblicherweise keine Seiteneffekte, da nur ein Wert berechnet wird. Eine Zuweisung dagegen hat einen Seiteneffekt, da sie den Programmstatus ändert. Betrachten wir zunächst folgende Ausdrücke:

```

(i + 1) + (i + 1)
2 * (i + 1)
(i == j) && (i == j)

```

Jeder dieser Ausdrücke ist eine Verschachtelung von Operatoren mit jeweils 2 Operanden. Die Auswertung dieser Ausdrücke liefert einen Wert, der weiter verwendet werden kann (z.B. auf der rechten Seite einer Zuweisung). Die Auswertung alleine bewirkt jedoch keine Zustandsänderung. Diese Ausdrücke haben daher keinen Seiteneffekt. Weiters hängt der Wert des Ausdrucks alleine vom Ausdruck selbst ab, d.h. von seinen Variablen, und nicht vom restlichen Programmzustand.

Es ist daher möglich den zweiten Ausdruck anstelle des ersten zu verwenden, beide ergeben an derselben Stelle im Programm denselben Wert. Genauso können wir den dritten Ausdruck mit Hilfe der Äquivalenz $(a \&\& a) == a$ durch $i == j$ ersetzen.

Betrachten wir nun folgende Ausdrücke mit Seiteneffekt:

```

(i++) + (i++)
2 * (i++)
(++i == j) && (++i == j)

```

Auch diese Ausdrücke liefern einen Wert. Gleichzeitig wird aber bei der Auswertung der Wert von `i` verändert. Daher ist der erste und der zweite Ausdruck nicht mehr äquivalent. Weiters liefert im ersten Ausdruck der linke Teilausdruck einen anderen Wert als der rechte. Der dritte Ausdruck ist immer *false*.

2.6.2 Funktionen und Prozeduren in der Programmiersprache Pascal

In vielen Programmiersprachen wird zwischen Funktionen und Prozeduren unterschieden. In Java werden beide Konstrukte durch das Konzept der Methode zusammengefasst. Funktionen in Programmiersprachen sind analog zur mathematischen Funktion. Reine Funktionen verhalten sich wie Ausdrücke ohne Seiteneffekt. Einer Funktion wird ein Wert übergeben und sie liefert als Ergebnis einen Wert zurück. Der Funktionsaufruf lässt den Programmzustand unverändert (solange das Ergebnis nicht zugewiesen wird). In Java lassen sich solche reinen Funktionen durch Methoden ohne Seiteneffekt implementieren.

Dagegen haben Prozeduren oft keinen Rückgabewert und stattdessen einen Seiteneffekt. Prozeduren greifen meist auf Objekte zu, die auch außerhalb des zur Prozedur gehörenden Blocks gültig sind.

Listing 2.41 zeigt die Definition einer Funktion in der Programmiersprache Pascal. Die Funktion berechnet die Summe der ganzen Zahlen im Intervall `m` bis `n`, die als formale Parameter in der Funktionsdeklaration angegeben sind. Der Datentyp `cardinal` steht in Pascal für positive ganze Zahlen und wird für die beiden Parameter gemeinsam nach dem Doppelpunkt innerhalb der runden Klammern angegeben. Der Rückgabebetyp der Funktion, der links der schließenden Klammer nach dem Doppelpunkt angegeben wird, ist ebenfalls `cardinal`. Die Funktion benutzt weiters zwei lokale Variablen `i` und `s`, ebenfalls vom Typ `cardinal`, die nur im folgenden Anweisungsblock gültig sind. Der eigentliche Anweisungsblock befindet sich zwischen den Schlüsselwörtern `begin` und `end`. Das Zuweisungssymbol ist `:=`. Die Anweisung `sum := s` entspricht der Anweisung `return s`; in Java.

Der Aufruf der Funktion kann beispielsweise folgendermaßen aussehen:

```
a := 5; b := 10;
c := sum(a,b);
writeln(c)
```

Listing 2.41: Eine Funktion in der Programmiersprache Pascal zur Berechnung der Summe ganzer Zahlen `m` bis `n`

```
function sum(m:cardinal, n:cardinal):cardinal;
var s: cardinal;
begin
  s := 0;
  while m <= n do
  begin
    s := s + m;
    m := m + 1
  end;

  sum := s
end;
```

In der zweiten Zeile wird das Ergebnis der Funktion, aufgerufen mit den Werten von `a` und `b`, der Variable `c` zugewiesen. In der dritten Zeile wird das Ergebnis ausgegeben.

Das Ergebnis des Aufrufs hängt nur von den Werten der aktuellen Parameter ab. Beim Aufruf der Methode werden die Werte der aktuellen Parameter an die formalen Parameter zugewiesen. Die Funktion arbeitet also mit *Kopien* der vom Aufrufer übergebenen Werte (siehe Abbildung 2.42). Werden diese beim Methodenablauf geändert, beeinflusst dies das aufrufende Programm nicht. Ein solcher Aufruf wird *call-by-value* ("Aufruf mit Wertzuweisung") genannt, da den formalen Parametern ein *Wert* zugewiesen wurde.

In Listing 2.43 wird eine Definition einer Prozedur `addsum` in Pascal gezeigt. Die Prozedur addiert so wie die Funktion aus Listing 2.41 die Summe der ganzen Zahlen von `m` bis `n`, jedoch hat sie keine Rückgabe. Stattdessen hat sie einen weiteren Parameter.

Die Definition des dritten Parameters (nach dem Schlüsselwort `var`) lässt erkennen, dass dieser eine besondere Rolle hat: Es ist ein *Variablenparameter*. Ihm wird nicht der Wert des aktuellen Parameters zugewiesen, sondern stattdessen wird vom Compiler seine Speicheradresse gleich der Adresse der vom Aufrufer übergebenen Variable gesetzt. Auf diese Weise gibt es nun zwei Namen für *dieselbe* Variable (siehe Abbildung 2.44). Einen solchen Aufruf bei dem nicht Werte zugewiesen werden, sondern Speicher-

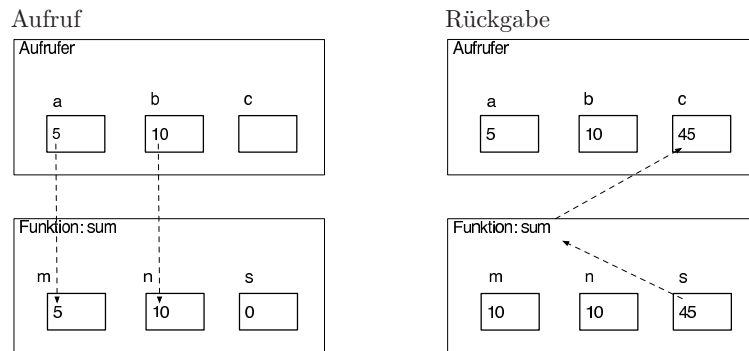


Abbildung 2.42: Aufruf der Pascal Funktion `sum` mit *call-by-value*-Schnittstelle. Die Zuweisungen, die implizit beim Funktionsaufruf durchgeführt werden sind als gestrichelte Pfeile dargestellt. Die Werte der Parameter ändern sich während der Berechnung. Für den Aufrufer sind diese Änderungen aber ohne Auswirkungen, da diese Änderungen nur lokale Variablen betreffen. Bei der Rückkehr aus dem Funktionsaufruf wird der Wert 45 geliefert, der der Variablen `c` zugewiesen wird.

Listing 2.43: Eine Prozedur in der Programmiersprache Pascal zur Berechnung der Summe ganzer Zahlen `m` bis `n`

```

procedure addsum(m,n:cardinal; var s:cardinal);
begin
  while m <= n do
  begin
    s := s + m;
    m := m + 1;
  end;
end;

```

adressen (Referenzen) gleichgesetzt werden, nennt man *call-by-reference*. Die beiden Parameter `m` und `n` sind dagegen *Wertparameter*. Ihnen wird beim Aufruf ein Wert zugewiesen (*call-by-value*).

Alle Änderungen, die die Prozedur auf der Variable `s` durchführt, wirken sich auch auf `c` aus, da `s` und `c` Namen für dieselbe Variable sind.

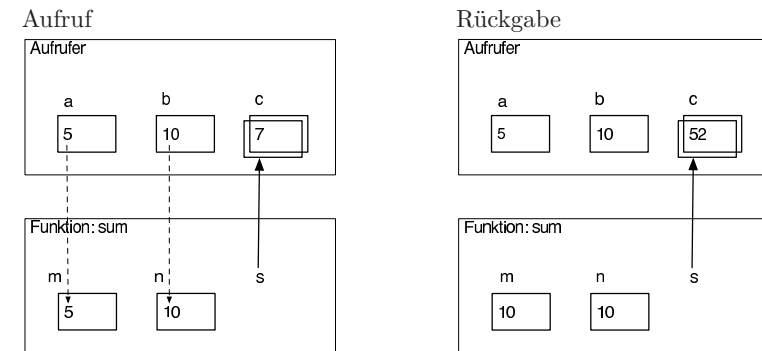


Abbildung 2.44: Aufruf der Pascal Funktion `addsum` mit *call-by-reference*-Schnittstelle. `s` ist ein *Variablenparameter*, während `m` und `n` die gewohnten *Wertparameter* sind. Die Zuweisungen, die implizit beim Funktionsaufruf durchgeführt werden sind als gestrichelte Pfeile dargestellt. Der durchgehende Pfeil veranschaulicht, dass mit dem Namen `s` im Block der Prozedur dieselbe Variable angesprochen wird, wie mit dem Namen `c` im Aufrufer. Er kann als Referenz aufgefasst werden.

Die Prozedur hat daher eine "nachhaltige" Wirkung. Die Ausgabe des folgenden Programmstücks ist 52:

```

a := 5; b := 10; c := 7;
addsum(a,b,c);
Writeln(c)

```

Mit Hilfe eines Variablenparameters kann man der Aufrufer also sowohl Werte an die Prozedur übergeben als auch Ergebnisse von der Prozedur erhalten (manchmal wird auch der Begriff *Durchgangparameter* benutzt). `addsum` addiert die Summe der natürlichen Zahlen von `a` bis `b` zu `c` hinzu. Prozeduren funktionieren also wie Zuweisungen oder andere Ausdrücke mit Seiteneffekt in Java.

2.6.3 Methoden mit Seiteneffekt in Java

In Pascal und vielen anderen Programmiersprachen gibt es die Unterscheidung zwischen Wertparametern (*call-by-value*) und Variablenparametern

(*call-by-reference*). In Java gibt es – streng genommen – diesen Unterschied nicht, obwohl auch hier oft von *call-by-reference* gesprochen wird, wenn ein formaler Parameter einen Referenztyp hat. Bei einem Methodenaufruf wird jedoch immer der *Inhalt* einer Variable an den formalen Parameter zugewiesen und nie eine Adresse gleichgesetzt. Jedoch ist es auch möglich, mit Hilfe von formalen Parametern mit Referenztyp Methoden mit Seiteneffekt zu definieren, die ähnlich funktionieren wie Prozeduren in Pascal.

Die iterative Methode `ggt(int, int)` in Listing 2.25 und die rekursive Methode aus Listing 2.30 sind reine Funktionen und verhalten sich wie Ausdrücke ohne Seiteneffekt. Es können zwar während der Ausführung der Methode Zustandsänderungen auftreten, diese beschränken sich aber auf den Speicherbereich der Methodenausführung, da ausschließlich lokale Variablen (Wertparameter) manipuliert werden, die für übergeordnete Programmteile (Aufrufer) nicht sichtbar sind.

Beim Aufruf der Methode werden die Werte der aktuellen Parameter an die formalen Parameter zugewiesen (*call-by-value*). Der Wert, den die Methode liefert, hängt ausschließlich von den aktuellen Parametern ab und ist unabhängig vom restlichen Programmzustand zum Zeitpunkt des Aufrufs. So kann beispielsweise `ggt(i, j) + ggt(i, j)` stets durch `2 * ggt(i, j)` ersetzt werden.

Listing 2.45 zeigt ein Beispiel einer Methode `r` mit Seiteneffekt. Im Gegensatz zu `ggt` manipuliert sie ein Objekt, das auch für übergeordnete Programmteile sichtbar ist und auch vor und nach dem Ablauf der Methode existiert. Dies wird dadurch ermöglicht, dass der Parametertyp ein Referenztyp ist. Beim Aufruf der Methode muss also als aktueller Parameter eine Referenz zugewiesen werden. Wie bei einfachen Werten wird diese Referenz bei der Zuweisung kopiert. Das verknüpfte Objekt wird dabei aber nicht dupliziert.

Weitere Referenzen auf dasselbe Objekt können in übergeordneten Programmteilen existieren, daher kann auch von dort auf das Objekt zugegriffen werden. Alle Veränderungen die `r` beim Ablauf auf dem Objekt bewirkt, sind auch für alle Programmteile sichtbar, die eine über eine Referenz auf das Objekt verfügen. Die Aufrufsstelle ähnelt einer *call-by-reference*-Schnittstelle, weil den formalen Parametern mit Referenztyp eine *Referenz* zugewiesen wird.

Jede Veränderung bleibt auch nach Verlassen der Methode erhalten, da keine Kopien für die Methode erzeugt werden. Der Nachteil hierbei besteht darin, dass eine ungewollte Beeinflussung von Hauptprogrammvariablen

Listing 2.45: Methoden mit Seiteneffekt.

```

1 import java.util.Arrays;
2
3 public class SequenceTest {
4     public static void main(String[] args) {
5         int[] ia = {3,2,6,5,4,1};
6
7         System.out.println(r(ia)+r(ia));
8         //4
9
10        System.out.println(2*r(ia));
11        //6
12
13        System.out.println(2*r(ia));
14        //2
15    }
16 }
17
18 /**
19  * kehrt die Reihenfolge im Array um und liefert danach
20  * das letzte Element zurück.
21  */
22 public static int r (int[] seq) {
23
24     for (int i = 0; i < seq.length/2; i++) {
25         int swap = seq[seq.length-i-1];
26         seq[seq.length-i-1] = seq[i];
27         seq[i] = swap;
28     }
29
30     return seq[seq.length-1];
31 }
32 }
33 }
```

im Unterprogramm möglich ist.

2.6.4 Funktionaler vs. prozeduraler Programmierstil

Wie bereits im Abschnitt 1.5.2 erwähnt, lässt sich jeder Algorithmus, je nach Programmierstil, als Verschachtelung von Ausdrücken (rein funktionaler Stil) oder als Folge von Operationen mit Seiteneffekt (prozeduraler Stil) ausdrücken, wobei Prozeduraufrufe so wie Zuweisungen als Opera-

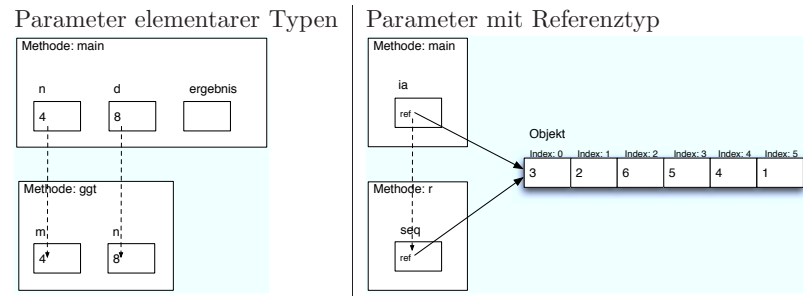


Abbildung 2.46: Zwei Arten von Parametern: Parameter mit elementarem Datentyp (links) und mit Referenztyp (rechts). Der links dargestellte Aufruf wird in Listing 2.30 (Zeile 16) durchgeführt. Der Aufruf rechts wird in Listing 2.45 (z.B. Zeile 7) durchgeführt. Referenzen sind als durchgezogene Pfeile dargestellt. Die Zuweisungen, die implizit beim Methodenaufruf durchgeführt werden sind als gestrichelte Pfeile dargestellt. Die Verwendung von Parameter mit Referenztyp ist eine Möglichkeit, Methoden mit Seiteneffekten (Prozeduren) zu definieren.

tionen mit Seiteneffekt gesehen werden können. In der funktionalen Programmierung hat die Auswertung von Ausdrücken und Funktionen keine Zustandsänderungen zur Folge. Es gibt in rein funktionalen Sprachen keine Anweisungen im Sinn der imperativen Programmierung, die eine Zustandsänderung bewirken, sondern nur Ausdrücke ohne Seiteneffekt.

Obwohl Java keine funktionale Programmiersprache ist, lassen sich Algorithmen auch funktional ausdrücken. Als Beispiel haben wir in Abschnitt 2.4 zwei verschiedene Möglichkeiten gesehen, den größten gemeinsamen Teiler von zwei Werten $ggt(m, n)$ zu berechnen. Vergleicht man die iterative Methode in Listing 2.25 mit der rekursiven Methode aus Listing 2.30, erkennt man, dass die rekursive Variante im Gegensatz zur iterativen Variante ohne Zuweisungen im Methodenrumpf, also ohne Zustandsänderungen von lokalen Variablen, auskommt. Es wird nur definiert, welche aktuellen Parameter die verschachtelten (rekursiven) Funktionsaufrufe haben. Die beim Methodenaufruf angegebenen aktuellen Parameter ändern sich beim Ablauf der Methode nicht. Der Ablauf gleicht also eher einer Auswertung. Die Methodendefinition folgt einem funktionalen Programmierstil. Die Definition der Methode ähnelt der mathematischen

rekursiven Definition des GGT. Ganz ähnlich ist es bei der Implementierung der Methode $ggt(int, int, int)$. Hier wird im Methodenrumpf nur ein verschachtelter Ausdruck ausgewertet. Dieses Beispiel zeigt, wie der funktionale Programmierstil ohne Zustände und Seiteneffekte auskommt.

Die iterative Methodendefinition betont hingegen die dynamisch Abfolge von Schritten zur Berechnung des Resultats. Die Ausführung bewirkt Zustandsänderungen von lokalen Variablen.

2.7 Kommunikation mit der Außenwelt

Kommt ein Programm zur Ausführung kann es auf unterschiedliche Arten mit der Außenwelt kommunizieren. Dabei werden Daten eingelesen, verarbeitet und eine Ausgabe erzeugt. Das Programm in Ausführung ist also ein Ein- und Ausgabe-System. Solche Systeme lassen sich nach der Art und Weise, wie sie mit der Außenwelt in Verbindung treten, in *transformatorische Systeme* oder *reaktive Systeme* unterteilen (siehe Abbildung 2.47). Diese Unterscheidung ist im Hinblick auf die Komplexität des Systemverhaltens wichtig. Transformatorische Systeme sind analog zu den reinen Funktionen (z.B. $ggt(int, int)$ im Abschnitt 2.4) ohne inneren Zustand, d.h. das Ein- zu Ausgabe-Verhalten ist unabhängig vom Zeitpunkt des Aufrufs. Im Gegensatz dazu haben reaktive Systeme einen inneren Zustand, der durch Eingaben verändert werden kann und der das Ein- zu Ausgabe-Verhalten mitbestimmt. Es kommt also eine zeitliche Dimension dazu. Die Laufzeit ist potentiell unbegrenzt und das System befindet sich in ständiger Interaktion mit seiner Umgebung. Ein Beispiel für ein reaktives System ist ein Betriebssystem oder ein Airbag-Steuersystem. Java Programme sind nur selten reine Funktionen sondern oft reaktive Systeme.

Wir wollen als Abschluss dieses Kapitels einen kurzen Einblick geben über Möglichkeiten zur Ein- und Ausgabe in Java:

2.7.1 Die Methode main

Die Methode `main` ist beim Ablauf eines Java-Programms der Einstiegspunkt. Wird eine Java-Klasse ausgeführt, ist die `main` Methode die erste, die vom Laufzeitsystem ausgeführt wird. Sie kann Anweisungen zur Ausführung anderer Methoden enthalten. Soll eine Java-Klasse ausführbar sein, muss diese Methode definiert sein. Der Methodenkopf muss wie folgt

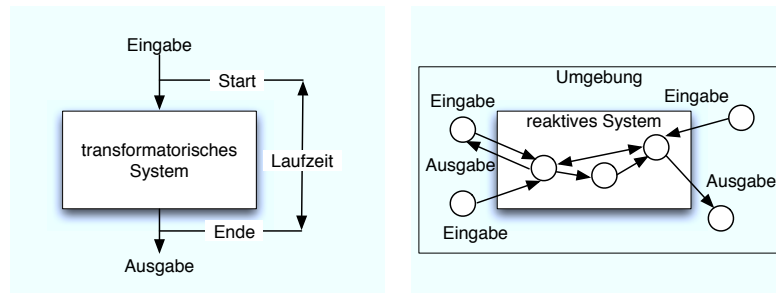


Abbildung 2.47: Die linke Darstellung zeigt ein *transformatorisches System*, das mit einer bestimmten Eingabe gestartet wird und nach begrenzter Laufzeit mit einer Ausgabe terminiert. Das System arbeitet ansonsten entkoppelt von der Außenwelt und es gibt keinen nach außen sichtbaren Zustand, d.h. die Ausgabe hängt ausschließlich von der Eingabe ab. Die rechte Darstellung zeigt ein *reaktives System*, das potentiell unendlich lange läuft und einen durch Eingaben veränderbaren inneren Zustand besitzt, der das Verhalten des Systems bestimmt. Das System ist in eine Umgebung eingebunden und *reagiert* auf Eingaben aus unterschiedlichen Quellen (z.B. Sensordaten) mit entsprechenden Ausgaben. Während transformatorische Systeme keine zeitliche Dimension haben sind reaktive Systeme *dynamische Systeme*, d.h. der sich über die Zeit ändernde Zustand spielt eine wesentliche Rolle.

aussehen:

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

Die Methode muss als öffentliche, statische Methode ohne Rückgabe mit der Signatur `main(String[])` definiert werden. Andernfalls wird diese Methode nicht als Einstiegspunkt von der virtuellen Maschine erkannt.

Die Deklaration von `main` beinhaltet einen formalen Parameter vom Typ `String[]`. Die aktuellen Parameter können beim Start der Klasse als durch Leerzeichen getrennte Zeichenketten angegeben werden. Da es sich um einen Parameter eines Array-Typs handelt, ist es möglich beliebig viele Zeichenketten als Argumente beim Programmstart anzugeben. Die Argumente werden immer als Zeichenketten den formalen Parametern zu-

gewiesen, auch wenn sie einzelne Zeichen oder Zahlen repräsentieren. Daher müssen die Zeichenketten in der Methode `main` ggfs. in numerische Werte umgewandelt werden. Dazu lassen sich beispielsweise die Methode `Integer.parseInt` für Integer-Zahlen und `Float.parseFloat` für Fließkommazahlen nutzen.

In Listing 2.48 ist ein Beispiel angegeben. Das Programm akzeptiert beim Programmstart eine beliebig lange Liste von positiven ganzen Zahlen und berechnet den größten gemeinsamen Teiler dieser Zahlen. Ein Aufruf des Programms von einem Kommandozeileninterpreter (Konsole) kann beispielsweise so aussehen:

```
$ java Euklid 16 8 128 12
```

Falls sich eine der angegebenen Zeichenketten nicht in eine ganze Zahl umwandeln lässt, bricht das Programm mit einer `NumberFormatException`⁷ ab. Beim Programmstart muss mindestens ein Argument angegeben werden, da das erste Argument (mit dem Index 0) in Zeile 4 gelesen wird. Wurden beim Programmaufruf keine Argumente angegeben, bewirkt die Anweisung in Zeile 4 einen Programmabbruch mit einer `ArrayIndexOutOfBoundsException`.

Listing 2.48: Ausführbare Klasse.

```
1 public class Euklid {
2     public static void main(String[] args) {
3
4         int ergebnis = Integer.parseInt(args[0]);
5
6         for (String input: args) {
7             ergebnis = ggt(ergebnis, Integer.parseInt(input));
8         }
9
10        System.out.println(ergebnis);
11    }
12
13    public static int ggt(int m, int n) {
14
15        return m == n ? m : m > n ? ggt(m - n, n) : ggt(m, n - m);
16
17    }
18 }
```

⁷Das Thema Ausnahmen (engl. *exceptions*) und deren Behandlung wird in späteren Kapiteln besprochen

19 }

Die Methode `main` hat keine Rückgabe, ist daher keine Funktion. In diesem Beispiel verarbeitet sie Eingabedaten und führt entsprechende Methodenaufrufe durch. Das Ergebnis wird durch das Erzeugen einer Ausgabe geliefert. Eine weitere Möglichkeit, Daten einzulesen besteht beispielsweise durch Verwendung der Klasse `Scanner`. Mit Hilfe eines Scanner-Objektes können Daten an beliebiger Stelle im Programm (also nicht nur zum Zeitpunkt des Aufrufs) eingelesen werden. So folgt der Zeitpunkt der Ein- und Ausgabe ganz dem Programmfluss, der durch die Programmstruktur bestimmt wird.

2.7.2 Kontrollfragen

- Was ist ein Algorithmus?
- Wie unterscheidet sich ein Kochrezept von einem Algorithmus?
- Was sind Ähnlichkeiten und Unterschiede von Objekten, Werten und Daten?
- Welche Möglichkeiten zur Steuerung des Programmflusses gibt es?
- Was ist eine Sequenz, Fallunterscheidung oder Iteration?
- Was ist ein Ausdruck?
- Was ist eine Variable?
- Was ist eine Zuweisung?
- Was ist eine Speicheradresse?
- Was ist der Unterschied zwischen Lebensdauer, Gültigkeitsbereich und Sichtbarkeitsbereich?
- Was ist der Unterschied zwischen Deklaration und Definition?
- Was ist eine Initialisierung?
- Erfolgt die Initialisierung automatisch?
- Was sind Datentypen?

- Welche elementaren Datentypen gibt es?
- Was sind Referenztypen?
- Was ist ein Operator, was ein Operand?
- Was heißt infix, präfix und postfix?
- Was ist ein L-Wert, was ein R-Wert?
- Was bedeutet `final`?
- Was ist eine Ausdrucksanweisung?
- In welcher Reihenfolge werden Ausdrücke ausgewertet?
- Was ist eine Operatorpriorität?
- Was ist die Assoziativität?
- Welche einstelligen und zweistelligen Operatoren gibt es in Java?
- Gibt es in Java dreistellige Operatoren?
- Wie wird bei einer Division gerundet?
- Was bedeutet `Infinity` und `NaN`?
- Was ist der Restwertoperator?
- Was ist ein Verkettungsoperator?
- Wie verhalten sich Zuweisungsoperatoren?
- Welche relationale Operatoren gibt es?
- Wie unterscheiden sich logische Operatoren von Bitoperatoren?
- Was macht der Bedingungsoperator?
- Was ist eine Typumwandlung?
- Was passiert bei einer Typumwandlung?
- Wie unterscheidet sich die einschränkende von der erweiternden Typumwandlung?

- Wann erfolgt eine implizite Typumwandlung, wann benötige ich eine explizite Typumwandlung?
- Was sind konstante Ausdrücke?
- Was ist ein Literal?
- Welche Fehlermöglichkeiten gibt es bei der Definition von Literalen?
- Was ist ein Block?
- Was ist ein innerer Block?
- Darf in einem inneren Block eine Variable mit dem selben Namen wie in einem äußeren Block verwendet werden?
- Welche Anweisungen können zur Selektion verwendet werden?
- Erklären Sie die `switch`-Anweisung!
- Was ist eine Funktion?
- Was sind Unterprogramme, was sind Routinen?
- Was ist eine Methode?
- Wie unterscheiden sich Funktionen, Methoden, Unterprogramme, Routinen und Prozeduren?
- Was ist der Unterschied zwischen formalen und aktuellen Parametern?
- Welche Aufgabe hat die `return`-Anweisung?
- Was bedeutet Überladen?
- Was geschieht bei einem Methodenaufruf?
- Was bedeutet Rekursion?
- Wie unterscheiden sich Rekursion und Iteration?
- Was sind Zusicherungen?
- Welche Arten von Zusicherungen gibt es?
- Warum verwendet man Zusicherungen?

- Mit welchen Anweisungen kann eine Iteration implementiert werden?
- Was ist ein Array?
- Was sind Indextypen, was sind Elementtypen?
- Werden Arrays automatisch initialisiert?
- Was ist eine Initialisierungsliste?
- Was ist ein mehrdimensionales Array?
- Wie kann eine `for`-Schleife spezifiziert werden?
- Was ist der Programmzustand?
- Was ist ein Seiteneffekt?
- Was ist der Unterschied zwischen Funktionen und Prozeduren in der Programmiersprache Pascal?
- Was ist ein Wertparameter, was ist ein Referenzparameter?
- Was bedeutet `call-by-value`, was `call-by-reference`?
- Wie unterscheidet sich der funktionale vom prozeduralen Programmierstil?
- Was ist ein transformatives, was ein reaktives System?
- Was ist das besondere an der Methode `main`?

3 Objektorientierte Konzepte

Java unterstützt vor allem das objektorientierte Paradigma der Programmierung und erweitert damit das prozedurale Paradigma, mit dem wir uns in Kapitel 2 auseinandergesetzt haben. Der wesentliche und namensgebende Aspekt ist die Unterstützung benutzerdefinierter Objekte als wichtigstem Abstraktionsmittel. Vor allem große Programme profitieren davon. Zusammen mit inkrementellen und zyklischen Softwareentwicklungsprozessen – siehe Abschnitt 1.6.1 – hat sich die objektorientierte Programmierung in den letzten Jahren zu dem in der Praxis am häufigsten eingesetzten Programmierparadigma entwickelt. Wir wollen uns in diesem Kapitel damit beschäftigen, worauf es in der objektorientierten Programmierung ankommt und durch welche Sprachkonzepte Java dazu beiträgt, die objektorientierte Programmierung angenehmer zu gestalten.

3.1 Das Objekt

Das Objekt steht im Zentrum aller Überlegungen zur objektorientierten Programmierung. Wir betrachten zunächst auf abstrakte Weise, was wir unter einem Objekt verstehen und was wir uns von der objektorientierten Programmierung erwarten. Danach beschreiben wir ganz allgemein die wichtigsten Eigenschaften von Softwareobjekten.

3.1.1 Abstrakte Sichtweisen

Wesentliche Einheiten der in Kapitel 2 gezeigten Programme sind Funktionen und Prozeduren (die in Java Methoden heißen – siehe Abschnitt 2.4). Wir haben also einen *prozeduralen Programmierstil* benutzt. Dieser Programmierstil ist dadurch gekennzeichnet, dass ein Programm, den verwendeten Algorithmen entsprechend, in sich gegenseitig aufrufende, den Programmzustand verändernde Prozeduren zerlegt wird. Die prozedurale Programmierung ist sehr gut dazu geeignet, kleine Programme zu schreiben, die einzelne Algorithmen implementieren.

Das primäre Einsatzgebiet der *objektorientierten Programmierung* ist dagegen die Konstruktion großer Programme mit einem langen Softwarelebenszyklus. Alleine daraus können wir schon einige Problembereiche und Ziele ableiten:

- Große Programme beruhen auf einer Vielzahl an einzelnen Algorithmen, die auf komplexe Weise miteinander verbunden sind. Im Mittelpunkt der objektorientierten Programmierung steht nicht so sehr die Implementierung einzelner Algorithmen, wie wir das von der prozeduralen Programmierung kennen, sondern die Integration vieler Algorithmen zu einer Einheit. Natürlich ist es auch in der objektorientierten Programmierung notwendig, einzelne Algorithmen zu implementieren. Wenn aber die Größe und Komplexität der Aufgabe steigt, sind wir als Menschen (auch mit sehr viel Programmiererfahrung) nicht mehr in der Lage, die ganze Aufgabe zu überblicken und durch einen einzigen Algorithmus zu lösen. Wir können nur einzelne Teilaufgaben unabhängig voneinander durch je einen eigenen Algorithmus lösen. Mit der Summe der Teile ist aber noch lange nicht die ganze Aufgabe gelöst. Wir müssen die einzelnen Teile noch so miteinander verbinden, dass daraus ein in sich konsistentes ganzes Programm entsteht. In der objektorientierten Programmierung brauchen wir Mittel und Wege, um Programmteile weitgehend unabhängig voneinander zu halten und trotzdem miteinander zu verbinden.
- Aufgrund der großen Komplexität ist es in der Regel auch nicht möglich, ein großes Programm in einem einzigen Schritt aus vielen einzelnen Teilen zusammenzusetzen. Das entstehende Programm könnten wir nicht durchschauen. Viel eher verwenden wir inkrementelle Softwareentwicklungsprozesse, in denen wir Anfangs nur einen kleinen Teil der Aufgabe lösen und das Programm nacheinander, Schritt für Schritt, um weitere Teile ergänzen. Dabei nehmen wir in den vorangegangenen Schritten gewonnene Erfahrungen in die nächsten Schritte mit. Die objektorientierte Programmierung hat zum Ziel, Programme im Laufe der Zeit ausbauen zu können, ohne bereits bestehende Programmteile ständig umändern zu müssen.
- Langlebige Software muss über einen langen Zeitraum gewartet werden. Die Wartung ist sehr aufwendig und kostenintensiv. Häufig fließt ein Mehrfaches der eigentlichen Entwicklungskosten in die Wartung. Es zahlt sich daher aus, Programme so zu schreiben, dass sie mög-

lichst gut wartbar sind. Die einfache Wartung ist ein primäres Ziel der objektorientierten Programmierung.

- Die Struktur eines Programmes ist ein entscheidendes Qualitätsmerkmal hinsichtlich Einfachheit, Verständlichkeit und Wartbarkeit. Welche Struktur gut oder weniger gut geeignet ist, hängt von vielen Faktoren ab, nicht zuletzt von der Art des Programms. Die objektorientierte Programmierung bietet eine große Vielfalt an Möglichkeiten zur Strukturierung und erlaubt uns (mit entsprechender Erfahrung), eine gute Struktur zu wählen.

Diese Ziele sind durchwegs recht anspruchsvoll. Auf den ersten Blick scheint es fast unmöglich zu sein, dass uns etwas so Einfaches wie Objekte den Zielen näherbringen könnte. Softwareobjekte sind ja nichts anderes als Abbildungen gegenständlicher oder abstrakter Objekte aus der realen Welt, etwa Allee-bäume, Kugelschreiber, Notizblöcke, Buchstaben und Zahlen. So gut wie alles kann man als Objekt auffassen. Die Magie der Objekte kommt hauptsächlich dadurch zustande, dass wir als Menschen es gewohnt sind, mit Objekten umzugehen, Objekte miteinander in Beziehung zu setzen und Interaktionen zwischen Objekten zu verstehen. So ist für uns ganz selbstverständlich zu begreifen, was es bedeutet, mit einem Kugelschreiber Buchstaben und Zahlen in einen Notizblock zu schreiben, oder dass der Notizblock aus dem Holz von Bäumen, vielleicht auch einem Alleebaum, erzeugt wurde. Nicht nur Programme, sondern auch die reale Welt, in der wir leben, ist hochgradig komplex. Die Einteilung der Welt in einzelne Objekte ist eine menschliche Vorgehensweise, um mit der Komplexität der Welt zurechtzukommen. Das haben wir von klein auf gelernt. Die objektorientierte Programmierung ist also nichts anderes als ein Versuch, diese menschliche Vorgehensweise im Umgang mit Komplexität auf die Programmierung zu übertragen. Nicht die Programme können gut mit Objekten umgehen, sondern wir Menschen können einfacher mit Programmen umgehen, die ähnlich strukturiert sind wie unsere reale Welt. Nur weil wir in der realen Welt hochgradig komplexe Zusammenhänge zu verstehen gelernt haben, können wir das auch in der Software.

Kaum jemand von uns wird im Detail wissen, wie aus einem Alleebaum ein Notizblock entsteht oder warum eine auf gewisse Art geschwungene Linie für einen bestimmten Buchstaben steht. Trotzdem können wir den Buchstaben im Notizblock problemlos lesen. Genausowenig müssen wir in der objektorientierten Programmierung im Detail wissen, welcher Wert zu welchem Zeitpunkt an welche Variable zugewiesen wird, um zu ver-

stehen, wie die Objekte zusammenhängen und miteinander interagieren. Durch den Umgang mit Objekten abstrahieren wir über Details. In der objektorientierten Programmierung verwenden wir ähnliche Formen der Abstraktion, die wir ganz allgemein im menschlichen Denken einsetzen.

In Abschnitt 1.3.3 haben wir Objekte als abstrakte Maschinen betrachtet. Das hilft uns dabei, ganz unterschiedliche Sichtweisen eines Objekts miteinander zu verknüpfen. Einerseits brauchen wir die oben beschriebene Außenansicht eines Objekts, in der wir über so viele Details wie möglich abstrahieren, um die Komplexität beim Verbinden einzelner Objekte zu einem größeren Ganzen in den Griff zu bekommen. Andererseits müssen wir die Maschine aber auch in allen Details implementieren, da eine Abstraktion alleine nicht lauffähig ist. Irgendwie wurde unser Notizblock hergestellt und hat sich die Schreibweise für Buchstaben entwickelt, auch wenn wir darüber abstrahieren. Trotzdem gibt es nicht einfach nur zwei Sichtweisen, eine abstrakte Außenansicht und eine detailreiche Innenansicht, sondern ein sehr komplexes Gefüge an unterschiedlichen Sichtweisen. Auch jemand, der direkt in die Herstellung des Notizblocks involviert ist, kennt nur einen winzigen Ausschnitt aus dem gesamten Produktionsprozess. So hat jemand, der den Alleebaum umgesägte, wahrscheinlich keine Ahnung davon, dass daraus ein Notizblock entstand. Es grenzt fast an ein Wunder, dass wir ganz ohne zentrale Steuerung einen Notizblock in die Hand bekommen haben. Ähnlich ist es in der objektorientierten Programmierung. Viele abstrakte Maschinen (die von außen als Objekte betrachtet werden) arbeiten zusammen, ohne dass eine Maschine Details der anderen kennt. Zusammen lassen die Maschinen Produkte entstehen, ganz ohne zentrale Kontrolle des Produktionsprozesses.

Aus dem Blickwinkel der Programmierung macht vielleicht gerade dieser Aspekt den entscheidenden Unterschied zwischen einem prozeduralen und objektorientierten Stil aus: In der prozeduralen Programmierung wollen wir uns stets die Kontrolle über alle Details des gesamten Programmablaufs verschaffen. In der objektorientierten Programmierung geben wir dagegen die zentrale Kontrolle auf. Jede abstrakte Maschine (also die Innenansicht eines Objekts) hat zwar volle Kontrolle darüber, wie sie mit anderen Objekten, von denen sie nur die Außenansicht kennt, umgeht. Die Maschinen kommunizieren miteinander und tauschen dabei Informationen aus. Sie geben dabei aber nur so viel über sich preis, wie für eine produktive Zusammenarbeit notwendig ist.

Bei der Programmkonstruktion entwickeln wir eine abstrakte Maschine nach der anderen, jede möglichst unabhängig von den anderen. Neue Ma-

schinen können jederzeit hinzukommen und mit Kommunikationskanälen bereits bestehender Maschinen verbunden werden. Interne Details einer Maschine sind auch im Nachhinein leicht änderbar, weil andere Maschinen ja nichts darüber wissen. Das erleichtert die Wartung. Abstrakte Maschinen sind auf vielfältige Art miteinander kombinierbar. Auf diese Weise können wir problemlos die Struktur unseres Programmes schaffen, die wir uns wünschen.

3.1.2 Faktorisierung: Prozeduren und Objekte

Unter *Faktorisierung* versteht man die Aufteilung großer Programme in kleine Einheiten, in denen zusammengehörige Eigenschaften und Aspekte des Programms zusammengefasst sind. Der Begriff geht aus der Analogie zur Mathematik hervor, wo man unter Faktorisierung das Zerlegen eines mathematischen Objekts (beispielsweise eines algebraischen Terms) in seine Faktoren (Ausklammerung einfacherer Terme) versteht.

In der Programmierung ist der Zweck der Faktorisierung ähnlich: Wenn zum Beispiel mehrere Stellen in einem Programm aus denselben Sequenzen von Befehlen bestehen, soll man diese Stellen durch Aufrufe einer Methode ersetzen, die genau diese Befehle ausführt. Gute Faktorisierung führt dazu, dass zur Änderung aller dieser Stellen auf die gleiche Art und Weise eine einzige Änderung der Methode ausreicht. Bei schlechter Faktorisierung hätten alle Programmstellen gefunden und einzeln geändert werden müssen, um denselben Effekt zu erreichen. Gute Faktorisierung verbessert auch die Lesbarkeit des Programms, beispielsweise dadurch, dass die Methode einen Namen bekommt, der ihre Bedeutung widerspiegelt.

Auch das Programm aus Listing 2.30 zur rekursiven Berechnung des GGT besteht aus kleineren Einheiten. So gibt es beispielsweise die Methode `ggT`. Diese ist als eigene Einheit für die Berechnungen der mathematischen Funktion zuständig und hat eine entsprechende Bezeichnung. Sie arbeitet als reine Funktion unabhängig vom Anweisungsblock von `main`. Daher wäre es leicht möglich, den Algorithmus in ihrer Methodendefinition durch einen anderen GGT-Algorithmus zu ersetzen, ohne dass im restlichen Programm etwas geändert werden müsste.

Prozeduren haben, wenn sie ausgeführt werden, einen eigenen Speicherbereich, der solange existiert, bis die Ausführung der Prozedur beendet ist. Danach wird der von den lokalen Variablen belegte Speicher der Prozedur freigegeben. Der Programmzustand ist dennoch *global* (das heißt, über alle Variablen des Programms) zu bewerten, da eine Prozedur auch auf Va-

riablen außerhalb ihres Blocks zugreifen kann (Variablenparameter bzw. Parameter mit Referenztyp). Vor allem ist der Programmzustand deswegen global, weil wir bei der prozeduralen Programmierung versuchen, die Kontrolle über alle Details zu behalten. Wir glauben das Programm nur dann verstehen zu können, wenn wir auch alle möglichen Programmmzustände in ihrer Gesamtheit verstehen.

Die objektorientierte Programmierung bietet zusätzliche Möglichkeiten zur Faktorisierung. Der Begriff des Objekts rückt in den Mittelpunkt. Einige spezielle, nur auf eingeschränkte Weise verwendbare Arten von Objekten haben wir in Kapitel 2 bereits kennen gelernt. Eine davon sind Zeichenketten vom Typ `String`, die beliebig viele Zeichen aneinanderreihen und ähnlich den Werten elementarer Typen auf funktionale Weise (ohne Seiteneffekte) verwendbar sind. Eine andere Art sind Array-Objekte, also Objekte, die eine bestimmte Anzahl an Elementen desselben Datentyps enthalten. Hierin unterscheiden sie sich von elementaren Werten. Während Array-Objekte aber als Elemente nur Instanzen *eines* bestimmten Datentyps enthalten, können Objekte allgemein auch Instanzen unterschiedlicher Typen gemeinsam enthalten.

Zum Ablegen der Daten verfügt das Objekt über *Objektvariablen*, die manchmal auch *Instanzvariablen*, *Datenfelder* oder (wenn der Kontext klar ist) kurz *Variablen* genannt werden. Eine Objektvariable ist einfach nur eine Variable, die in einem Objekt angelegt ist. Ein Objekt kann beliebig viele Objektvariablen beliebiger Typen enthalten.

Neben Variablen enthält ein Objekt auch Methoden. Die Variablen und Methoden in einem Objekt sind untrennbar miteinander verbunden: Während die Methoden die Daten in den Objektvariablen zur Erfüllung ihrer Aufgaben benötigen, ist die genaue Bedeutung dieser Daten oft nur den Methoden des Objekts bekannt. Ohne die Methoden sind die Daten sinnlos. Die Methoden und Daten stehen zueinander in einer engen logischen Beziehung. Wegen dieser engen Beziehung wird ein Objekt oft als *Kapsel* verstanden, die zusammengehörende Variablen und Methoden zusammenhält. Diese Kapsel bildet eine untrennbare Einheit in der Software – siehe Abbildung 3.1. Das Zusammenfügen von Daten und Methoden zu einer Einheit nennt man daher *Kapselung* (*encapsulation*).

Während die Faktorisierung in der prozeduralen Programmierung ausschließlich über Methoden erfolgt, stehen zur Faktorisierung in der objektorientierten Programmierung auch und vorwiegend die Objekte zur Verfügung, welche Daten und Methoden zu Einheiten zusammenfassen. In der

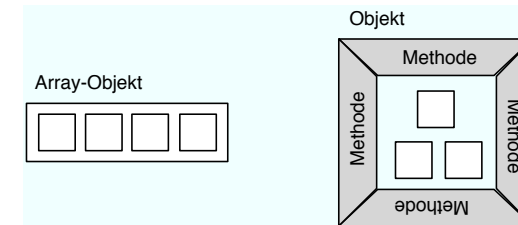


Abbildung 3.1: Links ein Array, deren Elemente von außen über ihren Index direkt zugreifbar sind; rechts ein Objekt als Kapsel mit Objektvariablen (versteckt im Inneren) und Methoden (von außen sichtbar rundherum angeordnet).

objektorientierten Programmierung bestehen Programme zur Laufzeit aus mehreren bis vielen Objekten. Diese kommunizieren miteinander, indem sie Methoden aufrufen, die zu anderen Objekten gehören. Bei Ausführung der Methoden wird in der Regel auf Objektvariablen zugegriffen. Aber üblicherweise greift man nicht direkt auf Variablen eines anderen Objekts zu, da man die genaue Bedeutung dieser Variablen im Gegensatz zu den Methoden nicht kennt.

Die Faktorisierungsmöglichkeiten in der objektorientierten Programmierung sind äußerst vielfältig. Aber bei Weitem nicht jede Faktorisierung ist gut. Wenn man beispielsweise Methoden, die eng zusammenhängen und einander häufig aufrufen, auf zwei verschiedene Objekte verteilt, dann ist auch der Algorithmus, der hinter diesen Methoden steckt, auseinandergerissen. In diesem Fall ist es schwieriger, Änderungen am Algorithmus vorzunehmen. Wenn man umgekehrt Methoden, hinter denen zwei relativ unabhängige Algorithmen stecken, den Algorithmen entsprechend auf zwei Objekte aufteilt, ist es leichter, die Algorithmen unabhängig voneinander zu ändern. Die Qualität der Faktorisierung liegt in unserer Verantwortung.

3.1.3 Datenabstraktion

In Abbildung 3.1 sind die Methoden um die Objektvariablen herum angeordnet. Diese Anordnung soll andeuten, dass die Außenansicht eines Objekts von den Methoden und nicht den Variablen gebildet wird. Während man auf die Elemente eines Arrays von außen direkt zugreifen kann, will man auf die Variablen eines Objekts von außen nicht direkt zugreifen können.

Man soll auf das Objekt nur zugreifen, indem man ihm *Nachrichten* schickt. Eine Nachricht ist eine Aufforderung an das Objekt zur Ausführung einer seiner Methoden und enthält die Information darüber, welche Methode mit welchen aktuellen Parametern ausgeführt werden soll. Eigentlich ist das Schicken einer Nachricht nichts anderes als der Aufruf einer Methode in einem Objekt. In der objektorientierten Programmierung verwendet man (im Vergleich zur prozeduralen Programmierung) eine etwas andere Terminologie um klar zu machen, dass wir es mit voneinander weitgehend unabhängigen Objekten zu tun haben, die miteinander kommunizieren. Auch wenn hinter Methodenaufrufen und dem Nachrichtensenden annähernd dieselben Mechanismen stecken, so ist die pragmatische Verwendung doch genauso unterschiedlich wie die Terminologie. Man denkt beim Nachrichtensenden in erster Linie nicht daran, dass dadurch ein bestimmter Teil eines wohldefinierten Algorithmus ausgeführt wird, sondern hat nur eine grobe Vorstellung davon, was die Methode bewirkt. Wie wir in Abschnitt 3.3 noch sehen werden, kennen wir die Methode, die durch die Nachricht zur Ausführung kommt, im allgemeinen gar nicht im Detail.

An dieser Stelle wäre ein konkretes Java-Beispiel angebracht. Damit haben wir aber ein Problem: In Java lassen sich Objekte nicht direkt ausdrücken, sondern sie werden erst zur Laufzeit als Instanzen von Klassen erzeugt – siehe Abschnitt 3.2. Daher müssen wir uns hier noch mit abstrakten Beispielen begnügen. Nehmen wir an, wir haben ein Objekt, das einen farbigen Punkt in einem zweidimensionalen Koordinatensystem darstellt. Eine Referenz auf dieses Objekt liegt in einer Variablen namens `punkt`. Durch `punkt.verschiebe(1.2,-1.0)` schicken wir diesem Objekt die Nachricht `verschiebe(1.2,-1.0)`. Der Punkt wird darauf reagieren, indem er die Methode `void verschiebe(double,double)` ausführt und (vermutlich) seine Position um 1,2 Einheiten nach rechts und eine Einheit nach unten verschiebt. Wenn die Methode ein Ergebnis zurückgibt, dann liefert eine Nachricht auch eine Antwort zurück. So schicken wir durch `punkt.farbe()` dem Objekt die Nachricht `farbe()`, und eine Ausführung der Methode `String farbe()` im Objekt wird als Ergebnis bzw. Antwort eine Zeichenkette (vielleicht `"rot"`) zurückgeben. In welcher Form das Objekt die Koordinaten speichert und wie die Farbe ermittelt wird, wissen wir nicht. Solche Details bleiben uns verborgen, weil wir das Objekt nur von außen betrachten. Aufgrund der Einfachheit des Beispiels können wir uns aber vorstellen, wie das Objekt von innen betrachtet aussehen könnte. Diese Vorstellung kann aber täuschen. Möglicherweise sind die Koordinaten intern als Polarkoordinaten abgelegt

und nicht als kartesischen Koordinaten – wir wissen es nicht. Es ist auch problemlos möglich, eine Darstellungsform durch eine andere zu ersetzen. Wir müssen nur wissen, dass die Parameter von `verschiebe` im kartesischen Koordinatensystem dargestellt sind.

Oft ist ein Großteil des Inhalts eines Objekts nach außen nicht sichtbar, und wir können uns das interne Aussehen auch nur schwer vorstellen. Beispielsweise ist es beim Lenken eines Fahrzeugs für den Fahrer nicht nötig zu wissen, welche Gelenke und Gestängeteile dabei wie bewegt werden, solange das Bewegen des Lenkrades die erwartete Wirkung zeigt. Die Außenansicht eines Objekts, das die Lenkung darstellt, entspricht nur dem Lenkrad. Wie die Bedienung eines Lenkrads erlernt und geübt werden muss, so ist auch die Verwendung eines Objekts nicht selbstverständlich, auch wenn die interne Funktionsweise der Lenkung dafür nicht bekannt zu sein braucht. Gänzlich andere Fähigkeiten braucht man zur Konstruktion einer Lenkung. Es ist dazu gar nicht nötig, ein Fahrzeug selbst lenken zu können, solange man dem Benutzer ein funktionsfähiges Lenkrad zur Verfügung stellen kann. Genauso unterschiedlich von der Außenansicht wirkt die Innenansicht eines Objekts. Man braucht dem Benutzer nur die zur Verwendung des Objekts nötigen Methoden bereitstellen.

Ein Objekt wird häufig als *Black Box* (den deutschen Begriff „schwarze Schachtel“ verwendet man kaum) verstanden. Das bedeutet, man sieht von außen nur die Form der Schachtel, aber nicht deren Inhalt – ein idealisiertes Bild zur Veranschaulichung des Unterschiedes zwischen Innen- und Außenansicht eines Objekts. Ganz so idealisiert ist die Unterscheidung in der Praxis doch wieder nicht. Manchmal bezeichnet man ein Objekt als *Grey Box*, also als Schachtel, die Teile des Inhalts bei genauem Hinsehen erkennen lässt. Das kann man eventuell mit einem Smartphone vergleichen. Meist weiß man nur, wie man damit telefoniert oder andere Dinge macht, die in der Bedienungsanleitung klar beschrieben sind. Aber manchmal werden auch Details der internen Implementierung erkennbar, wenn man sich die Mühe macht sich anzusehen, wie man damit eigene Applikationen schreibt, oder gelegentlich auch durch Softwarefehler.

Die hier beschriebenen Eigenschaften von Objekten werden unter dem Begriff *Datenabstraktion* zusammengefasst. Es geht im Wesentlichen um zwei Aspekte – Kapselung und *Data Hiding* – die nur zusammen ihre Vorteile ausspielen können. Unter *Data Hiding* versteht man das Verstecken interner Details bzw. das Verhindern direkter Zugriffen auf diese Details von außen. Zur Umsetzung dieser Prinzipien unterscheidet man die

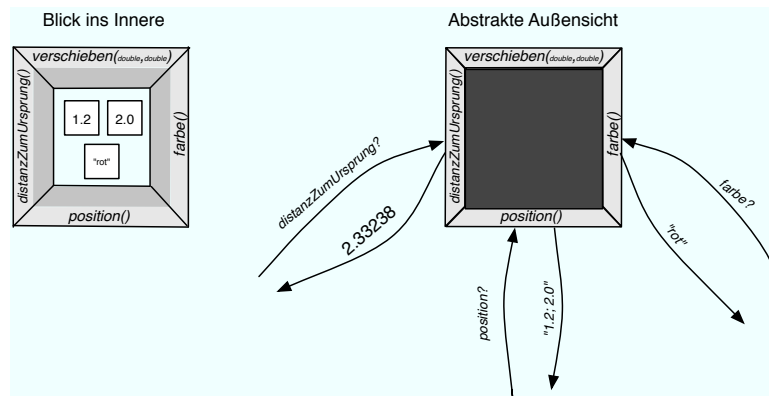


Abbildung 3.2: Links ist die innere Struktur eines Objekts als Kapsel mit Objektvariablen und Methoden dargestellt. Methodendeklarationen sind in hellgrau und deren Implementierungen in dunkelgrau gehalten. Rechts ist Datenabstraktion schematisch dargestellt. Daten und Implementierung der Methoden gehören nicht zur Außensicht und sind im Inneren verborgen. Der interne Zustand, der durch die Werte in den Objektvariablen bestimmt wird, ist nicht sichtbar. Nur die Schnittstelle des Objekts, die aus den Methodendeklarationen und deren abstrakten Beschreibungen gebildet wird, ist von außen sichtbar. Das Objekt reagiert auf Nachrichten (als Pfeile dargestellt) mit entsprechenden Antworten, die vom Zustand des Objekts abhängen können.

Schnittstellen eines Objekts von seiner *Implementierung*. Die Implementierung entspricht der Innenansicht, jede Schnittstelle einer Außenansicht. Wie der Plural erkennen lässt, kann jedes Objekt auch mehrere, unterschiedliche Schnittstellen haben, wie wir in Abschnitt 3.3 sehen werden. Eine Schnittstelle legt fest, welche Nachrichten ein Objekt versteht und (in groben Zügen und auf abstrakte Weise) wie das Objekt auf eine Nachricht reagiert. Beispielsweise legt eine Schnittstelle fest, dass ein Punkt-Objekt die Nachrichten *verschiebe* (mit passenden Argumenten) und *farbe* versteht und darauf durch Verschieben des Punktes um die angegebenen Einheiten in kartesischen Koordinaten bzw. Rückgabe der Farbe des Punktes als Zeichenkette reagiert. Eine andere Schnittstelle könnte nur die Nachricht *verschiebe* (mit einer entsprechenden Beschreibung) festlegen. Verschiedene Schnittstellen erlauben die Verwendung desselben

Objekts zu unterschiedlichen Zwecken. Die Implementierung des Objekts legt das in den Schnittstellen unvollständig beschriebene Verhalten im Detail fest. Die Beschreibung der internen Darstellung der Koordinaten in einem Punkt-Objekt gehört zur Implementierung.

3.2 Die Klasse und ihre Instanzen

Viele objektorientierte Sprachen einschließlich Java beinhalten ein Klassenkonzept: Jedes Objekt gehört zu genau einer Klasse, die die Struktur des Objekts – dessen Implementierung – im Detail beschreibt. Eine Klasse kann man als Bauplan verstehen, nach dem zur Laufzeit neue Objekte dieser Klasse erzeugt werden können. In der objektorientierten Programmierung schreiben wir hauptsächlich Klassen und denken gleichzeitig in Objekten, so wie ein Architekt beim Zeichnen von Bauplänen an entsprechende Gebäude denkt.

3.2.1 Variablen und Methoden

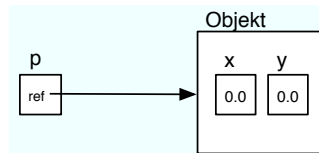
Beginnen wir mit einem Beispiel: In einem Programm soll es Punkte, also Objekte der Klasse `Punkt` geben. Ein Punkt beschreibt eine Position in einem zweidimensionalen Koordinatensystem, wobei die beiden kartesischen Koordinaten durch Fließkommazahlen festgelegt werden. In einer ganz einfachen Variante könnte die Klasse `Punkt` in Java so aussehen:

```
class Punkt {
    double x; // x-Koordinate
    double y; // y-Koordinate
}
```

Auf Grundlage dieser Klassendefinition können neue Punkte erzeugt werden. In folgenden Programmzeilen wird die Klasse `Punkt` benutzt:

```
Punkt p; // Punkt ist Referenztyp
p = new Punkt(); // neue Instanz von Punkt
```

Jede Klasse entspricht einem Referenztyp. Die erste Zeile deklariert daher eine Variable `p` vom Typ `Punkt`, die eine Referenz auf ein Objekt vom Typ `Punkt` enthalten kann. In der zweiten Zeile wird vom einstelligen Präfix-Operator `new` ein Objekt vom Typ `Punkt` erzeugt. Der Operand dieses Operators ist der Name der Klasse des zu erzeugenden Objekts.

Abbildung 3.3: Variable `p` enthält Referenz auf Objekt vom Typ `Punkt`

Auf die Bedeutung der runden Klammern werden wir in Abschnitt 3.2.4 eingehen. Der Rückgabewert des Ausdrucks auf der rechten Seite des Zuweisungsoperators ist eine Referenz auf das neu erzeugte Objekt. Sie wird der Variablen `p` zugewiesen – siehe Abbildung 3.3.

In einem Programm können beliebig viele Objekte derselben Klasse erzeugt werden. Der Ausdruck `new Punkt()` liefert bei jeder Auswertung ein neues Objekt der Klasse `Punkt`. Man nennt die Objekte einer Klasse auch *Instanzen* einer Klasse, genauso wie man von den Instanzen eines Typs spricht. Häufig verwendet man einfach nur den Namen der Klasse oder des Typs zur Bezeichnung von Objekten. Beispielsweise spricht man von „einem Punkt“ oder von „Punkten“, wenn man eine Instanz bzw. mehrere Instanzen von `Punkt` meint. Hinsichtlich der Terminologie ist Vorsicht angebracht: Mit einem (bestimmten oder unbestimmten) Artikel vor einem Begriff oder wenn er im Plural steht, bezieht sich der Begriff auf ein Objekt, ohne Artikel davor und in Einzahl dagegen auf einen Typ oder eine Klasse. Sobald man sich daran gewöhnt hat, wirkt diese Sprechweise ganz selbstverständlich und natürlich.

Prinzipiell ist ein Zugriff auf die Inhalte eines Objekts mit dem Punktoperator „.“ möglich. Der linke Operand ist eine Referenz auf das Objekt, und der rechte ist der Name des angesprochenen Objektinhalts. So kann man mit `p.x` und `p.y` auf die Objektvariablen des Punktes in der Variablen `p` zugreifen. Listing 3.4 zeigt, wie man damit von außen die Objektvariablen setzen und lesen kann.¹

Listing 3.4: Negativbeispiel: `Punkt` und `PunktTester` mit Direktzugriffen

```

class Punkt {
    double x;
    double y;
}

class PunktTester {
    public static void main(String[] args) {
        Punkt p;
        p = new Punkt(); //erster Punkt
        p.x = 1.0;
        p.y = 1.5;

        Punkt q;
        q = new Punkt(); //zweiter Punkt
        q.x = 2.1;
        q.y = -1.2;

        System.out.println("Punkt 1: (" + p.x + ", " + p.y + ")");
        System.out.println("Punkt 2: (" + q.x + ", " + q.y + ")");
    }
}

```

Direkte Zugriffe sind jedoch unerwünscht. Im Sinne der Datenabstraktion sollten solche Zugriffe unterbunden und stattdessen geeignete Schnittstellen definiert werden. So wie `Punkt` derzeit definiert ist, kommen wir nicht ohne direkte Zugriffe aus. Obwohl der Programmcode in Listing 3.4 in dem Sinne richtig ist, dass er vom Compiler übersetzt werden kann und zur Laufzeit das macht, was wir von ihm erwarten, ist er dennoch ein Beispiel dafür, wie man es nicht machen sollen. Durch direkte Zugriffe wird die Erweiterung und Wartung des Programms erschwert. Bei diesem kleinen Beispiel wird uns das gar nicht auffallen, aber bei großen Programmen führt so etwas langfristig zu gewaltigen Problemen.

Wie wir in Abschnitt 3.1 gesehen haben, besteht der eigentliche Zweck eines Objekts darin, Daten und Methoden zu einer Einheit zusammenzufassen – Prinzip der Kapselung. Das machen wir in Listing 3.5. Zusätzlich zu den Objektvariablen enthalten Punkte zwei Methoden. Damit ist jedes Objekt der Klasse `Punkt` nun selbst in der Lage, seine Position zu verändern (Methode: `verschieben`) und seine Distanz zum Ursprung des Koordinatensystems zu ermitteln (`distanzzumUrsprung`). Mit Hilfe

¹In Listing 3.4 wie in vielen weiteren Listings werden mehrere Klassen definiert. Tatsächlich sollte jede Klasse in einer eigenen Datei stehen, die (abgesehen von der Endung `.java`) so heißt wie die Klasse. Mehrere Klassen in einem Listing erhöhen die Übersichtlichkeit ganz kleiner Beispielprogramme. Aber für die Programmierpraxis mit wesentlich größeren durchschnittlichen Klassengrößen sind mehrere Klassen pro Datei gänzlich ungeeignet.

Listing 3.5: Kapselung: Objektvariablen und Methoden bilden eine Einheit

```

1 class Punkt {
2     double x = 0.0;
3     double y = 0.0;
4
5     void verschiebe(double deltaX, double deltaY) {
6         x = x + deltaX;
7         y = y + deltaY;
8     }
9
10    double distanzZumUrsprung() {
11        return Math.sqrt(x*x + y*y);
12    }
13 }
14
15 class PunktTester {
16     public static void main(String[] args) {
17         Punkt p;
18         p = new Punkt();
19         p.verschiebe(1.0,1.5);
20         p.verschiebe(1.0,-1.5);
21
22         System.out.println(p.distanzZumUrsprung());
23     }
24 }

```

dieser beiden Methoden ist es nicht mehr notwendig, von außen direkt auf die Objektvariablen zuzugreifen.

Beim Erzeugen eines neuen Punktes in Zeile 18 werden die Variablen `x` und `y` des neuen Objektes mit `0.0` initialisiert, wie in den Deklarationen der Variablen in den Zeilen 2 und 3 ersichtlich ist. Der Aufruf in Zeile 19 bewirkt, dass das Objekt in der Variablen `p` (das ist das durch den Inhalt von `p` referenzierte Objekt) die Methode `verschiebe` mit den aktuellen Parametern `1.0` und `1.5` ausführt. Man sagt, die Nachricht `verschiebe(1.0,1.5)` wird an `p` geschickt. In den Zeilen 6 und 7 bekommen die Variablen des Objekts durch die Zuweisungen neue Werte. Nach Beendigung der Methodenausführung wird mit der Anweisung in Zeile 20 fortgesetzt. Die Variablen `x` und `y` existieren auch nach Beendigung der Methode weiter, da sie keine lokalen Variablen sind, sondern Objektvariablen. In Zeile 20 wird die Nachricht `verschiebe(1.0,-1.5)` an `p` geschickt und damit erneut die Methode `verschiebe` desselben

Objekts mit anderen aktuellen Parametern aufgerufen. Das führt dazu, dass die Zeilen 6 und 7 erneut ausgeführt werden. Die Variablen `x` und `y` enthalten zu diesem Zeitpunkt noch die Werte, die bei der letzten Verschiebung gespeichert wurden – `1.0` und `1.5`. Zu diesen Werten werden die Parameter addiert. Danach enthalten `x` und `y` die Werte `2.0` und `0.0`. Nachdem ein Punkt in der Methode `main` erzeugt und zweimal verschoben wurde, erzeugt die Anweisung in Zeile 20 die Ausgabe `2.0`.

Die Variante in Listing 3.5 ist aufgrund von Kapselung deutlich besser als jene in Listing 3.4. Dennoch ist auch diese Variante noch nicht perfekt, da es noch immer möglich wäre, direkt auf die Variablen `x` und `y` eines Punktes zuzugreifen, obwohl es dafür keinen Grund gibt. Um Datenabstraktion zu erreichen, benötigen wir noch Data Hiding. Durch gezielte Steuerung der Sichtbarkeit können wir Zugriffe auf `x` und `y` von außen verhindern.

3.2.2 Sichtbarkeit

Im Gegensatz zu lokalen Variablen sind Objektvariablen überall im Programm gültig, wo es eine Referenz auf das Objekt gibt. Das heißt, wenn `p` eine Referenz auf ein Objekt vom Typ `Punkt` ist, dann wissen wir, dass es die Objektvariablen `p.x` und `p.y` gibt. Die Lebensdauer der Objektvariablen entspricht der Lebensdauer des Objekts. Der Speicherbereich für die Objektvariablen wird reserviert, wenn das Objekt über den `new`-Operator erzeugt wird. Konzeptionell endet die Lebensdauer erst mit dem Ende der Programmausführung. Wir werden jedoch noch sehen, dass der Speicherbereich von Objekten, auf die kein Zugriff mehr möglich ist, auch schon früher vom System automatisch freigegeben werden kann.

Der Gültigkeitsbereich lokaler Variablen stimmt mit dem Sichtbarkeitsbereich überein. Dagegen kann der Sichtbarkeitsbereich von Objektvariablen gegenüber dem Gültigkeitsbereich eingeschränkt sein. Beispielsweise sind die beiden Objektvariablen `x` und `y` in Listing 3.6 mit dem Modifier `private` deklariert, der dafür sorgt, dass die Variablen außerhalb der Klasse `Punkt` nicht sichtbar und daher auch nicht zugreifbar sind, obwohl sie gültig sind, also in jeder Instanz von `Punkt` existieren.

Der Modifier `private` kann auf alle Deklarationen bzw. Definitionen innerhalb einer Klasse angewandt werden, insbesondere auch auf Methoden. Er sorgt dafür, dass die durch die Deklarationen und Definitionen eingeführten Namen nur innerhalb der Klasse sichtbar sind. Ähnlich

Listing 3.6: Datenabstraktion: Trennung der Innen- von der Außenansicht

```

public class Punkt {
    private double x = 0.0;
    private double y = 0.0;

    public void verschiebe(double deltaX, double deltaY) {
        x = x + deltaX;
        y = y + deltaY;
    }

    public double distanzZumUrsprung() {
        return Math.sqrt(x*x + y*y);
    }
}

public class PunktTester {
    public static void main(String[] args) {
        Punkt p;
        p = new Punkt();
        p.verschiebe(1.0,1.5);
        p.verschiebe(1.0,-1.5);

        System.out.println(p.distanzZumUrsprung());
    }
}

```

wie `private` kann man auch den Modifier `public` verwenden, der dafür sorgt, dass die durch die Deklarationen und Definitionen eingeführten Namen überall sichtbar sind, wo sie gültig sind; das ist im Wesentlichen das ganze Programm. Die beiden Methoden `verschiebe` und `distanzZumUrsprung` in Listing 3.6 sind daher überall sichtbar, und wir können entsprechende Nachrichten von überall aus an jedes Objekt vom Typ `Punkt` schicken.

In Listing 3.6 sind die beiden Klassen als `public` definiert, obwohl die Definitionen in keiner anderen Klasse stehen. Das ist ein Sonderfall. Derart definierte Klassen sind sowohl als Klassen als auch als Typen überall verwendbar. Im Gegensatz dazu wären als `private` definierte Klassen nur innerhalb der Datei sichtbar, in der die Definitionen stehen. Daher sind `private` Klassen nur sinnvoll, wenn sie zusammen mit anderen Klassen in einer Datei definiert sind. Umgekehrt darf es nicht mehrere `public` Klassen in derselben Datei geben.

Neben `public` und `private` gibt es noch zwei weitere Sichtbarkeitsvarianten. Die *Default-* oder *Paket-Sichtbarkeit* haben wir automatisch immer dann, wenn wir keinen Modifier angeben, um die Sichtbarkeit zu regeln. Dabei sind Namen innerhalb des aktuellen Paketes sichtbar, also innerhalb aller Dateien mit Java-Quellcode, die im selben Verzeichnis stehen. Für kleine Java-Programme kommen wir mit Paket-Sichtbarkeit als Ersatz für `public` aus, aber für größere Programme sowie Klassen, die in mehrere Programme eingebunden werden sollen, ist Paket-Sichtbarkeit eher selten und nur für spezielle Zwecke sinnvoll. Wir wollen uns von Anfang an einen Programmierstil angewöhnen, der auch für größere Projekte geeignet ist. Daher sollten wir als Programmieranfänger immer einen Modifier hinschreiben. Ausgenommen davon sind nur jene Fälle, in denen wir unbedingt Paket-Sichtbarkeit brauchen, und in denen wir unsere Entscheidung für Paket-Sichtbarkeit begründen können. Es sollte nicht passieren, dass wir einfach darauf vergessen, einen Modifier hinzuschreiben.

Die vierte Sichtbarkeitsvariante verwendet den Modifier `protected`. Diese Namen sind im selben Paket und in allen Klassen sichtbar, die von der Klasse, in der die Definitionen bzw. Deklarationen stehen, abgeleitet sind; damit werden wir uns später beschäftigen. Auch diese Sichtbarkeitsvariante wird nicht häufig verwendet.

Die Steuerung der Sichtbarkeit verhilft uns zu einer Trennung der Innen- und Außenansicht. Jedoch ist die Trennung nicht so gestaltet, dass manches nur innerhalb eines Objekts und anderes auch außerhalb sichtbar ist. Vielmehr kommt es auf die Klasse an, in welcher der Code steht, mit dem auf das Objekt zugegriffen wird. Um diesen Unterschied klar zu machen, fügen wir folgende Methode zur Klasse `Punkt` hinzu:

```

public double entfernung(Punkt p) {
    double dx = x - p.x;
    double dy = y - p.y;
    return Math.sqrt(dx*dx + dy*dy);
}

```

Da `entfernung` innerhalb von `Punkt` steht, dürfen wir in dieser Methode natürlich auf die Objektvariablen `x` und `y` zugreifen. Allerdings greifen wir auch auf die Objektvariablen `p.x` und `p.y` zu, die zu einem anderen Objekt gehören. Das ist erlaubt, weil `p` vom Typ `Punkt` ist: Innerhalb der Klasse `Punkt` dürfen wir auf alles zugreifen, das in der Klasse `Punkt` deklariert oder definiert wurde, auch auf `private` Inhalte. Wenn

`entfernung` nicht innerhalb von `Punkt` stehen würde, dürften wir weder auf `x` und `y` noch auf `p.x` und `p.y` zugreifen. Bezüglich Sichtbarkeit kommt es also nicht darauf an, zu welchem Objekt etwas gehört, sondern nur darauf, in welcher Klasse etwas steht. Diese Form der Steuerung der Sichtbarkeit gibt uns mehr Freiheiten und feinere Kontrolle. Allerdings kann diese Lösung gelegentlich zu Verwirrungen führen. Schließlich denken wir meist in Objekten und deren Innen- und Außenansichten.

In der Praxis sollen Objektvariablen fast immer als `private` deklariert sein. Aber auch für Methoden ist `private` oft sinnvoll. Ein Beispiel:

```
public double pfadlaenge(Punkt[] ps) {
    double sum = 0.0;
    for (int i=1; i < ps.length; i++)
        sum += ps[i-1].entfernung(ps[i]);
    return sum;
}
```

Die Methode steht in der Klasse `Punkt` und berechnet die Länge eines Pfades, der durch alle Punkte im Array `ps` läuft. Dazu verwendet sie die oben eingeführte Methode `entfernung`. Wenn wir `pfadlaenge` außerhalb von `Punkt` verwenden müssen, kann diese Methode nicht als `private` definiert sein; daher ist sie `public`. Wenn wir allerdings `entfernung` nur für die Berechnung von Pfadlängen brauchen und außerhalb von `Punkt` nicht aufrufen, sollten wir für `entfernung` den Modifier `private` verwenden, nicht `public`. Generell sollte so wenig wie möglich nach Außen sichtbar sein.

Programmieranfänger meinen oft, dass es besser wäre, vieles nach Außen sichtbar zu machen, damit man, falls das im Laufe der Programmentwicklung notwendig werden sollte, problemlos darauf zugreifen kann. Dahinter steckt jedoch ein grundlegend falscher Denkansatz. Die Trennung von Innen und Außen bewirkt vor allem, dass man nach Außen nicht sichtbare Implementierungsdetails einer Klasse jederzeit ändern kann. Wenn man vieles nach Außen sichtbar macht, geht dieser Vorteil verloren. Falls man als `private` definierte Methoden doch irgendwann außerhalb der Klasse verwenden muss, ist es in der Regel leicht, `private` durch `public` zu ersetzen. Dagegen ist es meist mit großem Aufwand verbunden, nach Außen sichtbare Methoden im Nachhinein zu ändern. Man muss lernen, Data Hiding nicht als Einschränkung zu betrachten, sondern als große Chance für zukünftige Änderungen und Erweiterungen.

Wenn man Lehrbücher aufschlägt, die in Java oder die objektorientierte Programmierung einführen, findet man als Beispiele häufig Methoden wie diese (als Teil der Klasse `Punkt`):

```
public void setX(double newX) { x = newX; }
public double getX() { return x; }
```

Es handelt sich um sogenannte *Setter-* bzw. *Getter-Methoden*, die nur eine Objektvariable setzen oder abfragen. Durch solche Methoden bekommt man Zugriff auf Objektvariablen, obwohl die Variablen selbst als `private` deklariert sind. Lehrbücher verwenden diese Beispiele schlicht wegen ihrer Einfachheit. In der Programmierpraxis sollte man Setter- und Getter-Methoden vermeiden, so gut es möglich ist, da durch sie die Vorteile des Data Hiding weitgehend verloren gehen.

Wie wir in Abschnitt 2.3.1 gesehen haben, verhindert Java, dass an einer Programmstelle mehrere lokale Variablen gleichen Namens gültig sind. Das gilt jedoch nur für lokale Variablen. Es ist in Java erlaubt, dass eine lokale Variable (oder ein formaler Parameter) denselben Namen wie eine Objektvariable hat. In diesem Fall wird die Objektvariable von der lokalen Variablen *verdeckt*. Das heißt, wenn man auf eine Variable dieses Namens zugreift, erhält man die lokale Variable, nicht die Objektvariable. Wie wir später sehen werden, gibt es Möglichkeiten, trotzdem auf die Objektvariable zuzugreifen. Dennoch ist es ratsam, Objektvariablen und lokale Variablen unterschiedlich zu benennen.

3.2.3 Identität und Gleichheit

Meistens existieren mehrere Objekte einer Klasse gleichzeitig. Durch folgende Programmzeilen werden zwei Objekte der Klasse `Punkt` erzeugt:

```
Punkt a1 = new Punkt();
Punkt a2 = new Punkt();

a1.verschiebe(1.0, 2.5);
a2.verschiebe(1.0, 2.5);
```

Das Ergebnis der Ausführung dieser Zeilen ist in Abbildung 3.7 veranschaulicht. Es gibt zwei Objekte mit den gleichen Koordinaten. Wir sagen, die beiden Objekte sind gleich, aber nicht identisch.

Dagegen wird durch folgende Zeilen nur ein Objekt erzeugt. Jedoch gibt es zwei Variablen, die dieses Objekt referenzieren.

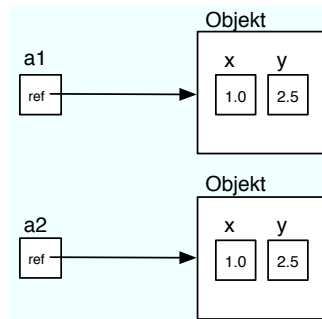


Abbildung 3.7: Zwei Punkte mit den gleichen Koordinaten

```
Punkt b1 = new Punkt();
Punkt b2 = b1;

b1.verschiebe(1.0, 2.5);
b2.verschiebe(1.0, 2.5);
```

Das Resultat dieser Anweisungen ist in Abbildung 3.8 veranschaulicht. Über beide Variablen sprechen wir *dasselbe* Objekt an. Daher wird durch die Anweisungen derselbe Punkt zwei mal verschoben. Wir sagen, die beiden durch **b1** und **b2** referenzierten Objekte² sind identisch oder kurz **b1** und **b2** sind identisch.

Die obigen Abbildungen veranschaulichen folgende Eigenschaften eines Objekts, die wir schon in Abschnitt 1.3.3 angesprochen haben:

Identität (identity): Ein Objekt ist durch seine unveränderliche Identität, die es bei seiner Erzeugung durch den `new`-Operator bekommt, eindeutig gekennzeichnet. Wenn man beispielsweise `new Punkt()` zweimal ausführt, bekommt man zwei Objekte mit unterschiedlichen Identitäten. Über seine Identität kann man das Objekt ansprechen, ihm also eine Nachricht schicken. Vereinfacht kann man sich die Identität als die Adresse des Objekts im Speicher vorstellen. Dies ist aber nur eine Vereinfachung, da die Identität erhalten bleibt, wenn sich die

²Meist sprechen wir im Plural von identischen Objekten, obwohl die Eigenschaft „identisch“ klar ausdrückt, dass wir es nur mit einem Objekt zu tun haben.

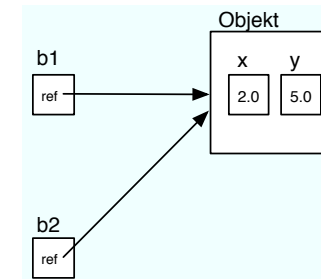


Abbildung 3.8: Dasselbe Objekt über mehrere Variablen angesprochen

Adresse ändert, zum Beispiel beim Verschieben des Objekts bei der Speicheroptimierung am Rande der Garbage Collection (das ist die Freigabe des Speichers nicht mehr zugreifbarer Objekte). Jedenfalls gilt: Wenn mehrere Referenzen gleichzeitig auf denselben Speicherplatz verweisen, dann sind die referenzierten Objekte *identisch*. Es handelt sich nur um ein Objekt – siehe Abbildung 3.8.

Zustand (state): Der Zustand setzt sich aus den Werten der Variablen im Objekt zusammen. Er ist in der Regel durch Zuweisung neuer Werte an die Objektvariablen änderbar. Zwei Objekte sind *gleich* (*equal*) wenn sie denselben Zustand (und dasselbe Verhalten ausgedrückt durch dieselbe Klasse) haben. Objekte können auch gleich sein, wenn sie nicht identisch sind – siehe Abbildung 3.7; dann sind sie *Kopien* voneinander. Zustände gleicher Objekte können sich unabhängig voneinander ändern; die Gleichheit geht dadurch verloren. Identität kann durch Zustandsänderungen nicht verloren gehen.

Verhalten (behavior): Das Verhalten eines Objekts beschreibt, wie sich das Objekt beim Empfang einer Nachricht verhält, das heißt, was das Objekt beim Aufruf der entsprechenden Methode macht. Das, was gemacht wird, hängt ab von

- der Nachricht, also dem Methodennamen zusammen mit den aktuellen Parametern,
- der aufgerufenen Methode (genauer: der Implementierung dieser Methode oder einer abstrakten Vorstellung davon)
- und dem Zustand des Objekts.

Zwei Objekte haben dasselbe Verhalten, wenn sie bei gleicher Nachricht und im gleichen Zustand das gleiche machen (und diese Eigenschaft für alle möglichen Nachrichten und Zustände gilt). Das Verhalten eines Objekts wird durch die Klasse des Objekts – konkret durch die Methodendefinitionen in der Klasse – festgelegt. Zwei Objekte derselben Klasse haben immer dasselbe Verhalten. Aber auch zwei Objekte unterschiedlicher Klassen können dasselbe Verhalten haben. Wie wir noch sehen werden, sind in der objektorientierten Programmierung auch Implikationen des Verhaltens von großer Bedeutung. Dabei verhält sich ein Objekt so wie ein anderes, solange man sich auf Nachrichten beschränkt, die das eine Objekt versteht; das andere Objekt könnte zusätzliche Nachrichten verstehen.

Die beiden relationalen Operatoren „==“ und „!=“ vergleichen zwei Werte elementarer Typen auf Gleichheit – siehe Abschnitt 2.2.2. Angewandt auf Objekte (also Instanzen von Referenztypen) vergleichen sie die Objekte jedoch auf Identität, nicht auf Gleichheit. Beispielsweise liefert `b1==b2` (Abbildung 3.8) als Ergebnis `true`, aber `a1==a2` (Abbildung 3.7) liefert `false`.

In jeder Klasse, die vom Java-System vorgegeben ist (beispielsweise `String`), finden wir die Methode `equal`, die ein Objekt dieser Klasse mit einem anderen Objekt auf Gleichheit vergleicht. Methoden für Vergleiche auf Gleichheit müssen wir uns in unseren eigenen Klassen selbst definieren – siehe Abschnitt 3.4.3.

Der Unterschied zwischen Gleichheit und Identität ist bedeutend, und die Verwechslung dieser beiden Begriffe stellt einen häufigen Anfängerfehler dar. Betrachten wir dazu ein Beispiel:

```
String s = "Ergebnis = " + (1 + 1);
String t = "Ergebnis = 2";
System.out.println(s == t); // false
System.out.println(s.equals(t)); // true
```

Die Zeichenkette in der Variablen `s` wird zur Laufzeit berechnet, während jene in `t` als Literal vorgegeben ist. Aufgrund ihrer unterschiedlichen Entstehung sind diese beiden Zeichenketten nicht identisch, aber gleich. Folglich liefert `s==t` als Ergebnis `false`, aber `s.equals(t)` ergibt `true`. Aus Unachtsamkeit oder Unwissenheit schreibt man leicht `s==t`, obwohl man die Zeichenketten auf Gleichheit vergleichen möchte, und wundert sich dann, warum sie nicht identisch sind. Gerade Zeichenketten sollte man immer mittels `equals` vergleichen, niemals mittels „==“.

Listing 3.9: Initialisierung durch Konstruktor

```
public class Punkt {
    private double x;
    private double y;

    public Punkt(double initX, double initY) {
        x = initX;
        y = initY;
    }

    ...
}
```

3.2.4 Kontext und Initialisierung

Neue Objekte werden durch den `new`-Operator erzeugt, der als Operanden die Klasse des zu erzeugenden Objekts nimmt. Beispielsweise erzeugt `new Punkt()` ein Objekt der Klasse `Punkt`. Das ist jedoch nur ein Teil der Wahrheit. Eine wichtige Frage ist die nach der Initialisierung der Objektvariablen. Die Initialisierung direkt bei der Deklaration wie in Listing 3.5 und 3.6 ist nur in sehr einfachen Fällen möglich, da die Klasse des zu erzeugenden Objekts keine Information über das Umfeld hat, in dem das Objekt verwendet werden soll. Dass jeder neu erzeugte Punkt auf dem Ursprung des Koordinatensystems liegen soll, ist eine durch nichts begründbare Annahme.

Dieses Problem wird durch *Konstrukturen* gelöst. Listing 3.9 definiert einen Konstruktor, der die Initialisierung mit Werten vornimmt, die als formale Parameter vorliegen. Ein neues Objekt dieser Variante von `Punkt` erzeugt man durch `new Punkt(1.3, 2.7)`, wenn der Punkt anfangs an den Koordinaten 1,3 und 2,7 liegen soll. Der Operand von `new` gibt also neben dem Typ des Objekts auch aktuelle Parameter an, die an den Konstruktor weitergeleitet werden. Das entspricht dem Aufruf eines Konstruktors ähnlich dem Aufruf einer Methode. Anders als Methoden werden Konstrukturen aber nur bei der Objekterzeugung aufgerufen, niemals in einem bereits initialisierten Objekt. Syntaktisch unterscheiden sich Konstrukturen von Methoden dadurch, dass kein Ergebnistyp spezifiziert ist und ihr Name dem der Klasse entspricht. Im Rumpf eines Konstruktors kann im Wesentlichen dasselbe stehen wie im Rumpf einer Methode.

Auch von Klassen ohne Konstruktor können Instanzen erzeugt werden. Ist kein Konstruktor definiert, wird automatisch folgender Default-Konstruktor ohne Parameter angenommen, der nichts macht:

```
public Klassenname() {}
```

Mit Hilfe des Default-Konstruktors ist es in allen Punkt-Varianten, außer jener in Listing 3.9, möglich, durch `new Punkt()` ein Objekt zu erzeugen. Die Punkt-Variante in Listing 3.9 erlaubt das nicht, weil der einzige verfügbare Konstruktor zwei Parameter hat.

Es kann auch mehrere Konstruktoren in einer Klasse geben. Beispielsweise könnten wir zu Listing 3.9 folgende Konstruktoren hinzufügen:

```
public Point() {                public Point(Point p) {
    x = 0.0;                    x = p.x;
    y = 0.0;                    y = p.y;
}
```

Damit liefert eine Ausführung von `new Punkt()` dasselbe Ergebnis wie eine von `new Punkt(0.0,0.0)`, und `new Punkt(p)` liefert eine Kopie eines Punktes `p`. Der Compiler entscheidet anhand der Parameterzahl sowie der Typen der Parameter im Operanden von `new`, welcher Konstruktor zu verwenden ist.

Generell spricht man von *Überladen*, wenn derselbe Name für Unterschiedliches steht und der Compiler anhand der Parameter eine Auswahl treffen muss. In Java können nicht nur Konstruktoren überladen sein, sondern auch Methoden. Es ist also möglich, dass mehrere Methoden denselben Namen haben, wenn sich die Methoden in der Anzahl oder den Typen der formalen Parameter voneinander unterscheiden.

Die beiden zu Listing 3.9 hinzugefügten Konstruktoren kann man etwas kürzer auch so definieren:

```
public Point() {                public Point(Point p) {
    this(0.0,0.0);              this(p.x,p.y);
}
```

In diesen Beispielen hat `this` eine ganz spezielle Bedeutung: Die Anweisung `this(...)` ruft einen anderen Konstruktor derselben Klasse auf. So ruft der parameterlose Konstruktor den Konstruktor mit zwei Parametern auf. Für diesen Aufruf ist spezielle Syntax nötig, da Konstruktoren ja nicht wie normale Methoden aufgerufen werden können. Eine Anweisung

der Form `this(...)` darf nur als allererste Anweisung im Rumpf eines Konstruktors verwendet werden, sonst nirgends.

Ungefähr dasselbe kann man statt mit `this(...)` auch durch Aufruf einer (im Idealfall privaten) Methode erreichen, welche die eigentliche Initialisierung vornimmt. Jeder Konstruktor ruft dieselbe Methode auf. Folgendes Beispiel (als Teil von `Punkt`) zeigt neben dieser Möglichkeit zur Initialisierung auch eine ganz andere Art der Verwendung von `this`:

```
public Point(double x, double y) {
    this.init(x,y);
}
private void init(double x, double y) {
    this.x = x;
    this.y = y;
}
```

Hier wird `this` als *Pseudovariablen* verwendet, also so etwas ähnliches wie eine Variable, an die man jedoch keinen Wert zuweisen kann. Der Inhalt von `this` steht für eine Referenz auf das Objekt, in dem man sich gerade befindet. Im Rumpf des Konstruktors wird die Nachricht `init(x,y)` an `this` geschickt, also an das neue Objekt, das gerade initialisiert wird. Statt `this.init(x,y)` kann man auch kurz `init(x,y)` schreiben, da eine Nachricht automatisch an das Objekt geschickt wird, in dem man sich gerade befindet, wenn kein anderer Empfänger angegeben ist. Das entspricht einem einfachen Methodenaufruf.

