TU WIEN Informatics

Franz Puntigam

Einführung in die Programmierung 2

Skriptum zur Lehrveranstaltung 185.A92 an der TU Wien (2) 2025

Inhaltsverzeichnis

Vorwort

Einführung in die Programmierung 2 ist eine Lehrveranstaltung für alle Studierenden der Informatik und Wirtschaftsinformatik an der TU Wien. Das vorliegende Skriptum erfüllt mehrere Aufgaben:

- Vor Beginn der Lehrveranstaltung bietet es eine Vorschau auf das Kommende. Kapitel 1 gibt einen Überblick über vorausgesetzte und zu erwerbende Kompetenzen sowie Inhalte und Aufbau der Lehrveranstaltung. Unentschlossenen sei insbesondere Abschnitt 1.3.3 empfohlen.
- Begleitend zur Lehrveranstaltung dient es als
  - Lehrbuch mit Programmbeispielen und Details zu Inhalten der Lehrveranstaltung,
  - Sammlung von Aufgaben (Kapitel 2 bis 4), die das Erreichen der Lernziele unterstützen,
  - Hilfsmittel beim Lösen von Aufgabenblättern und zur Vorbereitung auf geleitete Übungen und Tests.

Viel Spaß beim Lesen und Lernen!

1 Überblick

Einführung in die Programmierung 2 (kurz EP2) soll erst nach erfolgreicher Teilnahme an *Einführung in die Programmierung 1* (EP1) absolviert werden. Dementsprechend werden in EP2 Kompetenzen vorausgesetzt, die zu den Lernergebnissen von EP1 gehören. Dennoch wollen wir uns zunächst einige Begriffe und Konzepte in Erinnerung rufen um einen Überblick über die schon bekannten Themenbereiche zu erhalten – eine Bestandsaufnahme der vorausgesetzten Kompetenzen. Danach folgt ein Überblick über Themenbereiche, die in EP1 noch nicht ausreichend behandelt wurden. Viele davon bilden inhaltliche Schwerpunkte von EP2. Nebenbei wird die Struktur dieses Skriptums mit den darin verwendeten Symbolen beschrieben. Eine Auflistung der Kompetenzen, die in EP2 erworben werden sollen, sowie Hinweise auf die empfohlene Verwendung des Skriptums und Vorgehensweisen beim Lernen schließen dieses Kapitel ab.

1.1 Erworbene Kompetenzen

Von uns, die als Studierende an EP2 teilnehmen, wird erwartet, schon zu Beginn des Semesters die in diesem Abschnitt beschriebenen Kompetenzen mitzubringen. Die Verwendung des Wortes *Kompetenz* deutet an, dass es im Allgemeinen nicht reicht, nur die Definitionen oder Bedeutungen der verwendeten Begriffe wiedergeben zu können. Vielmehr müssen wir in der Lage sein, bestimmte natürlichsprachig beschriebene Aufgaben selbständig in ausführbare Programme mit vorgegebenen Eigenschaften umzusetzen, Programmtexte nachzuvollziehen, Eigenschaften von Programmen herauszufinden und so weiter.

praktische Kompetenzen

1.1.1 Grundlegende Programmierkonzepte

Zu den wichtigsten Voraussetzungen zählt tiefgehendes Wissen über grundlegende Konzepte einer Programmiersprache. In EP2 verwenden wir ausschließlich Java. Unter grundlegenden Konzepten verstehen wir

einfache imperative Sprachkonzepte (wie Variablen, Typen, Operatoren, diverse Programmverzweigungen und Schleifen) aber auch manchmal fordernde Konzepte wie Methoden, rekursive Methodenaufrufe und ein- sowie mehrdimensionale Arrays.

Ausdruck

Ausdrücke, die zu *Werten* ausgewertet werden können, sind wesentliche Bestandteile fast aller Programmiersprachen.

Literal

Literale sind Ausdrücke, die Werte direkt darstellen, z. B. `1`, `2.3`, `'a'` und `"A"`. Durch Anwendung von (in Java stets vordefinierten) *Operatoren* werden Ausdrücke zu komplexeren Ausdrücken kombiniert, wie in `1 + 2`, wobei `+` ein Operator ist und `1` und `2` *Operanden* sind.

Variable

Variablen enthalten Werte. Wir unterscheiden zwischen lesenden und schreibenden Variablenzugriffen, da Schreibzugriffe unabhängig vom alten Wert in die Speicheradresse der Variablen schreiben, während Lesezugriffe unabhängig von der Adresse den Wert der Variablen zurückgeben. *Zuweisungen* erfolgen über Ausdrücke wie `x = y`, wobei `x` geschrieben und `y` gelesen wird. Operanden sind je nach Operator und Position *L-Werte,* die wie links von `=` eine beschreibbare Adresse darstellen, oder *R-Werte,* also normale Werte, die rechts von `=` stehen. Manche Operanden, etwa die von `++`, müssen sowohl als L-Werte als auch als R-Werte verwendbar sein. Unmittelbar nach der lokalen *Variablendeklaration* `int x, y=0;` sind beispielsweise die Ausdrücke `x = y`, `x = y + 1` und `y++` erlaubt, jedoch nicht `y + 1 = y`, `1 = 2`, `y = x` und `x++` (weil `x` an dieser Stelle noch nicht initialisiert ist).

Typ

Jedes Literal hat einen vorgegebenen *deklarierten Typ* und jede Variable einen im Programm deklarierten Typ; jeder Operator ist auf bestimmte Typen anwendbar und liefert ein Ergebnis vom entsprechenden Typ. So hat auch jeder Ausdruck einen eindeutigen deklarierten Typ. Typen können die Lesbarkeit eines Programms deutlich erhöhen. Allerdings können implizite Umformungen von Werten eines Typs in Werte eines anderen Typs auch leicht zu Missverständnissen führen.

Seiteneffekt

Die Ausführung mancher Ausdrücke, insbesondere von *destruktiven Zuweisungen,* hat *Seiteneffekte,* die Programmzustände verändern. Eine Zuweisung ist destruktiv, wenn sie einen alten Wert in einer Variablen überschreibt – im Gegensatz zu einer Zuweisung, die eine Variable *initialisiert,* ihr also erstmalig einen Wert zuweist. Seiteneffekte bilden die Grundlage für *Anweisungen* im Programm, die Programmzustände verändern ohne Ergebnisse zu liefern. Nacheinander stehende Anweisungen werden nacheinander, also *sequenziell* ausgeführt. Es gibt eine Reihe von *Kontrollstrukturen,* die andere Formen der Ausführung unterstützen, vor allem verschiedene Arten von *Schleifen* für die *wiederholte Ausführung* (wobei jeder Schleifendurchlauf *Iteration* heißt) und *Programmverzweigungen* für die Ausführung eines von mehreren möglichen Programmzweigen. In Java kommen als Schleifen vorwiegend `while`- und `for`-Schleifen einschließlich der For-Each-Schleifenvariante sowie als bedingte Anweisungen `if`-Anweisungen zum Einsatz.

