FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

PK11W–2

Zweiter Teil des Skriptums zu

PROGRAMMKONSTRUKTION

185.A02 Grundlagen der Programmkonstruktion
183.592 Programmierpraxis

im Wintersemester 2011/2012

Franz Puntigam
Michael Reiter
Andreas Krall

Fakultät für Informatik
Technische Universität Wien

Inhaltsverzeichnis

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 Aufmerksam 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 Urache 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 `-Xssn1m` 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 demostrieren 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 exisierende 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 dazugehörenden 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()` and 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-Tase) 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 abfrägt; 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 Virenscanner 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 Situtationen 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 Zahlenliterale 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 Zahlenliterale 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 reelen 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 hinausschieben, 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-

[1] Im englischsprachigen Raum und in Programmen wird statt Komma „,“ ein Punkt „.“ verwendet.

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 ±4,9e−324 bis ±1,7e+308 für `double` und ±1,4e−45 bis ±3,4e+38 für `float` reichen meist aus. Die beiden Referenztypen `Double` und `Float` entsprechen hinsichtlich der Wertebereiche und Genauigkeiten ihren primitiven Gegenstücken. 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*

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 Operation ergeben Infinity, −Infinity oder NaN wenn ein Operand Infinity oder −Infinity ist, und alle Oerationen 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 Defaultwert[2] 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

[2] Unter einem *Defaultwert* oder kurz *Default* versteht man ganz allgemein einen Wert, der dann zu verwenden ist, wenn kein anderer Wert explizit vorgegeben ist.

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

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 verharrt in abstraktem Denken, und als konrete 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 Angreifern bevorzugte Möglichkeit 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 verwechset 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.

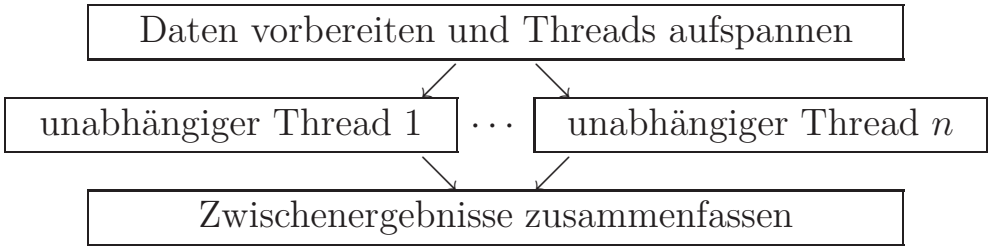
- Ein Thread kann `x` mit `y` gerade in dem Augenblick vergleichen, in dem ein anderer Thread `x` schon erhöht hat, `y` aber noch nicht.

- 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 Schleifenrümpfen, 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 einstehen, 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 Detailliertheitsgrad 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<B>` auch dann kein Untertyp von `Liste<A>` wenn `B` ein Untertyp von `A` ist. Man könnte sonst etwa in ein Objekt vom Typ `Liste<B>` ein Objekt vom Typ `A` einfügen, das keine Instanz von `B` ist. Implizite Unterypbeziehungen 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 ausschaut, 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. Augrund 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 Situation ist eine gesunde Portion an Misstrauen angebracht, die sich darin äußert, dass man Daten oder Programmteile ignoriert oder nocheinmal 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 Überprü-

fungen. 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 nocheinmal, 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 nocheinmal 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 genausogut 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 Programmteils 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 Einahltung 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 aufführen.

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, vieleicht 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 Abstraktionsneveau 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 Zwischencodes 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-Bibliothen 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 nocheinmal alles überprüft wird, damit auch Fehler des Compilers oder Manipulationen des Zwischencodes 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 Programmdaten.

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 mittlerweise 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 Java-Script 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?

- 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.