Sowohl im Konstruktor als auch in der Methode `init` haben die formalen Parameter dieselben Namen wie die Objektvariablen von `Punkt`, das heißt, die formalen Parameter verdecken die Objektvariablen³ – siehe Abschnitt 3.2.2. Die Namen `x` und `y` bezeichnen also die formalen Parameter, nicht die Objektvariablen. In der Methode `init` verwenden wir `this` um dennoch auf die Objektvariablen zuzugreifen; `this.x` und `this.y` bezeichnet die über `this` zugreifbaren Variablen, also die Objektvariablen.

Zur Laufzeit ist (mit einer Einschränkung, die wir bald näher betrachten werden) immer klar, in welchem Objekt man sich gerade befindet und welchen Inhalt `this` hat. Innerhalb eines Konstruktors ist es das Objekt, das gerade initialisiert wird. Innerhalb einer Methode ist es das Objekt, an

³Es wird ausdrücklich *nicht* empfohlen, Objektvariablen durch formale Parameter oder lokale Variablen zu verdecken. In diesem Beispiel machen wir das nur um zu demonstrieren, wie man dennoch auf Objektvariablen zugreifen kann, die auf diese Weise verdeckt sind.

welches die Nachricht geschickt wurde, die zur Ausführung der Methode geführt hat (also der Empfänger der Nachricht).

Die Pseudovariabel `this` verwendet man hauptsächlich dazu, eine Referenz auf das Objekt, in dem man sich gerade befindet, als aktuellen Parameter zu übergeben. Beispielsweise erzeugt die Anweisung

```
Punkt p = new Punkt(this);
```

(unter Verwendung des oben eingeführten Konstruktors mit einem Parameter) eine Kopie des Punktes, in dem man sich gerade befindet, und legt die Kopie in der Variablen `p` ab.

In Abschnitt 3.2.2 haben wir als Beispiel die Methode `pfadlaenge` definiert. Im Gegensatz zu den meisten Beispiel-Methoden in diesem Kapitel greift diese Methode nirgends auf das Objekt zu, in dem sich die Methode befindet. Weder wird auf eine Objektvariable zugegriffen noch eine Methode in `this` aufgerufen. Trotzdem muss diese Methode in der Klasse `Punkt` definiert sein, da Zugriff auf die (in einer Variante) private Methode `entfernung` nötig ist. Methoden wie `pfadlaenge` können mit dem Modifier `static` versehen werden um festzulegen, dass `this` nicht benötigt wird, etwa so:

```
public static double pfadlaenge(Punkt[] ps) {
    ...
}
```

In solchen *statischen* Methoden darf `this` nicht verwendet werden. Wir rufen statische Methoden auch nicht auf, indem wir eine Nachricht an ein Objekt der entsprechenden Klasse schicken, sondern indem wir eine Nachricht direkt an die Klasse schicken:

```
double l = Punkt.pfadlaenge(...);
```

Innerhalb der Klasse `Punkt` können wir den Klassennamen weglassen, sodass sich normale Methodenaufrufe ergeben, wie wir sie in Kapitel 2 mehrfach gesehen haben.

Statische Methoden dienen der Vereinfachung. Eine häufige Verwendung sollte man jedoch vermeiden, da dies auf einen veralteten prozeduralen Programmierstil hindeutet. In einem modernen objektorientierten Programmierstil muss man immer wieder auf `this` zugreifen, sodass sich statische Methoden nicht dafür eignen.

Für einen wichtigen Spezialfall brauchen wir unbedingt eine statische Methode, nämlich für `main`. Ohne diese Methode wäre es gar nicht möglich, ein Java-Programm auszuführen. Diese Methode muss (bis auf den Namen des formalen Parameters) immer genau so definiert sein:

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

Klassen, die diese Methode enthalten, sind ausführbar, das heißt, man kann `main` mittels `java Klassenname` in der übersetzten Klasse zur Ausführung bringen. Alle anderen Klassen sind nicht ausführbar.

Neben Methoden können auch Objektvariablen als statisch deklariert werden. Diese Variablen gehören dann zur Klasse selbst, nicht zu Instanzen der Klasse. Daher nennt man solche Variablen auch *Klassenvariablen*. Jede Klassenvariable existiert nur einmal pro Klasse, während Objektvariablen einmal pro Objekt existieren. Dieser Unterschied macht den Umgang mit Klassenvariablen schwierig und fehleranfällig. Außer für Spezialfälle sollte man Klassenvariablen daher nicht verwenden.

3.2.5 Konstanten

Es gibt jedoch eine wichtige Ausnahme von obiger Regel: Klassenvariablen, die als `static` und `final` deklariert sind, haben stets denselben Wert, sodass es keine Rolle spielt, ob sie zu einem Objekt oder zur Klasse gehören. Beispielsweise kann man Namen für konstante Werte auf diese Weise einführen:

```
public static final double PI = 3.14;
```

Da derart deklarierte Variablen während des gesamten Programmablaufs immer denselben Wert haben, sind sie eigentlich keine „Variablen“ im eigentlichen Sinne des Wortes mehr – die Werte können nicht variieren. Man spricht daher von *Konstanten*, zu denen man auch die Literale zählt. Zur Unterscheidung von normalen Variablen schreibt man deren Namen daher meist nur in Großbuchstaben. Von der Terminologie her werden Konstanten definiert – nicht nur deklariert – weil deren Werte ein für alle Mal festgelegt werden. Die Verwendung solcher Konstanten ist durchaus empfehlenswert, weil sie die Lesbarkeit von Programmen verbessert. Während man normale Variablen nicht als `public` deklarieren soll, sind `public` Konstantendefinitionen häufig sinnvoll. So wie in unserem Beispiel ist `PI` auch in der Klasse `Math` vordefiniert, jedoch mit maximaler Genauigkeit. Durch `Math.PI` können wir direkt auf den Wert zugreifen.

Listing 3.10: Beispiel eines Aufzähltyps

```

public class EnumExample {
    public enum Jahreszeiten { FRUEHLING, SOMMER, HERBST, WINTER;
        public int wert() {return this.ordinal() + 1;}
    }
    public static void main(String [] args) {
        System.out.println(Jahreszeiten.valueOf("SOMMER"));
        for (Jahreszeiten j: Jahreszeiten.values()) {
            System.out.println(j + " = " + j.wert());
        }
    }
}

```

Eine spezielle Form einer Klassendefinition ist die Definition eines Aufzähltyps (*Enum*). Aufzähltypen werden dort eingesetzt, wo eine kleine Menge von konstanten Objekten benötigt wird. Die Klassendefinition beginnt nach dem Schlüsselwort `enum` mit einer Auflistung konstanter Objekte gefolgt von der restlichen Klassendefinition.

Der Aufzähltyp `Jahreszeiten` in Listing 3.10 definiert eine Klasse mit den vier konstanten Instanzen `FRUEHLING`, `SOMMER`, `HERBST` und `WINTER`. Die vordefinierte Methode `values()` liefert alle Objekte eines Aufzähltyps, `valueOf(String)` zu einem String das dazugehörige Objekt und `ordinal()` die Ordnungszahl eines Objekts. Die Objekte sind in der Reihenfolge ihrer Definition mit 0 beginnend fortlaufend durchnummeriert. Dadurch kann einfach über alle Elemente eines Aufzähltyps iteriert werden. Das Programm erzeugt folgende Ausgabe:

```

SOMMER
FRUEHLING = 1
SOMMER = 2
HERBST = 3
WINTER = 4

```

Da Aufzähltypen nur konstante Objekte enthalten dürfen, gibt es einige Einschränkungen bei der Klassendefinition, auf die hier nicht näher eingegangen wird.

Generell ist eine häufige Verwendung von Aufzählungstypen oder vieler logisch zusammengehöriger Konstanten zu vermeiden, da sie in der Regel in Mehrfachverzweigungen eingesetzt werden. Wie wir in Abschnitt 3.3.2

sehen werden, entsprechen solche Mehrfachverzweigungen einem eher veralteten prozeduralen Programmierstil. Es ist vorteilhaft, stattdessen modernere objektorientierte Konzepte wie dynamisches Binden einzusetzen – siehe folgenden Abschnitt.

3.3 Interfaces und dynamisches Binden

In Abschnitt 3.1.3 haben wir erwähnt, dass jedes Objekt mehrere Schnittstellen haben kann. Jedoch ermöglicht eine Klassendefinition, wie wir sie in Abschnitt 3.2 eingeführt haben, zwar die Steuerung der Sichtbarkeit, aber keine eigenen Schnittstellen je nach Anforderungen. Dafür bietet Java *Interfaces* als eigenes Sprachkonzept. Über Interfaces kann man beliebig komplexe Schnittstellenstrukturen beschreiben und damit klare Trennungen zwischen Teilen eines objektorientierten Programms einführen. Zunächst betrachten wir, wie Interfaces spezifiziert werden und untersuchen dann *dynamisches Binden* als einen wesentlichen technischen Aspekt, den wir für den wirkungsvollen Einsatz von Interfaces unbedingt brauchen. Schließlich beschäftigen wir uns damit, welche Schnittstellenstrukturen günstig sind, und was wir damit erreichen können.

3.3.1 Interfaces zur Schnittstellenbeschreibung

Listing 3.11 und Listing 3.12 geben Beispiele für einfache Interfaces, die jeweils nur eine Methode spezifizieren. Auf den ersten Blick sieht eine Interface-Definition fast wie eine Klassendefinition aus, jedoch enthält sie im Wesentlichen nur Signaturen von Methoden, keine vollständigen Methoden. Zur Beschreibung von Schnittstellen braucht man nicht zu wissen, wie die Methoden genau definiert sind. Vorläufig reicht es zu wissen, dass Methoden mit einer bestimmten Signatur vorhanden sind. Alle Methoden in einem Interface werden als überall sichtbar (also `public`) und nicht-statisch angenommen. Weil das ohnehin klar ist, braucht man das Wort `public` nicht hinzuschreiben, und `static` ist natürlich verboten. Es ist schwierig bis unmöglich, alleine aus der Signatur einer Methode herauszulesen, wozu diese Methode dient. Daher verwenden wir Kommentare zur Beschreibung der Methoden.

Neben Methodensignaturen dürfen in Interfaces auch Konstanten wie `PI` definiert werden, also Variablen, die sowohl `static` als auch `final`

Listing 3.11: Interface eines verschiebbaren Objekts

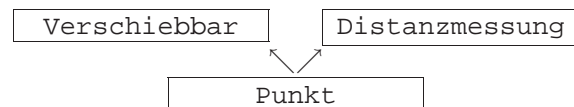
```
public interface Verschiebbar {
    // verschiebe Objekt um deltaX Einheiten nach rechts
    // und um deltaY Einheiten nach oben (für positive Parameter)
    // bzw. nach links und unten (für negative Parameter)
    void verschiebe(double deltaX, double deltaY);
}
```

Listing 3.12: Interface eines Objekts mit Messung der Distanz zum Ursprung

```
public interface Distanzmessung {
    // gib Distanz zum Ursprung zurück
    double distanzZumUrsprung();
}
```

(und implizit auch `public`) sind und direkt in der Deklaration initialisiert werden. Andere Variablen sind nicht erlaubt.

Listing 3.13 zeigt ein Beispiel für eine Klasse, die diese beiden Interfaces *implementiert*. Die implementierten Interfaces müssen in der Klassendefinition, voneinander durch Beistriche getrennt, nach dem Schlüsselwort `implements` angegeben werden. Der Compiler überprüft, ob es zu allen Signaturen in den Interfaces entsprechende Methodendefinitionen in der Klasse gibt. Die Klasse `Punkt` in Listing 3.13 muss aufgrund der Interfaces die Methoden `verschiebe` und `distanzZumUrsprung` als `public` und nicht `static` definieren. Daneben dürfen natürlich auch weitere Methoden und Variablen definiert bzw. deklariert sein. In einem Interface definierte Konstanten brauchen dagegen nicht noch einmal in der Klasse definiert werden; auf sie kann man ohne Weiteres lesend zugreifen. Folgendes Diagramm veranschaulicht die Beziehungen zwischen den Klassen und Interfaces entsprechend Listing 3.13:



Alle Methodensignaturen von `Punkt` lassen sich beispielsweise folgendermaßen in ein Interface packen:

Listing 3.13: Implementierung mehrerer Interfaces

```
public class Punkt implements Verschiebbar, Distanzmessung {
    private double x, y;

    public Punkt(double initX, initY) {
        x = initX;
        y = initY;
    }

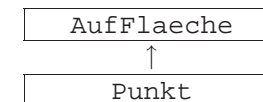
    public void verschiebe(double deltaX, double deltaY) {
        x = x + deltaX;
        y = y + deltaY;
    }

    public double distanzZumUrsprung() {
        return Math.sqrt(x*x + y*y);
    }

    public double entfernung(Punkt p) {
        double dx = x - p.x;
        double dy = y - p.y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

```
public interface AufFlaeche {
    void verschiebe(double dX, double dY);
    double distanzZumUrsprung();
    double entfernung(Punkt p);
}
```

Wenn `Punkt` statt `Verschiebbar` und `Distanzmessung` nur das Interface `AufFlaeche` implementiert, ergibt sich diese einfache Struktur:



In der Praxis wird man das jedoch nicht auf diese Weise machen, sondern so wie in Listing 3.14. Auf das Schlüsselwort `extends` folgen die Namen von Interfaces, deren Inhalte zusammen mit dem Inhalt der geschweiften Klammern das Interface bilden. Genauso wie obige Defi-

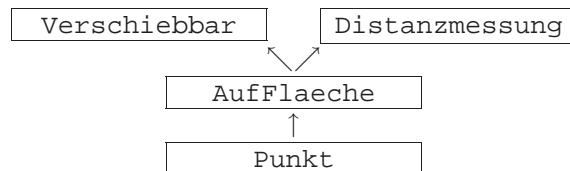
Listing 3.14: Interface eines Objekts im Koordinatensystem

```
public interface AufFlaeche extends Verschiebbar, Distanzmessung {
    // berechne Entfernung zum Punkt p
    double entfernung(Punkt p);
}
```

dition spezifiziert die Definition in Listing 3.14 drei Methoden. Zusätzlich haben wir aber auch mehr Struktur in die Interfaces hineingebracht: Wir wissen, dass `AufFlaeche` die beiden Interfaces `Verschiebbar` und `Distanzmessung` erweitert. Angenommen, `Punkt` sei statt wie in Listing 3.13 so wie hier implementiert:

```
public class Punkt implements AufFlaeche {
    ...
}
```

Dann implementiert `Punkt` nicht nur das Interface `AufFlaeche`, sondern auch die Interfaces `Verschiebbar` und `Distanzmessung`. Jedes Objekt vom Typ `Punkt` hat somit mindestens vier Schnittstellen:⁴ Drei Schnittstellen werden von den Interfaces gebildet. Die spezifischste Schnittstelle entspricht dem öffentlich sichtbaren Inhalt von `Punkt`; sie enthält neben den in `AufFlaeche` spezifizierten Methoden auch den Konstruktor. Folgendes Diagramm veranschaulicht die Schnittstellenstruktur:



Jede Schnittstelle entspricht einem Referenztyp. Ein Objekt mit mehreren Schnittstellen hat gleichzeitig auch ebensoviele Typen. Eine Variable vom Typ `Verschiebbar` kann auch ein Objekt vom Typ `Punkt` enthalten, weil jede Instanz von `Punkt` auch eine Instanz von `Verschiebbar` ist. Ebenso kann man an eine Methode, die einen formalen Parameter vom Typ `Distanzmessung` hat, einen aktuellen Parameter vom Typ

`AufFlaeche` übergeben, weil jede Instanz von `AufFlaeche` auch eine Instanz von `Distanzmessung` ist. Jedoch muss nicht jede Instanz von `Distanzmessung` eine Instanz von `Verschiebbar` sein. Falls es in obigem Diagramm einen Pfad (entlang der Pfeile) von einem tiefer stehenden zu einem höher stehenden Kästchen gibt, dann kann man eine Instanz des entsprechenden tiefer stehenden Typs in einer Variable des höher stehenden Typs ablegen.

Diese Eigenschaft, dass ein Objekt mehrere Typen haben kann, nennt man *Polymorphismus* (zu Deutsch etwa „Vielgestaltigkeit“). Von den Möglichkeiten des Polymorphismus, genauer gesagt vom *enthaltenden Polymorphismus* (weil jede Instanz eines Typs auch Instanz eines anderen Typs ist), auch *Untertyp-Polymorphismus* genannt, wird in der objektorientierten Programmierung ständig Gebrauch gemacht. Letzterer Begriff kommt daher, dass man jeden Typ U als *Untertyp* eines Typs T bezeichnet, wenn es im Diagramm von U aus (durch verfolgen der Pfeile) einen Pfad zu T gibt; in diesem Fall heißt T *Obertyp* von U . Beispielsweise ist `Punkt` ein Untertyp von `AufFlaeche`. Aber `Punkt` ist auch ein Untertyp von `Verschiebbar` genauso wie von `Distanzmessung`. Auch `AufFlaeche` ist ein Untertyp von `Verschiebbar` und ebenso von `Distanzmessung`. Umgekehrt ist `Verschiebbar` ein Obertyp von `Punkt` und `AufFlaeche`. Zur Vereinfachung des Umgangs mit Polymorphismus gilt zusätzlich, dass jeder Typ gleichzeitig Unter- bzw. Obertyp von sich selbst ist. So ist `Punkt` Untertyp und Obertyp von `Punkt`. Jede Instanz eines Untertyps des Typs einer Variablen kann in der Variable abgelegt werden.

Die Vorteile des Untertyp-Polymorphismus kommen erst zum Tragen, wenn es mehrere unterschiedliche Implementierungen derselben Schnittstelle gibt. Die Klasse `Scheibe` in Listing 3.15 implementiert ebenso wie `Punkt` das Interface `AufFlaeche`. Im Unterschied zu einem `Punkt` hat eine `Scheibe` eine Ausdehnung, die durch den Radius der runden Scheibe festgelegt ist. Die Methoden `distanzzumUrsprung` und `entfernung` berechnen den Abstand zwischen dem Ursprung bzw. vorgegebenen `Punkt` und dem nächsten `Punkt` auf der Scheibe. Sie liefern daher `0.0` zurück, wenn der Ursprung oder gegebene `Punkt` auf der Scheibe liegt. Jedenfalls machen die beiden Methoden in `Scheibe` im Detail etwas anderes als die entsprechenden Methoden in `Punkt`. Zusammen mit `Scheibe` ergibt sich folgendes *Typdiagramm* – so bezeichnet man ein Diagramm von Untertypbeziehungen:

⁴In Abschnitt 3.4 werden wir sehen, dass jede Klasse zumindest eine weitere Schnittstelle hat, die der Schnittstelle der vorgegebenen Klasse `Object` entspricht.

Listing 3.15: Weitere Implementierung von Interfaces

```

public class Scheibe implements AufFlaeche {
    private double x, y, r;

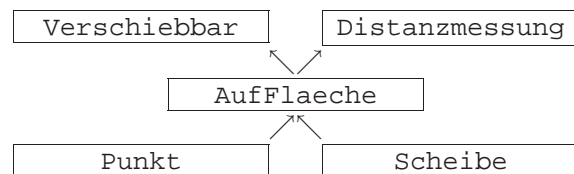
    public Scheibe(double initX, double initY, double radius) {
        x = initX;
        y = initY;
        r = Math.abs(radius); // darf nicht negativ sein
    }

    public void verschiebe(double deltaX, double deltaY) {
        x = x + deltaX;
        y = y + deltaY;
    }

    public double distanzZumUrsprung() {
        return this.entfernung(new Punkt(0.0, 0.0));
    }

    public double entfernung(Punkt p) {
        Punkt q = new Punkt(x,y);
        double d = q.entfernung(p);
        return Math.max(d - r, 0.0);
    }
}

```



Listing 3.16 zeigt in einem Beispiel die Verwendung von Polymorphismus. Als Parameter *v* der Methode `bewege` wird ein Objekt vom Typ `Verschiebbar` erwartet. Mehr braucht man an dieser Stelle nicht über *v* zu wissen, da nur die in `Verschiebbar` spezifizierte Methode aufgerufen wird. Als aktueller Parameter kann bei einem Aufruf von `bewege` jedes beliebige Objekt vom Typ `Verschiebbar` verwendet werden, beispielsweise vom Typ `Punkt` oder `Scheibe`, aber auch von jeder anderen Implementierung des Interfaces `Verschiebbar`, die nicht bekannt zu sein braucht. Analog dazu ist der formale Parameter von `gibDistanzAus`

Listing 3.16: Beispiel für Verwendung von Interfaces

```

public class FlaechenTester {
    private static void bewege(Verschiebbar v) {
        v.verschiebe(1.5,2.5);
    }

    private static void gibDistanzAus(Distanzmessung d) {
        System.out.println(d.distanzZumUrsprung());
    }

    public static void teste(AufFlaeche f) {
        for (int i=0; i < 5; i++) {
            bewege(f);
            gibDistanzAus(f);
        }
    }

    public static void main(String[] args) {
        teste(new Punkt(0.0, 0.0));
        teste(new Scheibe(0.0, 0.0, 2.1));
    }
}

```

vom Typ `Distanzmessung`, da nichts über dieses Objekt bekannt zu sein braucht, als dass es eine Methode `distanzZumUrsprung` gibt. Die Methode `teste` benötigt einen Parameter *f* vom Typ `AufFlaeche`, da sowohl `bewege` als auch `gibDistanzAus` mit *f* als Argument aufgerufen werden und `AufFlaeche` Untertyp von `Verschiebbar` und `gibDistanzAus` ist. In `main` wird `teste` für ein Objekt vom Typ `Punkt` und eines vom Typ `Scheibe` aufgerufen. Wir brauchen also nur eine Methode um Berechnungen auf Objekten unterschiedlicher Typen durchführen zu können. Das reduziert den Programmieraufwand. Wie wir später noch sehen werden, wird dadurch auch die Wartung vereinfacht.

3.3.2 Dynamisches Binden

Bei genauer Betrachtung der Methode `gibDistanzAus` in Listing 3.16 stellt sich die Frage, welche Methode durch `v.distanzZumUrsprung()` überhaupt aufgerufen wird – die Methode, die in `Punkt` definiert ist, oder jene in `Scheibe`, oder vielleicht sogar eine ganz andere. Eine eindeutige Antwort darauf gibt es nicht. Nach einem Aufruf von `gibDistanzAus`

kann die eine Methode namens `distanzZumUrsprung` ausgeführt werden, beim nächsten Aufruf eine andere desselben Namens. Das hängt vom Objekt ab, für das der formale Parameter `d` steht. Tatsächlich ist diese Unklarheit ein Grund dafür, warum wir in der Regel vom Senden einer Nachricht an ein Objekt sprechen und nicht vom Aufruf einer Methode. Wir kennen die aufzurufende Methode im Allgemeinen ja gar nicht. Die Entscheidung darüber, welche Methode ausgeführt wird, trifft das Objekt, an das wir die Nachricht schicken. Wenn das Objekt durch `new Punkt(...)` erzeugt wurde, wird `distanzZumUrsprung` so wie in `Punkt` definiert ausgeführt, wenn es durch `new Scheibe(...)` erzeugt wurde, dann so wie in `Scheibe` definiert.

Wie in diesem Beispiel kommt es oft vor, dass die auszuführende Methode nicht schon vom Compiler bestimmt wird und immer gleich ist, sondern vom Empfänger der Nachricht abhängt, der jedesmal anders sein kann. Man spricht von *dynamischem Binden*, wenn die auszuführende Methode wie im Beispiel erst zur Laufzeit bestimmt wird, und von *statischem Binden*, wenn bereits der Compiler die auszuführende Methode kennt.

Referenzvariablen (und formale Parameter) haben in objektorientierten Sprachen mehr als nur einen Typ. Das, was wir bis jetzt kurz als Typ einer Variablen bezeichnet haben, ist eigentlich der *deklarierte Typ* der Variablen, also der Typ, der in der Variablendeklaration angegeben ist. Daneben gibt es auch den *dynamischen Typ* der Variablen (umgangssprachlich oft auch *echter* oder *tatsächlicher* Typ genannt), der vom Variableninhalt abhängt: Der dynamische Typ entspricht der Klasse, als deren Instanz das Objekt in der Variablen erzeugt wurde. Beispielsweise ist der deklarierte Typ des formalen Parameters `d` von `gibDistanzAus` stets `Distanzmessung`. Wenn das Objekt in `d` durch `new Punkt(...)` erzeugt wurde, dann ist `Punkt` der dynamische Typ von `d`, und wenn das Objekt durch `new Scheibe(...)` erzeugt wurde, dann `Scheibe`. Während der deklarierte Typ einer Variablen immer gleich bleibt, kann sich der dynamische Typ im Laufe der Zeit ändern. Für dynamisches Binden wird immer der dynamische Typ herangezogen.

Listing 3.17 zeigt ein Beispiel, wie dynamisches Binden eingesetzt werden kann. Das Interface `Beurteilung` spezifiziert zwei Methoden, die in drei Untertypen von `Beurteilung` implementiert werden. Die Methode `test` in `Test1` verwendet dynamisches Binden, indem Nachrichten an den formalen Parameter `b` geschickt werden.

Listing 3.17: Beispiel für Verwendung von dynamischem Binden – empfohlen

```
public interface Beurteilung {
    boolean bestanden();
    String toString();
}

public class Auszeichnung implements Beurteilung {
    public boolean bestanden() {
        return true;
    }
    public String toString() {
        return "mit Auszeichnung bestanden";
    }
}

public class Bestanden implements Beurteilung {
    public boolean bestanden() {
        return true;
    }
    public String toString() {
        return "bestanden";
    }
}

public class NichtBestanden implements Beurteilung {
    public boolean bestanden() {
        return false;
    }
    public String toString() {
        return "nicht bestanden";
    }
}

public class Test1 {
    public static String test (Beurteilung b) {
        String s = "Prüfung " + b.toString() + "! ";
        if (b.bestanden())
            return s + "Herzlichen Glückwunsch!";
        else
            return s + "Vielleicht das nächste Mal.";
    }
}
```

Jede Aufgabe ist statt durch dynamisches Binden auch mittels bedingter Anweisungen (vor allem `switch`-Anweisungen) lösbar. Listing 3.18 zeigt

Listing 3.18: Vermeidung von dynamischem Binden – nicht empfohlen

```

public class Test2 {
    public static final int AUSZEICHNUNG    = 1;
    public static final int BESTANDEN      = 2;
    public static final int NICHT_BESTANDEN = 5;

    public static String test (int b) {
        String s = "Prüfung ";
        String e = "Da ist etwas flasch gelaufen.";
        switch (b) {
            case AUSZEICHNUNG:
                s += "mit Auszeichnung ";
            case BESTANDEN:
                s += "bestanden";
                e = "Herzlichen Glückwunsch!";
                break;
            case NICHT_BESTANDEN:
                s += "nicht bestanden";
                e = "Vielleicht das nächste Mal.";
                break;
        }
        return s + "! " + e;
    }
}

```

eine kürzere Lösung der Aufgabe, die auch von `Test1` in Listing 3.17 gelöst wird. Mit wenig Programmiererfahrung könnte man meinen, dass die Lösung in Listing 3.18 besser wäre als jene in Listing 3.17. Das ist jedoch ganz sicher nicht der Fall. Einerseits sind das Interface und die Klassen, die das Interface implementieren, zur allgemeinen Verwendung gedacht, nicht nur als Hilfsmittel zur Erstellung der Testklasse. Die Methode `test` in Listing 3.17 ist deutlich kürzer als jene in Listing 3.18, weil die Hauptaufgaben von den anderen Klassen übernommen werden. Wenn man das Beispiel erweitert, kann man erwarten, dass ein Großteil des Codes, der zu Listing 3.17 hinzukommt, kürzer ist als jener, der zu Listing 3.18 hinzukommt, weil im ersten Fall eher auf die bereits bestehenden Klassen zurückgegriffen werden kann. Der Vorteil der Kürze von Listing 3.18 kann sich im Laufe der Entwicklung des Programms in das Gegenteil verkehren. Andererseits ist die Lösung in Listing 3.18 auch recht anfällig für Fehler:

- Es wird eine kaum sinnvolle Zeichenkette zurückgegeben, wenn man `test` mit einem anderen Argument als 1, 2 oder 5 aufruft. Auch

wenn man durch einen `default`-Zweig diese Fälle abfangen würde, wüsste man noch immer nicht, welche besser geeignete Zeichenkette man zurückgeben könnte. Dieses Problem besteht bei einer Lösung mittels dynamischem Binden nicht, da bereits der Compiler sicherstellt, dass `test` nur mit Instanzen von `Beurteilung` aufgerufen wird. Allerdings muss man sicherstellen, dass nicht `null` als Argument übergeben wird.

- Der Code in Listing 3.18 ist sehr kompakt, weil viele Möglichkeiten zum Sparen von Codezeilen ausgenutzt wurden. Etwa werden die beiden Fälle von `AUSZEICHNUNG` und `BESTANDEN` zum Großteil durch gemeinsamen Code abgearbeitet. Leider geht diese Sparsamkeit auf Kosten der Lesbarkeit. Man muss davon ausgehen, dass bei einer künftigen Programmänderung Teile des Programms falsch verstanden und dadurch auf unbeabsichtigte Weise geändert werden. Jede einzelne Klasse in Listing 3.17 ist dagegen viel einfacher verständlich und damit weniger fehleranfällig.
- Mehrere Aspekte des Programms sind in Listing 3.18 ineinander verwoben. So kümmert sich dieselbe `switch`-Anweisung um die Bezeichnung der Beurteilung (`s`) und den Text am Ende (`e`). Derartige ist problematisch, nicht nur weil es die Lesbarkeit erschwert, sondern auch weil der gemeinsame Kontrollfluss für beide Aspekte durch Programmänderungen leicht verloren geht und kleine Erweiterungen daher mit hoher Wahrscheinlichkeit große Änderungen nach sich ziehen. Das Interface muss dagegen eine logische Struktur vorgeben, sodass derartige Verwebungen verschiedener Aspekte besser verhindert werden. Es ist einfach, das Programm auf verschiedene Weisen zu erweitern, beispielsweise durch Hinzufügen einer Klasse `NichtBeurteilt`.

Derartige Probleme sind nicht typisch für das Beispiel, sondern treten ganz allgemein zusammen mit (vor allem mehrfachen) Programmverzweigungen häufig auf. Zusammengefasst gilt, dass dynamisches Binden fast immer besser ist als die Verwendung von `switch`-Anweisungen oder ineinander geschachtelten `if`-Anweisungen. Bevor man solche Anweisungen schreibt, sollte man sich überlegen, wie man die Aufgabe stattdessen durch dynamisches Binden lösen kann. Dynamisches Binden sorgt ebenso wie die dafür vorgesehenen Anweisungen für mehrfache Programmverzweigungen. Es verbessert durch die Art und Weise, wie das Programm dabei in Klassen

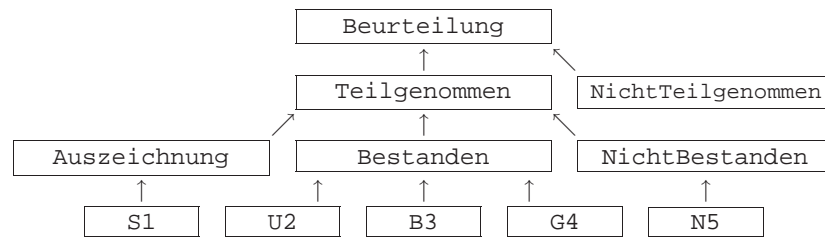


Abbildung 3.19: Typdiagramm mit Spezialisierungen von Beurteilungen

und Interfaces aufgeteilt wird, darüber hinaus meist auch die Programmstruktur und Wartbarkeit.

Dynamisches Binden ist aufgrund der Mehrfachverzweigung bei der Ausführung um eine Winzigkeit weniger effizient als statisches Binden. Manchmal ist man deswegen versucht, auf dynamisches Binden zu verzichten. Allerdings ist dynamisches Binden etwas effizienter als die Ausführung einer `switch`-Anweisung. Wenn man also statisches Binden durch eine bedingte Anweisung erkaufte, hat man in der Summe wahrscheinlich an Effizienz eingebüßt. Effizienzüberlegungen soll man hinsichtlich dynamischem Binden also prinzipiell beiseite lassen.

3.3.3 Spezialisierung und Ersetzbarkeit

In Abschnitt 3.3.1 haben wir gesehen, wie man durch die Verwendung von Interfaces Untertypbeziehungen einführen und Typdiagramme aufbauen kann. Typdiagramme geben die Struktur eines Programms vor und sind daher von entscheidender Bedeutung. In der Programmierpraxis müssen wir ständig Typdiagramme aufbauen und uns dabei für bestimmte Strukturen und gegen andere Strukturen entscheiden. Wir benötigen Hilfsmittel um zu erkennen, wie gut ein bestimmtes Typdiagramm für ein Programm geeignet ist. Solche Hilfsmittel wollen wir hier kurz ansprechen.

Zwei eng miteinander verbundene Hilfsmittel bestehen im Verstehen von Typdiagrammen als *Spezialisierungen* und in der Verwendung von *Analogien zur realen Welt*.

Zur Erklärung verwenden wir das Typdiagramm in Abbildung 3.19. Es ist leicht zu sehen, dass die Begriffe, die hier verwendet werden, aus der realen Welt kommen; sie können auf Zeugnissen und Teilnahmebestätigun-

gen vorkommen. Die Abkürzungen S1 bis N5 sind an der TU Wien übliche Kürzel für „sehr gut“ bis „nicht genügend“. Wegen der Analogie zur realen Welt ist auch ohne Erklärungen offensichtlich, wofür diese Begriffe stehen. Auch die Beziehungen zwischen den Begriffen sind offensichtlich: Es zeigt nur dann ein Pfeil von einem Untertyp U zu einem Obertyp T , wenn U eine spezielle Variante von T ist; U ist also eine Spezialisierung von T . Im Beispiel ist *Beurteilung* der allgemeinste Begriff, der für viele unterschiedliche Beurteilungen stehen kann. Etwas spezieller ist *Teilgenommen* (im Unterschied zu *NichtTeilgenommen*), und *Bestanden* ist noch spezieller. Eine Unterscheidung zwischen noch spezielleren Beurteilungen als konkreten Noten, etwa B3, brauchen wir im Beispiel nicht. Von oben nach unten werden die Begriffe im Diagramm immer spezieller. Manchmal spricht man auch von *ist-ein-Beziehungen*. Beispielsweise *ist B3 eine Beurteilung der Art Bestanden*, und *Bestanden ist eine Beurteilung der Art Teilgenommen*. Ganz allgemein beschreibt ein Typdiagramm Spezialisierungen von Typen. Daneben muss auch gelten, dass Untertypen zumindest alle Signaturen von Methoden spezifizieren müssen, die auch von den Obertypen spezifiziert werden.

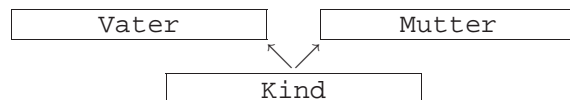
Zumindest zu Beginn der Entwurfs-Phase (siehe Abschnitt 1.6.1) kennt man die von den Typen spezifizierten Methoden noch nicht. Trotzdem kann man schon ein Typdiagramm erstellen, alleine aufgrund von Spezialisierungen in der Analogie zur realen Welt. Wenn man dabei umsichtig vorgeht, passen die Signaturen der Methoden später wahrscheinlich zusammen. Falls nicht, hat man etwas übersehen und muss die Strukturen anpassen. Obwohl das passieren kann, bilden Spezialisierungen zusammen mit Analogien zu realen Welt ein wichtiges Denkmuster, an das wir uns bei der Entwicklung von Typdiagrammen stets halten sollen.

Es passiert leicht, dass man zu Beginn zu detailreiche Typdiagramme zeichnet. Möglicherweise wurde obiges Typdiagramm als Grundlage zur Erstellung von Listing 3.17 verwendet. Allerdings wurden beim Schreiben des Programmcodes viele der Typen im Diagramm weggelassen, weil sie nicht gebraucht werden. Vielleicht stellt der Code in Listing 3.17 auch nur eine Zwischenphase dar, und es kommen später weitere Typen hinzu. Es könnte auch beides gleichzeitig zutreffen: Derzeit brauchen wir nur einen kleinen Teil der Typen, später könnten weitere Typen notwendig werden. Ein auf die reale Welt und Spezialisierungen konzentrierter Entwurf lässt sich oft relativ einfach erweitern, da man von Anfang an alles aus einem weiteren Blickwinkel betrachtet, als dies zur Lösung der Aufgabe unbedingt notwendig wäre.

Ein weiterer wichtiger Begriff im Zusammenhang mit Spezialisierungen ist das *Prinzip der Ersetzbarkeit*: Ein Typ U soll genau dann Untertyp eines Typs T sein, wenn Objekte vom Typ U überall verwendbar sind, wo Objekte vom Typ T erwartet werden. Das Ersetzbarkeitsprinzip ist eng mit Spezialisierungen verknüpft. Wenn U einen Spezialfall von T darstellt, dann ist jedes Objekt vom Typ U auch ein Objekt vom Typ T , und dann kann man natürlich auch jedes Objekt vom Typ U überall dort verwenden, wo eines vom Typ T erwartet wird. Im Vergleich zur Spezialisierung drückt das Ersetzbarkeitsprinzip aber etwas genauer das Wesentliche aus. Wenn nicht klar ist, ob U als Spezialisierung von T angesehen werden soll, dann gibt das Ersetzbarkeitsprinzip eine klare Antwort darauf.

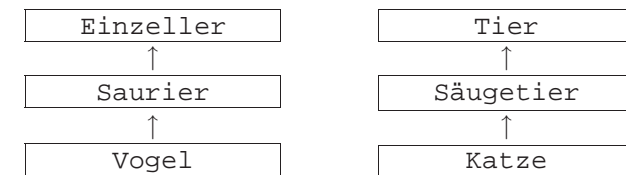
Das Ersetzbarkeitsprinzip drückt ganz direkt aus, worauf es bei Untertypbeziehungen ankommt: Wenn ein Programmstück für eine Instanz eines Obertyps funktioniert, dann wird dieses Programmstück auch für eine Instanz eines Untertyps funktionieren. Bei der Erstellung des Programmstücks denken wir nur an den Obertyp und brauchen nicht zu wissen, welche Untertypen es gibt. Aber bei der Einführung der Untertypen müssen wir uns vergewissern, dass das Ersetzbarkeitsprinzip erfüllt ist. Insgesamt passen die Typen dann zusammen. Beispielsweise schreiben wir die Methode `gibDistanzAus` in Listing 3.16 ohne Wissen über die Klassen, die das Interface `Distanzmessung` implementieren. Aber beim Schreiben von Klassen wie `Punkt` und `Scheibe` müssen wir nicht nur ein entsprechendes Interface implementieren, sondern auch dafür sorgen, dass die in `Distanzmessung` spezifizierte Methode `distanzZumUrsprung` tatsächlich so funktioniert wie im Interface beschrieben – siehe Listing 3.12. Die Methode muss also die Distanz bis zum Ursprung zurückgeben.

Die in einem Typpdiagramm beschriebene Struktur nennt man auch *Typhierarchie*. Dieser Name ist nicht ganz optimal gewählt, weil es sich dabei nicht unbedingt um eine hierarchische Struktur handeln muss, bei der ein Typ an der Spitze steht und sich die Struktur nach unten immer weiter verzweigt. In Abschnitt 3.3.1 haben wir Beispiele für nicht-hierarchische Strukturen gesehen. Manchmal spricht man statt von einer Typhierarchie auch von einer *Vererbungshierarchie*. Dieser Name kommt von der Vererbung in objektorientierten Sprachen, die wir in Abschnitt 3.4 kennenlernen werden. Leider wird dieser Begriff häufig falsch verstanden:



Dieses Diagramm zeigt keine Typ- oder Vererbungshierarchie, weil `Kind` keine Spezialisierung von `Mutter` und `Vater` sein kann. Der Begriff Vererbung hat in diesem Zusammenhang nichts damit zu tun, dass Kinder von ihren Eltern abstammen. Ein Objekt vom Typ `Kind` ist, entsprechend den Erfahrungen aus der realen Welt, eine Person, die nicht gleichzeitig Mutter und Vater von sich selbst sein kann. Von einer Spezialisierung und vom Ersetzbarkeitsprinzip würde man sich das erwarten. Solche Diagramme setzen immer Typen zueinander in Beziehung, keine Objekte.

Folgende beiden Diagramme sollen noch klarer machen, was eine Typ- oder Vererbungshierarchie sein kann und was nicht:



Die Hierarchie auf der linken Seite stellt die evolutionäre Entwicklung vom Einzeller über den Saurier bis zum Vogel dar. Das ist *keine* Typ- oder Vererbungshierarchie, weil ein Vogel nicht gleichzeitig Saurier und Einzeller sein kann. Es spielt keine Rolle, dass in einem Vogel noch Erbgut eines Sauriers und Einzellers steckt. Ein Programmstück, das mit allen Einzellern umgehen kann, ist von Sauriern und Vögeln eventuell überfordert. Die Hierarchie auf der rechten Seite stellt dagegen schon eine Typ- oder Vererbungshierarchie dar, da bekanntlich jede Katze ein Säugetier und jedes Säugetier ein Tier ist. Ein Programmstück, das mit allen Tieren umgehen kann, kann sicher auch mit allen Säugetieren und Katzen umgehen.

3.4 Vererbung

Eines der bekanntesten und zugleich am häufigsten missverstandenen Konzepte der objektorientierten Programmierung ist die Vererbung. Dabei wird eine Klasse aus einer anderen Klasse abgeleitet. Auch mittels Vererbung lassen sich Typhierarchien ähnlich denen aufbauen, die wir zusammen mit Interfaces in Abschnitt 3.3 gesehen haben. Zusätzlich eignet sich die Vererbung dazu, in einer Klasse definierte Methoden in eine andere Klasse zu übernehmen, also Methoden zu erben. Es braucht viel Programmiererfahrung, um den Zwiespalt zu meistern, der sich häufig zwischen dem Aufbau guter Typhierarchien und dem Erben von Methoden ergibt.

Listing 3.20: B3 als eine von Bestanden abgeleitete Klasse

```

public class Bestanden implements Beurteilung {
    public boolean bestanden() {
        return true;
    }
    public String toString() {
        return "bestanden";
    }
}

public class B3 extends Bestanden {
    public String toString() {
        return "mit \"befriedigend\" bestanden";
    }
    public int toInt() {
        return 3;
    }
}

```

3.4.1 Ableitung von Klassen

Vererbung ist anhand eines Beispiels einfach zu verstehen. In Listing 3.20 wird die Klasse B3 von Bestanden abgeleitet, wobei Beurteilung aus Listing 3.17 übernommen ist. Die Klasse Bestanden definiert⁵ die beiden Methoden bestanden und toString. Aufgrund von Vererbung durch die Klausel extends Bestanden erbt B3 diese beiden Methoden. Jedoch definiert B3 selbst eine Methode toString mit derselben Signatur wie in Bestanden. Die von B3 aus Bestanden ererbte Methode toString wird durch die in B3 definierte gleichnamige Methode überschrieben. Zusätzlich definiert B3 die Methode toInt, die in Bestanden nicht vorkommt. Jede Instanz der Klasse B3 hat die Methoden toString und toInt so wie in B3 definiert und zusätzlich die Methode bestanden so wie in Bestanden definiert. Weiters gilt, dass B3 ein Untertyp von Bestanden ist. Das ist möglich, weil die abgeleitete Klasse (man nennt sie auch *Unterklasse*) durch das Erben der Methoden der *Basisklasse* (also der Klasse, von der abgeleitet wird, auch *Oberklasse* genannt) zumindest

⁵Man sagt auch, Bestanden *implementiert* die beiden Methoden, um zu betonen, dass Anweisungen im Rumpf der Methoden vorhanden sind, die Methoden also tatsächlich definiert und nicht nur deklariert sind.

Listing 3.21: Eine von Scheibe – siehe Listing 3.15 – abgeleitete Klasse

```

public class FarbigeScheibe extends Scheibe {
    private long r; // Farbe der Scheibe als RGB-Code

    public FarbigeScheibe (double initX, double initY,
                          double radius, long farbeRGB) {
        super(initX, initY, radius);
        r = farbeRGB;
    }

    public long farbe() { return r; } // OK

    // public double entfernung(Punkt p) {
    //     return (new Punkt(x,y)).entfernung(p);
    // } // Syntaxfehler: x und y nicht sichtbar !!
}

```

alle Methoden spezifiziert, die in der Oberklasse spezifiziert sind. In dieser Hinsicht ähnelt die Ableitung von Klassen der Implementierung von Interfaces (abgesehen davon, dass jede Klasse nur von *einer* anderen Klasse abgeleitet wird). Man kann B3 als Teil des Typdiagramms in Abbildung 3.19 verstehen, und S1 bis N5 können analog dazu definiert sein, wobei S1 von Auszeichnung und N5 von NichtBestanden abgeleitet ist.

Das Vererbungsbeispiel in Listing 3.21 zeigt den Umgang mit Objektvariablen und Konstruktoren. Die Klasse FarbigeScheibe erbt die Variablen und Methoden von Scheibe. Zur Veranschaulichung deklariert FarbigeScheibe eine Objektvariable r, die genau so heißt wie eine Variable in Scheibe. Aus zwei Gründen ergibt sich daraus kein Konflikt: Einerseits ist r in Scheibe als private deklariert (so wie auch r in FarbigeScheibe, aber darauf kommt es hier nicht an) und daher außerhalb von Scheibe nicht sichtbar. In FarbigeScheibe sind x, y und r aus Scheibe nicht zugreifbar. Andererseits kann man in Untertypen Variablen deklarieren, obwohl sie dieselben Namen wie aus der Oberklasse ererbte und sichtbare Variablen haben; in diesem Fall verdecken die in der Unterklasse deklarierten Variablen jene der Oberklasse. Jede Verwendung von r in FarbigeScheibe bezieht sich auf die in FarbigeScheibe deklarierte Variable. Da x und y nicht sichtbar sind, kann entfernung nicht (wie in Listing 3.21 auskommentiert) überschrieben werden.

Um `entfernung` durch Weglassen von `//` aus Listing 3.21 zu überschreiben, müssten wir `x` und `y` in `FarbigeScheibe` sichtbar machen, indem wir die Variablen in `Scheibe` so deklarieren:

```
protected double x, y, r;
```

Als `protected` deklarierte Variablen sind in abgeleiteten Klassen sichtbar. Allerdings stellt der direkte Zugriff auf Variablen, die in der Oberklasse deklariert wurden, einen schlechten Programmierstil dar und soll daher nach Möglichkeit vermieden werden.⁶

Konstruktoren dienen hauptsächlich der Initialisierung von Variablen. Dafür ist es natürlich notwendig, dass die Konstrukturen Zugriff auf die Variablen haben, auch auf die privaten. Wegen der Einschränkungen der Sichtbarkeit kann der Konstruktor in `FarbigeScheibe` die von der Oberklasse geerbten Variablen gar nicht selbst initialisieren. Die spezielle Anweisung `super(...)` in der ersten Zeile im Konstruktor ruft zum Zwecke der Initialisierung den Konstruktor der Oberklasse auf. Eine solche Anweisung kann so wie und anstatt von `this(...)` – siehe Abschnitt 3.2.4 – nur als erste Anweisung im Rumpf eines Konstruktors vorkommen. In jedem Fall wird zuerst der Konstruktor der Oberklasse ausgeführt, und erst dann sind die anderen Anweisungen im Rumpf des Konstruktors ausführbar. Falls ein Konstruktor weder mit `super(...)` noch mit `this(...)` beginnt, wird implizit `super()` aufgerufen, bevor der eigentliche Konstruktor ausgeführt wird. So ruft der in B3 in Listing 3.20 implizit vorhandene Default-Konstruktor `B3()` den ebenfalls implizit vorhandenen Default-Konstruktor `Bestanden()` der Oberklasse auf.⁷ Falls ein Konstruktor mit `this(...)` beginnt, wird ein anderer Konstruktor derselben Klasse ausgeführt, der zu Beginn ebenfalls einen Konstruktor der Oberklasse ausführt. Jede Instanz einer Klasse wird daher immer schrittweise von oben nach unten in der Typhierarchie initialisiert.

⁶Bjarne Stroustrup, der Entwickler von C++ und damit einer der Urväter aller stark typisierten objektorientierten Programmiersprachen einschließlich Java, hat die Einführung von `protected` in C++ vor einigen Jahren als einen seiner größten Fehler bezeichnet. Durch `protected` werden die ansonsten klar geregelten Zuständigkeiten der Programmierer für bestimmte Programmteile durchbrochen.

⁷Default-Konstruktoren sind immer parameterlos und nur dann vorhanden, wenn kein anderer Konstruktor definiert wurde. In `Scheibe` gibt es einen Konstruktor mit drei Parametern und daher keinen Default-Konstruktor. Würde man `super(...)` im Konstruktor von `FarbigeScheibe` weglassen, dann würde implizit mittels `super()` ein parameterloser Konstruktor der Oberklasse aufgerufen, der jedoch gar nicht existiert. Das würde einen Syntaxfehler ergeben. Aus demselben Grund würde es einen Syntaxfehler geben, wenn man in `FarbigeScheibe` keinen Konstruktor definieren würde.

Nehmen wir an, `x` und `y` wären in `FarbigeScheibe` sichtbar und `entfernung` wäre (durch Weglassen von `//` aus Listing 3.21) überschrieben. Dann wäre indirekt auch `distanzZumUrsprung` geändert, da diese Methode `entfernung` aufruft. Obwohl `distanzZumUrsprung` in `Scheibe` und nicht in `FarbigeScheibe` definiert ist, würde wegen dynamischem Binden in einer Instanz von `FarbigeScheibe` dennoch `entfernung` so wie in `FarbigeScheibe` definiert ausgeführt. Beim Aufruf nicht-statischer Methoden wird immer dynamisch gebunden.

Im Allgemeinen bietet Vererbung folgende Möglichkeiten:

- Alle in der Oberklasse deklarierten bzw. definierten Variablen und Methoden werden von der Unterklasse geerbt, wenn sie nicht mit dem Modifier `static` gekennzeichnet sind. Die ererbten Variablen und Methoden sind auch in Instanzen der Untertypen vorhanden, so als ob sie in der Unterklasse deklariert bzw. definiert worden wären.
- Als `private` deklarierte bzw. definierte Variablen und Methoden sind in der Unterklasse nicht sichtbar, obwohl sie gültig sind.⁸ Sie sind von der Unterklasse aus nicht zugreifbar, aber die von der Oberklasse ererbten Methoden können darauf zugreifen.
- Ererbte Methoden, die in der Unterklasse sichtbar sind, können in der Unterklasse *überschrieben* werden. Dazu definiert man in der Unterklasse eine Methode mit derselben Signatur wie in der Oberklasse.⁹
- Die Unterklasse kann die Oberklasse *erweitern*, also zusätzliche Variablen und Methoden deklarieren bzw. definieren. Jede Deklaration einer Variablen in der Unterklasse erweitert die Oberklasse unabhängig von ererbten Variablen. Eine Definition einer Methode erweitert die Oberklasse nur dann, wenn sie keine Methode überschreibt.
- Wenn eine in der Unterklasse neu eingeführte Variable denselben Namen hat wie eine ererbte Variable, so wird die ererbte Variable *verdeckt*. In der Unterklasse ist also nur die in der Unterklasse deklarierte Variable sichtbar.

⁸Auch Variablen und Methoden mit Default-Sichtbarkeit sind in der Unterklasse nicht sichtbar, falls die Oberklasse in einem anderen Paket, also in einem anderen Verzeichnis definiert ist als die Unterklasse.

⁹Die Sichtbarkeit der Methode in der Unterklasse kann jedoch weniger stark eingeschränkt sein als die in der Oberklasse. Außerdem darf seit der Java-Version 1.5 der Ergebnistyp der Methode in der Unterklasse ein Untertyp des Ergebnistyps in der Oberklasse sein. Methoden, die mit dem Modifier `final` versehen sind, dürfen nicht überschrieben werden.

- Beim Senden einer Nachricht an ein Objekt wird die auszuführende Methode stets durch dynamisches Binden ermittelt.¹⁰ Beim Zugriff auf eine Variable erfolgt dagegen kein dynamisches Binden.
- Beim Erzeugen eines neuen Objekts wird zuerst immer ein Konstruktor der Oberklasse ausgeführt, dann der Konstruktor der Unterklasse. Mittels `super(. . .)` kann man den Konstruktor der Oberklasse bestimmen und Parameter weiterleiten.

Statische Variablen und Methoden sind von der Vererbung nicht betroffen, da sie zu den Klassen selbst und nicht zu Instanzen der Klassen gehören. Ähnlich verhält es sich mit Konstruktoren. Sie werden nicht weitervererbt, sondern müssen in jeder Klasse neu definiert werden.

3.4.2 Klassen versus Interfaces

Im Prinzip kann man komplexe Typhierarchien wie die in Abbildung 3.19 alleine nur mittels Vererbung ohne Zuhilfenahme von Interfaces aufbauen. Allerdings gibt es dabei eine Einschränkung: In Java kann eine Klasse immer nur von *einer* anderen Klasse abgeleitet sein. Jedes Interface kann dagegen mehrere andere Interfaces erweitern, und jede Klasse kann mehrere Interfaces implementieren. Andererseits werden in Interfaces (abgesehen von Konstanten) keine Variablen deklariert und keine Methoden definiert.

Auf gewisse Weise ist Vererbung mächtiger: Wenn zwei Klassen dasselbe Interface implementieren, kann es vorkommen, dass beide Klassen dieselbe Methode auf dieselbe Art und Weise definieren müssen. Wenn zwei Klassen dagegen von einer weiteren Klasse abgeleitet sind, braucht die gemeinsame Methode nur einmal in der Oberklasse definiert zu werden, und die beiden Unterklassen erben diese Methode. Aus diesem Grund wird der Ableitung von Klassen gegenüber der Verwendung von Interfaces oft bevorzugt. Das funktioniert aber nur, wenn die Typen tatsächlich eine Hierarchie bilden, also von jedem Typ im Typdiagramm höchstens ein Pfeil auf einen anderen Typ ausgeht. Werden zusätzliche Pfeile auf weitere Obertypen gebraucht, müssen dafür Interfaces verwendet werden.

Ein weiterer Unterschied besteht darin, dass durch Anwendung des Operators `new` Instanzen von Klassen erzeugt werden können, während von

Interfaces selbst keine Instanzen erzeugbar sind. Manchmal ist das Erzeugen von Instanzen unerwünscht, obwohl man Methoden erben möchte. Für diesen Zweck gibt es in Java Klassen, die so wie *Auszeichnung* in Listing 3.22 mit dem Modifier `abstract` definiert werden. Von solchen Klassen kann keine Instanz erzeugt werden. Trotzdem gibt es zur Initialisierung auch in abstrakten Klassen Konstruktoren – zumindest einen Default-Konstruktor. Weil keine Instanzen abstrakter Klassen existieren, können auch Methoden dieser Klassen mit dem Modifier `abstrakt` versehen werden. Solche abstrakten Methoden sind nicht implementiert, sondern entsprechen Signaturen wie in Interfaces. Nicht-abstrakte Klassen, die von abstrakten Klassen abgeleitet sind, müssen entsprechende Methoden definieren. Abgesehen davon werden abstrakte Klassen wie nicht-abstrakte Klassen behandelt. Somit stellen abstrakte Klassen eine Zwischenform zwischen nicht-abstrakten Klassen (von denen Instanzen erzeugbar sind) und Interfaces (die keine Variablendeklarationen und vollständigen Methodendefinitionen enthalten können) dar.

Statt mit `abstract` können Klassen und Methoden auch mit dem Modifier `final` versehen sein, der in diesem Fall eine andere Bedeutung als im Zusammenhang mit Variablen und Konstanten hat und beinahe das Gegenteil von `abstract` ausdrückt: Von einer `final` Klasse darf keine andere Klasse abgeleitet werden, während eine abstrakte Klasse nur sinnvoll ist, wenn andere Klassen von ihr abgeleitet werden. Eine `final` Methode darf in einer abgeleiteten Klasse nicht überschrieben werden, während eine abstrakte Methode in einer nicht-abstrakten Klasse überschrieben werden muss. Es ist klar, dass abstrakte Klassen und Methoden nicht `final` sein können und abstrakte Methoden niemals in nicht-abstrakten Klassen vorkommen dürfen. In einigen objektorientierten Programmierstilen werden die meisten Klassen als `final` oder `abstract` gekennzeichnet, um deutlich zu machen, welche Klassen dazu gedacht sind, erweitert bzw. nicht erweitert zu werden. In anderen Programmierstilen werden `final` Klassen dagegen kaum verwendet, weil man annimmt, dass fast alle Klassen erweiterbar sind. Praktisch gesehen macht das wenig Unterschied, da ungeplante Ableitungen von Klassen in Java und ähnlichen Sprachen kaum vorkommen. Im Vergleich zu abstrakten Methoden sind `final` Methoden eher nur in Ausnahmefällen sinnvoll.

Listing 3.22 zeigt die Verwendung von `abstract` und `final` in einem Beispiel. Die Klasse *Auszeichnung* muss abstrakt sein, weil sie eine abstrakte Methode `toInt` enthält. Wenn wir eine Variable `x` vom dekla-

¹⁰Davon ausgenommen sind `final` Methoden und alle Methoden in `final` Klassen – siehe Abschnitt 3.4.2. Auch für private Methoden ist dynamisches Binden nicht nötig, weil diese Methoden nicht überschrieben werden können.

Listing 3.22: Abstrakte und final Klassen und Methoden

```

public abstract class Auszeichnung implements Beurteilung {
    public final boolean bestanden() {
        return true;
    }
    public String toString() {
        return "mit Auszeichnung bestanden";
    }
    public abstract int toInt();
}

// österreichische Variante: 1 ist die beste Note
public final class S1 extends Auszeichnung {
    public int toInt() {
        return 1;
    }
}

// schweizer Variante: 5 ist die beste Note
public class Fuenf extends Auszeichnung {
    public final int toInt() {
        return 5;
    }
}

```

rierten Typ `Auszeichnung` haben, können wir `x.toInt()` ausführen; die Nachricht wird abhängig vom dynamischen Typ von `x`, beispielsweise `S1` oder `Fuenf`, vom Empfänger beantwortet werden. Sowohl `S1` als auch `Fuenf` muss `toInt` definieren. Die Methode `toString` kann in `S1` und `Fuenf` überschrieben werden, muss aber nicht. Auch die Anweisung `x.toString()` wird durch dynamisches Binden abhängig vom dynamischen Typ von `x` ausgeführt. Bei Bedarf ist es auch später noch leicht möglich, die Methode zu überschreiben. Uns ist wichtig, dass die Methode `bestanden` in `Auszeichnung` stets `true` zurückgibt und haben sie daher als `final` gekennzeichnet. Dadurch kann `bestanden` in den Unterklassen nicht unabsichtlich überschrieben werden. Für einen Aufruf von `x.bestanden()` reicht statisches Binden. Die Klasse `S1` ist als `final` definiert, sodass keine Klasse von `S1` ableitbar ist. Man kann alle Methoden in `S1`, egal ob in dieser Klasse definiert oder ererbt, als `final` ansehen, da sie nicht überschreibbar sind. Die Klasse `Fuenf` ist nicht `final`, wohl aber die Methode `toInt` in dieser Klasse sowie die ererbte Methode

Listing 3.23: Kombinierte Verwendung von Klassen und Interfaces

```

public interface Note {
    int toInt();
}

public class B3 extends Bestanden implements Note { ... }

public abstract class Auszeichnung implements Beurteilung, Note {
    ...
}

public class Test {
    public static void test (Note n) {
        System.out.println(n.toInt());
    }
}

```

`bestanden`. Es ist zwar möglich, weitere Klassen von `Fuenf` abzuleiten, aber `toInt` und `bestanden` dürfen dabei nicht überschrieben werden.

Listing 3.23 zeigt, wie man Klassen und Interfaces miteinander kombinieren kann. Viele Typen in der Typhierarchie in Abbildung 3.19 wie beispielsweise `B3` und `Auszeichnung` haben zwar eine Methode `toInt`, aber es gibt keinen gemeinsamen Obertyp all dieser Typen. Einen solchen gemeinsamen Obertyp können wir über das Interface `Note` einführen. Klassen wie `B3` und `Auszeichnung` implementieren dieses Interface. Nun ist es möglich, die Methode `test` in `Test` mit Instanzen all dieser Typen als aktuellen Parametern auszuführen, ohne dass dazu größere Umstrukturierungen der Typhierarchie in Abbildung 3.19 notwendig wären. Auf diese Weise können wir über Interfaces stets zusätzliche Struktur in unsere Typhierarchie bringen.

3.4.3 Von Object abwärts

Alle Java-Klassen zusammengenommen bilden eine Typhierarchie, an deren Spitze die Klasse `Object` steht. Wenn wir eine Klasse ohne `extends`-Klausel definieren (also keine Oberklasse angeben), dann wird die Klasse implizit von `Object` abgeleitet. Dadurch hat jede Klasse im von den Klassen gebildeten Typdiagramm genau einen Pfeil auf eine andere Klasse, von der sie abgeleitet ist. Davon ausgenommen ist nur die Klasse `Object`

selbst. Das bedeutet, dass jede Klasse direkt oder indirekt von `Object` abgeleitet ist und alle Methoden von `Object` erbt. Jedes Objekt ist Instanz von `Object` und hat alle Methoden, die in `Object` spezifiziert sind. Tatsächlich sind in `Object` einige häufig verwendete Methoden definiert. Einige davon wollen wir näher betrachten.

Im Interface `Beurteilung` in Listing 3.17 haben wir eine parameterlose Methode `toString` eingeführt, die eine Zeichenkette zur Beschreibung des Objekts vom Typ `Beurteilung` zurückgibt. Alle Klassen, die von `Beurteilung` abgeleitet sind, müssen diese Methode definieren. Tatsächlich wäre es gar nicht notwendig, diese Methode in `Beurteilung` zu spezifizieren, da bereits `Object` eine entsprechende Methode `toString` definiert. Allerdings ist `toString` in `Object` so definiert, dass die Methode zwar eine eindeutige, für jedes Objekt unterschiedliche Zeichenkette zurückgibt, die jedoch hauptsächlich eine wenig aussagekräftige Zahl enthält. In der Praxis ist es häufig notwendig, `toString` zu überschreiben, damit eine aussagekräftige Zeichenkette zurückgegeben wird.

In Abschnitt 2.2.2 haben wir den Verkettungsoperator kennengelernt: Falls im Ausdruck `x+y` einer der beiden Operanden vom Typ `String` ist, dann wird auch der andere Operand in eine Zeichenkette umgewandelt, und die beiden Zeichenketten werden zu einer Zeichenkette zusammengefügt. Für elementare Typen wie ganze Zahlen kann man sich leicht vorstellen, auf welche Weise ein Operand in eine Zeichenkette umgewandelt wird. Für Referenztypen kann man sich das nicht so einfach vorstellen. Hier kommt die Methode `toString` ins Spiel: Wenn ein Operand des Verkettungsoperators von einem Referenztyp ist, dann wird das Objekt durch einen Aufruf von `toString` in eine Zeichenkette umgewandelt. Beispielsweise hätten wir in Listing 3.17 statt

```
String s = "Prüfung " + b.toString() + "! ";
```

kürzer und mit derselben Semantik folgende Zeile schreiben können:

```
String s = "Prüfung " + b + "! ";
```

Durch dynamisches Binden wird stets die richtige, das ist die dem dynamischen Typ des Operanden entsprechende Implementierung von `toString` gewählt. Damit die Umwandlung in eine Zeichenkette für alle Objekte funktioniert, muss `toString` in `Object` definiert sein.

In Abschnitt 3.2.3 haben wir gesehen, dass wir streng zwischen Identität und Gleichheit von Objekten unterscheiden müssen. Dabei haben wir auch die Methode `equals` kennengelernt, die in `Object` etwa so definiert ist:

```
public boolean equals(Object that) {
    return this == that;
}
```

Das heißt, so wie in `Object` definiert vergleicht die Methode nur auf Identität. In vielen Klassen ist eine andere Definition für Gleichheit sinnvoll, und diese Methode muss entsprechend überschrieben werden. Beispielsweise ist `equals` in `String` überschrieben.

Überraschenderweise ist es recht schwierig, `equals` auf vernünftige Weise zu überschreiben. Das Problem besteht darin, dass der Parameter von `equals` immer vom Typ `Object` ist, obwohl wir eigentlich nur Objekte des gleichen Typs miteinander vergleichen wollen, also beispielsweise Zeichenketten mit Zeichenketten. Zur Lösung benötigen wir zusätzliche Sprachkonstrukte.

Listing 3.24 zeigt, wie `equals` in einigen Klassen, die wir schon betrachtet haben (Listings 3.13, 3.15 und 3.21), überschrieben werden kann. Dabei wenden wir bisher noch nicht eingeführte Sprachkonstrukte an:

Typvergleich: Der Infix-Operator `instanceof` stellt fest, ob der linke Operand eine Instanz des Referenztyps im rechten Operanden ist, und liefert als Ergebnis einen Wahrheitswert zurück. Beispielsweise gibt „`o instanceof Scheibe`“ als Ergebnis `true` zurück, wenn der dynamische Typ von `o` gleich `Scheibe` oder `FarbigeScheibe` ist; `FarbigeScheibe` ist Untertyp von `Scheibe`, und daher ist jede Instanz von `FarbigeScheibe` auch Instanz von `Scheibe`.

Cast: Im Gegensatz zum Cast auf einen elementaren Typ – siehe Abschnitt 2.2.3 – ändert ein Cast auf einen Referenztyp den *deklarierten Typ* eines Ausdrucks. Beispielsweise wird durch `(Scheibe)o` der deklarierte Typ von `o`, das ist `Object`, auf `Scheibe` geändert; damit wird es möglich, auf die Variablen `x`, `y` und `r` dieses Objekts zuzugreifen, da der deklarierte Typ des Objekts gleich der Klasse ist, in der die Anweisung steht. Ohne Cast sind die Variablen nicht zugreifbar, da sie in `Object` unbekannt sind. Generell ist ein Cast auf einen Typ `A` nur möglich, wenn der dynamische Typ des Ausdrucks ein Untertyp von `A` (einschließlich des Typs `A` selbst) ist. Das wird zur Laufzeit überprüft. Ist das nicht der Fall, kommt es zu einem Laufzeitfehler – siehe Kapitel 5. In Listing 3.24 besteht diese Gefahr nicht, da vorher über Typvergleiche sichergestellt wurde, dass der Ausdruck einen passenden dynamischen Typ hat.

Listing 3.24: Überschreiben von equals

```

public class Punkt implements Verschiebbar, Distanzmessung {
    private double x, y;
    public boolean equals(Object o) {
        return o instanceof Punkt
            && x == ((Punkt)o).x && y == ((Punkt)o).y;
    }
    ...
}

public class Scheibe implements AufFlaeche {
    private double x, y, r;
    public boolean equals(Object o) {
        return o instanceof Scheibe && r == ((Scheibe)o).r
            && x == ((Scheibe)o).x && y == ((Scheibe)o).y;
    }
    ...
}

public class FarbigeScheibe extends Scheibe {
    private long r;
    public boolean equals(Object o) {
        return o instanceof FarbigeScheibe
            && r == ((FarbigeScheibe)o).r
            && super.equals(o);
    }
    ...
}

```

Super: Bei der Definition von `equals` in `FarbigeScheibe` stehen wir vor einem Dilemma: Wir haben keinen Zugriff auf die in der Oberklasse als `private` deklarierten Variablen. Um diese Variablen zu vergleichen, müssen wir eine in `Scheibe` definierte Methode aufrufen. Die Methode `equals` in `Scheibe` würde genau das machen, was wir brauchen, aber wir können diese Methode nicht aufrufen, da sie in `FarbigeScheibe` überschrieben ist. Die Pseudo-Variable `super` schafft Abhilfe: Über `super` können wir eine Methode der Oberklasse aufrufen, auch wenn sie in der Unterklasse überschrieben wurde, sodass `super.equals(o)` die Methode `equals`, wie in `Scheibe` definiert, ausführt.

Insbesondere die Verwendung von Casts auf Referenztypen gilt als sehr fehleranfällig und ist nach Möglichkeit zu vermeiden. In diesem Beispiel

gibt es jedoch keine einfache andere Möglichkeit. Wir müssen damit leben, dass es nicht für alle Probleme eine elegante Lösung gibt, und wir müssen die Fehleranfälligkeit durch besondere Vorsicht kompensieren. Der Lösungsansatz in Listing 3.24 funktioniert nur, wenn wir `equals` in allen Klassen auf ähnliche Weise überschreiben. Wenn wir das nicht machen, wird `equals` manchmal fälschlicherweise `true` zurückgeben, obwohl nichteinmal die dynamischen Typen der verglichenen Objekte übereinstimmen. Würden wir die Methode in `FarbigeScheibe` nicht überschreiben, dann könnte sogar ein Vergleich einer nicht-farbigem Scheibe mit einer farbigen Scheibe `true` ergeben.

In Java sind auch Klassen Objekte. Sie sind Instanzen der Klasse `Class`. Die in `Object` definierte parameterlose Methode `getClass` gibt eine Referenz auf die Klasse des Objekts zurück. Statt wie in Listing 3.24 mittels `instanceof` auf eine Untertypbeziehung könnten wir auch das Übereinstimmen der Typen durch „`this.getClass() == o.getClass()`“ überprüfen. Damit würde die Fehleranfälligkeit reduziert, wenn wir auf das Überschreiben von `equals` vergessen, da ein Vergleich von Instanzen unterschiedlicher Klassen immer `false` ergeben würde. Allerdings könnte auch diese Variante nicht verhindern, dass beim Vergleich zweier farbiger Scheiben durch eine ererbte Methode die Farbe unberücksichtigt bliebe.

Die parameterlose Methode `hashCode` errechnet aus jedem Objekt eine Zahl vom Typ `int` mit der Eigenschaft, dass bei Anwendung auf gleiche Objekte auch gleiche Zahlen zurückkommen. Die Umkehrung gilt nicht, das heißt, zwei Objekte können ungleich sein, obwohl `hashCode` gleiche Zahlen liefert. Zahlen mit solchen Eigenschaften werden beispielsweise im Zusammenhang mit Hashtabellen gebraucht – siehe Abschnitt 4.3.3. Obwohl praktisch alle Beschreibungen und Manuals davor warnen, setzen unerfahrene Programmierer `hashCode` manchmal als Ersatz für `equals` ein, indem sie die zurückgegebenen Zahlen miteinander vergleichen. Das darf man auf keinen Fall machen, weil ja unterschiedliche Objekte zu gleichen Zahlen führen können und Fehler, die dadurch entstehen, sehr schwer zu finden sind.

Weiters gibt es in `Object` noch die Methode `clone` zum Erzeugen einer Kopie eines Objekts, die Methode `finalize` zum Aufräumen in nicht mehr benötigten Objekten vor Freigabe des entsprechenden Speicherbereichs, sowie die Methoden `notify`, `notifyAll` und `wait` in verschiedenen Varianten, die zur Synchronisation in der nebenläufigen Programmierung benötigt werden. Einige dieser Methoden werden wir in späteren Kapiteln noch kurz ansprechen.

3.5 Quellcode als Kommunikationsmedium

Die objektorientierte Programmierung bietet eine gigantische Vielfalt an Gestaltungsmöglichkeiten für die Faktorisierung. Es braucht viel Erfahrung um gute von schlechten Programmstrukturen unterscheiden zu können. Ein Programm ist noch lange nicht gut, nur weil es läuft und annähernd das macht, was man von ihm erwartet. Die Qualität hängt vor allem auch davon ab, wie einfach das Programm zu ändern und zu warten ist. In diesem Abschnitt werden wir einige entsprechende Kriterien betrachten.

Ein Aspekt ist im Hinblick auf die Änderbarkeit und Wartbarkeit besonders wichtig: Die an der Entwicklung beteiligten Personen müssen miteinander kommunizieren und relevante Informationen austauschen, damit alle Teile eines Programms gut zusammenpassen. Das ist schwierig, wenn diese Personen nicht zur selben Zeit am selben Ort sind. So müssen Teile des Codes, der heute geschrieben wird, auch in zehn, zwanzig oder dreißig Jahren noch weiterentwickelt und gewartet werden. Das Verständnis des Lösungsansatzes darf in dem langen Zeitraum nicht verloren gehen. Man muss Programmcode also derart schreiben, dass alle wichtigen Informationen darin enthalten und leicht auffindbar sind. Für Programmierer stellt der Quellcode das wichtigste Kommunikationsmedium dar.

3.5.1 Namen, Kommentare und Zusicherungen

Kommentare im Programm sind als Träger von Informationen aller Art unverzichtbar. Früher hat man sich oft mit dem Tipp an Programmieranfänger begnügt, alles was im Programm passiert durch Kommentare verständlich zu machen. Die meisten Programmier-Lehrbücher (einschließlich dieses Skriptums) verwenden Kommentare, um Leser auf interessante Programmstellen hinzuweisen und um einem ungeübten Programmierer ein Programmkonstrukt oder den Ablauf eines einfachen Algorithmus zu erklären. Programmieranfänger ahmen diesen Stil gerne nach. In der Praxis sind solche Kommentare jedoch oft sinnlos und sogar störend, da die an der Entwicklung beteiligten Personen in der Regel ja keine Anfänger mehr sind und durch umfangreiche, nichtssagende Kommentare vom Wesentlichen abgelenkt werden. Es kommt nicht auf den Umfang, sondern die Qualität der Kommentare an. Gute Kommentare beschreiben das, was ein geübter Programmierer nicht mit einem Blick aus dem Programmcode herauslesen kann oder im Normalfall nicht weiß, aber zum Verständnis des Programms wissen muss.

Auch Namen von Klassen, Methoden und Variablen geben einem geübten Programmierer viel Information und sollen daher passend gewählt werden. Namen stellen eine Analogie zur realen Welt her und wirken auf der intuitiven Ebene. Etwas, was intuitiv klar ist, braucht man nicht mehr durch Kommentare zu erklären, oder zumindest werden die Kommentare einfacher. Namen sollen Kommentare unterstützen. Schlecht gewählte Namen führen leicht zu einer falschen Intuition, zu einem falschen Programmverständnis und schließlich zu Fehlern. Wenn man keinen gut geeigneten Namen findet, ist es besser, einen nichtssagenden Namen zu wählen als einen, der falsch verstanden werden kann.

Nicht nur die Art, sondern vor allem auch die Auffindbarkeit von Informationen in einem Programm ist von großer Bedeutung, da man nicht davon ausgehen kann, dass eine einzelne Person jedes Detail eines großen, oft Hunderttausende oder Millionen von Zeilen umfassenden Quellcodes kennt. Ein gutes Programm enthält genau die Informationen, die man braucht, an genau der Stelle, an der man sie erwartet.

Die Faktorisierung spielt dabei eine wesentliche Rolle. In einem gut faktorisierten Programm findet man sich rasch zurecht, bei schlechter Faktorisierung wird man Informationen trotz guter Kommentare nur schwer finden. Wir werden uns in Abschnitt 3.5.2 mit Kriterien zur Unterscheidung zwischen guten und schlechten Faktorisierungen auseinandersetzen. Zuvor betrachten wir einen anderen Aspekt: So wie das ganze Programm gut faktorisiert sein muss, so müssen auch die Kommentare strukturiert sein um sich zurechtzufinden. Man muss wissen, wo man nach einem bestimmten Kommentar suchen soll um bestimmte Informationen zu erhalten.

Wichtige Punkte, an denen man nach Informationen sucht, sind Schnittstellen zwischen Programmteilen. In Java sind das Definitionen bzw. Deklarationen von Klassen, Interfaces, Methoden, Konstruktoren und Objektvariablen. Meist schreibt man Kommentare unmittelbar davor hin oder ganz am Anfang in den Rumpf, bei kurzen Definitionen bzw. Deklarationen manchmal auch danach. Folgende Informationen sollten in den Kommentaren enthalten sein:

Klassen und Interfaces: An dieser Stelle sucht man nach allgemeinen Informationen zum entsprechenden Typ. Kommentare beschreiben den Zweck und die Grobstruktur von Objekten dieses Typs. Besonderheiten in der Benutzung müssen herausgestrichen werden.

Methoden und Konstruktoren: Hier erwartet man alle Informationen, die man beim Schicken einer entsprechenden Nachricht bzw. zum

Erzeugen eines Objekts benötigt. Wenn dies durch die Namen und den Kontext nicht offensichtlich ist, soll kurz erklärt sein, welche Bedeutungen die Parameter haben, unter welchen Bedingungen und mit welchen Argumenten man eine Nachricht schicken oder ein Objekt erzeugen kann, was die Methode oder der Konstruktor macht und was ein möglicher Ergebniswert besagt.

Objektvariablen: Wenn dies aus den Namen und dem Kontext nicht klar hervorgeht, ist eine kurze Erklärung des Zwecks sinnvoll. Vor allem müssen Eigenschaften der Variableninhalte beschrieben sein, die für die Konsistenz des Objekts notwendig sind.

Informationen, die leicht aus der Syntax herauszulesen sind (etwa die Oberklasse, Typen von Parametern, etc.), sollen nicht in Kommentaren wiederholt werden. Wichtig sind dagegen Informationen, die man zur Benutzung und Erzeugung eines Objekts braucht – nicht so sehr die internen Angelegenheiten. Daher sind Kommentare zu privaten Einheiten weniger wichtig als zu überall sichtbaren und verwendbaren Einheiten.

Listing 3.25 gibt ein Beispiel für Kommentare in einer Klasse. Die Rümpfe des Konstruktors und der Methoden wurden entfernt um zu demonstrieren, dass die in den Signaturen und Kommentaren gegebenen Informationen ausreichen, um ein Objekt dieser Klasse zu erzeugen und zu verwenden. Genau für diesen Zweck setzen wir Kommentare am sinnvollsten ein.

Wie wir bereits in Abschnitt 1.5.4 gesehen haben, kann man Kommentare auch als Zusicherungen verstehen. Zusicherungen beschreiben Bedingungen, die eingehalten werden müssen. In der Regel unterscheiden wir nicht zwischen Zusicherungen und Kommentaren, die einfach nur nähere Erläuterungen geben. Wir müssen davon ausgehen, dass ein Leser jeden Kommentar als Zusicherung versteht und sich darauf verlässt. So wie schlecht gewählte Namen können auch ungenaue oder unzutreffende erläuternde Kommentare zu Fehlern führen.

Als Zusicherungen betrachtet kann man Kommentare an Objektschnittstellen auch in folgende Kategorien einteilen:

Vorbedingungen: Das sind Bedingungen in Methoden und Konstruktoren, die vor der Ausführung erfüllt sein müssen. Meist wird verlangt, dass Argumente bestimmte Eigenschaften haben. Dadurch wird die Implementierung der Methoden und Konstruktoren vereinfacht. Beispielsweise könnten wir in Listing 3.25 verlangen, dass `radius` im

Listing 3.25: Kommentare in einer Klasse – siehe Listing 3.15

```
// Eine Scheibe ist ein kreisrundes zweidimensionales Objekt auf
// einer Fläche. Es ist auf der Fläche verschiebbar.
public class Scheibe implements AufFlaeche {
    private double x, y; // die beiden Koordinaten (karthesisch)
    private double r;    // nicht-negativer Radius der Scheibe

    // Der Mittelpunkt der erzeugten Scheibe liegt anfangs auf
    // den karthesischen Koordinaten "initX" und "initY".
    // Der Absolutwert von "radius" bestimmt den Scheibenradius.
    public Scheibe(double initX, double initY, double radius) {
        ...
    }

    // Verschiebe die Scheibe auf der Fläche um deltaX Einheiten
    // nach rechts und deltaY Einheiten nach oben.
    // Negative Parameterwerte bewirken eine Verschiebung
    // nach links bzw. unten.
    public void verschiebe(double deltaX, double deltaY) {
        ...
    }

    // Bestimme die Entfernung vom Ursprung des Koordinatensystems
    // bis zum am nächsten gelegenen Punkt auf der Scheibe.
    // Ergebnis ist 0.0 wenn die Scheibe über dem Ursprung liegt.
    public double distanzZumUrsprung() {
        ...
    }