Anweisung

Kontrollfluss

Der *Kontrollfluss* eines Programms bestimmt die Ausführungsreihenfolge von Anweisungen und Ausdrücken. Kombinationen aus Sequenzen, Wiederholungen und Verzweigungen reichen aus um alle in der Praxis nötigen Kontrollflüsse darzustellen. Der *Datenfluss* in einem Programm ist implizit über das Schreiben und Lesen derselben Variablen festgelegt. Kontrollfluss und Datenfluss zusammen bestimmen, was ein Programm macht. Häufig geben wir den Kontrollfluss auf der Ebene von Anweisungen durch Kontrollstrukturen vor. Aber mithilfe rekursiver Methoden ist jeder sinnvolle Kontrollfluss auch ohne Seiteneffekte ausschließlich auf der Ebene von Ausdrücken darstellbar. Der Verzicht auf nach außen sichtbare Seiteneffekte hat den Vorteil, dass sich ein vereinfachter, leicht nachvollziehbarer Datenfluss ergibt.

Datenfluss

Methode

Eine *Methode* fasst Anweisungen zu einer Einheit zusammen, die beliebig oft durch *Aufruf* über den Methodennamen zur Ausführung gebracht wird. In einem Aufruf können der aufgerufenen Methode *Argumente,* auch *aktuelle Parameter* genannt, übergeben werden, die innerhalb der Methode als Inhalte *formaler Parameter* (kurz *Parameter*) ähnlich zu Variablen zugreifbar sind. Eine Methode kann am Ende der Methodenausführung ein *Ergebnis* zurückgeben. Der Aufruf einer Methode ist ein Ausdruck, der zum Methodenergebnis ausgewertet wird.

Rekursion

Lokale Variablen und formale Parameter jeder Methodenausführung liegen in einem eigenen *Stack-Frame,* sodass sie verschieden von lokalen Variablen und formalen Parametern anderer Methodenausführungen sind. Dadurch kann eine Methode *rekursiv,* also durch nochmaligen Aufruf der gerade ausgeführten Methode (mehrfach) überlappend ausgeführt werden, ohne dass die Ausführungen einander behindern. Rekursive Methoden können Schleifen ersetzen und bei mehrfach rekursiven Aufrufen auch komplexe Kontrollflüsse entstehen lassen.

Termination

Bei Rekursion über Methoden kann es genau so wie bei Iterationen über Schleifen passieren, dass die Berechnungen (zumindest konzeptionell) nicht *terminieren,* also zu keinem Ende kommen. Damit das nicht passiert, müssen wir für *Fundiertheit* und *Fortschritt* sorgen. Das bedeutet, jeder Rekursionsschritt und jede Iteration muss die Berechnung um einen bestimmten Mindestbetrag näher an die Erfüllung einer erreichbaren *Abbruchbedingung* heranführen.

Referenz

Der Umgang mit Werten *elementarer Typen* wie `int` und `char` ist einfach, aber zur Lösung komplexer Aufgaben oft nicht ausreichend. Daher gibt es neben elementaren Typen auch *Referenztypen,* deren Werte auch als *Objekte* bezeichnet werden. Beispiele dafür sind Zeichenketten vom Typ `String` sowie *Arrays* von verschiedenen Typen. Während eine Zeichenkette eine unveränderliche Folge von Zeichen darstellt, ist ein Array eine Folge von Werten, den *Arrayeinträgen,* die jeweils über einen Index angesprochen und durch einen neuen Eintrag ersetzt werden können. Arrayeinträge können wiederum Arrays sein, wodurch sich mehrdimensionale Arrays ergeben. Arrays dienen beispielsweise als Zwischenablage für Werte, die während der Programmausführung entstehen und später wieder verwendet werden. Anders als bei der Ablage von Werten in einfachen Variablen muss die Anzahl der benötigten Werte nicht schon im Programm statisch festgelegt sein, sondern ist während der Programmausführung bestimmbar.

null

Jede Variable und jeder Arrayeintrag eines Referenztyps kann statt eines Objekts dieses Typs auch `null` enthalten. Beim Umgang mit Referenztypen sind häufige Überprüfungen auf `null` etwa durch Anweisungen wie `if (x == null) { ... } else { ... }` kaum vermeidbar. Außerdem ist streng zwischen der Identität und Gleichheit von Objekten zu unterscheiden. Während der Identitätsvergleich `x==y` feststellt, ob die Variablen `x` und `y` dasselbe Objekt referenzieren, prüft die Methode `x.equals(y)` auf Gleichheit. Identische Objekte bleiben stets identisch und gleich, während die Gleichheit zweier Objekte durch Änderung eines der beiden Objekte verloren gehen oder auch entstehen kann. Sind `x` und `y` identisch, so wirkt sich jede Änderung von `x` gleichzeitig auch auf `y` aus. Dadurch kann eine Zustandsänderung an einer Stelle im Programm gleichzeitig an einer ganz anderen Programmstelle sichtbar werden. Referenztypen sind daher ein sehr mächtiges Werkzeug in der Programmierung, das jedoch, wenn falsch verstanden, gefährlich sein kann.

1.1.2 Vorgehensweisen beim Programmieren

Das Verstehen der Konzepte einer Programmiersprache ist eine Sache, der praktische Umgang damit eine andere. Wir sind damit konfrontiert, eine in einer natürlichen Sprache beschriebene Programmieraufgabe innerhalb eines vorgegebenen Zeitrahmens in ein Programm oder einen Programmteil umsetzen zu müssen. Häufig sind zusätzliche Vorgaben einzuhalten, etwa die Aufgabe ohne Verwendung von Schleifen zu lösen.

Wir könnten solange verschiedene Programmtexte ausprobieren, bis die Aufgabe gelöst zu sein scheint. Allerdings würden wir damit den Zeitrahmen kaum einhalten können. Gezielte Vorgehensweisen sind nötig.

unbenannte Muster

In EP1 haben wir wichtige Schritte solcher Vorgehensweisen wiederholt gesehen und praktisch angewandt, meist jedoch ohne sie zu benennen oder genauer zu untersuchen. Daher fehlt uns noch die Terminologie in diesem Bereich. Dennoch wollen wir versuchen, die uns schon bekannten Schritte kurz zusammenzufassen.

Anforderungen verstehen

Am Beginn steht die *Anforderungsanalyse,* das heißt, wir müssen herausfinden, was zu tun ist. Das ist einer der schwierigsten Schritte, weil jede natürlichsprachige Beschreibung einen Interpretationsspielraum lässt, die Umsetzung in ein Programm dagegen präzise sein muss. Wir brauchen Erfahrung, die vom Machen kommt. Beschreibungen enthalten wiederholt einander ähnliche Muster, an denen wir uns orientieren. Die ersten Aufgaben waren recht detailreich beschrieben, spätere Aufgaben ließen etwas mehr Interpretationsspielraum. In Zukunft werden Aufgabenstellungen weniger präzise werden, weil der bewusste Umgang mit Interpretationsspielräumen geübt werden soll.

Interpretationsspielräume können wir zur Vereinfachung von Programmen nutzen, aber müssen es noch nicht. Umgekehrt wird erwartet, dass wir uns an präzise Details der Aufgabenbeschreibungen halten. Es ist nicht akzeptabel, Teile der Beschreibung abzuändern weil wir dies für sinnvoll halten oder schon eine Lösung einer ähnlichen Aufgabe kennen. Es wird auch erwartet, dass wir Aufgabenbeschreibungen genau lesen. Beispielsweise besagt „i ist größer 0“ nicht das Gleiche wie „i ist nicht negativ“. Viele Einschränkungen haben zum Ziel, die Implementierung zu vereinfachen. Manchmal können wir den Grund für eine Einschränkung nicht erkennen, vielleicht weil wir einen Lösungsweg gewählt haben, in dem die Einschränkung keine Rolle spielt. Das bedeutet aber nicht, dass dieser Lösungsweg falsch ist. Jeder Lösungsweg ist erlaubt, solange er der Beschreibung der Aufgabe entspricht.

Vorgehen

Der nächste Schritt besteht im *Entwurf* eines Programms zur Lösung der Aufgabe, das heißt, wir brauchen eine Idee, wie wir vorgehen. In den ersten Aufgaben war der Lösungsweg genau vorgezeichnet, in späteren Aufgaben nicht mehr so genau. Hinter fast allen Teilaufgaben stecken bestimmte Muster, für die wir schon Lösungswege kennengelernt haben. Jetzt sollten wir schon einen ausreichend großen Satz an Mustern kennen um mit allen üblichen Teilen von Programmieraufgaben in der imperativen Programmierung zurechtzukommen. Ein relativ kleiner Satz an Mustern reicht aus, da sich alle Aufgaben auf die se-

quenzielle, wiederholte und alternative Ausführungen von Anweisungen und Ausdrücken zurückführen lassen.

Schwieriger ist die Anforderungsanalyse und der Entwurf, wenn die Aufgabe *Domain-Wissen* voraussetzt, also Wissen, das über das Allgemeinwissen hinausgeht. Dieses Wissen müssen wir uns erst erarbeiten. Die meisten Aufgaben sind so gestaltet, dass sie möglichst wenig Domain-Wissen benötigen. Sollte doch Domain-Wissen nötig sein, wird es in der Aufgabenstellung mitgeliefert – beispielsweise die genauen Regeln eines zu implementierenden Spiels. Eine Ausnahme stellt Domain-Wissen im Bereich der Programmierung dar. Es wird davon ausgegangen, dass wir Domain-Wissen in diesem Bereich haben. Insbesondere müssen wir Fachbegriffe, die uns bereits bekannt sein sollten, so wie definiert interpretieren.

Implementieren

Unter der *Implementierung* der Aufgabe verstehen wir die Umsetzung des Entwurfs in ausführbaren Programmtext. Mit dem nötigen Detailwissen über die verwendeten Sprachkonstrukte der Programmiersprache sollte dieser Schritt der einfachste sein. Manchmal gibt die Entwicklungsumgebung (IDE) Hinweise auf Fehler und zeigt Möglichkeiten auf, wie wir weiter vorgehen können. Es muss uns jedoch bewusst sein, dass die IDE keinerlei Wissen über unsere Aufgabe und unseren Entwurf hat und gegebene Tipps, die wir nur blind befolgen, wahrscheinlich in eine ganz falsche Richtung führen. Daher erfordert auch der Umgang mit der IDE Erfahrung, insbesondere weil es sich dabei um recht umfangreiche und komplexe Software handelt. Die größten Teile dieser Software können wir glücklicherweise einfach ignorieren, ohne dadurch wesentliche Nachteile zu haben. Aber wir müssen erkennen, welche Tipps der IDE in eine für die Lösung unserer Aufgabe richtige Richtung lenken und welche nicht. Auch hierfür brauchen wir Erfahrung, die wir durch das Lösen vieler Aufgaben bekommen.

Testen und Debuggen

In den seltensten Fällen werden unsere Programme gleich auf Anhieb fehlerfrei funktionieren. Wir müssen daher in der Lage sein, durch *Testen* Fehler als solche zu erkennen und durch das *Nachvollziehen des Programmablaufs* die Ursache zu finden und zu beseitigen. Auch dazu braucht es Erfahrung durch Üben. Werkzeuge wie *Debugger* sind hilfreich beim Nachvollziehen des Programmablaufs, aber auch einfache Techniken wie das Einstreuen von Ausgaben an kritischen Programmstellen können zum Auffinden einer Fehlerursache beitragen. Dennoch ist von allen Schritten der Programmentwicklung dieser wahrscheinlich am stärksten von der Intuition abhängig. Es gibt keine Vorgehensweise, die uns sicher zu einem fehlerfreien Programm führt.

Abgesehen von der Notwendigkeit zu üben sind die hier genannten Vorgehensweisen recht vage. In EP2 werden wir verschiedene Konzepte und Techniken kennenlernen, die für einige dieser Schritte konkretere Vorgehensweisen ermöglichen. Ein allgemein gültiges Rezept, dessen Befolgung immer zu guten Programmen führt, gibt es jedoch nicht.

1.1.3 Gute Programme

Wir hören es immer wieder: „Hauptsache, das Programm funktioniert." Natürlich sollen Programme so funktionieren, wie wir es erwarten. Wirklich spannend wird die Beschäftigung mit der Programmierung aber erst, wenn wir darüber hinausgehen und Unterschiede zwischen „funktionierenden" Programmen betrachten. Wir erkennen, dass einige deutlich besser sind als andere, auch wenn sie das gleiche machen.

Einfachheit

Programme sollen so *einfach* wie möglich sein. Software ist von Natur aus komplex, und jede zusätzliche Erhöhung der Komplexität lässt die Entwicklungszeiten und vor allem die für die *Wartung* nötigen Zeiten gewaltig ansteigen. Der Punkt, an dem die Komplexität nicht mehr handhabbar ist, wird schnell erreicht. Auf Einfachheit zu achten bringt uns daher bares Geld und macht komplexe Software erst möglich.