    // Bestimme die Entfernung des Punktes p bis zum am nächsten
    // gelegenen Punkt auf der Scheibe.
    // Ergebnis ist 0.0 wenn p auf der Scheibe liegt.
    public double entfernung(Punkt p) {
        ...
    }
}
```

Konstruktor einen nicht-negativen Wert hat, sodass man auf die Berechnung des Absolutwertes im Rumpf des Konstruktors verzichten kann. Methoden und Konstruktoren sind jedoch einfacher zu verwenden, wenn es keine oder möglichst wenige Vorbedingungen gibt. Daher kommen wir in Listing 3.25 ohne Vorbedingungen aus und nehmen einen etwas höheren Implementierungsaufwand in Kauf.

Nachbedingungen: Das sind Bedingungen in Methoden und Konstruktoren, die nach der Ausführung erfüllt sein müssen. Nachbedingungen beschreiben, was die Methoden und Konstruktoren machen und welche Ergebnisse zurückgegeben werden. Aufrufer verlassen sich darauf. Alle Kommentare beim Konstruktor und bei den Methoden in Listing 3.25 stellen Nachbedingungen dar.

Invarianten: Das sind Bedingungen auf Variablen und auf Klassen und Interfaces, die stets eingehalten werden müssen, damit die Programmzustände konsistent sind. Beispielsweise ist die Bedingung, dass `r` in Listing 3.25 keinen negativen Wert hat, eine Invariante. Auch die Eigenschaft, dass eine Scheibe ein kreisrundes zweidimensionales Objekt ist, kann als Invariante aufgefasst werden. Während der Ausführung einer Methode und vor der Initialisierung des Objekts darf eine Invariante kurzfristig verletzt sein. Aber nach Ausführung jeder Methode und jeden Konstruktors müssen alle Invarianten erfüllt sein.

History Constraints: Sie ähneln Invarianten, schränken aber die Entwicklung von Objekten im Laufe der Zeit ein. Beispielsweise könnte ein History Constraint besagen, dass die Zahl in einer Variablen zwar immer kleiner, aber niemals größer werden darf. Ein anderer History Constraint könnte verlangen, dass eine Methode nur unmittelbar nach einer bestimmten anderen Methode ausgeführt wird. Oft stehen History Constraints in Beschreibungen von Klassen und Interfaces, manchmal auch bei Methoden und Variablen. Während Vor- und Nachbedingungen sowie Invarianten schon lange voneinander unterschieden werden, haben sich History Constraints erst vor kurzem als eigene Kategorie etabliert und sind daher nicht so bekannt.

Die Unterscheidung zwischen diesen Arten von Zusicherungen ist wichtig, weil sie zu unterschiedlichen Zeitpunkten gelten müssen. Vorbedingungen gelten vor der Ausführung, Nachbedingungen danach und Invarianten im Wesentlichen davor und danach, während History Constraints die Entwicklung eines Objekts generell beschränken.

Namen und Kommentare können altern. Das bedeutet, dass sie, obwohl ursprünglich sorgsam und gut gewählt, im Laufe der Zeit durch Änderungen des Programmcodes an Aussagekraft verlieren oder sogar ihre Korrektheit einbüßen. Es ist notwendig, bei Programmänderungen immer auch die Kommentare zu berücksichtigen und gegebenenfalls anzupassen.

Manchmal ist es unvermeidlich, Variablen, Methoden, etc. umzubenennen.

Erfahrene Programmierer verlassen sich nicht unvoreingenommen auf Namen und Kommentare. Sie überprüfen gelegentlich, wie gut die Namen und Kommentare den tatsächlichen Sachverhalt treffen. Entsprechend dem Prüfergebnis bewerten sie ihr Vertrauen in das Programm. Vertrauen auf andere Programmierer ist eine Grundvoraussetzung bei der Entwicklung großer Programme, da man ohne Vertrauen lieber Programmteile noch einmal schreibt als bereits vorhandene Programmteile zu verwenden, oder zumindest unnötige Überprüfungen einbaut. Aus diesem Grund können schlecht gewählte Namen und Kommentare zu viel unnötigem Programmcode und zu langen Entwicklungszeiten führen. Gute Namen und Kommentare sind daher ein essentielles Qualitätsmerkmal.

3.5.2 Faktorisierung, Zusammenhalt und Kopplung

Wie schon mehrfach betont ist die Faktorisierung (also die Zerlegung eines Programms in kleinere Einheiten) von entscheidender Bedeutung. Ein Programm ist dann gut faktorisiert wenn

- alle zusammengehörigen Eigenschaften und Aspekte des Programms zu übersichtlichen Einheiten zusammengefasst sind
- und Eigenschaften und Aspekte, die nichts miteinander zu tun haben, so klar voneinander getrennt sind, dass sie unabhängig voneinander geändert werden können.

Gute Faktorisierung erhöht die Wahrscheinlichkeit für lokale Programmänderungen. Bei einer lokalen Änderung braucht man das Programm nur an einer Stelle zu betrachten und kann es auch an dieser Stelle anpassen. Eine nichtlokalen Änderung verlangt die Betrachtung mehrerer oder vieler Programmstellen und gleichzeitige Anpassungen an mehreren Stellen. Faktorisierung und Wartbarkeit sind daher eng miteinander verknüpft.

Die Forderung nach einer guten Faktorisierung nützt uns nicht viel, solange wir keine einfachen und schon frühzeitig heranziehbaren Beurteilungskriterien dafür haben, ob die Faktorisierung gut ist. Im Nachhinein, also nachdem wir die Programmänderungen schon durchgeführt haben, nützt uns das Wissen darüber, wie einfach die Änderungen durchzuführen waren, nicht mehr viel. Leider gibt es keine klar nachvollziehbaren derartigen Beurteilungskriterien. Es gibt jedoch grobe Faustregeln, die uns oft, aber nicht immer den richtigen Weg weisen. Zwei solche Faustregeln betrachten wir etwas näher:

Klassenzusammenhalt: Darunter versteht man den *Grad des Zusammenhangs zwischen den Inhalten einer Klasse*. Die Inhalte entsprechen, vereinfacht ausgedrückt, den Variablen und Methoden der Klasse, aber auch den Kommentaren, welche diese Inhalte beschreiben. Der Klassenzusammenhalt ist hoch, wenn alle Variablen und Methoden gut zusammenpassen und die Namen und Kommentare den Zweck und die Verwendung von Instanzen der Klasse treffend beschreiben. Ist der Klassenzusammenhalt hoch, so verringert er sich deutlich wenn man eine Gruppe von Variablen und Methoden entfernt oder hinzufügt, sinnändernd umbenennt oder Kommentare inhaltlich ändert. Bei niedrigem Klassenzusammenhalt haben derartige Änderungen nur geringfügige Auswirkungen auf den Klassenzusammenhalt oder können ihn sogar erhöhen. Durch gedanklich durchgeführte Änderungen der Klasse kann man den Klassenzusammenhalt grob abschätzen. Das Ziel ist ein möglichst hoher Zusammenhalt aller Klassen in einem Programm. Er weist auf eine gute Faktorisierung hin.

Objektkopplung: Das ist die Stärke der *Abhängigkeiten der Objekte voneinander*. Es werden alle Objekte in einem laufenden System gemeinsam betrachtet. Die Objektkopplung ist stark, wenn die Objekte viele nach außen sichtbaren Methoden und Variablen haben, viele Nachrichten an andere Objekte (nicht `this`) geschickt und Variablen in anderen Objekten zugegriffen werden, und die Anzahl der Parameter dieser Methoden hoch ist. Auch die Objektkopplung lässt sich gedanklich ganz gut abschätzen. Das Ziel ist eine möglichst schwache Objektkopplung. Sie weist darauf hin, dass notwendige Änderungen des Programms eher lokal bleiben.

Die Begriffe Klassenzusammenhalt und Objektkopplung sind schon lange in Verwendung, haben sich aber bisher einer exakten Definition erfolgreich widersetzt. Der Klassenzusammenhalt beschreibt einfach nur den von uns gefühlten Grad des Zusammenhangs zwischen den Klasseninhalten, keineswegs einen auf einer Skala messbaren, normierten Grad. Genauso entspricht die Objektkopplung einer geschätzten Stärke der Abhängigkeiten zwischen den Objekten. Es gibt zwar zahlreiche Versuche, diese Begriffe auf eine wohldefinierte, normierte Basis zu stellen, allerdings mit wenig Erfolg. Im Gegensatz zu gemessenen Größen tendieren Abschätzungen dazu, dass wir unbewusst Gewichtungen vornehmen. Aspekte, die uns wichtig erscheinen, gehen stärker in die Abschätzung ein. Vielleicht ist eine grobe Abschätzung einer gemessenen Größe deswegen oft überlegen.

Tatsächlich dürfte ein anderer Grund dafür ausschlaggebend sein, dass wir uns auf grobe Abschätzungen und kaum definierte Kriterien verlassen: Wir vergleichen im Kopf unterschiedliche Lösungsansätze schon lange bevor es Programmcode gibt, in dem wir etwas messen könnten. Im direkten Vergleich zweier Ansätze kann man schon recht früh abschätzen, welcher Ansatz wahrscheinlich zu einem höheren Klassenzusammenhalt und einer schwächeren Objektkopplung führen wird. Auf Basis dieser Abschätzungen treffen wir unsere Designentscheidungen. Genauere Definitionen der Begriffe wären dafür kaum hilfreich.

Tendenziell besagt ein hoher Klassenzusammenhalt, dass ein Programm schon eine recht gute Struktur hat und es daher wahrscheinlich nicht mehr nötig sein wird, größere Verbesserungen an der Struktur vorzunehmen. Bei schwächerer Objektkopplung ist die Wahrscheinlichkeit höher, dass eine Änderung in einer Klasse keine weiteren Änderungen in anderen Klassen notwendig macht. Glücklicherweise stehen diese beiden Begriffe in engem Zusammenhang zueinander: Oft ist der Klassenzusammenhalt genau dann hoch, wenn die Objektkopplung schwach ist, und umgekehrt. Das kommt daher, dass eine Programmstruktur, also die Faktorisierung genau dann als gut betrachtet wird, wenn möglichst viele Änderungen lokal durchführbar sind. Nicht selten ist der Klassenzusammenhalt einfacher gefühlsmäßig fassbar als die Objektkopplung, weil man nur einzelne Klassen und nicht das ganze System zu betrachten braucht. Dennoch lassen sich Rückschlüsse auf die Objektkopplung ziehen.

Auch die beste Faktorisierung kann nicht garantieren, dass alle Programmänderungen lokal durchführbar sind. Wenn man beispielsweise die Signatur einer nicht-privaten Methode ändert – sagen wir, `verschiebe` aus Listing 3.25 bekommt einen zusätzlichen Parameter –, dann ist davon nicht nur die Definition der Methode selbst betroffen, sondern auch jede Programmstelle, an der eine entsprechende Nachricht geschickt wird. Alle diese Programmstellen muss man finden und ändern. Generell wirken sich Änderungen an Schnittstellen immer auf mehrere oder viele Stellen im Programm aus. Daher soll man sich bemühen, Schnittstellen so stabil wie möglich zu halten, sie also niemals ohne wichtigen Grund zu verändern.

Zu den Schnittstellen gehören auch die Kommentare an den Schnittstellen. Wenn man beispielsweise den Kommentar zu `verschiebe` in Listing 3.25 so abändert, dass positive Werte von `delta x` eine Verschiebung nach links bewirken, dann ergeben sich daraus nicht nur Änderungen der Methode, sondern auch Änderungen an allen Stellen, an denen eine solche Nachricht geschickt wird – genauso wie bei einer Änderung der Signatur.

Wer einmal begriffen hat, was ein Kommentar bewirkt und welche Auswirkungen eine Änderung eines Kommentars haben kann, wird in diesem Zusammenhang kaum mehr das Wort „harmlos“ in den Mund nehmen. Auswirkungen einer Änderung eines Kommentars sind selten lokal. Eine Änderung in einer Methode, bei der die Signatur und die Kommentare unverändert bleiben können, ist dagegen wirklich recht harmlos. Änderungen an privaten Methoden sind auch dann relativ harmlos, wenn die Signatur und die Kommentare davon betroffen sind. Alle Aufrufe erfolgen ja nur innerhalb der Klasse, und weitere nötige Änderungen bleiben auf die Klasse beschränkt. Bei schlechter Faktorisierung haben wir insgesamt mehr nicht-private Methoden und eine höhere Wahrscheinlichkeit dafür, dass Änderungen an deren Schnittstellen nötig sind.

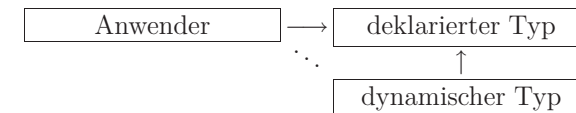
3.5.3 Ersetzbarkeit und Verhalten

Wir wissen aus Abschnitt 3.3.3, dass Ersetzbarkeit eine entscheidende Rolle für Untertypbeziehungen spielt: Ein Typ U ist genau dann Untertyp eines Typs T , wenn Objekte vom Typ U überall verwendbar sind, wo Objekte vom Typ T erwartet werden. Dieses Ersetzbarkeitsprinzip ist *nicht* automatisch erfüllt, wenn die Signaturen zweier Objekte zusammenpassen. Darüber hinaus muss auch gelten, dass sich alle Methoden in jedem Objekt vom Typ U so verhalten, wie man es sich von entsprechenden Methoden in Objekten vom Typ T erwartet. Die Kompatibilität der Signaturen wird vom Compiler überprüft. Für die Überprüfung der Kompatibilität des Verhaltens sind ausschließlich wir beim Programmieren verantwortlich. Wir müssen dafür sorgen, dass eine `extends`- oder `implements`-Klausel nur dort verwendet wird, wo das Verhalten kompatibel ist. Machen wir das nicht, kommt es zu schweren Fehlern.

Um bezüglich des Verhaltens kompatibel zu sein muss eine Methode im Untertyp dasselbe machen wie die entsprechende Methode im Obertyp. Allerdings kann die Methode im Untertyp diese Aufgabe auf andere Weise erledigen als die im Obertyp. Beispielsweise verlangt der Kommentar der Methode `distanzzumUrsprung` im Interface `Distanzmessung` in Listing 3.12, dass die Distanz zum Ursprung zurückgegeben wird. Jede Implementierung dieser Methode muss das machen. Jedoch ist die Beschreibung im Interface ungenau. Die Methode `distanzzumUrsprung` in der Klasse `Scheibe` in Listing 3.25 präzisiert die Beschreibung derart, dass die Distanz zum nächstgelegenen Punkt der Scheibe zurückgegeben wird. Der Kommentar in der Klasse ist mit jenem im Interface kompatibel.

Wie in diesem Beispiel geht man davon aus, dass das Verhalten von Methoden durch deren Namen und Kommentare so weit beschrieben ist, dass man für die Verwendung der Methoden (also das Senden entsprechender Nachrichten) nichts Zusätzliches wissen muss. Namen und Kommentare *spezifizieren das Verhalten*. Genau diese Spezifikationen des Verhaltens sind für die Einhaltung des Ersetzbarkeitsprinzips entscheidend. Um festzustellen, ob zwei Spezifikationen von Methoden kompatibel sind, vergleichen wir die Kommentare, da die Namen ohnehin gleich sein müssen. Die Kommentare müssen inhaltlich übereinstimmen, abgesehen davon, dass Kommentare im Untertyp präziser sein dürfen als im Obertyp.

Ersetzbarkeit hilft dabei, unterschiedliche Teile eines Programms voneinander zu *entkoppeln*. Folgende Grafik macht das deutlich:



Ein Programmstück, hier Anwender genannt, sendet Nachrichten an ein Objekt, von dem nur der deklarierte Typ bekannt ist. Der Anwender hat über den Empfänger nur Informationen, die im deklarierten Typ enthalten sind, und anhand dieser Informationen wird garantiert, dass das Objekt entsprechende Methoden ausführen kann – dargestellt durch den Pfeil nach rechts. Der dynamische Typ ist Untertyp des statischen Typs – dargestellt durch den Pfeil nach oben. Sowohl der Anwender als auch der dynamische Typ sind über Pfeile an den statischen Typ gekoppelt, aber der dynamische Typ bleibt dem Empfänger trotzdem verborgen – angedeutet durch die gepunktete Linie.

Der wichtigste Vorteil der Entkopplung wird klar, wenn wir den dynamischen Typ gegen einen anderen Untertyp des deklarierten Typs austauschen: Für den Anwender ändert sich dadurch genau gar nichts. Wenn wir Änderungen am dynamischen Typ vornehmen, sodass die Untertypbeziehung erhalten bleibt, werden keinerlei Änderungen am Anwender notwendig. Solange wir den deklarierten Typ (einschließlich aller Kommentare) unverändert lassen und für eine Untertypbeziehung (einschließlich kompatiblen Verhaltens) sorgen, bleiben Änderungen des dynamischen Typs lokal und damit harmlos. Besonders wichtig ist die Entkopplung dann, wenn es Anwender gibt, die wir gar nicht kennen. Dann müssen wir den deklarierten Typ auf jeden Fall unverändert lassen.

Mittels Entkopplung können Untertypbeziehungen viel zur Wartbarkeit beitragen. Die Wartbarkeit ist besser, wenn Objektvariablen und formale

Parameter mit *stabilen* Typen deklariert sind, die sich im Laufe der Zeit kaum ändern. Stabil sind vor allem Typen, die häufig verwendet werden und deswegen bereits gut getestet sind. Generell sind Typen weiter oben in der Typhierarchie stabiler als solche weiter unten. Für deklarierte Typen verwendet man besser Interfaces oder abstrakten Klassen, die nur dem einen Zweck dienen, der benötigt wird, als mit spezialisierten Klassen, deren vielfältige Verwendungsmöglichkeiten man gar nicht braucht. Dadurch bleiben notwendige Änderungen viel häufiger lokal.

Programmieranfänger glauben oft, das Erben von Methoden aus einer Oberklasse würde eine bedeutende Einsparung bringen, was die Größe des zu schreibenden Programmcodes betrifft. Tatsächlich sind die Einsparungen durch das Erben selbst meist sehr bescheiden. Dagegen bringen die Untertypbeziehungen, die durch das Ableiten von Klassen oder Implementieren von Interfaces eingeführt werden, häufig wirklich große Einsparungen. Damit ist gemeint, dass ein einziges Programmstück (etwa der Anwender in obiger Grafik) mit vielen Objekten unterschiedlicher dynamischer Typen umgehen kann. Wenn es die Unterscheidung zwischen dem deklarierten und dynamischen Typ nicht gäbe, müsste man einen eigenen Anwender für jeden benötigten Typ schreiben. Das wäre ein großer Aufwand, den man sich durch die Untertypbeziehungen erspart.

Daraus kann man eine Empfehlung ableiten: Beim Aufbau einer Typhierarchie soll man *nicht* darauf achten, möglichst viele Methoden von einer Oberklasse zu erben. Stattdessen soll man dafür sorgen, dass jeder Anwender einen deklarierten Typ haben kann, der den tatsächlichen Bedürfnissen entspricht. Dabei muss man sich ganz auf das Verhalten der Methoden konzentrieren, also dafür sorgen, dass jede Methode in einem Untertyp ein zur entsprechenden Methode im Obertyp kompatibles Verhalten hat. Wenn man so vorgeht, entstehen Typhierarchien, die für eine gute Entkopplung sorgen, und in denen Typen weit oben in der Typhierarchie eher stabil sind. Anfangs hat man vielleicht das Gefühl, dass man dabei durch eine höhere Anzahl von Klassen und Interfaces unnötigen Programmcodes schreibt. Letztendlich erspart man sich jedoch das Schreiben von viel Programmcodes, und das Programm wird einfacher wartbar.

3.6 Objektorientiert programmieren lernen

Die objektorientierte Programmierung hat sich für größere Projekte auf breiter Front durchgesetzt. Ohne entsprechende Kenntnisse kann man

kaum noch Software entwickeln. Daher gibt es mehrere Lehrveranstaltungen, die sich diesem Thema annehmen. Die *Objektorientierte Modellierung* hat einen Schwerpunkt im Entwurf von Klassen und Typhierarchien und deren Darstellung. In *Objektorientierte Programmiertechniken* geht es um die Konstruktion gut wartbarer Software mit Hilfe des objektorientierten Programmierparadigmas. Zahlreiche weitere Lehrveranstaltungen setzen gute objektorientierte Programmierkenntnisse voraus.

3.6.1 Konzeption und Empfehlungen

Objektorientiert programmieren zu lernen ist ein langwieriger Prozess. Man darf sich nicht erwarten, damit gleich von Anfang an größere Softwareprojekte realisieren zu können. Es braucht Zeit, bis man die dafür notwendige Erfahrung hat.

Aus diesem Grund ist dieses Kapitel so aufgebaut, dass man

- erfährt, welche Ziele man in der objektorientierten Programmierung verfolgt und welche Konzepte und Techniken dahinter stecken,
- die entsprechenden Sprachkonstrukte von Java kennen und auf kleine Aufgaben anwenden lernt,
- eine grobe Vorstellung davon bekommt, wie man in der objektorientierten Programmierung vorgeht und worauf es dabei ankommt.

Mit den hier vermittelten Kenntnissen kann man noch lange keine guten objektorientierten Programme schreiben. Aber man sollte zumindest wissen, warum Java-Programme so aufgebaut sind, wie wir sie kennengelernt haben, und wozu die Sprachkonstrukte da sind. Diese Kenntnisse reichen zum Schreiben eigener kleiner Programme aus. Wenn man viel programmiert und dabei auch die objektorientierten Sprachkonstrukte einsetzt und deren Ziele im Blick behält, bekommt man mit der Zeit ein Gefühl für die richtige Verwendung. Man entwickelt seinen eigenen Programmierstil. Später im Studium, wenn man an größere Softwareprojekte herangeht und die objektorientierten Konzepte gezielt einsetzen muss, ist ein aus eigener Erfahrung entwickelter Programmierstil von unschätzbarem Vorteil. Dieser Programmierstil wird sich ständig weiterentwickeln.

Die wichtigste Empfehlung lautet daher wieder einmal, viel zu üben und praktische Erfahrung zu sammeln. Dabei darf man sich jedoch nicht von kleinen Rückschlägen entmutigen lassen. Oft lernt man aus Fehlern mehr

als aus erfolgreichen Versuchen. Das gilt besonders in der objektorientierten Programmierung, in der es eine riesige Zahl an Möglichkeiten gibt, von denen aber nur wenige wirklich gut sind. Man wird in viele Fallen tappen bevor man die größten Fallen kennt und einen guten Weg zwischen den Fallen hindurch findet.

3.6.2 Kontrollfragen

- Wodurch unterscheidet sich der objektorientierte vom prozeduralen Programmierstil?
- Was ist ein Objekt?
- Für welche Arten von Programmen eignet sich die objektorientierte Programmierung gut?
- Mit welchen Problemen muss man bei der Entwicklung großer Programme rechnen?
- Was versteht man unter inkrementeller Softwareentwicklung?
- Was bedeutet der Begriff Faktorisierung? Wann ist eine Faktorisierung gut, wann nicht?
- Wodurch unterscheiden sich Objektvariablen von lokalen Variablen?
- Was ist und wozu dient Kapselung, Data Hiding und Datenabstraktion?
- Was ist eine Nachricht, und warum spricht man vom Senden von Nachrichten und nicht einfach nur vom Aufruf von Methoden?
- Was versteht man unter einer Schnittstelle eines Objekts, was unter seiner Implementierung?
- Was sind und wozu verwendet man Klassen?
- Wie kann man in Java die Sichtbarkeit beeinflussen?
- Wo sollen die meisten Objektvariablen sichtbar sein?
- Was haben Getter- und Setter-Methoden mit Data Hiding zu tun?
- Erklären sie die Begriffe Identität, Zustand und Verhalten.

- Wie vergleicht man in Java Objekte auf Identität bzw. Gleichheit?
- Wozu dient ein Konstruktor und wie definiert man ihn?
- Wofür verwendet man die Pseudovariable `this` und Ausdrücke der Form `this(...)`?
- Wie setzt man statische Methoden und Klassenvariablen ein?
- Was unterscheidet Konstanten von Klassenvariablen?
- Wozu dienen Interfaces?
- Was meint man, wenn man von der Implementierung eines Interfaces spricht?
- Wann spricht man von Polymorphismus? Welche Rolle spielt der Polymorphismus in der objektorientierten Programmierung?
- Unter welchen Bedingungen ist ein Typ U Untertyp eines Typs T ?
- Wozu benötigt man dynamisches Binden?
- Inwiefern hängt dynamisches Binden mit Mehrfachverzweigungen zusammen?
- Warum ist dynamisches Binden gegenüber `switch`-Anweisungen zu bevorzugen?
- Welchen Zweck haben Spezialisierungen und Analogien zur realen Welt?
- Was besagt das Ersetzbarkeitsprinzip?
- Was versteht man unter Vererbung?
- Erklären Sie die Begriffe Basisklasse, abgeleitete Klasse, Unterklasse und Oberklasse.
- Was ist eine überschriebene Methode?
- Warum deklariert man Variablen nicht generell als `protected`?
- Wie werden Objekte abgeleiteter Klassen initialisiert?

- Wozu dient `super(...)` und wo kann diese Anweisung verwendet werden?
- Unterscheidet sich ein Methodenaufruf von einem Variablenzugriff hinsichtlich dynamischem Binden?
- Zu welchem Zweck kann man Klassen und Methoden mit einem Modifizier `abstract` bzw. `final` versehen?
- Wie kann man durch Interfaces zusätzliche Struktur in ein Programm bringen?
- Welche Methoden sind in `Object` vorhanden und welchen Zweck haben sie?
- Wie kann man zur Laufzeit den dynamischen Typ, also die Klasse eines Objekts feststellen (drei Möglichkeiten)?
- Was unterscheidet Casts auf Referenztypen von solchen auf elementaren Typen? Warum soll man sie vermeiden?
- Wodurch unterscheiden sich die Pseudovariablen `this` und `super` voneinander?
- Warum eignet sich `hashCode` nicht für Vergleiche von Objekten?
- Welche Informationen soll in Kommentaren von Klassen, Interfaces, Methoden, Konstruktoren und Objektvariablen enthalten sein?
- Welche Arten von Zusicherungen in Form von Kommentaren kann man unterscheiden?
- Inwiefern können Namen und Kommentare altern? Was kann man dagegen tun?
- Wie können schlecht gewählte Namen und Kommentare zu unnötigem Programmcode führen?
- Was zeichnet gut faktorisierte Programme aus?
- Erklären Sie die Begriffe Klassenzusammenhalt und Objektkopplung. Wie hängen sie mit der Faktorisierung zusammen?
- Wie kann man den Klassenzusammenhalt und die Objektkopplung abschätzen?

- Wann sind notwendige Änderungen von Kommentaren gefährlich, wann eher harmlos?
- Wie spezifiziert man das Verhalten?
- Wann ist das Verhalten eines Untertyps mit dem eines Obertyps kompatibel?
- Wodurch entkoppelt Ersetzbarkeit Programmteile voneinander?
- Welche Typen sind eher stabil?
- Wo soll man besonders auf stabile Typen achten?
- Warum ist es nicht sinnvoll, möglichst viel Programmcode von Oberklassen erben zu wollen?

4 Daten, Algorithmen und Strategien

Wir beschäftigen uns nun mit einigen prinzipiellen Vorgehensweisen beim Lösen von Programmieraufgaben. Die Komplexität vieler Aufgaben lässt sich durch Anwendung bestimmter Denkmuster deutlich vereinfachen. Zunächst betrachten wir einige Begriffe etwas näher und stellen schließlich die wichtigsten Strategien zur Lösung allgemeiner Aufgaben vor.

4.1 Begriffsbestimmungen

Die Suche nach geeigneten Algorithmen und Datenstrukturen steht im Mittelpunkt der Konstruktion von Programmen. Obwohl dieser Aspekt der Programmierung sehr viel Kreativität erfordert und nicht automatisierbar ist, so lassen wir uns bei der Suche danach dennoch von bestimmten Strategien leiten, die schon in vielen Fällen erfolgreich waren.

4.1.1 Algorithmus

Der Begriff *Algorithmus* stammt vom im Mittelalter verwendeten lateinischen Begriff *algorismus*, der die Kunst des Rechnens mit Zahlen bezeichnete. Tatsächlich dürfte *algorismus* aus der Verstümmelung des Beinamens eines arabischstämmigen Mathematikers entstanden sein. In jüngerer Zeit hat sich die Bedeutung des Begriffs Algorithmus mit dem Aufkommen von Computern etwas verschoben. Im Mittelpunkt steht nicht mehr nur das Rechnen mit Zahlen, sondern ein Algorithmus ist ganz allgemein ein System von Regeln zur schrittweisen Umformung von Zeichenreihen. Dazu zählen Anweisungen zur formalen Verarbeitung von Informationen. Es gibt zahlreiche Versuche, Algorithmus als modernen Begriff zu definieren, jedoch keine allgemein akzeptierte Variante. Klar ist, dass es sich beim Algorithmus um eine eindeutige Handlungsvorschrift handeln muss, deren Befolgung auf die Lösung eines Problems abzielt. Schwierigkeiten bereitet dagegen die Forderung, dass nach endlich vielen Schritten tatsächlich eine Lösung eines Problems berechnet ist. Aufgrund der Unentscheidbarkeit des Halteproblems (siehe Abschnitt 1.5.3) hätten wir keine Möglichkeit zu

Listing 4.1: Unterschiedliche Implementierungen desselben Algorithmus

```

int sumupLoop (int n) {
    int sum = 0;
    while (n > 0) {
        sum += n;
        n--;
    }
    return (sum);
}

int sumupRec (int n) {
    if (n > 0) {
        return (n + sumupRec(n - 1));
    }
    else {
        return (0);
    }
}

```

entscheiden, ob eine bestimmte Handlungsvorschrift einen Algorithmus darstellt oder nicht. Aus praktischer Sicht spielen solche Spitzfindigkeiten glücklicherweise keine Rolle. Wir wollen eher ein Gefühl dafür vermitteln, was ein Algorithmus ist bzw. wann zwei Algorithmen gleich sind.

Betrachten wir als Beispiel die Berechnung der Summe aller Zahlen von 1 bis zu einer Obergrenze n , also $1+2+\dots+n$. Dazu können wir folgenden einfachen Algorithmus verwenden: Solange n größer 0 ist, addieren wir n zur Summe, vermindern n um 1 und wiederholen den Vorgang mit der verminderten Zahl n . Listing 4.1 zeigt zwei unterschiedliche Implementierungen dieses Algorithmus. Die Methode `sumupLoop` verwendet eine Schleife, um Wiederholungen auszudrücken, `sumupRec` verwendet dazu Rekursion. Trotz großer Unterschiede gehen beide Methoden nach demselben Algorithmus vor. An diesem Beispiel können wir erkennen, dass wir zwischen einem Algorithmus und den Implementierungen dieses Algorithmus unterscheiden müssen.

In der Mathematik nennt man solche Summen Dreieckszahlen. Man kann sie mit der gaußschen Summenformel $1+2+\dots+n = \frac{n \cdot (n+1)}{2}$ berechnen. Die Methode `triangleNum` in Listing 4.2 löst dieselbe Aufgabe wie die beiden `sumup`-Varianten (Berechnen einer Dreieckszahl) mit Hilfe dieser Formel. Auch in den Fällen, die durch die Formel nicht abgedeckt sind ($n \leq 0$), liefert `triangleNum` dasselbe Ergebnis. Die Algorithmen sind jedoch gänzlich verschieden. Dieses Beispiel zeigt, dass ein und dasselbe Problem durch unterschiedliche Algorithmen lösbar ist.

Alle Methoden in den Listings 4.1 und 4.2 haben dieselben funktionalen Eigenschaften (siehe Abschnitt 1.6.4), liefern also dieselben Ergebnisse. In den nichtfunktionalen Eigenschaften unterscheiden sie sich jedoch.

Listing 4.2: Ein anderer Algorithmus zur Lösung desselben Problems

```

int triangleNum (int n) {
    if (n > 0) {
        return (n * (n + 1) / 2);
    }
    else {
        return (0);
    }
}

```

Die Methode `triangleNum` ist den beiden `sumup`-Varianten vorzuziehen: Der Ressourcenverbrauch, insbesondere die Laufzeit, ist vor allem für größere Zahlen n deutlich kleiner, da nur ein einziger Ausdruck berechnet wird, während in den `sumup`-Varianten viele einzeln berechnete Summanden aufaddiert werden. Außerdem ist dieser eine Ausdruck für jemanden, der die Formel kennt, einfach verständlich. Die Formel ist im statischen Programmcode direkt ersichtlich, während wir uns für ein Verständnis der `sumup`-Varianten in den dynamischen Programmablauf hineinendenken müssen. Andererseits gibt die ursprüngliche Darstellung des Problems durch $1+2+\dots+n$ bereits einen dynamischen Ablauf vor, und es ist nicht offensichtlich, dass die Formel dasselbe Ergebnis berechnet.

Wenn wir uns mit Algorithmen beschäftigen, stellen wir uns solche Fragen wie in diesen Beispielen. Es geht darum, auf welche Weise ein Problem gelöst werden kann und welche Eigenschaften entsprechende Algorithmen aufweisen. Wir wollen einen für unseren Zweck gut geeigneten Algorithmus finden. Die Qualität der Algorithmen ist im Großen und Ganzen nicht von Sprachen und Implementierungsvarianten abhängig: Nichtfunktionale Eigenschaften unterschiedlicher Algorithmen unterscheiden sich viel stärker voneinander als unterschiedliche Implementierungen desselben Algorithmus. Wenn wir Algorithmen betrachten, abstrahieren wir über Details von Programmiersprachen und Implementierungen.

4.1.2 Datenstruktur

Eine *Datenstruktur* beschreibt, wie die Daten relativ zueinander angeordnet sind und wie auf die einzelnen *Datenelemente* (kurz *Elemente*) zugegriffen werden kann. Zur Charakterisierung einer Datenstruktur sind hauptsächlich die Zugriffsoperationen entscheidend. Hier ist eine kleine

Auswahl aus der großen Zahl an sinnvollen Datenstrukturen:

Array: Die Elemente dieser einfachen Datenstruktur liegen nebeneinander im Speicher und sind über einen Index direkt und effizient adressierbar, siehe Abschnitt 2.5. Die Anzahl der Elemente ist beschränkt und wird spätestens bei der Erzeugung des Arrays festgelegt.

Verkettete Liste: Jeder Listeneintrag verweist auf den nächsten Listeneintrag, siehe Abschnitt 4.2.1. Die Anzahl der Elemente ist dadurch nicht beschränkt und das Hinzufügen weiterer Elemente gestaltet sich einfach. Jedoch sind die Elemente nicht direkt über einen Index zugreifbar, und die Suche nach bestimmten Elementen ist aufwendig.

Binärer Baum: Jeder Eintrag verweist auf bis zu zwei weitere Einträge, wobei eine bestimmte Sortierung der Einträge eingehalten wird, siehe Abschnitt 4.2.3. Damit ergeben sich ähnliche Eigenschaften wie bei der verketteten Liste, jedoch ist die Suche nach Elementen effizienter und das Hinzufügen neuer Elemente etwas aufwendiger.

Hashtabelle: Jedes Element wird in einer Tabelle mit fixer Größe an einem Index abgelegt, der sich aus dem Element selbst errechnet, siehe Abschnitt 4.3.3. Solange die Tabelle nur zu einem kleinen Teil gefüllt ist, sind sowohl Hinzufügen als auch Suche recht effizient. Bei stärkerer Füllung werden jedoch alle Zugriffe ineffizient.

Stack: Aus einem Stack können Elemente nur in der Reihenfolge gelesen und entfernt werden, die genau umgekehrt zur Reihenfolge des Einfügens ist. Einige Algorithmen brauchen diese Eigenschaft. In diesen Fällen sind Stacks sehr effizient, in anderen Fällen wenig sinnvoll. Stacks werden beispielsweise zur Implementierung von Programmiersprachen gebraucht, um geschachtelte Methodenaufrufe abzubilden.

Bei der Konstruktion von Programmen entscheiden wir anhand der benötigten Eigenschaften der Zugriffsoperationen, welche Datenstruktur zur Lösung unserer Aufgabe am ehesten passt. Häufig kombinieren wir mehrere einfache Datenstrukturen zu einer größeren, beispielsweise zu einem Array von verketteten Listen. Auf diese Weise können wir über einen Index rasch auf die gewünschte Liste zugreifen und haben die Möglichkeit, ohne großen Aufwand beliebig viele Einträge an einem Arrayindex abzulegen – falls wir diese Eigenschaften brauchen. Durch eine gute Kenntnis einfacher

Listing 4.3: Ein durch ein Array implementierter Stack

```

1 public class IntStack { // Stack von ganzen Zahlen
2     private int[] elems; // Array enthält Stackelemente
3     private int top = 0; // nächster freier Arrayindex
4
5     // Initialisierung mit Array der Größe max; max > 0
6     public IntStack (int max) {
7         elems = new int[max];
8     }
9
10    // elem wird auf den Stack gelegt
11    // ArrayIndexOutOfBoundsException falls kein freier Platz
12    public void push (int elem) {
13        elems[top] = elem;
14        top++;
15    }
16
17    // oberster Eintrag wird vom Stack geholt
18    // ArrayIndexOutOfBoundsException falls kein Eintrag vorhanden
19    public int pop() {
20        top--;
21        return (elems[top]);
22    }
23 }

```

Datenstrukturen lässt sich oft ohne großen Aufwand eine Datenstruktur mit allen benötigten Eigenschaften zusammensetzen.

Die Klasse `IntStack` in Listing 4.3 implementiert einen Stack für ganze Zahlen. Stackinträge werden in einem Array abgelegt, dessen Größe im Konstruktor bestimmt wird. Bei Über- oder Unterschreitung der Arraygrenzen wird bei Arrayzugriffen eine Exception geworfen, um die man sich bei der Verwendung des Stacks kümmern muss – siehe Kapitel 5.

Betrachten wir einige allgemeine Eigenschaften von Datenstrukturen:

- Ähnlich wie bei Algorithmen müssen wir streng zwischen Datenstrukturen und Implementierungen von Datenstrukturen unterscheiden. Unter einer bestimmten Datenstruktur verstehen wir eine Ansammlung von Daten mit bestimmten Zugriffsoperationen und Eigenschaften. Es gibt viele Möglichkeiten, ein und dieselbe Datenstruktur zu implementieren. Beispielsweise könnten wir einen Stack auch durch eine verkettete Liste statt einem Array implementieren. Datenstrukturen hängen nicht von Programmiersprachendetails ab.

- Man kann eine Datenstruktur zu Hilfe nehmen, um eine andere zu implementieren. Datenstrukturen sind ja vor allem durch ihre Zugriffsoperationen bestimmt, und die lassen sich anpassen.
- Zugriffsoperationen werden durch Algorithmen festgelegt. Im Stack-Beispiel sind die Zugriffsoperationen sehr einfach, sodass von den Algorithmen nicht viel zu erkennen ist. Aber wir werden noch andere Beispiele sehen, wo Zugriffsoperationen durch komplexere Algorithmen beschrieben werden. Diese Algorithmen sind typisch für bestimmte Datenstrukturen. Sie bestimmen deren Eigenschaften.

Datenstrukturen und Algorithmen hängen stark voneinander ab. Algorithmen setzen bestimmte Datenstrukturen voraus, sodass Algorithmen nur zusammen mit den Datenstrukturen entwickelt werden können. Die Auswahl geeigneter Datenstrukturen ist in der Regel wichtiger als die der Algorithmen, weil über ein und dieselbe Datenstruktur mehrere Algorithmen ausgeführt werden müssen. Meist setzen wir ja dieselben Daten für mehrere Zwecke ein. Beim Entwickeln von Algorithmen ist daher Vorsicht angebracht: Wenn man eine Datenstruktur zu sehr an einen Algorithmus anpasst, sodass sie nur für diesen einen Algorithmus gut geeignet ist, dann ist es unter Umständen schwierig, andere auf dieselbe Datenstruktur angewiesene Algorithmen zu entwickeln.

4.1.3 Lösungsstrategie

Unter einer *Strategie* versteht man das langfristig orientierte Vorgehen in grundlegenden Fragen. Dabei sollen grundsätzliche, für den Erfolg entscheidende Ziele erreicht werden, sogenannte *strategische Ziele*.

Im Zusammenhang mit der Programmkonstruktion ist vor allem ein strategisches Ziel von überragender Bedeutung: *Einfachheit*. Wir stehen oft einem hohen Grad an Komplexität gegenüber, einerseits wegen der unvermeidbaren inhaltlichen Komplexität der zu lösenden Aufgaben, andererseits aber auch, weil sogar Programme zur Lösung einfacher Aufgaben dazu tendieren, im Laufe der Zeit immer umfangreicher, komplizierter und undurchschaubarer zu werden. Vor allem gegen Letzteres müssen wir ankämpfen. Nur wenn wir die Strukturen auf Dauer einfach halten, haben wir eine Chance, inhaltlich komplexe Aufgaben in den Griff zu bekommen. Das gilt auf allen Ebenen. Sowohl die Gesamtstruktur des Programms als auch einzelne Algorithmen und Datenstrukturen sollen einfach bleiben.

In diesem Kapitel werden wir folgende Strategien betrachten, die alle eine Vereinfachung eines Systems bzw. von Algorithmen und Datenstrukturen zum Ziel haben:

Teile und Herrsche. Lösungsansätze für inhaltlich komplexe Aufgaben sind oft nur schwer zu finden. Um die Suche zu beschleunigen, nehmen wir manchmal an, dass bestimmte vereinfachende Eigenschaften erfüllt sind und suchen nach einer Lösung unter diesen Annahmen. Dann sorgen wir dafür, dass die Annahmen erfüllt werden.

Top down. Wir gehen streng hierarchisch vor und konstruieren ein System anfangs nur auf einer sehr allgemeinen, abstrakten Ebene. Erst wenn wir diese Ebene im Detail verstanden haben, behandeln wir die einzelnen Teile des Systems auf gleiche Weise, bis wir ganz unten angelangt sind und alle Teile implementiert haben.

Bottom up. Manchmal ist die genau entgegengesetzte Strategie sinnvoll: Wir beginnen damit, wahrscheinlich benötigte Programmteile auf unterster Ebene zu implementieren und arbeiten uns langsam in Richtung höherer, abstrakterer Ebenen vor.

Schrittweise Verfeinerung. Wenn die Komplexität vor allem durch den großen Umfang einer Aufgabe bestimmt ist, geht man oft so vor, dass man anfangs nur einen kleinen Teil der Aufgabe löst. Danach ergänzt man die Lösung Schritt für Schritt um die fehlenden Teile.

Verwendung vorgefertigter Teile. Für häufig wiederkehrende Aufgaben gibt es fertige Lösungen, die man direkt verwenden kann.

So überzeugend das Streben nach Einfachheit als wichtigstem Ziel ist, so schwierig ist es in der Praxis manchmal, dieses Ziel im Auge zu behalten. Das hat mehrere Ursachen:

- Während man sich mit einem einzelnen Algorithmus in einem großen System beschäftigt, konzentriert man sich allzuleicht nur auf die Effizienz dieses einen Algorithmus und übersieht die Zusammenhänge mit dem großen Ganzen. Sowohl die Einfachheit als auch die Effizienz des Gesamtsystems kann darunter leiden.
- Während des Programmierens besteht die Gefahr der falschen Einschätzung der Komplexität. Man glaubt, eine einfache, effiziente Lösung gefunden zu haben, aber andere Personen verstehen diese Lösung nicht, und nach einiger Zeit versteht man sie selbst nicht mehr.

Diese Gefahr ist besonders groß, wenn es sich um eine trickreiche Lösung handelt, auf die man anfangs besonders stolz ist.

- Die offensichtlichsten Algorithmen und Datenstrukturen sind nicht immer die einfachsten und effizientesten. Häufig ist es so wie in den Beispielen in Abschnitt 4.1.1, dass einfachere und effizientere Algorithmen und Datenstrukturen mehr Wissen erfordern als komplexere. Daher kann Einfachheit den Entwicklungsaufwand erhöhen.

Die Vermeidung der ersten beiden Ursachen kostet viel Überwindung. Man muss sich dazu durchringen, für manche Teile der Aufgabe eine vermeintlich gute Lösung gegen eine weniger schöne auszutauschen, um einem in diesem Augenblick unbeständig erscheinenden strategischen Ziel näher zu kommen. Langfristig zahlt es sich aber aus, bewährten Strategien zu folgen und übertriebenes Effizienzdenken auf der Detailebene aufzugeben.

4.2 Rekursive Datenstrukturen und Methoden

Fast alle größeren Datenstrukturen sind *rekursiv* definiert. Das bedeutet, dass der Name der Datenstruktur innerhalb der Definition der Datenstruktur (direkt oder indirekt) vorkommt, genauso wie eine rekursive Methode innerhalb der Methode (direkt oder indirekt) aufgerufen wird. Algorithmen auf rekursiven Datenstrukturen sind meist auf natürliche Weise durch rekursive Methoden ausdrückbar. Zu jeder rekursiven Methode gibt es auch eine äquivalente nicht-rekursive (also iterative) Methode. Im Gegensatz dazu sind rekursive Datenstrukturen kaum durch nicht-rekursive Datenstrukturen ersetzbar. Wir betrachten rekursive Datenstrukturen anhand einer verketteten Liste und eines binären Baumes.

4.2.1 Verkettete Liste

Listing 4.4 zeigt wesentliche Teile der Implementierung einer verketteten Liste, die ganze Zahlen enthält. Wir verwenden zwei Klassen. Eine davon (`IntList`) repräsentiert die Datenstruktur mit allen öffentlich sichtbaren Zugriffsoptionen nach außen. Die andere (`IntListNode`) stellt die eigentliche rekursive Datenstruktur dar, wird aber nur innerhalb von `IntList` benötigt und ist außer durch `IntList` nicht direkt verwendbar; daher ist diese Klasse genauso wie ihre Methoden nicht `public`.

Listing 4.4: Verkettete Liste ganzer Zahlen

```

1 public class IntList {                // Liste ganzer Zahlen
2     private IntListNode head = null; // Listenkopf = 1. Element
3     public void add (int elem) {      // füge elem am Anfang ein
4         head = new IntListNode (elem, head);
5     }
6     public boolean contains (int elem) { // elem in Liste?
7         return head != null && head.contains(elem);
8     }
9     public void remove (int elem) {
10        // Löschen des ersten Vorkommens von elem aus der Liste
11        // Liste bleibt unverändert, wenn elem nicht vorkommt
12        if (head != null) {
13            head = head.remove(elem); // neuer Anfang nach Löschen
14        }
15    }
16 }
17
18 class IntListNode {                  // Listenknoten, verwendet von IntList
19     private int elem;                // das eigentliche Listenelement
20     private IntListNode next;        // nächster Knoten = Listenrest
21     IntListNode (int elem, IntListNode next) {
22         this.elem = elem;
23         this.next = next;
24     }
25     boolean contains (int e) { /*suche e in Restliste */ ... }
26     IntListNode remove (int e) { /*lösche e aus Restliste*/ ... }
27 }

```

Man erkennt an der Variablen `next` in `IntListNode` (Zeile 20), dass diese Klasse eine rekursive Datenstruktur darstellt: Der Typ der Variablen entspricht dem Namen der Klasse, in der die Variable definiert ist. Damit definiert `IntListNode` eine potentiell unendlich große Datenmenge: Eine Instanz der Klasse enthält in der Variablen `next` eine weitere Instanz der Klasse, diese enthält ebenso eine Instanz, diese wieder eine, und so weiter. Auf den ersten Blick scheint es so, als ob wir gar keine Instanz von `IntListNode` erzeugen könnten, da wir dafür schon eine existierende Instanz derselben Klasse benötigen würden. Es geht aber trotzdem, da wir in Java statt einer Instanz einer Klasse auch `null` verwenden können. Wir erzeugen also eine erste Instanz von `IntListNode`, in der `next` den Wert `null` enthält. Danach erzeugen wir ein Instanz, in der `next` die erste Instanz enthält, und so weiter. Rekursive Datenstrukturen beschreiben

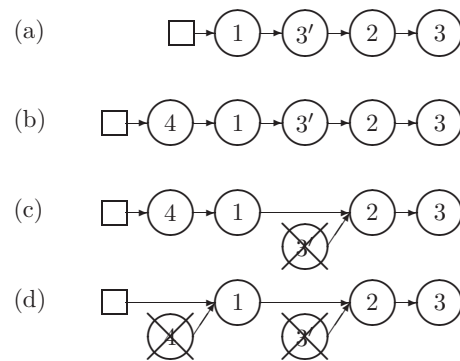


Abbildung 4.5: Grafische Darstellung einiger Listenoperationen

also beliebig große Datenmengen, aber keine unendlich großen, die im Speicher eines Computers ja niemals Platz haben würden.

Abbildung 4.5 zeigt schematisch einige verkettete Listen. Ein Kreis entspricht einer Instanz von `IntListNode`, die Zahl im Kreis dem Inhalt der Variablen `elem`, und ein von einem Kreis ausgehender Pfeil dem Inhalt von `next`. Ohne ausgehenden Pfeil enthält die Variable den Wert `null`. Auf Instanzen von `IntListNode` greifen wir über Instanzen von `IntList` zu, die durch kleine Kästchen symbolisiert sind. Die Variable `head` in jeder Instanz von `IntList` enthält den *Listenkopf*, das ist die Instanz von `IntListNode`, mit der die Liste beginnt. Ein von einem Kästchen ausgehender Pfeil zeigt auf den Inhalt dieser Variablen.

Eine solche Form der Darstellung kennen wir aus der Mathematik und verwenden wir oft in der Informatik. Es handelt sich um einen *gerichteten Graphen*. Kreise und Kästchen stellen verschiedene Arten von *Knoten* des Graphen dar (daher der Name `IntListNode`), Pfeile die *Kanten*. Praktisch alle rekursiven Datenstrukturen lassen sich durch gerichtete Graphen veranschaulichen. Eigenschaften dieser Graphen entsprechen auch Eigenschaften der Datenstrukturen. Eine Liste wird immer durch einen zusammenhängenden (das heißt, alle Knoten sind durch Kanten miteinander verbunden) gerichteten Graphen symbolisiert, in dem von jedem Knoten höchstens eine Kante ausgeht. In unserer speziellen Variante enthält jede Liste genau ein Kästchen am Anfang, und jeder Kreis enthält eine Zahl.

Operationen auf einer Liste dürfen die Listeneigenschaften nicht verletzen. Von `IntList` werden nur drei einfache Operationen unterstützt – siehe Listing 4.4 mit Methodenimplementierungen in Listing 4.6:

- `add` erzeugt einen neuen Listenknoten und fügt ihn ganz am Anfang ein, vor eventuell bereits vorhandenen Listenknoten. Der Graph (b) in Abbildung 4.5 wird beispielsweise durch Einfügen von 4 in die durch den Graphen (a) dargestellte Liste erzeugt. Die Reihenfolge des Einfügens bleibt erhalten. Liste (a) kann so entstanden sein:

```
IntList x = new Intlist();
x.add(3); x.add(2); x.add(3); x.add(1);
```

Die Zahl 3 wurde doppelt eingefügt, wobei die zuletzt eingefügte Zahl 3 zwecks Unterscheidbarkeit in Abbildung 4.5 durch 3' markiert ist.

- `contains` sucht eine Zahl in der Liste und gibt einen Wahrheitswert zurück, der besagt, ob die gesuchte Zahl mindestens einmal in der Liste enthalten ist. Der wesentliche Teil der Implementierung in Listing 4.6 wandert rekursiv über die Listenknoten, bis der gesuchte Eintrag gefunden ist oder keine weiteren Knoten mehr vorhanden sind. Dabei kommen die Kurzschlussoperatoren `&&` und `||` zum Tragen: Der rechte Teilausdruck wird nur ausgewertet, wenn die Auswertung des linken `true` (bei `&&`) bzw. `false` (bei `||`) ergibt.
- `remove` löscht den ersten Knoten aus der Liste, der die als Argument übergebene Zahl enthält, falls es einen solchen Knoten gibt. Diese in `IntList` implementierte Methode gibt kein Ergebnis zurück. Die entsprechende Methode in `IntListNode` (siehe Listing 4.6) liefert als Ergebnis den Rest der Liste nach dem Löschen. Das erleichtert die rekursive Implementierung: Wir brauchen nur Knoten für Knoten über die Liste zu wandern und, wenn wir den gesuchten Knoten gefunden haben, dessen Nachfolger zurückgeben; sonst kommt der Knoten selbst (also `this`) zurück. Anschließend setzen wir in jedem besuchten Knoten die Variable `next` auf das Ergebnis des rekursiven Aufrufs. Entsprechend muss auch `remove` in `IntList` die Variable `head` auf das Ergebnis des Aufrufs von `remove` in `IntListNode` setzen. Die Liste (c) in Abbildung 4.5 zeigt, was passiert, wenn wir in der Liste (b) `remove(3)` aufrufen: Es folgen Aufrufe gleichnamiger Methoden in den ersten drei Knoten (4, 1 und 3'), wobei der letzte Aufruf den Knoten mit 2 zurückgibt, der dann in der Variablen

Listing 4.6: Rekursives Suchen und Löschen auf verketteter Liste

```
// Diese Methoden stehen in der Klasse IntListNode
boolean contains (int e) { // suche e rekursiv in Restliste
    return elem == e || (next != null && next.contains(e));
}

IntListNode remove (int e) { // lösche e aus Rest, Anfang = this
    // Ergebnis ist Rest nach Löschen
    if (elem == e) {
        return next; // wenn this zu löschen ist:
        // neuer Rest = next (ohne this)
    } else if (next != null) { // sonst bei weiteren Knoten:
        next = next.remove(e); // Lösche e aus übriger Liste
    } // und next = neue übrige Liste
    return this; // neue Restliste beginnt mit this
}
```

next des Knotens 1 abgelegt wird, womit 2 auf 1 folgt. Die anderen Aufrufe geben `this` zurück, wodurch die ersten Knoten unverändert bleiben. Der Knoten 3' ist somit aus der Liste entfernt, obwohl er noch immer existiert und den Knoten 2 enthält. Über die Liste ist der Knoten 3' nicht mehr zugreifbar. Da er auch über keinen anderen Weg zugreifbar ist, wird er vom Java-System irgendwann aus dem Speicher entfernt. Auf ähnliche Weise entsteht der Graph (d) durch Löschen von 4 aus (c). Hier wird jedoch der erste Knoten gelöscht, sodass `head` in `IntList` auf den Nachfolgeknoten 1 gesetzt wird.

4.2.2 Rekursion versus Iteration

Am Beispiel der verketteten Liste können wir sehen, wie rekursive Datenstrukturen oft auf natürliche Weise rekursive Implementierungen der Zugriffsoperationen ergeben. Im Gegensatz zu rekursiven Datenstrukturen sind rekursive Methoden immer durch nicht-rekursive Methoden ersetzbar, die statt der Rekursion eine Schleife (also Iteration) verwenden. Die Methoden in Listing 4.7 entsprechen denen in Listing 4.6, vermeiden jedoch Rekursion. Im direkten Vergleich ist sofort zu erkennen, dass die rekursiven Methoden kürzer und einfacher sind als die nicht-rekursiven, obwohl sie dieselben Algorithmen implementieren. Analoge Beobachtungen machen wir oft. Trotzdem wird Rekursion von Personen mit wenig

Listing 4.7: Iteratives Suchen und Löschen auf verketteter Liste ganzer Zahlen

```
// Diese Methoden stehen in der Klasse IntListNode
boolean contains (int e) { // suche e iterativ in Restliste
    IntListNode node = this; // der gerade durchsuchte Knoten
    do { // Schleife über Knoten:
        if (node.elem == e) { // wenn gesuchte Zahl gefunden:
            return true; // beende erfolgreiche Suche
        }
        node = node.next; // sonst suche in nächstem Knoten
    } while (node != null); // solange noch Knoten vorhanden
    return false; // e nicht in Liste vorhanden
}

IntListNode remove (int e) { // lösche e aus Liste (iterativ)
    if (elem == e) { // wenn 1. Element zu löschen:
        return next; // Ergebnis ist Rest (ohne this)
    } // sonst lösche e aus Restliste
    IntListNode node = this; // betrachte Nachfolger
    while (node.next != null) { // solange Nachfolger da:
        if (node.next.elem == e) { // Nachfolger zu löschen:
            node.next = node.next.next; // hänge Nachfolger aus
            return this; // und beende Löschen
        } // (gleicher Anfang)
        node = node.next; // sonst nächster Knoten
    }
    return this; // Listenanfang bleibt unverändert
}
```

Programmiererfahrung häufig als schwierig empfunden. Wir wollen die Unterschiede zwischen Rekursion und Iteration daher näher betrachten.

Die iterativen Methoden brauchen zusätzliche lokale Variablen, um in einer Schleife über den Knoten der Liste stets zu wissen, welcher Knoten gerade betrachtet wird. In den rekursiven Methoden übernimmt `this` diese Aufgabe. Alleine schon durch diesen Unterschied sind iterative Methoden deutlich länger. In `contains` kommt hinzu, dass wir statt ganz einfacher Kurzschlussoperatoren aufwendigere bedingte Anweisungen verwenden müssen. In `remove` wird der iterative Code aus zwei anderen Gründen länger: Einerseits müssen wir das Löschen des ersten Knotens anders behandeln als das Löschen eines weiteren Knotens, da im ersten Fall `remove` in `IntList` einen Teil zu erledigen hat, während im zweiten Fall `remove` in `IntListNode` alleine dafür zuständig ist. Dazu brauchen wir eine Fallunterscheidung. Andererseits können wir im zweiten Fall die

einfache Technik, durch die wir den neuen Listenrest als Ergebnis eines rekursiven Aufrufs bekommen, nicht anwenden. Stattdessen müssen wir in der Schleife stets vorausblicken, um den für das Löschen nötigen Vorgängerknoten nicht zu verlieren. Beispielsweise müssen wir die gesuchte Zahl mit `node.next.elem` vergleichen, nicht einfach nur mit `node.elem`. Das vergrößert den Code, verschlechtert die Lesbarkeit und erhöht die Anzahl der Speicherzugriffe. Manchmal benötigt man sogar einen zusätzlichen Stack, um Rekursion durch Iteration zu ersetzen (Beispiel in Abschnitt 4.5.4).

Ein Nachteil rekursiver Varianten besteht in der hohen Anzahl an Methodenaufrufen. Jeder Aufruf kostet etwas Zeit und Speicherplatz. Damit wird oft begründet, warum man iterative Varianten vorzieht. Diese Argumentation trifft nur selten zu, da der zusätzliche Ressourcenbedarf für Aufrufe kaum ins Gewicht fällt, aber iterative Varianten durch komplizierteren Code oft einen höheren Ressourcenbedarf haben. In Abschnitt 4.3.1 werden wir die Zusatzkosten der Rekursion abschätzen lernen.

Rekursive Methoden und rekursive Datenstrukturen haben viele Gemeinsamkeiten mit vollständiger Induktion. Diese mathematische Beweismethode beruht auf den natürlichen Zahlen. Wir beweisen, dass eine Aussage für die Zahl 1 (oder 0) gilt. Wenn diese Aussage unter der Annahme, dass sie für eine beliebige natürliche Zahl n gilt, auch für $n + 1$ gilt, dann gilt sie tatsächlich für jede natürliche Zahl n . Als Beispiel wollen wir einen Beweis für die gaußsche Formel $\sum_{i=1}^n i = \frac{n \cdot (n + 1)}{2}$ zeigen:

Induktionsanfang: Für $n = 1$ stimmt die Formel: $\sum_{i=1}^1 i = 1 = \frac{1 \cdot (1 + 1)}{2}$

Induktionsschritt: Unter der Annahme, dass die Formel für n gilt, muss sie auch für $n + 1$ gelten. Wir müssen $\sum_{i=1}^{n+1} i = \frac{(n + 1) \cdot (n + 2)}{2}$ zeigen:

$$\sum_{i=1}^{n+1} i = \left(\sum_{i=1}^n i \right) + n + 1 = \frac{n \cdot (n + 1)}{2} + n + 1 = \frac{n \cdot (n + 1) + 2 \cdot (n + 1)}{2}$$

Da die Formel für den Induktionsanfang und den Induktionsschritt gilt, gilt sie für jede natürliche Zahl n .

Der Induktionsanfang spielt eine wichtige Rolle. Nur wenn es eine Basis gibt, auf die wir aufbauen können, ist auch der Induktionsschritt sinnvoll.

Auch für rekursive Datenstrukturen brauchen wir eine Basis. Beispielsweise können wir Listeneigenschaften folgendermaßen sicherstellen:

Induktionsanfang: Die vom Konstruktor erzeugte leere Liste erfüllt alle Listeneigenschaften. Das müssen wir überprüfen.

Induktionsschritt: Unter der Annahme, dass x eine Liste ist, die alle Listeneigenschaften erfüllt, muss x auch nach Ausführung jeder beliebigen Listenoperation eine Liste sein und alle Listeneigenschaften erfüllen. Auch das müssen wir für jede Listenoperation überprüfen.

Unter diesen Bedingungen wissen wir, dass die Listeneigenschaften stets erfüllt sind. Aufgrund der Einfachheit der Listeneigenschaften ist leicht zu sehen, dass sie erfüllt sind: Von jedem Knoten kann immer nur eine Kante ausgehen, da keine weiteren Variablen dafür vorhanden sind. Nur die Bedingung, dass der Graph zusammenhängend ist, erfordert Achtsamkeit. Bei jeder Änderung der Liste müssen wir sicherstellen, dass Listenteile nicht unabsichtlich verloren gehen.

Ein gutes Verständnis der vollständigen Induktion ist sehr hilfreich im Umgang mit rekursiven und iterativen Methoden. Formal können wir über vollständige Induktion beweisen, dass nach Beendigung eines Aufrufs für alle betrachteten Datenelemente bestimmte Eigenschaften erfüllt sind. Beispielsweise soll `remove` eine Liste unverändert lassen, wenn das zu löschende Element nicht enthalten ist, und wenn ein solches Element vorhanden ist, soll die Liste um genau einen Knoten kürzer werden. Mit etwas Erfahrung stellen wir beim Programmieren entsprechende Überlegungen an, auch ohne Notwendigkeit für einen formalen Beweis.

Ein Beweis dafür, dass `remove` eine Liste unverändert lässt, könnte etwa so aussehen:

Induktionsanfang: Wenn die Liste leer ist, sind die Bedingungen trivialerweise erfüllt, da `remove` in `IntListNode` gar nicht aufgerufen wird. Zur Überprüfung aller Teile der Bedingung müssen wir einen etwas komplexeren Anfang wählen: Die Bedingungen sind auch erfüllt, wenn die Liste genau ein Element enthält. Das sieht man durch Betrachtung aller möglichen Fälle, die dabei auftreten können.

Induktionsschritt: Unter der Annahme, dass die Bedingungen für eine Liste mit n Elementen (wobei $n \geq 1$) erfüllt ist, müssen sie auch für eine Liste mit $n + 1$ Elementen erfüllt sein. Auch das ist durch Betrachtung aller möglicher auftretender Fälle leicht zu sehen.

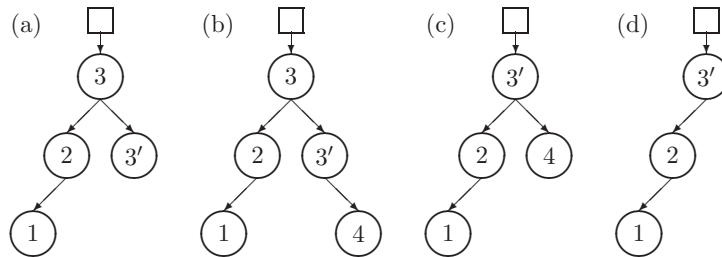


Abbildung 4.8: Grafische Darstellung einiger Baumoperationen

Für Aufrufe rekursiver Methoden und Schleifendurchläufe in iterativen Methoden müssen wir dieselben Überprüfungen anstellen. Das verdeutlicht die prinzipielle Übereinstimmung zwischen rekursiven und iterativen Methoden, abgesehen von Details.

4.2.3 Binärer Baum

Die Suche in einer verketteten Liste kann lange dauern, vor allem wenn das gesuchte Element nicht vorhanden ist. Dann müssen alle Listenknoten durchsucht werden. Bäume können diesen Aufwand reduzieren: Jeder Baumknoten kann mehrere Nachfolgeknoten haben, in binären Bäumen bis zu zwei. Entsprechend dem Wert des gesuchten Eintrags wird die Suche mit dem einen oder dem anderen Nachfolgeknoten fortgesetzt. So muss in den meisten Fällen nur eine kleine Auswahl an Knoten durchsucht werden, um festzustellen, ob ein Wert enthalten ist oder nicht.

Abbildung 4.8 zeigt eine schematische Darstellung einiger binärer Bäume. Traditionell wachsen Bäume in der Informatik von oben nach unten. Den obersten durch einen Kreis dargestellten Knoten nennt man die *Wurzel* des Baums. Knoten, die keine Nachfolgeknoten haben, nennt man *Blätter*. Im Baum (a) enthält die Wurzel den Wert 3 und die Blätter die Werte 1 und 3'. Jeder Knoten im Baum ist Wurzel eines *Teilbaums*. Beispielsweise ist der Knoten 2 im Baum (a) Wurzel des Teilbaums bestehend aus den Knoten 2 und 1. Dieser Teilbaum ist der linke Teilbaum unter dem Knoten 3, während der rechte Teilbaum nur aus dem Knoten 3' besteht.

Listing 4.9: Binärer Baum ganzer Zahlen

```

1 public class IntTree {                                // binärer Baum
2     private IntTreeNode root = null;                 // Wurzel des Baums
3     public void add (int e) {                         // füge e in Baum ein
4         if (root == null) {                          // wenn Baum noch leer:
5             root = new IntTreeNode(e);               // Wurzel = 1. Knoten
6         } else {                                     // wenn Baum nicht leer:
7             root.add(e);                             // füge in Wurzel ein
8         }
9     }
10    public boolean contains (int elem) { // ist elem im Baum?
11        return root != null && root.contains(elem);
12    }
13    public void remove (int elem) { // lösche ein elem
14        if (root != null) { // wenn Baum nicht leer:
15            root = root.remove(elem); // lösche aus Wurzel
16        }
17    }
18 }
19
20 class IntTreeNode { // Knoten im Baum
21     private int elem; // eigentliches Element
22     private IntTreeNode left = null; // linker Teilbaum
23     private IntTreeNode right = null; // rechter Teilbaum
24     IntTreeNode (int e) {
25         elem = e;
26     }
27     void add (int e) { /* füge e in Baum ein */ ... }
28     boolean contains (int e) { /* suche e im Baum */ ... }
29     IntTreeNode remove (int e) { /* lösche e aus Baum */ ... }
30 }

```

Ein nicht-leerer binärer Baum hat folgende Eigenschaften: Er ist ein zusammenhängender gerichteter Graph, in dem jeder kreisförmige Knoten mit einem *Label* (in unserem Beispiel einer Zahl) versehen ist und genau eine eingehende sowie höchstens je eine linke und rechte ausgehende Kante hat. Dabei haben wir auch die vom Kästchen kommende Kante gezählt; wenn wir diese Kante ignorieren, hat die Wurzel keine eingehende Kante. Für den binären Baum und jeden seiner Teilbäume gilt, dass

- die Wurzel des Baums bzw. Teilbaums eindeutig bestimmt ist,
- jeder Knoten im linken Teilbaum unter dieser Wurzel ein Label hat, das kleiner dem Label der Wurzel ist (z.B. sind 1 und 2 kleiner 3),

Listing 4.10: Rekursives Einfügen in einen binären Baum

```

void add (int e) { // Klasse IntTreeNode; füge e in Baum ein
    if (e < elem) { // wenn e in linken Teilbaum soll:
        if (left != null) { // wenn linker Teilbaum existiert:
            left.add(e); // füge e in linken Teilbaum ein
        } else { // wenn kein linker Teilbaum:
            left = new IntTreeNode(e); // erzeuge neuen Teilbaum
        }
    } else /* e >= elem */ { // e soll in rechten Teilbaum
        if (right != null) { // wenn rechter Teilbaum da:
            right.add(e); // füge e rechts ein
        } else { // kein rechter Teilbaum:
            right = new IntTreeNode(e); // erzeuge neuen Teilbaum
        }
    }
}
}

```

Listing 4.11: Iteratives Einfügen in einen binären Baum

```

void add (int e) { // Klasse IntTreeNode; füge e in Baum ein
    IntTreeNode node = this; // betrachteter Knoten
    while(true) { // endlos wiederholt:
        if (e < node.elem) { // e soll nach links
            if (node.left != null) { // wenn Teilbaum da:
                node = node.left; // weiter mit Teilbaum
            } else { // kein Teilbaum da:
                node.left = new IntTreeNode(e); // neuer Teilbaum
                return;
            }
        } else /* e >= node.elem */ { // e soll nach rechts
            if (node.right != null) { // wenn Teilbaum da:
                node = node.right; // weiter mit Teilbaum
            } else { // kein Teilbaum da:
                node.right = new IntTreeNode(e); // neuer Teilbaum
                return;
            }
        }
    }
}
}

```

- und jeder Knoten im rechten Teilbaum unter dieser Wurzel ein Label größer oder gleich dem Label der Wurzel hat.

Listing 4.12: Rekursive Suche in einem binären Baum

```

boolean contains (int e) { // IntTreeNode; suche e im Baum
    if (e < elem) { // e vielleicht im linken Teilbaum
        return left != null && left.contains(e);
    } else if (e == elem) { // e gefunden
        return true;
    } else /* e > elem */ { // e vielleicht im rechten Teilbaum
        return right != null && right.contains(e);
    }
}

```

Listing 4.13: Iterative Suche in einem binären Baum

```

boolean contains (int e) { // IntTreeNode; suche e im Baum
    IntTreeNode node = this; // betrachteter Knoten
    do { // wiederhole:
        if (e < node.elem) { // wenn e vielleicht links:
            node = node.left; // links weiter
        } else if (e == node.elem) { // wenn e gefunden:
            return true; // Suche erfolgreich
        } else /* e > node.elem */ { // wenn e eventuell rechts:
            node = node.right; // rechts weiter
        }
    } while (node != null); // solange Knoten vorhanden
    return false; // e nicht gefunden
}

```

Listing 4.9 zeigt Teile der Implementierung eines binären Baums über ganzen Zahlen. Wie bei der Liste benötigen wir zwei Klassen, eine für die externe Darstellung und eine für Baumknoten. Es werden auch in etwa dieselben Listenoperationen unterstützt: Einfügen, Suchen und Löschen. Um die komplexeren Eigenschaften eines binären Baumes zu erzielen, müssen in den Implementierungen dieser Operationen jedoch viel mehr Fallunterscheidungen getroffen werden als in Listen.

Listing 4.10 zeigt eine rekursive Implementierung der Einfügeoperation `add` in `IntTreeNode` und Listing 4.11 eine entsprechende iterative Implementierung. Ein neuer Knoten kann nur als Blatt in den Baum eingehängt werden. Wir suchen eine geeignete Stelle, indem wir an der Wur-

zel beginnend entsprechend dem Label im linken oder rechten Teilbaum weitersuchen, bis wir eine freie Stelle gefunden haben. Der Baum (b) in Abbildung 4.8 entsteht durch Einfügen von 4 in den Baum (a). Da 4 größer oder gleich 3 ist, wird bei der Suche nach einem freien Platz stets der rechte Teilbaum gewählt.

Man sieht deutlich, dass bei der Suche nach einem freien Platz nur ein Teil des Baums betrachtet werden muss. Je mehr Knoten ein Baum hat, desto kleiner ist der Anteil der Knoten, der üblicherweise zu betrachten ist. Der Baum (a) könnte dadurch entstanden sein, dass zuerst 3 (die Wurzel) eingefügt wurde, dann 2, dann 3' und schließlich 1. Die Reihenfolge könnte aber auch 3, 3', 2, 1 oder 3, 2, 1, 3' gewesen sein. Im Gegensatz zur verketteten Liste ist die Reihenfolge der Einfügungen in einen binären Baum nicht erkennbar.

Die Listings 4.12 und 4.13 zeigen Implementierungen der Suche im binären Baum. Die Suche nach einem bestimmten Knoten ähnelt der Suche nach einem freien Platz beim Einfügen. Da nur ein Teil des Baums durchsucht werden muss, ist die Suche im binären Baum meist deutlich effizienter als die Suche in einer Liste. Es gibt aber auch entartete Bäume wie (d) in Abbildung 4.8, wo der Baum die Form einer Liste hat. In solchen (bei großen Bäumen sehr unwahrscheinlichen) Fällen dauert die Suche wegen zusätzlicher Größenvergleiche länger als in einer Liste.

Das Löschen eines Knotens aus einem binären Baum ist eine recht aufwendige Operation, wie man an den Implementierungen in den Listings 4.14 und 4.15 erkennen kann. Am Anfang steht wiederum die Suche nach dem zu löschenden Knoten. Dieser kann zwei Teilbäume haben, die nach dem Löschen übrig bleiben. Der Vorgängerknoten kann aber nur einen Teilbaum aufnehmen. Beispielsweise passiert das, wenn wir aus dem Baum (b) in Abbildung 4.8 den Wurzelknoten 3 löschen wollen: Übrig bleibt ein Teilbaum mit den Knoten 2 und 1 und einer mit den Knoten 3' und 4, aber in der Variablen `root` in `IntTree` findet nur ein Baum Platz.

Es gibt mehrere Möglichkeiten, um mit diesem Problem umzugehen. Wir wählen eine einfach implementierbare Lösung: Wenn zwei Teilbäume übrig bleiben, fügen wir den linken Teilbaum an der so weit links wie möglich stehenden freien Stelle in den rechten Teilbaum ein; im Code in Listing 4.14 verwenden wir dafür eine eigene private Methode, in Listing 4.15 ist das Verschieben des Teilbaums in die Methode `remove` integriert. Im Beispiel wird der Teilbaum mit den Knoten 2 und 1 da-

Listing 4.14: Rekursive Suche mit Löschen in einem binären Baum

```
IntTreeNode remove (int e) { // lösche ein e aus Baum
    // Ergebnis: Wurzel nach dem Löschen
    if (e < elem) {           // e vielleicht im linken Teilbaum:
        if (left != null) {   // falls es linken Teilbaum gibt:
            left = left.remove(e); // neuer Teilbaum nach Löschen
        }
    } else if (e == elem) {   // this ist zu löschen:
        if (right == null) {  // falls kein rechter Teilbaum:
            return left;      // Ergebnis ist linker Teilbaum
        }
        if (left != null) {   // falls beide Teilbäume vorhanden
            right.addTree(left); // füge linken Teilb. rechts ein
        }                    // jetzt kein linker Teilbaum
        return right;         // Ergebnis ist rechter Teilbaum
    } else /* e > elem */ {   // e vielleicht im rechten Teilbaum:
        if (right != null) {  // falls es rechten Teilbaum gibt:
            right = right.remove(e); // Teilbaum ev. verändert
        }
    }
    return this;              // Wurzel des Baums unverändert
}

private void addTree (IntTreeNode t) { // füge Baum links ein
    if (left != null) {           // wenn es linken Teilbaum gibt:
        left.addTree(t);          // füge in linken Teilbaum ein
    } else {                      // Knoten am weitesten links:
        left = t;                 // hier soll der Baum hin
    }
}
```

durch zum linken Teilbaum des Knotens 3'. Übrig ist danach nur mehr ein Teilbaum, und der wird an die Stelle des gelöschten Knotens gesetzt. Baum (c) in Abbildung 4.8 zeigt das Ergebnis der Löschoperation ohne den gelöschten Knoten. Wenn der zu löschende Knoten keinen oder nur einen Teilbaum hat, ist die Löschoperation einfacher. Der entartete Baum (d) entsteht beispielsweise durch Löschen des Knotens 4 aus dem Baum (c), ohne dass dadurch Änderungen der Baumstruktur notwendig wären.

4.3 Algorithmische Kosten

Unterschiedliche Algorithmen zur Lösung desselben Problems haben unterschiedliche Eigenschaften. Vor allem brauchen Sie unterschiedlich viele

Listing 4.15: Iterative Suche mit Löschen in einem binären Baum

```

IntTreeNode remove (int e) {          // lösche ein e aus Baum
    IntTreeNode node = this;          // aktuell betrachteter Knoten
    IntTreeNode last = null;          // zuletzt betrachteter Knoten
    do {                               // wiederhole:
        if (e < node.elem) {           // e vielleicht links:
            last = node;               // merke Knoten
            node = node.left;          // gehe links weiter
        } else if (e == node.elem) {   // e gefunden, node löschen:
            IntTreeNode subst;         // subst ersetzt node
            if (node.right == null) {   // kein rechter Teilbaum:
                subst = node.left;      // linker Zw. statt node
            } else {                   // rechter Teilbaum da:
                subst = node.right;      // rechter statt node
                if (node.left != null) { // beide Zweige da:
                    IntTreeNode r = node.right;
                    while (r.left != null) { // suche Knoten
                        r = r.left;        // ganz links
                    }                       // und füge dort den
                    r.left = node.left;    // linken Zweig an
                }
            }
        }                               // (jetzt node ersetzen)
        if (last == null) {             // Wurzel ist zu ersetzen:
            return subst;               // neue Wurzel
        } else if (last.left == node) {
            last.left = subst;          // linker Vorgängerzweig
        } else /* last.right == node */ {
            last.right = subst;         // rechter Vorgängerzweig
        }
        return this;                   // Wurzel unverändert
    } else /* e > node.elem */ { // e vielleicht rechts:
        last = node;                   // merke Knoten
        node = node.right;              // gehe rechts weiter
    }
    } while (node != null);             // solange es Knoten gibt
    return this;                       // Wurzel unverändert
}

```

Ressourcen wie Laufzeit und Speicher. Den Ressourcenverbrauch eines Algorithmus bezeichnet man als dessen *Aufwand* oder *Kosten*. Es ist gar nicht einfach, die genauen Kosten eines Algorithmus anzugeben, da sie von zahlreichen Faktoren abhängen. Auf jedem Rechner, für jede Datenmenge und für jede Implementierung kann der Algorithmus unterschiedlich viele Ressourcen verbrauchen. Mit einem einfachen Messen des Ressourcenver-

brauchs ist es daher nicht getan. Wir beschäftigen uns hier mit der Frage, wie man die algorithmischen Kosten abschätzen kann, ohne Details der Hardware, der Programmiersprache, der Implementierung und der Datenmenge zu kennen.

4.3.1 Abschätzung algorithmischer Kosten

Kostenabschätzungen liefern keine genauen Werte, sondern nur ganz grobe Anhaltspunkte für die tatsächlichen Kosten. Diese Grundregel müssen wir immer im Auge behalten, wenn wir Kosten abschätzen. Es kommt nicht darauf an, ob wir eine Aufgabe mit 10, 100 oder 1000 Anweisungen lösen und dafür 5, 50, oder 500 Variablen brauchen. Solche Unterschiede sind vernachlässigbar klein, auch wenn es sich dabei um Vielfache handelt. Einige Implementierungen brauchen mehr Anweisungen als andere, einige Computer führen in derselben Zeiteinheit mehr Anweisungen aus als andere, und ein Compiler spart durch Optimierungen mehr Zeit ein als ein anderer. Für den Vergleich von Algorithmen spielen diese Unterschiede keine Rolle. Andererseits kommt es darauf an, ob eine Operation mit insgesamt 10 Anweisungen gelöst wird, oder mit 10 Anweisungen pro Datenelement in einer Datenstruktur. Wenn die Datenstruktur Millionen oder Milliarden von Datenelementen enthält, werden aus den 10 Anweisungen schnell zig Millionen oder Milliarden. Das ist nicht vernachlässigbar.

Die Kosten für das Einfügen einer Zahl in eine verkettete Liste wie in Listing 4.4 hängen nicht von der Anzahl der Elemente in der Liste ab. Man sagt, die Kosten für das Einfügen sind *konstant*, wobei wir die Anzahl der dafür nötigen Anweisungen und Variablen gar nicht zählen. Man sagt auch, die Kosten sind *von der Ordnung 1*, formal durch $O(1)$ bezeichnet.

Das Suchen eines Elements in der verketteten Liste verlangt dagegen, dass wir die Liste vom ersten Element bis zum gesuchten Element oder bis zum Ende durchwandern. Die Kosten hängen von der Anzahl der Elemente in der Liste und der Position ab, an der sich das gesuchte Element befindet. Im schlechtesten Fall muss die ganze Liste durchwandert werden. Wenn die Liste n Elemente enthält, ist der zeitliche Aufwand im schlechtesten Fall *von der Ordnung n* oder formal $O(n)$. Man sagt, die Kosten sind *linear* zur Anzahl der Listenelemente. Durchschnittlich müssen wir $n/2$ Knoten betrachten. Aber $1/2$ ist nur ein konstanter Faktor, den wir genauso wie die Anzahl der Anweisungen pro betrachtetem Knoten ignorieren. Daher verursacht die Suche auch im Durchschnitt zeitliche Kosten von $O(n)$. Zusätzlich haben wir noch einen konstanten Aufwand

zum Starten der Suche in `IntList`. Niedrigere (z.B. konstante) Kosten fallen aber im Vergleich zu höheren (z.B. linearen) Kosten nicht ins Gewicht und können vernachlässigt werden. Man sagt, die höheren Kosten *dominieren*. Das Löschen eines Elements aus der Liste wird von der Suche nach dem zu löschenden Element dominiert, sodass wir auch dafür sowohl im Durchschnitt als auch im schlechtesten Fall lineare Kosten haben.

Beim Einfügen eines Elements in einen binären Baum braucht nur ein Teil der vorhandenen Knoten betrachtet werden, bis wir die passende freie Stelle gefunden haben. Durch mathematische Analysen, auf die wir hier nicht näher eingehen, kann man zeigen, dass die zeitlichen Kosten dafür durchschnittlich *logarithmisch* zur Anzahl n der Knoten im Baum sind, also formal von $O(\log(n))$. Diese Kosten liegen zwischen $O(1)$ und $O(n)$. Allerdings können Bäume auch zu Listen entartet sein (so wie Baum (d) in Abbildung 4.8), sodass die Kosten im schlechtesten Fall so hoch sein können wie für die Suche in der verketteten Liste, also linear zur Anzahl der Knoten im Baum. Die Kosten für das Suchen im bzw. Löschen aus dem binären Baum sind auch so hoch wie für das Einfügen, nämlich durchschnittlich $O(\log(n))$ und im schlechtesten Fall $O(n)$. Das ist so, weil für alle diese Zugriffsoperationen die Suche nach einer bestimmten Stelle im Baum dominiert. In manchen Varianten von Bäumen (z.B. sogenannten AVL-Bäumen) wird beim Einfügen und Löschen zusätzlicher Aufwand betrieben, um das Entarten der Bäume zu vermeiden und dadurch die Kosten auch im schlechtesten Fall auf $O(\log(n))$ zu halten.

Konstante, logarithmische und lineare Kosten gelten in der Regel als recht niedrig. Viele Algorithmen haben quadratische ($O(n^2)$), kubische ($O(n^3)$) oder noch höhere Kosten. Aber alle *polynomialen* Kosten, das sind Kosten der Form $O(n^k)$ für alle beliebigen Zahlen k sind noch klein im Vergleich zu *exponentiellen* Kosten $O(2^n)$. Algorithmen mit exponentiellen Kosten werden häufig als beinahe unbrauchbar angesehen, da damit nur sehr kleine Probleme (mit sehr kleinem n) in vertretbarer Zeit und mit dem zur Verfügung stehenden Speicher lösbar sind. Beispielsweise sind $2^{32} = 4.294.967.296$ (etwa vier Milliarden) Operationen für manche Aufgaben gerade noch in vertretbarer Zeit ausführbar und ebenso viele Speicherzellen in neueren Computern gerade noch vorhanden, aber $2^{64} = 18.446.744.073.709.551.616$ Operationen bzw. Speicherzellen ziemlich sicher nicht mehr, obwohl 32 und 64 nur kleine Zahlen sind. Auch exponentielle Kosten sind noch lange nicht die höchsten vorstellbaren Kosten. Höhere als exponentielle Kosten sind zwar von theoretischer, aber kaum von praktischer Bedeutung.

Es ist ziemlich offensichtlich, dass verkettete Listen und binäre Bäume einen zur Anzahl der Knoten linearen Speicherbedarf haben. Der Speicherbedarf für einzelne Operationen ist dagegen für die iterativen Varianten konstant. In den rekursiven Varianten sind jedoch zusätzliche Kosten versteckt: Rekursive Ausführungen von Methoden existieren gleichzeitig und brauchen daher auch gleichzeitig Speicherplatz. Bei der rekursiven Suche im bzw. dem rekursiven Löschen aus einer verketteten Liste mit n Knoten haben wir gleichzeitig bis zu n rekursive Ausführungen, im Schnitt $n/2$. Aus diesem Grund sind auch die Speicherkosten linear, nicht nur die zeitlichen Kosten. Die zeitlichen Kosten der rekursiven Varianten sind dagegen von derselben Ordnung wie die der iterativen Varianten. Entsprechend sind die Speicherkosten für die rekursiven Varianten des Einfügens, der Suche und des Löschens im binären Baum im Durchschnitt logarithmisch und im schlechtesten Fall linear, genauso wie die zeitlichen Kosten, für die iterativen Varianten aber konstant. Verkompliziert wird die Kostenabschätzung dadurch, dass ein guter Compiler für manche Arten der Rekursion den zusätzlichen Platzbedarf eliminieren kann, sodass er nicht größer ist als der für entsprechende iterative Varianten.

Kostenabschätzungen liefern gute Vorhersagen für die *Skalierbarkeit*: Ein Algorithmus skaliert gut, wenn er für große Datenmengen ebenso einsetzbar ist wie für kleine. Er skaliert schlecht, wenn er zwar für kleine Datenmengen gut einsetzbar ist, aber nicht für große. Wenn wir wissen, wie groß die Datenmenge sein wird, auf die wir einen Algorithmus anwenden, bringt uns die Kostenabschätzung nichts; unter dieser Voraussetzung hat jeder Algorithmus konstante Kosten. Wie groß der konstante Aufwand ist, können wir durch Messen der Laufzeit bzw. des Speicherverbrauchs feststellen. Oft sind schlecht skalierende Algorithmen bei bekannter Datengröße effizienter als gut skalierende. Ein häufiger Fehler besteht darin, zum Vergleich von Algorithmen Messungen mit einer bestimmten Datenmenge vorzunehmen und den dabei effizientesten Algorithmus auch auf größere Datenmengen anzuwenden; die damit getroffene Wahl ist wahrscheinlich schlecht, da die Skalierbarkeit außer Acht gelassen wurde.

Der Ressourcenbedarf soll ausgewogen sein. Es bringt nichts, wenn der Speicherverbrauch klein bleibt, aber der zeitliche Aufwand zu groß wird, oder umgekehrt. Ein effizienter Programmteil bringt auch nichts, wenn die Ineffizienz anderer Teile nur kleine Datenmengen zulässt. Beispielsweise haben wir unter diesen Gesichtspunkten wahrscheinlich keinen Vorteil dadurch, dass wir rekursive durch iterative Varianten der Algorithmen ersetzen, da ohnehin andere Kosten dominieren.

Nicht nur für Algorithmen, sondern auch für Probleme kann man Aufwandsabschätzungen machen. Die Kosten eines Problems entsprechen den Kosten des effizientesten Algorithmus, der das Problem lösen kann.

4.3.2 Kosten im Zusammenhang

Bei der Konstruktion von Programmen haben wir viel Gestaltungsfreiheit. Wir können entscheiden, welche Operationen benötigt werden und wann welche Operationen auszuführen sind. Diese Freiheit macht es oft schwer, geeignete Datenstrukturen und Algorithmen zu wählen, da ein direkter Vergleich bei stark unterschiedlichen Ansätzen kaum möglich ist. Als Beispiel betrachten wir die sortierte Ausgabe aller Zahlen in einer Datenstruktur. Listing 4.16 zeigt Methoden, die als Teile von `IntList` und `IntListNode` den Inhalt einer Liste ausgeben bzw. sortieren. Um eine sortierte Ausgabe zu erhalten, müssen wir die Liste zuerst sortieren und dann die sortierte Liste ausgeben. Die Methoden in Listing 4.17, hinzugefügt zu `IntTree` und `IntTreeNode`, erlauben dagegen die sortierte Ausgabe, ohne vorher eine Methode zum Sortieren aufrufen zu müssen.

Der zeitliche Aufwand für `sort` in Listing 4.16 ist quadratisch: Wir laufen so oft vom Anfang der Liste bis zum Ende und vertauschen dabei die Elemente von je zwei benachbarten Knoten, wenn diese in der falschen Reihenfolge stehen, bis ein Durchlauf durch die Liste keine Änderung mehr bewirkt. Dieses Sortierverfahren nennt sich *Bubblesort*. Ein Listendurchlauf verursacht Kosten von $O(n)$. Im schlechtesten Fall muss das letzte Listenelement ganz an den Anfang wandern. Dafür sind n Listendurchläufe notwendig, was Kosten von $O(n^2)$ ergibt. Im Schnitt werden halb so viele Durchläufe gebraucht, das ist ebenso ein quadratischer Aufwand. Der Platzbedarf für das Sortieren ist dagegen konstant.

Der zeitliche Aufwand für `print` ist sowohl in der Liste als auch im Baum linear, da dabei jeweils jedes Element genau einmal besucht wird. Der Platzbedarf für `print` ist in der Liste konstant. Im Baum ist er jedoch wegen der rekursiven Implementierung im Durchschnitt logarithmisch (weil die Anzahl der überlappenden rekursiven Aufrufe von der Tiefe des Baums abhängt) und im schlechtesten Fall (entarteter Baum) linear.

Wir wollen nun ermitteln, wie hoch der Aufwand insgesamt ist, wenn wir eine Datenstruktur mit n Elementen aufbauen und die Elemente anschließend sortiert ausgeben. Die Kosten für das Einfügen eines Elementes

Listing 4.16: Ausgeben und Sortieren einer Liste

```
// Klasse IntList; gib alle Elemente der Liste aus
public void print () { // Ausgabe in Reihenfolge der Elemente
    if (head != null) {
        head.print();
    }
    System.out.println("fertig");
}

// Klasse IntList; sortiere alle Elemente der Liste aufsteigend
public void sort () {
    if (head != null) {
        head.sort();
    }
}

// Klasse IntListNode; hier passiert die eigentliche Ausgabe
void print () { // Ausgabe in Reihenfolge der Listenelemente
    IntListNode p = this;
    do {
        System.out.println(p.elem);
        p = p.next;
    } while (p != null);
}

// Klasse IntListNode; hier wird tatsächlich sortiert
void sort() { // Algorithmus ist "Bubblesort"
    boolean changed; // haben sich Änderungen ergeben?
    do {
        changed = false;
        IntListNode s = this; // smaller; sollte kleiner sein
        IntListNode l = next; // larger; sollte größer sein
        while (l != null) {
            if (l.elem < s.elem) { // Elemente zu vertauschen
                int i = s.elem;
                s.elem = l.elem;
                l.elem = i;
                changed = true;
            }
            s = l; // weiter mit nächstem s,l-Paar
            l = l.next;
        }
    } while (changed); // solange etwas geändert wurde
}
```

in die verkettete Liste sind konstant, die für das Einfügen von n Elementen daher sowohl hinsichtlich der Zeit als auch des Platzes $O(n)$. Anschließend

Listing 4.17: Sortierte Ausgabe aller Elemente in binärem Baum

```
// Klasse IntTree; gib alle Elemente aufsteigend sortiert aus
public void print() {
    if (root != null) {
        root.print();
    }
    System.out.println("fertig");
}

// Klasse IntTreeNode; hier passiert die eigentliche Arbeit
void print() {
    if (left != null) {
        left.print();          // gib alle kleineren Elemente aus,
    }
    System.out.println(elem); // dann das eigene
    if (right != null) {
        right.print();         // und schließlich alle größeren
    }
}
```

sortieren wir die Liste mit einem zeitlichen Aufwand von $O(n^2)$ und geben die Listenelemente mit einem zeitlichen Aufwand von $O(n)$ aus. Da es sich dabei nur um eine konstante Zahl von Schritten handelt, ergibt sich der Gesamtaufwand aus dem Schritt mit den dominierenden Kosten, das ist das Sortieren. Der gesamte zeitliche Aufwand bei Verwendung der Liste ist daher $O(n^2)$. Weil die Operationen nur konstanten Platzbedarf haben, dominiert die Liste selbst den Platzbedarf mit $O(n)$.

Wegen seiner Einfachheit haben wir in Listing 4.16 ein recht ineffizientes Sortierverfahren gewählt. Ein effizienteres Verfahren, z.B. Quicksort (siehe Abschnitt 4.4.2) mit einem durchschnittlichen Aufwand von $O(n \cdot \log(n))$ und $O(n^2)$ im schlechtesten Fall, kann den gesamten zeitlichen Aufwand im Schnitt auf $O(n \cdot \log(n))$ reduzieren. Auch mit dem effizientesten Sortierverfahren wird das Sortieren noch immer dominieren.

Verwenden wir statt der Liste einen Baum, können wir auf das Sortieren verzichten und brauchen nur n Elemente einzufügen und anschließend auszugeben. Der zeitliche Aufwand für das Einfügen eines Elements beträgt im Durchschnitt $O(\log(n))$ und im schlechtesten Fall $O(n)$. Das Einfügen von n Elementen hat den n -fachen zeitlichen Aufwand, das ist im Durchschnitt $O(n \cdot \log(n))$ und im schlechtesten Fall $O(n^2)$. Das ist auch der Gesamtaufwand, da er gegenüber dem linearen Aufwand für das Ausge-

ben dominiert. Hinsichtlich des Platzbedarfs dominiert auch bei Verwendung des Baums die Datenstruktur selbst mit $O(n)$, da für Operationen gleichzeitig nie mehr als der Platz für eine Einfügeoperation bzw. für die Ausgabe benötigt wird. Es spielt in der Gesamtbetrachtung also keine Rolle, ob wir für das Einfügen die iterative oder rekursive Variante wählen.

Wie das Beispiel zeigt, führt eine Aufwandsabschätzung nicht selten zu unerwarteten Ergebnissen. Obwohl wir bei Verwendung der Liste einen zusätzlichen aufwendigen Sortierschritt benötigen, ist der Aufwand genau gleich wie bei Verwendung des Baums im schlechtesten Fall. Wenn wir Bubblesort durch Quicksort ersetzen, kommt die Lösung mit der Liste auch im Durchschnitt auf genau dieselben Kosten wie die Lösung mit dem Baum. Durch Verwendung anderer Sortierverfahren und den Einsatz von AVL-Bäumen statt einfacher Bäume kommen beide Lösungsvarianten auch im schlechtesten Fall auf gleiche Kosten von $O(n \cdot \log(n))$.

Bei Betrachtung der beiden Lösungsvarianten im größeren Zusammenhang ergeben sich große Unterschiede. Wenn wir die Elemente beispielsweise auch in der Reihenfolge ausgeben müssen, in der sie eingefügt wurden, kommt nur die Lösung mit der Liste in Frage, da Bäume die Reihenfolge nicht erhalten können. Wenn hingegen häufig Elemente in der Datenstruktur gesucht werden müssen, ist die Lösung mit dem Baum deutlich effizienter. Beim Konstruieren von Programmen müssen wir also stets vorausblicken und überlegen, wie wir in Zukunft auf unsere Datenstrukturen zugreifen müssen. Um die beste Datenstruktur zu finden, braucht es viel Erfahrung und auch eine große Portion Glück.

4.3.3 Zufall und Wahrscheinlichkeit

Der Aufwand einer Operation ist häufig vom Zufall abhängig. Beispielsweise ist die Suche in einer verketteten Liste sehr effizient, wenn das gesuchte Element gleich am Anfang steht, aber ineffizient, wenn es erst am Ende zu finden ist. Das macht Aufwandsabschätzungen schwierig. Ein Hilfsmittel zur Untersuchung des zufallsbedingten Aufwands ist die Betrachtung einer großen Anzahl von Fällen. Statt einer Suche betrachten wir vielfach wiederholte Suchvorgänge mit unterschiedlichen Zahlen und Listen. Damit kann man die Durchschnittskosten eines Suchvorgangs berechnen.

Es macht einen Unterschied, welche Fälle wir betrachten. Beispielsweise verursacht die Suche im binären Baum im schlechtesten Fall linearen Aufwand, aber im Durchschnitt nur logarithmischen Aufwand. Für die Berechnung des Durchschnitts müssen wir Annahmen darüber treffen, welche

Zahlen in welcher Reihenfolge in die Bäume eingefügt wurden. Üblicherweise nehmen wir an, dass es sich um gleichverteilte Zufallszahlen handelt. Diese Annahme führt, wie man mit dem nötigen Hintergrundwissen über Wahrscheinlichkeitsrechnung zeigen kann, zu logarithmischem Aufwand, weil Bäume, die dabei entstehen, im Durchschnitt eine logarithmische Tiefe (Anzahl von Ebenen) haben. Allerdings sieht man in der Praxis deutlich häufiger, als die Theorie vermuten lässt, dass binäre Bäume entarten. Der Fehler liegt nicht an falschen Berechnungen, sondern fast immer daran, dass die implizit angenommene Gleichverteilung nicht gegeben ist. Die Zahlen, mit denen wir den Baum aufgebaut haben, sind die Ergebnisse von Berechnungen oder Messungen und stehen oft in Beziehungen zueinander. Das ist keine Gleichverteilung. Wenn wir Zahlen nur in aufsteigender Reihenfolge einfügen, beispielsweise die gemessenen Höhen eines aufsteigenden Ballons, bekommen wir garantiert einen zu einer Liste entarteten Baum mit dem schlechtest möglichen Aufwand bei allen Zugriffsoperationen. Ein Baum ist dafür einfach nicht geeignet. Wenn wir dagegen tatsächlich gleichmäßig verteilte Zahlen einfügen, ist ein Baum fast immer recht effizient. Bei der Auswahl einer Datenstruktur müssen wir also auch auf die Verteilung der benötigten Daten achten.

Binäre Bäume wurden entwickelt, um aus der zufälligen Verteilung von Daten Vorteile ziehen zu können. Viele Datenstrukturen und Algorithmen nutzen den Zufall aus. Eine *Hashtabelle* wie in Listing 4.18 ist ein gutes Beispiel dafür. Im Wesentlichen ist die Hashtabelle ein Array, in dem die Daten liegen. Der Index im Array, an dem ein bestimmtes Element zu finden ist, wird direkt aus dem Element errechnet. Diesen Index nennt man *Hashwert*. In Instanzen der Klasse `IntHashtable` verwenden wir als Hashwert den Absolutwert¹ des Elements modulo der Größe des Arrays. Dieser Wert liegt im Indexbereich des Arrays. Hashwerte können auf beliebige Weise berechnet werden, solange wiederholte Berechnungen für dasselbe Datenelement immer denselben Hashwert ergeben. Da der Indexbereich der Hashtabelle beschränkt ist, können auch die für unterschiedliche Datenelemente berechneten Hashwerte gleich sein. In diesem Fall *kollidieren* die Elemente miteinander. Es gibt mehrere Möglichkeiten, um mit Kollisionen umzugehen: Wir können z.B. linear die nächste freie Stelle in der Tabelle suchen oder einen weiteren Hashwert berechnen. In Listing 4.18 erlauben wir tatsächlich mehrere Elemente am selben Index

¹Der Absolutwert einer nicht-negativen Zahl ist die Zahl selbst, und der einer negativen Zahl ist die negierte Zahl (die durch die Negation positiv wird). Der Absolutwert ist nie negativ.

Listing 4.18: Hashtabelle ganzer Zahlen mit verketteten Listen

```

1 public class IntHashtable {           // Hashtabelle ganzer Zahlen
2     private IntListNode[] tab;        // IntListNode aus List. 4.4
3     public IntHashtable (int size) {   // Tabellengröße size > 0
4         tab = new IntListNode[size];
5     }
6     private int hashCode (int elem) { // berechne Hashwert (h)
7         return Math.abs(elem % tab.length); // 0 <= h < tab.length
8     }
9     public void add (int elem) {       // Füge elem ein
10        int hash = hashCode(elem);
11        tab[hash] = new IntListNode (elem, tab[hash]);
12    }
13    public boolean contains (int elem) { // Suche elem
14        int hash = hashCode(elem);
15        return tab[hash] != null && tab[hash].contains(elem);
16    }
17    public void remove (int elem) {    // Lösche elem
18        int hash = hashCode(elem);
19        if (tab[hash] != null) {
20            tab[hash] = tab[hash].remove(elem);
21        }
22    }
23 }

```

und speichern diese in einer verketteten Liste. Der Programmcode von `IntHashtable` entspricht dem von `IntList` in Listing 4.4, abgesehen davon, dass wir nicht nur einen Listenkopf, sondern ein Array von Listenköpfen haben. Der Hashwert bestimmt den zu verwendenden Listenkopf.

Die Qualität einer Hashtabelle wird hauptsächlich von der Qualität der berechneten Hashwerte bestimmt. Solange sich die Hashwerte gut auf den gesamten Indexbereich verteilen und die Tabelle groß genug ist, sind alle Zugriffsoperationen sehr effizient: Einfügen, Suchen und Löschen haben im besten Fall nur konstanten zeitlichen Aufwand. Wenn jedoch alle Hashwerte gleich sind, bestimmte Werte überwiegen oder die Tabelle zu klein wird, dann dominieren die Kollisionsbehandlungen bzw. die Listenoperationen, die Kosten für das Suchen und Löschen werden dadurch linear. In der Praxis sind gut dimensionierte Hashtabellen für viele Anwendungen tatsächlich sehr effizient. Allerdings führen falsch dimensionierte Hashtabellen entweder zu einem erheblichen Speicherverbrauch oder zu fast genau so langen Zugriffszeiten wie eine einfache verkettete Liste.

4.4 Teile und Herrsche

Eine in einigen Bereichen sehr erfolgreiche Lösungsstrategie nennt sich *Teile und Herrsche* (engl. *divide and conquer*). Dabei zerlegt man ein Problem in mehrere Teilprobleme, deren Lösung man anderen Programmteilen überlässt, und kümmert sich nur mehr um die korrekte Zusammenführung aller Teilergebnisse. Wir betrachten diese Strategie anhand einiger Beispiele im Zusammenhang mit Arrays ganzer Zahlen.

4.4.1 Das Prinzip

Um das Prinzip hinter Teile und Herrsche verstehen zu lernen, suchen wir nach einem effizienten Algorithmus zum Sortieren der Zahlen in einem Array. In Abschnitt 4.3.2 haben wir bereits das Sortierverfahren *Bubblesort* kennengelernt. Dieses Verfahren ist einfach zu verstehen und natürlich auch auf einem Array anwendbar. Aber es ist recht ineffizient, und wir suchen nach einem effizienteren Verfahren.

Es ist hilfreich, auf bewährte Techniken zurückzugreifen und diese an die aktuellen Gegebenheiten anzupassen. Betrachten wir Techniken, die in vielen Büros verwendet werden, um Akten sortiert in einem Aktenschrank abzulegen. Beispielsweise können wir für jeden Akt, sobald er verfügbar ist, sofort die richtige Stelle im Aktenschrank suchen und ihn dort verwahren. Wenn mehrere Akten gleichzeitig anfallen, ist diese Vorgehensweise ineffizient, weil man immer wieder von neuem nach der richtigen Stelle suchen und Aktenordner durchblättern muss. Wer das häufig macht, findet rasch eine effizientere Lösung: Man sammelt einen Stapel von Akten, die einzusortieren sind, und ordnet die Akten im Stapel irgendwie per Hand oder hält ihn von Anfang an sortiert. Bei überschaubarer Größe des Stapels ist das nicht schwer. Danach sortiert man die Akten in der Reihenfolge wie im Stapel in den Aktenschrank ein. Aufgrund der Sortierung braucht man dafür den Aktenschrank nur einmal von vorne nach hinten durchzugehen, da der nächste Akt nur an einer Stelle hinter dem vorangegangenen eingeordnet werden kann. Das vereinfacht die Suche erheblich.

Diese Vorgehensweise funktioniert gut, weil es einfach ist, zwei sortierte Aktenstapel zu einem zusammenzufügen: Man legt die beiden Stapel nebeneinander und nimmt jeweils einen Akt von einem der beiden Stapel, den in der Sortierung vorangehenden. Es reicht ein Vergleich pro Akt, der zeitliche Aufwand ist daher linear. Das lässt sich auch in einem Computerprogramm einfach realisieren, ist aber noch kein Sortierverfahren, da wir

Listing 4.19: Mergesort

```

1 // sortierte elems aufsteigend; Methode steht in beliebiger Klasse
2 public static void mergesort (int[] elems) {
3     if (elems.length > 1) { // zu sortieren?
4         int[] left = new int[elems.length / 2];
5         for (int i = 0; i < left.length; i++) { // 1. Teil
6             left[i] = elems[i]; // wird kopiert
7         }
8         mergesort(left); // und sortiert
9         int[] right = new int[elems.length - left.length];
10        for (int i = 0; i < right.length; i++) { // 2. Teil
11            right[i] = elems[left.length + i]; // wird kopiert
12        }
13        mergesort(right); // und sortiert
14        // ab hier werden die zwei sortierten Teile zusammengefügt
15        for (int i=0, l=0, r=0; i < elems.length; i++) {
16            if (r >= right.length // 2. Teil OK
17                || (l < left.length // 1. Teil offen
18                    && left[l] <= right[r])) {
19                elems[i] = left[l]; // vom 1. Teil
20                l++;
21            } else {
22                elems[i] = right[r]; // vom 2. Teil
23                r++;
24            }
25        }
26    }
27 }

```

davon ausgehen, dass die beiden Stapel, die wir zusammenfügen, schon sortiert sind. Die Strategie Teile und Herrsche bringt uns rasch zu einer vollständigen Lösung: Wir teilen die zu sortierende Datenmenge in zwei Teile und sortieren die beiden Teile, anschließend brauchen wir die beiden sortierten Teile nur mehr (auf gleiche Weise wie zwei Aktenstapel) zu einem Ganzen zusammenfügen. Zum Sortieren der Teile verwenden wir rekursive Aufrufe des Algorithmus. Listing 4.19 zeigt eine einfache Implementierung von *Mergesort*, dem hier beschriebenen Sortierverfahren. Die Rekursion bricht ab, wenn das zu sortierende Array nur ein Element hat, denn dann ist es ohne Zutun bereits sortiert.

Im Unterschied zu früheren Abschnitten dieses Kapitels sortieren wir hier nur einfache, als Argumente übergebene Arrays, keine Listen oder

Bäume, damit wir uns besser auf das Sortieren als die dahinterliegenden Datenstrukturen konzentrieren können. Natürlich kann man die Algorithmen problemlos so abändern, dass Sie auf Listen funktionieren. Auf den algorithmischen Aufwand hat diese Änderung keine nennenswerten Auswirkungen: Der zeitliche Aufwand $T(n)$ setzt sich zusammen aus dem linearen Aufwand $O(n)$ für das Erstellen der Kopien und das Zusammenfügen der sortierten Teile sowie dem Aufwand $2 \cdot T(\frac{n}{2})$ für das Sortieren der beiden Teile. Mit dem nötigen mathematischen Hintergrundwissen kann man daraus einen zeitlichen Aufwand von $T(n) = O(n \cdot \log(n))$ ableiten. Das ist sehr effizient. Noch dazu muss man bedenken, dass immer derselbe Aufwand entsteht, im schlechtesten Fall genauso wie im besten. Nicht ganz so gut schaut der Algorithmus hinsichtlich des Speicherbedarfs aus: Weil wir Teile des Arrays wiederholt kopieren, haben wir auch einen Speicherbedarf von $O(n \cdot \log(n))$. Durch einige Tricks und einen geschickteren Umgang mit Speicher könnten wir den Speicherbedarf auf $O(n)$ reduzieren. Darauf verzichten wir hier aber, da der entsprechende Algorithmus nicht mehr einfach zu verstehen ist.

Wie Mergesort zeigt, führt Teile und Herrsche auf natürliche Weise zu rekursiven Programmen, wenn zur Lösung der Teilprobleme wieder dieselbe Aufgabe auf einer Teilmenge der Daten zu lösen ist. Nicht jedes Teilproblem ist von derselben Art wie das Gesamtproblem, und daher führt nicht jede Anwendung der Strategie zur Rekursion. Umgekehrt beruht in gewisser Weise fast jede rekursive Methode auf Teile und Herrsche, da der rekursive Aufruf zur Lösung eines Teilproblems dient. Insofern haben wir schon viele Anwendungen dieser Strategie in früheren Beispielen gesehen.

Als Lösungsstrategie funktioniert Teile und Herrsche dann gut, wenn wir so wie bei der Suche nach einem Sortierverfahren vorgehen: Zuerst suchen wir einen vielversprechenden Lösungsansatz, der durchaus auch woanders als im engeren Bereich der Aufgabe angesiedelt sein kann. Vermutlich wird der Ansatz von einigen Voraussetzungen ausgehen, die nicht automatisch erfüllt sind. Wir suchen also nach Möglichkeiten, die Voraussetzungen zu erfüllen, wobei wir wieder Teile und Herrsche als Lösungsstrategie einsetzen können. Die Suche ist erfolgreich, sobald alle Voraussetzungen hinreichend gut erfüllt sind. Wenn wir für irgendeine Voraussetzung keine passende Lösung finden, dann müssen wir unser Glück mit einem anderen Ansatz versuchen. Teile und Herrsche ist keineswegs eine Technik, deren Anwendung immer zum Ziel führt. Vielmehr ist es eine Strategie, die uns bei der Suche nach einer kreativen Lösung unterstützt. Fehlende Kreativität lässt sich durch keine Strategie ersetzen.

4.4.2 Pragmatische Sichtweise

Obwohl Mergesort speziell hinsichtlich des Aufwands im schlechtesten Fall sehr gut abschneidet, verwendet man zum Sortieren heute viel häufiger *Quicksort*, siehe Listing 4.20. Quicksort ist in der Praxis meist (das heißt, im Durchschnitt) schneller als Mergesort. Gründe dafür sind

- recht hohe Kosten für das nötige Kopieren der Daten in Mergesort, die sich zwar in der theoretischen Berechnung des Aufwands nicht niederschlagen weil konstante Faktoren unberücksichtigt bleiben, aber bei Laufzeitmessungen sehr rasch bemerkbar machen,
- die Einfachheit wichtiger Teile von Quicksort, die für kleine konstante Faktoren und damit durchschnittlich für kurze Laufzeiten sorgen.

Es gibt zahlreiche Varianten von Quicksort mit verschiedensten Optimierungen, welche die Laufzeit in bestimmten Situationen weiter zu verbessern helfen. Listing 4.20 zeigt eine einfache Variante, in der das Wesentliche an diesem Sortierverfahren leicht zu erkennen ist. Quicksort bestimmt zuerst irgendein Element der Datenstruktur zum sogenannten *Pivot-Element* (in Deutsch: Angel- bzw. Drehpunkt). Dann wird die Datenstruktur in zwei Teile geteilt: Ein Teil enthält nur Elemente kleiner oder gleich dem Pivot-Element, der andere nur Elemente größer oder gleich dem Pivot-Element. Diese beiden Teile werden entsprechend Teile und Herrsche durch rekursive Aufrufe sortiert und durch Hintereinanderstellen wieder zu einer Einheit zusammengeführt. Das einfache Hintereinanderstellen reicht, da ja ein Teil nur alle kleineren und der andere nur alle größeren Elemente enthält. Im Unterschied zu Mergesort ist das Aufteilen der Elemente durch Berücksichtigung eines Pivot-Elements komplizierter, andererseits entfällt das aufwendige Zusammenfügen zweier sortierter Listen.

Die Effizienzvorteile von Quicksort im durchschnittlichen Fall ergeben sich vor allem aus der Einfachheit des Algorithmus. In der Variante in Listing 4.20 entfällt das Kopieren der Elemente und Hintereinanderstellen der beiden sortierten Teile, weil die zu sortierenden Teile des Arrays einfach nur über den linken und rechten Index (l und r) bestimmt werden. Man braucht den linken Teil nur zwischen den Indizes l und $i-1$ und den rechten zwischen i und r sortieren, und schon ist das Array zwischen l und r sortiert. Solche Verfahren, bei denen nicht kopiert zu werden braucht, nennt man *in-place* Sortierverfahren.

Listing 4.20: Quicksort

```

1 // sortiere elems aufsteigend; Methode steht in beliebiger Klasse
2 public static void quicksort (int[] elems) {
3     quicksort (elems, 0, elems.length - 1);
4 }
5 // sortiere elems aufsteigend von Index l bis Index r (inklusive)
6 private static void quicksort (int[] elems, int l, int r) {
7     int i = l;                // läuft von links nach rechts
8     int j = r;                // läuft von rechts nach links
9     int pivot = elems[(l+r)/2]; // trennt linken/rechten Teil
10    while (i <= j) {           // i und j aneinander vorbei?
11        while (elems[i] < pivot) { // alles kleiner pivot ...
12            i++;                // ... bleibt im linken Teil
13        }
14        while (pivot < elems[j]) { // alles größer pivot ...
15            j--;                // ... bleibt im rechten Teil
16        }
17        // hier gilt: elems[i] >= pivot >= elems[j]
18        if (i <= j) {           // nicht aneinander vorbei?
19            int h = elems[i];    // Tausch elems[i],elems[j]
20            elems[i] = elems[j]; // unverändert wenn i == j
21            elems[j] = h;
22            i++;                // und weiter zu ...
23            j--;                // ... nächsten Indizes
24        }
25    }
26    if (l < i-1) {              // links mehrere Elemente?
27        quicksort (elems, l, i-1); // sortiere linken Teil
28    }
29    if (i < r) {                // rechts mehrere Elemente?
30        quicksort (elems, i, r);   // sortiere rechten Teil
31    }
32 }

```

Die Zuordnung der Elemente zu den beiden Teilen ist auch nicht sehr aufwendig: Der Algorithmus läuft über die Indexvariablen i und j sowohl von links als auch von rechts solange in Richtung Mitte, bis sich i und j treffen. Dabei werden Elemente, die auf der linken Seite stehen, aber zur rechten gehören, mit solchen getauscht, die auf der rechten Seite stehen, aber zur linken gehören. Elemente mit einem Wert gleich dem Pivot-Element werden in den Tausch mit einbezogen. Dadurch verschiebt sich die Trennlinie zwischen dem linken und rechten Teil je nach Bedarf: Die Mitte ist dort, wo i und j sich treffen. Gelegentlich werden auch

gleiche Elemente getauscht, obwohl das gar nicht notwendig wäre. Zusätzliche Fallunterscheidungen zum Verhindern des unnötigen Tauschens würden die Laufzeit eher erhöhen als verringern.

Quicksort hat, wie Mergesort, durchschnittliche zeitliche Kosten von $O(n \cdot \log(n))$, aber im schlechtesten Fall von $O(n^2)$. Wie oben argumentiert, sind die üblicherweise gemessenen Laufzeiten für Quicksort jedoch deutlich kürzer als für Mergesort. Dieser scheinbare Widerspruch lässt sich dadurch klären, dass der schlimmste Fall bei Quicksort fast nie auftritt und Quicksort einen deutlich kleineren konstanten Faktor als Mergesort hat.

Ein kritischer Aspekt ist die Auswahl des Pivot-Elementes. Sein Wert entscheidet letztendlich, an welcher Stelle das Array in zwei Teile geteilt wird. Es ist anzustreben, dass beide Teile ungefähr gleich groß sind, denn dann bleiben die Kosten bei $O(n \cdot \log(n))$. Quadratischer Aufwand entsteht, wenn jede Aufteilung einen Teil mit nur einem Element und einen zweiten Teil mit dem ganzen Rest liefert. Bei Zufallszahlen ist das äußerst unwahrscheinlich. Prinzipiell kann man jedes Element im Array als Pivot-Element wählen. Wenn man jedoch beispielsweise immer das erste Element wählt, dann führt leider gerade ein bereits sortiertes Array zu quadratischem Aufwand: Das Pivot-Element ist das einzige Element des linken Teils. Deshalb wird das Pivot-Element in Listing 4.20 aus der Mitte genommen. Um sicher zu gehen, vergleicht man manchmal das erste Element, das letzte Element und ein Element aus der Mitte und wählt jenes mit dem mittleren Wert. Eine Garantie für ein gut gewähltes Pivot-Element gibt es nicht. Wenn man quadratischen Aufwand unbedingt ausschließen will, greift man beispielsweise auf Mergesort zurück.

4.4.3 Strukturelle Ähnlichkeiten

Hinter vielen bisher vorgestellten Verfahren steckt eine ähnliche Struktur. Beispielsweise haben wir in Abschnitt 4.3.2 einen binären Baum aufgebaut und dessen Elemente sortiert ausgegeben. Dafür war ein Aufwand von $O(n \cdot \log(n))$ im Durchschnitt und $O(n^2)$ im schlechtesten Fall nötig. Für das Sortieren eines Arrays mittels Quicksort haben wir denselben Aufwand. Dahinter steckt kein Zufall, sondern eine gemeinsame Struktur: In beiden Fällen haben wir etwas wiederholt in zwei Teile geteilt, einen linken und einen rechten Teilbaum bzw. einen linken und rechten Teil beim Sortieren. Die genaue Aufteilung wird in beiden Fällen vom Zufall gesteuert, und es ist möglich, dass die Strukturen entarten – ein Baum zu einer Liste, zwei gleichmäßig aufgeteilte Hälften des Arrays zum wiederholten

Listing 4.21: Anzahl der Versuche zum Erraten einer unbekannten Zahl

```
// Anzahl der Versuche, um Zufallszahl aus max Zahlen zu finden
public static int versuche (int max) {
    UnbekannteZahl zuErraten = new UnbekannteZahl(max);
    int i = 0; // untere Grenze
    int j = max - 1; // obere Grenze
    for (int anzahl = 0; ; anzahl++) { // zähle Versuche:
        int k = i + ((j - i) / 2); // probiere die Mitte
        if (zuErraten.kleiner(k)) {
            j = k - 1; // neue obere Grenze
        } else if (zuErraten.gleich(k)) {
            return anzahl;
        } else {
            i = k + 1; // neue untere Grenze
        }
    } // Schleife terminiert spätestens wenn i und j sich treffen
}
```

Abspalten nur eines Elements. Die Ähnlichkeit besteht trotz unterschiedlicher Ziele, ganz anderer Datenstrukturen und obwohl der wesentliche Teil bei Listen in den Datenstrukturen und beim Sortieren im Algorithmus enthalten ist. Auch wenn solche Ähnlichkeiten manchmal schwer zu beschreiben sind, kann man sie mit etwas Erfahrung rasch erkennen.

Die Kunst des Programmierens besteht häufig darin, Strukturen in einem Problem zu erkennen und bekannte Lösungsansätze für ähnlich strukturierte Probleme zu übernehmen. Mit der Zeit lernen wir viele Strukturen kennen, die wir in unterschiedlichster Form in unsere Programme einbauen. Viele Strukturen stammen aus unseren täglichen Erfahrungen, nicht nur im Zusammenhang mit dem Programmieren.

Nehmen wir als Beispiel die Methode `versuche` in Listing 4.21, die versucht, eine Zufallszahl möglichst rasch zu erraten, und die Anzahl der Versuche zurückgibt. Dazu setzen wir ein Suchverfahren ein, das wir vermutlich schon beim Ausprobieren von Zahlenraten in Abschnitt 1.1 intuitiv entwickelt haben: Wir betrachten den zur Verfügung stehenden Zahlenbereich, versuchen unser Glück mit einer Zahl aus der Mitte und grenzen, falls die Zahl nicht schon gefunden wurde, den zur Verfügung stehenden Zahlenbereich auf etwa die Hälfte ein. Nach wenigen Versuchen ist der Zahlenbereich so stark eingeschränkt, dass nur mehr eine Zahl in Frage kommt. Eine Zahl zwischen 0 und 99 haben wir so mit höchstens

Listing 4.22: Binäre Suche in einem sortierten Array

```
// suche x in elems und gib dessen Index zurück, sonst -1
// elems muss aufsteigend sortiert sein; Klasse beliebig
public static int index (int x, int[] elems) {
    int i = 0; // untere Grenze
    int j = elems.length - 1; // obere Grenze
    while (i <= j) {
        int k = i + ((j - i) / 2); // probiere die Mitte
        if (x < elems[k]) {
            j = k - 1; // links weitersuchen
        } else if (x == elems[k]) {
            return k;
        } else {
            i = k + 1; // rechts weitersuchen
        }
    }
    return -1; // x nicht gefunden
}
```

7 Versuchen erraten, durchschnittlich werden wir circa 5 Versuche brauchen. Generell brauchen wir für n mögliche Zahlen höchstens etwa $\text{ld}(n)$ Versuche. Wir nehmen den Logarithmus dualis, da wir bei jedem Versuch den Suchbereich auf ungefähr die Hälfte reduzieren. Der Aufwand von `versuche` entspricht daher auch $O(\log(n))$, da $\text{ld}(n) = \log(n)/\log(2)$ gilt und $\log(2)$ ein konstanter Faktor ist. Das ist gleichzeitig der durchschnittliche und der höchstmögliche Aufwand, da die meisten Zahlen erst im stark eingeschränkten Zahlenbereich gefunden werden. Meist findet man dieses Verfahren zum Erraten von Zahlen ganz natürlich auch ohne Hilfe und ohne Kenntnis von Logarithmen. Das ist ein Zeichen dafür, dass entsprechende Strukturen unserem Unterbewusstsein schon bekannt sind. Wir brauchen sie nur mehr abrufen. Leider lassen sich derart gute Algorithmen nicht immer so leicht finden, aber irgendein brauchbares Lösungsverfahren können wir für die meisten Probleme ganz intuitiv finden.

Wie Listing 4.22 zeigt, lässt sich der Algorithmus zum Erraten einer unbekannten Zahl zur *binären Suche* weiterentwickeln, einem effizienten Suchalgorithmus nach Zahlen in einem sortierten Array. Die Strukturen und damit der Aufwand sind fast identisch. Mit etwas Fantasie sind auch Ähnlichkeiten zwischen der binären Suche und der Suche in einem binären Baum zu erkennen. Auch im binären Baum beginnt die Suche in der Mitte

und setzt sich im linken oder rechten Teilbaum fort. Anders als ein binärer Baum kann ein sortiertes Array jedoch nicht zu einer Liste entartet sein.

4.5 Abstraktion und Generizität

Bisher haben wir nur Datenstrukturen über ganzen Zahlen betrachtet. Im Allgemeinen können dieselben Datenstrukturen Instanzen beinahe beliebiger Typen enthalten. Praktisch ergibt sich aber das Problem, dass wir in Java stets Typen der Datenelemente angeben müssen. Wenn wir statt einer Liste ganzer Zahlen eine Liste von Strings benötigen, müssen wir den gesamten Listen-Code wegen der geänderten Typen und wegen anderer Vergleichsoperationen neu schreiben. Eine zumindest teilweise Lösung dafür bietet *Generizität*: Dieser Sprachmechanismus erlaubt uns, Programmcode unabhängig von bestimmten Typen zu halten und ihn zu verallgemeinern. Zusammen mit verallgemeinertem Code wird es auch sinnvoll, Abstraktionen über Datenstrukturen einzuführen.

4.5.1 Generische Datenstrukturen

Listing 4.23 zeigt den Programmcode einer generischen Liste. Aus einer generischen Liste können Listen für ganz unterschiedliche Typen generiert werden – daher der Name. So ist `GenList<String>` eine Liste von Zeichenketten, `GenList<UnbekannteZahl>` eine Liste unbekannter Zufallszahlen und `GenList<Object>` eine Liste beliebiger Objekte. Der Code der generischen Liste unterscheidet sich nur in wenigen Punkten vom Code der ganzzahligen Liste in den Listings 4.4 und 4.6:

- Statt dem Elementtyp `int` wird durchwegs ein sogenannter *Typparameter* `A` verwendet. Der Typparameter ist selbst kein Typ, er steht aber für einen Typ bzw. wird später durch einen Typ ersetzt.
- Definitionen generischer Klassen deklarieren die in den Klassen vorkommenden Typparameter in spitzen Klammern (`<...>`). So enthält sowohl `class GenList<A>` als auch `class ListNode<A>` je eine Deklaration eines Typparameters `A`.
- Alle anderen Vorkommen von Namen generischer Klassen (aber nicht Konstruktoren) haben ebenfalls spitze Klammern, die Typen enthalten, welche die Typparameter in den Definitionen generischer

Listing 4.23: Generische verkettete Liste

```

1 public class GenList<A> {           // Liste von Instanzen von A
2     private ListNode<A> head = null; // Listenkopf
3     public void add (A elem) {       // füge elem am Anfang ein
4         head = new ListNode<A> (elem, head);
5     }
6     public boolean contains (A elem) { // elem in Liste enthalten?
7         return head != null && head.contains(elem);
8     }
9     public void remove (A elem) {
10        // Löschen des ersten Vorkommens von elem
11        // Liste unverändert wenn elem nicht vorkommt
12        if (head != null) {
13            head = head.remove(elem); // neuer Anfang nach Löschen
14        }
15    }
16 }
17 class ListNode<A> {                // Listenknoten
18     private A elem;                 // das eigentliche Listenelement
19     private ListNode<A> next;       // nächster Knoten = Listenrest
20     ListNode (A elem, ListNode<A> next) {
21         this.elem = elem;
22         this.next = next;
23     }
24     boolean contains (A e) {         // suche e im Rest der Liste
25         return e.equals(elem) || (next!=null && next.contains(e));
26     }
27     ListNode<A> remove (A e) {      // lösche e aus Restliste
28         if (e.equals(elem)) {       // ist this zu löschen?
29             return next;
30         } else if (next != null) {
31             next = next.remove(e);
32         }
33         return this;
34     }
35 }

```

Klassen ersetzen. Beispielsweise ist `GenList<String>` der Typ, der entsteht, wenn man in der Klasse `GenList<A>` jedes Vorkommen des Typparameters `A` durch `String` ersetzt. Eine Instanz von `GenList<String>` ist also eine Liste von Zeichenketten. Beim ersten Kontakt mit Generizität ist es verwirrend, dass innerhalb der Klassen `GenList<A>` und `ListNode<A>` der Typ `ListNode<A>` vorkommt. Auch dafür gilt, dass `ListNode<A>` für den Typ steht,

der entsteht, wenn man `A` durch den Typ ersetzt, für den `A` steht. Sobald `A` durch `String` ersetzt wird, steht auch `ListNode<A>` für `ListNode<String>`. Mit etwas Erfahrung ist die Verwendung von Typparametern statt Typen ganz selbstverständlich.

- Ein Vergleich beliebiger Objekte mittels `==` funktioniert zwar, liefert aber beispielsweise für Zeichenketten nicht das gewünschte Ergebnis. Deswegen wurde der Vergleich mittels `==` durch einen mittels `equals` ersetzt. Diese Methode ist programmierbar, sodass durch sie ganz unterschiedliche Arten von Vergleichen realisiert sein können. Das macht die generische Liste flexibel einsetzbar. Man kann jedoch nicht generell sagen, dass in generischen Klassen `equals` statt `==` zu verwenden ist. Der Unterschied ist semantischer Natur. Es kommt darauf an, welche Art von Vergleich benötigt wird.

Generizität ist ein sehr wertvolles Hilfsmittel vor allem bei der Erstellung von *Container-Klassen*, also von Klassen, deren Instanzen andere Objekte enthalten und den Zugriff darauf organisieren. Im Wesentlichen implementieren Container-Klassen Datenstrukturen. Man braucht nur eine generische Container-Klasse schreiben und bekommt dadurch Container-Klassen für alle möglichen Arten von Inhalten im Container.

Die Methode `equals` ist in `Object` als gleichbedeutend mit `==` vordefiniert, kann aber wie beispielsweise in `Student` in Listing 4.24 überschrieben werden. Zwei Instanzen von `Student` werden genau dann als gleich betrachtet, wenn die Matrikelnummern gleich sind. Damit liefert `contains` in einer Instanz von `GenList<Student>` immer `true`, wenn eine Instanz von `Student` mit der gesuchten Matrikelnummer vorhanden ist, egal ob die gefundene Instanz von `Student` identisch mit der gesuchten ist oder nicht. Auch `String` überschreibt `equals`, sodass alle Zeichen der Zeichenketten verglichen werden.

Als eine wenig attraktive Alternative zur Generizität könnten wir eine Container-Klasse schreiben, die beliebigen Inhalt haben kann, auch den, den wir tatsächlich haben wollen. Das wäre beispielsweise eine verkettete Liste, die alle Instanzen von `Object` enthalten kann. Allerdings ist es in dieser Alternative kaum möglich zu verhindern, dass auch andere Objekte in der Datenstruktur landen, als wir eigentlich haben wollen. Bevor Generizität mit Version 1.5 zu Java hinzugekommen ist, musste man diese Alternative verwenden. Inzwischen ist diese Technik jedoch überholt, auch wenn man gelegentlich noch immer auf alten Java-Code stößt, der auf die

Listing 4.24: Ausschnitt aus einer Klasse mit spezieller Vergleichsmethode

```
public class Student {
    private int mnr;           // Matrikelnummer
    ...                       // weitere Variablen und Konstruktoren
    public boolean equals (Object that) { // this gleich that?
        return this.getClass() == that.getClass()
            && mnr==((Student)that).mnr;
    }
    ...                       // weitere Methoden
}
```

se Art programmiert ist. Statt nicht-generischer Container-Klassen, die Instanzen von `Object` enthalten, sollte man heute nur mehr generische Container-Klassen schreiben. Falls man, was selten vorkommt, wirklich eine Datenstruktur braucht, die Beliebiges enthalten kann, geht das noch immer, indem man den Typparameter durch `Object` ersetzt. Beispielsweise kann eine Liste vom Typ `GenList<Object>` beliebige Objekte enthalten. Der Compiler garantiert jedoch, dass beispielsweise eine Instanz von `GenList<String>` nur Zeichenketten enthält.

Da Generizität erst nachträglich zu Java hinzugefügt wurde, ergeben sich wegen der notwendigen Kompatibilität zu älteren Versionen leider einige Nachteile, die in anderen Programmiersprachen nicht bestehen:

- Namen generischer Typen wie `GenList` haben auch ohne spitze Klammern eine semantische Bedeutung, die fast nie gebraucht wird und auf die wir deswegen nicht näher eingehen. Es gibt keine Fehlermeldung, wenn wir spitze Klammern vergessen. Im besten Fall bekommen wir eine unverständliche Compilermeldung (als *Note* bezeichnet). Deswegen müssen wir besonders auf spitze Klammern achten, um semantisch falsche Programme zu vermeiden.
- Typparameter dürfen nicht überall vorkommen, wo Typen notwendig sind. Beispielsweise können wir keine Instanz eines Typparameters erzeugen und kein Array anlegen, das Instanzen eines Typparameters enthält. Auch in Typabfragen mittels `instanceof` und in Typumwandlungen ist die Verwendung von Typparametern eingeschränkt.
- Typparameter können nur durch *Referenztypen* (das sind durch Klassen definierte Typen) ersetzt werden, nicht jedoch durch primitive Typen wie `int`, `double` oder `boolean`.

Der letzte Punkt wiegt anscheinend besonders schwer, weil dadurch die generische Liste doch kein Ersatz für die Liste ganzer Zahlen sein kann. Ganz so schlimm ist es glücklicherweise nicht. Um dieses Problem zu umgehen, gibt es in Java zu jedem primitiven Typ auch einen entsprechenden Referenztyp. Der Referenztyp zu `int` ist `Integer`, der zu `boolean` ist `Boolean` und so weiter. Jeder solche Referenztyp ist durch eine einfache Klasse definiert, die (ähnlich einer Schachtel oder *Box*) eine Instanz des entsprechenden primitiven Types und eine Reihe von Zugriffs- und Vergleichsoperationen darauf enthält. Beispielsweise vergleicht `equals` in diesen Klassen die Werte in der Box. Statt dem nicht erlaubten Typ `GenList<int>` können wir den Typ `GenList<Integer>` verwenden. Wenn beispielsweise `x` eine Instanz von `int` und `list` eine Instanz von `GenList<Integer>` ist, können wir `list.add(x)` aufrufen. Der Compiler wandelt `x` automatisch in eine Instanz von `Integer` um. Diese automatische Umwandlung nennt sich *Autoboxing*, und die automatische Umwandlung einer Instanz von `Integer` in eine von `int` *Autounboxing*. Java unterstützt Autoboxing und Autounboxing für alle primitiven Typen bzw. entsprechenden Referenztypen.

4.5.2 Gebundene Generizität

Einfache Generizität funktioniert wunderbar für einfache Datenstrukturen wie Listen, aber binäre Bäume oder Sortieralgorithmen können wir damit noch nicht implementieren. Dafür sind Größenvergleiche zwischen Elementen in der Datenstruktur nötig. Größenvergleiche sind aber nicht auf allen Objekten sinnvoll oder möglich.

Gebundene Generizität löst dieses Problem: Bei der Deklaration eines Typparameters gibt man einen Typ als *Schranke* an. Der durch die Schranke gebundene Typparameter kann nur durch einen Untertyp der Schranke ersetzt werden, und daher sind die in einer Instanz der Schranke zugreifbaren Methoden auch in einer Instanz des Typparameters zugreifbar.

Listing 4.25 veranschaulicht dieses Konzept an einer Erweiterung der Liste: Die „Werte“ aller in eine Liste eingefügten Elemente sollen aufsummiert werden. Für jeden Elementtyp kann der „Wert“ etwas anderes bedeuten. Wir legen nur fest, dass ein Aufruf der Methode `value` den Wert zurückgibt. Nun hat nicht jedes Objekt eine solche Methode. Daher legen wir durch eine Schranke auf dem Typparameter `A` fest, dass nur Untertypen des Interfaces `HasValue`, das die Methode deklariert,

Listing 4.25: Generische List mit Aufsummierung

```
public interface HasValue {           // als Schranke verwendeter Typ
    int value();                      // diese Methode wird benötigt
}

public class SumList<A extends HasValue> { // Schranke auf A
    private ListNode<A> head = null;
    private int sum = 0;
    public void add (A elem) {        // elem ist Instanz von HasValue
        head = new ListNode<A> (elem, head);
        sum += elem.value();          // daher elem.value() aufrufbar
    }
    ...
}

public class Student implements HasValue { // Untertyp d. Schranke
    private int numberOfCourses; // daher SumList<Student> erlaubt
    public int value() { // Implementierung der benötigten Methode
        return numberOfCourses;
    }
}
```

den Typparameter ersetzen dürfen. Im Beispiel ist `SumList<Student>` erlaubt, da `Student` das Interface und damit `value` implementiert. Dagegen ist `SumList<Object>` nicht erlaubt. Syntaktisch steht die Schranke in der Typparameterdeklaration nach dem Namen des Typparameters und `extends` in den spitzen Klammern. Hier steht immer `extends`, auch wenn die Schranke als Interface definiert ist. Innerhalb der generischen Klasse darf auf in der Schranke sichtbare Methoden und Variablen von Instanzen des Typparameters zugegriffen werden. Ein nicht gebundener Typparameter, also einer ohne Schranke, entspricht einem durch die Schranke `Object` gebundenen Typparameter.

Die generische Implementierung eines Baums kämpft mit einer weiteren Schwierigkeit: Ein Größenvergleich zwischen zwei Elementen ist nur sinnvoll, wenn beide Elemente einen auf gleiche Weise vergleichbaren Typ haben. Die Methode `equal` vergleicht `this` mit einer Instanz von `Object`, was nur möglich ist, weil ein Vergleich mit einer Instanz eines unbekannten anderen Typs immer `false` liefern kann. Dieser einfache Ausweg besteht bei Größenvergleichen nicht. Wir müssen sicherstellen, dass auch der formale Parameter der Vergleichsmethode den gewünschten Typ hat.

Listing 4.26: Generischer binärer Baum mit benötigten Klassen

```

public interface Comparable<T> { // Ergebnis<0: this < that
    int compareTo (T that);      //      =0: this.equals(that)
}                                //      >0: this > that
public class GenTree<A extends Comparable<A>> { // binärer Baum
    private TreeNode<A> root = null;
    public void add (A elem)      { /* füge elem ein */ ... }
    public boolean contains (A elem) { /* suche elem */ ... }
    public void remove (A elem)  { /* lösche elem */ ... }
}
class TreeNode<A extends Comparable<A>> { // Knoten im Baum
    private A elem;               // eigentliches Element
    private TreeNode<A> left = null; // linker Teilbaum
    private TreeNode<A> right = null; // rechter Teilbaum
    TreeNode (A e) { elem = e; }
    boolean contains (A e) {      // suche e im Baum
        int c = e.compareTo(elem); // Ergebnis des Vergleichs
        if (c < 0) {              // e in linkem Teilbaum?
            return left != null && left.contains(e);
        } else if (c == 0) {      // e gefunden
            return true;
        } else /* c > 0 */ {      // e in rechtem Teilbaum?
            return right != null && right.contains(e);
        }
    }
    void add (A e)                { /* füge e in Baum ein */ ... }
    TreeNode<A> remove (A e) { /* lösche e aus Baum */ ... }
}

public class Student implements Comparable<Student> {
    private int mnr = ...;
    public int compareTo (Student that) { // Vergleich über mnr
        return this.mnr - that.mnr;
    }
    public boolean equals (Object that) { // kompatibel: compareTo
        return this.getClass() == that.getClass()
            && mnr == ((Student)that).mnr;
    }
}

```

Rekursive gebundene Generizität bietet dafür eine Lösung, wie die in Listing 4.26 skizzierte generische Implementierung eines binären Baumes zeigt: Der Typparameter `A` ist durch `Comparable<A>` gebunden, der gerade deklarierte Typparameter kommt also in seiner eigenen Schranke vor

– eine Form von Rekursion. Der Typparameter `T` in `Comparable<T>` bestimmt den Typ des formalen Parameters von `compareTo`. Durch die Rekursion in der Schranke wird `T` durch denselben Typ ersetzt wie `A` in `GenTree<A>`, sodass `this` und `that` in `compareTo` denselben Typ haben müssen. Baumoperationen rufen diese Methode auf, wie beispielhaft für `contains` gezeigt. Die Klasse `Student` in Listing 4.26 implementiert `Comparable<Student>` und erfüllt damit die Bedingung, dass `A` Untertyp von `Comparable<A>` ist. Also ist `GenTree<Student>` erlaubt.

In der Praxis brauchen wir das Interface `Comparable<T>` nicht zu definieren, da es bereits genau so wie in Listing 4.26 vordefiniert ist. Glücklicherweise implementieren viele vordefinierte Klassen dieses Interface. Beispielsweise wird `Comparable<Integer>` von `Integer` implementiert und `Comparable<String>` von `String`. Wir können also problemlos Typen wie `GenTree<Integer>` und `GenTree<String>` verwenden.

4.5.3 Abstraktionen über Datenstrukturen

Generizität ermöglicht eine bestimmte Form der Abstraktion: Wir können Code, den wir nur einmal schreiben, für Datenstrukturen über unterschiedlichen Elementtypen einsetzen. Die Datenstrukturen selbst bleiben dabei stets gleich. Daneben hätten wir oft auch gerne eine andere Form der Abstraktion, bei der zwar die Elementtypen gleich bleiben, aber die Datenstrukturen variieren können. Dabei bekommen wir Zugriff auf eine Sammlung von Daten, müssen aber nicht zwischen Details von Listen, Bäumen, etc. unterscheiden. Letztere Form der Abstraktion lässt sich über Untertypbeziehungen realisieren.

Listing 4.27 zeigt ein Beispiel für die Verwendung von Untertypbeziehungen zwischen generischen Typen. Das Interface `Collection<E>` beschreibt Methoden, die man von vielen Arten von Datenstrukturen erwartet. So ist es möglich, dass zahlreiche Klassen wie `GenList<A>` und `GenTree<A>` dieses Interface implementieren. Untertypbeziehungen funktionieren auf generischen Typen genauso wie auf nichtgenerischen. Bei der Spezifikation der Obertypen werden die Typparameter des Untertyps mit denen der Obertypen in Beziehung gesetzt, sodass beispielsweise der Typparameter `A` von `GenList<A>` den Typparameter `E` von `Collection<E>` ersetzt. Typparameter der Untertypen können im Vergleich zu denen der Obertypen stärker eingeschränkt sein. So ist der Typparameter `A` von `GenTree<A>` durch die Schranke `Comparable<A>`

Listing 4.27: Datensammlung: Interface, Implementierungen, Verwendung

```

public interface Collection<E> {
    void add (E e);           // füge e zur Datenstruktur hinzu
    boolean contains (E e);   // ist e in Datenstruktur enthalten?
    void remove (E e);       // lösche ein e (falls vorhanden)
}

public class GenList<A> implements Collection<A> { ... }

public class GenTree<A extends Comparable<A>>
    implements Collection<A> { ... }

public class WordsInUse {           // Verwaltung bekannter Wörter
    private Collection<String> words; // Sammlung der Wörter
    public WordsInUse() {
        words = new GenTree<String>; // Datenstruktur nur hier
    }
    public boolean isNew (String word) { // word nicht vorhanden?
        if (words.contains(word)) {
            return false;
        } else {
            words.add(word);           // nachher sicher vorhanden
            return true;
        }
    }
}

```

gebunden, nicht jedoch `E` von `Collection<E>`. Allerdings werden durch die Einschränkung der Typparameter auch die Untertypbeziehungen eingeschränkt. Beispielsweise hat `Collection<String>` die Untertypen `GenList<String>` und auch `GenTree<String>` weil `String` ein Untertyp der Schranke ist. Aber `Collection<UnbekannteZahl>` hat nur den Untertyp `GenList<UnbekannteZahl>`. Aufgrund der Schranke ist `GenTree<UnbekannteZahl>` verboten. Dagegen dürfen einander entsprechende Typparameter in Obertypen nicht stärker eingeschränkt sein als in Untertypen. Wie Student in Listing 4.26 zeigt, können Obertypen Typparameter haben, obwohl es in Untertypen keine gibt. Auch Untertypen können Typparameter haben, die in Obertypen nicht vorkommen.

Listing 4.27 gibt ein Beispiel für die sinnvolle Verwendung abstrakter Datenstrukturen wie `Collection<E>`. Die Klasse `WordsInUse` verwaltet eine Menge von Wörtern. Jeder Aufruf von `isNew` fragt an, ob ein

bestimmtes Wort in der Menge enthalten ist. Wenn nicht, wird es eingefügt. Für `WordsInUse` spielt es (abgesehen von der Effizienz) keine Rolle, ob die Daten in einer verketteten Liste, einem binären Baum oder einer anderen Datenstruktur abgelegt werden. Nur zur Erzeugung der Datenstruktur muss deren Art bekannt sein. Wir sorgen durch Verwendung des Obertyps als Typ der Variablen `words` dafür, dass die Art der Datenstruktur außer an einer Stelle im Konstruktor unbekannt bleibt. Dadurch können wir die Art der Datenstruktur leicht durch eine Programmänderung an nur einer Stelle gegen eine andere austauschen. Das erhöht die Wartbarkeit. In großen Programmen ist diese Form der Abstraktion über Implementierungsdetails noch viel wichtiger als in diesem kleinen Beispiel.

Generizität und Untertypbeziehungen stellen unterschiedliche Abstraktionsformen zur Verfügung, die im Allgemeinen nicht gegeneinander austauschbar sind. Wie wir in Kapitel 3 gesehen haben, dienen Untertypbeziehungen dazu, Implementierungsdetails lokal gekapselt zu halten und durch Ersetzbarkeit den Austausch bzw. die Wiederverwendung von Programnteilen zu erleichtern. Wie obiges Beispiel zeigt, gilt das auch für Untertypbeziehungen auf generischen Typen. Generizität selbst unterstützt die Ersetzbarkeit nicht. Stattdessen erspart uns Generizität durch direkte Wiederverwendung von Programmcode Arbeit beim Schreiben von Container-Klassen. Rekursiv gebundene Generizität erlaubt uns, Typen von formalen Parametern so einzuschränken, dass sie gleich den Klassen sind, in denen die Methoden stehen. Das haben wir am Beispiel von `compareTo` in Listing 4.26 gesehen. Derartiges lassen Untertypbeziehungen wiederum nicht zu. In der praktischen Programmierung brauchen wir beide Formen der Abstraktion, Untertypbeziehungen fast überall, Generizität speziell im Zusammenhang mit Container-Klassen.

Im Zusammenhang mit Generizität unterstützt Java einige weitere Konzepte, um die Fähigkeit zur Abstraktion weiter auszubauen. So können einzelne Methoden, nicht nur ganze Interfaces oder Klassen, generisch sein. Beispielsweise definiert

Listing 4.28: Generische Identitätsfunktion

```

public static <A> A ident (A x) {
    return x;
}

```

eine generische Methode, die einfach nur den formalen Parameter zurückgibt. Sowohl der Typ des Parameters `x` als auch der Ergebnistyp ist durch

den Typparameter `A` festgelegt, der in den spitzen Klammern unmittelbar vor dem Ergebnistyp deklariert ist. Man kann die Methode mit einem Argument jeden beliebigen Referenztyps aufrufen und bekommt ein Ergebnis desselben Typs zurück. Da man den Umgang mit Algorithmen, Datenstrukturen und Lösungsstrategien auch ganz gut ohne generische Methoden verstehen kann, gehen wir hier nicht näher darauf ein.

Ein weiteres Konzept sind *Wildcard-Typen*, welche die Typen, die Typparameter ersetzen, ganz oder teilweise undefiniert lassen. Beispielsweise ist `Collection<?>` der Typ einer Datenstruktur mit unbekanntem Inhalt. Eine Instanz von `Collection<? extends UnbekannteZahl>` enthält Elemente eines beliebigen Untertyps von `UnbekannteZahl`, und eine Instanz von `Collection<? super UnbekannteZahl>` enthält Instanzen eines beliebigen Typs, der Obertyp von `UnbekannteZahl` ist. Sinnvolle Verwendungen solcher Wildcard-Typen sind kompliziert und zudem stark eingeschränkt. Deswegen gehen wir in diesem Skriptum nicht näher darauf ein.

4.5.4 Iteratoren

Häufig möchte man nacheinander alle Elemente eines Containers auslesen, ähnlich wie wir bereits in der Klasse `Zahlenraten` eine Zahl nach der anderen eingelesen haben. Das bewerkstelligen wir mit einem *Iterator*. Die Klasse `Print` in Listing 4.29 zeigt, wie Iteratoren verwendet werden. In diesem Beispiel werden alle Zeichenketten in einem Container `lines` nacheinander ausgelesen und ausgegeben. Dazu erzeugt der Ausdruck `lines.iterator()` einen neuen Iterator `i` über dem Container. Jeder Aufruf von `i.next()` liefert eine Zeichenkette aus dem Container zurück, und `i.hasNext()` überprüft, ob es weitere Zeichenketten gibt. Aufrufe von `i.next()` verändern den Zustand des Iterators, lassen aber den Container selbst unverändert.

Ein Container, auf dem ein Iterator erzeugt werden kann, heißt auch *Aggregat*. In Java sollte jedes Aggregat das Interface `Iterable<E>` implementieren, das nur eine Methode zum Erzeugen eines Iterators enthält. Auch `Collection<E>` erweitert `Iterable<E>` und unterstützt damit Iteratoren. Das Interface `Iterator` beschreibt die Methoden eines Iterators. Alle diese Schnittstellen sind ähnlich wie in Listing 4.29 (aber teilweise etwas umfangreicher) im Java-System vordefiniert.

Listing 4.29: Verwendung eines Iterators

```
public interface Iterator<E> { // Iterator über einem Aggregat
    A next();                // gib nächstes Element zurück
    boolean hasNext();        // gibt es weitere Elemente?
}

public interface Iterable<E> { // Aggregate impl. Iterable<E>
    Iterator<E> iterator();    // erzeuge Iterator auf Aggregat
}

// Instanzen von Collection<...> sind Aggregate:
public interface Collection<E> extends Iterable<E> { ... }

public class Print {          // Iterator-Verwendung (Beispiel)
    private Collection<String> lines;
    public Print (Collection<String> ls) {
        lines = ls;
    }
    public void print() {
        Iterator<String> i = lines.iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

Listing 4.30 zeigt wesentliche Teile der Implementierung eines Iterators auf einer verketteten Liste. Es ist wichtig, dass der Iterator selbst weiß, an welcher Stelle in der Liste er sich gerade befindet. Dazu dient die Variable `node` in `ListIter<A>`. Bei jeder Ausführung von `next` wird der Inhalt der Variablen um einen Listenknoten weiter nach hinten verschoben. Eine kleine zusätzliche Erschwernis entsteht in der Implementierung in Listing 4.30 dadurch, dass Instanzen von `ListIter<A>` nicht direkt auf solche von `ListNode<A>` zugreifen dürfen. Dieses Problem ist durch sogenannte *getter-Methoden* gelöst, die Lesezugriffe auf Variablen ermöglichen. Üblicherweise wollen wir getter-Methoden so gut es geht vermeiden, aber sie sind dem direkten Zugriff auf Variablen eines anderen Objekts, das zu einer anderen Klasse gehört, meist dennoch vorzuziehen.²

²Java unterstützt sogenannte *innere Klassen*, also Klassen, die zu bestimmten Objekten gehören. Innere Klassen ermöglichen den unbeschränkten Zugriff auf diese Objekte, auch

Listing 4.30: Implementierung eines Iterators über einer verketteten Liste

```

public class GenList<A> implements Collection<A> {
    private ListNode<A> head = null;
    public Iterator<A> iterator() { // neuer Iterator auf Liste
        return new ListIter<A>(head);
    }
    ...
}
class ListIter<A> implements Iterator<A> { // Listeniterator
    private ListNode<A> node; // eigenes node für jede Instanz
    ListIter (ListNode<A> head) { // initial. mit Listenanfang
        node = head;
    }
    public A next() { // gib nächstes Element in der
        if (node != null) { // Reihenfolge der Liste zurück
            ListNode<A> result = node;
            node = node.getNext();
            return result.getElem();
        }
        return null;
    }
    public boolean hasNext() { // weitere Listenelemente?
        return node != null;
    }
}
class ListNode<A> { // Erweiterung für Iterator nötig
    private A elem;
    private ListNode<A> next;
    ListNode (A elem, ListNode<A> next) {
        this.elem = elem;
        this.next = next;
    }
    A getElem() { return elem; } // getter für elem
    ListNode<A> getNext() { return next; } // getter für next
    ...
}

```

Es ist möglich, mehrere Iteratoren gleichzeitig auf demselben Container zu verwenden. Jeder der Iteratoren hat seine eigene Variable `node` und

auf private Teile. Wenn man Iteratoren als innere Klassen der Aggregate implementiert, gibt es keine Probleme mit der Zugreifbarkeit, und man kann sich die getter-Methoden ersparen. Wir haben hier auf die Verwendung innerer Klassen verzichtet, um uns auf das Wesentliche zu konzentrieren und die Lösung nicht mit weiteren, bisher nicht betrachteten Sprachkonzepten zu überfrachten.

Listing 4.31: Generischer Stack als verkettete Liste (ListNode aus Listing 4.30)

```

1 public class GenStack<A> { // Stack, als Liste implementiert
2     private ListNode<A> top = null; // oberstes Stackelement
3     public void push (A elem) { // gib elem auf Stack
4         top = new ListNode<A>(elem, top);
5     }
6     public A pop() { // nimm oberstes Element von Stack
7         if (top != null) { // falls vorhanden; sonst null
8             A result = top.getElem();
9             top = top.getNext();
10            return result;
11        }
12        return null;
13    }
14    public boolean isEmpty() { // ist der Stack leer?
15        return top == null;
16    }
17 }

```

kann somit unbeeinflusst von anderen Iteratoren die Liste durchwandern. Das ist ein wesentlicher Unterschied zu Methoden in einem Container, über die man direkt auf Elemente zugreift: Zugriffsoperationen ähnlich dem `next`, aber direkt im Container und ohne Iterator, verändern den Container; mehrere gleichzeitige Durchläufe würden sich gegenseitig beeinflussen. Als Beispiel dafür dient die Stackimplementierung in Listing 4.31: Eine Ausführung von `pop` nimmt zwar ein Element vom Stack, sehr ähnlich wie das eine Ausführung von `next` in einem Iterator macht. Allerdings wird das Element durch `pop` tatsächlich aus dem Container entfernt, während der Container selbst durch einen Iterator nicht verändert wird. Außerdem beeinflusst das Einfügen oder Löschen eines Listenelements den Iterator nicht bzw. beim Löschen nur auf die erwartete Weise. Der Iterator ist *robust* gegenüber diesen Operationen. Ein Aufruf von `push` im Stack wirkt sich dagegen sofort auf den nächsten Aufruf von `pop` aus.

Listing 4.32 zeigt die Implementierung eines Iterators auf einem binären Baum. Dieser Iterator gibt die Elemente in sortierter Reihenfolge zurück. Die Implementierung eines Baumiterators ist wesentlich aufwendiger als die eines Listeniterators. Der Grund dafür liegt darin, dass die Methode `next` den Baum auf ähnliche Weise durchwandern muss wie die Methode `print` in Listing 4.17 für die sortierte Ausgabe der Elemente, dabei aber

Listing 4.32: Implementierung eines sortierten Iterators über binärem Baum

```

public class GenTree<A extends Comparable<A>>
    implements Collection<A> {
    private TreeNode<A> root = null;
    public Iterator<A> iterator() { // neuer Iterator, iteriert
        return new TreeIter<A>(root); // in sortierter Reihenfolge
    }
    ...
}

class TreeIter<A extends Comparable<A>> implements Iterator<A> {
    private GenStack<TreeNode<A>> stack
        = new GenStack<TreeNode<A>>();
    TreeIter (TreeNode<A> n) { // stack enthält Pfad von Wurzel
        while (n != null) { // bis zu ganz linkem Blatt (top
            stack.push(n); // Knoten mit kleinstem Element)
            n = n.getLeft();
        }
    }
    public A next() { // gib nächstes Element zurück
        TreeNode<A> result = stack.pop(n); // top: nächster Knoten
        if (result != null) { // noch nicht am Ende:
            TreeNode<A> n = result.getRight(); // rechts weiter ->
            while (n != null) { // top = kleinstes
                stack.push(n); // Element rechts
                n = n.getLeft(); // von result
            }
            return result.getElem();
        }
        return null;
    }
    public boolean hasNext() { // weitere Elemente?
        return !stack.isEmpty();
    }
}

class TreeNode<A extends Comparable<A>> { // Erweiterung nötig
    private A elem;
    private ListNode<A> left, right;
    A getElem() { return elem; } // getter für elem
    ListNode<A> getLeft() { return left; } // getter für left
    ListNode<A> getRight() { return right; } // getter für right
    ...
}

```

keine Rekursion verwenden kann. Rekursion ist nicht möglich, weil das Durchwandern nach jedem gefundenen Element abgebrochen werden muss

und erst wieder beim nächsten Aufruf von `next` fortgesetzt werden kann. Die aktuelle Position im Baum lässt sich, anders als bei der Liste, nicht in nur einer einfachen Variablen festhalten. Programmiersprachen verwenden einen für den Programmierer nicht sichtbaren Stack, um darin die Variablen geschachtelter Methodenaufrufe zu speichern, auch die von rekursiven Aufrufen. Daher ist ein Stack oft eine hilfreiche Datenstruktur, um eine rekursive Methode in eine nicht-rekursive umzuwandeln. Wir nehmen den in Listing 4.31 implementierten Stack zu Hilfe. Auf dem Stack legen wir im Wesentlichen noch nicht besuchte Knoten des Baums ab, die den rekursiven Aufrufen von `left.print()` in Listing 4.17 entsprechen. Für Aufrufe von `right.print()` brauchen wir keinen Stack, da sich dafür eine Situation sehr ähnlich der mit der Liste ergibt. Im Konstruktor legen wir den ganzen *Pfad* (also alle beim Durchwandern besuchten Knoten) von der Baumwurzel bis zum am weitesten links stehenden Blattknoten auf den Stack. Das ist der Knoten mit dem kleinsten Element. Jeder Aufruf von `next` gibt das oberste Element am Stack zurück und legt danach den Pfad bis zum am weitesten links stehenden Knoten im rechten Teilbaum auf den Stack. Das ist das nächste Element. Falls es keinen rechten Teilbaum gibt, ist das nächste Element der Vorgänger im Baum, der als nächstes ganz oben am Stack liegt.

Es gibt viele Möglichkeiten, einen Baum zu durchwandern. Statt einer sortierten Ausgabe könnten wir beispielsweise immer zuerst die Wurzel ausgeben, dann den linken und schließlich den rechten Teilbaum. Wir könnten auch zuerst die beiden Teilbäume und erst dann die Wurzel ausgeben, oder zuerst die Wurzel, dann die Elemente eine Ebene unterhalb der Wurzel, dann jene zwei Ebenen darunter und so weiter. Egal welche Reihenfolge wir wählen, mit nur einer Variablen zur Speicherung des aktuellen Knotens kommen wir niemals aus. Wir brauchen entweder einen Stack, oder wir speichern in jedem Baumknoten auch den Vorgängerknoten, oder wir durchwandern den Baum mehrfach auf der Suche nach Knoten.

Iteratoren müssen keine Reihenfolge vorschreiben. Meist wird eine bestimmte Reihenfolge eingehalten, um den Iterator vielfältiger einsetzen zu können. Manchmal haben wir mehrere Iterator-Klassen mit unterschiedlichen Reihenfolgen auf derselben Container-Klasse implementiert, aber nur Instanzen von einer davon können über die in `Iterable<E>` spezifizierten Methode `iterator` erzeugt werden.

Iteratoren sind als Abstraktionsmittel sehr wertvoll. Man kann alle Elemente eines Containers besuchen, ohne die Art des Containers kennen zu müssen. Weil Iteratoren häufig eingesetzt werden, gibt es in Java eine spe-

zielle Syntax für `for`-Schleifen über Containern ähnlich den `for`-Schleifen über Arrays: In „`for (T v: c) ...`“ läuft die Schleifenvariable `v` vom Typ `T` über alle Elemente des Containers `c`, wobei `c` eine Instanz von `Iterable<T>` sein muss. Die Elemente werden über den Iterator ausgelesen und in der vom Iterator vorgegebenen Reihenfolge abgearbeitet. Die Methode `print` in Listing 4.29 können wir kürzer so schreiben:

Listing 4.33: Verwendung einer `for`-Schleife über einem Container

```
public void print() {
    for (String s: lines) {
        System.out.println(s);
    }
}
```

Diese Definition hat genau dieselbe Semantik wie jene in Listing 4.29.

4.6 Typische Lösungsstrategien

Einzelne Datenstrukturen und Algorithmen bilden den Kern vieler Programmieraufgaben. Häufig ist jedoch gar nicht deren Komplexität das Hauptproblem beim Lösen praktischer Aufgaben, sondern schlicht und einfach der Umfang der Aufgaben. Es ist so viel zu tun, dass man gar nicht weiß, wo man anfangen soll. Alle Details sind unmöglich auf einmal zu überblicken. Gerade in solchen Situationen bieten sich die hier diskutierten Lösungsstrategien an. Häufig stellen sich Aufgaben als schwieriger und umfangreicher heraus, als man Anfangs glaubt. Daher sind diese Lösungsstrategien für fast jede Programmieraufgabe sinnvoll.

4.6.1 Vorgefertigte Teile

Ein bei der Programmkonstruktion häufig gehörtes Sprichwort lautet: „*Man muss nicht immer wieder das Rad neu erfinden.*“ Für unzählige Aufgaben bzw. Teilaufgaben existieren bereits vollständige, gut durchdachte, effiziente und vielfach bewehrte Lösungen. Diese braucht man nur zu verwenden, statt immer wieder neuen Programmcode zur Lösung fast gleicher Aufgaben zu schreiben. So einfach und überzeugend die Aufforderung zur Verwendung fertigen Codes klingt, so schwierig ist dessen Befolgung in der Praxis. Folgende Gründe spielen dabei eine Rolle:

Unkenntnis: Einer der häufigsten Gründe dafür, dass man fertige Programmteile nicht verwendet, ist schlicht und einfach das fehlende

Wissen über deren Existenz. Alleine die Standardbibliotheken üblicher Java-Distributionen umfassen schon mehrere Tausend Klassen. Kaum jemand hat einen vollständigen Überblick darüber. Glücklicherweise reicht schon die Kenntnis eines kleinen Teils der Klassen aus, um für viele praktisch auftretende Teilaufgaben angepasste Lösungen zu finden.

Andererseits reicht es nicht, wenn man nur von der Existenz fertiger Lösungen weiß. Man muss auch lernen, die fertigen Lösungen richtig einzusetzen. Häufig ist das Erlernen des richtigen Einsatzes scheinbar aufwendiger, als selbst rasch eine neue Lösung zu entwickeln. Allerdings ist diese rasch entwickelte Lösung meist wenig durchdacht und daher fehleranfällig, was im Nachhinein zu einem viel höheren Aufwand führen kann als angenommen. Wenn man eine hochwertige und inhaltlich gut passende Lösung für eine Teilaufgabe gefunden hat, sollte man diese trotz hohem Einlernaufwand einsetzen.

Unterschiedliche Modelle: Hinter jedem Lösungsansatz steckt ein bestimmtes gedankliches Modell. Nur selten passen die Modelle hinter einer fertigen Lösung und hinter dem restlichen Programm zufällig gut zusammen. Damit die Modelle übereinstimmen, müssen wir das Programm schon im Hinblick auf die Verwendung bestimmter Teile entwickeln. Mittlerweile sind zumindest die Standardbibliotheken so weit gereift, dass die Modelle hinter häufig gemeinsam verwendeten fertigen Klassen gut zusammenpassen. Das ist keineswegs selbstverständlich. Wenn das Modell hinter einer fertigen Lösung einer Teilaufgabe ganz und gar nicht mit unserem Programm zusammenpasst, ist es möglicherweise besser, auf die fertige Lösung zu verzichten, als das ganze Programm an die fertige Lösung anzupassen. Zusätzlicher Code für Anpassungen kann umfangreich und fehleranfällig sein.

Mangelndes Vertrauen: Wer einen fertigen Programmteil einsetzt, muss darauf vertrauen, dass dieser Teil auf Dauer die Erwartungen erfüllt. Dazu gehört, dass der Programmteil frei von schädlichem oder sogar gefährlichem Code ist und Fehler, die im Laufe der Zeit auftauchen, rasch beseitigt werden. Man muss auf den Hersteller des verwendeten Programmteils vertrauen. Mangelndes Vertrauen ist eine von mehreren Ursache für das sogenannte *Not-Invented-Here-Syndrom*, das unter anderem bewirkt, dass in anderen Gruppen oder Unternehmen entwickelte Programmteile deutlich seltener eingebunden werden als

in der eigenen Gruppe oder dem eigenen Unternehmen entwickelte. Vertrauen kann man durch keine Technik erzwingen.

Ich kann es besser: Gelegentlich verzichtet man auf die Verwendung fertiger Teile, weil man glaubt, selbst eine bessere oder zumindest besser an die Aufgabe angepasste Lösung finden zu können. Tatsächlich können eigene Lösungen kleiner und einfacher sein, was durchaus vorteilhaft ist. Allerdings liegt die Ursache dafür oft darin, dass in der eigenen Lösung auf bestimmte Sonderfälle, die auftreten könnten, vergessen wurde und in Zukunft nötige Erweiterungen noch fehlen. Fertige Programmteile sind meist (aber leider nicht immer) sehr gut durchdacht, kümmern sich um alle Sonderfälle und haben schon für die am häufigsten nötigen künftigen Erweiterungen vorgesorgt. Das liegt daran, dass diese Teile schon vielfach eingesetzt wurden und daraus gewonnene Erfahrungen in die Weiterentwicklung eingeflossen sind. Dieser Entwicklungsvorsprung ist mit einer eigenen Lösung trotz besserer Anpassung an die Aufgabe nicht aufholbar.

Gerade für die wichtigsten Datenstrukturen bieten Standardbibliotheken (nicht nur für Java) umfangreiche fertige Lösungen. Arrays sind ohnehin in Java integriert. Als Ausgangspunkt für die Suche nach passenden Datenstrukturen in Java kann `Collection<E>` oder (noch umfangreicher) `Iterable<E>` dienen. Die wichtigsten Container-Klassen implementieren diese Interfaces. Dazu zählen beispielsweise `LinkedList<E>` (umfangreiche Implementierung einer verketteten Liste), `HashSet<E>` (eine Hashtabelle), `Stack<E>` und `TreeSet<E>` (eine effiziente Implementierung eines binären Baums, der nicht entartet sein kann). Daneben enthalten die Klassen `Collections` und `Arrays` eine Reihe von statischen Methoden, die zusammen mit Containern und Arrays verwendbar sind, beispielsweise zum Sortieren. Eigentlich gibt es keinen Grund, die Datenstrukturen, die wir in diesem Kapitel kennengelernt haben, selbst auszuprogrammieren – abgesehen vom didaktischen Gesichtspunkt, weil wir beim Programmieren auch über interne Details von Datenstrukturen Bescheid wissen müssen.

Im Vergleich zu den in Beispielen verwendeten Container-Klassen sind die von Java bereitgestellten Container-Klassen zwar etwas umfangreicher, aber sehr klar strukturiert. Das liegt daran, dass jahrelange Erfahrungen im Umgang mit ihnen in die Entwicklung eingeflossen sind. Dagegen sind die ursprünglich in den ersten Java-Versionen eingesetzten Container-Klassen nicht Teil der `Collection<E>` Typhierarchie und werden heu-

te kaum noch verwendet. Heute werden keine ausgefallenen Operationen mehr unterstützt, und die verfügbaren Operationen sind sehr einheitlich gestaltet. Diese Eigenschaften sind wichtig, damit man sich rasch in der Vielfalt der verfügbaren Datenstrukturen zurechtfinden kann. Gerade die einheitliche Gestaltung lässt ein ganz bestimmtes Modell entstehen. Es orientiert sich an den verfügbaren Operationen. Beim Programmieren in Java sollte man sich an dieses vorgegebene Modell halten, da es sonst recht schwierig wird, fertige Lösungen zu verwenden. Man braucht dazu eine gewisse Erfahrung im Umgang mit Standardbibliotheken.

Zusammenfassend kann man sagen, dass die Verwendung vorgefertigter Teile einerseits wichtig ist, andererseits aber ohne Erfahrung damit kaum erfolgreich sein wird. Einerseits kann man sich sehr viel Arbeit ersparen und zugleich die Zuverlässigkeit erhöhen. Andererseits ist die Verwendung fertiger Lösungen zu Beginn mit zusätzlicher Arbeit verbunden. Man sollte nicht beliebige fertige Teile einbinden, sondern nur solche, die sich bewehrt haben und zu denen man Vertrauen hat. Unter diesen Voraussetzungen zahlt es sich aus, Programme so zu strukturieren, dass sie mit Modellen hinter fertigen Lösungen zusammenpassen.

4.6.2 Top Down versus Bottom Up

Die *Top Down Strategie* versucht, Systeme anfangs auf einer abstrakten Ebene zu strukturieren und schrittweise mit Details anzureichern, bis alle Teile implementiert sind. Zuerst werden die wesentlichen Vorgänge identifiziert und diese dann getrennt voneinander implementiert.

Ein Beispiel soll diese Vorgehensweise demonstrieren: Wir entwickeln ein Programm, das Zahlen einliest und diese in sortierter Reihenfolge ausgibt. Einen Teil der Aufgabe haben wir bereits in Abschnitt 4.3.2 betrachtet. Nun wollen wir top down vorgehen: Aus der Aufgabe ergibt sich ganz natürlich eine Gliederung in drei Vorgänge: Einlesen der Zahlen, Sortieren und Ausgeben. Die direkte Umsetzung könnte so aussehen:

Listing 4.34: Top Down Strategie: Einlesen und Sortiertes Ausgeben

```
public static void main (String[] args) {
    List<Integer> nums = readNums();
    Collections.sort(nums);
    print(nums);
}
```


Als Container haben wir eine Instanz von `List<Integer>` gewählt, da die Klasse `Collections` dafür eine statische Methode zum Sortieren enthält. Um das Sortieren brauchen wir uns daher nicht weiter kümmern. Das Einlesen können wir untergliedern in das Erzeugen eines Scanners, das Erzeugen der Datenstruktur und eine Schleife zum Einlesen der Zahlen:

Listing 4.35: Top Down Strategie: Einlesen von Zahlen

```
private static List<Integer> readNums () {
    Scanner sc = new Scanner(System.in);
    List<Integer> nums = new LinkedList<Integer>();
    while (sc.hasNextInt())
        nums.add(sc.nextInt());
    return nums;
}
```

In einem weiteren Schritt sollten wir diese Methode durch Ausgabe von Eingabeaufforderungen und Fehlermeldungen bei falschen Eingaben benutzerfreundlicher gestalten. Darauf verzichten wir hier, um nicht vom Wesentlichen hinter der Top Down Strategie abzulenken. Schließlich brauchen wir auch noch eine Methode zum Ausgeben:

Listing 4.36: Top Down Strategie: Ausgeben

```
private static void print (List<Integer> nums) {
    for (int i: nums)
        System.out.println(i);
}
```

Wie wir sehen, führt die Top Down Strategie eher zu traditionellen imperativen statt zu objektorientierten Programmen; alle Methoden sind statisch, und wir können sie in eine beliebige Klasse geben. Eine Untergliederung der Aufgabe führt zur nächsten, und nach kurzer Zeit ist die Aufgabe gelöst. Allerdings ist die dabei entstehende Programmstruktur meist nicht optimal: Durch die frühe Unterteilung bleiben Querverbindungen zwischen den Teilen unerkannt. Oft sind fertige Lösungsteile nicht einsetzbar, weil sie nicht mit der Gliederung übereinstimmen.

Die *Bottom Up Strategie* geht genau umgekehrt vor. Man sucht zuerst nach einzelnen bereits gelösten Teilen, die wahrscheinlich gebraucht werden. Dann versucht man, daraus größere Teile zu formen, bis die eigentliche Aufgabe gelöst ist. Es ist einfach zu erkennen, dass eine sortierte Datenstruktur die Lösung der Beispielaufgabe vereinfacht. Wir brauchen einen Container, der das Einlesen und Ausgeben unterstützt – siehe Listing 4.37.

Listing 4.37: Bottom Up Strategie: Wichtige Teile zuerst

```
import java.util.Scanner;
import java.io.PrintStream;
public class SortedNums {
    private GenTree<Integer> nums = new GenTree<Integer>();
    public void readFrom (Scanner in) {
        while (in.hasNextInt())
            nums.add(in.nextInt());
    }
    public void printTo (PrintStream out) {
        for (int i: nums)
            out.println(i);
    }
}
```

Listing 4.38: Bottom Up Strategie: main kommt am Ende

```
public static void main (String[] args) {
    SortedNums nums = new SortedNums();
    nums.readFrom(new Scanner(System.in));
    nums.printTo(System.out);
}
```

Damit sind wir schon fast fertig. Es fehlt nur noch die Methode `main` wie in Listing 4.38.

Durch geschickte Auswahl der Teile, die bottom up erstellt werden, können bereits vorhandene Programmteile gut genutzt werden. Es ist leichter, vorhandene Beziehungen zwischen einzelnen Teilen zu erkennen und in schön strukturierte Klassen zu integrieren. Allerdings kann es bei größeren Aufgaben leicht passieren, dass wir uns verlaufen. Dann schreiben wir Klassen, die am Ende gar nicht gebraucht werden, weil sie die Lösung der Gesamtaufgabe nicht vereinfachen. Mit der Top Down Strategie passiert das nicht.

Häufig ist eine Kombination aus Top Down und Bottom Up Strategie sinnvoll. Klassen zur Lösung von Teilaufgaben, die sicher gebraucht werden, können wir schon einmal bottom up erstellen, noch bevor die Gliederung des Gesamtsystems bekannt ist. Die grobe Gliederung des Systems nehmen wir jedoch top down vor, damit wir uns nicht so leicht verlaufen.

Bei der Gliederung lassen wir uns bis zu einem gewissen Grad von den bereits bottom up erstellten Teilen leiten. Abstrakte Gliederungen entstehen eher top down, konkrete Realisierungen bottom up, und die beiden Strategien treffen sich irgendwo in der Mitte des Abstraktionsgrades.

4.6.3 Schrittweise Verfeinerung

Strategien wie Top Down und Bottom Up sind gut geeignet, um klar umrissene eher kleine Programmieraufgaben zu lösen. In der Praxis haben wir es häufig mit riesengroßen Aufgaben zu tun, die unmöglich in ihrer Gesamtheit durchschaubar sind. Prinzipiell könnten wir zwar trotzdem top down vorgehen, um Aufgaben in kleinere Teile zu zerlegen, aber das hat sich nicht bewährt. Wir bekommen viel zu spät Rückmeldungen darüber, ob ein gewählter Lösungsansatz funktioniert. Das heißt, wir müssen ein riesiges Programm entwerfen und implementieren, bevor wir es testen können. Tests werden ziemlich sicher schwerwiegende Mängel aufzeigen. Deren Ursachen sind alleine schon aufgrund der Größe des Programms nur schwer zu finden und noch schwerer zu beseitigen. Manchmal stellt sich erst beim Testen heraus, dass der generelle Lösungsansatz falsch ist und umfangreiche Teile des Systems neu entwickelt werden müssen. So manches Softwareprojekt ist deswegen schon gescheitert.

Die Strategie der *schrittweisen Verfeinerung* verspricht, dieses Problem in den Griff zu bekommen. Dabei lösen wir zu Beginn nur einen kleinen, überschaubaren Teil der Programmieraufgabe. Für diesen Teil durchlaufen wir alle Schritte der Softwareentwicklung von der Analyse über den Entwurf und die Implementierung bis zum Verifizieren, Testen und Validieren. Bei geeigneter Wahl des Programmteils sind Top Down und Bottom Up Strategien oft gut anwendbar. Vor allem durch die letzten Schritte bekommen wir gute Rückmeldungen über die bisherige Qualität des Programms. Solche Informationen nutzen wir bei der Weiterentwicklung. Wir erweitern unsere bisherige Lösung entsprechend der Aufgabe sowie den Rückmeldungen und führen für jede Erweiterung alle Softwareentwicklungsschritte durch. In den meisten Fällen wird sich unsere Vorstellung darüber, wie die Aufgabenstellung zu interpretieren ist, im Laufe der Zeit und mit den gewonnenen Erfahrungen ständig verschieben. Auch die Aufgabe selbst wird sich durch neue Wünsche künftiger Anwender aufgrund der Erfahrungen mit den ersten Programmversionen ändern. Durch die schrittweise Verfeinerung können wir bei Bedarf recht flexibel auf solche Änderungen reagieren.

Die Vorteile der schrittweisen Verfeinerung zeigen sich erst bei der Lösung großer Aufgaben. Deshalb wirkt jedes überschaubare Beispiel dafür eher gekünstelt. Wir wenden diese Strategie dennoch in zwei Varianten auf obiges Beispiel (Einlesen und sortiertes Ausgeben von Zahlen) an, ohne jedoch konkreten Programmcode vorzugeben:

- Zuerst entwickeln wir ein Programm, das nur Zahlen einliest und unsortiert wieder ausgibt.³ Dieses Programm zeigen wir den Anwendern und berücksichtigen in einem nächsten Verfeinerungsschritt die Rückmeldungen hinsichtlich der Benutzerschnittstelle. Schließlich bauen wir in einem letzten Schritt auch das Sortieren ein.
- In einer ganz anderen Art der Aufteilung beschäftigen wir uns zuerst mit dem Sortieren und entwickeln ein Programm, das Zufallszahlen sortiert und zum Testen ausgibt. Erst wenn das Sortieren funktioniert, erweitern wir das Programm um Benutzerschnittstellen (vor allem für das Einlesen), holen Rückmeldungen von den Benutzern ein und berücksichtigen diese in einem letzten Schritt.

Es ist schwer zu sagen, welche der hier skizzierten Vorgehensweisen besser ist. Eine Faustregel besagt, dass man zuerst immer jene Teile einer Aufgabe lösen soll, die am schwierigsten sind und auf die es am ehesten ankommt. Die erste Variante wird man wählen, wenn man die Interaktion mit den Benutzern für den schwierigsten und wichtigsten Teil des Programms hält, die zweite Variante, wenn man das technische Problem des Sortierens für schwierig und entscheidend hält. Diese Schwerpunktsetzung wird im fertigen Programm sichtbar sein. Die erste Variante führt wahrscheinlich zu einer ausgefeilten Benutzerschnittstelle, aber einem eher mittelmäßigen Sortierverfahren. In der zweiten Variante ist das Sortieren im Gegensatz zur Benutzerschnittstelle vermutlich viel ausgereifter.

Auch die Strategie der schrittweisen Verfeinerung kann zu Problemen führen: Bei einer Erweiterung des Programms stellt sich manchmal heraus, dass eine Datenstruktur oder die Faktorisierung des Programms für den neuen Programmteil nicht oder nur unzureichend geeignet ist. In solchen Fällen muss man die Datenstruktur oder Faktorisierung ändern. Letzteres nennt sich *Refaktorisierung* – ein eigener Name, weil Refaktorisierungen

³Bei Verwendung eines binären Baums ist diese Vorgehensweise nicht sehr sinnvoll, da sich eine sortierte Ausgabe praktisch von alleine ergibt. Aber man kann sich trotzdem vorstellen, wie man einen ersten Teil der Aufgabenstellung wählt.

so häufig nötig sind. Solche Änderungen können aufwendig sein. Trotzdem sollte man sie so bald wie möglich machen, da die Änderungen umso aufwendiger sind, je später man sie macht. In Extremfällen kann es passieren, dass in fast jedem Schritt solche Änderungen nötig sind, und die eigentliche Erweiterung des Programms nur sehr langsam oder gar nicht voranschreitet. Daher ist nur schwer planbar, wie lange es dauern wird, bis das Programm fertig ist, und welche Kosten die Fertigstellung verschlingen wird. Trotzdem hat sich die schrittweise Verfeinerung bewährt und wird in unzähligen Softwareprojekten praktisch eingesetzt.

Softwareentwicklungsprozesse auf dem Prinzip der schrittweisen Verfeinerung nennt man *inkrementelle Prozesse*. Dieser Name ist naheliegend. Heute spricht man häufig von *agilen Prozessen*, die Versuchen, die Softwareentwicklung flexibler und schlanker (mit weniger bürokratischem Aufwand) zu gestalten als dies mit traditionellen Entwicklungsprozessen möglich ist. Man möchte sich mehr auf die eigentlichen Ziele konzentrieren. Inkrementelle Prozesse bilden eine gute Basis für agile Prozesse.

4.7 Strukturen programmieren lernen

Das Thema *Algorithmen und Datenstrukturen* ist in der Informatik von großer Bedeutung. Dafür gibt es ein eigenes verpflichtendes Modul, in dem das Thema in wesentlich größerer Tiefe betrachtet wird, als es hier möglich ist. Erfolgreiche Lösungsstrategien und davon abgeleitete Methoden und Techniken werden in Teilen des Moduls *Modellierung* vorgestellt. Die generische Programmierung wird im Modul *Programmierparadigmen* eingehender behandelt.

4.7.1 Konzeption und Empfehlungen

Praktische Vorgehensweisen bei der Lösungsfindung und der Strukturierung von Programmen entwickeln sich im Laufe der Zeit, indem man immer wieder neue Programmieraufgaben löst. Übungsaufgaben sind meist so gewählt, dass man stets mit neuen, bisher unbekannten Aspekten und Problemen konfrontiert wird, für die man eigenständige Lösungen finden muss. Auch wenn es Hilfestellung in der Form gibt, dass die zur Lösung der Aufgaben nötigen Informationen angeboten werden und bei Bedarf Betreuer direkt weiterhelfen können, so erfordert das Lösen dennoch viel Kreativität und Selbstorganisation. Die Aufgaben sind anspruchsvoll und

werden mit der Zeit immer anspruchsvoller. Das ist nötig, damit man durch das Lösen der Aufgaben nicht nur bereits bekannten Stoff wiederholt, sondern auch gefordert ist, stets neue Vorgehensweisen und Techniken auszuprobieren und – bis zu einem gewissen Grad – auch eigene Vorgehensweisen zu entwickeln. Darauf kommt es letztendlich an: Man lernt, bekannte Vorgehensweisen abzuändern und an neue Gegebenheiten anzupassen, sodass sie sich zu eigenen Vorgehensweisen weiterentwickeln. Davon sind auch Algorithmen und Datenstrukturen betroffen. Man lernt, selbständig neue Algorithmen und Datenstrukturen zu entwickeln, wobei man bekannte Strukturen als Ausgangsbasis nimmt.

Die Fähigkeit zum selbständigen Entwickeln von Datenstrukturen und Algorithmen bzw. das selbständige Kombinieren bekannter Datenstrukturen und Algorithmen zu größeren Einheiten kennzeichnet einen ganz wichtigen Sprung beim Programmierenlernen. Sobald man das beherrscht, kann man viele Programmieraufgaben meistern. Das kann man mit dem Erlernen des Lesens und Schreibens vergleichen: Ein gutes Verständnis der Syntax und Semantik grundlegender Sprachkonstrukte sowie der pragmatischen Anwendung dieser Konstrukte ist eine Grundvoraussetzung für das Programmieren, so wie die Kenntnis einzelner Buchstaben und deren Zusammensetzung zu Wörtern bzw. einfachen Wortgruppen eine Grundvoraussetzung für das Lesen und Schreiben ist. Aber es braucht mehr. Die Strukturen in Programmen werden hauptsächlich als Algorithmen und Datenstrukturen sichtbar. Man muss diese Strukturen begreifen und eigene Strukturen entwickeln können, so wie man beim Lesen die Gedanken hinter den Sätzen verstehen und beim Schreiben eigene Gedanken in Sätze kleiden muss. Erst wenn man auch das beherrscht, kann man flüssig Programmieren bzw. Lesen und Schreiben lernen und einen eigenen Programmier- bzw. Schreibstil entwickeln.

4.7.2 Kontrollfragen

- Was versteht man unter Algorithmen und Datenstrukturen, und wie hängen diese beiden Begriffe zusammen?
- Unter welchen Bedingungen sind zwei Algorithmen bzw. Datenstrukturen gleich? Wann sind sie es nicht?
- Nennen Sie fünf unterschiedliche Datenstrukturen.
- Wozu dienen Lösungsstrategien?