Einfachheit hat viele Aspekte. Klarerweise sollen keine Programmteile vorhanden sein, die nirgends benötigt werden, denn sie verursachen nur Aufwand bei der Programmerstellung und können später bei der Wartung leicht zu Fehlern führen, ohne irgendeinen positiven Effekt zu haben. Wir müssen unnötige Programmteile, die nach Programmänderungen zurückbleiben, umgehend entfernen.

Unnötige Komplexität entsteht auch, wenn wir zu viel auf einmal machen und dabei eigentlich unabhängige Dinge als Einheit betrachten. Vergleichen wir folgende Methodenimplementierungen:

Listing 1.1 Beide Parameter als Einheit bearbeitet

```
1 // names of parameters with value 'true'
2 private static String ABString(boolean a, boolean b) {
3     return a && b ? "ab" : a ? "a" : b ? "b" : "";
4 }
```

Listing 1.2 Parameter getrennt bearbeitet und Teilergebnisse kombiniert

```
1 // names of parameters with value 'true'
2 private static String ABString(boolean a, boolean b) {
3     return (a ? "a" : "") + (b ? "b" : "");
4 }
```

Die Variante in Listing 1.1 benötigt mehr Fallunterscheidungen als jene in Listing 1.2. Die Anzahl an Fallunterscheidungen ist ein Indikator für die Einfachheit eines Programms. Fast jede vermiedene Fallunterscheidung vereinfacht und verbessert das Programm.

Es ist schwierig, unabhängige Teile als solche zu erkennen. Winzige Details in einer Aufgabenstellung können dafür sorgen, dass die Teile voneinander abhängig werden. In Listing 1.1 könnten wir beispielsweise `"ab"` durch `"a b"` ersetzen. In Listing 1.2 gibt es jedoch keine einfache Möglichkeit einer dazu analogen Änderung ohne zusätzliche Fallunterscheidung. Aus solchen Gründen ist es eine große Herausforderung, die Anzahl an Fallunterscheidungen so klein wie möglich zu halten.

So manche Fallunterscheidung ist gut „versteckt“, also nicht gleich erkennbar. Ein typisches Beispiel dafür ist das vorzeitige Abbrechen einer Schleifenausführung mittels `break`. Beim Beurteilen von Studierendenarbeiten fällt immer wieder auf, dass fast alle Lösungen mit `break` in einer Schleife schwere Logikfehler enthalten, offensichtlich weil einige der vielen dabei entstehenden Ausführungszweige nicht bedacht wurden. „Versteckte“ Fallunterscheidungen können die Einfachheit eines Programms durch reduzierte Lesbarkeit stärker negativ beeinflussen als dies deutlich erkennbare Unterscheidungen tun.

Abstraktion

Entsprechend des *Abstraktionsprinzips* (im Bereich der Programmierung) sollen gleiche (oder einander ähnliche) Programmteile so weit wie möglich vermieden werden. Die Befolgung des Abstraktionsprinzips ist eine der wichtigsten Maßnahmen um die Anzahl an Fallunterscheidungen zu reduzieren. Statt mehrerer gleicher Anweisungen hintereinander (schließt Anweisungsfolgen und Kontrollstrukturen ein) schreiben wir eher eine Schleife mit nur einer solchen Anweisung im Rumpf. Das wichtigste Ausdrucksmittel zur Vermeidung gleicher Programmteile ist die *Methode,* die beliebige Anweisungen zu einer Einheit zusammenfassen kann. Über Parameter ist sie oft auch dann noch einsetzbar, wenn Programmteile nicht gleich sondern nur einander ähnlich sind. Darüber hinaus hat jede Methode einen Namen, der beschreiben sollte, was die Methode tut und wozu sie verwendet wird. Statt an Anweisungen im Methodenrumpf denken wir beim Aufruf einer Methode nur ganz abstrakt an das, was durch den Namen suggeriert wird. Genau das steckt hinter dem Begriff *Abstraktion.* Durch Befolgung des Abstraktionsprinzips denken wir beim Programmieren in abstrakten Einheiten und nicht in wesentlich komplexeren Implementierungsdetails. Erst durch Abstraktion wird es möglich, komplexe Aufgabenstellungen mit guten Erfolgsaussichten in Angriff zu nehmen.

Es ist eine Binsenweisheit, dass wir auf einheitliche Formatierung mit die Struktur verdeutlichenden Einrückungen, aussagekräftige und dennoch kurze Namen sowie hilfreiche Kommentare achten und unnötig komplizierte Formulierungen vermeiden sollen. Unter Zeitdruck halten wir uns oft trotzdem nicht daran. Damit geht das Ziel dahinter verloren – die gute Lesbarkeit der Programme. Solange wir es nur mit kleinen, einfachen Programmen und Programmteilen zu tun haben, ist das nicht weiter tragisch. Aber bei der Entwicklung großer, komplexer Programme, die über einen langen Zeitraum gewartet werden müssen, sind solche scheinbaren Kleinigkeiten entscheidend. Deswegen finden wir bei fast allen international bedeutenden Softwareunternehmen klare Vorschriften für das Aussehen akzeptabler Programmtexte sowie Arbeitsabläufe zur Sicherstellung der Einhaltung. Es gibt aber auch viele (eher kleine) Softwareunternehmen, wo die kurzfristige Produktivität wichtiger ist als die langfristige Perspektive und daher auf derartige Vorschriften verzichtet wird.

Es gibt auch das umgekehrte Phänomen. Gerade Personen mit wenig Programmiererfahrung halten sich häufig ganz restiktiv an (vielfach stark simplifizierte) Regeln, auch in Fällen, in denen sie nicht gelten. Beispielsweise werden auch für nur lokal in Schleifen verwendete Laufvariablen gerne lange, beschreibende Namen verwendet, obwohl übliche Namen wie `i`, `j` und `k` vollkommen ausreichen. Das kann Entwicklungszeiten verlängern und die Wartung etwas erschweren, da sich die Intuition hinter Namen im Laufe der Zeit oft schleichend verändert. Unbedacht und rasch hingeschriebene Kommentare können die Wartung sogar sehr stark negativ beeinflussen. Wir müssen also den Zweck von Regeln im Auge behalten, sie nur bedacht, dem Zweck entsprechend anwenden und bewusst ignorieren, wo dieser Zweck nicht gegeben ist. Daher erfordern auch scheinbar ganz einfache Regeln sehr viel Wissen.