- Warum sind so viele Datenstrukturen rekursiv?
- Welche Zugriffsoperationen haben Stacks, verkettete Listen und binäre Bäume üblicherweise?
- Welche charakteristischen Merkmale zeichnen eine verkettete Liste und einen binären Baum aus?
- Wie hängen Datenstrukturen mit gerichteten Graphen zusammen?
- Wodurch unterscheiden sich rekursive Methoden von entsprechenden iterativen (und nicht rekursiven)?
- Was haben rekursive Datenstrukturen und rekursive Methoden mit vollständiger Induktion gemeinsam?
- Wie kann man den Aufwand eines Algorithmus abschätzen?
- Wofür stehen $O(1)$, $O(\log(n))$, $O(n)$, $O(n \cdot \log(n))$, $O(n^2)$ und $O(2^n)$? Wie wirkt sich eine Verdopplung oder Verhundertfachung von n aus?
- Wieso kann man konstante Faktoren bei der Aufwandsabschätzung einfach ignorieren?
- Wie hoch ist der Aufwand für das Einfügen bzw. das Suchen in der verketteten Liste, im binären Baum sowie in der Hashtabelle im Durchschnitt und im schlechtesten Fall? Was ist der jeweils schlechteste Fall und wann tritt er ein?
- Wie funktionieren Bubblesort, Mergesort und Quicksort? Wie hoch ist der Aufwand dafür im Durchschnitt und im schlechtesten Fall?
- Was ist eine binäre Suche?
- Was unterscheidet generische von nicht-generischen Klassen?
- Was unterscheidet einen Typ von einem Typparameter? Kann man Typen und Typparameter gleich verwenden?
- Wie kann man primitive Typen wie `int` als Elementtypen in generischen Containern verwenden?
- Wozu dienen Schranken bei gebundener Generizität?
- Welchen speziellen Zweck hat rekursive gebundene Generizität?
- Inwiefern ähneln sich Untertypbeziehungen und Generizität? Wodurch unterscheiden sie sich in ihrer Anwendbarkeit?
- Was sind und warum verwendet man Iteratoren?
- Welche Schwierigkeiten treten bei der Implementierung von Iteratoren im Zusammenhang mit Rekursion häufig auf? Wie löst man sie?
- Durch welches spezielle Sprachkonstrukt unterstützt Java die Verwendung von Iteratoren?
- Wodurch wird die Verwendung fertiger Programmteile erschwert? Wie kann man den Ursachen dafür begegnen?
- Welche Vor- und Nachteile hat die Top Down Strategie gegenüber der Bottom Up Strategie? Wie lassen sich diese beiden Strategien miteinander kombinieren?
- Für welche Aufgaben bietet sich die schrittweise Verfeinerung an?
- Mit welchen Teilaufgaben sollte man bei schrittweiser Verfeinerung beginnen? Warum ist das so?
- Welche Vorteile und Schwierigkeiten können sich aus der schrittweisen Verfeinerung ergeben?

5 Qualitätssicherung

Zahlreiche Faktoren beeinflussen die Qualität von Programmen – siehe Abschnitt 1.6. Hohe Qualitätsstandards sind nur zu erreichen, wenn die Programmkonstruktion verschiedene Maßnahmen zur Qualitätssicherung einschließt. Das beginnt schon bei der Spezifikation einer Programmieraufgabe. Ein bedeutendes Kriterium ist die gute Verständlichkeit der Spezifikation und des Programms aus statischer Sicht. Verständlichkeit trägt wesentlich zur Fehlervermeidung bei. Zusätzlich müssen wir die Qualität durch geeignetes Testen überprüfen, nicht nur am Ende, sondern wiederholt während der Programmkonstruktion. Wo ein statisches Programmverständnis – beispielsweise zur Feststellung von Fehlerursachen – nicht ausreicht, muss der dynamische Programmablauf im Detail nachvollziehbar sein. Ein sorgfältiger Umgang mit Ausnahmesituationen trägt ebenso zur Qualitätssicherung bei wie verschiedene Formen der Validierung.

5.1 Spezifikationen

Bereits in Abschnitt 1.6.4 haben wir Möglichkeiten gesehen, wie man eine Programmieraufgabe – also die gewünschten Eigenschaften der zu erstellenden Software – spezifizieren kann. Von der Form der Spezifikation hängt unter anderem ab, wie einfach es ist,

- die Spezifikation und das Programm statisch zu verstehen,
- das Übereinstimmen von Spezifikation und Programm zu verifizieren
- und das Programm zu testen.

Bei der Softwareentwicklung erstellt man in der Analysephase eine Anforderungsdokumentation und daraus ein Design und dessen Implementierung. Die Anforderungsdokumentation ist eine Form der Programmspezifikation. Wir verwenden diesen Begriff hier jedoch etwas allgemeiner: Jede Form einer mehr oder weniger rigorosen Beschreibung eines Systems ist eine Spezifikation. Dazu zählt auch die Beschreibung des Designs und die Implementierung selbst. Vor allem zählen auch Zusicherungen dazu.

Die Genauigkeit der Spezifikation eines Systems nimmt während dessen Entwicklung ständig zu: Am Anfang steht oft nur eine Sammlung informeller Beschreibungen von Anwendungsfällen. Mit der Zeit kommt Struktur in diese Sammlung, die teilweise auch formalisierbar ist, und Beschreibungen der Anwendungsfälle werden konkret. Zusammen mit dieser Struktur entwickelt sich eine Faktorisierung des Systems. Alle Teile des Systems werden, wie die Anwendungsfälle, immer genauer spezifiziert. Spätestens durch die Implementierung wird die Spezifikation formal. Durch die Verifikation stellen wir sicher, dass die Implementierung mit einer früheren Form der Spezifikation des Systems übereinstimmt.

5.1.1 Anforderungsspezifikation und Anwendungsfälle

Ein Zweig der Informatik beschäftigt sich mit dem *Requirements Engineering*, also der Entwicklung und Verwaltung der Anforderungen an ein System. Zahlreiche Verfahren werden eingesetzt, um zu einer möglichst klaren Anforderungsspezifikation zu kommen. Hier beschränken wir uns auf einen einfachen Ansatz: Wir beobachten künftige Anwender eines Systems (tatsächlich oder in unserer Vorstellung) und notieren alle Tätigkeiten, welche die Anwender erledigen sollen. Das Ergebnis ist eine Sammlung von *Anwendungsfällen*. Wir ordnen sie und bringen Struktur hinein, und schon haben wir eine Spezifikation des Systems.

Als Beispiel entwickeln wir ein Werkzeug zur Abschätzung der Komplexität eines Java-Programms. Es gibt nur einen Anwendungsfall:

Der Benutzer spezifiziert beim Programmstart die Namen beliebig vieler Java-Quelldateien als Kommandozeilenargumente. Am Bildschirm erscheinen folgende Daten zum Code (oder eine Fehlermeldung, falls eine Datei nicht existiert):

- Anzahl der in den Dateien definierten Klassen
- durchschnittliche Anzahl der nichtleeren Zeilen pro Klasse
- durchschnittliche Anzahl der Kommentarzeilen pro Klasse

Diesen Anwendungsfall kann man schon als Anforderungsspezifikation des Werkzeugs betrachten. Es wird nur beschrieben, *was* der Anwender tut oder erwartet, aber nicht, *wie* die Aufgabe gelöst werden soll.

Bei näherer Betrachtung stellt sich heraus, dass einige Details dieser Spezifikation nicht klar sind oder vielleicht sogar anders gedacht waren, als zu lesen ist. Wir treffen einige Annahmen zur Klarstellung der Aufgabe:

- Das Wort „Klassen“ sollte durch „Klassen bzw. Interfaces“ ersetzt werden, da die Ergebnisse damit an Aussagekraft gewinnen. Wir gehen davon aus, dass Interfaces ohne Absicht unerwähnt blieben.
- Wir betrachten alle Zeilen als leer, die nur white space enthalten. Gezählt werden sollen also nur Zeilen, die auch etwas anderes als Leerzeichen und Tabulatorzeichen enthalten.
- Als Kommentarzeilen betrachten wir Zeilen, die entweder „/ /“, „/*“ oder „*/“ außerhalb von Strings enthalten. Auch nichtleere Zeilen zwischen „/*“ und „*/“ zählen zu den Kommentarzeilen.
- Programmaufrufe ohne Dateien sollen zu Fehlermeldungen führen.

Nun stellt sich die Frage, was eigentlich die „richtige“ Anforderungsspezifikation ist, die ursprüngliche oder die verbesserte Spezifikation. Die Antwort darauf hängt von der Rolle ab, welche die Anforderungsspezifikation spielt. Oft handelt es sich dabei um einen Bestandteil eines Vertrages zwischen einem Auftraggeber und Auftragnehmer. In diesem Fall ist die Sache klar: Verträge müssen eingehalten werden, auch wenn zum Zeitpunkt der Vertragserstellung noch viele Details offen sind. Wir können die Spezifikation nicht ohne Weiteres durch Hinzunahme des ersten sowie des letzten Punktes obiger Annahmen ergänzen, da dies mit einer inhaltlichen Änderung einhergehen würde. Allerdings ist es durchaus möglich, dass sich Auftraggeber und Auftragnehmer auch noch nach Abschluss des Vertrages auf solche Vertragsänderungen einigen. Eine Ergänzung um den zweiten und dritten Punkt wäre dagegen problemlos möglich, weil nur unklare Begriffe deutlicher gemacht werden. Natürlich können Auftraggeber und Auftragnehmer auch gemeinsam eine genauere Klärung der Begriffe erarbeiten, ein neuer Vertrag entsteht dadurch nicht notwendigerweise.

Ähnlich verhält es sich bei Programmieraufgaben, die im Rahmen einer Übung zu lösen sind. Man muss sich an vorgegebene Spezifikationen halten, auch wenn man bei der Bearbeitung bemerkt, dass Verbesserungsmöglichkeiten bestehen. In Extremfällen kann man nachfragen, wenn die Aufgabenstellung unlogisch erscheint. Was nicht im Detail genau festgelegt ist, bietet einen Interpretationsspielraum, den man ausnutzen kann.

Es ist der Normalfall, wenn Spezifikationen im Laufe der Zeit genauer werden. Daher gestaltet man Verträge über die Entwicklung von Software oft auch derart, dass zwar klare Ziele festgelegt werden, aber Details der Anforderungsspezifikation offen bleiben. In diesen Fällen ist es wenig

sinnvoll, nur die groben Vorgaben in die Verifikation einzubeziehen. Man wird während der Entwicklung genauere Anforderungsspezifikationen für alle Teile des Systems erstellen und als Grundlage für die Verifikation verwenden. Jede Anforderungsspezifikation ist ein Vertrag, auch wenn der Vertrag nicht notwendigerweise zwischen Firmen oder Personen als Auftraggeber und Auftragnehmer abgeschlossen sein muss.

In der Programmkonstruktion verwenden wir Verträge ganz allgemein zur Festlegung von Anforderungsspezifikationen auf allen Ebenen, vom gesamten Programm über Objekte (repräsentiert durch Interfaces und Klassen) bis zu einzelnen Methoden. Diese Verträge legen auch die Schnittstellen zwischen den Programmteilen fest. Beispielsweise legen Interfaces und Klassen in Java die formalen Teile solcher Verträge zwischen Objekten fest. Vertragspartner sind Objekte – einerseits die Instanzen, welche die beschriebenen Methoden bereitstellen, andererseits Objekte, die diese Methoden aufrufen. Die Objekte, welche die Methoden bereitstellen, spielen die Rolle eines *Servers* (Anbieters von Dienstleistungen bzw. Auftragnehmers), während die Objekte, welche die Methoden aufrufen, die Rolle eines *Clients* (Kunden oder Auftraggebers) spielen. Vertragsbestandteile sind die Namen der Methoden, die Typen der übergebenen aktuellen Parameter sowie die Typen der Ergebnisse. Allerdings reichen diese formalen Vertragsbestandteile nicht aus, um das Verhältnis zwischen Client und Server vollständig zu spezifizieren. Daher enthalten Interfaces und Klassen in der Regel zusätzlich informelle Spezifikationen in Form von Kommentaren. Obwohl es sich „nur“ um Kommentare handelt, sind sie als wichtige Vertragsbestandteile Ernst zu nehmen.

5.1.2 Design by Contract

Zusicherungen auf Methoden haben wir in Kapitel 1 kennen gelernt. Wir unterscheiden *Vorbedingungen*, die bereits vor Ausführung einer Methode erfüllt sein müssen, von *Nachbedingungen*, die erst nach der Methodenausführung erfüllt sein müssen. In Abschnitt 3.5 haben wir gesehen, wie die Zusicherungen auf Objekte übertragen werden können und *Invarianten* quasi unveränderliche Eigenschaften beschreiben. Dabei ging es vor allem um die Rolle von Zusicherungen als Kommunikationsmittel zwischen Personen. Nun wollen wir diesen Blickwinkel erweitern und Zusicherungen als Bestandteile eines Vertrags zwischen einem Client und Server betrachten (der bei der Entwicklung der Klassen für Client und Server einzuhalten ist). Generell sieht ein Vertrag folgendermaßen aus:

Der Client kann durch Senden einer Nachricht an den Server einen angebotenen Dienst in Anspruch nehmen, wenn

- *die Schnittstelle des Servers eine entsprechende Methode beschreibt,*
- *jeder Argumenttyp in der Nachricht Untertyp des entsprechenden formalen Parametertyps der Methode ist*
- *und alle Vorbedingungen der Methode erfüllt sind.*

Der Server wird unter diesen Bedingungen die Methode ausführen und sicherstellen, dass unmittelbar nach Ausführung

- *eine Instanz des Ergebnistyps als Antwort zurückkommt*
- *und alle Nachbedingungen der Methode und Invarianten des Servers erfüllt sind.*

Entsprechend dem Vertrag kann sich der Server darauf verlassen, dass der Client für die Einhaltung der Vorbedingungen vor jeder Ausführung einer Methode sorgt. Der Client kann sich darauf verlassen, dass der Server für die Einhaltung der Nachbedingungen und Invarianten am Ende der Ausführung einer Methode sorgt. Es ist also klar geregelt, wer wofür zuständig ist und worauf man sich verlassen darf. Genau wegen dieser klaren Regelungen sind Zusicherungen für die Spezifikation in objektorientierten Programmen so wichtig, und man gibt den Zusicherungen eine zentrale Rolle im Entwurf von Klassen und ganzen Systemen.

Die Vorgehensweise, bei der man Klassen und Systeme durch solche Verträge beschreibt, nennt man *Design by Contract*. Wie der Name schon andeutet, gehört die Erstellung solcher Verträge zur Entwurfsphase, nicht zur Analysephase. Die Spezifikationen müssen eindeutig und recht genau sein, damit die Verträge ihren Zweck erfüllen können. Dieser Aspekt unterscheidet sie von den meist eher allgemein gehaltenen Spezifikationen, die als Anforderungsspezifikationen eine Grenze zwischen der Analyse- und Entwurfsphase bilden. Oft entstehen die wichtigsten und zentralsten Verträge für Design by Contract zu dem Zeitpunkt, an dem zusammen mit der Faktorisierung eines Systems Interfaces entwickelt werden. Viele weitere, eher weniger zentrale Verträge entstehen gleichzeitig mit der Implementierung von Klassen. Wenn wir Interfaces und Klassen entwickeln, sind nur wir selbst auch für die Entwicklung der Verträge verantwortlich. Entsprechend Design by Contract soll jedes Stückchen Code, das wir schreiben, einem Vertrag entsprechen. Dadurch gibt Design by Contract ein Denkmuster vor, das uns beim Entwickeln von Code leitet: Wir müssen stets

daran denken, welche Vertragsbestandteile die Methoden, die wir schreiben, erfüllen müssen, aber auch, was die Methoden, die wir aufrufen, von Clients erwarten. Andererseits dürfen wir ohne weitere Prüfung davon ausgehen, dass die Vorbedingungen der Methoden, die wir schreiben, erfüllt sind, und aufgerufene Methoden das machen, was ihre Nachbedingungen und die Invarianten versprechen.

Design by Contract verbessert vor allem das statische Verständnis des Programmcodes. Man weiß, was man sich vom Aufruf einer Methode erwarten kann, ohne den dabei ausgeführten Code nachvollziehen zu müssen.

Design by Contract gibt klare Richtlinien vor, in welcher Beziehung Zusicherungen in Unter- und Obertypen zueinander stehen müssen, damit Verträge erfüllbar sind:

- Vorbedingungen in Untertypen dürfen schwächer, aber nicht stärker als entsprechende Bedingungen in Obertypen sein. Wenn eine Vorbedingung im Obertyp beispielsweise $x > 0$ lautet, darf die entsprechende Bedingung im Untertyp $x \geq 0$ sein. Im Untertyp steht die Verknüpfung beider Bedingungen mit ODER.
- Nachbedingungen und Invarianten in Untertypen dürfen stärker, jedoch nicht schwächer als die Bedingungen in Obertypen sein. Wenn eine Nachbedingung oder Invariante im Obertyp z.B. $x \geq 0$ lautet, darf die entsprechende Bedingung im Untertyp $x > 0$ sein. Im Untertyp steht die Verknüpfung beider Bedingungen mit UND.

Diese Beziehungen stellen sicher, dass sich eine Instanz eines Untertyps so verhält, wie man es sich von einer Instanz des Obertyps erwartet. Über den Vergleich von Zusicherungen im Unter- und Obertyp wird daher die in Kapitel 3 aufgestellte Forderung nach gleichem Verhalten erfüllt, wobei die Spezifikation im Untertyp trotzdem genauer sein kann als die im Obertyp.

Zahlreiche Beispiele für Zusicherungen finden wir in den Beschreibungen der Java APIs (Application Programmer Interfaces).¹ Fast alle Zusicherungen sind als beschreibende Texte formuliert, ohne klare Unterscheidung zwischen Vor- und Nachbedingungen sowie Invarianten. Inhalte bestimmen, mit welcher Art von Zusicherung wir es zu tun haben:

- Einschränkungen auf formalen Parametern sowie alles, um das sich Aufrufer von Methoden kümmern müssen, sind Vorbedingungen.

¹Siehe beispielsweise die Beschreibungen der Klassen und Interfaces der Java Standard Edition 6 unter <http://download.oracle.com/javase/6/docs/api/>.

- Beschreibungen unveränderlicher Eigenschaften von Objektzuständen stellen Invarianten dar.
- Alles andere sind Nachbedingungen. Sie beschreiben, was die Methoden tun, und machen meist den Großteil der Zusicherungen aus.

Für die Einhaltung von Nachbedingungen und Invarianten ist der Server zuständig. Bei Invarianten ergibt sich das Problem, dass der Server möglicherweise keine vollständige Kontrolle über den Zustand des Objekts hat und die Invarianten deswegen gar nicht zusichern kann. Das ist der Fall, wenn Objektvariablen nicht nur von den Methoden des Objekts verändert werden, sondern auch von anderen Methoden. Die anderen Methoden könnten Invarianten verletzen, die sie gar nicht kennen. Aus diesem Grund sollte man *niemals schreibend auf Objektvariablen eines anderen Objekts zugreifen*. Ausnahmen von dieser Regel kann man nur machen, wenn man (beispielsweise bei privaten Variablen eines anderen Objekts derselben Klasse) die Zusicherungen des anderen Objekts genau kennt und einhält. Deshalb sollten Objektvariablen als `private` deklariert werden.

Java unterstützt `assert`-Anweisungen für überprüfte Zusicherungen. Für Vor- und Nachbedingungen sowie Invarianten spielen überprüfte Zusicherungen praktisch keine Rolle, weil viele der üblicherweise im Text ausgedrückten Eigenschaften nur sehr umständlich formalisierbar sind. Es liegt an uns, beim Programmieren auf die Einhaltung der Bedingungen zu achten. Einige Programmiersprachen wie *Eiffel* oder *D* bieten viel weiter gehende Sprachunterstützung für überprüfte Zusicherungen an. Aber auch in diesen Sprachen kann man nicht alle Zusicherungen in der Sprache ausdrücken, die wir gerne ausdrücken würden. Außerdem ist es zu spät, wenn falsche Zusicherungen erst zur Laufzeit erkannt werden. Eigentlich wollen wir solche Fehler statisch erkennen können.

5.1.3 Abstraktion und Intuition

So manche Bedingung lässt sich statt als Zusicherung auch als *Typ* ausdrücken. Das hat den Vorteil, dass der Compiler die Kompatibilität der Typen zueinander statisch überprüft und einen Fehler meldet, wenn Typen nicht zusammenpassen. Kommentare können altern, also nach Programmänderungen plötzlich nicht mehr stimmen. Mit Typen kann das in einem compilierbaren Programm nicht passieren.

Listing 5.1: Berechnung des Medians in sortiertem Array

```
// gib mittlere Zahl in nums zurück; nums sortiert
public static int median (int[] nums) {
    return nums[nums.length / 2];
}
```

Listing 5.2: Eigener Typ für sortiertes Array ganzer Zahlen

```
1 import java.util.Arrays
2 public class SortedIntArray {
3     private int[] elems; // elems bleibt stets sortiert
4     public SortedArray (int[] e) {
5         elems = Arrays.copyOf (e, e.length);
6         Arrays.sort(elems);
7     }
8     public int median() { // gib mittlere Zahl zurück
9         return elems[elems.length / 2];
10    }
11    public boolean member (int x) { // x enthalten?
12        ... /* binäre Suche */
13    }
14 }
```

Betrachten wir ein Beispiel dafür. Die statische Methode in Listing 5.1 soll den Median, also die Zahl mit dem mittleren Wert in einem Array ermitteln. Die Vorbedingung „nums sortiert“ stellt sicher, dass die Nachbedingung „gib mittlere Zahl in nums zurück“ eingehalten wird. Trotzdem ist diese Lösung nicht sehr befriedigend. Wird `median` unter Verletzung der Vorbedingung mit einem unsortierten Array aufgerufen, wird nur irgendeine zufällige Zahl im Array zurückgegeben. Solche Fehler sind schwer zu finden, da der falsche Aufruf überall erfolgen kann. Überprüfte Zusicherungen sind hier kaum sinnvoll einsetzbar, da häufig wiederholte Überprüfungen, ob ein Array sortiert ist, ziemlich aufwendig wären.

Listing 5.2 zeigt einen Ansatz zur Lösung dieses Problems: Wir packen das Array in ein eigenes Objekt und schränken Zugriffe auf das Array so ein, dass die gewünschte Eigenschaft (Sortiertheit) stets erhalten bleibt. Das im Konstruktor übergebene Array wird vor dem Sortieren kopiert um zu verhindern, dass es von außerhalb der neuen Instanz von

`SortedIntArray` verändert werden kann. Eine Invariante auf `elems` stellt sicher, dass das Array stets sortiert bleibt. Im Gegensatz zur Lösung mittels statischer Methode kennen wir alle Stellen im Programm, an denen die Zusicherung möglicherweise verletzt werden könnte, da `elems` nur innerhalb der Klasse sichtbar ist. Neben der Methode `median` können auch andere Methoden in der Klasse sinnvoll sein, die auf ein sortiertes Array angewiesen sind, beispielsweise eine binäre Suche. Auch Methoden zum Ändern des Arrays wären sinnvoll; sie müssen nur garantieren, dass das Array nach jeder Änderung noch immer sortiert ist.

Um den Median berechnen zu können, müssen wir mit diesem Lösungsansatz eine Instanz von `SortedIntArray` erzeugen. Der Compiler garantiert, dass `median` ohne ein solches Objekt nicht aufrufbar ist. Auf diese Weise hilft der Compiler, die Einhaltung der Invariante sicherzustellen. Statt einem Typ `int[]` zusammen mit einer Zusicherung „Array ist sortiert“ verwenden wir an vielen Stellen im Programm einfach nur den Typ `SortedIntArray` ohne Zusicherung. Das ist einfacher und sicherer.

Bei näherer Betrachtung stellen wir fest, dass `SortedIntArray` eine Abstraktion darstellt, also eine abstrakte Maschine mit bestimmten Eigenschaften. Wenn wir viele Zusicherungen brauchen, um das Gewünschte auszudrücken, ist das ein Hinweis darauf, dass sich das Programm durch Einführung neuer Klassen verbessern lässt. Diese Klassen kapseln die Zusicherungen. Ein gut strukturiertes Programm kommt meist mit nur wenigen Zusicherungen aus.

Gute Zusicherungen sind intuitiv klar. Es ist logisch, von einem Array in einer Klasse namens `SortedIntArray` zu verlangen, dass es sortiert ist. Namen von Klassen, Methoden, Variablen und Parametern kommt große Bedeutung zu, weil gut gewählte Namen die Intuition hinter den Zusicherungen klar machen. Zusicherungen, die man ohnehin aus den Namen ableitet, braucht man gar nicht hinzuschreiben. Beispielsweise sollte auch ohne Kommentare klar sein, dass eine Methode namens `sort` auf einer Datenstruktur die Elemente der Datenstruktur sortiert. Gut gewählte Namen machen ein Programm einfach lesbar. Die Intuition hinter den Namen bildet einen wichtigen Teil der Spezifikation eines Systems.

In der objektorientierten Programmierung sind Namen besonders wichtig: Softwareobjekte simulieren Objekte aus der realen Welt, und Namen setzen die Softwareobjekte in Relation zu realen Objekten. Aufgrund unserer Erfahrungen in der realen Welt verstehen wir die Software. In Java beruhen auch die Äquivalenz von Typen sowie Untertypbeziehungen auf Namen. Prinzipiell unterscheidet man zwei Arten solcher Beziehungen:

- Typäquivalenz aufgrund von Namensgleichheit bzw. explizite Untertypbeziehungen: Zwei Typen sind genau dann gleich, wenn die Typen dieselben Namen haben. Zwei Typen stehen genau dann in einer Untertypbeziehung, wenn eine explizite Beziehung zwischen den Namen dieser Typen (durch `extends`- und `implements`-Klauseln) hergestellt wurde. Stark typisierte objektorientierte Sprachen wie Java, C# und C++ verwenden hauptsächlich diese Konzepte.
- Typäquivalenz aufgrund von Strukturgleichheit bzw. implizite Untertypbeziehungen: Zwei Typen gelten als gleich, wenn ihre Instanzen dieselbe Struktur haben (unabhängig von Typnamen). Zwei Typen sind in Untertypbeziehung, wenn ein Typ zumindest alle Methoden unterstützt, die auch der andere unterstützt (auch ohne `extends`- oder `implements`-Klausel). Dynamisch typisierte objektorientierte Sprachen wie Ruby, Python und Smalltalk verwenden implizite Untertypbeziehungen. Typäquivalenz aufgrund von Strukturgleichheit wird aber beispielsweise auch in Teilen von C und C++ verwendet.

Programme in dynamischen objektorientierten Sprachen beruhen häufig auf *duck typing*. Diese Konzept ist nach einem Gedicht benannt, welches das Konzept gut beschreibt:

*When I see a bird that walks like a duck and swims like a duck
and quacks like a duck, I call that bird a duck.*

– James Whitcomb Riley

Anders formuliert: Wenn mein Objekt zumindest alle Methoden hat, die ich von einer Instanz von `Duck` erwarte, dann kann ich das Objekt als Instanz von `Duck` verwenden. Es könnte reiner Zufall sein, dass alle Methoden von `Duck` vorhanden sind. Das macht nichts, solange sich das Objekt so wie erwartet verhält. In stark typisierten Sprachen möchte man dagegen nicht von einer solchen Art von Zufall abhängig sein. Ein Objekt ist nur dann eine Instanz von `Duck`, wenn es durch `new X(...)` erzeugt wurde, wobei `X` gleich `Duck` oder ein explizit deklarierter Untertyp von `Duck` ist (beispielsweise durch `class X extends Duck { ... }`).

Der technische Unterschied zwischen expliziten Untertypbeziehungen und *duck typing* ist folgender: *duck typing* beruht nur auf der Unterstützung gleichartiger Methoden, während explizite Untertypbeziehungen zusätzlich auf weiteren Kriterien, die das Programm vorgibt, beruhen können. Im Wesentlichen werden diese zusätzlichen Kriterien die in Zusiche-

rungen ausgedrückten Eigenschaften sein. *Duck typing* ignoriert Zusicherungen, während explizite Untertypbeziehungen Zusicherungen berücksichtigen können. Auch explizite Untertypbeziehungen berücksichtigen Zusicherungen nur dann, wenn wir beim Programmieren dafür sorgen, dass die Zusicherungen zwischen Unter- und Obertypen zusammenpassen.

In der Praxis sind viele Zusicherungen gar nicht explizit ausgedrückt, nichteinmal als Kommentare. Sie werden nur durch Namen impliziert. Untertypbeziehungen müssen auch diese impliziten Zusicherungen erfüllen. Hier hilft uns wieder die Relation zur realen Welt. Menschen mit Programmiererfahrung sind meist recht gut darin, übliche Untertypbeziehungen zu erraten, auch wenn sie die Zusicherungen gar nicht im Detail betrachten. Das wird in der objektorientierten Modellierung ausgenutzt.

5.2 Statisches Programmverständnis

Eine Voraussetzung für die Steigerung der Qualität eines Programms ist ein gutes Verständnis des Programms. Neben der komplexen Semantik einiger Sprachkonstrukte macht es vor allem die riesige Zahl an möglichen Programmzweigen schwierig, den Überblick über ein Programm zu behalten. Wir beschäftigen uns hier mit Techniken, die uns das Verstehen eines Programms erleichtern. Dabei konzentrieren wir uns auf eine statische Sichtweise: Wir achten darauf, was bei Ausführung des Programms stets gleich bleibt, nicht auf das, was sich ständig ändert.

5.2.1 Typen und Zusicherungen

Aus deklarierten Typen können wir recht viel statische Information herauslesen. Der deklarierte Typ einer Variablen bleibt immer gleich, egal wie wir die Variable verwenden. Er beschreibt vor allem die Nachrichten, die vom Objekt in der Variablen verstanden werden. Auch die Vor- und Nachbedingungen der Nachrichten sowie Invarianten auf dem Objekt sind durch den deklarierten Typ ein für allemal festgelegt. Durch diese Eigenschaften erhöhen Typen die Lesbarkeit von Programmen. Sogar in dynamischen Sprachen, in denen Variablen ohne Typen deklariert werden, bekommen Variablen häufig Namen wie `sortedIntArray`, die den Typ des in der Variable enthaltenen Wertes beschreiben.

In stark typisierten Sprachen überprüft der Compiler die Kompatibilität zwischen den Typen. Die Kompatibilität zwischen Zusicherungen müssen

wir selbst überprüfen. Diese *Verifikation* machen wir, indem wir für jede Methode die Nachbedingungen und Invarianten aus den Anweisungen im Rumpf der Methode ableiten, wobei wir annehmen, dass die Vorbedingungen und Invarianten am Anfang erfüllt sind. Außerdem müssen wir uns vergewissern, dass bei jedem Methodenaufruf die Vorbedingungen der aufgerufenen Methode sowie Invarianten des Empfängers der Nachricht sowie aller Argumente erfüllt sind. Diese Überprüfungen können recht aufwendig sein. Zur Vereinfachung der Herleitung aller notwendiger Bedingungen können wir – sozusagen als Zwischenschritte im Beweis – weitere Zusicherungen in den Rumpf der Methode schreiben.

Listing 5.3 gibt ein Beispiel dafür. Die Klasse `SortedIntArray` aus Listing 5.2 wird um eine binäre Suche ähnlich der in Listing 4.22 erweitert. Um die Zusicherungen auf eine unzweideutige, formale Basis zu stellen, verwenden wir `assert`-Anweisungen. Hauptsächlich werden die Überprüfungen von eigens dafür implementierten privaten Methoden vorgenommen. Die Methode `elemsSorted` überprüft in einer einfachen Schleife, ob das Array `elems` sortiert ist. Zu Beginn und am Ende (also vor jeder `return`-Anweisung) muss die Invariante erfüllt sein. Da `elems` in einer Ausführung von `member` nirgends verändert wird, können wir davon ausgehen, dass das Array am Ende noch immer sortiert ist, wenn es zu Beginn sortiert war. Zu Beginn ist es sortiert, weil es am Ende jeder Methode von `SortedIntArray` sortiert ist und außer den Methoden dieser Klasse niemand auf das Array zugreifen kann.

Die Nachbedingung auf `member` könnte so klingen: „Das Ergebnis ist `true` wenn `x` in `elems` vorkommt und `false` wenn `x` in `elems` nicht vorkommt.“ In Listing 5.3 ist die Nachbedingung jedoch anders formuliert: `x == elems[k]` (für irgendein `k`) impliziert `true` als Ergebnis, und `clean(x, i, j) && i > j` impliziert `false`. Dabei stellt der Aufruf von `clean(x, i, j)` sicher, dass `x` in `elems` außerhalb des Indexbereiches `i` bis `j` nicht vorkommt. Im Spezialfall von `i > j` kann `x` in `elems` überhaupt nicht vorkommen. Diese etwas umständliche Ausdrucksweise hilft bei der Überprüfung, ob die Bedingungen zutreffen. Zur weiteren Unterstützung dient die Zusicherung `clean(x, i, j)` zu Beginn jeden Schleifendurchlaufs. Für den ersten Schleifendurchlauf ist diese Bedingung natürlich erfüllt, da `clean(x, 0, elems.length - 1)` immer `true` zurückgibt. In jedem Schleifendurchlauf wird irgendein Index `k` zwischen `i` und `j` gewählt, und falls `x` ungleich `elems[k]` ist, wird `k` zur neuen unteren oder oberen Grenze, je nach Größe des Wertes in `elems[k]`. Aufgrund der

Listing 5.3: Binäre Suche – Zusicherungen zur Überprüfung der Korrektheit

```
public class SortedIntArray {
    private int[] elems;           // elems stets sortiert
    ...
    public boolean member (int x) { // x enthalten?
        assert elemsSorted();      // Invariante hält zu Beginn
        int i = 0;                 // untere Grenze
        int j = elems.length - 1; // obere Grenze
        while (i <= j) {           // bis alles durchsucht:
            assert clean (x, i, j);
            int k = i + ((j - i) / 2); // probiere die Mitte
            if (x < elems[k]) {
                j = k - 1;          // links weitersuchen
            } else if (x == elems[k]) {
                assert elemsSorted(); // Invar. hält am Ende
                assert x == elems[k]; // x in elems => true
                return true;
            } else {
                i = k + 1;          // rechts weitersuchen
            }
        }
        assert elemsSorted();      // Invariante hält am Ende
        assert clean(x, i, j) && i > j; // x nicht in elems => false
        return false;
    }
    private boolean elemsSorted() { // ist elems sortiert?
        for (a = 1; a < elems.length; a++)
            if (elems[a - 1] > elems[a])
                return false;
        return true;
    }
    private boolean clean (int x, int i, int j) {
        while (--i >= 0)           // x nicht in elems bis i-1
            if (elems[i] == x)
                return false;
        while (++j < elems.length) // x nicht in elems ab j+1
            if (elems[j] == x)
                return false;
        return true;
    }
}
```

Sortiertheit von `elems` und der Auswahl der Grenzen gilt auch für die neuen Grenzen `clean(x, i, j)`, auch dann, wenn die Schleife wegen `i > j` abbricht. Damit haben wir schon die Nachbedingung gezeigt.

Bei dieser statischen Analyse der Methode haben wir viele Details ignoriert. Nachbedingung und Invariante hängen beispielsweise nicht davon ab, wie `k` gewählt wird. Solche Aspekte sind zwar für die Effizienz der Methode wichtig, aber nicht, um die Korrektheit hinsichtlich der Zusicherungen zu überprüfen. Generell wird das Programmverständnis erleichtert, wenn wir einen Aspekt nach dem anderen betrachten, nicht alle gleichzeitig.

Eigentlich brauchen wir keine einzige der `assert`-Anweisungen in Listing 5.3 (genausowenig wie die Methoden `elemsSorted` und `clean`) da wir aufgrund unserer Analysen wissen, dass diese Zusicherungen nicht verletzt sein können. Das ist typisch für alle empfohlenen Anwendungen von `assert`-Anweisungen: Man soll sie nur dort einsetzen, wo man weiß, dass die Bedingungen immer wahr sind. Wenn Zweifel daran bestehen, dass eine Bedingung erfüllt ist, kann man die Bedingung beispielsweise in einer `if`-Anweisung einsetzen, aber nicht in einer `assert`-Anweisung. Nicht erfüllte `assert`-Anweisungen führen in der Regel ja zum Programmabbruch. In Abschnitt 5.4 werden wir sehen, wie verletzte `assert`-Anweisungen beim Finden von Fehlern helfen können. Sie sind also nicht sinnlos. Die Methoden `elemsSorted` und `clean` dienen in erster Linie zur genauen Spezifikation der Zusicherungen, nicht dazu ausgeführt zu werden.

Die Überprüfung von Zusicherungen zur Laufzeit kann sehr aufwendig sein. Beispielsweise hat die normale binäre Suche nur einen Aufwand von $O(\log(n))$, aber $O(n \cdot \log(n))$ wenn Zusicherungen wie in Listing 5.3 überprüft werden. Ein solcher Aufwand ist meist inakzeptabel. Daher ist die Überprüfung von Zusicherungen fast immer ausgeschaltet. Wenn man Zusicherungen überprüfen lassen möchte, muss man beim Aufruf des Interpreters das Flag `-ea` setzen. Das heißt, wird das Programm durch „`java -ea Program`“ gestartet, werden `assert`-Anweisungen überprüft, wird es durch „`java -da Program`“ oder „`java Program`“ gestartet, ist die Überprüfung ausgeschaltet und das Programm läuft effizienter. Weil `assert`-Anweisungen nicht immer überprüft werden, dürfen sie keine Ausdrücke enthalten, die für die korrekte Programmausführung notwendig sind, sondern nur eigentlich überflüssige Überprüfungen.

5.2.2 Invarianten

Bisher haben wir den Begriff *Invariante* nur für eine bestimmte Form von Zusicherungen auf Objektschnittstellen verwendet. Diese Invarianten beschreiben Eigenschaften von Objektzuständen, die zu Beginn und am Ende jeder Methodenausführung erfüllt sein müssen. Daneben gibt es auch

Schleifeninvarianten. Das sind Eigenschaften, die zu Beginn und am Ende jeden Schleifendurchlaufs erfüllt sein müssen. In Listing 5.3 ist beispielsweise `clean(x, i, j)` eine Invariante auf der Schleife in `member`.

Schleifen sind oft besonders schwierig aus statischer Sicht zu verstehen, da der Programmfortschritt nur durch wiederholte dynamische Veränderungen des Zustandes erfolgen kann; Veränderungen sind nicht statisch. Meistens wissen wir nicht, wie oft eine Schleife durchlaufen wird. Daher ist es auch nicht möglich, die Schleife zu verstehen, indem wir sie gedanklich *ausrollen* (englisch *loop unrolling*), also die Schleife durch so viele hintereinander auszuführende Kopien des Schleifenrumpfes ersetzen, sooft die Schleife durchlaufen wird. Schleifeninvarianten bleiben von dynamischen Änderungen verschont und ermöglichen ein statisches Verständnis.

Generell funktionieren Schleifeninvarianten nach folgendem Schema: Eine Bedingung I ist eine Schleifeninvariante, wenn I zu Beginn und am Ende jedes Durchlaufs durch den Schleifenrumpf R erfüllt ist. Dadurch ist I auch vor dem ersten und nach dem letzten Schleifendurchlauf erfüllt. Für eine `while`-Schleife mit der Abbruchbedingung A gilt also:

```
assert I;
while (!A) { assert I; R; assert I; }
assert I && A;
```

Nach Beendigung der Schleife ist nicht nur die Invariante, sondern auch die Abbruchbedingung erfüllt. Die Schleifeninvariante `clean(x, i, j)` in Listing 5.3 wird genau nach diesem Schema verwendet, auch wenn aus Gründen der Vereinfachung einige `assert`-Anweisungen weggelassen wurden. Wie das Beispiel zeigt, müssen Schleifeninvarianten nicht unbedingt konstant sein. Die Variablen `i` und `j` können in jedem Schleifendurchlauf unterschiedliche Werte haben, aber die Bedingung gilt trotzdem.

Hinter Schleifeninvarianten steckt die vollständige Induktion als Beweistechnik: Den Induktionsanfang bildet die Erfüllung der Invarianten vor Schleifenbeginn. Der Induktionsschritt besteht darin, dass die Invariante auch für den $(n + 1)$ -ten Schleifendurchlauf gelten muss, wenn sie für den n -ten Durchlauf gilt. Das ist gewährleistet, weil die Invariante am Anfang und Ende jeden Schleifendurchlaufs gelten muss.

In der Praxis ist es oft schwierig, in einem bestehenden Programm passende Schleifeninvarianten zu finden. Wenn man sie einmal gefunden hat, ist es in der Regel leicht, die Korrektheit des Programms zu verifizieren. Andere Sprachkonstrukte sind einfacher statisch zu verstehen. Bei der Suche nach Schleifeninvarianten geht man oft von den Nachbedingungen aus,

die am Ende der Methodenausführungen gelten müssen. Sie liefern gute Hinweise darauf, welche Schleifeninvarianten benötigt werden, um das Programm verifizieren zu können. Dann sucht man nach einer Möglichkeit, diese Nachbedingungen zu erfüllen. Invarianten auf Objektschnittstellen eignen sich meist auch als Schleifeninvarianten, sind aber leider oft trivial (beispielsweise „`elems` wird nicht verändert“) und helfen daher kaum bei den schwierigen Teilen des Beweises der Nachbedingungen.

Während der Konstruktion eines Programms wissen wir, warum wir eine Schleife einsetzen und was diese Schleife bezweckt. Genau diesen Zweck können und sollen wir als Kommentar in das Programm schreiben. Solche Kommentare machen deutlich, wie wir beim Programmieren denken. Mit weniger Programmiererfahrung verdeutlichen die Kommentare eher den dynamischen Ablauf, der in der Syntax der Schleifen ohnehin gut erkennbar ist. Mit zunehmender Programmiererfahrung spiegeln die Kommentare immer stärker eine statische Denkweise wider. Im Idealfall lässt sich jeder solche Kommentar als Schleifeninvariante lesen. Dann ist es nur mehr ein kleiner Schritt von informellen Kommentaren zu formalen `assert`-Anweisungen. Es gilt also: Die Suche nach guten Schleifeninvarianten können wir bedeutend vereinfachen, wenn wir uns eine statische Denkweise bereits beim Schreiben der Schleife angewöhnen und die wichtigsten Gedanken als Kommentare niederschreiben.

Schleifen können durch Rekursion ersetzt werden. Rekursive Aufrufe werden durch geeignete Zusicherungen verständlich. Statt Schleifeninvarianten verwenden wir jedoch Vor- und Nachbedingungen bzw. Invarianten auf Objektschnittstellen. Ein weiterer wichtiger Unterschied besteht darin, dass rekursive Aufrufe überall innerhalb eines Methodenrumpfs erfolgen können, nicht nur wie bei Schleifen am Ende des Rumpfs. Entsprechend müssen Vorbedingungen und Invarianten unmittelbar vor jedem (rekursiven) Aufruf erfüllt sein. Während dieselbe Schleifeninvariante am Anfang und Ende einer Schleife erfüllt sein muss, können sich Vor- und Nachbedingungen einer Methode voneinander unterscheiden. Vor und nach rekursiven Aufrufen können ja noch weitere Anweisungen ausgeführt werden, die sich auf die Bedingungen auswirken. Damit geben uns rekursive Methoden etwas mehr Freiheit beim Programmieren und bei der Dokumentation von Programmen als Schleifen.

Bei der Konstruktion rekursiver Methoden gehen wir am besten genauso vor wie bei der von Schleifen: Wir halten den Zweck der Methoden und von rekursiven Aufrufen in Kommentaren fest und achten darauf, dass diese Kommentare eine möglichst statische Sichtweise widerspiegeln.

Obwohl in diesem Abschnitt von Verifikation und Beweisen gesprochen wird, dürfen wir nicht vergessen, dass das wichtigste Ziel hinter Zusicherungen die Verbesserung der Verständlichkeit des Programms ist. Beweisverfahren brechen komplizierte Aussagen auf eine Menge so einfacher Aussagen herunter, dass niemand an deren Korrektheit zweifeln kann. Genauso erklären Zusicherungen komplizierte Programmteile auf so einfache Weise, dass sie verständlich sind. Idealerweise geht beides Hand in Hand: Ein guter Beweis erklärt, wie und warum ein Programmteil funktioniert und macht ihn dadurch verständlich.

5.2.3 Termination

Neben der Einhaltung von Zusicherungen müssen wir auch auf die *Termination* von Schleifen und rekursiven Methoden achten. Zusicherungen versprechen nur, dass in jedem Schleifendurchlauf bzw. bei jedem rekursiven Aufruf bestimmte Eigenschaften erfüllt sind. Sie können keine Obergrenze für die Anzahl der Schleifendurchläufe oder Methodenaufrufe geben. Solche Obergrenzen müssen wir durch andere Techniken sicherstellen.

Die Einhaltung aller Zusicherungen garantiert die *partielle Korrektheit* des Programms, bei der alle Ergebnisse den Spezifikationen entsprechen, falls das Programm Ergebnisse liefert. Termination ist dafür nicht erforderlich. *Vollständige Korrektheit* erweitert die partielle Korrektheit um Termination und garantiert damit, dass das Programm Ergebnisse liefert, die den Spezifikationen entsprechen.

Eine Voraussetzung für die Termination einer Schleife ist, dass jeder einzelne Schleifendurchlauf uns ein ausreichend großes Stück näher an das Ergebnis heranbringt. Um die Termination einer Schleife zu beweisen, müssen wir den Fortschritt pro Schleifendurchlauf abhängig von Eingabewerten in Zahlen fassen. Damit können wir eine von der Eingabe abhängige obere Schranke für die Anzahl der Schleifendurchläufe berechnen. Diese Berechnung braucht nicht genau zu sein, sondern eine ganz grobe Abschätzung reicht. Wir müssen ja nur irgendeine ober Schranke finden. Allerdings dürfen wir bei der Abschätzung nur in Richtung einer höheren Schranke und eines kleineren Fortschritts pro Schleifendurchlauf ungenau sein. Wir dürfen niemals annehmen, dass in einem Schleifendurchlauf mehr gemacht wird als tatsächlich passiert, und wir dürfen keinesfalls eine kleinere Schranke als die tatsächliche Schranke annehmen.

Betrachten wir die Methode `member` aus Listing 5.3 als Beispiel. Meist liefert die Abbruchbedingung einen guten Hinweis darauf, wo wir nach

einer geeigneten Schranke suchen sollen. Wir können den Abstand zwischen i und j (wobei i kleiner oder gleich j ist) als obere Schranke für die Anzahl der Schleifendurchläufe heranziehen, wenn sichergestellt ist, dass sich in jedem einzelnen Schleifendurchlauf i um mindestens eins erhöht oder j um mindestens eins verringert. In jedem Schleifendurchlauf wird entweder i auf $k + 1$ oder j auf $k - 1$ gesetzt, wobei k gleich $i + ((j - i) / 2)$ ist. Damit der Abstand sich verringert, muss k zwischen i und j liegen, einschließlich i und j selbst. Für größere Abstände zwischen i und j ist diese Bedingung jedenfalls erfüllt, da k etwa in der Mitte zwischen i und j liegt. Ist i gleich j oder nur um eins kleiner als j , dann ist k gleich i und die Bedingung somit erfüllt. Am Beginn der Schleife ist der Abstand zwischen i und j gleich der Größe des Arrays. Aufgrund dieser Überlegungen wissen wir, dass die Schleife nach spätestens `elems.length` Durchläufen terminiert.

Die Berechnung einer Schranke für die Anzahl der Schleifendurchläufe hat viel mit der Aufwandsabschätzung eines Algorithmus im schlechtesten Fall gemeinsam. Im Detail zeigen sich jedoch Unterschiede. Die Aufwandsabschätzung für die binäre Suche in Abschnitt 4.4.3 hat gezeigt, dass wir nur logarithmischen Aufwand haben. Wir erwarten eine deutlich niedrigere Schranke für die Anzahl der Schleifendurchläufe, wenn wir berücksichtigen, dass der Abstand zwischen i und j in jedem Schleifendurchlauf zumindest halbiert wird. Für größere Abstände zwischen i und j ist diese Bedingung immer erfüllt. Ist aber i gleich j und der Abstand daher gleich null, dann ist auch der halbe Abstand gleich null, und aufgrund dieser Abschätzung erzielen wir in einem Schleifendurchlauf keinen Fortschritt, da i und j möglicherweise unverändert bleiben. Alleine aufgrund der Halbierung des Abstands ist keine Termination garantiert. Wir müssen die zusätzliche Information benutzen, dass (wie oben gezeigt) jeder Schleifendurchlauf i um mindestens eins erhöht oder j um mindestens eins verringert. Erst dadurch ergibt sich eine Schranke logarithmisch zu `elems.length`. Wenn wir nur die Termination der Schleife zeigen wollen, hat das Wissen über die Halbierung der Abstände keine Vorteile.

Wir müssen noch die Frage klären, wie groß der Fortschritt pro Schleifendurchlauf mindestens sein muss, damit Termination garantiert ist. Wenn jeder Schleifendurchlauf einen etwa gleich großen Fortschritt größer null erzielt und die zu bearbeitende Datenmenge endlich ist, dann terminiert die Schleife sicher irgendwann. Wenn der Fortschritt dagegen von Durchlauf zu Durchlauf immer kleiner wird, ist Vorsicht geboten, wie wir bei der Halbierung der Abstände zwischen i und j gesehen haben. Als Ver-

gleich können wir die Entwicklung mathematischer Reihen heranziehen: Bei Halbierung der Abstände wird im ersten Schritt die Hälfte der Suche erledigt, also $1/2$, im zweiten Schritt $1/4$ und so weiter. Insgesamt erledigen wir in n Schleifendurchläufen einen Anteil von $\sum_{i=1}^n 1/2^i$ der Suche. Aus der Mathematik wissen wir, dass $\sum_{i=1}^{\infty} 1/2^i = 1$ gilt und daher leider unendlich viele Schritte nötig wären. Die binäre Suche terminiert nur, weil in jedem Schleifendurchlauf tatsächlich etwas mehr als die Hälfte erledigt wird, nämlich die Hälfte plus einem Element (das, welches im Vergleich betrachtet wird). Die entsprechende Reihe $\sum_{i=1}^{\infty} 1/2^i + 1/k$ (wobei k die Anzahl der Elemente ist) geht gegen ∞ da auch $\sum_{i=1}^{\infty} 1/k$ gegen ∞ geht. Die Suche terminiert, sobald eine Partialsumme (bestehend aus den ersten Summanden) eins übersteigt. Zum Sicherstellen der Termination kommt es nicht auf den Beitrag des größten Gliedes der Reihe an, sondern auf die Beiträge der kleinsten Glieder.

Praktisch gesehen müssen wir Überlegungen zur Termination ständig während des Programmierens anstellen und dabei auch den Fortschritt pro Schleifendurchlauf beachten. Wenn wir das nicht machen, kann es beispielsweise sehr leicht passieren, dass wir in der binären Suche in Listing 5.3 statt `j=k-1` und `i=k+1` einfach nur `j=k` und `i=k` schreiben. Auf den ersten Blick wirkt die einfachere Variante genauso gut, und viele Testfälle werden keine Änderung im Programmverhalten zeigen. Aber die Termination ist nicht mehr garantiert (wenn die gesuchte Zahl nicht gefunden wird), da die Abstände tatsächlich nur halbiert werden. Irgendwann wird das Programm in einer Endlosschleife hängen bleiben.

Wenn wir statt Schleifen Rekursion verwenden, lässt sich die Termination auf genau die gleiche Art zeigen wie bei Schleifen. Wir müssen nur die Beiträge jeden einzelnen Methodenaufrufs an der gesamten zu erledigenden Arbeit aufsummieren. Das Programm wird terminieren, sobald eine Partialsumme der Reihe den Wert eins übersteigt.

Aus Abschnitt 1.5.3 wissen wir, dass das Halteproblem im Allgemeinen unentscheidbar ist. Daher gibt es Programme und Algorithmen, bei denen wir auch nach genauer Analyse nicht wissen, ob sie terminieren. Das soll jedoch keine Ausrede sein. In der Praxis setzen wir fast nur Algorithmen ein, von denen wir wissen, dass sie terminieren. Diese Termination kann man auch zeigen. Wenn die Termination nicht beweisbar ist, wird man fast immer einen anderen Algorithmus einsetzen und den möglicherweise nicht terminierenden Algorithmus als falsch betrachten. Nur ganz wenige hochgradig spezialisierte Leute beschäftigen sich mit Algorithmen in engen Nischenbereichen, die in machen Fällen nicht terminieren. In diesen

Bereichen sind viel genauere Analysen notwendig um sicherzustellen, dass die Nichttermination zu keinen Problemen führt.

Das Halteproblem wirkt sich praktisch so aus, dass Programmiersprachen keine Termination garantieren. Man kann in jeder vollständigen Programmiersprache Endlosschleifen schreiben. Zum Beweis der Termination müssen wir für jede Schleife andere, an die Situation angepasste Argumente anführen. Wenn wir eine Sprache so beschränken, dass jede Schleife und Rekursion garantiert terminiert, dann haben wir bereits in der Sprache eine Auswahl der erlaubten Argumente getroffen, und viele Schleifen, die aufgrund anderer Argumente terminieren würden, wären nicht erlaubt.

Termination ist ein sehr grobes Kriterium. Meist reicht es nicht, wenn eine Berechnung nur terminiert, sie soll im Normalfall auch nach möglichst kurzer Zeit terminieren. Dazu sind Aufwandsabschätzungen oder Laufzeitmessungen nötig, bei denen wir aber oft nur den durchschnittlichen Aufwand betrachten. Überlegungen zur Termination gelten dagegen immer für den schlechtesten Fall, der praktisch fast nie eintritt und daher auch durch Testen kaum zu finden ist.

Manche Programme sollen nicht terminieren. Beispielsweise erwarten wir von der Software einer Telefonanlage, dass sie ständig läuft und niemals aufhört, ihre Aufgaben zu erfüllen. Trotzdem und gerade deswegen ist es in solchen Systemen besonders wichtig, dass die einzelnen Teilaufgaben terminieren und jeder Schleifendurchlauf den nötigen Fortschritt bringt. Es gelten also im Wesentlichen alle in diesem Abschnitt angestellten Überlegungen. Allerdings ist die Datenmenge, auf die das Programm angewendet wird, unbegrenzt. Jeder auf einmal verarbeitete Ausschnitt aus der Datenmenge muss in endlicher (und in der Regel kurzer) Zeit ein Teilergebnis liefern, damit das Programm sinnvoll ist.

5.2.4 Beweise und deren Grenzen

Formale Beweise der Programmkorrektheit und ein statisches Programmverständnis gehen Hand in Hand. Wir verstehen ein Programm, indem wir uns überlegen, wie wir dessen Korrektheit beweisen können. Umgekehrt setzt ein Korrektheitsbeweis auch ein gutes Programmverständnis voraus.

Heute haben wir ein umfangreiches Sortiment an Werkzeugen, die uns dabei helfen, Beweise zu finden. Zu den einfachsten Werkzeugen zählen die Typen. Sie schränken die Konstruktion von Programmen derart ein, dass jedes Programm, das vom Compiler akzeptiert wird, bestimmte Eigenschaften erfüllt. Beispielsweise können nur solche Nachrichten gesen-

det werden, für welche die Empfänger passende Methoden implementiert haben. Wir verlassen uns darauf, dass aufgerufene Methoden tatsächlich existieren. Insofern beeinflussen Typen auch unseren Programmierstil und die Art und Weise, wie wir beim Programmieren denken. Das ist ein wichtiger Grund dafür, warum Typen nur mit eher einfachen und fast immer sinnvollen Eigenschaften umgehen können. Man kann nicht wegen einer Eigenschaft, die nicht in fast jedem Programm gebraucht wird, eine bestimmte Denkweise und Art der Programmkonstruktion vorgeben. Die Freiheit bei der Programmierung würde dadurch zu stark eingeschränkt.

In letzter Zeit wurden große Fortschritte bei der Steigerung der Effizienz von Verfahren zur automatischen Verifikation von Programmen erzielt. Vor allem *Model Checking* wird seit kurzem auch praktisch eingesetzt: Man übergibt einem Werkzeug nur eine Systembeschreibung (beispielsweise in Form eines Programms) und eine logische Eigenschaft, und das Werkzeug überprüft selbständig, ob die Eigenschaft mit der Systembeschreibung in Einklang steht. Das ist der Fall wenn – in der Terminologie der mathematischen Logik – die Eigenschaft ein *Modell* der Systembeschreibung ist. Im Idealfall findet das Werkzeug keinen Gegenbeweis und bestätigt damit die Eigenschaft. Sehr hilfreich ist die Tatsache, dass das Werkzeug ein Gegenbeispiel liefert, wenn die Eigenschaft nicht erfüllt ist. Es zeigt recht genau auf, an welchen Stellen man die Systembeschreibung bzw. das Programm ändern muss, damit die gewünschte Eigenschaft erfüllt wird. Trotz großer Fortschritte in den letzten Jahren lassen sich viele Eigenschaften leider noch immer nur für ganz kleine Programme beweisen. Oft hat das Werkzeug nicht genug Speicher zur Verfügung oder rechnet tagelang ohne erkennbaren Fortschritt. Für bestimmte Eigenschaften und Programme geeigneter Größe ist Model Checking dagegen sehr erfolgreich.

Ein bekannter Model Checker für Java-Programme ist *Java Pathfinder (JPF)*.² Er wird als „Schweizer Messer für die Java Verifikation“ beschrieben. Unter anderem kann man damit Zusicherungen in Form von über Annotationen (werden in Teil 2 angesprochen) formal definierten Vorbedingungen, Nachbedingungen und Invarianten überprüfen. Dabei wird bereits vor Programmausführung überprüft, ob alle Zusicherungen zur Laufzeit halten werden. Es ist klar, dass solche Überprüfungen sehr aufwendig sind und viel Zeit in Anspruch nehmen. Eine erfolgreiche Überprüfung der Zusicherungen kann die Zuverlässigkeit eines Programms erheblich steigern. Sogar wenn Fehler gefunden werden, steigt die Qualität des Programms

²Siehe <http://babelfish.arc.nasa.gov/trac/jpf>.

durch Beseitigung der Fehler. Allerdings kann JPF für einen Beweis sehr lange brauchen und wird in vielen Fällen in vertretbarer Zeit zu keinem Ergebnis kommen.

Generell kann man nur konkrete Eigenschaften beweisen, die sich in eine relativ einfache formale Form bringen lassen. Eine solche Eigenschaft wäre beispielsweise, dass das Ergebnis einer bestimmten Methode niemals den Wert `null` haben darf. Oft ist unsere Vorstellungen davon, welche Eigenschaften wir erwarten, aber nur sehr vage. Zum Beispiel wollen wir vermeiden, dass jemand unter Zuhilfenahme unseres Programms irgendeinen Schaden anrichtet. Es ist nicht klar, auf welche Weise konkreter Schaden angerichtet werden könnte. Natürlich soll kein Fremder Zugang zu nicht für die Öffentlichkeit bestimmten Daten bekommt oder diese sogar ändern können, aber es bleibt offen, wer als Fremder gilt und welche Daten genau öffentlich sichtbar sein können. Ein großes Problem sind Fehler in einem Programm, über die jemand mit guter Kenntnis des Programms etwas machen kann, wofür das Programm nicht vorgesehen ist und das man vermeiden möchte. Es ist unmöglich, alle solchen potentiellen Fehler aufzuzählen. Vieles von dem, was sich später als Fehler herausstellt, war ursprünglich als sinnvolles Feature gedacht. Wenn uns bewusst wäre, was alles als Fehler anzusehen ist, hätten wir die meisten davon (unabhängig von Beweisen) gar nicht gemacht. Im praktischen Einsatz zeigen sich deren Auswirkungen oft erst nach Jahren, falls überhaupt. Auch die ausgefeilteste Beweistechnik kann nichts ausrichten, wenn nicht klar ist, welche Eigenschaften erwünscht und welche unerwünscht sind.

Erwünschte und unerwünschte Eigenschaften können nahe beieinander liegen. Das zeigt sich deutlich bei *Denial-of-Service (DoS) Attacks*. Ziel der meisten Systeme ist es, alle Anfragen rasch zu bearbeiten. Aber es gibt auch anonyme Anfragen aus dem Web, die Böses im Schilde führen: Jemand kann in kurzer Zeit absichtlich so viele Anfragen stellen, dass das System überlastet ist und die Dienste nicht mehr ordnungsgemäß erfüllen kann. Auf diese Weise wird (mit entsprechend hohem Aufwand) fast jedes System vorübergehend unbrauchbar. Es gibt aber Gegenstrategien. Man unterscheidet ernst gemeinte Anfragen von möglicherweise bösen und unterbindet letztere. Vielleicht muss man sich einloggen oder vor Inanspruchnahme eine für Menschen leichter als für Maschinen lösbare Aufgabe lösen. Es wird damit schwieriger, das System zu benutzen. Natürlich wünschen wir uns einen direkten, offenen Zugang und ärgern uns über Einschränkungen. Aber aus dieser erwünschten Eigenschaft wird rasch eine unerwünschte und umgekehrt, wenn DoS Attacks auftreten.

Automatisierte Verfahren zur Programmverifikation erhöhen zwar die Zuverlässigkeit, aber sie tragen kaum zum Programmverstehen bei und versagen bei schwammigen Zielvorstellungen. Ein *Code Review* kann auch damit umgehen. Dabei begutachtet ein erfahrener Reviewer den Quellcode eines Programms bzw. Programnteils, stellt Fragen und macht Verbesserungsvorschläge dazu. Über Code Reviews werden Fehler aus ganz unterschiedlichen Bereichen gefunden, beispielsweise Verletzungen von Konventionen und Standards, schlechte oder falsche Kommentare, unklare, widersprüchliche oder verletzte Spezifikationen, Nichterfülltsein von Anforderungen und unzureichende Wartbarkeit. Gerade die Abdeckung eines so weiten Bereichs an potentiellen Fehlern, von denen viele einem formalen Beweis nicht zugänglich sind, trägt wesentlich zur Qualitätsverbesserung bei. Code Reviews stellen, im Gegensatz zum Testen, ein statisches Verfahren zur Qualitätsverbesserung dar. So wie beim Testen, aber anders als bei formalen Beweisen, wird keine Fehlerfreiheit in einem Bereich garantiert.

5.3 Testen

Testen ist zur Qualitätskontrolle unverzichtbar. So mancher Fehler tritt nur bei ausgiebigem Testen zu Tage. Wir wollen betrachten, wie sich das Testen auf die Qualität eines Programms auswirkt und welche Vorgehensweisen zu unterscheiden sind. Schließlich gehen wir auf Laufzeitmessungen ein, die als spezielle Form des Testens betrachtet werden können.

5.3.1 Auswirkungen auf Softwarequalität

Zweifellos erhöht Testen die Qualität eines Programms. Eine naive Annahme geht davon aus, dass ein gefundener Fehler gleich beseitigt wird und das Programm danach eine höhere Qualität hat. Ganz so einfach sind die Auswirkungen des Testens und der Fehlerkorrektur auf die Programmqualität jedoch nicht. Folgende Faktoren spielen eine Rolle:

- Testfälle können niemals alle Möglichkeiten abdecken. Auch sehr intensives Testen kann keine Fehlerfreiheit garantieren.
- Die Anzahl der gefundenen Fehler lässt Rückschlüsse auf die Anzahl der im Programm vorhandenen Fehler zu. Werden bei gleicher Testmethode in einem Programm *A* mehr Fehler gefunden als in einem Programm *B*, dann enthält *A* wahrscheinlich auch mehr nicht gefundene Fehler als *B*, und *B* ist somit von höherer Qualität.

- Das Beseitigen eines gefundenen Fehlers führt leicht zu weiteren Fehlern. Sobald man einen Fehler gefunden hat, ist die Verlockung groß, ihn sofort an der Stelle, an der er aufgetreten ist, zu beseitigen. Sehr oft liegt die tatsächliche Ursache des Fehlers jedoch irgendwo anders. Durch die Ausbesserung hat man daher nur ein Symptom beseitigt, aber nicht die Fehlerursache. Die Ausbesserung ist nicht nur nicht zielführend, sondern falsch: Wenn die eigentliche Fehlerursache später entdeckt und beseitigt wird, stellt sich die Ausbesserung plötzlich als neuer Fehler dar. Deshalb soll man vor Fehlerkorrekturen sehr sorgfältig die Ursachen erforschen und niemals einen Fehler ausbessern, dessen Ursache man nicht ganz genau kennt.
- Fehler können in jeder Phase der Softwareentwicklung passieren, auch in der Analyse- und Entwurfsphase. Möglicherweise sind auch Testfälle falsch. Bei der Suche nach der Fehlerursache darf man sich nicht nur auf die Implementierung konzentrieren.
- Manche Fehler in einem Programm werden nur mit äußerst kleiner Wahrscheinlichkeit sichtbar, wenn zufällig mehrere ganz selten erfüllte Voraussetzungen dafür zusammenkommen. Diese Fehler sind beim Testen kaum erkennbar. Mit etwas Glück treten sie auch im praktischen Einsatz des Programms nicht auf. Es ist daher kein Problem, wenn sie von niemandem erkannt werden.
- Fehler, die nur ganz selten zu Tage treten, können dazu benutzt werden, in ein System einzubrechen oder das System lahmzulegen. Angreifer stellen die Voraussetzungen für das Sichtbarwerden eines Fehlers, die normalerweise praktisch nie erfüllt sind, künstlich her und lösen damit den Fehler absichtlich aus. Erkannte Fehler mit schwerwiegenden Auswirkungen sind daher rasch zu beseitigen. Das gilt auch für Fehler, die bei der üblichen Verwendung des Programms nicht sichtbar werden können.
- Zur Absicherung gegen Missbrauch muss man auch das Programmverhalten bei unerwünschten und unüblichen Verwendungen testen. Es ist viel Fantasie und Fingerspitzengefühl nötig, um mögliche unerwünschte Verwendungsweisen zu erkennen.
- Es kommt vor, dass das Eindringen in ein System absichtlich ermöglicht wird, beispielsweise um das Testen und die Suche nach Fehlerursachen zu vereinfachen. Derartige ist gefährlich und zu vermeiden.

Zusammengefasst kann man aus diesen Punkten den Schluss ziehen, dass Testen in erster Linie zur *Qualitätskontrolle* dient. Durch intensives Testen stellen wir sicher, dass unser Programm die erwartete geringe Fehlerwahrscheinlichkeit aufweist – nicht zu verwechseln mit Fehlerfreiheit. In allen Phasen der Softwareentwicklung müssen wir uns darum bemühen, diese Qualität zu erreichen. Testen dient durch Aufdecken und anschließende Beseitigung von Fehlern, wenn überhaupt, nur sehr beschränkt zur direkten Qualitätsverbesserung. Eine *indirekte Qualitätsverbesserung* erhalten wir jedoch dadurch, dass uns beim Testen aufgedeckte Fehler gute Hinweise darauf geben, in welchen Bereichen wir die Qualität über andere Mittel verbessern müssen. Bereits in Abschnitt 1.6.2 haben wir gesehen, wie ein üblicher Ablauf der Programmkonstruktion aussieht: Auf das Testen folgt das Debuggen, bei dem Fehlerursachen ergründet werden. Erst nach dem Finden der tatsächlichen Fehlerursachen können wir uns einen Plan zurechtlegen, wie wir diese beseitigen. Häufig sind dafür größere Umstrukturierungen nötig. Die Qualitätsverbesserung tritt dadurch ein, dass wir die beim Testen und Debuggen gewonnene Erfahrung in die überarbeitete Planung und Umstrukturierung einbeziehen. Nur die Ursachen ganz trivialer Fehler sind sofort erkenn- und beseitigbar.

Zu einer merklichen Qualitätsverbesserung führen in der Regel nur Techniken, die uns dabei helfen, das Programm auf statische Weise zu verstehen. Dazu zählen Code Reviews (siehe Abschnitt 5.2.4), aber das Testen eher nicht. Ergebnisse von Testläufen helfen eventuell dabei, unsere Bemühungen zum statischen Verstehen auf die richtigen Programmteile zu lenken. Vor allem helfen aufgedeckte Fehler aber dabei, unsere Aufmerksamkeit bei Code Reviews auf bisher zuwenig beachtete Aspekte zu lenken. Nachdem wir beispielsweise eine Endlosschleife entdeckt haben, werden wir auch an ganz anderen Programmstellen verstärkt auf Schleifeninvarianten zum Beweis der Termination achten. Testergebnisse öffnen uns die Augen dafür, worauf wir achten müssen. Wird das Testen auf diese Weise zum Gewinnen von Erfahrung eingesetzt, kann das Erkennen eines Fehlers zahlreiche weitere Fehler vermeiden.

In früherer Zeit hat man Programmieren gelernt, indem man ein Programm vollständig auf Papier entwickelt und sich über Beweise vergewissert hat, dass es funktioniert. Erst wenn man sich sicher war, durfte man das Programm auch am Computer ausprobieren. Mit dieser Vorgehensweise wurde das statische Programmverständnis gefördert und die Verschwendung damals noch teurer Rechenzeit verhindert. Diese Vorgehensweise ist mit der beinahe unbeschränkten Verfügbarkeit von Computern

verschwunden. Allerdings hat sich am ursprünglichen Ziel nichts geändert: Auch heute sollte man darauf achten, nur fertige und gut durchdachte Programmteile zu übersetzen und auszuprobieren. Das Durchdenken des Programmcodes wie beim Code Review bringt uns qualitativ hochwertige Programme, nicht das Testen. Wiederholtes Testen und Ausbessern unausgereifter Programmteile kostet uns in der Summe viel Zeit. Diese Zeit ist besser in ein statisches Verstehen der Programmteile investiert.

Auf gewisse Weise können uns Testfälle doch helfen, ein Programm statisch zu verstehen: Testfälle können ähnlich wie Anwendungsfälle zur Spezifikation benutzt werden. Jeder Testfall setzt bestimmte Eingabewerte mit Ausgabewerten in Beziehung. Auch wenn Testfälle niemals alle Fälle, die in einem nichttrivialen Programm auftreten, abdecken können, so geben sie das gewünschte Programmverhalten für die abgedeckten Fälle doch ganz klar und unmissverständlich vor. Durch geschickte Auswahl der Testfälle und Abdeckung der Sonder- und Grenzfälle lässt sich trotzdem eine recht vollständige Beschreibung des gewünschten Verhaltens erzielen.

5.3.2 Testmethoden

Unter dem Testen eines Programms oder Programmteils versteht man vor allem das Ausprobieren des Programms bzw. Programmteils. Das klingt zwar einfach, ist es in der Praxis aber keineswegs. Im Laufe der Zeit haben sich unzählige Testmethoden entwickelt, die alle ihre Existenzberechtigung in bestimmten Bereichen haben. Um einen groben Überblick über die wichtigsten Testmethoden zu bekommen, klassifizieren wir sie nach verschiedenen Kriterien.

Viele Projekte setzen nacheinander folgende Teststufen ein:

Unittest: Dabei testet man die Funktionalität eines klar abgegrenzten Programmteils (das ist eine Unit, beispielsweise eine einzelne Klasse). Es wird überprüft, ob die Methoden so wie geplant laufen und die erwarteten Ergebnisse liefern. Man testet Units getrennt voneinander, weil jede einzelne übersichtlicher ist als das ganze Programm und Ursachen für aufgetretene Fehler daher leichter zu finden und beseitigen sind. Oft wird der Unittest durch eigens dafür geschriebenen Programmcode auf künstlichen Testdaten ausgeführt.

Integrationstest: Der Test überprüft die korrekte Zusammenarbeit zwischen den Units, die zuvor schon über Unittests einzeln getestet wurden. Schwerpunkte liegen auf den Schnittstellen zwischen den

Units sowie komplexeren Abläufen, welche die Funktionalität mehrerer Units beanspruchen. Wie beim Unittest verwenden wir oft eigenen Testcode und künstliche Testdaten.

Systemtest: Das gesamte System wird hinsichtlich der Erfüllung aller geforderten funktionalen und nichtfunktionalen Eigenschaften überprüft. Meist wird der Test auf einer (künstlichen) Testumgebung und auf künstlichen Testdaten ausgeführt.

Abnahmetest: Der Abnahmetest überprüft, ob das gesamte System auch unter realen Bedingungen mit realen Daten seine Aufgaben erfüllt. Nach Bestehen dieses Tests geht das System tatsächlich in Betrieb.

Diese Teststufen sind auch bei der Lösung von Übungsaufgaben hilfreich. Den Integrations- und Systemtest wird man aufgrund der Kleinheit der Aufgaben jedoch zu einer Einheit verschmelzen, und der Abnahmetest entspricht der Überprüfung des Programms im Rahmen der Beurteilung.

Abgesehen vom Abnahmetest werden diese Teststufen wiederholt ausgeführt. Nach jeder Änderung und neuerlichen Compilation ist ein Testlauf fällig. Meist werden für jeden Testlauf dieselben Testfälle und Testdaten verwendet, nur manchmal kommen neue Testfälle und Testdaten hinzu. Diese Art von wiederholtem Test nennt man *Regressionstest*. Regressionstests überprüfen, ob Testfälle, die in einer früheren Programmversion keine Fehler aufdecken konnten, auch in späteren Versionen fehlerfrei durchlaufen. Damit sollen Fehler, die sich durch eine Programmänderung eingeschlichen haben, ohne Verzögerung aufgedeckt werden. Wegen der häufigen Wiederholung der Testläufe erfolgt das Testen fast immer automatisiert, oft durch selbst geschriebene Testprogramme.

Der Wissensstand über die zu testenden Programmteile beeinflusst die Testfälle. Diesbezüglich können wir drei Testarten unterscheiden:

Black Box Test: Beim Black Box Test verwendet man keinerlei Wissen über die interne Realisierung von Programmdetails. Testfälle werden ausschließlich entsprechend dem gewünschten Verhalten des Programms anhand der Anforderungsdokumentation entworfen, häufig von auf das Testen spezialisierten Personen und nicht von den EntwicklerInnen des Systems. Black Box Tests werden vor allem für Systemtests und Abnahmetests eingesetzt.

White Box Test: Beim White Box Test basieren Testfälle auf der internen Struktur des Programms. Wir entwickeln die Testfälle zusammen

mit dem Programm. Dabei sorgen wir dafür, dass beispielsweise jeder Programmzweig oder die Auswirkung jeder Anweisung durch einen Testfall abgedeckt ist, um zumindest alle groben Fehler entdecken zu können. Im weitesten Sinn entsprechen auch `assert`-Anweisungen mit eingeschalteter Überprüfung White Box Tests. White Box Tests werden vor allem für Unittests und Integrationstests eingesetzt.

Grey Box Test: Dabei entwickeln wir Testfälle zur genauen Spezifikation des Programms noch vor dessen Implementierung. Wie beim Black Box Text haben wir zu diesem Zeitpunkt noch keine Information über die Programmstruktur. Da jedoch die Implementierung der Spezifikation folgt, ergibt sich am Ende eine fast so gute Abdeckung aller Programmzweige wie beim White Box Test.

Eine weitere Unterscheidungsmöglichkeit ergibt sich durch die inhaltlichen Aspekte des Testens. Hier betrachten wir nur eine kleine Auswahl:

Funktionaler Test: Er überprüft die funktionalen Eigenschaften vor allem im Hinblick auf Korrektheit und Vollständigkeit.

Nichtfunktionaler Test: Damit überprüft man nichtfunktionale Eigenschaften. Dazu zählen viele überprüfbare Aspekte aus den Bereichen Wartbarkeit, Gebrauchstauglichkeit und Zuverlässigkeit.

Schnittstellentest: Er überprüft, ob alle Komponenten eines Programms zusammenpassen und gemeinsam funktionieren.

Oberflächentest: Dieser Test bezieht sich auf die Verwendbarkeit und Funktionalität der Benutzerschnittstelle.

Stresstest: Ein Stresstest überprüft das Verhalten eines Systems unter Ausnahmebedingungen. Es gibt zahlreiche Varianten wie den *Crash-test*, bei dem man versucht, das System zum Absturz zu bringen, und einen *Lasttest*, bei dem man das Verhalten eines (etwa durch viele gleichzeitige Benutzer) über die Grenze belasteten Systems testet.

Sicherheitstest: Er prüft das System auf Sicherheitslücken.

Diese Aufzählung kann man beinahe endlos fortsetzen. Durch Tests kann man alle Aspekte überprüfen, die wichtig genug erscheinen.

Tests kosten Zeit und müssen organisiert werden. Obwohl man weiß, wie wichtig Tests sind, wird in vielen (vor allem kleinen) Projekten nur

halbherzig getestet. Das führt leider zu schlechter Qualität, auch deswegen, weil man während der Programmkonstruktion das Gefühl hat, dass Qualität gar nicht gefragt ist. Deshalb sollte man sich eine gute Teststrategie überlegen und von Anfang an durchziehen. Getestet werden sollte alles, was für das Projekt wichtig ist. Wenn man bereits zu Beginn die zu überprüfenden Aspekte kennt, wird man das Programm so konstruieren, dass diese Aspekte eine hohe Qualität bekommen. Die Teststrategie kann im weiteren Sinn auch als eine Art von Spezifikation betrachtet werden.

Es schadet nicht, neben den durchorganisierten Regressionstests das unfertige Programm einfach einmal auf eine unübliche Art aufzurufen und zu schauen, was passiert. Solche zufälligen und scheinbar gar nicht organisierten Tests decken oft diffizile Fehler auf, mit denen niemand gerechnet hat und die deswegen in den organisierten Tests nicht vorkommen. Daher ist es sinnvoll, auch zufällige Tests in die Teststrategie zu integrieren.

5.3.3 Laufzeitmessungen

Auf den ersten Blick scheinen Laufzeitmessungen an Programmen sehr einfach zu sein: Man braucht den zu messenden Ablauf nur zu starten und mit der Stoppuhr zu messen, wieviel Zeit verstreicht, bis der Ablauf endet. Leider bekommen wir auf diese Weise keine zuverlässigen Ergebnisse, die Rückschlüsse auf die Dauer des Ablaufs unter leicht geänderten Bedingungen zulassen. Die gemessene Zeit hängt von zahlreichen Faktoren ab, beispielsweise Art und Menge der verarbeiteten Daten, anderer am Rechner gleichzeitig laufender Software und einer Unzahl an winzigsten, undurchschaubaren Details in der Hard- und Software. So kann es ausreichen, sich vor der Messung unter einem andern Namen (mit denselben Rechten und im Hintergrund laufenden Programmen) am Rechner anzumelden, um ganz andere Ergebnisse zu bekommen. Genaue Gründe für derartige Effekte sind kaum zu finden. Mangels besserer Erklärung redet man sich gerne auf *Cache Effekte* aus, also Unterschiede in der Laufzeit, die dadurch verursacht werden, dass der Prozessor (meist wegen geänderter Speicheradressen) andere Datenmengen im Cache hält. Für bestimmte Programmstücke sind Laufzeitunterschiede durch solche Effekte bis zu einem Faktor zwei oder sogar mehr nicht außergewöhnlich.

Unter Linux lässt sich die Zeit mittels `time` messen:

```
time javac Hello.java
```

In diesem Beispiel messen wir die Zeit für das Übersetzen eines einfachen Java-Programms. Das Ergebnis besteht aus mehreren Zeitangaben:

```
real    0m0.863s
user    0m1.036s
sys     0m0.064s
```

Der **real**-Wert gibt an, wieviel Zeit vom Beginn bis zum Ende der Ausführung verstrichen ist, der **user**-Wert wieviel Zeit eines Prozessor-Kerns die eigentliche Programmausführung beansprucht hat, und der **sys**-Wert wieviel Zeit eines Prozessor-Kerns das Betriebssystem an Serviceleistungen für die Programmausführung erbracht hat. Es fällt auf, dass die **user**-Zeit mehr als eine Sekunde ausmacht, die Ausführung aber schon nach weniger als einer Sekunde fertig war. Der Grund dafür liegt in den vier Prozessor-Kernen, die unser Rechner hat. Offensichtlich wurden Programmteile parallel ausgeführt. Die Zeiten für **user** und **sys** könnten zusammen bis zu viermal so lang sein wie für **real**. Tatsächlich waren die Prozessor-Kerne nur zu einem kleinen Teil ausgelastet. Die restliche Zeit haben die Kerne mit der Ausführung anderer Programme oder mit Warten verbracht.

Wiederholte Zeitmessungen (auf demselben Rechner mit genau denselben Programmaufrufen) ergeben unterschiedliche Zeiten, beispielsweise eine **real**-Zeit zwischen etwa 0,7 und 1,3 Sekunden, eine **user**-Zeit zwischen 0,7 und 1,1 Sekunden und eine **sys**-Zeit zwischen 0,04 und 0,08 Sekunden. Diese Werte zeigen, wie groß die Laufzeitunterschiede schon unter unveränderten Bedingungen sind. Wenn wir die Bedingungen ändern, werden die Unterschiede noch deutlich größer.

Um den Laufzeitunterschieden auf die Spur zu kommen, wollen wir einige Ursachen für Verzögerungen betrachten.

Latenz: Heute spielt die Kommunikation zwischen Rechnern sowie zwischen einzelnen Komponenten (CPU, Speicher, Festplatten, etc.) innerhalb eines Rechners eine für die Effizienz entscheidende Rolle. Wenn benötigte Daten nicht vorhanden sind, sondern erst geholt werden müssen, entstehen Wartezeiten. Unter der Latenz (Verzögerung) versteht man die Zeit vom Senden bis zum Empfangen einer einfachen Nachricht (*Einweglatenz*) oder vom Senden einer Anforderung von Daten bis zum Erhalt des (ersten) Datenpakets (*Round-Trip Time*, *RTT*). Je größer die Latenz ist, desto länger muss man warten.

Bandbreite: Die Bandbreite gibt an, wie viele Daten in einer bestimmten Zeiteinheit übertragen werden können. Durch Übertragung größerer Blöcke auf einmal steigt die Bandbreite, da man durch nur eine Anforderung viele Daten bekommt. Allerdings ist das nur sinnvoll, wenn man alle diese Daten benötigt. In Anwendungen, in denen viele nebeneinander liegende Daten aus einer Quelle übertragen werden (beispielsweise Video Streaming), kommt es auf die Bandbreite an. In anderen Anwendungen, in denen viele Daten von unterschiedlichen Adressen geholt werden, ist die Latenz von größerer Bedeutung.

Feingranularer Parallelismus: Moderne Prozessoren steigern die Effizienz, indem sie mehrere aufeinanderfolgende Befehle gleichzeitig abarbeiten. Sie können auch kurze Wartezeiten überbrücken, indem sie Befehle, für die bereits alle Daten vorhanden sind, vorziehen. Beides geht aber nur, wenn die Befehle nicht voneinander abhängen. Typischerweise ist diese Art der parallelen Abarbeitung durch Abhängigkeiten auf wenige Befehle begrenzt und kann nur wenige kurze Verzögerungen ausgleichen.

Thread Parallelismus: Meist werden gleichzeitig mehrere Threads abgearbeitet. Muss ein Thread für längere Zeit auf Daten warten, wird während dieser Zeit ein anderer Thread ausgeführt. Das Umschalten zwischen Threads ist mit Aufwand verbunden, sodass sich der Gesamtaufwand durch häufiges Umschalten merklich erhöht.

Zeiten, in denen ein Prozessor-Kern nur kurz (z.B. auf Daten aus dem Speicher) wartet, sind in den **user** und **sys**-Werten enthalten, nicht jedoch Wartezeiten, die durch Thread Parallelismus ausgeglichen werden könnten. Unterschiedliche Wartezeiten wirken sich also nicht nur auf den **real**-Wert, sondern auch auf die **user** und **sys**-Werte aus.

Um unterschiedliche Messergebnisse auszugleichen, ist es üblich, Mittelwerte aus mehreren Messungen anzugeben. Diese Vorgehensweise suggeriert jedoch Objektivität, die tatsächlich meist nicht gegeben ist: Mehrfache Messungen unter denselben Bedingungen schalten ja nur ganz wenige zufällige Einflussgrößen aus, während andere auch in den Mittelwerten enthalten sind. Eigentlich müsste man viele Messungen unter unterschiedlichen Bedingungen (wie Datenmengen, Rechnerbelastungen, Hardware-Architekturen, Betriebssysteme, Compiler, Interpreter) durchführen und die Ergebnisse systematisch gegenüberstellen. Ein so hoher Aufwand ist in der Praxis jedoch fast nie gerechtfertigt.

Ein häufiger Fehler besteht darin, die Laufzeit für eine bestimmte Datenmenge zu messen und linear auf eine andere Datenmenge hochzurechnen. Das funktioniert nicht, weil die meisten Algorithmen einen nicht-linearen Aufwand haben. Oft misst man nicht nur die Laufzeit eines Algorithmus, sondern mehrerer Algorithmen mit unterschiedlichem Aufwand, die zusammen das Programm ergeben. Daher ist es auch nicht einfach, die Laufzeiten aus mehreren Messungen hochzurechnen. Um einigermaßen zuverlässige Aussagen über die Laufzeit bei größeren Datenmengen zu bekommen, muss man die Laufzeit auch tatsächlich mit entsprechend großen Datenmengen messen. Außerdem muss die Datenmenge realistisch sein. Reale Daten können zu ganz anderen Ergebnissen führen als generierte Zufallszahlen oder eine mehrfach kopierte kleinere Datenmenge. Zuverlässige Laufzeitmessungen erfordern viel Spezialwissen.

Meist gibt man sich mit groben Näherungswerten für die Laufzeit zufrieden. Man will ja beispielsweise nur feststellen, ob Antwortzeiten üblicherweise im erwarteten Bereich liegen. Für manche Systeme, sogenannte *Echtzeitsysteme* ist das zeitliche Verhalten jedoch kritisch und von überragender Bedeutung. Man unterscheidet *weiche* von *harten* Echtzeitsystemen. Erstere findet man z.B. in Spielen, wo Berechnungen nur eine gewisse Zeit dauern dürfen, weil sonst Darstellungsfehler auftreten oder die gewünschte Frame Rate nicht erreicht wird. Gelegentliche Zeitüberschreitungen sind zwar unerwünscht, aber tolerierbar.

In harten Echtzeitsystemen haben Zeitüberschreitungen dagegen fatale Folgen. Man kann sich leicht ausmahlen, was passiert, wenn die Lenkung oder Bremse in einem Fahrzeug verzögert reagiert. Sicherheitskritische Echtzeitsysteme setzen zahlreiche Verfahren und Techniken ein, um Ausfälle und Verzögerungen zu vermeiden. Mit einfachen Laufzeitmessungen ist es nicht mehr getan. Vielmehr kommen Kombinationen aus Laufzeitmessungen und Beweisverfahren zum Einsatz, die Rechtzeitigkeit garantieren sollen. Es werden auch gleichzeitig mehrere *redundante* Lösungen derselben Aufgabe berechnet, um im Falle des Versagens einer Berechnung doch noch rechtzeitig ein Ergebnis zur Verfügung zu haben. Harte Echtzeitsysteme werden so gut wie nie in Java geschrieben, da man näher an der Hardware bleiben und möglichst wenig von Einflüssen durch das Laufzeitsystem der Programmiersprache abhängig sein möchte. Hauptsächlich kommt die Programmiersprache C zum Einsatz. Jeder dieser Einflussfaktoren (Hardwarenähe, Redundanz, Beweisverfahren) bewirkt, dass die Entwicklung harter Echtzeitsysteme viel aufwendiger ist als die nicht zeitabhängiger Programme oder weicher Echtzeitsysteme.

Listing 5.4: Fehlerhaftes Programm

```

1 public class Rec {
2     public static void main (String[] args) {
3         rec(2);
4     }
5     private static int rec (int x) {
6         assert x > 0 : "x = " + x;
7         // System.out.println("x = " + x);
8         return rec (x - 1);
9     }
10 }
```

5.4 Nachvollziehen des Programmablaufs

Wenn Fehler aufgetreten sind, müssen wir deren Ursachen finden. Statisches Programmverständnis reicht dafür meist nicht aus, sodass wir in das laufende Programm *hineinschauen* müssen. Diesen Vorgang nennt man *Debuggen*. Wegen der gigantisch großen Anzahl an Programmpfaden ist es viel schwieriger, ein Programm durch Nachvollziehen aller möglichen dynamischen Abläufe zu verstehen, als sich ein statisches Programmverständnis anzueignen. Das Nachvollziehen eines Programmablaufs ist daher nur sinnvoll, um sich einzelne Programmpfade, auf denen Fehler passieren, näher zu betrachten.

5.4.1 Stack Traces und Debug Output

Listing 5.4 zeigt ein einfaches Programm mit einem offensichtlichen Fehler: Der rekursiven Methode `rec` fehlt eine Abbruchbedingung, sodass die Rekursion niemals terminiert. Dieser Fehler kann leicht statisch verstanden werden. Trotzdem verwenden wir dieses Beispiel als Basis für das Nachvollziehen des Programmablaufs.

Nach Start des Programms durch `java Rec` bekommen wir die Fehlermeldung „Exception in thread "main" java.lang.StackOverflowError“ gefolgt von einer langen Reihe von Zeilen „at Rec.rec(Rec.java:8)“. Ähnliche Fehlermeldungen bekommen wir immer, wenn das Laufzeitsystem einen Fehler entdeckt und als *Ausnahme* (engl. *exception*) zurückmeldet. Wie wir in Abschnitt 5.5 sehen werden, können die meisten Ausnahmen im Programm abgefangen und behandelt werden. In unserem Beispielprogramm

```
Exception in thread "main" java.lang.AssertionError: x = 0
    at Rec.rec(Rec.java:6)
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    at Rec.main(Rec.java:3)
```

Abbildung 5.5: Fehlermeldung nach Aufruf von `java -ea Rec`

machen wir das nicht, und ein Abfangen dieser Art von Ausnahmen ist generell sinnlos. Die unbehandelte Ausnahme führt zum Programmabbruch und zur Fehlermeldung. Der Inhalt der Fehlermeldung gibt uns wichtige Hinweise zur Fehlerursache: Wie der Name `StackOverflowError` des Typs der Ausnahme schon vermuten lässt, wurde die Ausnahme ausgelöst, weil der implizit vom Laufzeitsystem verwendeten Stack nicht groß genug war, um alle geschachtelten Methodenaufrufe zu enthalten. Auf diesem Stack wird bei jedem Methodenaufruf ein Eintrag gemacht und am Ende der Ausführung der Methode wieder entfernt. Der Stack enthält also stets Informationen zu allen geschachtelten Methodenaufrufen zu einem bestimmten Zeitpunkt; der oberste Stackeintrag entspricht der gerade ausgeführten Methode, alle anderen Einträge entsprechen Methodenausführungen, die auf ein Ergebnis eines geschachtelten Aufrufs warten. Die Fehlermeldung enthält den sogenannten *Stack Trace*, welcher wesentliche Teile der zum Zeitpunkt der Ausnahme im Stack enthaltenen Information umfasst. Die zweite Zeile zeigt die Programmstelle, an der die Ausnahme aufgetreten ist: „at Rec.rec(Rec.java:8)“ steht für Methode `rec` in Klasse `Rec`, definiert in der Datei `Rec.java`, und der Fehler ist bei Ausführung einer Anweisung in der Nähe von Zeile 8 entstanden. Die nächste Zeile gibt an, wo diese Methode aufgerufen wurde – zufällig die gleiche Stelle – und so weiter bis zur Methode `main`. Tatsächlich wird in diesem Beispiel nur der oberste Teil des gesamten Stack Trace ausgegeben, weil der sehr große Trace nicht mehr lesbar wäre. Der Stack Trace macht klar, was passiert ist: Dieselbe Methode `rec` wurde immer wieder von derselben Programmstelle aus aufgerufen, bis der Stack voll war. Das ist schon ein sehr deutlicher Hinweis auf die Fehlerursache.

Der Stack Trace wird nach jedem unvorhergesehenen Programmabbruch ohne weiteres Zutun ausgegeben. Es liegt an uns, die Informationen zum Programmablauf, der zum Abbruch geführt hat, richtig zu interpretieren. Mit etwas Geschick ist das gar nicht schwer. Allerdings kommt es vor, dass

der tatsächliche Programmablauf sich von dem erwarteten unterscheidet. Beispielsweise hätten wir erwartet, dass `rec` nur zweimal aufgerufen wird, weil die Zusicherung verlangt, dass `x` größer 0 ist. Um diesem Fehler auf die Spur zu kommen, schalten wir die Überprüfung von Zusicherungen ein. Bei Verwendung einfacher `assert`-Anweisungen bekommen wir bei Verletzung der Zusicherungen zwar eine Fehlermeldung mit einem Stack Trace, aber wir wissen dann noch immer nicht, welcher Wert von `x` den Fehler verursacht hat. Genau um in solchen Situationen mehr Informationen zu bekommen, gibt es eine erweiterte Form von `assert`-Anweisungen, die nach der Zusicherung einen Doppelpunkt und einen String enthält. Der zusätzliche String wird als Teil der Fehlermeldung ausgegeben. Abbildung 5.5 zeigt die Fehlermeldung nach Aufruf von `java -ea Rec`. Das zusätzliche Argument der `assert`-Anweisung in Listing 5.4 setzt einen String aus "`x =`" und dem Wert von `x` zusammen. Die Fehlermeldung enthält den resultierenden String "`x = 0`", also wissen wir, dass `x` den Wert 0 hat. Auf diese Weise können wir die Inhalte beliebiger Variablen als Teil von Fehlermeldungen ausgeben lassen. Zusammen mit dem Stack Trace erhalten wir so recht viel Information über den Programmzustand zum Zeitpunkt des Programmabbruchs.

Manchmal reicht auch die zusätzliche Information über den Programmzustand, die wir nach einer fehlgeschlagenen Überprüfung von Zusicherungen bekommen, nicht aus, um die Fehlerursache zu finden. Wir benötigen auch Informationen über Programmzustände vor dem Programmabbruch. Solche Information können wir über ganz normale Ausgaben erreichen, beispielsweise indem wir die in Listing 5.4 auskommentierte Anweisung ausführen lassen. Damit bekommt man rasch ein Verständnis dafür, wie sich die Werte von `x` in jedem Rekursionsschritt ändern.

Solche Debug-Anweisungen gehören nicht zum eigentlichen Programm, und man darf nicht vergessen, sie zu entfernen oder auszukommentieren, wenn man die Fehlerursache gefunden hat. Normale Programmläufe sollen den Debug Output ja nicht enthalten. Alternativ dazu kann man Debug-Anweisungen als bedingte Anweisungen im Programm belassen, beispielsweise in der Form `if(debug) System.out.println(...);`. Die Boole'sche Variable `debug` setzt man vielleicht entsprechend einem Kommandozeilenargument, das als Eintrag im Array `args` an die Methode `main` übergeben wird. Dadurch kann man den Debug Output beim Programmaufruf auf ähnliche Weise ein- und ausschalten, wie man das mit der Überprüfung von `assert`-Anweisungen macht. In größeren Programmen möchte man Debug Output nicht nur ein- und ausschalten, sondern man

benötigt viel genauere Kontrolle darüber, in welchem Bereich (Klasse, Paket, etc.) welche Art von Debug-Information ausgegeben werden soll. Mit Hilfe der hier gezeigten Technik lässt sich das über mehrere Boole'sche Variablen problemlos bewerkstelligen.

Debug Output, der in den normalen Output des Programms eingestreut ist, kann sehr störend sein. Man kann das vermeiden, indem man Debug Output statt in die Standard-Ausgabe in speziell dafür vorgesehene Dateien schreibt. Entsprechende Dateien nennt man *Logdateien* oder *Protokoll-Dateien*, weil in ihnen alle Ereignisse einer bestimmten Art protokolliert werden. Das Schreiben von Logdateien stört den normalen Programmablauf kaum. Gelegentlich schreibt man alle wichtigen Ereignisse in jedem Programmlauf (nicht nur beim Debuggen) in Logdateien, um für den Fall eines später entdeckten Fehlers leichter nachvollziehen zu können, wodurch der Fehler verursacht wurde. Betriebssysteme greifen häufig auf Logdateien zurück. So kann man beispielsweise auch im Nachhinein leicht feststellen, wann sich wer von wo auf einem Rechner angemeldet hat.

5.4.2 Debugger

Ein *Debugger* ist ein Werkzeug, das uns dabei hilft, den dynamischen Programmablauf durch folgende Funktionalität zu vergegenwärtigen:

- Zur Unterbrechung des Programmablaufs an einer bestimmten Stelle, beispielsweise am Anfang einer Methode oder einer Anweisung im Programm, setzen wir einen *Breakpoint* an diese Stelle. Sobald der Programmfluss diese Stelle erreicht, hält die Programmausführung an und gibt uns Gelegenheit zu weiteren Aktionen. Wir können den Programmfluss auch bei Zugriff auf eine bestimmte Variable oder bei Änderung des Wertes der Variablen unterbrechen lassen. Dazu setzen wir einen Breakpoint auf die Variable. Es können mehrere Breakpoints gleichzeitig aktiv sein und bewirken, dass die Programmausführung unterbrochen wird, sobald einer der Breakpoints erreicht ist.
- Während die Programmausführung unterbrochen ist, können wir den Programmzustand durch Betrachtung von Variableninhalten untersuchen. Es ist auch möglich, Werte von Variablen zu ändern und neue Breakpoints zu setzen bzw. existierende zu entfernen.
- Die unterbrochene Programmausführung kann auch wieder fortgesetzt werden. Neben der normalen Programmausführung mit Unter-

brechung bei Breakpoints können wir auch veranlassen, dass nur ein einzelner Schritt im Programm ausgeführt und die Ausführung danach sofort wieder unterbrochen wird. Für den Fall, dass der Schritt einen Methodenaufruf beinhaltet, können wir wählen, ob wir die Ausführung bereits zu Beginn dieser Methode wieder anhalten wollen (*step into*) oder erst nach Rückkehr aus der Methode (*step over*).

Meist verwenden wir integrierte Entwicklungsumgebungen wie Eclipse, die einen mit wenigen Mausklicks aufruf- und bedienbaren Debugger enthalten. Diese Funktionalität erlaubt uns auf komfortable Weise einen detaillierten Blick in das laufende Programm.

Ein Debugger dient hauptsächlich der Suche nach Fehlerursachen. Man kann ihn aber auch als Hilfsmittel beim Programmierenlernen einsetzen, um die schrittweise Veränderung bestimmter Variablenwerte nachzuvollziehen. Mit ausreichend Programmiererfahrung wird das kaum mehr gemacht, weil diese dynamische Möglichkeit, den Programmablauf nachzuvollziehen, einen im Vergleich zu den gewinnbaren Erkenntnissen zu großen Aufwand verursacht. Ein durch Lesen des Programms gewonnenes statisches Programmverständnis ist meist wertvoller, da es sich nicht nur auf einen Ablauf bezieht, sondern auf alle Abläufe auf einmal.

Beim Debuggen (also der Suche nach Fehlerursachen) ist es – trotz der Einfachheit in der Bedienung des Debuggers – manchmal recht schwierig, genau jene Abläufe im Programm zu sehen zu bekommen, die uns die Ursache des Fehlers deutlich machen. Folgende Probleme treten auf:

- Fehler passieren irgendwo in einem lange laufenden Programm, nicht gleich am Anfang. Wir wissen oft nicht, auf welche Programmstellen und Variableninhalte wir achten müssen. Sogar wenn wir wissen, dass der gesuchte Fehler in einer bestimmten Methode auftritt, so wissen wir meist nicht, in wievielen Methodenaufruf bzw. unter welchen Bedingungen er passiert und wo wir Breakpoints setzen müssen.
- Durch Verändern von Variablenwerten können wir Berechnungen abkürzen und rascher an die Stelle kommen, die uns interessiert. Das kann die Fehlersuche beschleunigen. Allerdings wirkt sich die Änderung möglicherweise auch ganz anders auf das Programmverhalten aus, sodass der gesuchte Fehler gar nicht oder an einer ganz anderen Stelle auftritt als bei einem normalen Programmlauf.
- Gelegentlich wirkt sich die Verwendung des Debuggers auch ohne Veränderung von Variablenwerten auf das Programmverhalten aus.

Klarerweise verändert sich die Laufzeit. Diese Änderung kann in den Ergebnissen sichtbar werden (z.B. wenn das Programm die Zeit misst und abhängig von der Berechnungsdauer andere Zweige ausführt).

- Auswirkungen eines Fehlers zeigen sich oft an anderen Stellen als dort, wo er wirklich passiert ist. Wenn wir uns ganz auf die Stelle konzentrieren, an der wir die Auswirkungen sehen, kommen wir dem Fehler nur sehr schwer auf die Spur.

Mangels genauerer Informationen beginnen wir die Suche nach einer Fehlerursache dort, wo sich der Fehler zeigt. Wir werden wahrscheinlich feststellen, dass an dieser Stelle eine Variable einen unerwarteten Wert hat oder ein Programmzweig ausgeführt wird, der eigentlich nicht ausgeführt werden sollte. Das gibt uns Anhaltspunkte für die weitere Suche. Wir werden also die Stelle suchen, an der die Variable ihren Wert bekommen hat bzw. an der in den aktuellen Programmzweig verzweigt wurde. Dazu muss der Debugger in der Regel das Programm noch einmal von vorne durchlaufen, weil wir die Programmstellen, die uns interessiert hätten, schon verpasst haben. Die Untersuchung dieser Stelle im Programm führt wahrscheinlich wieder zu einer noch früheren Stelle im Programm, an der wir suchen müssen, und so weiter. Dadurch gestaltet sich die Suche nach der Fehlerursache oft sehr langwierig.

Seit kurzem gibt es Debugger, die es uns nicht nur erlauben, die Berechnung Schritt für Schritt mitzuverfolgen, sondern auch zu früheren Programmzuständen zurückzukehren. Diese Debugger können die Fehlersuche etwas beschleunigen. Das Grundproblem, nämlich dass wir nach der sprichwörtlichen „Nadel im Heuhaufen“ suchen, bleibt jedoch bestehen.

Ein weiteres Problem erschwert das Debuggen zusätzlich: In komplexeren Programmen ist uns oft nicht bewusst, welchen Wert eine bestimmte Variable zu einem bestimmten Zeitpunkt haben und welcher Programmzweig durchlaufen werden soll. Solche Information müssen wir uns beschaffen, indem wir entsprechende Programmteile statisch genau zu verstehen versuchen. Manchmal werden wir in die Irre geleitet und nehmen einen Fehler an, wo gar keiner ist, oder betrachten einen falschen Wert als richtig. Dadurch geht viel Zeit verloren.

5.4.3 Eingrenzung von Fehlern

Die Suche nach Fehlerursachen hält stets neue Überraschungen bereit. Auch mit sehr viel Programmiererfahrung stehen wir immer wieder vor

bisher unbekannten Situationen und entdecken Fehler, mit denen niemand auch nur im Entferntesten gerechnet hätte. Es kommt auf viel Fingerspitzengefühl und Intuition sowie auf die Fähigkeit an, aus gewohnten Denkmustern auszubrechen.

Trotz der Unvorhersehbarkeit der Suche geht man fast immer nach demselben Schema vor: Durch Sammeln von Fakten grenzt man die Programmstellen immer weiter ein, bis man die Ursache des Fehlers durch statisches Verstehen des verbliebenen Programmcodes durchblickt. Erst wenn man am Programmcode (nicht nur durch Verfolgen des Programtablaufs) sieht, welche Situationen dazu führen, dass sich Fehler zeigen, hat man eine mögliche Ursache verstanden.

Beim Sammeln von Fakten durchlaufen wir zyklisch folgende Schritte:

Orientierung: In einem ersten Schritt müssen wir uns zur Orientierung einen Überblick darüber verschaffen, in welchem Bereich der Fehler auftritt. Vor allem müssen wir den entsprechenden Teil des Programmcodes betrachten und zu verstehen versuchen. Dabei sehen wir, wie die Methoden und Variablen zusammenhängen und mit dem Auftreten des Fehlers in Verbindung stehen. Falls wir aus dem Programmcode nicht genug Informationen herausfinden, hilft die Betrachtung des Programmzustandes mit einem Debugger weiter.

Hypothese: Wir stellen eine Hypothese darüber auf, welche möglichst kleine und klar abgegrenzte Menge an Variablen, Methoden und Anweisungen an der Entstehung des Fehlers beteiligt sein oder zumindest Hinweise auf die Fehlerursache geben könnte. Die Zustände dieser Variablen und das Verhalten dieser Methoden und Anweisungen müssen wir näher betrachten. Es liegt in der Natur einer Hypothese, dass wir nicht wissen, ob der Fehler tatsächlich dort liegt, wo wir in vermuten. Wir müssen also raten. Mit guten Programmkenntnissen (aufgrund der Orientierung) treffen wir dennoch oft eine gute Wahl.

Planung: Um die Hypothese zu bestätigen oder zu widerlegen, müssen wir die Entwicklung der Programmzustände in den betroffenen Bereichen feststellen. Zunächst legen wir uns einen Plan zurecht, wie wir interessante Ausschnitte aus den Programmzuständen leicht verfolgen können. Beispielsweise identifizieren wir Programmstellen, an denen wir Werte von Variablen in eine Datei ausgeben lassen. Stattdessen können wir auch Breakpoints für den Debugger festlegen, an

denen wir bestimmte Variablenwerte oder Programmabläufe durch schrittweise Ausführung näher betrachten wollen.

Durchführung: Nun sammeln wir die Daten wie geplant. Falls Probleme auftreten, müssen wir zurückgehen und die Planung (falls sich die praktische Ausführung der Pläne nicht einfach genug machen lässt) oder die Hypothese (falls die Durchführung an einer nicht haltbaren oder zu wenig stark fokussierten Hypothese scheitert) überarbeiten.

Auswertung: Schließlich werten wir die im vorigen Schritt gesammelten Daten aus. Bei großen Datenmengen kann das recht aufwendig sein und die Hilfe von (manchmal auch selbst erstellten) Werkzeugen oder Suchfunktionen im Editor erfordern. Wir stellen fest, bis wo welche Daten noch unseren Erwartungen entsprechen und ab wann sie falsch sind. Daraus gewinnen wir Erkenntnisse über die Stelle im Programm, an der die Fehlerursache liegt, und wie wir beginnend mit der Orientierung die Fehlerquelle weiter einschränken können. Manchmal stellt sich aber auch heraus, dass die gesammelten Daten keine Fehler zeigen oder von Anfang an nicht unseren Erwartungen entsprechen; dann ist die Hypothese falsch, und wir müssen mit einer neuen Hypothese weitersuchen. Gelegentlich sind einfach nur die Daten selbst mangelhaft; dann müssen wir zurück zur Planung und Durchführung, um brauchbare Daten zu erhalten.

Diese Schritte sind nicht immer klar voneinander getrennt. In einfacheren Fällen kann die Auswertung bereits während der Durchführung erfolgen, beispielsweise während wir Variablenwerte im Debugger betrachten. Wir können auch rasch Rückschlüsse ziehen, die sich unmittelbar auf die Hypothese auswirken, sodass wir gleich im selben Durchlauf durch den Debugger die Aufmerksamkeit auf andere Variablenwerte lenken.

Manche Fehlerursache ist schnell gefunden und beseitigt. Beispielsweise wird der Compiler einen Fehler melden, wenn der Name einer Variablen falsch geschrieben ist. Dieser Fehler fällt beim sorgfältigen Lesen der Fehlermeldung sofort auf und ist rasch behoben. Generell gilt, dass Ursachen für Fehler, die der Compiler meldet, viel rascher zu finden sind als Fehler, die erst beim Testen entdeckt werden. Zum einen liegt das an informativen Fehlermeldungen, zum anderen an der Art der Fehler. Compiler verstehen das Programm ja nicht und erkennen daher nur relativ einfache Inkonsistenzen im Programm, die wir mit einiger Programmiererfahrung (und mit viel Aufwand) auch selbst ohne größere Probleme erkennen könnten.

Viele Fehler, die erst beim Testen auftreten, haben ihren Ursprung in inhaltlichen Missverständnissen, und ihre Ursachen sind dadurch wesentlich schwieriger zu finden und noch schwieriger zu beseitigen.

Für den effizienten Umgang mit kleinen Fehlern ist es wichtig, Fehlermeldungen richtig zu lesen. Der Compiler kann nur Inkonsistenzen erkennen, aber keine Fehlerursachen. Trotzdem klingen manche Fehlermeldungen so, als ob der Compiler wüsste, was falsch ist. Beispielsweise meldet der Compiler bei inkompatiblen Typen, dass eine Instanz von `int` verlangt wird, aber eine Instanz von `String` im Programmcode gefunden wurde. Keinesfalls dürfen wir diese Meldung als Anweisung für das Ausbessern des Fehlers missverstehen und ohne weitere Prüfung die Instanz von `String` durch eine Instanz von `int` ersetzen. Der Compiler hat nur eine zufällige Annahme getroffen. Um die wirkliche Ursache des Fehlers zu finden, müssen wir selbst den Programmcode betrachten, die Inkompatibilität verstehen und schließlich beseitigen. Meist ist das leicht möglich.

Das Debuggen ist sehr lehrreich. Man lernt dabei vor allem, auf welche möglichen Gefahren man beim Programmieren achten muss. Es ist zwar nicht ausgeschlossen, dass man denselben Fehler mehrfach macht, aber die Wahrscheinlichkeit dafür wird mit der Erfahrung kleiner.

Es soll nochmal darauf hingewiesen werden, dass die Qualität eines Programms durch das Debuggen, also die Suche nach Fehlerquellen und deren Beseitigung, nur wenig verbessert wird. Für die Qualität ist das statische Verstehen des Programms viel wichtiger. Das Finden von Fehlerursachen hilft uns jedoch dabei, das Programm besser statisch zu verstehen. In diesem Zusammenhang ist es nicht verwunderlich, dass das Finden unerwarteter Fehlerursachen, auf die wir beim Lesen des Programmcodes nicht achten, am meisten zur Qualitätsverbesserung beiträgt. Gerade diese Fehlerursachen sind aber am schwersten zu finden.

5.5 Ausnahmebehandlung

Wenn der Java-Interpreter während der Programmausführung einen Fehler entdeckt, *wirft* er eine Ausnahme (Exception). Aber auch spezielle Anweisungen werfen Ausnahmen. Das Werfen einer Ausnahme unterbricht den normalen Programmfluss und leitet eine *Ausnahmebehandlung* ein (*Exception Handling*). Wenn im Programm vorgesehen, wird die geworfene Ausnahme *abgefangen* und ein alternativer Programmzweig als Ersatz für den unterbrochenen Zweig ausgeführt, sodass das Programm auch im

Fehlerfall weiterlaufen kann. Ohne Abfangen der Ausnahme wird das Programm abgebrochen, so wie in Abschnitt 5.4.1 beschrieben.

Zunächst betrachten wir, wie vom System geworfene Ausnahmen in Java abgefangen werden können. Danach beschäftigen wir uns mit selbstdefinierten Ausnahmen und praktischen Aspekten im Umgang mit Ausnahmefällen einschließlich dem Aufräumen, also der Beseitigung von Überresten abgebrochener Programmausführungen.

5.5.1 Abfangen von Ausnahmen

Es gibt zahlreiche Gründe, warum ein Java-Interpreter die Programmausführung wegen eines Fehlers nicht fortsetzen kann. Beispielsweise wirft er eine Ausnahme vom Typ `ArrayIndexOutOfBoundsException` wenn versucht wird, außerhalb der Arraygrenzen auf ein Array zuzugreifen, eine vom Typ `NullPointerException` wenn versucht wird, eine Nachricht an null zu schicken, eine vom Typ `ArithmeticException` wenn eine Zahl durch 0 dividiert werden soll, eine vom Typ `AssertionError` wenn die Bedingung in einer `assert`-Anweisung nicht erfüllt ist, und eine vom Typ `OutOfMemoryError` bzw. `StackOverflowError` wenn nicht genug Speicher für ein neues Objekt bzw. einen weiteren Methodenaufruf vorhanden ist. Diese und ähnliche Fehler lassen sich trotz Typüberprüfungen und sorgfältiger Programmierung nicht gänzlich ausschließen.

Geworfene Ausnahmen werden als ganz normale Objekte dargestellt. Alle Typen dieser Objekte erweitern die vordefinierte Klasse `Throwable`. Diese Klasse stellt eine Reihe von Methoden bereit, um etwas über die Ausnahme erfahren zu können. Die Methode `printStackTrace()` gibt beispielsweise einen Stack Trace aus – genau den, den wir bei einem Programmabbruch aufgrund der Ausnahme zu sehen bekommen.

Wenn wir nichts weiter unternehmen, wird die Programmausführung nach dem Werfen einer Ausnahme abgebrochen und der Stack Trace ausgegeben. Um das zu vermeiden, können wir geworfene Ausnahmen *abfangen*. Dazu verwenden wir Blöcke der Form `try{...} catch(T e){...}`, wobei der `try`-Block eine beliebige Sequenz von Anweisungen enthält, die wie alle anderen Anweisungen ausgeführt werden. Wird jedoch in irgendeiner Anweisung dieser Sequenz eine Ausnahme vom Typ `T` (oder einem Untertyp von `T`) geworfen, so wird als Ersatz für den `try`-Block der `catch`-Block ausgeführt. Die Ausführung des `try`-Blocks wird vorher an der Stelle abgebrochen, an der die Ausnahme geworfen wird (wie jede Ausführung beim Werfen einer Ausnahme abgebrochen wird). Innerhalb des

Listing 5.6: Veranschaulichung des Abfangens einer geworfenen Ausnahme

```

1 public class ExceptionTest {
2     public static void main (String[] args) {
3         try {
4             for (int i = 0; i < 3; i++)
5                 System.out.println(args[i]);           // Achtung:
6             }
7         catch (ArrayIndexOutOfBoundsException ex) { // gefährlich!
8             System.out.println("ERROR: Zu wenige Argumente!");
9         }
10        catch (Exception ex) {                       // OK
11            System.out.println("ERROR (abgefangen):");
12            ex.printStackTrace();
13        }
14        System.out.println("Nichts von einer Ausnahme zu sehen!");
15    }
16 }
```

`catch`-Blocks kann auf das Objekt `e`, das die abgefangene Ausnahme darstellt, wie auf einen formalen Parameter zugegriffen werden. Daher ähnelt `catch(T e){...}` syntaktisch der Definition einer Methode mit einem Parameter. Solche Blöcke nennt man *Exception Handler*. Nach erfolgreicher Beendigung eines Exception Handlers werden die darauf folgenden Anweisungen ganz normal ausgeführt, so als ob keine Ausnahme geworfen worden wäre. Wird im `try`-Block keine Ausnahme geworfen, so wird auch kein Exception Handler ausgeführt. Auf andere Ausnahmen als jene vom im Exception Handler genannten Typ `T` und allen Untertypen von `T` wirkt sich ein Exception Handler nicht aus; sie werden nicht abgefangen.

Das Beispiel in Listing 5.6 veranschaulicht das Abfangen von Ausnahmen. In der Methode `main` in `ExceptionTest` nehmen wir an, dass `args` mindestens drei Elemente hat, also `java ExceptionTest` mit mindestens drei Argumenten aufgerufen wurde. Falls das Array weniger Elemente enthält, wird beim ersten Zugriff auf den ersten nicht mehr erlaubten Index eine Ausnahme `ArrayIndexOutOfBoundsException` geworfen und im ersten Exception Handler abgefangen. Nach einem Programmaufruf werden daher die ersten drei Argumente bzw. bei weniger Argumenten die Argumente und eine Fehlermeldung ausgegeben, in jedem Fall gefolgt von der Zeile „Nichts von einer Ausnahme zu sehen!“, die am Ende von `main` ausgegeben wird.

In diesem Beispiel ist der `try`-Block mit zwei Exception Handlers versehen, die Ausnahmen unterschiedlicher Typen abfangen. Im Allgemeinen können wir beliebig viele Exception Handlers haben. Der zweite und jeder weitere Exception Handler kommt zum Tragen, wenn die geworfene Ausnahme eine Instanz des Typs von diesem Exception Handler ist, aber keine Instanz der Typen von den davor stehenden Exception Handlers. Die Klasse `Exception` ist der Obertyp aller Typen von Ausnahmen, die sinnvoll abgefangen werden können. Der zweite Exception Handler im Beispiel ist dafür gedacht, jede abfangbare Ausnahme außer `ArrayIndexOutOfBoundsException` abzufangen und den Stack Trace `ex` auszugeben, ohne jedoch das Programm gleich zu beenden. Wie das Beispiel zeigt, kann ein einziger Exception Handler viele unterschiedliche Arten von Ausnahmen abfangen – solche vom im Exception Handler bezeichneten Typ sowie allen Untertypen davon.

Nicht für alle Arten von Ausnahmen ist ein Abfangen sinnvoll. Von der Klasse `Throwable` sind die beiden Klassen `Exception` und `Error` direkt abgeleitet. Für Instanzen von `Exception` ist das Abfangen sinnvoll, für jene von `Error`, die nur bei wirklich schwerwiegenden Problemen geworfen werden, jedoch nicht. Ein typisches Beispiel für einen Untertyp von `Error` ist `StackOverflowError`. Der Versuch, eine Ausnahme dieses Typs abzufangen, kann nur wieder zum Werfen einer Ausnahme führen, da für eine sinnvolle Fortsetzung der Programmausführung nicht genug Speicherplatz am Stack vorhanden ist. Üblicherweise erkennt man Untertypen von `Error` daran, dass deren Name mit `Error` endet. Beispielsweise ist auch `AssertionError` eine Art von Ausnahme, die nicht abgefangen werden soll (obwohl dies technisch möglich wäre), weil die Verletzung einer Zusicherung auf einen inkonsistenten Programmzustand hindeutet. Beim Abfangen einer solchen Ausnahme würden weitere Berechnungen wahrscheinlich auf der Basis falscher Daten erfolgen.

Nach dem Werfen einer Ausnahme wird ein passender Exception Handler auf folgende Weise gesucht: Wurde beim Werfen gerade ein `try`-Block mit passendem Exception Handler ausgeführt, so wird die Programmausführung gleich mit diesem Handler fortgesetzt. Gibt es jedoch lokal keinen passenden Exception Handler, aber einen umgebenden `try`-Block mit passendem Exception Handler, so wird mit diesem fortgesetzt. Bei mehreren ineinander geschachtelten `try`-Blöcken wird also von allen damit verbundenen Exception Handlern der innerste gewählt, dessen Typ mit dem der Ausnahme übereinstimmt. Häufig findet man in einer Methode überhaupt keinen passenden Exception Handler. In diesem Fall wird die Suche nach

einem Exception Handler im Aufrufer der Methode fortgesetzt (so als ob die Ausnahme beim Aufruf der Methode geworfen worden wäre), dann in dessen Aufrufer und so weiter. Die Ausnahme wird zum Aufrufer *weitergeleitet* oder *propagiert* (nach engl. *to propagate* – sich fortpflanzen). Der erste passende Exception Handler kommt zum Zug. An der Stelle, an der beim Programmstart `main` aufgerufen wurde, wird schließlich jede Ausnahme abgefangen und der Stack Trace ausgegeben.

Wenn man eine Ausnahme abfängt, weiß man normalerweise nicht sicher, woher die Ausnahme stammt. Das ist ein häufiger Fehler im Umgang mit Ausnahmen. Beispielsweise nehmen wir in Listing 5.6 an, dass eine durch den ersten Exception Handler abgefangene Ausnahme beim Zugriff auf `args[i]` innerhalb des `try`-Blocks geworfen wurde. Diese Annahme wird zwar meistens zutreffen, muss aber nicht immer stimmen. Eine `ArrayIndexOutOfBoundsException` könnte auch während der Ausführung von `println` geworfen und bis zur Methode `main` weitergeleitet werden sein. Um die Quelle der Ausnahme festzustellen, müssten wir den Stack Trace analysieren. Über die Methoden von `Throwable` ist das machbar, aber aufwendig. Daher wird meist darauf verzichtet. Annahmen wie im ersten Exception Handler in Listing 5.6 sind gefährlich, und von der Verwendung dieser Programmier Technik ist daher stets abzuraten. Statt diese Ausnahme abzufangen sollten wir besser mittels `if`-Anweisung zwischen einem Programmaufruf mit ausreichend vielen und zuwenigen Argumenten unterscheiden und die Schleife nur über maximal `args.length` Elemente laufen lassen. Ausnahmebehandlungen sind nur für echte Ausnahmesituationen gedacht, nicht für Situationen, die auch leicht durch andere Sprachkonstrukte beherrschbar sind.

Der zweite Exception Handler im Beispiel trifft dagegen keine Annahmen über die Stelle, an der eine Ausnahme geworfen wird, und er ist auch nicht leicht durch andere Sprachkonstrukte ersetzbar. Genau für solche Fälle ist das Abfangen von Ausnahmen gedacht: Man reagiert angemessen auf die Ausnahme, indem man entsprechende Informationen sammelt und zur Verfügung stellt. Gleichzeitig vermeidet man einen sofortigen Programmabbruch und führt das Programm (möglicherweise eingeschränkt) fort, soweit die Ausnahme das erlaubt.

5.5.2 Umgang mit Ausnahmefällen

Bisher haben wir hauptsächlich Ausnahmen betrachtet, die vom System geworfen werden, um auf Fehler hinzuweisen. Es gibt aber auch Logik-

fehler, die einfach nur zu einem unerwarteten Programmverhalten führen, aber zu keinem vom System erkannten Fehler. Wenn wir im laufenden Programm eine fehlerhafte Situation erkennen, die eigentlich nicht auftreten darf, können wir genauso wie das System eine Ausnahme werfen. Für diesen Zweck gibt es die `throw`-Anweisung. Beispielsweise erzeugt

```
throw new Exception ("Ursache für Ausnahme");
```

eine neue Instanz von `Exception` und wirft diese Instanz als Ausnahme. Wie bei vom System geworfenen Ausnahmen wird die Ausführung daraufhin unterbrochen und nach einem passenden Exception Handler gesucht. Durch eine `throw`-Anweisung geworfene Ausnahmen werden genauso wie vom System geworfene abgefangen oder führen zum Programmabbruch.

Man kann nicht nur selbst Ausnahmen werfen, sondern auch eigene Typen von Ausnahmen einführen. Jede direkt oder indirekt aus `Throwable` abgeleitete Klasse ist als Ausnahme verwendbar. Üblicherweise leiten wir neue Typen von Ausnahmen von `Exception` ab, um klar zu machen, dass diese Ausnahmen abgefangen werden können.

Genaugenommen gehen wir meist davon aus, dass eigene Arten von Ausnahmen tatsächlich abgefangen werden. Abhängig von der Stellung des Typs der Ausnahme in der Klassenhierarchie wird das Abfangen sogar erzwungen: Methoden dürfen nur solche Typen von Ausnahmen weiterleiten, die im Kopf der Methode angeführt sind (siehe unten), sowie alle Untertypen von `Error` und `RuntimeException` (ein Untertyp von `Exception`). Die meisten vom System geworfenen Ausnahmen sind Instanzen einer dieser beiden Typen und können daher immer weitergeleitet werden. Auch Ausnahmen eigener Typen werden weitergeleitet, wenn sie als Unterklassen eines dieser beiden Klassen definiert wurden. Aber Ausnahmen, deren Typen, wie die meisten selbstdefinierten Ausnahmetypen, direkt von `Exception` abgeleitet wurden, werden nicht automatisch weitergeleitet. Diese Ausnahmen müssen durch einen entsprechenden Exception Handler abgefangen werden. Tun sie das nicht, meldet bereits der Java-Compiler einen Fehler, und die Klasse lässt sich nicht übersetzen.

Listing 5.7 zeigt ein Beispiel für die Definition eines eigenen Typs einer Ausnahme sowie die Deklaration einer von einer Methode weitergeleiteten Ausnahme. Die `throws`-Klausel bei der Methode `test` besagt, dass im Rumpf von `test` eine Ausnahme vom Typ `TestException` geworfen und an den Aufrufer weitergeleitet werden kann. Das bedeutet, dass jeder Aufrufer von `test` diese Ausnahme abfangen muss, wenn er sie nicht

Listing 5.7: Weiterleiten von Ausnahmen durch Methoden

```
1 class TestException extends Exception {
2     public TestException (String message) {
3         super(message);
4     }
5 }
6 public class ExceptionPropagationTest {
7     public static void main (String[] args) {
8         try {
9             for (int i = 0; i < args.length; i++)
10                 System.out.println(args[i] + ": " + test(args[i]));
11         }
12         catch (TestException ex) {
13             System.out.println (ex.getMessage());
14         }
15     }
16     private static int test (String s) throws TestException {
17         if (s.equals("end"))
18             throw new TestException ("Ende gut, alles gut");
19         else
20             return s.length();
21     }
22 }
```

aufgrund einer eigenen `throws`-Klausel weiterleiten darf. Die Methode `main` hat keine `throws`-Klausel für `TestException` und daher muss es einen entsprechenden Exception Handler geben.

Im Allgemeinen kann die `throws`-Klausel einer Methode beliebig viele durch Komma voneinander getrennte Untertypen von `Throwable` enthalten. Aufrufer müssen darauf vorbereitet sein, dass jede solche Ausnahme weitergeleitet werden kann. Genau darin liegt der Vorteil: Aufrufer wissen, mit welchen Arten von Ausnahmen sie rechnen müssen. Natürlich kann auch jede Instanz eines Untertyps als Ausnahme weitergeleitet werden, wenn die `throws`-Klausel einen Obertyp davon enthält.

Nachteile von `throws`-Klauseln bestehen im höheren Programmieraufwand und in vermindeter Flexibilität. Wenn Ausnahmen über mehrere Aufrufebenen hinweg weitergeleitet werden sollen, müssen viele Methoden mit `throws`-Klauseln ausgestattet werden. Oft entsteht die Notwendigkeit für das Weiterleiten von Ausnahmen erst recht spät in der Programmentwicklung. Beim Hinzufügen einer Ausnahme müssen vielleicht viele Methoden geändert werden, obwohl die eigentliche Änderung nur die

beiden Stellen betrifft, an denen die Ausnahme geworfen bzw. abgefangen wird. Aus diesem Grund möchte man `throws`-Klauseln vermeiden.

Auch beim Überschreiben von Methoden müssen `throws`-Klauseln berücksichtigt werden: Die Methode in der Unterklasse darf nicht mehr Ausnahmen weiterleiten als die überschriebene Methode der Oberklasse. In der `throws`-Klausel der Unterklasse dürfen wir im Vergleich zu der in der Oberklasse folglich nur Typen weglassen, aber keine neuen hinzufügen. Der Grund dafür ist die Ersetzbarkeit: Wird eine Instanz eines Untertyps dort verwendet, wo eine Instanz eines Obertyps erwartet wird, sind nur die `throws`-Klauseln der Methoden des Obertyps bekannt, und nur diese Ausnahmen müssen abgefangen werden. Wenn die Methoden der Unterklasse andere Ausnahmen weiterleiten würden als die der Oberklasse, könnten nicht alle Ausnahmen abgefangen werden. Die Einschränkung verhindert, dass das passieren kann. Andererseits erhöht die Einschränkung den Wartungsaufwand: Wenn im Laufe der Programmentwicklung eine weitere Ausnahme dazukommt, müssen nicht nur die Methoden geändert werden, welche diese Ausnahme weiterleiten, sondern auch alle entsprechenden Methoden in den Oberklassen. Falls wir keinen Zugriff auf die Oberklasse haben, ist es gar nicht möglich, die Ausnahme hinzuzufügen. Auf die Frage, ob die Vorteile der Verwendung von `throws`-Klauseln deren Nachteile überwiegen, gibt es keine klare Antwort. In manchen Fällen ist es sicher besser, eigene Klassen für Ausnahmen von `RuntimeException` statt von `Exception` abzuleiten.

Dagegen hat die Verwendung eigener Typen gegenüber der Verwendung vordefinierter Typen für Ausnahmen klare Vorteile: Mit eigenen Ausnahmen haben wir volle Kontrolle über alle Stellen im Programm, an denen die Ausnahmen geworfen werden können. In Abschnitt 5.5.1 haben wir gesehen, dass wir bei vom System geworfenen Ausnahmen nicht wissen, wo eine Ausnahme genau geworfen wird, und daher keine Annahmen darüber treffen dürfen. Für eigene Ausnahmen gilt das nicht. Das Abfangen der Ausnahmen wird dadurch wesentlich vereinfacht.

Ausnahmen erlauben uns, vorzeitig aus Sprachkonstrukten auszusteigen. Im Beispiel in Listing 5.7 steigen wir aus einer `for`-Schleife aus, sobald wir auf den String `"end"` treffen. Anders als bei der Verwendung von `break` müssen wir uns jedoch nicht an die lexikalische Struktur des Programms halten (wobei der Ausstiegspunkt textuell innerhalb des Schleifenrumpfes liegen muss), sondern können auch während der Ausführung einer im Schleifenrumpf aufgerufenen Methode aussteigen. Das erhöht die Flexibilität. Andererseits wird die Lesbarkeit vermindert, da die Stelle, an

der eine Ausnahmen abgefangen wird, weit von der Stelle des Werfens entfernt sein kann. Man muss beide Stellen kennen, um die Funktionsweise zu verstehen. Daher sollte man darauf verzichten, Ausnahmen nur zum Zwecke des vorzeitigen Ausstiegs aus Sprachkonstrukten einzusetzen.

Methoden haben nur *einen* Ergebnistyp, und wir können kein Ergebnis eines anderen Typs zurückgeben. Offensichtlich erhöht diese Einschränkung die Lesbarkeit von Programmen ganz wesentlich. Ausnahmen erlauben uns jedoch, diese Einschränkung zu umgehen. Beispielsweise gibt `test` in Listing 5.7 im Normalfall eine ganze Zahl zurück. Beim Werfen einer Ausnahme vom Typ `TestException` wird jedoch eine Zeichenkette an den Exception Handler übergeben und damit quasi an den Aufrufer zurückgegeben. Im Detail geschieht das folgendermaßen: Der Konstruktor von `TestException` gibt die Zeichenkette an den Konstruktor der Oberklasse `Exception` weiter, der die Zeichenkette in einer Objektvariablen speichert. Im Exception Handler liest die von `Exception` geerbte Methode `getMessage` diese Objektvariable aus. Obwohl es manchmal verlockend ist, sollte man aus Gründen der Lesbarkeit darauf verzichten, Ausnahmen nur zum Zwecke der Rückgabe eines Ergebniswertes einzusetzen, der nicht dem Ergebnistyp der Methode entspricht.

Der Einsatz von Ausnahmen ist in echten Ausnahmesituationen gerechtfertigt und ratsam, vor allem im Falle eines Fehlers. Ausnahmebehandlungen erlauben uns dabei, den Programmzustand nach einer unerwarteten Unterbrechung wieder in einen konsistenten Zustand zu bringen und die Programmausführung an einer geeigneten Stelle wieder fortzusetzen. In Abschnitt 5.5.3 werden wir ein gutes Beispiel dafür sehen. Ohne Ausnahmebehandlungen ist es sehr schwierig, mit solchen Ausnahmesituationen umzugehen. Vereinfacht kann man sagen, die Verwendung von Ausnahmen ist überall dort angebracht, wo Lösungen mit anderen Sprachkonstrukten nur unter viel höherem Aufwand möglich wären.

Nicht jeder Exception Handler kann die Auswirkungen der Ausnahme restlos beseitigen. Oft erledigt ein Exception Handler nur einen Teil der Arbeit und wirft dann eine weitere Ausnahme, die von einem weiteren Exception Handler an einer anderen Programmstelle abgefangen wird. Beispielsweise können wir im Exception Handler die gerade abgefangene Ausnahme `ex` durch eine Anweisung `throw ex;` gleich noch einmal werfen. Häufiger werfen wir im Exception Handler jedoch eine andere Ausnahme, beispielsweise durch `throw new Exception(ex);`. Dabei übergeben wir an den Konstruktor von `Exception` (oder eine davon abgeleitete Klasse) die ursprüngliche Ausnahme, die durch die Methode `getCause()`

jederzeit auslesbar ist, sodass keine Information verloren geht. Eine innerhalb eines Exception Handlers geworfene Ausnahme bricht die gesamte `try-catch`-Anweisung ab und kann nur durch eine weiter außen liegende solche Anweisung abgefangen werden.

5.5.3 Aufräumen

Das Werfen einer Ausnahme unterbricht die Programmausführung an einer unerwarteten Stelle und hinterlässt häufig Daten in einem inkonsistenten Zustand. Beim Abfangen der Ausnahme sollten wir aufräumen, also dafür sorgen, dass die Daten danach wieder konsistent sind. Allerdings können die Stellen des Werfens und Abfangens der Ausnahme weit auseinander liegen, sodass der Exception Handler keine Übersicht über den entstandenen Schaden und keinen Zugriff auf die inkonsistenten Daten hat. Außerdem ist beim Aufräumen eine unglaubliche Vielzahl an möglichen Fälle zu unterscheiden, die mit hoher Wahrscheinlichkeit zu Fehlern oder zumindest undurchschaubarem Programmcode führt.

Um das Aufräumen zu erleichtern, unterstützt Java `finally`-Blöcke, die auf `try`-Blöcke und beliebig viele (vielleicht auch keine) Exception Handler folgen und in jedem Fall ausgeführt werden, auch nach dem Werfen einer Ausnahme. In einem `finally`-Block steht der Programmcode zum Aufräumen von allem, was nach Ausführung des dazugehörigen `try`-Blocks aufgeräumt gehört. Im Normalfall wird zuerst ein `try`-Block vollständig ausgeführt, dann der `finally`-Block. Wird die Ausführung des `try`-Blocks aufgrund einer Ausnahme abgebrochen, so wird, falls vorhanden, ein passender Exception Handler zwischen `try`- und `finally`-Block ausgeführt und danach der `finally`-Block. Gibt es für diese Ausnahme keinen passenden Exception Handler zwischen `try`- und `finally`-Block, wird der `finally`-Block ausgeführt und danach die Ausnahme weitergegeben. Dasselbe passiert, wenn innerhalb eines Exception Handlers eine weitere Ausnahme geworfen wird. Falls während der Ausführung des `finally`-Blocks eine Ausnahme geworfen wird, so wird nur diese weitergeleitet, und allenfalls vorher geworfene Ausnahmen werden vergessen.

Listing 5.8 demonstriert in einem Beispielprogramm den richtigen Umgang mit Ausnahmen einschließlich dem Aufräumen. Das Programm öffnet eine Textdatei zum Lesen und eine andere zum Schreiben und kopiert den Inhalt der einen Datei Zeile für Zeile in die andere, wobei die Zeilennummer vor jede Zeile gestellt wird. Am Ende müssen beide Dateien

Listing 5.8: Aufräumen auch nach dem Werfen von Ausnahmen

```

1 import java.io.*;
2 public class Numbered {
3     public static void main(String[] args) {
4         if (args.length != 2) { // Aufruffehler -> keine Ausnahme
5             System.err.println("Usage: java Numbered <in> <out>");
6             return;
7         }
8         try {
9             BufferedReader in = null; // für finally sichtbar
10            BufferedWriter out = null; // Datei offen wenn != null
11            try {
12                String line;
13                in = new BufferedReader(new FileReader(args[0]));
14                out = new BufferedWriter(new FileWriter(args[1]));
15                for (int i=1; (line=in.readLine()) != null; i++) {
16                    out.write(String.format("%6d: %s", i, line));
17                    out.newLine();
18                }
19            }
20            finally { // immer ausgeführt, auch bei Ausnahme
21                if (in != null) // falls Datei geöffnet
22                    in.close(); // dann schließen
23                if (out != null)
24                    out.close();
25            }
26        }
27        catch (IOException ex) {
28            System.err.println("I/O Error: " + ex.getMessage());
29        }
30    }
31 }

```

geschlossen sein. Beim Öffnen, Lesen, Schreiben und Schließen der Dateien können Ausnahmen vom Typ `IOException` geworfen werden, die abgefangen werden müssen. Dafür haben wir einen Exception Handler. Eine Schwierigkeit besteht darin, dass wir offene Dateien auch dann schließen müssen, wenn beim Lesen oder Schreiben eine Ausnahme geworfen wurde. Deswegen sind zwei `try`-Blöcke ineinander geschachtelt. Der innere `try`-Block öffnet die Dateien und kopiert die Daten, und der dazugehörige `finally`-Block schließt die Dateien. So ist sichergestellt, dass die Dateien auch bei Abbruch der beiden `try`-Blöcke geschlossen werden. Der Exception Handler am äußeren `try`-Block fängt sowohl die im inneren

`try`-Block als auch im `finally`-Block geworfenen Ausnahmen des Typs `IOException` ab. Mit nur einem `try-catch-finally`-Konstrukt ist das nicht machbar.

Variablen, die innerhalb eines Blockes deklariert werden, sind außerhalb des Blockes nicht sichtbar. Das gilt auch für `try`-Blöcke. Daher sind die Variablen `in` und `out` nicht im inneren `try`-Block deklariert, sondern im äußeren. Sie müssen ja auch im `finally`-Block zugreifbar sein. Bei der Deklaration werden die beiden Variablen mit `null` initialisiert. Das ist notwendig, damit wir im `finally`-Block feststellen können, ob die zu schließenden Dateien überhaupt geöffnet waren, bevor möglicherweise eine Ausnahme geworfen wurde. Nicht geöffnete Dateien können und brauchen wir natürlich auch nicht schließen. Generell müssen wir beim Entwickeln von `finally`-Blöcken immer in Betracht ziehen, dass `try`-Blöcke nicht oder nur unvollständig ausgeführt wurden.

Zu Beginn des Programms wird die Anzahl der Argumente in `args` über eine einfache `if`-Anweisung abgefragt. Für diesen Zweck vermeiden wir Ausnahmebehandlungen so gut es geht, da falsche Programmaufrufe so häufig vorkommen, dass sie schon eher Normalfälle als Ausnahmefälle sind. Außerdem ist es kaum möglich, entsprechende Ausnahmen von Ausnahmen aufgrund von Programmierfehlern zu unterscheiden. Allerdings sind nicht alle falschen Benutzereingaben durch diese einfache Abfrage abgedeckt. Insbesondere kann es passieren, dass die zum Lesen zu öffnende Datei gar nicht existiert. Dieser Fall führt zu einer `IOException` und einer gut verständlichen Fehlermeldung. Oft entscheiden wir ganz pragmatisch, welche Situationen als Ausnahmefälle und welche als Normalfälle zu betrachten sind. Der einfachere Ansatz ist meist der bessere.

Das Schließen geöffneter Dateien ist ein Paradebeispiel für das Aufräumen. Java unterstützt eine ganze Reihe von Möglichkeiten für den Umgang mit Dateien. Ein Grundprinzip bleibt überall gleich: Zuerst werden Dateien entweder zum Lesen oder Schreiben geöffnet, dann wird daraus gelesen oder darin geschrieben, und schließlich müssen sie wieder geschlossen werden. Jede dieser Operationen kann eine Ausnahme vom Typ `IOException` werfen. Zwei Arten von Dateien werden unterschieden – *Byte Streams*, die nur rohe Daten (Bytes) enthalten, und *Character Streams*, deren Inhalt die Zeichen eines Textes sind. Diese Unterscheidung ist wichtig, weil Java unterschiedliche Zeichensätze unterstützt, deren Zeichen je nach Einstellungen unterschiedlich auf Bytes abgebildet werden. Bei Verwendung von Character Streams erfolgt diese Abbildung automatisch. Weiters kann man *gepufferte* von *ungepufferten* Dateizugriffen unterschei-

den. Ungepufferte Zugriffe verwenden zum Lesen und Schreiben direkt die entsprechenden Befehle des Betriebssystems, während gepufferte Zugriffe aus einem bzw. in einen zwischengeschalteten Puffer schreiben und lesen und erst bei Bedarf den Puffer mit der Datei abgleichen. Der Puffer bewirkt eine Effizienzsteigerung auf Kosten der Aktualität: Gelesene Daten entsprechen möglicherweise bereits einer älteren Dateiversion und geschriebene Daten werden erst verzögert in der Datei sichtbar. Die Variablen `System.in` sowie `System.out` und `System.err` sind Instanzen von `InputStream` sowie `PrintStream`, die ungepufferte Byte Streams darstellen.

Das Beispiel in Listing 5.8 verwendet gepufferte Character Streams. Da der gesamte Inhalt der beiden Dateien auf einmal verarbeitet (gelesen bzw. geschrieben) werden, kommt es nicht auf die Aktualität einzelner Textteile in den Dateien an. Instanzen der Klassen `FileReader` und `FileWriter` entsprechen ungepufferten Character Streams. Instanzen von `BufferedReader` und `BufferedWriter` fügen Puffer dazwischen. Zum Lesen verwenden wir die Methode `readLine`, die eine ganze Zeile auf einmal als Zeichenkette einliest, beim nächsten Aufruf die nächste Zeile und so weiter, bis die Datei keine weitere Zeile mehr enthält und `null` zurückkommt. Die Datei selbst bleibt beim Lesen unverändert. Jede Instanz von `FileReader` und `BufferedReader` ähnelt in vielerlei Hinsicht einem Iterator, wobei `readLine` der Iteratormethode `next` entspricht. Die Methoden `write` zum Schreiben einer Zeichenkette und `newLine` zum Beenden einer Zeile bewirken Änderungen der Datei `out`. Geschrieben wird die durch die statische Methode `format` in `String` erzeugte Zeichenkette. In der Formatzeichenkette `"%6d: %s"` stehen die auf `%` folgenden Zeichen für die Formatierung der Argumente, die zusätzlich an `format` übergeben werden. Durch `%6d` wird die Zahl `i` als ganzzahliger Wert mit 6 Stellen ausgegeben. Es folgen ein Doppelpunkt und ein Leerzeichen, die auch so ausgegeben werden. Schließlich steht `%s` für die Ausgabe von `line` als Zeichenkette.

5.6 Validierung

Der Begriff *Validierung* (zu deutsch *Bewertung*) ist überladen und kann vielerlei bedeuten. Wir wollen hier zwei ganz unterschiedliche Bedeutungen etwas näher betrachten, die beide im Zusammenhang mit der Programmkonstruktion stehen.

5.6.1 Validierung von Daten

Daten, die in einem Programm verarbeitet werden, können aus ganz unterschiedlichen Quellen stammen – beispielsweise aus einer Datenbank, einer Benutzereingabe, einer Messung oder dem Ergebnis früherer Berechnungen. Nicht alle Quellen sind gleich zuverlässig. Insbesondere sind Daten, die von Benutzern eingegeben wurden oder aus Messungen stammen, oft falsch. Wir wollen vermeiden, mit falschen Daten weiterzurechnen. Leider gibt es keine Möglichkeit festzustellen, ob die Daten richtig sind. Aber wir können überprüfen, ob sie stimmen könnten. Wenn beispielsweise jemand als Geburtsdatum den 30. Februar 2345 angibt oder ein Sensor im Hochofen eine Temperatur von -543°C misst, sind wir ziemlich sicher, dass die Daten nicht stimmen. Sie sind nicht *plausibel*. Unter Validierung verstehen wir in diesem Zusammenhang die Beurteilung, ob uns Daten als zuverlässig genug erscheinen, um damit weiterzurechnen, oder nicht. Das nennt man auch *Plausibilitätsprüfung*. Je nach Situation können wir nicht plausible Daten einfach verwerfen oder zurückweisen. Manchmal ist sogar ein Programmabbruch denkbar.

Bereits in den ersten Beispielen (in der Klasse `Zahlenraten` in Abschnitt 1.1) waren Plausibilitätsprüfungen nötig: Benutzereingaben sollen Zahlen im Wertebereich zwischen 0 und 99 darstellen. Einerseits wird überprüft, ob eine Eingabe überhaupt eine Zahl ist, andererseits ob die Zahl im gewünschten Wertebereich liegt. Diese beiden Überprüfungen müssen wir unterscheiden, weil unterschiedliche Maßnahmen zur Bereinigung der Situation nötig sind. Im einen Fall muss die falsche Eingabe entfernt werden um weitermachen zu können, im anderen nicht. Solche Notwendigkeiten für Unterscheidungen kommen häufig vor. Trotzdem sollte man die Anzahl solcher Unterscheidungen auf das absolut notwendige Maß beschränken. Je mehr Fälle man unterscheiden muss, desto größer ist die Wahrscheinlichkeit, dass wir wichtige Fälle übersehen und mit falschen Daten arbeiten. Einfache und klare Bedingungen dafür, wann die Daten als plausibel gelten sind sehr erstrebenswert.

Bei Plausibilitätsprüfungen darf man weder zu lockere noch zu restriktive Kriterien anlegen. Man kommt leicht in Versuchung, alles ausschließen zu wollen, wofür man sich gerade keine sinnvolle Verwendung vorstellen kann. Das vermindert die Verwendbarkeit des Programms. Im Bewusstsein dieser Gefahr kommt man umgekehrt auch in Versuchung, fast alles zu tolerieren. Dies kann später zu Problemen führen, weil an zahlreichen Stellen im Programm unnötige Sonderfälle behandelt werden müssen. Die

richtige Wahl der Plausibilitätskriterien erfordert Erfahrung und viel Fingerspitzengefühl.

Man könnte Plausibilitätsprüfungen dort vornehmen, wo die in den Überprüfungen enthaltenen Bedingungen benötigt werden. Diese Strategie würde leider dazu führen, dass große Teile des Programmcodes mit unzusammenhängenden, eingestreuten Plausibilitätsprüfungen übersetzt wären. Das wäre schlecht faktorisierter Code. Es ist eher sinnvoll, Plausibilitätsprüfungen an Schnittstellen vorzunehmen, wo die Daten in das System übernommen werden, für Benutzereingaben beispielsweise an der Benutzerschnittstelle. An diesen Stellen sind Situationen, die aus fehlgeschlagenen Überprüfungen entstehen, meist auch am leichtesten zu bereinigen.

Im Idealfall hat man ein eigenes Modul (z.B. eine Klasse oder ein Paket) für die Plausibilitätsprüfungen. Damit kann man Überprüfungen vereinheitlichen, die sonst an verschiedenen Schnittstellen gemacht werden müssten, beispielsweise an der Benutzerschnittstelle und an der Schnittstelle zu einer Datenbank. Die zentrale Stelle lässt sich leichter konsistent halten. Implausible Daten müssen jedoch von den einzelnen Schnittstellen bereinigt werden, weil beispielsweise auf falsche Benutzereingaben ganz anders reagiert werden muss als auf falsche Daten, die aus einer Datenbank gelesen werden.

Plausibilitätsprüfungen sind klar von Zusicherungen zu unterscheiden. Während Zusicherungen in korrekt funktionierenden Programmen eigentlich niemals verletzt sein dürften, muss man bei Plausibilitätsprüfungen stets mit einer Verletzung rechnen. Die Überprüfung von Zusicherungen ist daher im Normalfall ausgeschaltet, Plausibilitätsprüfungen dürfen dagegen nicht ausgeschaltet werden. Daher eignen sich `assert`-Anweisungen nicht für Plausibilitätsprüfungen. Ein weiterer Unterschied besteht darin, dass Zusicherungen über das ganze Programm verstreut überall vorkommen, während Plausibilitätsprüfungen in gut faktorisierten Programmen nur an Schnittstellen zur Außenwelt (oder in einem zentralen Modul, das von diesen Schnittstellen verwendet wird) vorkommen.

Generell gilt der Grundsatz: „Never trust the user.“ Also alles, was direkt oder indirekt von Benutzern eines Systems kommt, sollte überprüft werden. Allerdings gilt dieser Grundsatz nicht für Entwickler. Man muss darauf vertrauen, dass alle an der Konstruktion des Programms beteiligten Personen die Regeln und Zusicherungen einhalten. Ohne Vertrauen kann kein gutes Programm entstehen. Anwender des Programms kennen die Regeln und Zusicherungen aber nicht. Plausibilitätsprüfungen sind daher kein Zeichen mangelnden Vertrauens, sondern eine Notwendigkeit.

5.6.2 Validierung von Programmen

Bei der Verifikation und beim Testen wird festgestellt, wie gut das Programm hinsichtlich der Anforderungsspezifikation ist und in welchen Bereichen Verbesserungsbedarf besteht. Im Softwareengineering ist die Validierung die Bewertung des Programms hinsichtlich der tatsächlichen Anforderungen. Die Anforderungen können sich verschoben haben, sodass sie von der Spezifikation nur mehr unzureichend widerspiegelt werden. Die Verifikation überprüft, ob das Programm *richtig entwickelt* wird, während die Validierung überprüft, ob das *richtige Programm* entwickelt wird.

Durch verschiedene Maßnahmen versucht man sicherzustellen, dass das richtige Programm entwickelt wird:

- Gespräche mit künftigen Anwendern, um Unklarheiten und Fehler in der Analyse möglichst früh aufzudecken
- Entwicklung von Benutzeroberflächen-Prototypen (Programmen mit Benutzerschnittstellen ähnlich denen der fertigen Produkte, aber ohne Funktionalität dahinter) mit dem Zweck, künftigen Benutzern die Verwendung der Programme zu zeigen und Feedback einzuholen
- in manchen Projekten Einbeziehung und Mitarbeit eines künftigen Benutzers (als Fachexperten) in die gesamte Entwicklung
- Inkrementelle Entwicklungsmethoden, die ein frühes Feedback durch Anwender ermöglichen
- kurze Releasezyklen, die es erlauben, rasch auf geänderte Anforderungen zu reagieren

Eine wichtige Frage ist die nach den Konsequenzen der Validierung. An obiger Liste ist sofort zu sehen, dass rasches Feedback von künftigen Anwendern im Mittelpunkt der Maßnahmen steht. Sobald wir am Feedback erkennen, dass die Entwicklung der Software in eine falsche Richtung läuft, müssen wir sofort gegensteuern. Die Entwicklung wird damit von den Wünschen der Anwender getrieben.

Ganz so einfach ist die Sache aber nicht. Viele Anwender wünschen sich natürlich immer bessere Programme, die alles können und gleichzeitig einfach zu bedienen sind. Alleine das ist schon ein Widerspruch in sich, da mit dem Funktionsumfang üblicherweise auch die Komplexität der Bedienung steigt. Viel wichtiger sind die Kosten. Auch wenn es möglich wäre, sehr gute Programme zu entwickeln, stehen die Ressourcen dafür meist nicht

zur Verfügung. Vor allem bei inkrementellen und agilen Softwareentwicklungsprozessen stehen wir ständig vor der Entscheidung, welcher Schritt als nächster gemacht werden soll. Wir können uns für den Schritt entscheiden, der den Wünschen der Anwender am ehesten entgegenkommt, aber vielleicht die Kosten in die Höhe treibt, oder den, der die Kosten im Rahmen hält, aber manche Anwenderwünsche unberücksichtigt lässt.

Um den Kostenfaktor einzubeziehen, sollten wir statt von künftigen Anwendern eher von Auftraggebern (bzw. dessen Vertretern) sprechen. Von ihnen stammt in der Regel das Geld. Aber auch dabei müssen wir vorsichtig sein, da die Auftraggeber meist nicht selbst Anwender sind. Es müssen sowohl Auftraggeber als auch echte Anwender einbezogen werden. Im Endeffekt treffen zwar die Auftraggeber alle Entscheidungen, die sich auf die Kosten auswirken, aber ein vernünftiger Auftraggeber wird sich an gerechtfertigten Wünschen der Anwender orientieren. Gerade inkrementelle und agile Softwareentwicklungsprozesse funktionieren nur unter Einbeziehung aller Beteiligten gut. Andererseits kann sich die Einbeziehung zu vieler Personen in Entscheidungsprozesse auch negativ auswirken – nach dem Motto: „Viele Köche verderben den Brei.“ Es ist nicht leicht, ein ausgewogenes Maß zu finden.

Man kann die Validierung als *wirtschaftlichen Begriff* betrachten. Dabei stellt man, unter Abschätzung des Risikos, den Wert des Produkts ganz nüchtern in Relation zu den Kosten der Realisierung. Das Produkt ist das zu konstruierende Programm. Sowohl der Wert des Programms als auch die Kosten der Realisierung sind nur grob abschätzbar. Genau diese Unwägbarkeiten lassen sich durch eine Abschätzung des Risikos beurteilen. Man wird ein Projekt nur machen, wenn es mit ausreichender Wahrscheinlichkeit einen Gewinn verspricht. Die Benutzbarkeit des Programms ist in dieser Betrachtung nur einer von vielen Faktoren, die den Wert des Programms bestimmen.

Solche wirtschaftlichen Überlegungen kann und soll man auch bei laufenden Projekten anstellen. Mitten im Projekt lassen sich der Wert des Produkts, die Kosten und das Risiko besser abschätzen als vor Beginn. Die Validierung beantwortet beispielsweise folgende Fragen:

- Zahlt es sich aus, das Programm weiterzuentwickeln, oder soll man das Projekt abbrechen und die Investition als verloren betrachten?
- Wohin soll die Weiterentwicklung gehen – in Richtung einer Minimalversion oder in Richtung eines großen umfangreichen Systems?

- Wie kann man durch unterstützende Maßnahmen (Werbekampagnen, Ver- oder Zukauf von Lizenzen, Standardisierung, Nutzung von Synergien, etc.) den Wert des Programms erhöhen und die Entwicklungskosten reduzieren?

Manchmal bezeichnet man einfach nur den Abnahmetest als Validierung. Der Abnahmetest untersucht, ob das Programm im praktischen Betrieb das hält, was man sich anfangs davon versprochen hat, nicht nur, ob irgendwelche künstlichen Spezifikationen erfüllt sind. Insofern ist diese Bezeichnung gerechtfertigt. Die genaue Form des Abnahmetests ist meist vertraglich festgelegt. Sollten Fehler auftreten oder die Benutzbarkeit nicht gegeben sein, muss das Programm nachgebessert werden.

5.7 Qualität sichern lernen

Die Bedeutung der Qualitätssicherung in der Programmierung wird häufig unterschätzt. Es ist nicht damit getan, ein fertiges Programm auszuprobieren, um deren Qualität festzustellen. Die Qualität muss in jeder Phase der Softwareentwicklung gewährleistet werden. Wir müssen unser gesamtes Vorgehen bei der Programmkonstruktion auf Qualität ausrichten.

5.7.1 Konzeption und Empfehlungen

In diesem Kapitel wurde immer wieder betont, dass die Qualität unserer Programme ganz wesentlich von unserem statischen Verständnis der Programme abhängt. Ein solches Programmverständnis können wir uns nur durch praktisches Programmieren aneignen. Es kommt kaum darauf an, was wir programmieren, sondern hauptsächlich darauf, dass wir viel programmieren und uns ständig darum bemühen, alle Details unserer Programme statisch zu verstehen. Die Qualität der Programme steigt mit unserer Programmiererfahrung. Natürlich hilft theoretisches Wissen dabei, das Programmierenlernen zu beschleunigen, aber ohne eigene praktische Erfahrungen nützt theoretisches Wissen nicht viel. Man muss dieses Wissen praktisch anwenden, um wirklich zu erfahren, was die Qualität eines Programms ausmacht und wie man eine gute Qualität sicherstellt. Das gilt vor allem auch für das Testen und Debuggen. Ein Gefühl dafür bekommt man nur, wenn man es selbst macht.

Im Gegensatz zum vorigen Kapitel wurde in diesem Kapitel theoretisches Wissen relativ direkt mit nur wenigen Beispielen vermittelt. Der

Grund dafür liegt darin, dass vorgegebene Beispiele kaum dazu geeignet sind, ein Gefühl für praktische Probleme im Zusammenhang mit dem Programmverstehen und der Qualitätssicherung entstehen zu lassen. Im Idealfall sollte man durch Programmieren eigene, individuelle Erfahrungen sammeln und diese Erfahrungen in Relation zum theoretischen Wissen setzen. Das selbständige, individuelle Lösen von Problemen fördert die Kreativität, und Kreativität spielt sowohl beim Verstehen von Programmen als auch beim Testen und Debuggen eine große Rolle.

5.7.2 Kontrollfragen

- Was versteht man unter einer Spezifikation?
- Wie genau soll eine Spezifikation sein? Von welchen Faktoren hängt diese Genauigkeit ab?
- Auf welche Arten kann man ein Programm spezifizieren?
- Wie geht man vor, wenn man Widersprüche oder Ungenauigkeiten in einer Spezifikation entdeckt?
- Was versteht man unter Design by Contract?
- Welche Bestandteile eines Softwarevertrags sind vom Client zu erfüllen, welche vom Server?
- Woran erkennt man, ob eine Bedingung ein Vorbedingung, Nachbedingung oder Invariante darstellt?
- Wie müssen sich Vor- und Nachbedingungen bzw. Invarianten in Unter- und Obertypen zueinander verhalten?
- Warum stellen Invarianten einen Grund dafür dar, dass man keine `public` Variablen verwenden sollte?
- Inwiefern lassen sich Bedingungen statt als Zusicherungen auch in Form von Typen ausdrücken?
- Welche Rolle spielen Namen in Programmen (im Zusammenhang mit Zusicherungen)?
- Was versteht man unter der Namensgleichheit bzw. Strukturgleichheit von Typen?

- Was ist Duck Typing, was das Gegenteil davon?
- Warum soll man Programme eher statisch als dynamisch verstehen?
- Wozu verwenden wir `assert`-Anweisungen?
- Sind `assert`-Anweisungen auch sinnvoll, wenn deren Überprüfung ausgeschaltet ist? Warum?
- Was ist eine Schleifeninvariante? Wozu verwenden wir sie?
- Wann muss eine Schleifeninvariante gelten?
- Wie gehen wir vor, um geeignete Schleifeninvarianten zu finden?
- Wann ist der richtige Zeitpunkt um Zusicherungen (insbesondere auch Schleifeninvarianten) in den Programmcode zu schreiben?
- Welche Formen von Zusicherungen ersetzen Schleifeninvarianten bei Verwendung von Rekursion statt Iteration?
- Wodurch unterscheidet sich die partielle von der vollständigen Korrektheit eines Programms?
- Wann terminiert eine Schleife oder Rekursion?
- Was muss man zeigen, um die Termination formal zu beweisen?
- Gibt es praktische Unterschiede zwischen der Berechnung des zeitlichen Aufwands und dem Beweis der Termination? Wenn ja, welche?
- Spielen Terminationsbeweise auch in Programmen, die niemals terminieren sollen, eine Rolle?
- Inwiefern hängen Korrektheitsbeweise mit dem statischen Verstehen eines Programms zusammen?
- Was kann man mittels Model Checking machen?
- Was versteht man unter Denial-of-Service-Attacken?
- Kann man die Fehlerfreiheit eines Programms sicherstellen? Wenn ja, wie?
- Ist es besser, beim Testen viele Fehler zu finden als wenige? Warum?

- Wie können beim Beseitigen eines Fehlers neue Fehler entstehen?
- Soll man auch Fehler korrigieren, bei denen die Wahrscheinlichkeit für ein zufälliges Auftreten des Fehlers äußerst gering ist? Warum?
- Aus welchen Gründen könnte jemand das Eindringen in ein System absichtlich ermöglichen?
- Wodurch führt Testen zu einer Qualitätsverbesserung?
- Was versteht man unter Code Reviews?
- Welche Ziele, Gemeinsamkeiten und Unterschiede gibt es zwischen Unit-, Integrations-, System- und Abnahmetests.
- Was versteht man in der Informatik unter einem Stresstest?
- Was charakterisiert White-, Grey- und Black-Box-Tests?
- Wofür stehen bei Laufzeitmessungen `real`-, `user`- und `sys`-Werte?
- Wovon hängen Laufzeiten ab, und wodurch unterscheiden sich gemessene Laufzeiten oft so stark voneinander?
- Wodurch unterscheiden sich Latenz und Bandbreite voneinander, und wann dominiert einer dieser Begriffe den anderen?
- Kann man gemessene Laufzeiten auf größere Datenmengen hochrechnen?
- Was versteht man unter weichen und harten Echtzeitsystemen?
- Was ist ein Stack Trace? Welche Informationen enthält er?
- Wie kann man mittels `assert` Anweisungen beim Auftreten von Fehlern etwas über den Programmzustand erfahren?
- Was versteht man unter Debug Output?
- Wozu dienen Logdateien?
- Was kann man mit einem Debugger machen?
- Warum ist das Finden von Fehlerursachen oft so schwierig?

- Wie gehen wir beim Sammeln von Fakten zum Auffinden einer Fehlerursache vor?
- Warum lassen sich Ursachen für Fehler, die vom Compiler gemeldet werden, häufig einfacher finden als die für andere Fehler?
- Was versteht man unter einer Ausnahmebehandlung?
- Wie und warum fängt man Ausnahmen ab?
- Welche Ausnahmen können in Java immer an den Aufrufer propagiert werden, welche nicht?
- Wie wird nach dem passenden Exception Handler gesucht?
- Wofür sind Ausnahmebehandlungen gedacht, wozu sollten sie eher nicht verwendet werden?
- Wozu dienen die Schlüsselwörter `throw` und `throws` in Java, und wie verwendet man die entsprechenden Sprachkonzepte?
- Welche Schwierigkeiten können durch `throws`-Klauseln entstehen?
- Was macht man mit `finally`-Blöcken? In welchen Situationen werden sie ausgeführt?
- Wofür sind bedingte Anweisungen besser geeignet als Zusicherungen?
- Was unterscheidet Byte Streams von Character Streams in Java?
- Was unterscheidet gepufferte von ungepufferten Dateizugriffen?
- Wozu dient die Methode `format` in `String`?
- Was ist eine Plausibilitätsprüfung?
- Wohin soll man den Code für Plausibilitätsprüfungen schreiben, wohin nicht?
- In welchen Situationen soll man Plausibilitätsprüfung vornehmen, in welchen nicht?
- Wodurch ähneln Plausibilitätsprüfungen und Zusicherungen einander, wodurch unterscheiden sie sich?
- Was kann man mit der Validierung von Programmen bewirken?

6 Vorsicht: Fallen!

Die Programmkonstruktion ist ein Abenteuer. In zahlreichen Winkeln und Ecken lauern unbekannte Gefahren und drohen unsere Anstrengungen zunichte zu machen. Aber genau darin liegt der Reiz. Nur wer sich in die Abgründe der Programmierung wagt und es schafft, die unzähligen Fallen auf dem Weg zur Problemlösung geschickt zu umgehen, kann den Stolz auf diese Leistung verstehen. Dafür wird man sich immer wieder in neue, noch gefährlichere Programmierabenteuer stürzen.

Um unsere Feinde zu besiegen, müssen wir sie kennen. In diesem Kapitel wollen wir einige gefährliche Fallen auf dem Weg zu einem hochwertigen Programm näher betrachten. Wir werden sehen, wie wir Fallen rechtzeitig erkennen und umgehen können.

6.1 Beschränkte Ressourcen

Die wichtigsten Ressourcen eines Computers sind beschränkt. Beispielsweise haben Hauptspeicher und Festplatten bestimmte Größen, und es ist eine gewisse Anzahl an Prozessorkernen mit beschränkter Rechenleistung vorhanden. Alle Programme auf dem Computer teilen sich diese Ressourcen. Daher müssen die Programme sparsam damit umgehen. Beispielsweise wird ein Programm, das ständig neuen Speicher anfordert ohne nicht mehr gebrauchten Speicher freizugeben, nach kurzer Zeit den gesamten verfügbaren Speicher aufgebraucht haben. Meist ist es einfach, neue Ressourcen anzufordern, aber viel schwieriger, belegte Ressourcen wieder freizugeben. Das kann zu unnötiger Ressourcenverschwendung führen.

6.1.1 Speicherverwaltung

Speicher ist eine der wichtigsten beschränkten Ressourcen. Daher muss einer effizienten Verwaltung des Speichers große Aufmerksamkeit gewidmet werden. In Java ist die Speicherverwaltung zum größten Teil automatisiert, das heißt, das System übernimmt den Hauptteil der Aufgabe, und beim Programmieren brauchen wir nur darauf zu achten, dass wir das System

dabei nicht behindern. Konkret verwendet Java einen *Garbage Collector* (auf deutsch etwa „Müllsammler“), der den von nicht mehr zugreifbaren Objekten belegten Speicherplatz automatisch wieder freigibt. Der Garbage Collector arbeitet unsichtbar im Hintergrund.

Wie in fast allen aktuellen Programmiersprachen unterscheiden wir in Java zwei Speicherbereiche, den *Stack* und den *Heap*. Wie wir schon gesehen haben, wird bei jedem Methodenaufruf ein neuer Eintrag auf den Stack gelegt. Der Stackeintrag enthält neben Verwaltungsinformation (wie beispielsweise die Adresse, an die das Programm nach Beendigung der Methode zurückkehren soll) die Parameter sowie lokalen Variablen der Methode. Bei Beendigung der Methode wird der Stackeintrag wieder abgebaut. Der Heap enthält alle durch `new` erzeugten Objekte, die auch nach Beendigung einer Methode erhalten bleiben. Heapeinträge können nur durch den Garbage Collector abgebaut werden. Der Garbage Collector sucht regelmäßig nach allen zugreifbaren Objekten und markiert diese, danach entfernt er alle nicht markierten und daher auch nicht zugreifbaren Objekte. Er beginnt mit der Suche im Stack, genauer bei den Parametern und lokalen Variablen in allen Stackeinträgen. Parameter und Variablen, die Objekte enthalten, verweisen auf die im Heap von den Objekten belegten Speicherbereiche. Von dort sucht der Garbage Collector in allen Variablen der Objekte (Objektvariablen und statische Variablen) weiter, bis alle auf diese Weise auffindbaren Objekte markiert sind.

Garbage Collection ist eine bewährte Technik, die uns beim Programmieren viel Arbeit abnimmt. Auf eine Kleinigkeit müssen wir aber doch achten: Oft bleiben Objekte zugreifbar, obwohl sie tatsächlich nicht mehr benötigt werden. Dadurch geht Speicher verloren. Wir können den Garbage Collector unterstützen, indem wir die Inhalte aller nicht mehr benötigten Variablen auf `null` setzen. Damit können die von den Objekten in den Variablen belegten Speicherzellen möglicherweise freigegeben werden, wenn diese Objekte nicht auch noch in anderen Variablen liegen. Generell ist es eine gute Idee, an nicht mehr benötigte Variablen `null` zuzuweisen. Unabhängig davon, ob zusätzlicher Speicher freigegeben wird, können wir beim Lesen des Programms sofort erkennen, dass über die Variablen nicht mehr auf die vorher darin enthaltenen Objekte zugegriffen wird. Das verbessert die Lesbarkeit.

Man fragt sich, wie die Speicherverwaltung eine „Falle“ sein kann, wenn Garbage Collection bis auf das notwendige auf-`null`-Setzen von Variablen automatisch und meist sehr gut funktioniert. Das Problem ist, dass Garbage Collection zwar meistens, aber eben nicht immer so gut funktioniert.

Unter anderem treten folgende Schwierigkeiten auf:

- Garbage Collection kostet etwas Zeit und ist gefühlt immer dann notwendig, wenn man es sich am wenigsten wünscht. Tatsächlich braucht Garbage Collection insgesamt oft weniger Zeit als die explizite Verwaltung von Speicher durch das Programm. Diese Tatsache ändert jedoch nichts am Gefühl, dass man beim Programmieren kaum Kontrolle darüber hat, wann Garbage Collection zu kleinen Verzögerungen bei den Antwortzeiten führen kann.
- Das Freigeben nicht mehr benötigten Speichers ist nur ein kleiner Teil der Speicherverwaltung. Man kann Programme schon durch die Wahl der Algorithmen so auslegen, dass sie viel oder wenig Speicher verbrauchen. Wenn man eine Variante wählt, in welcher der benötigte Speicher den verfügbaren übersteigt, kann auch die beste automatische Speicherverwaltung nicht genug Speicher zur Verfügung stellen.
- Es gibt Bereiche, in denen eine automatische Speicherverwaltung nicht sinnvoll ist, vor allem in sicherheitskritischen Systemen. Hier braucht man Garantien dafür, dass der benötigte Speicher in einer festgelegten Zeit verfügbar ist. Eine automatische Speicherverwaltung kann das nicht garantieren, wenn das Programm immer wieder neuen Speicher anfordert. Man muss also das ganze Programm so auslegen, dass der gesamte benötigte Speicher von Anfang an verfügbar ist. In diesem Bereich verwendet man eher Sprachen wie C, mit denen man auf niedriger Ebene mehr Kontrolle hat.
- Manchmal ist es aufwendig, nicht mehr benötigte Variablen auf `null` zu setzen, wenn man keinen direkten Zugriff darauf hat. In diesen Fällen verzichtet man bewusst darauf und nimmt zugunsten kürzerer Laufzeiten einen höheren Speicherverbrauch in Kauf. Auswirkungen auf sehr lange laufende Programme sind jedoch kaum abschätzbar.

Ironischerweise führt gerade der Versuch, die Speicherverwaltung in Java besser zu kontrollieren, zu vielfältigen Problemen. Meist wollen wir die Speicherverwaltung kontrollieren, weil sich irgendwo ein Problem bei der automatischen Speicherverwaltung zeigt. Jedoch betreffen Eingriffe nur selten die Ursache des Problems und führen deshalb oft nicht zum Ziel. Zumindest folgende Eingriffsmöglichkeiten stehen zur Verfügung:

- Eine Ausnahme vom Typ `StackOverflowError` wird geworfen, wenn der Stack zu klein ist. Java-Interpreter lassen uns die Größe eines Stacks selbst bestimmen. Beispielsweise können wir über die Option `-Xss1m` beim Start des Java HotSpot-Interpreters die Stackgröße auf 1 Megabyte setzen. Allerdings ist die Ursache für das Werfen dieser Ausnahme in den meisten Fällen eine Endlosrekursion wie im Beispiel in Listing 5.4. Ein größerer Stack löst das Problem nicht, sondern es dauert nur länger, bis die Ausnahme geworfen wird. Um rascher eine Fehlermeldung zu sehen, wird die Stackgröße normalerweise klein, aber für die meisten Programme groß genug gewählt. Es kann selten aber doch vorkommen, dass ein bestimmtes Programm mit der üblichen Stackgröße eine Ausnahme wirft, mit einem größeren Stack aber nicht. Daher kann man sein Glück nach einem `StackOverflowError` mit einem größeren Stack versuchen, auch wenn der Versuch kaum mit Erfolg gekrönt sein wird.
- Anders als der Stack wird der Heap automatisch vergrößert, wenn es notwendig ist und der Computer genug Speicher hat. Trotzdem können wir eingreifen: Beim Start des Java HotSpot-Interpreters legen wir beispielsweise über die Option `-Xmsn6m` eine Mindestgröße von sechs Megabyte und über `-Xmxn66m` eine Maximalgröße von 66 Megabyte fest. Das ist etwa beim Testen des Programms sinnvoll, um die Lauffähigkeit auf kleineren Computern zu überprüfen oder die Auswirkungen der Garbage Collection zu untersuchen.
- Es ist möglich, die Garbage Collection in Java explizit aufzurufen:

```
Runtime r = Runtime.getRuntime();
r.gc();
```

Wenn wir die Garbage Collection an Stellen ausführen lassen, an denen sie weniger stört, ist die Wahrscheinlichkeit kleiner, dass sie auch an anderen Stellen notwendig ist. Diese Technik leidet aber darunter, dass wir die Garbage Collection wahrscheinlich viel häufiger durchführen, als sie notwendig wäre. Außerdem ist es schwer, Stellen im Programm zu finden, an denen die Garbage Collection eine ausreichende Menge an Speicher freigeben kann.

- Garbage Collection ist kompliziert und von vielen Parametern abhängig. Moderne Java-Systeme lassen diese Parameter steuern, und

ein Experte kann damit tatsächlich Verbesserungen erzielen. Allerdings sollte man ohne spezielles Wissen über Garbage Collection und Details von Java-Interpretern die Finger davon lassen, da man viel eher eine Verschlechterung als eine Verbesserung erzielt.

- Jedes Java-Objekt hat die Methode `finalize`, die ausgeführt wird, bevor Garbage Collection den Speicherplatz für das Objekt freigibt. Darin könnte man theoretisch irgendwelche Ressourcen freigeben. Leider hat das Überschreiben von `finalize` durch eine nicht-leere Methode auch unerwünschte Auswirkungen. So kann der Speicherplatz nicht gleich freigegeben werden, weil vorher die Methode ausgeführt werden muss, und der Speicherbedarf kann dadurch steigen. Daher und wegen der mangelnden Kontrollierbarkeit des Zeitpunktes der Ausführung wird von `finalize` kaum Gebrauch gemacht.
- Man kann die automatische Speicherverwaltung bewußt umgehen. Beispielsweise legt man gleich zu Beginn der Programmausführung für jede Objektart eine Liste mit einer ausreichenden Anzahl dieser Objekte an. Wird ein Objekt benötigt, nimmt man das erste Objekt aus der entsprechenden Liste und initialisiert es neu. Wenn es nicht mehr benötigt wird, hängt man es wieder in die Liste ein. Durch solche *Free Lists* ist es nicht nötig, nach Beendigung der Startphase neue Objekte zu erzeugen, und die Garbage Collection ist unnötig. Allerdings muss man sich selbst um die Speicherverwaltung kümmern. Das ist ein großer Aufwand, und man macht dabei leicht Fehler.

Für fast alle Programmieraufgaben ist es am besten, die Speicherverwaltung dem System zu überlassen und nur unterstützend durch Zuweisung von `null` an nicht mehr zugegriffene Variablen einzugreifen.

6.1.2 Dateien und Co

Freier Platz auf Festplatten wird von einem *Dateisystem* automatisch verwaltet. Ähnlich wie bei der Speicherverwaltung hat man beim Programmieren mit der Verwaltung von Plattenplatz im Normalfall nichts zu tun. Man muss nur gelegentlich nicht mehr benötigte Dateien löschen.

Dennoch bilden Dateien eine beschränkte Ressource, die wir beim Programmieren selbst verwalten müssen. Möglicherweise wollen mehrere Programme gleichzeitig auf dieselbe Datei schreiben, oder ein Programm will auf eine Datei schreiben, während ein anderes von ihr liest. Wenn wir nicht

aufpassen und auf eine korrekte Zugriffsreihenfolge achten, ergibt sich ein Durcheinander, sodass die gelesenen Daten nichts mehr mit dem zu tun haben, was die anderen Programme zu schreiben glauben.

Wie wir im Beispiel in Abschnitt 5.5.3 gesehen haben, werden alle Dateien zuerst geöffnet, dann gelesen oder geschrieben und am Ende wieder geschlossen. Diese Vorgehensweise hat sich bewährt, um ein allzugroßes Durcheinander zu vermeiden. Allerdings müssen wir sorgfältig darauf achten, dass auch tatsächlich alle Dateien am Ende wieder geschlossen sind.

In diesem Abschnitt geht es um Ein- und Ausgabe ganz allgemein, da Dateien nur einen Spezialfall der Ein- und Ausgabe darstellen. Auf ein Terminal wird beispielsweise genau so zugegriffen wie auf eine Datei. Eine ganze Reihe von Klassen kümmert sich um die Ein- und Ausgabe:

File: Instanzen dieser Klasse repräsentieren Dateien und erlauben beispielsweise die Feststellung der Dateiart und -größe sowie das Umbenennen. Schreib- und Lesezugriffe werden jedoch nicht unterstützt.

InputStream und OutputStream: Unter einem *Stream* versteht man einen Datenstrom, von dem immer wieder neue Daten gelesen bzw. auf den stets neue Daten geschrieben werden können. Es hängt von der Art des Streams ab, woher die Daten kommen bzw. wohin sie gehen. Instanzen von `InputStream` und `OutputStream` operieren auf rohen Bytes und interpretieren diese nicht. Diese beiden Klassen haben zahlreiche Unterklassen mit unterschiedlichen Eigenschaften. Beispiele sind `FileInputStream` und `FileOutputStream` für die ungepufferte Ein- und Ausgabe sowie `BufferedInputStream` und `BufferedOutputStream` für die gepufferte Ein- und Ausgabe. Die Klasse `PrintStream` erweitert `OutputStream` um Methoden zur Ausgabe von Daten unterschiedlicher Arten in einem Binärformat.