1.2 Themenbereiche

Obige Beschreibung vorausgesetzter Kompetenzen wurde bewusst untergliedert um aufzuzeigen, dass sich Programmierkenntnisse nicht nur auf Kenntnisse von Programmierkonzepten im engeren Sinn beschränken. Wir müssen auch darauf achten, wie wir beim Erstellen von Programmen vorgehen um komplexere Aufgaben lösen zu können, und dafür sorgen, dass unsere Programme gewisse Qualitätskriterien erfüllen. Während in EP1 grundlegende Programmierkonzepte noch ganz

im Vordergrund gestanden sind, werden wir uns in EP2 neben weiter fortgeschrittenen Programmierkonzepten verstärkt mit Techniken beschäftigen, die vor allem der Erhöhung des Abstraktionsgrads und der Qualitätssicherung dienen. Auch ausgewählte Algorithmen und Datenstrukturen werden intensiv behandelt. Bevor wir die in EP2 behandelten Themenbereiche überblicksmäßig betrachten, führen wir eine etwas ausführlichere thematische Gliederung dieser Themenbereiche an.

Abstraktion
Qualität

1.2.1 Thematische Gliederung

Wir gliedern die Themenbereiche nach folgenden Gesichtspunkten:

Berechnungsmodell: Darunter verstehen wir die konzeptuelle, logische Maschinerie, durch die Eingabewerte entsprechend einem Programm in Ausgabewerte umgeformt werden. Hinter jeder Programmiersprache steckt ein Berechnungsmodell, und die Sprache lässt die Bestandteile der Maschinerie mehr oder weniger direkt erkennen. Hinter Java steckt ein Berechnungsmodell, in dem Zuweisungen Werte in Variablen und als Arrayeinträge ablegen, Auswertungen von Ausdrücken Ergebnisse liefern und Seiteneffekte haben, sich Funktionen (als Methoden) gegenseitig aufrufen, Anweisungen hintereinander ausgeführt werden und Kontrollstrukturen für bedingte und wiederholte Ausführungen sorgen. Die meisten grundlegenden Konzepte der Programmierung betreffen das Berechnungsmodell. Zur Kennzeichnung von Aspekten der Programmierung im Zusammenhang mit einem Berechnungsmodell verwenden wir das Symbol in der Randleiste; es stellt einen Mechanismus in Form roter Zahnräder dar, der Ausdrücke wie `2*3` in Ergebnisse wie `6` umformt.

Algorithmen und Datenstrukturen: Wir implementieren mit Hilfe einer Programmiersprache Algorithmen, die Datenstrukturen verwenden bzw. bereitstellen. Algorithmen beschreiben das Verhalten von Programmen auf einer viel abstrakteren Ebene als Implementierungen. So bleiben Algorithmen recht unabhängig von einem konkreten Berechnungsmodell. Datenstrukturen beschreiben ebenso unabhängig von einem Berechnungsmodell, wie Daten in Datenmengen zusammenhängen und wie wir über Algorithmen darauf zugreifen können. Algorithmen hängen stark mit Datenstrukturen zusammen. Beim Entwickeln von Algorithmen und Datenstrukturen konzentrieren wir uns auf das Essenzielle und

lassen Implementierungsdetails unberücksichtigt. So können wir allgemeingültige Eigenschaften besser erkennen. Das wird durch das Symbol in der Randleiste ausgedrückt.

Programmorganisation: Es ist eine gewaltige Herausforderung, bei der Entwicklung und Wartung großer und langlebiger Software den Überblick zu behalten. Berechnungsmodelle alleine reichen dazu nicht aus. Wir müssen in abstrakten Einheiten denken, den Programmtext in überschaubare Teile untergliedern und die Teile auf gut durchdachte Weise miteinander kombinieren um eine Chance zu haben, die gigantische Komplexität zu beherrschen. Konzepte der Programmorganisation wurden zu diesem Zweck entwickelt. Sie dienen unter anderem dazu, Programmteile verändern zu können, ohne andere Programmteile in Mitleidenschaft zu ziehen. Die Anwendung solcher Konzepte bezeichnen wir als *Programmieren im Großen,* während wir bei Anwendung von Konzepten eines Berechnungsmodells vom *Programmieren im Kleinen* sprechen. Das Programmieren im Großen ist viel herausfordernder als das Programmieren im Kleinen. Wie im Symbol in der Randleiste stehen beim Programmieren im Großen die Querverbindungen zwischen den Programmteilen im Mittelpunkt der Überlegungen.

Qualitätssicherung: Software soll zahlreiche Qualitätskriterien erfüllen, etwa möglichst fehlerfrei, nützlich und gut wartbar sein. Es gibt eine Reihe von Techniken um bereits bei der Entwicklung eine gute Programmqualität sicherzustellen, im Nachhinein zu überprüfen, Qualitätsmängel zu beheben sowie die Auswirkungen von Fehlern zur Laufzeit möglichst rasch einzugrenzen. Wir verwenden das Symbol in der Randleiste für alle Bereiche der Qualitätssicherung, nicht nur im Zusammenhang mit Bugs.

Software-Engineering: Darunter verstehen wir ein systematisches, ingenieurmäßiges Vorgehen bei Entwurf, Implementierung, Test sowie Wartung von Software. Auch vieles, was wir unter anderen Begriffen (wie etwa Programmorganisation) ansprechen, zählt dazu. Das Symbol für Software-Engineering in der Randleiste verwenden wir dann, wenn es vorrangig um die Zusammenarbeit mehrerer Personen oder anfallende Kosten geht.

Java

Sprachen und Werkzeuge: Der Schwerpunkt dieses Skriptums liegt auf Inhalten, die unabhängig von bestimmten Programmiersprachen und Werkzeugen ganz allgemein gültig sind. Jedoch lässt

sich eine Einführung in die Programmierung in der nötigen Tiefe kaum sprachunabhängig gestalten. Wir verwenden durchwegs Java als Programmiersprache und IntelliJ IDEA als integrierte Entwicklungsumgebung. Textpassagen, die sich schwerpunktmäßig auf Konzepte der Programmiersprache, einer Klassenbibliothek oder eines Softwareentwicklungswerkzeugs beziehen, werden durch einen entsprechenden Kurztext in der Randleiste gekennzeichnet. Nicht auf diese Weise gekennzeichnet sind Programmtexte, deren Fokus auf der beispielhaften Darstellung allgemeingültiger Konzepte liegt.

1.2.2 Konzepte, Vorgehensweisen und Techniken

Daten-abstraktion

Ein zentraler Begriff in EP2 ist die *Abstraktion,* insbesondere die *Datenabstraktion* als Hilfsmittel zur Zerlegung großer Programme in kleinere Einheiten. Während die Methodenabstraktion Anweisungen zu einer Methode zusammenfasst, fasst die Datenabstraktion beliebig viele Variablen und Methoden zu einer größeren Einheit, den *abstrakten Datentyp* zusammen.[1] Zur Verwendung abstrakter Datentypen brauchen wir (wie bei Methodenaufrufen) keine Implementierungsdetails zu kennen, sondern haben abstrakte Vorstellungen von überschaubaren Einheiten im Kopf – eine Voraussetzung für die Realisierung größerer Programme.