Reader und Writer: Instanzen dieser beiden Klassen sind Streams von Zeichen, nicht von Bytes. Zu den zahlreichen Unterklassen gehören beispielsweise `FileReader` und `FileWriter` (ungepuffert) sowie `BufferedReader` und `BufferedWriter` (gepuffert). Als Brücken zwischen Zeichen- und Byte-Streams fungieren die Klassen `InputStreamReader` und `OutputStreamWriter`. Die Klasse `PrintWriter` erweitert `Writer` um Methoden zur Umwandlung von Daten anderer Arten in für Menschen lesbare Zeichenketten mit anschließender Ausgabe.

Scanner: Wie wir bereits in Abschnitt 1.1 gesehen haben, erlauben Instanzen dieser Klasse das Einlesen von Daten unterschiedlicher Arten. Das erwartete Datenformat entspricht dem, das beim Ausgeben über `PrintWriter` erzeugt wird, abgesehen von einfachen Zeichenketten nicht dem von `PrintStream` erzeugten Binärformat.

System und Console: Die Klasse `System` stellt durch statische Variablen und Methoden eine Verbindung zum Betriebssystem her. Über die Variablen `in` (vom Typ `InputStream`) sowie `out` und `err` (vom Typ `PrintStream`), die der Standardein- und -ausgabe und der Fehlerkonsole entsprechen, werden häufig einfache Ein- und Ausgaben realisiert. Aufrufe von `System.console()` liefern die einzige Instanz von `Console`, welche die Möglichkeiten der Standardein- und -ausgabe erweitert, vor allem für Passwortabfragen.

Listing 6.1 erweitert das Beispiel aus Listing 5.8 um eine zweite Ausgabedatei. Abgesehen davon, dass die zu Beginn jeder Zeile ausgegebene Zeilennummer in einer Datei nur vier statt sechs Ziffern umfassen kann, wird in beide Ausgabedateien dasselbe geschrieben. Ein weiterer kleiner Unterschied zu Listing 5.8 ist das zusätzliche Argument im Konstruktor für `FileWriter`: Der Wahrheitswert besagt, ob die geschriebenen Daten an eine bereits bestehende Datei angehängt werden sollen. Ist dieser Wert `false` (oder gar nicht angegeben), so wird der Inhalt einer bereits bestehende Datei desselben Namens beim Öffnen gelöscht.

Mit diesem Beispiel demonstrieren wir, was passiert, wenn mehrfach auf dieselbe Datei geschrieben wird. Dazu rufen wir das Programm über den Befehl „`java Numbered2 a b b`“ auf, wobei `a` eine längere Textdatei sein soll. An die Datei `b` wird der Inhalt von `a` zwei mal (mit Nummern vor jeder Zeile) angehängt, allerdings vermutlich nicht ganz so wie erwartet: Zuerst wird ein mehrere Zeilen umfassender Textblock entsprechend `out1` angehängt, dann ein ebenso langer entsprechend `out2`, dann wieder einer entsprechend `out1`, und so weiter. An den Grenzen zwischen diesen Blöcken steht nicht unbedingt ein Zeilenumbruch. Dieses Ergebnis erhalten wir, weil bei der gepufferten Ausgabe ein interner Puffer einer bestimmten Größe immer wieder befüllt und dann in einem Schritt in die Datei geschrieben wird. Die Größe der Blöcke entspricht der Puffergröße. Wenn wir wollen, dass der Inhalt der Datei `b` eine Zeile entsprechend `out1` enthält, dann eine Zeile entsprechend `out2` und so weiter, so müssen wir dafür sorgen, dass der Pufferinhalt nach jeder aus-

Listing 6.1: Klasse zum Testen der mehrfachen Verwendung einer Datei

```

1 import java.io.*;
2 public class Numbered2 {
3     public static void main(String[] args) {
4         if (args.length != 3) {
5             System.err.println("Usage: java Numbered in out1 out2");
6             return;
7         }
8         try {
9             BufferedReader in = null;
10            BufferedWriter out1 = null;
11            BufferedWriter out2 = null;
12            try {
13                String line;
14                in = new BufferedReader(new FileReader(args[0]));
15                out = new BufferedWriter(new FileWriter(args[1], true));
16                out = new BufferedWriter(new FileWriter(args[2], true));
17                for (int i = 1; (line = in.readLine()) != null; i++) {
18                    out1.write(String.format("%6d: %s", i, line));
19                    out1.newLine();
20                    out2.write(String.format("%4d: %s", i, line));
21                    out2.newLine();
22                }
23            }
24            finally {
25                if (in != null)
26                    in.close();
27                if (out1 != null)
28                    out1.close();
29                if (out2 != null)
30                    out2.close();
31            }
32        }
33        catch (IOException ex) {
34            System.err.println("I/O Error: " + ex.getMessage());
35        }
36    }
37 }

```

gegebenen Zeile in die Datei geschrieben wird. Das erreichen wir durch Aufruf von `out1.flush()` unmittelbar nach `out1.newLine()` und `out2.flush()` nach `out2.newLine()`. Mit dieser Änderung erhalten wir das erwartete Ergebnis. Auch eine ungepufferte statt einer gepufferten Ausgabe liefert dieses Ergebnis.

Etwas anders sieht das Ergebnis aus, wenn wir das zweite Argument von `FileWriter` weglassen: Wir erhalten die Ausgabe entsprechend `out2` gefolgt von einem Ende der Ausgabe von `out1`. Die Erklärung ist einfach: Beim Öffnen wird ein möglicherweise schon vorhandener Inhalt gelöscht. Unabhängig davon, ob nach jeder Zeile `flush` aufgerufen wird oder nicht, wird zuerst immer etwas entsprechend `out1` geschrieben, dann entsprechend `out2`. Da nicht an die existierende Datei angehängt wird, sondern an das, was bereits über denselben Stream geschrieben wurde, überschreibt die Ausgabe von `out2` das, was vorher von `out1` ausgegeben wurde. Das Ende der Ausgabe von `out1` bleibt sichtbar, weil über `out1` insgesamt mehr Zeichen ausgegeben werden als über `out2`.

Bei Aufruf des Programms mit drei gleichen Argumenten, also etwa „java Numbered2 a a a“, erhalten wir Folgendes: Falls das zweite Argument von `FileWriter` weggelassen wird, erhalten wir eine leere Datei, weil beim Öffnen der Inhalt gelöscht wird, sodass danach auch nichts mehr gelesen werden kann. So wie in Listing 6.1 ergibt sich jedoch eine Endlosschleife: Die Datei wird stest um neue Zeilen erweitert, die dann wieder gelesen werden und neue Zeilen generieren.

All diese Varianten von unerwartetem Verhalten bekommen wir, obwohl das Programm keinen erkennbaren Fehler enthält. Es handelt sich nur um das normale Verhalten von Ein- und Ausgabe in ungewöhnlichen Situationen. Um solches Verhalten zu vermeiden, müssen wir die Ursachen dafür vermeiden. Beispielsweise können wir zu Beginn des Programms sicherstellen, dass alle drei Argumente verschieden sind.

Folgende Fallen tauchen bei der Ein- und Ausgabe immer wieder auf:

- In unüblichen Programmpfaden, beispielsweise nach dem Werfen einer Ausnahme, wird auf das Schließen oder Hinausschreiben vergessen. Wir haben bereits in Abschnitt 5.5.3 gesehen, wie man solche Fälle richtig handhabt. Am besten funktionieren `finally`-Blöcke für das Freigeben von Ressourcen in lokalen Variablen.
- Das Freigeben von Ressourcen ist unmöglich, sobald das Objekt, das eine Objektvariable mit der Ressource enthält, nicht mehr zugreifbar ist. Unzugreifbare Objekte werden durch Garbage Collection entsorgt. Es sollte nicht passieren, dass unzugreifbare Objekte neben Speicher auch andere Ressourcen wie Dateien blockieren. Die Methode `finalize` (siehe Abschnitt 6.1.1) wurde eingeführt, damit auch nicht mehr zugreifbare Objekte die von ihnen belegten Ressourcen freigeben können. Praktisch ist diese Methode aber nur be-

schränkt verwendbar, weil nicht voraussehbar ist, ob und wann sie ausgeführt wird. Abgesehen von einer generell vorsichtigen Programmierung gibt es keine Technik, um diese Gefahr zu vermeiden. Am besten legt man wichtige Ressourcen nicht in Objektvariablen ab.

- Es gibt unterschiedliche Standards, die festlegen, wie Zeichen auf Bytes abgebildet werden. Man nennt sie *Zeichen-Codierungen*. In Java stellen die Klassen `CharsetEncoder` und `CharsetDecoder` die Funktionalität für die Umwandlung von Zeichen in Bytes und umgekehrt bereit. Instanzen von `Charset` sind Zeichenmengen zusammen mit den dazugehörigen Instanzen von `CharsetEncoder` und `CharsetDecoder`. Beim Öffnen einiger Arten von Streams (etwa vom Typ `InputStreamReader`) kann man eine Instanz von `Charset` bzw. `CharsetEncoder` oder `CharsetDecoder` angeben. Gibt man keine Codierung an, so wird ein Default verwendet, der üblicherweise vom Betriebssystem vorgegeben ist. Gelegentlich werden Daten mit einer anderen Codierung gelesen als sie geschrieben wurden. Derartige Fehler äußern sich meist dadurch, dass Sonderzeichen und Umlaute falsch oder gar nicht dargestellt werden. Um solche Probleme gering zu halten, gibt man meist keine eigene Codierung an. Trotzdem können Konflikte aufgrund unterschiedlicher Codierungen auftreten, weil Einstellungen am Betriebssystem falsch sind oder Dateien zwischen Rechnern mit unterschiedlichen Codierungen ausgetauscht werden.
- Wie wir gesehen haben, ergibt sich eine ganze Reihe von möglicherweise unerwünschten Effekten, wenn mehrfach auf dieselbe Datei geschrieben oder gleichzeitig geschrieben und gelesen wird. Solche Effekte können wir ausschließen, wenn eine Datei nur einmal zum Schreiben geöffnet werden darf. Eine einfache Möglichkeit dazu bieten *Lock-Dateien*, das sind Dateien, die beim Öffnen einer anderen Datei angelegt und beim Schließen wieder gelöscht werden. Vor dem Öffnen müssen wir überprüfen, ob bereits eine Lock-Datei existiert. Auf manchen Systemen bietet die Klasse `FileLock` erweiterte Möglichkeiten. Allerdings lösen Lock-Dateien und Ähnliches das Problem nur in einfachen Fällen, weil gleichzeitige Schreibzugriffe manchmal durchaus erwünscht und notwendig sind, beispielsweise beim Schreiben von Logdateien – siehe Abschnitt 5.4.1. Unerwünschte Effekte lassen sich dabei nur durch vorsichtige Programmierung und den Aufruf von `flush()` und den richtigen Stellen vermeiden.

- Durch `PrintStream` geschriebene Binärdaten können durch Instanzen von `Scanner` außer für Zeichenketten nicht mehr eingelesen werden. Daher sollte man zum Schreiben eher `PrintWriter` verwenden. Insbesondere sollte man über die häufig verwendeten Variablen `System.out` und `System.err` nur Zeichenketten ausgeben, da diese Variablen vom Typ `PrintStream` sind.

6.1.3 Antwortzeiten

Eine weitere wichtige beschränkte Ressource ist Rechenzeit. Natürlich wollen wir effiziente Programme schreiben, die möglichst wenig Rechenzeit brauchen und kurze Antwortzeiten haben. Unter der *Antwortzeit* verstehen wir die Zeit, die zwischen der Eingabe von Daten (beispielsweise Drücken der Enter-Taste) und dem Erhalt eines Ergebnisses vergeht. Kurze Antwortzeiten erhöhen die Benutzerfreundlichkeit eines Programms. Erwartete Antwortzeiten hängen stark von der Art der Aufgabensellung ab, angefangen von Sekundenbruchteilen bis zu Stunden oder sogar Tagen. Die Rechenleistung heutiger Computer ist reichlich bemessen, sodass kurze Antwortzeiten ohne allzugroßen Aufwand realisierbar sein sollten.

Die vorsichtige Formulierung deutet es schon an: Man wird immer wieder mit Programmen konfrontiert, von denen man aufgrund ihrer einfachen Aufgabe kurze Antwortzeiten erwartet, die tatsächlich aber deutlich länger brauchen. Gründe dafür können sehr vielfältig sein:

- Möglicherweise sind benötigte andere Ressourcen nicht sofort verfügbar. Wenn Daten aus einer Datenbank gelesen oder aus dem Internet geholt werden, stellt meist nicht die Rechenzeit, sondern die Wartezeit auf die Daten den entscheidenden Faktor für die Verzögerung dar. Man sieht das bei Laufzeitmessungen daran, dass die *real-Zeit* deutlich größer als die *user-Zeit* ist. In diesem Fall bringt es kaum etwas, das Programm hinsichtlich der Rechenzeit zu optimieren. Die einzige sinnvolle Maßnahme zur Verkürzung der Antwortzeiten besteht darin, benötigte Daten gleichzeitig anzufordern, nicht ein Datenelement nach dem anderen. Oft lässt sich das aber nur über nebenläufige Programmierung bewerkstelligen, einem schwierigen und sehr fehleranfälligen Bereich der Programmierung – siehe Abschnitt 6.3.
- Nicht selten macht das Programm viel mehr als erwartet. Ein Grund dafür kann sein, dass die erwartete und tatsächliche Komplexität der

Aufgabe weit auseinander liegen. Es kann aber auch sein, dass das Programm neben der eigentlichen Aufgabe noch ganz andere, versteckte Aufgaben erledigt. So gibt es Programme, die das Benutzerverhalten analysieren und Ergebnisse per Internet an zentrale Sammelstellen schicken. Manches Programm richtet durch die Erledigung versteckter Aufgaben Schaden an. Vielleicht gibt es geheime Informationen wie Passwörter oder Kreditkartendaten weiter, ermöglicht anderen Personen Zugriff auf das System oder verschickt verbotene Massenmails. Gelegentlich erkennt man Schadsoftware an unerwartet langen Antwortzeiten oder an unerklärlichen Netzwerkaktivitäten.

- Die Präsentation der Ergebnisse ist oft sehr aufwendig gestaltet. Man gibt sich nur mehr selten mit einfachen Textausgaben auf einem Terminal zufrieden. Eine graphisch ansprechende Oberfläche mit individuell abgestimmten Bildern und vielleicht dazu passender Musik kann jedoch nicht nur den Entwicklungsaufwand, sondern auch die Antwortzeiten in die Höhe treiben, ohne den Nutzen zu erhöhen.
- Bei der Konstruktion von Programmen liegen die Schwerpunkte oft auf kurzen Entwicklungszeiten und der einfachen Wartung, nicht auf der Laufzeiteffizienz. Das hat Vorteile. Aus diesem Grund werden bewusst einfachere Datenstrukturen und Algorithmen gewählt sowie bereits fertige Programmteile eingebunden, auch wenn sie für den Einsatzzweck nicht optimal sind. Solange die Antwortzeiten trotzdem noch tolerabel sind, ist dagegen nichts einzuwenden. Man tauscht kurze Antwortzeiten gegen kurze Entwicklungszeiten.
- Heute werden oft aufwendige Technologien eingesetzt, die eine einfachere Verwendung und Wartung der Software versprechen. Die Softwareindustrie bietet eine Vielzahl an Technologien an, die alle eine bestimmte Berechtigung haben. Beispiele sind Komponentenmodelle, Webanbindungen und Datenbankanbindungen. Wenn wir mehrere Technologien in unsere Programme einbinden, erhalten wir zwangsläufig viele übereinander liegende Schichten der Softwarearchitektur. Obwohl der Einsatz bewährter Technologien oft vorteilhaft ist, kann ein unüberlegter Technologieeinsatz problematisch sein: Man bindet sich an manchmal sehr kurzlebige Technologien, durch viele übereinander liegende Schichten erhöhen sich die Antwortzeiten, und bei nicht genau auf die Aufgabe passenden Technologien kommen die erhofften Vorteile nicht zum Tragen. Es entsteht die Gefahr der Ab-

hängigkeit von bestimmten Technologien, die uns spezielles Expertenwissen und eine effiziente Softwareentwicklung vorgaukeln.

- Manchmal werden Programme nicht für kurze Antwortzeiten sondern beispielsweise geringen Speicherverbrauch optimiert. Laufzeit und Speicherverbrauch sind in der Regel gegeneinander austauschbare Ressourcen. Beispielsweise kann man mehrfach benötigte Werte immer wieder neu berechnen, oder nur einmal berechnen, zwischenspeichern und bei der nächsten Verwendung wieder laden. Welche Variante günstiger ist, hängt von vielen Faktoren ab. Das Zwischenspeichern und Suchen nach gespeicherten Werten kann auch aufwendig sein, möglicherweise aufwendiger als Neuberechnungen.
- Man muss das Gesamtsystem im Auge haben, nicht nur einzelne Anwendungen. Beispielsweise kann man *aktiv* auf Ereignisse wie das Drücken von Tasten warten, indem man in einer Schleife wiederholt den Status der Tastatur abfragt; man spricht von *Busy Waiting*. Alternativ dazu kann man *passiv* warten, indem man eine Methode aufruft, die unterstützt vom Betriebssystem die Ausführung des Programms unterbricht, bis eine Zeile eingegeben ist. Möglicherweise kann man durch Busy Waiting um eine Winzigkeit rascher reagieren, da das Betriebssystem kein unterbrochenes Programm fortsetzen muss. Allerdings zahlt man einen hohen Preis, da das Programm ständig mit Abfragen beschäftigt ist und möglicherweise anderen Programmen die Rechenzeit wegnimmt. So kann man die Antwortzeit eines Programms minimal verkleinern, indem man die Antwortzeiten anderer Programme möglicherweise stark erhöht. Vom Gesamtsystem her betrachtet ist Busy Waiting keine gute Technik.

Wir haben bereits gesehen, dass die Auswahl geeigneter Datenstrukturen und Algorithmen einen großen Einfluss auf Laufzeiten und Antwortzeiten haben kann. Sorgfalt bei der Auswahl ist ratsam. Andererseits haben wir auch gesehen, dass Laufzeitmessungen schwierig sind und leicht zu wertlosen Ergebnissen führen. Diese Erkenntnis ist eine der Grundlagen für folgende Regeln bezüglich der händischen (nicht vom Compiler durchgeführten) Programmoptimierung:

- Wer kein Experte dafür ist, sollte keine Optimierungen machen.
- Wer Experte dafür ist, sollte noch keine Optimierungen machen.

Dass Nichtexperten die Finger von Optimierungen lassen sollten, ist klar: Die Gefahr von Fehlern durch Unkenntnis komplizierter Sonderfälle ist zu groß. Aber auch Experten machen vieles falsch. Durch zu starke Konzentration auf die Laufzeit geht die einfache Wartbarkeit verloren. Optimierungen sind daher nur für Programmteile sinnvoll, die sehr stabil sind und sich wahrscheinlich kaum mehr ändern werden. Wenn man zu früh optimiert, ist der Aufwand vergebens, weil jede später notwendige Programmänderung die Optimierung wahrscheinlich wieder zunichte macht.

Optimierungen sind extrem aufwendig. Niemand wird ein ganzes Programm optimieren, sondern nur jene kleinen Teile, die am meisten Zeit verschlingen, und das auch nur dann, wenn Antwortzeiten zu lang sind.

Um ihre Aufgaben erfüllen zu können, benötigen Programme Zugang zu Ressourcen. Man kann ihnen den Zugang zu diesen Ressourcen nicht verweigern, sondern höchstens auf das nötige Maß einschränken. Betriebssysteme bieten Möglichkeiten zur Beschränkung des Zugangs zu Ressourcen. Beispielsweise lässt sich die Zugreifbarkeit von Dateien steuern, eine Obergrenze für den Speicherverbrauch einführen, eine Priorität bei der Vergabe von Rechenzeit an Programme festlegen, die Anzahl offener Netzwerkverbindungen begrenzen, und so weiter. Einerseits sollen Programme Zugang zu den benötigten Ressourcen bekommen, andererseits aber möglicher Schaden durch Schadsoftware begrenzt werden. Zugangsbeschränkungen können leider nur die allerschlimmsten Auswirkungen schädlicher Software reduzieren, keinesfalls alle Schäden vermeiden.

Es gibt zahlreiche Ansätze, um Schäden durch versteckte Aktivitäten von Programmen gering zu halten. Manche Leute setzen nur *Open Source Software* ein, damit der Quellcode von Programmen offengelegt ist und versteckte, möglicherweise schädliche Programmteile direkt im Quellcode gefunden werden können. Das funktioniert leider nur begrenzt, weil niemand viele Millionen Codezeilen überblicken kann. Andere Leute bestehen auf *zertifizierte Software*, bei der eine als seriös angesehene Firma die Software untersucht und bestimmte Eigenschaften garantiert. Leider gibt es sehr unterschiedliche Arten von Zertifikaten, zumeist solche, die nur den Ursprung der Software zertifizieren. Der dazugehörige Vertrag schließt in der Regel jegliche Haftung für Schäden aus. Solche Zertifikate sind relativ wertlos. Häufig werden Virens Scanner eingesetzt, die ein System regelmäßig auf das Vorhandensein von als schädlich bekannter Software untersuchen. Allerdings werden im Wesentlichen nur solche Programme gefunden, deren schädliche Wirkung schon bekannt ist. Ein wirklich sicherer Schutz wird durch keine der vielen möglichen Maßnahmen erreicht.

primitiver Typ	Referenztyp	Anzahl Bits	größte darstellbare Zahl
byte	Byte	8	127
short	Short	16	32.767
int	Integer	32	2.147.483.647
long	Long	64	9.223.372.036.854.775.807
existiert nicht	BigInteger	nach Bedarf	unbeschränkt

Abbildung 6.2: Größe ganzzahliger Typen in Java

6.2 Grenzwerte

Nicht nur im Großen, sondern auch im Kleinen finden wir überall Grenzen und Schranken. So sind Zahlen der Typen `int` und `Integer` in Java auf 32 Bit begrenzt, und Zahlen kleiner als $-2^{31} = -2.147.483.648$ und größer als $2^{31} - 1 = 2.147.483.647$ sind damit nicht darstellbar. Fließkommazahlen haben zwar einen viel größeren Wertebereich, aber die Genauigkeit beim Rechnen mit diesen Zahlen ist begrenzt. Abseits von Zahlen haben auch Datenstrukturen begrenzte Kapazitäten. Ein Array enthält eine vorher bestimmte Anzahl an Elementen, und das Ende einer Liste wird durch `null` dargestellt. Wir müssen in Programmen Vorkehrungen treffen, um mit Situationen umzugehen, bei denen wir auf Grenzen stoßen.

6.2.1 Umgang mit ganzen Zahlen

Computer können sehr schnell und praktisch fehlerfrei rechnen. Ein gewöhnlicher Computer unter dem Schreibtisch schafft einige Milliarden primitiver Rechenoperationen pro Sekunde. Leider hat das auch seinen Preis: Der Computer wurde für hohe Rechenleistung ausgelegt, nicht für die einfache, bequeme und sichere Nutzung der Rechenleistung. Zur Erzielung dieser Leistung muss man einige Unannehmlichkeiten in Kauf nehmen.

Eine der größten Unannehmlichkeiten kommt von der begrenzten Anzahl an Bits für die Zahlendarstellung in den primitiven ganzzahligen Typen `byte`, `short`, `int` und `long`. Abbildung 6.2 stellt diese Typen gegenüber. Wenn die größte darstellbare Zahl n ist, dann ist die kleinste darstellbare Zahl $-(n + 1)$, da 0 bezüglich der Darstellung zu den positiven Zahlen gehört. Auch wenn Instanzen von `long` sehr große Zahlen darstellen können, so kommt es in der Praxis dennoch vor, dass noch größere

Listing 6.3: Programm zur Demonstration eines Überlaufs

```

1 public class OverflowTest {
2     public static void main(String[] args) {
3         System.out.println("50000 * 50000 = " + (50000 * 50000));
4     }
5 } // Ausgabe: "50000 * 50000 = -1794967296" wegen Überlauf!

```

Zahlen gebraucht werden. Für diesen Fall gibt es die Klasse `BigInteger`, deren Instanzen beliebig große Zahlen (sofern der Computer genügend Speicher dafür hat) sein können. Wie jede Klasse ist `BigInteger` ein Referenztyp, und Rechenoperationen auf Instanzen von Referenztypen sind viel langsamer als solche auf Instanzen von primitiven Typen. Es gibt zu jedem primitiven Typ einen Referenztyp, weil manchmal Referenztypen notwendig sind – beispielsweise für Generizität. Aus Effizienzgründen verwenden wir trotzdem immer, wo dies möglich ist, primitive Typen.

Beim Programmieren haben wir die Wahl zwischen effizienten Typen und unbeschränkten Typen, beides zugleich geht aber nicht. In den allermeisten Fällen ist der Wertebereich von `int` oder zumindest `long` bei weitem ausreichend. Das Problem besteht eher darin, dass wir gelegentlich nicht wissen, ob der Wertebereich in Einzelfällen nicht doch zu klein sein könnte. Ein einfaches Programm in Listing 6.3 demonstriert, was passiert, wenn der Wertebereich zu klein wird: Das Ergebnis der Berechnung ist ohne Vorwarnung und ohne geworfener Ausnahme einfach nur falsch. Wir erwarten uns, dass die Multiplikation von 50.000 mit sich selbst 2.500.000.000 ergibt. Dieses Ergebnis ist größer als die größte durch `int` darstellbare Zahl, und es kommt zu einem *Überlauf*. Tatsächlich stimmen die letzten 32 Bit in der Binärdarstellung der Zahlen $-1.794.967.296$ und $2.500.000.000$ überein, aber wir würden mindestens 33 Bit brauchen, um zwischen der negativen und positiven Zahl unterscheiden zu können. Bei einem Überlauf werden alle Bits, für die kein Platz ist, einfach abgeschnitten. Dasselbe gilt für einen *Unterlauf*, bei dem das Ergebnis zu klein ist, um vollständig dargestellt werden zu können.

Die Zahl 50000 in unserem kleinen Programm ist vom Typ `int`, weil alle einfachen Zahlenliterals ohne besondere Kennzeichnung vom Typ `int` sind. Daher ist auch das Ergebnis der Berechnung vom Typ `int`. Den Überlauf in Listing 6.3 können wir leicht vermeiden, indem wir zumindest

eines der beiden Literale durch `50000L` ersetzen. Durch Anhängen von `L` erhalten wir ein Literal vom Typ `long`, und die Multiplikation liefert ein Ergebnis vom Typ `long`, wenn mindestens ein Operand von diesem Typ ist. Tatsächlich haben in der Praxis viele Über- und Unterläufe ihren Ursprung in der Verwendung von `int` wo `long` angebracht wäre. Ein guter Teil davon lässt sich einfach durch Anhängen von `L` an Zahlenliterals vermeiden. Aber auch `long` kann Über- und Unterläufe nicht generell ausschließen. Wir brauchen im Beispiel nur größere Zahlen zu verwenden, um einen Überlauf von `long` zu erhalten.

Wenn wir Instanzen von `BigInteger` verwenden, können keine Über- und Unterläufe passieren. Stattdessen werden bei Bedarf weitere Bits hinzugefügt. Entsprechende Überprüfungen und die aufwendige Verwaltung des Speichers bei nicht fix vorgegebener Objektgröße sind jedoch sehr aufwendig und kosten viel Zeit. Außerdem ist die Erzeugung von Instanzen von `BigInteger` umständlich, weil wir dafür Konstruktoren benötigen, und Berechnungen sind komplizierter hinzuschreiben.

Beim Rechnen mit ganzen Zahlen müssen wir auf eine Reihe möglicher Fallen achten:

- Zur Vermeidung von Über- und Unterläufen müssen wir den Wertebereich so abschätzen, dass es sicher zu keinen Über- und Unterschreitungen kommt. Die häufig geübte Praxis nach dem Motto „nehmen wir `long`, dann wird schon nichts passieren“ bieten keinen ausreichenden Schutz. Wenn der Wertebereich nicht abschätzbar ist, beispielsweise weil die möglichen Werte von Parametern nicht genau genug bekannt sind, dann müssen wir auf `BigInteger` zurückgreifen oder Werte an geeigneten Stellen dynamisch (durch `if`-Anweisungen) überprüfen. Insbesondere müssen wir Daten aus anderen Quellen auf Plausibilität prüfen – siehe Abschnitt 5.6.1.
- Größenabschätzungen von Zahlen sind nicht nur zur Vermeidung von Über- und Unterläufen notwendig. Beispielsweise nehmen wir, wenn wir Zahlen ausgeben, oft eine Obergrenze für die Anzahl der Stellen dieser Zahlen an. So haben wir in der Klasse `Numbered2` in Listing 6.1 nur vier bzw. sechs Stellen für die Zeilennummer vorgesehen. Derartige Einschränkungen kann `BigInteger` nicht umgehen.
- Fließkommazahlen haben einen größeren Wertebereich als ganze Zahlen. Daher kommt immer wieder jemand auf die Idee, eine Fließkommazahl statt einer ganzen Zahl zu verwenden, wenn der Wertebereich

nicht klar genug abschätzbar ist. Allerdings sind Fließkommazahlen kein Ersatz für ganze Zahlen. Ganze Zahlen nimmt man, wenn man fehlerfrei, also ohne Rundungsfehler rechnen muss. Beim Rechnen mit Fließkommazahlen entstehen Rundungsfehler. Zur Berechnung von Näherungswerten sind Fließkommazahlen jedoch gut geeignet.

- Programmänderungen können leicht dazu führen, dass Wertebereiche von Zahlen verändert werden. Die Abschätzungen der Wertebereiche, die man ursprünglich gemacht hat, sind damit hinfällig. Daran muss man bei der Programmänderung denken.
- Ein spezielles Problem ist die Division durch null, deren Ergebnis undefiniert ist. In Java wird bei einer versuchten Division durch null eine `ArithmeticException` geworfen. Unabhängig davon, ob man diese Ausnahme abfängt oder schon vorher prüft, ob der Divisor gleich null ist, muss man für diesen Fall einen eigenen Programmzweig vorsehen. Einfacher ist es, wenn man aus einer statischen Analyse des Programms weiß, dass der Divisor nicht gleich null sein kann. In sehr vielen Fällen ist das möglich, weil Divisionen durch null nicht sinnvoll sind und sich Beziehungen zwischen den Daten ganz natürlich so ergeben.
- Divisionen liefern im Allgemeinen keine ganzzahligen Ergebnisse. Daher wird der Divisionsoperator „/“ auf ganzen Zahlen, der ein in Richtung null gerundetes Ergebnis liefert, von einem Operator „%“ zur Berechnung des Divisionsrestes begleitet. Die Berechnung des Divisionsrestes entspricht der Modulo-Berechnung, wenn beide Operanden nicht negativ sind. Eine mathematische Definition für Modulo auf negativen Zahlen gibt es nicht. Wenn die Möglichkeit besteht, dass ein Operand negativ ist, müssen wir meist spezielle Vorkehrungen treffen, die von Programm zu Programm verschieden sind.
- Der Wertebereich ganzer Zahlen ist nicht vollkommen symmetrisch in positive und negative Zahlen geteilt. Daher kann auch die Negation durch „-“ zu einem Überlauf führen: Die Negation der kleinsten darstellbaren Zahl liefert jeweils wieder die kleinste darstellbare Zahl, nicht die größte darstellbare Zahl. Also liefert `- -2147483648` als Ergebnis wieder `-2147483648`, nicht `2147483648`.
- Für Gleichheitsvergleiche auf Instanzen von Referenztypen müssen wir die Methode `equals` verwenden, für Vergleiche von Instanzen

primitiver Typen gibt es dagegen nur `==`. Der Grund dafür besteht darin, dass `==` die Objektidentität vergleicht, nicht die Gleichheit. Gleichheit und Identität bei primitiven Typen sind nicht unterscheidbar, sodass `equals` dafür weder nötig noch definierbar ist (da primitive Typen keine Untertypen von `Object` sind). Der Vergleich `„new Integer(125) == new Integer(125)“` liefert `false`, weil die zwei Objekte unabhängig voneinander erzeugt wurden. Es macht keinen Unterschied, ob sie denselben Wert haben. Dagegen liefert `„new Integer(125).equals(new Integer(125))“` immer `true`.

- Noch diffiziler sind die Unterschiede zwischen `equals` und `==` bei Verwendung von Autoboxing – siehe Abschnitt 4.5.1. Wenn wir zwei Variablen `x` und `y` vom Typ `Integer` haben, so werden durch die Anweisungen `x=128;` und `y=128;` implizit zwei neue Instanzen von `Integer` erzeugt und an die Variablen zugewiesen. Die Auswertung des Ausdrucks `x==y` liefert danach wie erwartet `false`. Wenn wir stattdessen aber die Zuweisungen `x=127;` und `y=127;` machen, ergibt `x==y` danach `true`. Die Erklärung besteht darin, dass Autoboxing für Zahlen bis 127 keine neuen Objekte erzeugt, sondern bereits vorher bereitgestellte Instanzen von `Integer` verwendet. Bei zwei Verwendungen derselben Zahl wird also dasselbe Objekt eingesetzt. Für größere Zahlen wird diese Technik nicht angewandt, weil sonst zu viele fertige Objekte bereitgestellt werden müssten. Dieses Beispiel zeigt, wie sehr wir auf die Unterscheidung zwischen `equals` und `==` achten müssen. Mit einfachem Ausprobieren ist es nicht getan.

6.2.2 Rundungsfehler

Es ist bekannt, dass sich reelle Zahlen im Allgemeinen nicht mit endlich vielen Nachkommastellen genau darstellen lassen. Das gilt für Dezimalzahlen genauso wie für Binärzahlen. Deshalb kann auch ein Computer reelle Zahlen nicht genau darstellen, und beim Rechnen mit reellen Zahlen müssen wir Rundungsfehler in Kauf nehmen.

In der Programmierung unterscheiden wir hauptsächlich zwei Darstellungsarten für Zahlen mit Nachkommastellen:

Fließkommazahlen: Je nach Typ bestehen diese Zahlen aus einer fixen Anzahl an Ziffern (binär oder dezimal). Das Komma innerhalb der Zahl ist in einem weiten Bereich verschiebbar. Nehmen wir an, eine

Fließkommazahl besteht aus sechs Dezimalziffern. Dann ist 1,23456 genauso darstellbar wie 123,456 und 12345,6, und wir können das Komma auch über den Bereich der sechs Ziffern hinauschieben, wodurch Nullen vorne oder hinten angehängt werden. So sind auch die Zahlen 0,000123456 und 123456000,0 durch denselben Typ und mit derselben Genauigkeit darstellbar. Zur Vereinfachung schreibt man diese Zahlen häufig in der Exponentenschreibweise $1.23456e-4$ und $1.23456e8$ an,¹ also die Zahl 1,23456 multipliziert mit 10^{-4} bzw. 10^8 , oder anders formuliert, das Komma um vier Stellen nach links bzw. acht Stellen nach rechts verschoben. Die Besonderheit von Fließkommazahlen besteht darin, dass die Rundungsfehler von der Größe der Zahl abhängen. In der Nähe von null wird sehr genau gerechnet und auf viele Nachkommastellen gerundet, während die Rundungsfehler zunehmen, je weiter man sich von null entfernt, egal ob in Richtung positiver oder negativer Zahlen. Diese Eigenschaft ist besonders sinnvoll, wenn man physikalische Größen, Wahrscheinlichkeitswerte oder Ähnliches ausdrückt.

Festkommazahlen: Festkommazahlen haben dagegen eine fix festgelegte Anzahl von Stellen nach dem Komma. Dadurch wird über den gesamten Wertebereich hinweg mit demselben Rundungsfehler gerechnet. Manche, aber nicht alle Arten von Festkommazahlen sind dazu geeignet, Geldbeträge auszudrücken und mit Geldbeträgen zu rechnen. Für Geldbeträge gibt es gesetzlich festgelegte Vorschriften, wie gerechnet und gerundet werden muss. Leider sind diese Vorschriften nicht überall auf der Welt ganz gleich.

Für Festkommazahlen kennt Java keine primitiven Typen. Wir müssen auf den Referenztyp `BigDecimal` zurückgreifen. Diese Klasse bietet zahlreiche Möglichkeiten zur Festlegung der Anzahl an Nachkommastellen sowie der Art und Weise, wie gerundet wird. Damit eignen sich Instanzen dieses Typs für das Rechnen mit Geldbeträgen. Der Wertebereich ist wie bei `BigInteger` nicht beschränkt.

Das gesetzeskonforme Rechnen mit Geldbeträgen erfordert viel Spezialwissen. Es ist noch leicht nachvollziehbar, dass normalerweise auf ganze Cent (also zwei Nachkommastellen für Euro-Beträge) genau gerechnet und Beträge ab 0,5 Cent aufgerundet, kleinere Beträge abgerundet wer-

den müssen. Schwierigkeiten bereitet dagegen die Forderung, dass Summen immer genau zu stimmen haben. Wenn beispielsweise 1,00 Euro in drei gleiche Teile geteilt werden soll, so erhalten wir zwei Beträge von 0,33 Euro und einen von 0,34 Euro; drei Beträge von 0,33 Euro sind nicht erlaubt, da wir dabei nur auf eine Summe von 0,99 Euro kommen würden. Hintergründe solcher Schwierigkeiten sind eher juristischer Natur.

Java unterstützt zwei primitive Typen von Fließkommazahlen, `double` und `float`. Normalerweise verwendet man `double`. Die viel ungenaueren Fließkommazahlen vom Typ `float` kommen nur in Spezialfällen zur Anwendung, in denen es auf eine sehr kompakte Darstellung oder die Ausnutzung von Spezialhardware (etwa in Graphik-Prozessoren) ankommt. Mit `double` wird auf ungefähr 15 Dezimalstellen genau gerechnet, mit `float` nur auf etwa 7 Stellen genau. Wertebereiche von $\pm 4,9e-324$ bis $\pm 1,7e+308$ für `double` und $\pm 1,4e-45$ bis $\pm 3,4e+38$ für `float` reichen meist aus. Die beiden Referenztypen `Double` und `Float` entsprechen hinsichtlich der Wertebereiche und Genauigkeiten ihren primitiven Gegenständen. Ein Referenztyp für Fließkommazahlen mit erweitertem Wertebereich und höherer Genauigkeit ist (anders als bei ganzen Zahlen) standardmäßig jedoch nicht vorgesehen. Dafür besteht kaum Bedarf.

Auch das Rechnen mit Fließkommazahlen erfordert viel Spezialwissen. Das Hauptproblem stellt das Runden dar. Manchmal bekommen gerade die Stellen, die durch Runden ungenau sind, große Bedeutung. Konkret passiert das bei der Subtraktion von zwei fast gleich großen Zahlen: Beispielsweise gilt $1,2345678 - 1,2345677 = 0,0000001$, wobei die Differenz jedoch um 8 Dezimalstellen weniger genau ist als die beiden anderen Zahlen. Bei Verwendung von `float` würde das bedeuten, dass wir alle relevanten Stellen verloren haben, und die Differenz nur mehr Rundungsfehler widerspiegelt. Wenn wir mit dieser Zahl weiterrechnen, können wir uns kein Ergebnis erwarten, das auf irgendeine Weise sinnvoll wäre. Dieses Problem nennt man *Auslöschung*. Ähnlich wie bei einem Überlauf bei ganzen Zahlen gibt es weder vom Compiler noch zur Laufzeit einen Hinweis auf die erfolgte Auslöschung, sondern es sind einfach nur die Ergebnisse wenig sinnvoll. Am problematischsten ist natürlich der Fall, dass wir durch vollständige Auslöschung alle sinnvollen Stellen verlieren. Oft verlieren wir nicht alle, sondern nur einige Stellen. Auch dadurch wird das Endergebnis weniger genau. Die Anzahl der Stellen, die im Endergebnis noch sinnvoll sind, hängt hauptsächlich vom gewählten Algorithmus für die Berechnung ab. Man spricht von einem *gut konditionierten* Algorithmus, wenn im Endergebnis viele Stellen sinnvoll sind und von einem *schlecht konditionierten*

¹Im englischsprachigen Raum und in Programmen wird statt Komma „.“ ein Punkt „.“ verwendet.

Algorithmus, wenn nur wenige Stellen sinnvoll sind. Die Verwendung von Additionen und Subtraktionen führt oft zu schlechterer Konditionierung als die von Multiplikationen und Divisionen. Manche Aufgaben beruhen aufgrund ihrer Natur auf bezüglich Auslöschung gefährlichen Additionen und Subtraktionen. Solche schlecht konditionierten Aufgaben sind generell nur mit mehr oder weniger schlecht konditionierten Algorithmen lösbar. Gut konditionierte Aufgaben sind dagegen sowohl mit gut als auch schlecht konditionierten Algorithmen lösbar. Wir müssen also stets darauf achten, einen möglichst gut konditionierten Algorithmus zu finden.

Anders als bei ganzen Zahlen ergeben Überläufe von Fließkommazahlen nicht irgendeine falsche Fließkommazahl, sondern den speziellen Wert *Infinity*, der auch als Konstante `POSITIVE_INFINITY` in `Double` definiert ist. Ebenso ergibt die Division einer positiven Zahl durch `0.0` *Infinity*. Ein Unterlauf ergibt wie die Division einer negativen Zahl durch `0.0` den speziellen Wert *-Infinity* bzw. `Double.NEGATIVE_INFINITY`. Mit diesen speziellen Werten kann man auch rechnen. So ergibt die Division von *Infinity* durch `-2.0` den Wert *-Infinity*. Manche derartige Berechnungen ergeben jedoch keinen Sinn. Beispielsweise ist völlig unklar, welchen Wert die Division von *Infinity* durch *Infinity* ergeben soll. In solchen Fällen bekommen wir den speziellen Wert *NaN* (*Not a Number*) bzw. `Double.NaN`. Alle Operationen ergeben *Infinity*, *-Infinity* oder *NaN* wenn ein Operand *Infinity* oder *-Infinity* ist, und alle Operationen ergeben *NaN* wenn ein Operand *NaN* ist. Ein weiterer Unterschied zu ganzen Zahlen besteht darin, dass zwischen `0.0` und `-0.0` unterschieden wird. Diese beiden Fließkommazahlen stehen ja nicht genau für den Wert null, sondern für beliebige positive bzw. negative Zahlen, deren Absolutwerte kleiner als die kleinste darstellbare Zahl ist, genauso wie *Infinity* für beliebige Zahlen größer der größten darstellbaren Zahl steht.

Folgende Fallen lauern im Bereich der Fließkommazahlen:

- Wenn uns nicht bewusst ist, wie viel Genauigkeit wir durch einen schlecht konditionierten Algorithmus verlieren, nehmen wir meist eine viel zu große Genauigkeit der Ergebnisse an. Die Verwendung von `double` statt `float` kann die Genauigkeit zwar erhöhen, aber nur um wenige Stellen. Ein Algorithmus, der bei Verwendung von `float` zur vollständigen Auslöschung führt, ist häufig auch bei Verwendung von `double` nicht viel besser.
- Aufgrund von Rundungsfehlern ist es nicht sinnvoll, Fließkommazahlen mittels `==` zu vergleichen. Meist wollen wir Zahlen als gleich be-

trachten, auch wenn sie sich um einen kleinen Betrag unterscheiden. Statt `„x == y“` schreiben wir eher `„Math.abs(x - y) < eps“`, wobei die statische Methode `abs` aus der Klasse `Math` den Absolutwert berechnet und die Variable `eps` den maximal an dieser Stelle erwarteten Rundungsfehler enthält. Üblicherweise bezeichnet man Rundungsfehler durch den griechischen Buchstaben ε (Epsilon). Der Wert von `eps` sollte sowohl von den erwarteten Wertebereichen als auch den Anzahlen der zuverlässigen Stellen von `x` und `y` abhängen.

- Eine spezielle Form der Rundung ist die *Absorption*: Addiert man beispielsweise `1e10` mit `1e-10`, so ist das Ergebnis wieder `1e10`, weil die kleine Zahl völlig in den Rundungsfehlern untergeht. Normalerweise ist eine Absorption kein großes Problem. Werden jedoch viele kleine Zahlen zu einer großen addiert, kann sich die Rundung auswirken, da die Summe der kleinen Zahlen im Vergleich zur großen nicht vernachlässigbar ist. Es kommt etwas anderes heraus, wenn man die Summe der kleinen Zahlen zur großen Zahl addiert als wenn man jede kleine Zahl einzeln zur großen addiert. Übliche Rechengesetze wie Assoziativ- und Distributivgesetz gelten nicht.
- So sinnvoll die Verwendung spezieller Werte wie *Infinity* und *NaN* durch Experten in der Praxis ist, so schwierig ist der Umgang mit ihnen durch Nichtexperten. Man muss stets damit rechnen, dass ein Ergebnis ein spezieller Wert ist. Besonderheiten treten bei Vergleichen mittels `==` auf: Der Vergleich von `0.0` und `-0.0` ergibt immer `true` (obwohl zwischen diesen beiden Werten unterschieden wird), während der Vergleich von *NaN* mit *NaN* immer `false` ergibt. Um festzustellen, ob eine Zahl `x` den Wert *NaN* hat, verwenden wir die spezielle Methode `Double.isNaN(x)` (oder schlicht `x==x`, weil dieser Vergleich außer für *NaN* immer `true` ergibt). Für die Methode `equals` in `Double` gelten diese Besonderheiten nicht; das erleichtert beispielsweise das Einfügen von Fließkommazahlen in generische Hashtabellen. Bei genauerer Betrachtung sind diese Besonderheiten sinnvoll. Aber wenn man wenig Erfahrung im Umgang mit Fließkommazahlen hat, erscheinen einem die Besonderheiten als Fallen.

6.2.3 Null

Jede Variable, deren Typ ein Referenztyp ist, kann statt einer Instanz dieses Typs `null` enthalten. Wie wir in Kapitel 4 gesehen haben, brauchen

wir `null` (oder etwas Vergleichbares) als Basis rekursiver Datenstrukturen. Normalerweise markiert `null` das Ende einer rekursiven Datenstruktur, beispielsweise das Ende einer Liste, einen nicht vorhandenen Zweig eines Baumes oder einen leeren Eintrag in einer Hashtabelle. Oft verwenden wir `null` auch unabhängig von rekursiven Datenstrukturen. Beispielsweise kann `null` in einem formalen Parameter bedeuten, dass statt eines übergebenen Arguments ein von der Methode bestimmter Defaultwert² verwendet werden soll.

Leider beherbergt der Umgang mit `null` einige Unannehmlichkeiten und Fallen. Insbesondere müssen wir Fallunterscheidungen machen, bevor wir eine Nachricht an den Inhalt einer Variablen schicken, der möglicherweise `null` ist. Das heißt, statt einer einfachen Anweisung `x.foo()`; müssen wir eine viel kompliziertere bedingte Anweisung verwenden:

```
if (x != null) {
    x.foo();
} else {
    ... mache etwas anderes ...
}
```

Wir haben nicht nur einen erhöhten Schreibaufwand, sondern wir müssen uns auch überlegen, was im `else`-Zweig zu tun ist. Aus der falschen Annahme, dass `x` nicht `null` sein kann, oder schlicht aus Unachtsamkeit verwenden wir gar nicht so selten eine einfache Anweisung, obwohl eine bedingte Anweisung notwendig wäre. Zur Laufzeit wird in solchen Fällen eine `NullPointerException` geworfen. Nicht umsonst zählen solche Ausnahmen zu den häufigsten Ursachen für einen unvorhergesehenen, fehlerhaften Programmabbruch.

Gelegentlich findet man Programmcode, in dem zur Vermeidung bedingter Anweisungen Nachrichten ganz bewusst an Inhalte von Variablen geschickt werden, obwohl die Variablen `null` enthalten können. Entsprechende Ausnahmen werden in einem eigens dafür vorgesehenen `catch`-Block abgefangen. Von einer solchen Vorgehensweise ist aber dringend abzuraten: Einerseits weiß man meist nicht sicher, sondern vermutet nur, wo genau die Ausnahme geworfen wurde. Solche Annahmen können falsch sein, sodass zur Ausnahmebehandlung gänzlich ungeeigneter Code ausgeführt wird. Andererseits ist jede Ausnahmebehandlung aus semantischer

Sicht hochgradig komplex und damit fehleranfällig. Gerade im Zusammenhang mit dem Umgang mit `null` können wir uns diese unnötige Komplexität leicht ersparen.

Es ist sehr wichtig, dass man durch Zusicherungen (vor allem an Schnittstellen, also in Vor- und Nachbedingungen) klar macht, welche Variablen und formale Parameter `null` enthalten und welche Methoden `null` zurückgeben können und welche nicht. Wenn `null` erlaubt ist, muss natürlich auch geklärt werden, wofür `null` steht. Nur so entsteht ein Vertrag zwischen einem Client und einem Server, bei dem beide Vertragspartner wissen, was der jeweils andere von ihnen erwartet. Java selbst bietet dafür derzeit leider kaum brauchbare Unterstützung. Es gibt schon seit einiger Zeit Überlegungen, das Typsystem von Java dahingehend zu erweitern, dass man im Typ ausdrücken kann, ob `null` erlaubt ist oder nicht. Technische Möglichkeiten zur statischen Überprüfung dieser Eigenschaft durch den Compiler wären vorhanden. Jedoch würde die erweiterte Typinformation das Problem nur zum Teil lösen. Man wüsste zwar, wo `null` erlaubt ist, aber nicht, wofür `null` steht. Auch könnte man nicht mit Bedingungen umgehen, die klarer beschreiben, in welchen Situationen `null` erlaubt ist und in welchen nicht.

Betrachten wir als Beispiel einen Iterator, etwa `ListIter<A>` aus Listing 4.30. Wenn ein Aufruf von `next()` kein weiteres Element im Aggregat zurückgeben kann weil keines mehr vorhanden ist, wird `null` zurückgegeben. Allerdings wird `null` auch dann zurückgegeben, wenn das Aggregat `null` als Element enthält. Aus dem Ergebnis `null` von `next()` dürfen wir nicht schließen, dass der Iterator keine weiteren Elemente zurückgibt, da `null` in zwei ganz unterschiedlichen Bedeutungen vorkommen kann. Zur Unterscheidung benötigen wir die Methode `hasNext()`.

Wir wollen vermeiden, dass `null` in mehreren Bedeutungen vorkommt. Das gelingt aber nicht immer. In diesen Fällen müssen wir, wie im Iterator, andere Unterscheidungsmöglichkeiten vorsehen. Am besten weisen wir auf diesen Umstand in den Zusicherungen ganz klar hin, damit Aufrufer keine falschen Annahmen treffen.

Der Bedarf an einer Verwendung von `null` ergibt sich sehr oft von alleine, beispielsweise weil wir rekursive Datenstrukturen aufbauen müssen oder in manchen Situationen kein passendes Objekt haben, um es als Argument zu übergeben oder als Ergebnis zurückzugeben. Ohne dringende Notwendigkeit sollten wir die Verwendung von `null` jedoch vermeiden.

Listing 6.4 zeigt einen typischen Fall einer vermeidbaren Verwendung

²Unter einem *Defaultwert* oder kurz *Default* versteht man ganz allgemein einen Wert, der dann zu verwenden ist, wenn kein anderer Wert explizit vorgegeben ist.

Listing 6.4: Beispiel zur Verwendung von null

```

public interface Motor { double gCO2proKm(); }

public class Benzinmotor implements Motor {
    private double literPro100km;    // Verbrauch in l/100km
    public int double gCO2proKm() {
        return literPro100km * 23.7; // Umrechnung in Gramm CO2/km
    }
    ...                               // Konstruktor, etc.
}

public class Fahrzeug1 {
    private Motor motor;             // null für Fahrrad, etc.
    public double gCO2proKm() {
        if (motor != null) {         // Bedingte Anweisung !
            return motor.gCO2proKm();
        } else {
            return 0.0;
        }
    }
    ...
}

```

von null. Instanzen der Klasse `Fahrzeug1` stellen Fahrzeuge aller Art dar, deren Kraftstoffverbrauch vom `Motor` bestimmt wird. Für Fahrzeuge ohne `Motor`, z.B. Fahrräder, enthält die Variable `motor` den Wert `null`. Daher brauchen wir bedingte Anweisungen, wenn wir Nachrichten an `motor` schicken. Wenn wir an vielen Stellen auf `motor` zugreifen, ergibt sich durch den Sonderfall von Fahrzeugen ohne `Motor` eine deutlich höhere Komplexität. Wie Listing 6.5 zeigt, können wir die Komplexität reduzieren indem wir den Sonderfall vermeiden. Wir brauchen nur eine zusätzliche Klasse `KeinMotor`, deren Instanzen in Fahrzeugen ohne `Motor` verwendet werden, sodass es keinen Grund mehr für `null` in `motor` gibt. Die Unterscheidung zwischen Motorarten (Benzinmotor, Dieselmotor oder auch kein Motor) erfolgt durch dynamisches Binden, nicht durch bedingte Anweisungen. Da wir im Beispiel ohnehin dynamisches Binden brauchen, ist der Ansatz von `Fahrzeug2` in allen Belangen dem von `Fahrzeug1` überlegen – Schreibaufwand, Laufzeiteffizienz, Wartbarkeit, etc.

Eine solche Technik können wir immer verwenden, um `null` zu vermeiden. So könnten wir zwei Arten von Listenknoten unterscheiden, einen

Listing 6.5: Beispiel zur Vermeidung von null (erweitert Listing 6.4)

```

public class KeinMotor implements Motor {
    public int double gCO2proKm() { // motorlose Fahrzeuge haben
        return 0.0;                // keinen Verbrauch
    }
    ...                             // Konstruktor, etc.
}

public class Fahrzeug2 {
    private Motor motor;            // darf nicht null sein
    public double gCO2proKm() {
        return motor.gCO2proKm(); // keine bedingte Anweisung
    }
    ...
}

```

leeren (der als Ersatz für `null` dient) und einen nichtleeren, und ein gemeinsames Interface dafür einführen. Damit bräuchten wir `null` nicht zur Darstellung des Endes einer Liste und ganz allgemein nicht als Basis für rekursive Datenstrukturen. Allerdings wäre eine solche Listenimplementierung der üblichen Implementierung nicht (oder zumindest nicht eindeutig) überlegen. Anders als in obigem Beispiel brauchen wir in der üblichen Listenimplementierung kein dynamisches Binden zur Unterscheidung zwischen unterschiedlichen Arten von Listenknoten und nur eine Listenknotenklasse statt zwei Klassen und einem Interface. Diese Technik würde also ebenso einen Zusatzaufwand nach sich ziehen wie bedingte Anweisungen für `null`. Daher wird häufig `null` verwendet. Der wichtigste Grund dafür ist jedoch schlicht und einfach die Tatsache, dass die Mehrzahl der Programmierer es so gewohnt ist. Es würde auch ohne `null`, dafür mit selbstdefinierten Klassen als Ersatz für `null` ganz gut gehen.

6.2.4 Off-by-one-Fehler und Pufferüberläufe

Viele Informatiker leiden unter einer Krankheit: Wenn man mit ihnen spricht, kommen sie kaum auf das Wesentliche, sondern verrennen sich ständig in Verallgemeinerungen und Nebensächlichkeiten. Glücklicherweise liegt die Ursache dafür nicht darin, dass Informatiker das Wesentliche nicht erkennen. Ganz im Gegenteil. Sie sind es gewohnt, in sehr komplexen Zusammenhängen und auf hohem Abstraktionsniveau zu denken.

Sie sind es jedoch auch gewohnt, sich neben dem Wesentlichen auf Sonderfälle, Randbedingungen und Grenzen zu konzentrieren. Nur auf diese Weise können qualitativ hochwertige Programme entstehen. Der wesentliche Kern ist im Vergleich zum gesamten Algorithmus oder Programm meist nur recht klein. Ein weitaus größerer Teil dient der Initialisierung, der Behandlung von Sonderfällen und Ähnlichem.

Fehler passieren eher bei Initialisierungen, in Abbruchbedingungen und bei der Behandlung von Sonderfällen als in den meist besser durchdachten wesentlichen Teilen. Diese gefährlichen Stellen kann man durchwegs als Grenzen betrachten – als Grenzen zwischen der normalen Ausführung und dessen Beginn, dessen Ende, oder einem alternativen Pfad dazu, dem in manchen Fällen gefolgt werden muss. Die Erfahrung zeigt, dass gerade an diesen Grenzen sehr häufig sogenannte *Off-by-one-Fehler* auftreten, also Fehler, in denen beispielsweise eine Schleife mit einem um eins zu kleinen oder zu großen Index beginnt oder um eins zu früh oder zu spät abbricht. Man kann sich viele mehr oder weniger triviale Gründe für solche Fehler vorstellen – beispielsweise dass man manchmal bei null und manchmal bei eins zu zählen beginnt, dass man einen kleiner- mit einem kleinergleich-Operator verwechselt, und so weiter. Auch wenn man weiß, dass der Wertebereich von 0 bis 100 insgesamt 101 Zahlen umfasst, ist man durch die Magie runder Zahlen immer wieder verwirrt.

Ein nicht ganz so trivialer Grund dürfte ebenso eine wichtige Rolle spielen: Während man zur Lösung des Kerns einer Aufgabe sehr abstrakt denkt, etwa an Beziehungen zwischen Variablen ohne bestimmte Werte, muss man an den Grenzen an ganz konkrete Variablenwerte denken. Der Übergang zwischen abstraktem und konkretem Denken (und umgekehrt) wirkt als Bruchlinie, an der man leicht die Zusammenhänge verliert. Gelegentlich gelingt der Übergang nicht vollständig. Man verharret in abstraktem Denken, und als konkrete Werte an den Grenzen nimmt man einfach jene Werte, die man aus ähnlichen (aber nicht gleichen) Situationen noch in Erinnerung hat. Die können natürlich falsch sein, liegen aber oft in der Nähe der richtigen Werte.

In Kapitel 5 haben wir schon viele über Qualitätssicherung erfahren. All das gilt natürlich auch und insbesondere für Grenzen. Zur Vermeidung von Fehlern an Grenzen sollten wir folgendes Beachten:

- In Zusicherungen müssen wir vor allem Bedingungen festhalten, welche die Grenzen klar beschreiben. Beispielsweise haben wir in der Klasse `UnbekannteZahl` in Listing 1.2 festgelegt, dass die unbe-

kannte Zahl zwischen 0 und `grenze-1` liegt, wobei `grenze>0` gilt. Das ist ein typischer Fall der Beschreibung von Grenzen. Den Normalfall brauchen wir kaum zu beschreiben, weil der ohnehin klar ist.

- Code Reviews können Fehler an Grenzen recht erfolgreich aufzeigen. Wichtig ist jedoch, dass wir uns wirklich vergewissern, dass die Grenzen passen und den Code nicht nur oberflächlich überfliegen. Ordentlich durchgeführte Code Reviews sind sehr anstrengend. Man kann sich kaum länger als etwa 20 Minuten ohne Unterbrechung so gut auf den Code konzentrieren, dass dabei Fehler auffallen.
- Glücklicherweise fallen viele Off-by-one-Fehler beim Testen gleich auf, aber leider nicht alle. Wir sollten auf Testfälle achten, die Randbedingungen, Sonderfälle und Grenzen überprüfen. Oft machen solche Testfälle den weitaus überwiegenden Anteil an Testfällen aus.
- Gerade Off-by-one-Fehler verführen leicht dazu, dass man einen beim Testen als falsch erkannten Wert gleich nach oben oder unten korrigiert, ohne der Ursache des Fehlers vorher genau auf den Grund gegangen zu sein. So etwas kann weitere Fehler nach sich ziehen.
- Auch Grenzen sollten wir statisch verstehen, nicht nur durch Verfolgen des dynamischen Programmablaufs. Weil die Grenzen häufig viel linearer (ohne Schleifen) und mit Konstanten übersät sind, ist der dynamische Ablauf meist leichter nachvollziehbar als im Kern. Es ist jedoch wichtig, dass man die Gesamtheit statisch versteht, da sonst der oben erwähnte Übergang an der Bruchlinie zwischen abstraktem und konkretem Denken noch schwieriger wird.
- Um Termination garantieren zu können, muss man die Grenzen beachten. Überprüfungen der Termination führen fast automatisch dazu, dass man die Grenzen statisch versteht. Nicht zuletzt aus diesem Grund sollte man sich der Termination wirklich vergewissern.
- Das Aufräumen nach geworfenen Ausnahmen passiert meist an Grenzen. Hierbei ist besondere Vorsicht nötig, weil der genaue Programmzustand in der Regel unbekannt ist.
- Plausibilitätsprüfungen für Daten, die aus der Umgebung kommen, werden oft an Grenzen durchgeführt und beeinflussen die Grenzen. Sie dürfen keinesfalls vernachlässigt werden, auch aus Gründen der Sicherheit vor Angriffen (siehe unten).

Die Programmkonstruktion ist eine intellektuell anstrengende Tätigkeit, die noch dazu fast immer unter Zeitdruck erfolgen muss. Unter diesen Bedingungen ist es durchaus verständlich, dass man versucht, sich auf das Wesentliche zu konzentrieren und Nebensächlichkeiten eher beiseite zu lassen, so wie es die meisten Menschen machen. Genau deswegen passieren aber so viele Fehler an Grenzen. Erfahrene Softwareentwickler sehen die Grenzen nicht als Nebensache und ersparen sich deswegen viel Zeit für das Debuggen. Man braucht große Erfahrung um zu verstehen, welche Aspekte der Softwareentwicklung wesentlich und welche nebensächlich sind. Die Wichtigkeit der Grenzen sollte man keinesfalls unterschätzen.

Falsch angenommene Grenzen bei Zugriffen auf Speicherbereiche (beispielsweise Arrays) stellen ein ganz besonders schwerwiegendes Problem dar. Es könnte fälschlicherweise auf etwas zugegriffen werden, was gar nicht mehr zum Speicherbereich gehört – etwa den tausendsten Eintrag in einem Array, obwohl das Array nur Platz für zehn Einträge hat. Bei der Entwicklung von Java wurde viel unternommen, damit so etwas nicht passieren kann. Beispielsweise wird beim Versuch, außerhalb des Indexbereichs auf ein Array zuzugreifen, eine Ausnahme geworfen. Aber auch der Java-Interpreter und das darunter liegende System können Fehler haben und in ganz seltenen Fällen den falschen Zugriff nicht verhindern. Besonders leicht passiert das auf einfacher und daher schlecht abgesicherter Hardware und Betriebssystemunterstützung, heute insbesondere auf Smartphones. Durch schreibende Zugriffe außerhalb des Speicherbereichs wird etwas verändert, das nichts mit dem Array zu tun hat. Dabei können wichtige Daten und vielleicht auch das Programm selbst zerstört werden. Aus diesem Grund müssen wir besonders darauf achten, dass wir niemals auf etwas zugreifen, das außerhalb der Grenzen liegt.

Die wirkliche Gefahr bei Zugriffen außerhalb der Grenzen ist die, dass jemand dadurch die Kontrolle über eine Maschine erlangen kann, der keinen Zugang haben soll. Wenn ein Angreifer einen Fehler kennt, durch den Zugriffe außerhalb eines Speicherbereichs möglich sind, kann er ganz gezielt solche (für übliche Anwendungen sinnlose und daher nicht getestete) Daten in ein Programm füttern, sodass bestimmte Teile des gerade ausgeführten Programms überschrieben werden. Statt der ursprünglich im Programm stehenden Anweisungen werden danach die vom Angreifer eingeschleusten Anweisungen ausgeführt. Auf diese Weise bekommt der Angreifer Zugang zu allem, worauf das Programm zugreifen darf.

Einem Angreifer reicht es, wenn er nur eine einzige Stelle in einem von vielen Programmen kennt, um Zugriff auf die Maschine zu erlangen. Da-

her spielt es keine Rolle, wie klein die Wahrscheinlichkeit für einen solchen Fehler ist. Immerhin muss es im Falle von Java einen Fehler im Java-Interpreter, im Betriebssystem, oder in der Hardware und einen dazu passenden Fehler in einem Programm geben. Bei der riesigen Größe des Interpreters und Betriebssystems sowie der gigantischen Zahl an Java-Programmen wird sich auch bei sehr kleiner Wahrscheinlichkeit irgendwann ein solcher Fehler zeigen. Entsprechende Angriffe finden immer wieder statt. Häufiger sind derartige Angriffe über Programme in anderen Sprachen wie beispielsweise C, in denen Zugriffe außerhalb der Grenzen weniger gut abgesichert sind.

Am häufigsten sind Angriffe über *Pufferüberläufe*. Dabei werden im Programm vom Benutzer eingegebene Daten in einen bestimmten Speicherbereich kopiert. Wenn man nicht genau überprüft, ob die eingegebenen Daten im Speicherbereich Platz haben, kann ein Angreifer entsprechend lange (im Normalfall sinnlose) Daten in das Programm füttern, die dann andere Speicherbereiche, vor allem Teile des Programms überschreiben. Seit vielen Jahrzehnten stellen Pufferüberläufe eine von vielen Angriffsmöglichkeiten dar, um in ein System einzubrechen. Trotz aller Bemühungen ist es bisher nicht gelungen, Programme in dieser Beziehung sicher zu machen. Sicherheit auf der Ebene von Programmiersprachen und Betriebssystemen reicht dafür nicht aus. Auch wir müssen bei der Programmkonstruktion mitspielen, um unseren Programmen keinen Angriffspunkt zu bieten. Daher ist es ganz wichtig, dass wir Daten, die von außerhalb kommen, immer auf Plausibilität prüfen. Vor allem müssen wir sicherstellen, dass die Daten dort, wo sie hinkommen, auch Platz finden.

6.3 Nebenläufigkeit

Die nebenläufige Programmierung erlebt gerade eine Renaissance: Prozessoren in aktuellen Rechnern enthalten zum überwiegenden Teil mehrerer Prozessor-Kerne, aber ohne spezielle Unterstützung durch das Programm werden die Möglichkeiten nicht in vollem Umfang genutzt. Nebenläufige Programmierung ist eine Form einer solchen Unterstützung. Aber leider führt Nebenläufigkeit zu einer viel größeren Programmkomplexität, die ohne spezielle Erfahrung in diesem Bereich nicht zu beherrschen ist. Wir wollen betrachten, was nebenläufige Programmierung ist, wie sie in Java umgesetzt ist, welche Alternativen es dazu gibt und mit welchen Schwierigkeiten man dabei umgehen muss.

6.3.1 Parallelität und Nebenläufigkeit

Zunächst klären wir einige Begriffe, die im Alltag manchmal etwas schlampig verwendet und daher gelegentlich miteinander verwechselt werden:

Parallelität: Darunter versteht man die gleichzeitige (= parallele) Ausführung mehrerer Programme oder Programmteile auf mehreren Prozessoren, Prozessor-Kernen oder Recheneinheiten. Ziel ist die Leistungssteigerung durch optimale Ausnutzung vorhandener Hardware. Der Begriff gibt keine bestimmte Maßnahme oder Technik vor, wie diese Ausnutzung erfolgt. Man unterscheidet häufig feingranuläre (durch mehrere parallele Recheneinheiten pro Prozessor-Kern) von grobkörniger Parallelität (durch mehrere Prozessoren oder Prozessor-Kerne).

Parallelisierung: Das ist die Aufspaltung eines Programms in mehr oder weniger unabhängige Teile, die parallel ausgeführt werden können. Das Ziel ist ausschließlich die optimale Ausnutzung der Hardware. Man unterscheidet die automatische Parallelisierung, bei der ein Compiler die Aufteilung vornimmt, von der manuellen Parallelisierung, die bei der Programmkonstruktion erfolgt. Auf feingranulärer Ebene (einzelne Maschinenbefehle) wird heute fast immer automatisch durch den Compiler und die Hardware parallelisiert – siehe Abschnitt 5.3.3. Auf der grobkörnigen Ebene (größere Programmteile) wird in der Java-Programmierung eher selten parallelisiert, weder automatisch noch manuell.

Nebenläufigkeit: Darunter versteht man die Strukturierung eines Programms auf eine Art und Weise, dass einzelne Teile so miteinander kommunizieren, als ob sie gleichzeitig ausgeführt werden würden. Es spielt keine Rolle, ob sie tatsächlich gleichzeitig ausgeführt werden, oder ob die Gleichzeitigkeit nur simuliert wird. Nebenläufigkeit impliziert also einen bestimmten Programmierstil. Dieser Programmierstil ist beispielsweise gut dafür geeignet, gleichzeitig auf mehrere unabhängige Ereignisse zu warten und rasch auf jedes eingetretene Ereignis zu reagieren, obwohl man im Vorhinein nicht weiß, wann welches Ereignis eintritt – etwa ein Web-Browser, der gleichzeitig auf den Empfang unterschiedlicher Web-Inhalte wartet. Nebenläufigkeit eignet sich zur Parallelisierung auf grobkörniger Ebene. Allerdings ergibt sich Parallelität nicht in jedem Fall, da – wie beim Warten auf mehrere Ereignisse – durch Nebenläufigkeit nicht unbedingt mehrere

Berechnungen gleichzeitig durchgeführt werden können (es wird ja hauptsächlich nur gewartet). Das Ziel der Nebenläufigkeit ist häufig eine Vereinfachung der Softwarestruktur bzw. -architektur, nur gelegentlich die optimale Ausnutzung der Hardware. Nebenläufigkeit ist daher ein allgemeinerer Begriff als Parallelität.

Auf feingranuläre Parallelität und Parallelisierung gehen wir nicht näher ein, da wir uns beim Programmieren kaum darum kümmern brauchen. Parallelität und Nebenläufigkeit werden auf grobkörniger Ebene vor allem durch folgende Techniken unterstützt:

Multiprocessing: Ein *Prozess* (engl. *process*, manchmal auch *task* genannt) ist die Ausführung eines Programms. Beim Multiprocessing können mehrere Prozesse gleichzeitig oder derart überlappt laufen, dass es so aussieht, als ob sie gleichzeitig laufen würden. Jeder Prozess hat seinen eigenen Namensraum, das heißt, Variablen und Objekte, die im einen Prozess existieren, sind von anderen Prozessen aus nicht zu sehen. Daher sind Prozesse ziemlich unabhängig voneinander, abgesehen davon, dass gemeinsame Ressourcen wie Dateien von allen Prozessen zusammen verwendet werden. Diese Ressourcen werden jedoch in jedem Prozess anders angesprochen, beispielsweise über unterschiedliche Streams. Beim Programmieren brauchen wir uns (abgesehen von der Verwaltung gemeinsamer Ressourcen) kaum um Multiprocessing kümmern. Wir können jedoch auch innerhalb eines Programms bzw. Prozesses neue *Prozesse aufspannen* – eine andere Bezeichnung für Programme aufrufen.

Multithreading: Ein *Thread* (deutsch etwa „Faden“) ist ein Ausführungsstrang innerhalb eines Prozesses. Jeder Prozess hat mindestens einen Thread. Man spricht von Multithreading wenn ein Prozess mehrere Threads haben kann, die gleichzeitig oder derart überlappt laufen, dass es so aussieht, als ob sie gleichzeitig laufen würden. Im Unterschied zu Prozessen haben Threads keine eigenen Namensräume, sodass mehrere Threads auf dieselben Variablen und Objekte zugreifen können. Nur lokale Variablen (die innerhalb von Methoden und Konstruktoren deklariert wurden) sind in anderen Threads nicht sichtbar. Beim Programmieren müssen wir daher darauf achten, dass sich Threads beim möglicherweise gleichzeitigen Zugriff auf gemeinsame Variablen und Objekte nicht gegenseitig behindern.

Listing 6.6: Java-Programm mit mehreren Threads

```

class Counter {
    private int x = 0, y = 0;
    public void increment() {
        if (x != y) {
            System.out.println("x = " + x + "; y = " + y);
            x = y = 0;
        }
        x++;
        y++;
    }
}

class Worker implements Runnable {
    private Counter counter;
    public Worker (Counter c) {
        counter = c;
    }
    public void run() {
        for(int i = 0; i < 100000; i++)
            counter.increment();
    }
}

public class TestMultithreading {
    public static final void main (String[] args) {
        for(int i = 0; i < 10; i++) {
            Worker w = new Worker(new Counter());
            new Thread(w).start();
        }
    }
}

```

Multiprocessing im eigentlichen, oben beschriebenen Sinn ist in Java zwar möglich, wird aber absichtlich stark erschwert. Der Grund besteht darin, dass man dabei von Java aus einen Java-Interpreter oder ein anderes Programm außerhalb der Welt von Java startet. Beides ist unerwünscht, weil man dadurch die Schutzmechanismen von Java umgeht und Zugang zu Ressourcen benötigt, auf die man keinen Zugriff haben soll.

Multithreading innerhalb eines Java-Interpreters wird dagegen durch eine Reihe von Maßnahmen unterstützt. Listing 6.6 zeigt ein Beispielprogramm, in dem zehn Threads unabhängig voneinander gleiche Aufga-

ben bearbeiten. Die Methode `main` in `TestMultithreading` erzeugt zehn Instanzen der Klasse `Worker`, jeweils mit einer anderen Instanz von `Counter` als Argument, und zehn Instanzen der vordefinierten Klasse `Thread`, die jeweils einen eigenen Thread darstellen. Das an den Konstruktor von `Thread` übergebene Argument muss – so wie Instanzen von `Worker` – vom Typ `Runnable` sein und die in diesem Interface spezifizierte Methode `run` definieren. Sobald die Nachricht `start` an den Thread geschickt wird, beginnt der Thread zu laufen und die Methode `run` im `Worker` auszuführen. Diese Methode ruft wiederholt `increment` in der Instanz von `Counter` auf. Wenn die Ausführung von `run` zu Ende ist, dann ist auch der entsprechende Thread beendet.

Die Methode `increment` in `Counter` macht etwas Eigenartiges: Bevor die beiden Variablen `x` und `y` erhöht werden, wird überprüft, ob sie ungleiche Werte enthalten und gegebenenfalls eine Meldung ausgegeben und die Variablenwerte auf 0 gesetzt. Das scheint unnötig, denn aus der Betrachtung von `Counter` ergibt sich offensichtlich, dass `x` und `y` niemals ungleiche Werte enthalten können. Ausführungen des Programms bestätigen, dass auch bei sehr vielen Überprüfungen `x` und `y` niemals voneinander verschieden sind.

Eine kleine Änderung des Programms hat schwerwiegende Auswirkungen. Wenn wir den Rumpf von `main` durch folgende Zeilen ersetzen, teilen sich alle `Worker` eine Instanz von `Counter`:

```

Counter counter = new Counter();
for(int i = 0; i < 10; i++) {
    Worker w = new Worker(counter);
    new Thread(w).start();
}

```

Mit dieser Änderung zeigen Programmläufe, dass sich `x` und `y` in dieser einen Instanz sehr wohl voneinander unterscheiden können. Mehrere Programmläufe können Unterschiede zwischen `x` und `y` an ganz unterschiedlichen Stellen entdecken. Die Ursache liegt darin, dass mehrere Threads gleichzeitig `increment` im selben Objekt ausführen und dabei auf die beiden Variablen zugreifen können, sodass beispielsweise `x` mit `y` genau dann verglichen wird, wenn `x` schon verändert wurde, `y` aber noch nicht. Das hat eine ganze Reihe eigenartiger und in der Regel unerwünschter Auswirkungen. In der unveränderten Version von Listing 6.6 passiert das nicht, weil jeder Thread auf einem anderen Objekt operiert.

6.3.2 Race Conditions und Synchronisation

Das, was in obigem Beispiel passiert, ist ein typischer Fall einer *Race Condition*. Das heißt, die Ergebnisse von Berechnungen können davon abhängen, ob ein Thread schneller ist als ein anderer. Mehrere Ausführungen desselben Programms können zu unterschiedlichen Ergebnissen führen, weil der zeitliche Ablauf jeden einzelnen Threads nicht genau genug vorherbestimmt ist. Je nach Situation ist ein Thread manchmal etwas schneller oder langsamer als sonst. Winzigste Unterschiede reichen aus.

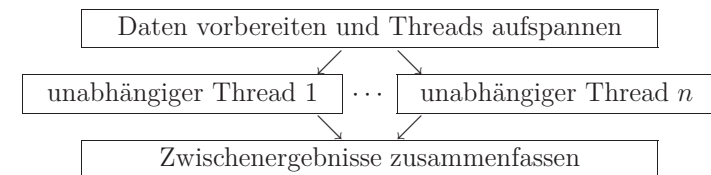
In der abgeänderten Version des Beispiels in Listing 6.6 führen unter anderem folgende zeitliche Abhängigkeiten zu unerwarteten Werten:

- Zwei Threads führen die Anweisung `x++;` oder `y++;` etwa gleichzeitig aus. Diese Anweisungen lesen zuerst den Wert der Variablen, erhöhen ihn, und schreiben dann den neuen Wert in die Variable zurück. Wenn beide Threads gleichzeitig denselben Wert der Variablen lesen, schreiben sie auch denselben (um eins erhöhten) Wert zurück, das heißt, die Variable wird in zwei Ausführungen von `increment` insgesamt nur um eins erhöht. Falls so etwas aufgrund kleinster Zeitunterschiede nur bei einer der beiden Variablen passiert, wird eine Variable um eins und die andere um zwei erhöht.
- Es ist möglich, dass ein Thread nach dem Lesen und vor dem Zurückschreiben eines Variablenwertes vorübergehend unterbrochen wird und danach einen Wert zurückschreibt, der (nach zwischenzeitlicher Ausführung von `increment` durch andere Threads) schon lange nicht mehr aktuell ist.
- Ein Thread kann `x` mit `y` gerade in dem Augenblick vergleichen, in dem ein anderer Thread `x` schon erhöht hat, `y` aber noch nicht.
- Das Ausgeben einer Zeichenkette dauert wesentlich länger als eine normale Ausführung von `increment`. Dadurch ist die Wahrscheinlichkeit hoch, dass mehrere Threads (oft so viele, wie Prozessorkerne vorhanden sind) einen Unterschied zwischen `x` und `y` feststellen bevor die Unterschiede beseitigt werden.

Wenn man mehrere Probeläufe macht und die Ausgaben genau betrachtet, bekommt man ein Gefühl dafür, welche dieser Ursachen bei welchen Werten aufgetreten sind. So lernt man zu verstehen, was durch Nebenläufigkeit alles passieren kann. Auch bei Verwendung von nur einer statt

der zwei Variablen `x` und `y` gibt es Probleme; auch dann entstehen Fehler beim Zählen der Aufrufe von `increment`. Wir verwenden hier die beiden Variablen nur um die Probleme besser sichtbar werden zu lassen.

Der wichtigste Lösungsansatz besteht darin, dafür zu sorgen, dass eine Variable niemals für mehrere Threads gleichzeitig zugreifbar ist. Das unveränderte Programm in Listing 6.6 macht genau das, indem jeder Thread ein anderes Objekt bearbeitet. Wenn man beispielsweise die Anzahl der Aufrufe von `increment` ermitteln will, kann man das auch mit einem eigenen Zähler pro Thread bewerkstelligen und am Ende in nur einem Thread die einzelnen Zählerwerte aufsummieren. Schematisch kann man sich das etwa so vorstellen:



Dabei bezieht sich der Begriff „unabhängig“ darauf, dass die im Thread verwendeten Daten nicht gleichzeitig von anderen Threads verwendet werden. Für die meisten Aufgaben kann man, wenn man sich bemüht, eine Lösung nach diesem Schema finden, wobei die in den unabhängigen Threads zu lösenden Teilaufgaben möglichst umfangreich sein und etwa gleich lange brauchen sollen. Allen anderen Lösungsansätzen, die wir gleich ansprechen werden, sind solche Lösungen vorzuziehen: Sie sind einfach zu verstehen und können die Möglichkeiten paralleler Hardware sehr gut ausnützen.

Nicht immer ist es sinnvoll, die Objekte, auf denen unterschiedliche Threads operieren, voneinander zu trennen. Für solche Fälle bietet Java Möglichkeiten zur *Synchronisation* von Threads. Eine Möglichkeit besteht darin, Methoden, die in mehreren Threads gleichzeitig aufgerufen werden könnten, mit dem Modifier `synchronized` zu definieren:

```
public synchronized void increment() {...}
```

Zur Laufzeit sorgt Java dafür, dass solche Methoden auf demselben Objekt nicht gleichzeitig, sondern nur hintereinander ausgeführt werden können. Wird `increment` von mehreren Threads ungefähr gleichzeitig aufgerufen, so werden die Threads in eine Warteliste gestellt und können mit der Ausführung erst fortfahren, wenn alle Threads vor ihnen die Ausführung der Methode schon beendet haben. So ist garantiert, dass die oben skizzierten Probleme nicht auftreten können.

Wenn man diese Lösung in der modifizierten Variante des Programms in Listing 6.6 einsetzt, kann man kaum mehr von einer parallelen Programmausführung sprechen: Die wesentlichen Programmteile werden alle hintereinander, also *sequentiell* ausgeführt. Wahrscheinlich dauert die Ausführung dieses Programms deutlich länger als die eines vergleichbaren Programms ohne Nebenläufigkeit, weil auch das Aufspannen der Threads und vor allem die sehr häufig nötige Synchronisation viel Rechenzeit verschlingt. Die Synchronisation über `synchronized` Methoden ist nur sinnvoll, wenn die Ausführung dieser Methoden im Vergleich zum gesamten Programm nur einen ganz kleinen Teil der Rechenzeit benötigt. In diesem Fall ist die Wahrscheinlichkeit dafür, dass mehrere Aufrufe fast gleichzeitig erfolgen, recht gering, und die Laufzeiteinbußen durch die Synchronisation bleiben in einem vertretbaren Rahmen.

Listing 6.7 zeigt eine komplexere Form der Synchronisation, in der ein Thread für längere Zeit warten muss. Neben `increment` gibt es in der Klasse `Counter` eine zweite `synchronized` Methode. Zu jedem Zeitpunkt kann im selben Objekt höchstens eine dieser beiden Methoden ausgeführt werden, wodurch niemals mehrere Threads gleichzeitig auf `x` zugreifen können. Zusätzlich wollen wir, dass ein Thread, der `wow` ausführen möchte, solange wartet, bis `x` mindestens einen Wert von 200.000 hat. Dazu verwenden wir die in `Object` definierten Methoden `wait` und `notifyAll`: Nach einem Aufruf von `wait` werden Ausführungen von `synchronized` Methoden auf dem Objekt vorübergehend wieder möglich, aber der aktuelle Thread muss in der Regel so lange warten, bis er durch die Ausführung von `notify` oder `notifyAll` auf demselben Objekt durch einen anderen Thread wieder aufgeweckt wird. Der Unterschied zwischen `notify` und `notifyAll` besteht nur darin, dass `notify` irgendeinen auf dem Objekt wartenden Thread aufweckt, während `notifyAll` alle aufweckt. Aufgeweckte Threads dürfen natürlich erst dann weitermachen, wenn keine andere `synchronized` Methode auf dem Objekt mehr läuft. Wie in `wow` wird `wait` praktisch immer in einer Schleife aufgerufen, da Threads in Ausnahmefällen auch ohne vorherige Ausführung von `notify` oder `notifyAll` aufgeweckt werden können. Außerdem muss man die Ausnahme `InterruptedException` abfangen, die auftritt, wenn ein wartender Thread abgebrochen wird.

Ziel der Synchronisation sind *atomare Aktionen*. Das bedeutet, dass eine Gruppe von Anweisungen (welche die atomare Aktion bildet) als eine Einheit ausgeführt wird und Objektzustände, die kurzzeitig zwischen der

Listing 6.7: Synchronisation und längeres Warten (Worker aus Listing 6.6)

```
class Counter {
    public static final int BARRIER = 200000;
    private int x = 0, y = 0;
    public synchronized void increment() {
        x++;
        y++;
        if (x == BARRIER)
            notifyAll();
    }
    public synchronized void wow() {
        while (x < BARRIER) {
            try { wait(); }
            catch(InterruptedException ex) { return; }
        }
        System.out.println("Wow! Schon bei " + x + "!");
    }
}

public class TestWaiting {
    public static final void main (String[] args) {
        Counter counter = new Counter();
        for(int i = 0; i < 10; i++) {
            Worker w = new Worker(counter);
            new Thread(w).start();
        }
        for(int i = 0; i < 3; i++)
            counter.wow();
    }
}
```

Ausführung von zwei Anweisungen der Gruppe bestehen, in keinem anderen Thread sichtbar werden. In Listing 6.7 bleibt der Objektzustand, in dem `x` bereits erhöht wurde, `y` aber nicht, allen anderen Threads verborgen. Atomare Aktionen sind das eigentliche Ziel der Synchronisation. Leider ist es in Java gar nicht so einfach, sichere atomare Aktionen zu erreichen. Beispielsweise könnte zwischen der Ausführung von `x++` und `y++` eine Ausnahme geworfen werden und somit der zwischenzeitliche Zustand doch sichtbar werden. Weiters muss man darauf achten, dass durch einen Aufruf von `wait` atomare Aktionen unterbrochen werden. Zustände zum Zeitpunkt des Aufrufs von `wait` werden sichtbar. Trotz Synchronisation ist die nebenläufige Programmierung eine fehleranfällige Angelegenheit.

Synchronisation braucht man in Java für jeden Zugriff auf eine Variable, wenn mehrere Threads auf die Variable zugreifen könnten, also auch dann, wenn die gesamte atomare Aktion nur im einfachen Lesen oder Schreiben einer Variablen besteht. Das ist notwendig, weil das Java-System andernfalls Optimierungen durchführt, durch die ein Thread manchmal nur veraltete Variableninhalte zu sehen bekommt – Variableninhalte, die durch andere Threads schon längst verändert wurden. Für genau diese Form von sehr einfachen atomaren Aktionen gibt es in Java eine andere, einfachere Form der Synchronisation: Lese- und Schreibzugriffe auf Variablen, die mit dem Modifier `volatile` deklariert wurden, werden automatisch als synchronisiert betrachtet. In manchen Fällen ist das sinnvoll, aber größere atomare Aktionen, die Race Conditions verhindern, lassen sich damit kaum durchführen.

6.3.3 Gegenseitige Behinderung

Die Synchronisation nebenläufiger Threads kann zu einer ganzen Reihe von Problemen führen. Synchronisation bedeutet im Wesentlichen, dass ein Thread behindert wird, damit ein anderer Thread seine Aufgabe ungehindert ausführen kann. Die Threads behindern sich also gegenseitig.

Man kann eine Analogie zum Straßenverkehr herstellen, wo Verkehrsregeln einige Verkehrsteilnehmer behindern, damit andere ungehindert vorankommen. Solche Verkehrsregeln müssen wohlüberlegt sein, damit nicht der gesamte Verkehr ins Stocken gerät. Sie müssen gefährliche Situationen vermeiden, aber auch gerecht sein, damit nicht bestimmte Verkehrsteilnehmer ständig benachteiligt werden. Man muss Straßen so planen, dass man in jeder Situation unter Beachtung der Verkehrsregeln irgendwann – möglicherweise nach einer bestimmten Wartezeit – wieder weiterkommt.

Auch durch Synchronisation werden gefährliche Situationen vermieden. Man muss nebenläufige Systeme (analog zu den Straßen) so bauen, dass alle Threads irgendwann ihre Aufgaben erfüllen können. Diese Bedingung ist keineswegs automatisch erfüllt. Beispielsweise kann ein Thread alles blockieren, sodass andere Threads nie zum Zug kommen. Es ist auch möglich, dass mehrere Threads sich gegenseitig blockieren, sodass keiner der Threads zum Zug kommt und das gesamte System steht. Wir müssen sicherstellen, dass das System am Leben bleibt und sogenannte *Liveness Properties* erfüllt. Dabei müssen wir folgende Situationen verhindern:

Starvation: Das bedeutet, dass bestimmte Ressourcen, beispielsweise der Zugriff auf eine bestimmte Variable, häufig und für lange Zeit von

bestimmten Threads blockiert wird, sodass andere Threads kaum Zugang zu diesen Ressourcen bekommen. Diese anderen Threads können ihre Aufgaben nicht mehr zeitgerecht erfüllen, sie „verhungern“ also schön langsam. Um Starvation zu vermeiden, darf der Zugang zu Ressourcen nicht für längere Zeit blockiert werden. Dementsprechend dürfen `synchronized` Methoden zur Erledigung ihrer Aufgaben nur kurze Zeit benötigen.