Data-Hiding

Manche Implementierungsdetails abstrakter Datentypen sind von außen nicht verwendbar. Sie können geändert werden, ohne die Verwendung der abstrakten Datentypen zu beeinflussen. Damit können wir die Programmteile relativ unabhängig voneinander gestalten und auch unabhängig voneinander weiterentwickeln – eine Voraussetzung für die effiziente Zusammenarbeit mehrerer Personen im Team.

Abstraktions-hierarchie

Wir können ein und denselben abstrakten Datentyp auf unterschiedlichen Abstraktionsebenen betrachten, wobei jede Ebene unterschiedlich viel von der Implementierung preisgibt. So entstehen ganze Abstraktionshierarchien. Weit oben in einer Hierarchie steht nur ein kleiner Funktionsumfang zur Verfügung, dafür bestehen viele Weiterentwicklungsmöglichkeiten. Weit unten ist der verfügbare Funktionsumfang groß, aber Abhängigkeiten erschweren die Weiterentwicklung. Für jede Anwendung können wir die am besten passende Abstraktionsebene

[1] Auch das gezielte Sichtbarmachen von Daten und Methoden gehört dazu. *Datenabstraktion* bezeichnet sowohl das generelle Konzept, als auch den Vorgang des Zusammenfassens, sowie manchmal eine dabei entstehende Einheit. Um Verwirrungen zu vermeiden, nennen wir eine Einheit eher *abstrakter Datentyp.*

Ersetzbarkeit

wählen. Abstraktionshierarchien ermöglichen *Ersetzbarkeit,* das heißt, wir können einen abstrakten Datentyp durch einen anderen ersetzen, der dieselbe Abstraktion darstellt. Das geht sogar zur Laufzeit.

fertiger abstrakter Datentyp

Programmierumgebungen enthalten viele fertig implementierte abstrakte Datentypen. Vor allem typische Datenstrukturen und häufig eingesetzte Algorithmen stehen dadurch meist in hoher Qualität zur Verwendung bereit. So ersparen wir uns das Schreiben großer Mengen an Programmtext. Allerdings müssen wir dahinterliegende abstrakte Konzepte verstehen. Wir tauschen also den Aufwand für das Schreiben eigenen Programmtexts gegen den für die Einarbeitung in abstrakte Konzepte – meist ein für uns günstiger Tausch.

rekursive Datenstruktur

Nicht nur Methoden, auch Datenstrukturen können *rekursiv* sein. Rekursive Datenstrukturen sind in der Praxis wichtig, weil sie unlimitierte Mengen an Daten aufnehmen können, beschränkt nur durch die Speichergröße. Je nach Art, beispielsweise *Liste* oder *Baum,* haben Datenstrukturen unterschiedliche Eigenschaften. Durch die Wahl von Implementierungsdetails der auf Datenstrukturen definierten Zugriffsmethoden (Algorithmen) lassen sich die Eigenschaften wesentlich beeinflussen. Gleichzeitig gibt es auch eine Systematik hinter den Zugriffsmethoden, die einen hohen Abstraktionsgrad ermöglicht.

Design by Contract

Dadurch, dass wir zum Schreiben von Kommentaren gedrängt werden, übersehen wir leicht deren Bedeutung. In aktuellen Vorgehensweisen der Softwareentwicklung sind bestimmte Kommentare essentielle Bestandteile der Festlegung von Abstraktionen und wegen der nötigen Kommunikation zwischen Personen in der Softwareentwicklung langfristig gesehen viel wichtiger als Implementierungsdetails. Hinter diesen Kommentaren steckt ein Konzept namens *Design-by-Contract,* das einen unglaublich starken Einfluss auf die Gesamtstruktur eines (größeren) Programms hat. Entsprechend dieses Konzepts sind Kommentare nicht beliebig, sondern nach mehreren Kriterien gut strukturiert.

Programme statisch verstehen

Das wichtigste Qualitätskriterium eines Programms besteht darin, dass es das fehlerfrei leistet, wofür es eingesetzt wird. Um diese Qualität zu erreichen müssen wir einerseits verstehen, was das Programm leisten soll, andererseits auch, was es tatsächlich tut. Letzteres nennen wir *Programmverstehen.* Es reicht nicht, nur bestimmte Abläufe im Programm *nachzuvollziehen,* weil es so viele davon gibt, dass wir niemals alle abdecken können. Wir wollen Programme auch *statisch verstehen,* also nur aus dem Programmtext selbst Rückschlüsse auf die Funktionsweise ziehen, ohne einzelne Abläufe nachzuvollziehen. Bestimmte Vorgehensweisen erleichtern das statische Programmverstehen. Eine wich-

Code-Review

tige Maßnahme zur Qualitätssicherung ist ein *Code-Review,* also das Durchlesen eines Programmtexts mit dem Ziel, ihn nach vorgegebenen Kriterien zu beurteilen.

Testen

Obwohl die Programmqualität vor allem vom guten Problem- und Programmverstehen kommt, müssen wir auch *testen* um mit etwas Glück unvorhergesehenes Fehlverhalten aufzudecken. Es gibt viele unterschiedliche Vorgehensweisen beim Testen. Eine Gemeinsamkeit besteht darin, dass Fehler nur mit einer gewissen Wahrscheinlichkeit, niemals zuverlässig gefunden werden. Testen kann Schwachstellen aufzeigen. Aber alleine durch Testen und Korrigieren gefundener Fehler wird man kaum zu qualitativ hochwertigen Programmen kommen.

Fehler stellen unter anderem ein Sicherheitsrisiko dar, weil sie zum Einbrechen in Systeme genutzt werden. Fehler sind niemals gänzlich auszuschließen. Quasi als zweites Sicherheitsnetz müssen wir zur Laufzeit an kritischen Stellen immer wieder überprüfen, ob es Anzeichen für einen aufgetretenen Fehler gibt. Im Fehlerfall dürfen wir betroffene Programmteile nicht weiter ausführen. Das heißt, wir brauchen auch zur Laufzeit ein Konzept für den Umgang mit Fehlern.

Überprüfung von Daten

Ein Spezialfall ist der Umgang mit Daten, die von außen in das Programm eingelesen werden. Wir müssen stets davon ausgehen, dass diese Daten fehlerhaft sein können. Bevor wir mit diesen Daten arbeiten, müssen wir sie überprüfen.

Programme sollen natürlich effizient sein. Manchmal wird Wert auf *Optimierungen* gelegt, die sich aber oft nicht wie erwartet auswirken. Laufzeiteffizienz hängt von vielen kaum durchschaubaren Parametern ab. Auf kleine Optimierungen, die oft die Lesbarkeit und Zuverlässigkeit mindern, werden wir daher verzichten. Durch Auswahl passender Datenstrukturen und Algorithmen haben wir eine effektivere Möglichkeit, die Laufzeiteffizienz zu beeinflussen. Meist ist Laufzeiteffizenz nur ein untergeordnetes Ziel. In der Regel ist *Programmiereffizienz* wichtiger – also kurze Entwicklungszeiten und einfache Wartung.