Livelock: Bei einem Livelock versuchen mehrere Threads ständig erfolglos, miteinander in Kontakt zu treten. Beispielsweise wartet Thread *A* darauf, dass Thread *B* einen bestimmten Wert in die Variable *x* schreibt, und Thread *B* wartet darauf, dass Thread *A* einen Wert in *y* schreibt. Statt wirklich zu warten lesen die beiden Threads die Variablen immer wieder in der Hoffnung, dass der andere Thread den Wert in der Zwischenzeit verändert hat. Beide Threads wollen zuerst den erwarteten Wert aus einer Variablen lesen, bevor sie die jeweils andere Variable ändern. Das heißt, die beiden Threads sind am Leben und sehr aktiv, aber in der Berechnung geht nichts weiter weil beide Threads ihre Zeit nur mit aktivem Warten verbringen.

Deadlock: Ähnlich wie bei einem Livelock wartet zum Beispiel ein Thread *A* darauf, dass Thread *B* etwas macht, und Thread *B* wartet darauf, dass *A* etwas macht. Anders als bei einem Livelock sind die Threads jedoch nicht aktiv, sondern hängen in einer Warteliste. Dort bleiben sie ewig hängen. Zur Konkretisierung des Beispiels nehmen wir an, dass *A* gerade eine synchronisierte Methode in einem Objekt *x* ausführt und *B* die entsprechende Methode in einem Objekt *y*. Nun möchte *A* dieselbe Methode auch in *y* und *B* dieselbe Methode in *x* ausführen. Allerdings geht das nicht sofort, weil die Methode in *x* gerade von *A* und jene in *y* gerade von *B* ausgeführt wird, und nicht mehrere Threads gleichzeitig eine synchronisierte Methode ausführen dürfen. Daher werden beide Threads in Wartelisten gehängt und warten darauf, dass die jeweils andere fertig wird. Und wenn sie nicht abgebrochen wurden, warten sie noch heute.

Zur Vermeidung von Starvation sollen synchronisierte Methoden nur kurz laufen. Zur Vermeidung von Livelocks soll man aktives Warten vermeiden, auch um beim Warten keine unnötige Rechenzeit zu verschwenden. Es gibt aber keine einfach zu befolgende Regel zur Vermeidung von Deadlocks. Die möglichen Ursachen von Deadlocks sind dafür zu vielfältig.

Oft geht man einfach so vor, dass man bei der ersten Konstruktion eines Programms zwar peinlich genau auf atomare Aktionen und die Vermeidung von Race Conditions achtet, aber Liveness Properties und insbesondere die Gefahr von Deadlocks nur am Rande berücksichtigt. Dafür legt man bei den Testläufen unter anderem besonderes Augenmerk auf Liveness Properties. Es fällt hoffentlich auf, wenn das System aufgrund solcher Probleme nicht richtig funktioniert. Erst wenn man die Ursachen der Probleme erkannt hat, kann man sie – oft unter sehr hohem Aufwand – beseitigen. Wirklich empfehlenswerte Techniken zur Früherkennung und Vermeidung möglicher Deadlocks gibt es leider nicht. Seit kurzem gibt es auch Werkzeuge, die über formale Techniken (Modell Checking) mögliche Deadlocks in Java-Programmen herausfinden können. Aber diese Werkzeuge sind noch sehr fragil und funktionieren nicht überall.

Leider ist auch das Testen nebenläufiger Programme recht kompliziert. Da kleinste Unterschiede im zeitlichen Verhalten zu ganz anderen Ergebnissen führen können, ist es mit einigen wenigen Testläufen nicht getan. Es sind viele Testläufe auf unterschiedlicher Hardware und unter unterschiedlichen Betriebssystemen nötig.

In letzter Zeit entstehen immer mehr Softwarepakete, Technologien und Spracherweiterungen um auch weniger erfahrenen Personen die nebenläufige Programmierung zu ermöglichen. Tatsächlich sind Vereinfachungen möglich. Das Hauptproblem sind jedoch nicht kleine Mängel in den Programmiersprachen, sondern die hohe Komplexität der nebenläufigen Programmierung an sich. Auch mit den besten Werkzeugen lässt sich diese Komplexität nicht beseitigen. Man benötigt gänzlich neue Berechnungsmodelle um nennenswerte Verbesserungen zu erzielen. Damit verbunden sind neue Programmierparadigmen. Allerdings kann man etablierte Paradigmen nicht von heute auf morgen durch neue ersetzen, da damit viel Wissen und Erfahrung über die Entwicklung guter Software verloren gehen würde. Es wird daher noch lange dauern, bis die nebenläufige Programmierung genauso einfach sein wird wie die sequentielle.

Zusammengefasst kann man nur eine Empfehlung geben: Nebenläufige Programmierung ist nichts für Anfänger – also Hände weg davon.

6.4 Einfachheit und Flexibilität

Es ist klar, dass Programme einfach sein sollen. Natürlich wünschen wir uns auch möglichst viel Flexibilität von den Sprachen und Werkzeugen,

die wir verwenden. In gewisser Weise ist das jedoch ein Widerspruch in sich: Einfache Programme benötigen keine übermäßig große Flexibilität. Tatsächlich sind wir beim Programmieren ständig der Versuchung ausgesetzt, alles als viel komplizierter zu betrachten als es tatsächlich ist. Für eine einfache Lösung braucht man viel Erfahrung und nicht selten auch eine große Portion Mut. In diesem Abschnitt betrachten wir einige Ursachen dafür, dass der Wunsch nach Flexibilität zu einer Falle werden kann.

6.4.1 Strukturierte Programmierung

Die *strukturierte Programmierung* gilt schon seit gut 40 Jahren als eines der wichtigsten Grundprinzipien für die Gestaltung von Programmen und Programmiersprachen. Die Kernaussage dahinter ist die Einfachheit: Alle Programme werden durch wenige einfache Strukturen beschrieben:

- Sequenzen (hintereinander auszuführender Anweisungen)
- Alternativen (durch ein- und mehrfache Verzweigungen)
- Wiederholungen (durch Schleifen oder Rekursion)

Auch moderne Sprachen und aktuelle Programmierparadigmen folgen im Wesentlichen diesem Prinzip. Oft sind diese Strukturen direkt in Form von Kontrollstrukturen verfügbar.

Das Ziel der strukturierten Programmierung besteht darin, Algorithmen und Programme so darzustellen, dass ihr Ablauf für einen Leser einfach zu erfassen ist. Wichtig ist auch, dass die Kontrollstrukturen uneingeschränkt miteinander kombinierbar sein müssen. Für beide Aspekte (einfache Erfassbarkeit und Kombinierbarkeit) ist es wichtig, dass alle Kontrollstrukturen *einen* klar vorgegebenen Anfangspunkt und *einen* genauso klar vorgegebenen Endpunkt besitzen. Die Betonung liegt dabei auf *einen*. Damit wird die Freiheit bei der Programmierung gelegentlich eingeschränkt. Das Prinzip der strukturierten Programmierung sagt sehr deutlich, dass es besser ist, die Einschränkungen in Kauf zu nehmen um eine einfachere Erfassbarkeit und Kombinierbarkeit der Sprachkonstrukte zu erreichen.

Viele typische Fallen haben damit zu tun, dass wir aus Gründen der Flexibilität oder einfach nur um weniger Code schreiben zu müssen die Richtlinie von nur einem Anfangs- und Endpunkt vernachlässigen. Fast alle Sprachen bieten dazu die Gelegenheit. Das Ergebnis sind Programme, die schwer zu lesen sind bzw. bei denen man leicht kleine aber wichtige Feinheiten im Programmablauf übersieht. Mit der Zeit werden dadurch

Fehler in das Programm eingeschleust. Folgende derartige Fallen sind für viele prozedurale und objektorientierte Programme typisch:

Goto: Das ist ein Klassiker und quasi der Gegenpol zur strukturierten Programmierung: In vielen vor allem älteren Programmiersprachen kann man durch den Befehl „`goto x`“ die Ausführung abbrechen und an einer anderen Stelle, die durch `x` bezeichnet ist und innerhalb derselben Prozedur liegt, fortsetzen. Damit kann der Programmfluss beliebig kontrolliert werden. Beispielsweise kann man damit sehr einfach Schleifen realisieren, aber der Phantasie sind kaum Grenzen gesetzt. Die unkontrollierte Verwendung von `goto` macht fast jedes Programm unverständlich.

Man darf `goto` nicht mit einem Prozedur- oder Methodenaufruf verwechseln. Bei einem solchen Aufruf wird eine neue lokale Umgebung aufgebaut, darin der Code der Prozedur oder Methode ausgeführt und am Ende möglicherweise ein Ergebnis an den Aufrufer zurückgegeben. Alle Bedingungen der strukturierten Programmierung sind dabei erfüllt. Ein `goto` baut keine neue Umgebung auf, sondern operiert in der Umgebung der Programmstelle, an die gesprungen wird. Es wird beispielsweise nicht kontrolliert, ob die an dieser Stelle verwendeten Variablen schon initialisiert sind. Es ist auch keine Rückkehr an die Stelle vorgesehen, an der `goto` ausgeführt wurde.

Fall-through: In Listing 2.24 haben wir ein Java-Beispiel dafür gesehen, wie man in einer `switch`-Anweisung Code schreiben kann, der in mehreren Zweigen, also für mehrere `case`-Klauseln ausgeführt wird. Wird die Anweisungssequenz einer `case`-Klausel nicht mit `break` abgeschlossen, fällt man automatisch in den Code für die nächste `case`-Klausel (fall through). Das ist ein Beispiel für einen schlechten Programmierstil, welcher der strukturierten Programmierung klar widerspricht. Man rechnet nicht damit, dass über mehrere Wege an den Beginn des Codes einer `case`-Klausel verzweigt wird, was zu Fehlern führt. Die Probleme ähneln denen von `goto`, sind aber meist nicht ganz so gravierend.

In aktuellen Programmiersprachen gibt es eigentlich keinen Grund mehr für Fall-through. Es ist vielleicht notwendig, die eine oder andere Zeile mehr in den Programmcode zu schreiben. Diese zusätzlichen Zeilen sind jedoch im Hinblick auf die Lesbarkeit und Wartbarkeit notwendig und keineswegs überflüssig. Der Compiler nutzt Optimie-

rungen, um gemeinsame Programmteile zusammenzulegen. Man gewinnt durch Fall-through also auch nichts an Programmeffizienz.

Break: Am Ende jeder `case`-Klausel in einer `switch`-Anweisung ist eine `break`-Anweisung notwendig, um Fall-through zu vermeiden. Allerdings ist `break` auch zum vorzeitigen Ausstieg aus einer Schleife verwendbar. Eine solche Verwendung ist zu vermeiden, da dadurch das Ziel eines einzigen, klar definierten Endpunkts verletzt wird. Manchmal ist zusätzlicher Programmcode notwendig (etwa eine zusätzliche Variable und bedingte Anweisung zum Vergleich des Variableninhalts) um eine solche `break`-Anweisung zu vermeiden. Diesen meist sehr kleinen zusätzlichen Aufwand soll man für eine saubere Programmstruktur in Kauf nehmen.

Continue: In einer Schleife kann `continue` verwendet werden, um noch vor Beendigung einer Iteration für die nächste Iteration an den Anfang der Schleife zurückzuspringen. Die Ähnlichkeit von `continue` mit `goto` fällt sofort auf. Am problematischsten ist die Tatsache, dass Seiteneffekte, die gegen Ende der Schleife passieren sollten, wegen der Beendigung der Iteration nicht passieren – was meist beabsichtigt ist. Aber eine `continue`-Anweisung irgendwo im Schleifenrumpf übersieht man leicht und fügt bei Programmänderungen auch Seiteneffekte, die in jeder Iteration passieren müssen, am Ende des Schleifenrumpfs ein. Sogar wenn man die `continue`-Anweisung bemerkt, sind Programmänderungen schwierig, weil man die Struktur des Schleifenrumpfs dafür meist komplett umschreiben muss. Daher sollte man, wie zur Vermeidung von `break`-Anweisungen in Schleifenrumpfen, zusätzlichen Programmcode für eine saubere Programmstruktur in Kauf nehmen. Die Gefahren von `break` ähneln denen von `continue`, jedoch sind die Auswirkungen von `continue` oft schwerwiegender, weil am Ende eines Schleifenrumpfs oft Vorbereitungen für die nächste Iteration getroffen werden, die nach Ausführung einer `break`-Anweisung keine Rolle spielen.

Return: Mit einer `return`-Anweisung steigt man aus einer Methode aus. Wenn die Methode Ergebnisse zurückgibt, ist ein `return` notwendig. Problematisch ist jedoch die Tatsache, dass man durch `return` die Methode an beliebiger Stelle verlassen kann. Um Unklarheiten zu vermeiden sollte man `return`-Anweisungen nur an solchen Stellen einsetzen, wo man sie sich als Programmierer erwartet. Das ist vor

allem das Ende der Methode. Aber auch am Ende einzelner Zweige von bedingten Anweisungen stehen häufig `return`-Anweisungen. Die Auswirkungen ähneln denen von `break` zum Ausstieg aus einer Schleife, sind aber meist weniger gravierend, da nach Ausführung einer `return`-Anweisung die lokalen Variablen der Methode ohnehin nicht mehr gültig sind. Trotzdem muss man darauf achten, dass vor Ausführung der `return`-Anweisung alle Objektvariablen die gewünschten Werte enthalten.

Ausnahmen: Auch das Werfen einer Ausnahme unterbricht den Kontrollfluss und steht damit ganz klar im Widerspruch zur strukturierten Programmierung. Solange Ausnahmen wirklich nur in seltenen Ausnahmefällen geworfen werden, ist dagegen nichts einzuwenden. Für die normalen Fälle kann das Programm noch immer schön strukturiert sein. Ausnahmen können sogar helfen, die Programmstruktur zu verbessern, indem der Code frei von zahlreichen Überprüfungen und Sonderbehandlungen für seltene Spezialfälle bleibt. Andererseits werden Ausnahmen manchmal nicht nur in Ausnahmefällen geworfen, sondern ganz bewusst zur Umgehung des normalen Kontrollflusses – quasi als `goto`, das auch über mehrere Methodenaufrufe hinweg funktioniert. Davon ist dringend abzuraten. Ein solches Programm ist sehr schwer zu verstehen und zu warten.

Dangling-else: `if`-Anweisungen gibt es in der Form mit und ohne `else`-Zweig. In einer Anweisung der Form `if(a)if(b) C;else D;` ist einem Leser oft nicht klar, ob das `else` sich auf das erste oder zweite `if` bezieht. Das nennt man Dangling-else-Problem. In der Sprachdefinition ist klar geregelt, dass sich das `else` in diesem Fall auf das zweite (innere) `if` bezieht. Ein Leser wird aber oft dadurch in die Irre geleitet, dass die Formatierung des Codes eine andere Strukturierung widerspiegelt als tatsächlich vorhanden ist, etwa so:

```
if(a) if(b) C;
else D;
```

An der Formatierung kann man erkennen, dass der Programmierer wahrscheinlich etwas anderes gemeint hat, als tatsächlich im Code steht. Solche Situationen, die leicht bei Programmänderungen entstehen, vermeidet man am besten durch geschwungene Klammern um die einzelnen Programmzweige.

Gelegentlich spricht man verallgemeinernd immer dann von einem Dangling-else-Problem, wenn man im Programmcode den hinteren Teil einer Anweisung (z.B. das `else`) nicht auf einen Blick mit dem Beginn dieser Anweisung verbinden kann, oder wenn der Beginn überhaupt fehlt. Anfangs- bzw. Endpunkt der Kontrollstruktur hängen nicht klar zusammen. Zur Vermeidung sollte man auf die richtige Formatierung achten, bei der die Intuition mit der tatsächlichen Programmstruktur übereinstimmt und einzelne Codestücke nicht zu lang sind. Die strukturierte Programmierung lässt nur zusammen mit der richtigen Formatierung ihre Vorteile wirksam werden.

6.4.2 Typische Fallen objektorientierter Sprachen

Die objektorientierte Programmierung bietet eine Vielzahl an Möglichkeiten zur Strukturierung von Programmen. Aber gerade diese Vielfalt stellt auch eine Gefahrenquelle dar. Man kann leicht etwas falsch verstehen oder falsch verwenden. Die Folgen können schwerwiegend sein, während die Ursachen oft unklar bleiben. Folgende Themenbereiche spielen dabei immer wieder eine Rolle:

Ersetzbarkeit: Die objektorientierte Programmierung lebt vor allem davon, dass Instanzen von Untertypen verwendbar sind, wo Instanzen von Obertypen erwartet werden. Schwere Fehler entstehen, wenn man Ersetzbarkeit fälschlicherweise annimmt. Der Compiler erkennt und verhindert Fehler bezüglich der Ersetzbarkeit, die sich in der Signatur widerspiegeln. Insbesondere muss die Signatur einer überschreibenden Methode jener der überschriebenen Methode im Wesentlichen entsprechen. Aber das in Kommentaren beschriebene Verhalten von Unter- und Obertypen ist für den Compiler unverständlich und nicht überprüfbar. Darum müssen wir uns beim Programmieren selbst kümmern. Wir müssen sicherstellen, dass sich ein Objekt jedes Untertyps so verhält, wie wir es uns von einem Objekt eines Obertyps erwarten. Die Ursachen andernfalls entstehender Fehler sind schwer zu finden und zu beseitigen. Kommentare sind von entscheidender Bedeutung. Vergessene, veraltete oder unverständlich formulierte Kommentare können große Software-Projekte scheitern lassen.

Zusicherungen: Kommentare treten vor allem in Form von Zusicherungen auf Methoden auf. Das ist gut so. Aber manchmal verwendet

man Zusicherungen, um komplizierte Bedingungen für die Verwendung von Methoden festzulegen, damit die Methoden etwas einfacher zu implementieren sind. Das ist keine gute Idee, weil sich die Komplexität des Programms durch zusätzlichen Code beim Senden entsprechender Nachrichten meist unnötig erhöht.

Kovarianz: Einige wenige Probleme lassen sich in der objektorientierten Programmierung prinzipiell nicht auf typsichere Weise lösen. Das betrifft sogenannte *kovariante Probleme*, bei denen wir uns wünschen würden, dass ein formaler Parameter in einer überschreibenden Methode ein (ungleicher) Untertyp des entsprechenden Parametertyps der überschriebenen Methode ist. In Kapitel 3 haben wir als Beispiel dafür die Methode `equals` betrachtet, die in `Object` definiert ist und in vielen Klassen überschrieben wird. Der formale Parameter dieser Methode ist immer `Object`. Eigentlich würden wir uns wünschen, dass der Typ des formalen Parameters jeweils gleich der Klasse wäre, in der die Methode steht. Aus prinzipiellen Gründen ist das jedoch in keiner Programmiersprache möglich, die stark typisiert ist und Ersetzbarkeit auf diesem Parameter unterstützt. Glücklicherweise betrifft dieses Problem nur wenige Methoden.

Als Falle stellen sich weniger die kovarianten Probleme an sich dar, als viel mehr die zahlreichen Lösungsversuche. Viele kovariante Probleme lassen sich durch eine etwas andere, verallgemeinernde Formulierung umgehen. Aber die Lösungsversuche enden häufig in fehleranfälligen Code-Stücken, die nur unter einer Vielzahl komplizierter Bedingungen verwendbar sind.

Cast: Ein Lösungsansatz für kovariante Probleme besteht in der Verwendung von Casts auf Referenztypen. Obwohl Casts zur Laufzeit auf Korrektheit überprüft werden – es wird sichergestellt, dass das Objekt tatsächlich den gewünschten Typ hat – sind Casts generell sehr fehleranfällig. Meist wissen wir erst zur Laufzeit, welchen dynamischen Typ das Objekt hat. Die Ausnahme, die bei einem falschen dynamischen Typ geworfen wird, können wir in der Regel nicht auf sinnvolle Weise abfangen. Sie führt nicht selten zum Programmabbruch. Daher sollten wir Casts so gut es geht vermeiden.

Generizität: Eine Lösung vieler solcher Probleme scheint die Generizität zu bieten. Mittels Generizität lassen sich sowohl so manche kovariante Probleme lösen als auch viele Casts vermeiden. Vor allem zur

Realisierung von Datenstrukturen ist Generizität heute unverzichtbar. Eine Eigenschaft kann Generizität alleine aber nicht bieten – nämlich Ersetzbarkeit. Also gerade der Bereich, in dem die objektorientierte Programmierung die größten Stärken hat, ist durch die Generizität nicht abgedeckt. Wenn man eine auf Ersetzbarkeit beruhende Lösung gegen eine auf Generizität beruhende austauscht (beispielsweise um Casts zu vermeiden), so verzichtet man dabei auf Ersetzbarkeit. Da die Ersetzbarkeit für die Wartung sehr vorteilhaft ist, sollte man nicht ohne wichtigen Grund darauf verzichten.

Sichtbarkeit: Einschränkungen der Sichtbarkeit (etwa durch `private`) bewirken genau das, was der Begriff ausdrückt: Sie schränken uns in der Freiheit beim Programmieren ein. Das ist der Grund, warum wir gelegentlich gerne auf diese Einschränkung verzichten. Dabei vergessen wir aber, dass die Einschränkung einen Sinn hat: Sie hilft Objekte voneinander zu entkoppeln und nötige Programmänderungen lokal zu halten. Das ist eine wichtige Voraussetzung für gute Wartbarkeit. Deswegen sollen wir die Sichtbarkeit niemals unnötig weit ausdehnen. Einmal ausgedehnt lässt sich die Sichtbarkeit nur mehr sehr schwer wieder auf einen kleineren Bereich beschränken.

Vererbung: Die Vererbung zwischen Klassen erfüllt eine wichtige Funktion. Sie erlaubt Unterklassen Methoden ohne Neudefinition aus Oberklassen zu übernehmen. Aus Oberklassen ererbte Methoden können auf private Variablen der Oberklassen zugreifen, die in den Unterklassen nicht sichtbar sind. Ein übertriebener Einsatz von Vererbung hat aber auch Schattenseiten. Es bringt kaum Vorteile, wenn man von wenig stabilen Klassen ableitet, da diese sich häufig so ändern, dass auch die Unterklassen davon betroffen sind. Wenn man unter allen Umständen Vererbung einsetzen will, obwohl das in einer bestimmten Situation schwierig ist, läuft man Gefahr, dass man dabei die Ersetzbarkeit verletzt. Das ist unbedingt zu vermeiden. Ersetzbarkeit ist in jedem Fall wichtiger als das Erben einiger Methoden.

Effizienz: Natürlich möchten wir möglichst effiziente Programme schreiben. Aber ein unkontrollierter Effizienz-Wahn ist unbedingt zu vermeiden. Manchmal versucht man mit allen Mitteln (etwa durch Definition von Methoden als `static`, `final` oder `private`) die Notwendigkeit von dynamischem Binden zu vermeiden, weil man weiß, dass statisches Binden um eine Spur effizienter ist. Derartige Bemü-

hungen bewirken jedoch sehr oft genau das Gegenteil. Sogar wenn das Programm minimal effizienter werden sollte, verliert man viel. Dynamisches Binden ist ja die Basis der Ersetzbarkeit. Ohne dynamisches Binden sind die Programme weitaus schwieriger zu warten.

Funktionalität: Die objektorientierte Programmierung beruht auf der Simulation von Objekten aus der realen Welt. Mit wenig praktischer Programmiererfahrung passiert es leicht, dass man zu viele oder die falschen Aspekte der realen Welt in einem zu hohen Detaillierungsgrad simuliert. Man entwickelt also eine Funktionalität die nie gebraucht wird. Es ist schwierig, die richtige Balance zwischen einer sehr pragmatischen Vorgehensweise (ohne Rücksicht auf die reale Welt) und einer detailgetreuen Simulation zu finden. Das ist einer der Gründe, warum die objektorientierte Programmierung so schwierig ist. Man muss viel Erfahrung sammeln, bevor man diese Fallen sicher meistert.

6.4.3 Spezielle Fallen in Java

Die meisten der oben beschriebenen Fallen existieren in vielen oder den meisten (objektorientierten) Programmiersprachen. Es gibt aber auch Fallen, die vor allem in Java und Java-ähnlichen Sprachen existieren:

Klasse oder Interface: Für den Aufbau von Untertypbeziehungen eignen sich Klassen und Interfaces. Jedoch sind Klassen derart eingeschränkt, dass eine Klasse nur von einer anderen Klasse abgeleitet sein kann. Dagegen besteht bei Interfaces die Beschränkung, dass keine Methoden geerbt werden können. In der Praxis kann man in einer frühen Entwicklungs-Phase oft kaum entscheiden, ob in einem bestimmten Fall eine Klasse oder ein Interface besser geeignet ist. Diese Entscheidung ist nicht nur für Programmieranfänger schwer zu treffen. Auch andere Sprachen wie C# haben dieses Problem.

Mit der Unterscheidung zwischen Klassen und Interfaces wollte man Schwierigkeiten umgehen, die zusammen mit Mehrfachvererbung etwa in C++ auftreten. Vor allem beim Programmieren in älteren C++-Versionen wurde Mehrfachvererbung häufig falsch eingesetzt. Das wollte man in Java vermeiden. Allerdings weiß man inzwischen mehr darüber, wie die Vererbung von Code aus der Oberklasse funktioniert, und es existieren ausgefeiltere Konzepte dafür als ursprüng-

lich in C++. Diese Konzepte haben noch nicht ihren Weg in Java und Java-ähnliche Sprachen gefunden.

Überladen: Methoden können in Java überladen sein, wobei mehrere Methoden desselben Namens mit unterschiedlichen Parametertypen in derselben Klasse existieren. Überladen ist gelegentlich nützlich, weil man nicht künstlich unterschiedliche Namen für ähnliche Funktionalität finden muss. Aber gerade in Java ist Überladen auch gefährlich, weil man manchmal nur schwer erkennen kann, welche der Methoden mit gleichem Namen aufgerufen wird. Zur Unterscheidung zwischen den Methoden werden die deklarierten Typen der Argumente herangezogen, nicht wie beim Senden von Nachrichten die dynamischen Typen. Daraus ergeben sich oft sehr diffizile Unterschiede, die übersehen werden und zu Fehlern führen. Am besten setzt man Überladen nur dort ein, wo Verwechslungen ausgeschlossen sind, beispielsweise weil alle Methoden gleichen Namens eine unterschiedliche Anzahl an Parametern haben.

Überladen versus Überschreiben: Eine Methode der Unterklasse überschreibt eine Methode der Oberklasse, wenn sie (bis auf den Ergebnistyp) dieselbe Signatur hat. In allen anderen Fällen, in denen wir in der Unterklasse eine Methode einführen, die gleich heißt wie eine Methode aus der Oberklasse, wird die aus der Oberklasse ererbte Methode mit der in der Unterklasse definierten Methode überladen; es existieren also beide Methoden gleichzeitig. So einfach die Regel zur Unterscheidung zwischen Überschreiben und Überladen klingt, so fehleranfällig ist der Umgang damit in der Praxis. Es passiert leicht, dass man unabsichtlich beispielsweise den Typ eines Parameters ändert oder zwei Parameter vertauscht. Der Compiler akzeptiert Derartiges ohne Warnungen oder Fehlermeldungen. Erst beim Testen kann man ein unerwartetes, eigenartiges Programmverhalten feststellen. Andere Sprachen wie beispielsweise C++ sind hinsichtlich des Überladens von Methoden restriktiver, sodaß der Compiler in den meisten Fällen, wo unabsichtlich überladen statt überschrieben werden könnte, eine Fehlermeldung gibt.

In neueren Java-Versionen kann man beim Überschreiben der Methode (direkt vor die Definition der überschreibenden Methode) die Annotation `@Override` hinschreiben. Dann überprüft der Compiler, ob tatsächlich eine Methode überschrieben wird, und gibt eine

Fehlermeldung aus, wenn das nicht der Fall ist. Die Verwendung von `@Override` ist sinnvoll und empfehlenswert. Allerdings wirkt diese Technik nur in eine Richtung – also wenn eine Methode irrtümlich überladen wird, obwohl sie überschrieben werden soll. Wenn eine Methode unabsichtlich überschrieben wird, obwohl wir Überladen erwarten, gibt es keine Fehlermeldung. Die meisten integrierten Entwicklungsumgebungen bieten eine andere Lösung für dieses Problem an: In der Regel werden überschriebene Methoden farblich anders gekennzeichnet als überladene und fallen daher gleich beim Schreiben des Codes auf, nicht erst beim Übersetzen.

Ersetzbarkeit von Arrays: Über Generizität ist es prinzipiell (das heißt, in allen Programmiersprachen) nicht möglich, sogenannte implizite Untertypbeziehungen einzuführen: So ist `Liste` auch dann kein Untertyp von `Liste<A>` wenn `B` ein Untertyp von `A` ist. Man könnte sonst etwa in ein Objekt vom Typ `Liste` ein Objekt vom Typ `A` einfügen, das keine Instanz von `B` ist. Implizite Untertypbeziehungen sind also nicht sicher, und der Compiler garantiert, dass keine solchen Untertypbeziehungen verwendet werden. Aber in Java (genauso wie in C#) ist ein Array-Typ `B[]` Untertyp des Array-Typs `A[]` wenn `B` ein Untertyp von `A` ist. Dadurch können wir beispielsweise folgenden Programmcode schreiben:

```
B[] bs = ...;
A[] as = bs;      // weil B Untertyp von A
as[0] = new A();  // Fehler !!
```

Das in der dritten Zeile in das Array geschriebene Objekt ist danach auch in `bs`, obwohl `bs` nur Objekte vom Typ `B` enthalten soll, keine vom Typ `A`. In solchen Fällen wird zur Laufzeit eine Ausnahme geworfen, da der Compiler solche Fehler meist nicht feststellen kann. Die Verwendung von Untertypbeziehungen zwischen Arrays ist in Java (und C#) also sehr gefährlich und fehleranfällig. Am besten verzichtet man darauf.

Raw Types: Generizität ist in Java nachträglich eingeführt worden, ohne die Zwischensprache oder die Standardbibliotheken dafür in nennenswerter Weise ändern zu müssen. Leider hält die dabei entstandene Form der Generizität einige Fallen bereit. Einerseits werden

bestimmte sinnvoll erscheinende Operationen einfach nicht unterstützt, beispielsweise das Erzeugen eines Arrays, das Instanzen eines Typparameters enthält. Der Compiler kann das Programm nicht übersetzen, wenn wir solche Operationen zu verwenden versuchen. Schwerwiegender ist aber ein anderes Problem: Wenn beispielsweise `List<A>` ein Typ ist, dann ist auch `List` ein Typ, ein sogenannter Raw Type. Bei der Verwendung von Raw Types sind einige Typüberprüfungen ausgeschaltet. In speziellen Sonderfällen ist die Verwendung sinnvoll, jedoch immer gefährlich und fehleranfällig. Ohne auf diese Sonderfälle einzugehen sagen wir generell, dass Raw Types zu vermeiden sind. Meist wollen wir gar keine Raw Types verwenden, sondern vergessen einfach auf das Anschreiben der spitzen Klammern. Solche Fehler kann der Compiler leider nicht erkennen. Es werden einfach nur bestimmte Überprüfungen ausgeschaltet und somit inkorrekte Programme akzeptiert. Daher müssen wir stets darauf achten, spitze Klammern hinzuschreiben.

Pakete: Neben Klassen braucht man in Programmen auch größere Strukturierungseinheiten, die häufig Module oder Pakete heißen. Java hat dafür eine sehr einfache Lösung gewählt: Alle Klassen, die im selben Verzeichnis stehen, bilden zusammen ein Paket. So überzeugend diese Lösung auf den ersten Blick aussieht, so problematisch ist sie in der Praxis. Oft möchte man die Verzeichnisstruktur ändern und an neue Gegebenheiten anpassen. Das geht aber nicht, weil man dabei auch das Paket ändern und damit vermutlich zerstören würde. Fast alle Programmiersprachen haben in dieser Hinsicht bessere Strukturierungsmöglichkeiten im Großen anzubieten als Java.

Falsche Sicherheit: Java gilt als sichere Programmiersprache, weil sehr viele Fehler bereits vom Compiler erkannt werden und viele weitere zur Laufzeit zum Werfen einer Ausnahme führen. Es wird sichergestellt, dass Java-Programme keinen Zugriff zu Ressourcen bekommen, zu denen sie keinen Zugriff haben sollen. Aufgrund dieser Sicherheit, die in einigen Bereichen zweifelsfrei gegeben ist, trauen wir Java-Programmen. Leider ist die Sicherheit nicht in allen Bereichen gegeben. Natürlich kann man auch in Java fehlerhafte und bösartige Programme schreiben, und auch ein Java-Interpreter selbst kann fehlerhaft sein. Oft wird die Sicherheit von Java deutlich überschätzt. Das führt dazu, dass wir manchmal zu wenig testen und uns einfach auf etwas verlassen, was nicht gegeben ist.

6.5 Vertrauen und Kontrolle

Programmieren ist fast immer Teamarbeit. Teamarbeit funktioniert nur, wenn man den Kolleginnen und Kollegen im Team Vertrauen entgegenbringt und sich selbst als vertrauenswürdig erweist. Allerdings ist das Vertrauen nicht in jedem Fall gerechtfertigt. In manchen Situationen ist eine gesunde Portion an Misstrauen angebracht, die sich darin äußert, dass man Daten oder Programmteile ignoriert oder noch einmal überprüft, bevor man sie verwendet. Diese Form der Kontrolle verursacht zum Teil erhebliche Kosten. Daher ist das gegenseitige Vertrauen in einem Team ein wesentlicher Faktor bei der Konstruktion von Programmen.

6.5.1 Defensive und offensive Programmierung

Defensive Programmierung ist zu einem häufig verwendeten Begriff geworden. Darunter versteht man einen Programmierstil, bei dem man sich nicht blind auf irgendwelche Bedingungen verlässt, sondern lieber alles mehrfach überprüft. Die defensive Programmierung folgt also dem Grundsatz: „Vertrauen ist gut, Kontrolle ist besser.“ Dem steht ein *offensiver Programmierstil* gegenüber, der eher diesem Grundsatz folgt: „Solange sich niemand beschwert, wird es schon passen.“ Wenn es um die Konstruktion hochwertiger Software geht, hat ein offensiver Programmierstil einen negativen Beigeschmack. Man hat stets das Gefühl, dass man wichtige Bedingungen einfach nur aus Schlampigkeit nicht überprüft und irgendwann das gesamte Programm zum Müll geworfen werden muss, weil sich irgendwo schwerwiegende, irreparable Mängel zeigen. Das kann passieren. Es kann auch sein, dass trotz eines defensiven Programmierstils schwerwiegende, irreparable Mängel auftreten – jedoch mit geringerer Wahrscheinlichkeit. Viel schwerwiegender ist bei einem defensiven Programmierstil die Gefahr, dass sich durch ein Übermaß an Kontrolle und mehrfachen Überprüfungen die Programmentwicklung oder das entstehende Programm als zu ineffizient und (hinsichtlich der Entwicklungskosten oder des Ressourcenbedarfs zur Laufzeit) teuer erweist, und die Entwicklung daher abgebrochen wird.

Oft wird der Begriff eines offensiven Programmierstils nur als Synonym für einen schlampigen Programmierstil verwendet und daraus implizit abgeleitet, dass ein defensiver Programmierstil stets vorzuziehen ist. Diese Sichtweise geht jedoch an der Sache vorbei. In vielen Situationen ist ein offensiver Programmierstil durchaus vorteilhaft und hat nichts mit Schlamperei zu tun, sondern nur mit dem Vermeiden unnötiger Überprüfungen.

Tatsächlich kann auch ein defensiver Programmierstil schlampig sein, wenn man einfach alle Bedingungen überprüft, die einem als möglicherweise sinnvoll erscheinen. Überprüft werden sollen nur im Programmablauf nötige Bedingungen, keinesfalls irgendwelche anderen vielleicht sinnvollen Bedingungen.

Ein guter Programmierstil wird an allen Stellen defensiv sein, wo dies nötig ist, und an allen anderen Stellen offensiv. In folgenden Fällen wird diesbezüglich häufig ein falscher Programmierstil eingesetzt:

Validierung: Daten, die aus externen Quellen stammen (beispielsweise Benutzereingaben), müssen in jedem Fall überprüft werden – siehe Abschnitt 5.6.1. Auch bei einem sehr offensiven Programmierstil darf man davon nicht abgehen. Schwierig werden solche Validierungen der Daten vor allem dann, wenn keine klaren Zuständigkeiten dafür bestehen, also beispielsweise bestimmte Eigenschaften eines Wertes an der einen Stelle überprüft werden, andere Eigenschaften an einer anderen Stelle. In diesen Fällen ist das Programm bezüglich der Validierung schlecht faktorisiert. Es passiert leicht, dass man einerseits auf bestimmte Überprüfungen vergisst und andererseits (aus Angst vor vergessenen Überprüfungen) unnötige bzw. bereits erfolgte Überprüfungen wiederholt macht. Beides ist schlecht, sowohl vergessene als auch unnötige Überprüfungen. Die einzige sinnvolle Lösung besteht in einer derartigen Strukturierung des Programms, dass stets klar ist, wo und von wem die Validierungen durchgeführt werden. Ein defensiver hat gegenüber einem offensiven Programmierstil hinsichtlich solcher Validierungen keine Vorteile, kann jedoch bis zu einem gewissen Grad eine schlechte Faktorisierung verstecken. Die Kombination von schlechter Faktorisierung und defensivem Programmierstil wirkt sich fast immer negativ aus: Es wird viel unnötiger Programmcode für unnötige bzw. wiederholte Überprüfungen geschrieben, diese erhöhen den Ressourcenverbrauch, und nicht selten sind eigentlich gleichbedeutende Überprüfungen an unterschiedlichen Programmstellen nicht miteinander konsistent.

Zusicherungen: Kommentare an Programmschnittstellen sind in der Regel als Zusicherungen (z.B. Vor- und Nachbedingungen) zu verstehen. In Zusicherungen formulierte Bedingungen sind unbedingt einzuhalten. Für die Einhaltung von Vorbedingungen sind Clients (Aufrufer von Methoden und Konstruktoren) zuständig, für andere Arten von Zusicherungen Server (die ausgeführten Methoden bzw. Konstruk-

toren). Das ist durch Design by Contract ganz klar geregelt – siehe Abschnitt 5.1.2. Trotzdem stellt sich immer wieder die Frage, ob man nicht zusätzlich zu den Kommentaren auch Überprüfungen der Bedingungen machen sollte – nur zur Sicherheit. In jedem Softwarevertrag gibt es zwei Vertragspartner. Entsprechend muss man zwei Fälle unterscheiden:

- Der Vertragspartner, der zur Erfüllung einer Bedingung verpflichtet ist, muss unter allen Umständen für die Einhaltung sorgen (als Client für Vorbedingungen, als Server für die anderen Zusicherungen). Man muss wissen, dass die Bedingungen erfüllt sind. Das geht nur, wenn man die Algorithmen von Anfang an entsprechend auslegt. Mit einer einfachen Überprüfung kommt man meist nicht aus, da man ja auch im Falle des Scheiterns einer Überprüfung für die Einhaltung der Bedingung sorgen muss – beinahe ein Widerspruch. Der Unterschied zwischen defensiver und offensiver Programmierung spielt dafür keine Rolle.
- Der andere Vertragspartner kann sich darauf verlassen, dass die Bedingungen eingehalten sind (der Server auf Vorbedingungen, der Client auf andere Zusicherungen). Dafür sind keine Überprüfungen nötig. Bei einem defensiven Programmierstil überprüft man viele Bedingungen trotzdem noch einmal, bei einem offensiven Stil verzichtet man darauf. In einem beschränkten Ausmaß können solche Überprüfungen zum Aufdecken von Fehlern durchaus hilfreich sein. Eine vollständige Überprüfung ist dagegen nicht zweckmäßig. Für manche Bedingungen wären die Überprüfungen einfach viel zu aufwendig. Systematisch durchgeführte unnötige Überprüfungen führen auch leicht dazu, dass der Vertragspartner seine Verantwortung für die Einhaltung des Vertragsbestandteils nicht mehr ernst nimmt; schließlich werden die Bedingungen ohnehin noch einmal überprüft. Das kann längerfristig dazu führen, dass die Bedingungen entgegen den Erwartungen nur einmal überprüft werden, und zwar an einer dafür schlecht geeigneten Stelle. Überprüfungen durch `assert`-Anweisungen schaden aber ziemlich sicher nicht. Man weiß ja, dass `assert`-Anweisungen nur selten ausgeführt werden und kann sich daher nicht auf die Überprüfung verlassen.

Ein Vertragspartner darf sich auf nichts verlassen, was vom anderen nicht zugesichert wird. Daran ändern auch Überprüfungen nichts.

Wiederverwendbarkeit: Wenn man ein Programmstück (etwa eine Methode) konstruiert, hat man dafür meist einen bestimmten Anwendungsfall im Kopf. Die Zusicherungen richten sich nach diesem Anwendungsfall aus. Es passiert leicht, dass restriktive Bedingungen formuliert werden, die in diesem Anwendungsfall erfüllt sind, in anderen vielleicht ebenso sinnvollen Anwendungsfällen aber nicht. Durch solche Bedingungen verhindert man die Wiederverwendung des Programmstücks in einem anderen Kontext. Daher sollte man unnötig restriktive Bedingungen vermeiden, vor allem solche, die zur Ausführung des Programmstücks gar nicht nötig sind. Überprüfungen in `assert`-Anweisungen können helfen, unnötige Bedingungen aufzudecken. Schließlich wird man sich beim Programmieren (genauer: bei der statischen Analyse des Programmstücks) stets fragen, was die `assert`-Anweisung bewirkt. Falls sie innerhalb des Programmstücks wirkungslos bleibt, kann man die Bedingung genauso gut entfernen. Das ist ein Beispiel dafür, wie ein defensiver Stil die Qualität eines Programms gegenüber einem offensiven Stil auf unerwartete Weise verbessern kann. Auf den ersten Blick vermutet man ja leicht genau das Gegenteil, dass nämlich ein defensiver Stil die Wiederverwendbarkeit durch zusätzliche Überprüfungen einschränken würde. Entscheidend ist in diesem Fall aber nicht die Unterscheidung zwischen defensiv und offensiv, sondern vielmehr die Genauigkeit der Analyse des Programmstücks. Ein defensiver Stil führt häufiger zu einer genauen Analyse als ein offensiver.

6.5.2 Programmierstil und Vertrauen

Die Wiederverwendung von Programmteilen ist vor allem eine Vertrauensfrage: „Kann ich darauf vertrauen, dass ein fremdes (= nicht von mir geschriebenes) Programmstück die von mir erwartete Qualität hat?“ In diesem Zusammenhang ist Qualität sehr allgemein zu verstehen. Es geht nicht nur um Funktionalität und Zuverlässigkeit, sondern auch um Stabilität der Schnittstellen (sodass Ersetzbarkeit gewährleistet bleibt) und die Qualität der Wartung (falls irgendwo Mängel auftreten). Man muss nicht nur den Programmteilen vertrauen, sondern vor allem auch den Entwicklern der Programmteile. Letzteres ist ein soziales Problem und mit den Mitteln der Technik kaum in den Griff zu bekommen. Aber Programmcode, als Kommunikationsmedium betrachtet, bietet doch eine Entscheidungsbasis zur Abschätzung der Vertrauenswürdigkeit. Mit etwas Erfahrung erkennt

man rasch, wieviel Aufwand in die Entwicklung eines Programmstücks gesteckt wurde. Es ist leicht, sich auf guten, bewährten und offensichtlich stabilen Code zu verlassen. Andererseits sollte man sich ohnehin nicht auf mangelhaften, instabilen Code verlassen.

Beim Programmieren muss man viel tun, um Vertrauen in den Programmcode zu gewinnen. Hier sind einige Aspekte zusammengefasst, die für das Vertrauen entscheidend sind:

Schnittstellen: Die für die Benutzung eines Programmstücks notwendigen Informationen müssen klar und deutlich bekanntgegeben werden. Außerdem müssen die Schnittstellen einfach und in sich logisch gestaltet sein. In allen anderen Fällen ist das Programmstück für Personen, die an der Entwicklung nicht direkt beteiligt waren, einfach nicht verwendbar. Die gute Ausgestaltung der Schnittstelle zum Programmstück ist jedoch mit viel Arbeit und hohen Kosten verbunden. Wenn man will, dass das Programmstück auch von anderen verwendet wird, führt daran aber kein Weg vorbei. Nicht für jedes Stückchen Code zahlt sich dieser Aufwand aus. Man muss sich eindeutig dafür oder dagegen entscheiden, ein Programmstück für die Verwendung durch andere vorzusehen. Wenn man sich dafür entscheidet, muss man auch den nötigen Aufwand hineinstecken. Wenn man sich dagegen entscheidet, darf man nicht erwarten, dass der Code trotzdem von anderen verwendet wird. Im Zweifelsfall wird man sich eher gegen die Verwendung durch andere entscheiden, oft auch deswegen, weil nicht ausreichend viele Ressourcen für eine gute Ausgestaltung der Schnittstelle zur Verfügung stehen.

Verständlichkeit: Man will natürlich nur Programmstücke ausreichender Qualität verwenden. Dabei stellt sich aber die Frage, woran man die Qualität messen kann. Ein einfaches und gleichzeitig objektives Entscheidungskriterium zur Beantwortung der Frage gibt es nicht. Oft kann aber schon ein kurzer Blick in den Programmcode viel verraten: Einfacher, logisch strukturierter und gut lesbarer Code wird positiv auffallen, auch wenn man nicht den gesamten Code von vorne bis hinten durchschaut. Kompliziert gestalteter (sogenannter *barocker*) Code wird negativ auffallen. Diese Kriterien sind nicht wirklich objektiv, da ein Grund für barocken Code auch in der Komplexität der Aufgabe liegen kann. Mit ausreichend Erfahrung kann man aber abschätzen, wieviel Aufwand betrieben wurde, um barocken Code zu vermeiden. Vor allem erkennt man auch Stellen, an denen die Ver-

mutung besteht, dass der Code absichtlich undurchschaubar gehalten wurde. Bereits einige wenige solche Stellen lassen das Vertrauen in den Code schwinden. Man weiß einfach nicht, warum der undurchschaubare Code existiert. Es kann sein, dass

- beim Korrigieren von Fehlern nicht behutsam vorgegangen und zusätzlicher Code zum Umgehen von Fehlern eingebaut wurde, wodurch der Code wahrscheinlich noch sehr fehlerhaft ist,
- keine klare Lösung einer Teilaufgabe besteht und viel Code für den Umgang mit in Tests vorgekommenen Einzelfällen eingefügt wurde, der aber im Allgemeinen nicht funktioniert,
- oder absichtlich irgendetwas versteckt werden soll, vielleicht sogar Code, der dem Anwender Schaden zufügt.

Alle diese Möglichkeiten stellen Gründe dar, den Code nicht zu verwenden. Daher sollte man sich sehr darum bemühen, den Anschein von undurchschaubarem Code gar nicht erst aufkommen zu lassen. Vorsichtig sollte man sein, wenn man den Code durch Kommentare verständlicher machen möchte: Gelegentlich wird der Code gerade durch umfangreiche und wenig aussagekräftige oder sogar offensichtlich falsche Kommentare erst recht undurchschaubar.

Entwicklungsprozesse: Nicht nur der Code selbst dient zur Abschätzung der Qualität, sondern auch die eingesetzten Entwicklungsprozesse. Es muss transparent sein, wie bei der Konstruktion jeden Programnteils vorgegangen wird. Vor allem ist interessant, welche Maßnahmen zur Qualitätskontrolle angewandt werden und auf welche Weise auf das Bekanntwerden von Fehlern reagiert wird. Mit ausreichend Erfahrung kann man erkennen, ob der Programmcode mit den bekannt gemachten Entwicklungsprozessen zusammenpasst. Treten dabei Diskrepanzen auf, wird das Vertrauen sehr rasch schwinden.

Wartung: Software lebt nur solange sie gewartet wird. Das gilt auch für einzelne Programmstücke. Wenn der Anschein entsteht, dass hinter einem Programmstück keine Person oder Personengruppe mehr steht, die das Programmstück mit ausreichend Mitteln am Leben erhält, wird man nicht mehr darauf vertrauen.

Standardkonformität: Standards sind häufig umstritten. Einerseits garantiert die Einhaltung von Standards ein Mindestmaß an Qualität, auf das man sich verlassen kann, andererseits hinken Standards der

technischen Entwicklung immer hinterher. Trotzdem ist es notwendig, sich an Standards zu halten, wenn sie in dem betrachteten Bereich existieren und anwendbar sind. Wenn man Standards ignoriert, besteht schnell der Verdacht, dass man in diesem Bereich einfach nicht genug Wissen hat, um ein Problem effektiv lösen zu können.

6.5.3 Einheitliche Regeln

Innerhalb eines Teams ist gegenseitiges Vertrauen das oberste Gebot. Ohne Vertrauen kann keine brauchbare Software entstehen. Misstrauen innerhalb eines Teams führt zu

- Eigenbrötelei und mangelnder Kommunikation im Team,
- Mehrgleisigkeiten, weil Lösungen mehrfach entwickelt werden statt bereits fertiger Lösungen zu verwenden,
- viel zusätzlichem Code für eigentlich sinnlose Überprüfungen,
- hoher Fehleranfälligkeit durch widersprüchliche überprüfte Bedingungen und inkonsistente Mehrfachlösungen,
- hohem Ressourcenverbrauch sowohl in der Entwicklung als auch in der Programmausführung
- und schließlich sehr oft zum Scheitern eines Projekts.

Aus diesen Gründen muss man viel unternehmen um das gegenseitige Vertrauen zu fördern. Es reicht nicht, nur die soziale Einbindung aller Teammitglieder in das Team zu unterstützen, da das Vertrauen, wie oben beschrieben, zu einem guten Teil auch vom Programmierstil abhängt. Daher ist es wichtig, dass alle Teammitglieder einem einheitlichen gemeinsamen Programmierstil folgen. In kleinen Teams von Leuten, die über einen längeren Zeitraum zusammen Programme entwickeln, entsteht von selbst ein gemeinsamer Stil. Dieser Stil hängt häufig von der Aufgabenteilung und den speziellen Fähigkeiten einzelner Teammitglieder ab. Daher entwickeln unterschiedliche Teams meist auch unterschiedliche Stile.

In großen Teams bzw. Unternehmen würde sich eine unüberschaubare Ansammlung unterschiedlicher Programmierstile ergeben, wenn jede kleine Personengruppe einen eigenen Stil entwickelt. Um dem vorzubeugen werden häufig klare Vorgaben gemacht. Beispielsweise wird festgelegt,

- an welche Stellen welche Art von Kommentaren zu schreiben ist,
- wie Einrückungen und Klammerungen zu verwenden sind,
- an welchen Programmstellen wer welche Art von Änderungen vornehmen darf und wie die Änderungen zu dokumentieren sind,
- wie beim Entwurf und beim Testen vorzugehen ist,
- und weitere Punkte könnte man fast endlos aufzählen.

Solche Regeln sollen einen einheitlichen Programmierstil erzwingen und damit dem Team oder Unternehmen diesbezüglich eine eigene Identität verschaffen und das gegenseitige Vertrauen fördern. Insofern erfüllen die Regeln einen ähnlichen Zweck wie einheitliche Kleidung oder ein Firmenlogo. Aber der Zweck der Regeln geht klar darüber hinaus. Die Regeln haben sich im Laufe der Zeit aus der Erfahrung entwickelt. Sie zeigen einen Weg vor, wie man typische Probleme rasch erkennen oder gar nicht erst entstehen lassen kann. Ergibt sich eine neue Klasse von Problemen, so sucht man nach einem Ausweg in Form neuer Regeln, welche die negativen Auswirkungen dieser Probleme verhindern sollen. Im Laufe der Zeit ergibt sich ein Regelsystem, auf das man beim Programmieren vertrauen kann. Man vertraut nicht nur dem Regelsystem, sondern auch allen Personen, die sich beim Programmieren an das Regelsystem halten. So ergibt sich mit der Zeit ein schlagkräftiges Team, das auf der Basis des Vertrauens auch komplexe Software auf effiziente Weise entwickeln kann.

6.6 Mythen

Die Programmierung sowie Programmiersprachen und Programmierstile von einer Unzahl an Mythen umrankt. Es liegt in der Natur von Mythen, dass manche von ihnen sich auch bei eingehender und objektiver Untersuchung als zutreffend erweisen, andere wiederum nicht. Häufig sind sie jedoch so nebulös, dass eine objektive Untersuchung gar nicht möglich ist. Oft erweist sich ein Mythos als kurzfristige Modeerscheinung, gelegentlich als etwas sehr Dauerhaftes.

Manchmal entwickeln sich Mythen hin zu eigenen Ideologien oder beinahe schon religiösen Glaubenswahrheiten. Wahrscheinlich kennt jeder Informatiker jemanden, der eine bestimmte Programmiersprache, ein Betriebssystem, eine Marke oder eine Technologie als die oder das einzig Wahre

betrachtet. So jemand wird unzählige Argumente dafür finden und Gegenargumente einfach ignorieren. Eine derartige Ideologisierung kommt der Industrie sehr entgegen und wird auf vielfache Weise gefördert. In gewisser Weise hängt die Ideologisierung mit einheitlichen Regeln und darauf begründetem Vertrauen zusammen. Man vertraut auf das, an das man glaubt. Gleichzeitig entwickelt man einen Glauben an das, auf das man vertraut. Ohne Vertrauen ist keine vernünftige Softwareentwicklung möglich, und ganz ohne Glaube an das, was man macht und an die Werkzeuge, die man verwendet, kann man kaum erfolgreich sein.

Man muss aber auch realistisch sein. Es ist durchaus angebracht, einem Mythos zu folgen, der im Kern einer objektiven Untersuchung standhält. Andererseits ist es gefährlich, einem falschen Mythos zu folgen. Die Schwierigkeit besteht darin, falsche von wahren Mythen und kurzfristige Modetrends von dauerhaften Entwicklungen zu unterscheiden. Meist ist man selbst nicht in der Lage, diese Entscheidung zu treffen.

In Dingen, die man selbst nicht machen kann, verlässt man sich gerne auf andere. Nicht selten folgt man einem Mythos, weil das ein Vorbild, also eine andere Person, auf die man vertraut, auch macht. Leider weiß man in der Regel nicht, warum das Vorbild dem Mythos folgt. Vielleicht hat das Vorbild die Sache tatsächlich genau analysiert, vielleicht hat es generell einen guten Riecher für künftige Entwicklungen, vielleicht jagt es selbst aus Unwissenheit nur einem anderen Vorbild hinterher, und vielleicht gefällt sich das Vorbild einfach in dieser Rolle oder wird sogar dafür bezahlt, einen Mythos unter die Leute zu bringen. Der Wahrheitsgehalt eines Mythos hat jedenfalls wenig mit der Anzahl der Leute zu tun, die einem Mythos folgen. Es passiert immer wieder, dass eine größere Anzahl von Leuten den Empfehlungen eines „Gurus“ in der Programmierung blind Folge leistet, obwohl diese Empfehlungen nicht sinnvoll sind.

Es ist wichtig, dass man selbst mit der Zeit ein Gespür für den Wahrheitsgehalt von Mythen entwickelt. Man ist gut beraten, alle Empfehlungen, die man immer wieder bekommt, stets zu hinterfragen. Als Beispiel kann dieses Skriptum dienen. Darin werden unzählige Tipps und Ratschläge gegeben, wie bestimmte Sprachkonstrukte zu verwenden sind und wie nicht. Statt dem blind Folge zu leisten, sollte man alles hinterfragen. Nur wenn man selbst zur Überzeugung gelangt, dass die Begründungen stichhaltig sind, soll man sich danach richten. Wenn man vom Gegenteil überzeugt ist, wird man sich ohnehin nicht an eine Empfehlung halten. Solange man nicht überzeugt ist, sollte man nach tieferen Begründungen suchen. Das ist aufwendig. Daher hält man sich häufig auch an Empfehlungen,

die man nicht wirklich versteht. Genau aus solch unreflektiertem Befolgen und Weitergeben von Empfehlungen entstehen nicht selten Mythen von zweifelhaftem Wahrheitsgehalt.

Im Folgenden betrachten wir einige konkrete Mythen beispielhaft.

6.6.1 Paradigmen und Mythen

Imperative Programmierung effizient: Imperative Sprachen sind näher an der Hardware als deklarative Sprachen. Es ist allgemein bekannt, dass man in deklarativen Sprachen auf einem deutlich höheren Abstraktionsniveau programmiert und dadurch deutliche Einbusen hinsichtlich Laufzeiteffizienz in Kauf nehmen muss. Andererseits ist es leichter, auf höherem Abstraktionsniveau zu programmieren.

So weit der Mythos. Im Kern steckt einiges an Wahrheit, aber Größe und Wichtigkeit der Unterschiede wird häufig stark überschätzt. Imperative Sprachen erlauben zwar ein niedrigeres Abstraktionsniveau, aber meist programmiert man trotzdem auf hohem Abstraktionsniveau, was diesen Unterschied fast aufhebt. Bei imperativen Sprachen kann man sich leicht vorstellen, wie Sprachkonstrukte in die Sprache der Maschine übersetzt werden, bei deklarativen Sprachen ist das schwieriger. Das bedeutet jedoch nicht, dass deklarative Programme schwer übersetzbar sind, sondern nur, dass man sich kaum mit Details der Übersetzung auseinandersetzt. Aber es stimmt, dass ein höheres Abstraktionsniveau oft zu weniger effizientem Code führt – jedoch unabhängig vom Paradigma. Die Effizienz kommt hauptsächlich von der Wahl der richtigen Algorithmen unabhängig vom Abstraktionsniveau. Es stimmt auch, dass die Programmierung auf höherem Niveau leichter, also mit weniger Fachwissen machbar ist. Einige Sprachen auf hohem Niveau wurden speziell dafür geschaffen, auch Personen mit wenig Wissen darüber das Programmieren zu ermöglichen. Programme, die mit wenig Wissen erstellt werden, sind von Natur aus oft ineffizient und von schlechter Qualität. Schuld daran ist das mangelnde Wissen, nicht das Abstraktionsniveau.

Auf Objekte kommt es an: Heute wird fast nur mehr objektorientiert programmiert. Praktisch alle alten Programmiersprachen sind inzwischen in einer objektorientierten Variante verfügbar. Der Grund dafür besteht in den Vorteilen der objektorientierten Programmierung, vor allem hinsichtlich der Wartbarkeit von Programmen.

Auch hinter diesem Mythos steckt ein wahrer Kern. Es wird aber gerne übersehen, dass die objektorientierte Programmierung nur in einem bestimmten Bereich anderen Paradigmen gegenüber überlegen ist, nämlich für große, langlebige Programme. Für ein schnell erstelltes kleines Hilfsprogramm, das nur einmal zur Ausführung kommt, ist die objektorientierte Programmierung denkbar ungeeignet. Man muss viel in einen sauberen objektorientierten Stil investieren, um die Vorteile nutzen zu können. Bei einem kleinen, nur einmal verwendeten Programm zahlt sich dieser Aufwand niemals aus. Nicht jedes Programm in einer objektorientierten Sprache ist wirklich in einem objektorientierten Stil geschrieben. Rein prozedurale Programme in einer objektorientierten Sprache verzichten auf die Vorteile der Objektorientiertheit. Tatsächlich wird weit weniger objektorientiert programmiert als die Verwendung und Popularität von Programmiersprachen vermuten lässt. Im Bereich großer und langlebiger Programme haben sich objektorientierte Stile jedoch klar durchgesetzt.

Objektorientiertheit längst out: Die objektorientierte Programmierung hat die 90er-Jahre geprägt, ist inzwischen aber total veraltet. Heute haben wir viel bessere und coolere Programmierstile wie beispielsweise ...

Derartiges hört man immer wieder. Der wichtigste Grund besteht einfach darin, dass man ein bestimmtes Schlagwort (das die Punkte ersetzt) als neuen Programmierstil etablieren möchte. Einige dieser Schlagwörter hatten nur eine kurze Lebensdauer und waren bald veraltet. Andere haben sich länger gehalten und stellen heute etablierte Begriffe dar – etwa die aspektorientierte oder komponentenbasierte Programmierung. Wie weit sich z.B. Cloud-Computing durchsetzen wird, werden wir erst sehen. Bei keinem Begriff kann man aber sagen, dass die objektorientierte Programmierung dadurch verdrängt worden wäre. Ganz im Gegenteil. Die objektorientierte Programmierung bietet die Grundlage, auf der sehr viele neuere Konzepte und Ideen fußen. Sie entwickelt sich ständig weiter und greift neue Ideen auf. Natürlich kann es irgendwann so weit sein, dass der Name einer neuen Idee die objektorientierte Programmierung in den Hintergrund drängt, so wie die objektorientierte Programmierung die prozedurale Programmierung in den Hintergrund gedrängt hat. Das bedeutet aber nicht das Ende der objektorientierten Programmierung, sondern ein Weiterleben in neuer Form unter neuem Namen.

Funktionale Programmierung für Freaks: Die funktionale Programmierung scheidet die Geister. Es gibt einige „Spinner“, die die funktionale Programmierung extrem gut beherrschen und damit unglaubliche Sachen machen. Aber für normale Programmierer ist das nichts. Alleine schon deswegen, weil es keine Schleifen gibt und alles mit Rekursion gemacht wird, kann ein funktionales Programm niemals effizient sein.

So lautet ein weit verbreiteter Mythos. Abgesehen davon, dass die funktionale Programmierung viele Geister scheidet, ist an diesem Mythos kaum etwas Wahres. Es stimmt schon lange nicht mehr, dass Schleifen prinzipiell effizienter sind als Rekursion – weder hinsichtlich der Ausführungsgeschwindigkeit noch hinsichtlich der Verständlichkeit. In einem funktionalen Stil kann man die meisten Algorithmen viel einfacher und kürzer ausdrücken als in einem imperativen Stil. So rasch und einfach wie in modernen funktionalen Sprachen kann man in keinem anderen Paradigma einfache Programme entwickeln. Nur hinsichtlich Wartbarkeit großer Programme kommen funktionale Programme nicht ganz an objektorientierte Programme heran. Wegen dieser Vorteile ist die funktionale Programmierung nicht auf „Spinner“ beschränkt. Man kann auch in einer objektorientierten Sprache einen funktionalen Stil verwenden und damit das Beste aus beiden Welten kombinieren. Es ist kein Zufall, dass wichtige objektorientierte Sprachen in letzter Zeit um Sprachkonstrukte (z.B. Lambda-Ausdrücke) erweitert wurden um die funktionale Programmierung besser zu unterstützen. Allerdings kann man die Kombination der beiden Paradigmen nicht beliebig weit treiben. So ist die referenzielle Transparenz eine funktionale Eigenschaft, die in direktem Widerspruch zu zustandsbehafteten Objekten steht. Weiters ist die in modernen funktionalen Sprachen verwendete Typinferenz nicht beliebig mit Untertypbeziehungen kombinierbar.

Typüberprüfungen schützen vor Fehlern: Beim Programmieren passieren Fehler. Typüberprüfungen durch den Compiler können viele Fehler rasch entdecken. Nur so sind wir vor solchen Fehlern geschützt. Daher nehmen wir beim Programmieren einen Mehraufwand in Kauf und deklarieren Typen.

Leider ist auch das ein Mythos, der nur zu einem kleinen Teil zutrifft. Es stimmt zwar, dass der Compiler eine Klasse von Fehlern garantiert verhindern kann. Aber davon sind nur eher einfache Fehler betroffen,

die man zu einem guten Teil auch ohne Compiler recht rasch finden könnte. Die Vorteile starker Typisierung liegen eher in einem anderen Bereich: Durch die Festlegung von Typen erhöhen wir die Lesbarkeit von Programmen, da Typen eine ähnliche Rolle wie Kommentare spielen, aber dahingehend überprüft sind, dass sie zusammenpassen. Weiters unterstützen Typen eine statische Denkweise beim Programmieren. Die bessere Lesbarkeit und statische Denkweise kann Fehler verhindern, nicht die Typüberprüfungen selbst. Wenn man trotz Typen nur an den dynamischen Programmablauf denkt und unleserlichen Code schreibt, können Typüberprüfungen die Anzahl der Fehler kaum reduzieren.

Zukunft ist dynamisch: Die Deklaration von Typen ist überflüssig. Man verringert die Sicherheit vor Fehlern, weil man beim Programmieren und Testen weniger Vorsicht walten lässt. Außerdem schränkt man die Flexibilität unnötig ein. Daher ist es kein Zufall, dass in jüngster Zeit vermehrt dynamische Sprachen entwickelt werden. Die Zeit der stark typisierten Sprachen ist vorbei.

Auch dieser Mythos stimmt nicht ganz. Die Deklaration von Typen kann die Wahrscheinlichkeit von Fehlern durch bessere Lesbarkeit und statisches Denken reduzieren. Mit der Vorsicht beim Programmieren hat das wenig zu tun. In allen Programmierstilen kann man Vorsicht walten lassen oder auch nicht. Es stimmt jedoch, dass die Flexibilität eingeschränkt wird. Viele Einschränkungen werden bewusst gemacht um gefährliche Programmierstile zu verhindern. Andere Einschränkungen sind eine unerwünschte Konsequenz daraus. Aktuelle Sprachen sind trotz starker Typisierung recht flexibel, verlangen aber viel Wissen um die Flexibilität nutzen zu können. Das ist erwünscht, da Flexibilität ohne das nötige Wissen zu fehleranfälligen Programmen führt. Es stimmt auch, dass in letzter Zeit vermehrt dynamische Sprachen entwickelt werden. Dynamische Sprachen sind in einigen Bereichen vorteilhaft, beispielsweise beim Zusammenfügen bereits existierender Programme zu neuen, größeren Programmen – sogenannte Glue-Sprachen. Dieser Anwendungsbereich ist im Wachstum begriffen. Daneben besteht ein Gegentrend zu stark typisierten Java-ähnlichen Sprachen. Vor einigen Jahren hat man für vieles, was vorher mit dynamischen Sprachen gemacht wurde, plötzlich Java eingesetzt. Dafür ist Java nicht ideal. Jetzt sucht man wieder nach dynamischen Lösungen für diese Aufgaben.

Technologien sind entscheidend: Es kommt nicht so sehr auf das Paradigma an als darauf, ob für eine Sprache die benötigten Technologien zur Verfügung stehen. An eine Sprache kann man sich anpassen. Fehlende Technologien kann man aber nur schwer ersetzen.

In diesem Mythos steckt viel Wahrheit. Nicht selten ist die Unterstützung einer bestimmten, von vielen Leuten gewünschten Technologie der Grund dafür, dass sich eine Sprache in der Praxis durchsetzt. Beispielsweise ist *Ruby on Rails*, ein Framework für die effiziente Erstellung von Web-Applikationen der wichtigste Grund dafür, dass sich die Programmiersprache *Ruby* so rasch durchgesetzt hat. Unzählige mit Java zusammenhängende Technologien führen dazu, dass Java nur schwer zu ersetzen ist. Andererseits besteht eine große Gefahr in der Abhängigkeit von bestimmten Technologien. Man muss in der Informatik klar zwischen einer Technik und einer Technologie unterscheiden: Eine *Technik* ist eine bestimmte Vorgehensweise zur Erreichung eines Ziels (unter Verwendung technischer Hilfsmittel). Beispielsweise garantiert man die Termination einer Schleife, indem man jeder Iteration eine Zahl zuordnet, die mit jeder Iteration strikt kleiner wird, aber nie unter 0 kommen kann. Das ist eine Technik. Techniken kann man erlernen oder erfinden, aber nicht kaufen. Eine *Technologie* ist dagegen ein ins eigene Programm einbindbarer Code oder ein Werkzeug, das bei der Erfüllung einer Aufgabe hilft. Man muss zwar auch die Anwendung einer Technologie erlernen, aber man muss zusätzlich den Code oder das Werkzeug haben, also meist käuflich erwerben. Zur Entwicklung einer Technologie reicht es nicht, nur etwas zu erfinden, man muss die Erfindung auch in ein Programmstück umsetzen. Die Problematik der Abhängigkeit von einer Technologie besteht darin, dass sie altert. Wenn der Code oder das Werkzeug nicht mehr gewartet wird oder sich so ändert, dass er oder es den Erwartungen nicht mehr entspricht, muss man auf eine andere Technologie umsteigen. Es kann sehr schwer sein, eine andere passende Technologie aufzutreiben. Man soll eine Technologie also nur einsetzen, wenn tiefes Vertrauen in die Technologie besteht und es keine andere einfache Möglichkeit gibt, das Gewünschte zu erreichen. Techniken sind und bleiben dagegen immer einsetzbar.

Diese Liste könnte man endlos fortsetzen. Fast alles, was man über Paradigmen hört oder liest, muss hinterfragt werden, weil die Aussagen so allgemein sind, dass sie kaum in jedem Fall zutreffen werden.

6.6.2 Mythen in Java

Portabilität: Java verwendet JVM-Code als Zwischencode um sicherzustellen, dass Java-Anwendungen auf jedem System lauffähig ist, das Java unterstützt. Auf diesem Gebiet spielt Java einer Vorreiterrolle.

Der Mythos ist zum Teil richtig. Durch Verwendung des Zwischen-codes erreicht Java tatsächlich einen hohen Grad an Portabilität, da für die Ausführung neben dem Zwischencode nur ein Java-Interpreter nötig ist. Zwischencode alleine reicht zur Erreichung hoher Portabilität jedoch nicht aus. Viele Anwendungen verlangen, dass neben dem Java-Interpreter am System auch Klassen-Bibliotheken in einer bestimmten Version vorinstalliert sind. Vor allem auf kleinen Systemen sind nicht immer alle Standard-Bibliotheken in der neuesten Version vorhanden. Dies schränkt die Portabilität wieder ein. Zwischencode war schon lange vor Java auf vielen Systemen in Verwendung. Die Vorreiterrolle von Java besteht darin, dass eine ganze Reihe an Maßnahmen gesetzt wurde, um den JVM-Code über einen sehr langen Zeitraum stabil und damit portabel zu halten.

Sichere Sprache: Java ist eine sichere Sprache. Bei der Entwicklung von Java wurde darauf geachtet, dass bereits der Compiler alles überprüft, was überprüfbar ist, und zur Laufzeit noch einmal alles überprüft wird, damit auch Fehler des Compilers oder Manipulationen des Zwischen-codes keine schwerwiegenden Auswirkungen haben. Daher ist es kaum möglich, über Java in ein System einzudringen.

Es stimmt, dass die Java-Entwickler mehr für die Sicherheit der Sprache unternommen haben als die Entwickler manch anderer Sprachen und Programmiersysteme. Aber hundertprozentige Sicherheit kann Java nicht garantieren. Auch wenn Fehler des Compilers ausgeschaltet sind, kann man immer noch über Fehler des Interpreters in ein System eindringen. Zusätzliche Überprüfungen zur Laufzeit erhöhen die Ausführungszeiten von Programmen. Prinzipiell kann ein Java-Interpreter zwar vieles überprüfen, aber einige Überprüfungen (die unter der Annahme eines korrekten Compilers unnötig sind) bleiben aus Effizienzgründen meist abgeschaltet. Weil man weiß, dass man über Compiler und Interpreter nur einen bestimmten Grad an Sicherheit erreichen kann, setzt man heute fast durchwegs zusätzliche Maßnahmen zur Steigerung der Sicherheit ein. Dazu zählen beispielsweise Viren-Checker sowie die signierte Übertragung von Programmdateien.

Bei der Konstruktion von Programmen greift man auf die Unterstützung durch formale Methoden oder Laufzeitüberprüfungen im Programm zurück, die gelegentlich weit über das hinausgehen, was von Java normalerweise überprüft wird. So kann man beim Programmieren (durch verstärktes Problembewusstsein, etwa durch Validierung aller von außerhalb stammenden Daten) oft mehr an Sicherheit erreichen, als wenn man sich auf die Sicherheit einer Sprache verlässt.

Fallen beseitigt: Java hat aus älteren objektorientierten Sprachen wie C++ gelernt und die wichtigsten Fallen, die immer wieder zu Fehlern geführt haben, beseitigt. Aus diesem Grund hat man typische Fehlerquellen wie das Überladen von Operatoren oder die Mehrfachvererbung beseitigt und eine bessere Unterstützung für den Umgang mit Ausnahmen eingeführt.

So oder ähnlich lauten übliche Werbeaussagen aus der Zeit, in der Java noch jung war. Tatsächlich stimmen einige damalige Aussagen aus heutiger Sicht nicht mehr ganz. Beispielsweise ist es richtig, dass Mehrfachvererbung in C++ vor langer Zeit oft ganz falsch verwendet wurde und sich deswegen als Falle erwiesen hat. Aber die Vererbungskonzepte haben sich mittlerweile genauso weiterentwickelt wie das Wissen um den optimalen Einsatz dieser Konzepte. Viele Programmierer würden heute lieber Mehrfachvererbung in Java haben als die (aus mancher Sicht nicht ganz geglückte) Trennung zwischen Interfaces mit Mehrfachvererbung und Klassen mit Einfachvererbung. Überladene Operatoren gibt es in Java ohnehin, beispielsweise + für die ganzzahlige Addition, die Fließkomma-Addition und die Verkettung von Strings. Aber Programmierer können nicht, wie in anderen Sprachen, selbst Operatoren überladen. Das ist gelegentlich ein Vorteil, manchmal aber auch ein Nachteil. Unter der besseren Unterstützung von Ausnahmen versteht man die Notwendigkeit von `throws`-Klauseln in Methoden-Köpfen sowie die `finally`-Blöcke. Neuere Sprachen, die aus Java gelernt haben, verwenden meist jedoch keine `throws`-Klauseln mehr, weil sie teilweise zu einem unerwünschten und zweckentfremdeten häufigen Einsatz von Ausnahmebehandlungen führen. Auch einige Details von `finally`-Blöcken haben sich als nicht optimal erwiesen. Das soll nicht als Kritik an Java verstanden werden, sondern spiegelt eine allgemeine Erkenntnis wider: Sprachen entwickeln sich weiter. Etwas, das zu Beginn als große Neuerung gepriesen wird, stellt sich später als doch nicht ganz so optimal her-

aus. Wer später kommt, kann von den Vorgängern lernen. Aber ohne große Neuerungen kann sich keine Sprache langfristig durchsetzen, weil gerade diese Neuerungen den Charakter der Sprache bestimmen, obwohl sie meist nicht so wertvoll sind, wie anfangs vermutet.

Internet-Sprache: Java ist die Sprache des Internet und des Web.

Eigentlich hat Java nichts mit dem Web zu tun, außer dass das Web und Java etwa zur selben Zeit groß geworden sind. Wegen des zeitlichen Zusammentreffens haben die Java-Entwickler versucht, sich an den Trend zum Web anzuheften – beispielsweise durch Java-Beans, die aber nie besonders häufig eingesetzt wurden. Wesentlich häufiger wird im Web JavaScript eingesetzt, eine dynamische Sprache, die (abgesehen vom Namen) nichts mit Java zu tun hat. Erst in jüngerer Zeit konnte Java im Internet in bedeutenderem Ausmaß Fuß fassen, vor allem durch JSF2, einem Framework-Standard zur Entwicklung grafischer Benutzeroberflächen für Webapplikationen. Der Grund für die Verwendung von Java in diesem Bereich ist vermutlich nur die allgemeine Popularität von Java, nicht irgendeine spezielle Spracheigenschaft.

C viel effizienter als Java: Objektorientierte Sprachen können durch dynamisches Binden generell nie so effizient sein wie konventionelle imperative Sprachen. Vor allem C ist die wichtigste Sprache, wenn es auf Effizienz ankommt, da es kein dynamisches Binden gibt und man sehr nah an der Hardware programmieren kann. Eventuell kommt noch C++ in Frage, wenn man auf dynamisches Binden verzichtet. Aber Java hat keine Chance, hinsichtlich Effizienz auch nur in die Nähe von C und C++ zu kommen.

Dieser Mythos wird vor allem bei Elektrotechnikern, Physikern und teilweise auch technischen Informatikern als unumstößliche Wahrheit angesehen. Auch bei anderen Informatikern ist er weit verbreitet, obwohl eine wichtige Aussage darin sehr wahrscheinlich nicht zutrifft: Dynamisches Binden kann effizienter sein als beispielsweise eine Mehrfachverzweigung durch eine `switch`-Anweisung in C oder C++. Mit dynamischem Binden können sich also effizientere Programme ergeben als ohne. Programme in C und C++ werden heute meist mit demselben Compiler übersetzt, sodass einander entsprechende Programme auch gleich effizient sind. Es gibt auch zahlreiche Vergleiche hinsichtlich der Effizienz einander ähnlicher C++

und Java-Programme. Meist ist dabei C++ tatsächlich etwas effizienter, aber nur minimal – etwa innerhalb von 20%. Solche kleinen Unterschiede sind kaum nennenswert. Trotzdem steckt hinter dem Mythos mehr Wahrheit, als diese Vergleiche erahnen lassen: In den Vergleichen werden ja nur einander ähnliche Programme betrachtet. Tatsächlich legen typische C-Programmierer wesentlich mehr Augenmerk auf die Effizienz als Java-Programmierer, und diese Unterschiede können sich drastisch auswirken. So brauchen typische C++ und Java-Programme (bei einem vergleichbaren objektorientierten Programmierstil) oft vierzig Mal so lange als ein auf Effizienz getrimmtes C-Programm, das dieselbe Aufgabe erledigt. Der Unterschied liegt also nicht in den Sprachen und Compilern, sondern im Programmierstil. Wer will könnte auch in Java effizient ausführbaren Code schreiben. Bei der Programmierung in Java legt man aber mehr Wert auf die effiziente Entwicklung (also das rasche Erstellen des Codes) und die gute Wartbarkeit und verzichtet dabei auf Ausführungseffizienz.

Java ist ein alter Dinosaurier: Java stammt ungefähr aus der Mitte der 90er-Jahre und hat sich seither nur sehr langsam verändert. Andere Sprachen entwickeln sich wesentlich dynamischer weiter und bieten daher bereits viel fortschrittlichere Konzepte. In naher Zukunft wird Java so veraltet sein, dass es zum Aussterben verurteilt ist.

Wie Programme haben auch Programmiersprachen einen bestimmten Lebenszyklus. Bei erfolgreichen Sprachen kommt nach einer eher langsamen Einführungsphase ein steiler Aufstieg gefolgt von einem langsamen Abstieg. Java steht eher am Beginn der Abstiegsphase. Wenn sich Java so wie üblich entwickelt, dann wird die Sprache noch lange existieren, auch wenn die Bedeutung abnimmt. Die geringe Dynamik lässt sich vor allem durch den Erfolg leicht erklären: Aufgrund des intensiven Einsatzes und der großen Anzahl an Java-Programmierern würden starke Änderungen einfach nicht akzeptiert werden. Anders verhält es sich mit nicht ganz so erfolgreichen, aber dennoch gut unterstützten Sprachen: Anpassungen an neue Gegebenheiten müssen rasch erfolgen, damit die Sprache an Bedeutung gewinnen kann, auch wenn dadurch einige etablierte Programmierer verschreckt werden. An älteren Sprachen (etwa C++ oder Smalltalk) sieht man, wie nach einer längeren Abschwungphase oft wieder mehr Dynamik in die Entwicklung kommt, die zu einem Wiederaufleben führt. Das hat eben damit zu tun, dass ein etwas geringerer Erfolg die

Dynamik fördert. Vermutlich wird die Dynamik in der Entwicklung von Java irgendwann zunehmen und der Sprache damit ein langes Leben garantieren. Aber irgendwann wird auch Java überholt sein und durch eine andere Sprache ersetzt werden. Welche Art von Sprache das sein könnte, zeichnet sich noch nicht ab.

Auch diese Liste kann man beliebig lang fortsetzen. Generell muss man bei Mythen Vorsicht walten lassen. Nicht alles, was einer gängigen Meinung entspricht, trifft auch wirklich zu. Hinter fast jedem Mythos steckt ein Körnchen Wahrheit, aber auch viel Dichtung. Wenn man die Wahrheit wissen möchte, muss man viel tiefer blicken und sich sehr eingehend mit einer Frage beschäftigen. Rasche Antworten greifen meist zu kurz.

Ein Appell am Ende: Man darf nicht alles glauben, was man liest oder hört, sondern muss sich stets ein eigenes Bild machen und eine eigene Meinung bilden. Das gilt auch im Bereich der Programmierung. Zu viele falsche Propheten verbreiten Meinungen, auf die man besser nicht hören sollte. Sogar Vorbilder, auf die man normalerweise vertrauen kann, irren sich gelegentlich. Das einzige, worauf man sich verlassen sollte, ist die eigene Erfahrung und das eigene Wissen – nicht zu verwechseln mit Mythen, die man im Laufe der Zeit aufgesammelt hat. Wenn man einer Sache nicht sicher ist, probiert man sie einfach aus, sofern das möglich ist.

Man darf auch diesem Skriptum nicht trauen. Darin sind sicherlich viele Fehler enthalten. Am besten probiert man alle Beispielprogramme aus und versucht sie zu verbessern. Der Stichhaltigkeit der Argumente sollte man sich selbst vergewissern und die Argumente zu widerlegen versuchen. So eignet man sich echtes Wissen an – im Gegensatz zu Mythen, die man einfach unreflektiert übernimmt und nacherzählt. Echtes Wissen anzusammeln ist nicht der einfachste Weg. Aber einfachere Wege stellen sich oft als sehr lang heraus und führen nicht selten am Ziel vorbei.

6.7 Fallen umgehen lernen

Man spricht genau deshalb von Fallen, weil man mit hoher Wahrscheinlichkeit hineinfällt. Kein Trick lässt uns Fallen rechtzeitig erkennen. Einzig und alleine Wissen und Erfahrung helfen dabei. Es ist keine Schande, in eine Falle zu tappen, solange man daraus etwas lernt. Man muss in viele Fallen getappt sein, bevor man die gängigsten Fallen sicher erkennt. Der wichtigste Ratschlag besteht wieder einmal einfach darin, viel zu Program-

mieren und sich von möglichen Fallen nicht abhalten zu lassen. Jede Falle, in die man überwindet, verbessert die eigenen Fähigkeiten.

6.7.1 Kontrollfragen

- Welche beiden grundlegenden Speicherbereiche werden in Java (und fast allen anderen Programmiersprachen) unterschieden, und welche Daten liegen in diesen Speicherbereichen?
- Wozu dient Garbage Collection und wie erledigt ein Garbage Collector seine Aufgabe?
- Wie kann man beim Programmieren den Garbage Collector unterstützen?
- Welche Fallen bestehen bei Garbage Collection?
- Wie kann man beim Programmieren die Garbage Collection beeinflussen?
- Bei einem `StackOverflowError` kann man den Stack zu vergrößern versuchen. Warum hat man damit nur selten Erfolg?
- Wozu dient die Methode `finalize`, und warum wird sie nur selten verwendet?
- Wie verwendet man eine Free List?
- Welche Arten von Streams können wir in Java unterscheiden?
- Wozu benötigt man im Zusammenhang mit Streams die Methode `flush`?
- Was kann passieren, wenn mehrfach von derselben Datei gelesen bzw. auf dieselbe Datei geschrieben wird?
- Welche Fehler passieren leicht beim Umgang mit Dateien?
- Was ist eine Zeichen-Codierung?
- Wozu dienen Lock-Dateien?
- Was versteht man unter einer Antwortzeit?
- Wodurch kann die Antwortzeit stärker als erwartet erhöht werden?

- Was bedeutet Busy Waiting?
- Welche Ansätze gibt es, um Schäden durch versteckte Aktivitäten von Programmen gering zu halten?
- Welche Fallen lauern typischerweise beim Rechnen mit ganzen Zahlen?
- Was ist ein Überlauf oder Unterlauf?
- Wann müssen wir `BigInteger` statt `int` oder `long` einsetzen?
- Welche Arten von Problemen bei nicht abschätzbar großen Zahlen kann auch `BigInteger` nicht vermeiden?
- Warum ist es meist keine gute Idee, ganze Zahlen durch Fließkommazahlen zu ersetzen, wenn der Wertebereich der ganzen Zahlen möglicherweise nicht ausreicht?
- Wieso ist es auch bei ganzen Zahlen wichtig, klar zwischen `equals` und `==` zu unterscheiden?
- Wofür verwendet man `BigDecimal`?
- Welche Schwierigkeiten treten beim Rechnen mit Geldbeträgen auf?
- Was ist eine Auslöschung, und welche Algorithmen und Probleme sind (im Zusammenhang mit Fließkommazahlen) gut bzw. schlecht konditioniert?
- Wie entsteht `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` und `NaN`?
- Ist `0.0` dasselbe wie `-0.0`? Was ergibt `0.0 == -0.0`?
- Warum sollte man Fließkommazahlen weder mittels `==` noch mittels `equals` vergleichen?
- Wie kann Absorption bei Fließkommaberechnungen zu einem Problem werden?
- Welche Fallen lauern im Umgang mit `null`?
- Welche Vorteile dürfen wir uns dadurch erhoffen, dass wir die Verwendung von `null` auf das unbedingt nötige Ausmaß reduzieren (Beispiel)?

- Was sind off-by-one-Fehler, und wodurch entstehen sie?
- Wie kann man off-by-one-Fehler vermeiden oder erkennen?
- Wie kann man durch Ausnutzen eines schlecht überprüften Randbereichs einen Computer angreifen bzw. in ihn eindringen?
- Was sind Pufferüberläufe, warum stellen sie eine große Gefahr dar, und was kann man dagegen tun?
- Wodurch unterscheidet sich die Parallelität von der Nebenläufigkeit?
- Welche Unterschiede gibt es zwischen Multiprocessing und Multithreading?
- Was versteht man unter dem Aufspannen eines Threads?
- Was ist eine Race Condition?
- Was passiert bei der Synchronisation und wozu braucht man Synchronisation?
- Warum spielen atomare Aktionen bei der nebenläufigen Programmierung eine wichtige Rolle?
- Wozu verwendet man `wait` und `notify` in Java?
- Wodurch können sich nebenläufige Threads gegenseitig behindern (Liveness Properties)?
- Warum sollen synchronisierte Methoden nur kurz laufen?
- Welches Ziel verfolgt die strukturierte Programmierung?
- Welche Strukturen setzt man in der strukturierten Programmierung ein?
- Was ist schlecht an Goto, Fall-through, Break, Continue, Return, Ausnahmen und Dangling-else?
- Welche typischen Fallen lauern in der objektorientierten Programmierung?
- Welche Fallen sind typisch für Java?

6 Vorsicht: Fallen!

- Wie unterscheidet sich die defensive von der offensiven Programmierung?
- In welchen Zusammenhängen ist ein defensiver Programmierstil nötig?
- Warum sind Überprüfungen mancher Bedingungen (bei einem defensiven Programmierstil) im Zusammenhang mit Zusicherungen oft schwierig bzw. verzichtbar?
- Warum kann ohne Vertrauen keine gute Software entstehen?
- Welche Aspekte sind zur Gewinnung von Vertrauen in Programmcode wichtig?
- Wie kann Misstrauen im Team zum Scheitern von Softwareprojekten führen?
- Wozu dienen Teamregeln?
- Warum ist es notwendig, ein Gespür für den Wahrheitsgehalt von Mythen zu entwickeln?
- Zählen Sie typische Mythen im Bereich der Programmierparadigmen und von Java auf und analysieren Sie deren Wahrheitsgehalt.