Optimierung

Effizienz

1.2.3 Sprachliche Ausdrucksmittel

Grundlegende Programmierkonzepte (wie in EP1 behandelt) sind in Java fast direkt als sprachliche Ausdrucksmittel widergespiegelt. Daneben finden wir Ausdrucksmittel, die bei der Umsetzung oben angerissener Konzepte, Vorgehensweisen und Techniken helfen – allerdings weitaus weniger direkt. Wir müssen zwischen einem Konzept und dessen sprachlicher Umsetzung im Programm unterscheiden. Im Vordergrund der Überlegungen steht immer das Konzept an sich, etwa Datenabstraktion. Erst danach wählen wir sprachliche Ausdrucksmittel zur Umsetzung, etwa eine Klasse.

Objekt

Klasse

Datenabstraktion wird in Java in der Regel durch *Objekte* realisiert, die als Instanzen von *Klassen* zur Laufzeit erzeugt und über *Referenzen* angesprochen werden. Für `String` und Arrays kennen wir die Ausdrucksmittel schon. Wir werden lernen, eigene Datenabstraktionen zu entwickeln. Gleichzeitig sind Objekte, Klassen und Referenzen auch wichtige sprachliche Ausdrucksmittel zur Realisierung von Datenstrukturen. Es hängt nur von der Betrachtungsweise ab, welcher Aspekt im Vordergrund steht. Aus Sicht der Datenabstraktion kommt es darauf an, dass eine Ansammlung von Variablen und Methoden als Einheit betrachtet und der Zugriff darauf gezielt eingeschränkt wird. Als Datenstruktur betrachtet kommt es darauf an, wie die Daten in den über Methoden zugreifbaren Variablen voneinander abhängen.

Interface

Abstraktionshierarchien werden in Java über *Interfaces* realisiert, gelegentlich auch durch *Unterklassen.* Obwohl viele Bücher über Java die *Vererbung* in den Mittelpunkt stellen, ist Vererbung (im ursprünglichen Sinn) in der Praxis von untergeordneter Bedeutung. Viel mehr kommt es darauf an, mithilfe von Interfaces geeignete Abstraktionsebenen zu gestalten, die sowohl für gute Verwendbarkeit sorgen, als auch eine möglichst unabhängige Weiterentwicklung erlauben.

Ersetzbarkeit beruht auf Abstraktionshierarchien. Für die Ersetzbarkeit zur Laufzeit ist zusätzlich *dynamisches Binden* von Methodenaufrufen erforderlich. Dieser ganz natürlich in Java und alle anderen objektorientierten Sprachen integrierte Mechanismus ist eigentlich nur eine Form der Programmverzweigung, die den auszuführenden Programmzweig anhand des Typs eines Objekts wählt. Dennoch ist dynamisches Binden von zentraler Bedeutung für die praktische Nutzung der Datenabstraktion und von Klassenbibliotheken. Durch Verwendung der zahlreichen Klassen in fertigen Klassenbibliotheken gestaltet sich die Programmentwicklung wesentlich effizienter.

Ausnahmebehandlung

Für das *Exception-Handling* existiert eine Sammlung zusammenhängender Mechanismen, die es erlauben, auf zur Laufzeit aufgetretene Fehler angemessen zu reagieren. Beispielsweise muss ein Laufzeitfehler nicht gleich zum Programmabbruch führen. Unter idealen Bedingungen kann das Programm nach Beseitigung der Fehlerursache normal fortgesetzt werden. Meist ist es jedoch nur möglich, die fehlerhaften Zustände so weit einzugrenzen, dass eine Fehlermeldung ausgegeben oder mit nicht betroffenen Programmteilen fortgesetzt werden kann.

Ein- und Ausgabe

Bei Weitem nicht für alle wichtigen Konzepte, Vorgehensweisen und Techniken gibt es eigene, fest in eine Programmiersprache integrierte Ausdrucksmittel. Häufig erfolgt die Unterstützung durch Klassen in Klassenbibliotheken, die hilfreiche abstrakte Datentypen bereitstellen. Beispielsweise gibt es in Java keine eigenen sprachlichen Ausdrucksmittel für die Ein- und Ausgabe von Daten. Allerdings gibt es zahlreiche fertige Klassen, über die wir auch komplexe Formen der Ein- und Ausgabe relativ einfach realisieren können. Das hat Vorteile: Neue Formen der Ein- und Ausgabe (etwa zur Kommunikation über zukünftige, derzeit noch unbekannte Kommunikationskanäle) können leicht hinzugefügt werden, ohne die Sprache selbst oder bestehende Klassen ändern zu müssen. Stattdessen werden neue Klassen hinzugefügt.

Implementierung

Wir haben schon festgestellt, dass viele Algorithmen und Datenstrukturen bereits in fertigen Klassen in hoher Qualität bereitstehen und in der Praxis meist in dieser Form verwendet werden. Dennoch hat EP2 auch einen Schwerpunkt in der Implementierung einiger grundlegender Algorithmen und Datenstrukturen. Sie sind in der Informatik von so zentraler Bedeutung, dass alle Studierenden eines entsprechenden Fachs überall auf der Welt erfahren müssen, wie man sie selbst implementieren kann. Dieser Schwerpunkt ist eher didaktisch begründet, nicht so sehr durch den Praxisbezug. Die Implementierung einiger Datenstrukturen, insbesondere von Listen und Bäumen, setzt profundes Wissen über den Umgang mit Objekten und Referenzen voraus und zeigt Möglichkeiten der Datenabstraktion auf. Bäume machen auch deutlich, wie wichtig rekursive Methoden sind. Insofern stellen diese Datenstrukturen bestens geeignete Beispiele für den Umgang mit zahlreichen nichttrivialen sprachlichen Ausdrucksmitteln dar, die in leicht abgewandelter Form in der Praxis von großer Bedeutung sind.

1.3 Lehrveranstaltung und Lernen

Konkretisieren wir nun, was wir in EP2 auf welche Weise lernen.

1.3.1 Voraussetzungen und erwartete Lernergebnisse

Nachdem EP2 auf EP1 aufbaut, wird erwartet, dass alle an EP2 teilnehmenden Studierenden bereits vor Beginn der Lehrveranstaltung Kompetenzen haben, die den erwarteten Lernergebnissen von EP1 entsprechen. Konkreter als in Abschnitt 1.1 sind das diese:

Fachliche und methodische Kenntnisse: Studierende können bereits vor der Teilnahme an EP2 Folgendes beschreiben: Voraussetzung

- grundlegende prozedurale Konzepte von Java,
- grundlegende Such- und Sortier-Algorithmen auf Arrays,
- Fehlerquellen in einfachen prozeduralen Programmen.

Kognitive und praktische Fertigkeiten: Studierende können bereits vor der Teilnahme an EP2 einfache Programme in Java (ähnlich denen aus EP1) erstellen, nachvollziehen, debuggen, modifizieren und dokumentieren. Voraussetzung

Soziale und persönliche Kompetenzen: Studierende können bereits vor der Teilnahme an EP2 Voraussetzung

- beim Lösen von einfachen Programmieraufgaben selbstständig vorgehen,
- einfache Programmeigenschaften kommunizieren.

Es bestehen keine besonderen formalen Voraussetzungen für die Teilnahme an EP2 – abgesehen von den allgemein üblichen Voraussetzungen wie Zulassung zu einem entsprechenden Studium und Anmeldung zur Lehrveranstaltung. Insbesondere wird nicht überprüft, ob Studierende die erwarteten Kompetenzen mitbringen. Dennoch ist es in EP2 unmöglich, auf Probleme jener Studierender Rücksicht zu nehmen, die nicht die nötigen Kompetenzen mitbringen.

EP2 ist darauf ausgelegt, dass erfolgreiche Studierende am Ende der Lehrveranstaltung diese Lernergebnisse erzielt haben:

Fachliche und methodische Kenntnisse: Nach positiver Absolvierung von EP2 können die Studierenden Folgendes beschreiben: Ergebnisse

- systematische Vorgehensweisen beim Programmieren (vor allem zum Erstellen und Verwenden von Datenabstraktionen, Durchlaufen und Modifizieren rekursiver Datenstrukturen, sowie zur Dokumentation),
- beispielhaft ausgewählte Algorithmen, Datenstrukturen und Datenabstraktionen,
- häufige Fehlerquellen sowie Techniken zur Qualitätssicherung (Validierung von Eingabedaten, Exception-Handling, Testen, Dokumentation, Code-Review).

Ergebnisse

Kognitive und praktische Fertigkeiten: Nach positiver Absolvierung von EP2 können die Studierenden

- Inhalte von in natürlicher Sprache formulierten Programmieraufgaben in ausführbare Java-Programme umsetzen,
- beschriebene Datenabstraktionen, Algorithmen und Datenstrukturen implementieren und verwenden,
- Sprachelemente und Techniken zur Datenabstraktion, zum Durchlaufen und Modifizieren rekursiver Datenstrukturen, zur Ein- und Ausgabe über Dateien sowie zur Qualitätssicherung anwenden.

Ergebnisse

Soziale und persönliche Kompetenzen: Nach positiver Absolvierung von EP2 können die Studierenden

- beim Lösen von Programmieraufgaben selbständig vorgehen sowie in Zweierteams zusammenarbeiten,
- Eigenschaften von Programmen kommunizieren.

EP2 hat zum Ziel, mitgebrachte Kompetenzen weiter auszubauen sowie in einigen Bereichen neue Kompetenzen zu erwerben, damit am Ende die erwarteten Lernergebnisse vorhanden sind. Die Lernaktivitäten in der Lehrveranstaltung sind darauf ausgelegt, das Ziel

Aufwand

- bei Mitbringen aller vorausgesetzten Kompetenzen
- unter intensiver und zeitlich effizienter Mitarbeit
- mit einem Lernaufwand von 4 ECTS (durchschnittlich etwa 100 Arbeitsstunden) erreichbar zu machen.

Werden nicht alle vorausgesetzten Kompetenzen in vollem Umfang mitgebracht oder ist die Mitarbeit oberflächlich bzw. ineffizient, muss mit einem viel höheren zeitlichen Aufwand oder dem Scheitern gerechnet werden. Aber auch unter optimalen Bedingungen kann der nötige Lernaufwand aufgrund persönlicher Unterschiede im Lernfortschritt stark schwanken. Die Beurteilung orientiert sich nicht am Lernaufwand, sondern wird hauptsächlich davon abhängen, ob und wie gut erwartete Lernergebnisse zu den Testzeitpunkten vorhanden sind.

1.3.2 Struktur der Lehrveranstaltung

Die Lehrveranstaltung ist aus folgenden Lernaktivitäten aufgebaut:

Vorträge: Vorlesungen (Vorträge) geben einen Überblick über die behandelten Themen, mit Beispielprogrammen und kleinen Aufgaben. Nebenbei ergeben sich Möglichkeiten zum Aufbau und zur Pflege von Kontakten zu anderen Studierenden. Die Vorträge werden aufgezeichnet und können genauso wie Folien und Beispielprogramme nachträglichen angesehen werden, jedoch ohne die soziale Komponente der Vorträge.

Skriptum: Das Skriptum fasst auf EP2 zugeschnittene Informationen detailreicher und in sich konsistenter zusammen als die Vorträge, Folien und Beispielprogramme. Es dient zur Vorbereitung auf Vorträge genauso wie zum Nachlesen nach den Vorträgen oder als Erinnerungshilfe zur Vorbereitung auf Übungen und Tests. Allerdings fehlt die dynamische und soziale Komponente der Vorträge.

Hausübungen: Jede der sieben Hausübungen fasst mehrere Programmieraufgaben zu einem IDEA-Projekt zusammen, die örtlich ungebunden innerhalb von etwa einer Woche zu lösen und abzugeben sind. Hausübungen bieten eine Gelegenheit zum Üben der wichtigsten Themen auf strukturierte Weise. Sie bereiten gezielt auf geleitete Übungen und Tests vor.

Geleitete Übungen: Sieben der neun geleiteten Übungen haben eine Hausübung zum Thema, die restlichen beiden dienen der direkten Testvorbereitung. Gelegentlich stellen wir unsere Lösung einer Aufgabe vor. Zusätzlich müssen wir unter Zeitdruck alleine oder im Zweierteam kleine Programmieraufgaben lösen. Wegen des direkten Feedbacks ist diese Lernaktivität die wahrscheinlich wichtigste.

Tutorium: Zu gewissen Zeiten treffen wir uns mit anderen Studierenden zu einem zwanglosen Erfahrungsaustausch. Zu diesen Zeiten sind auch Tutorinnen, Tutoren oder Lehrende anwesend um etwaige Fragen zu beantworten.

Reguläre Tests: Etwa zu Semestermitte und am Semesterende findet je ein (regulärer) Test zu jeweils 75 Minuten am Computer statt, in dem neben Multiple-Choice-Aufgaben vor allem Programmieraufgaben zu lösen sind. Die Testergebnisse bilden die wichtigste Grundlage für die Beurteilung.

Nachtragstests: Zu jedem regulären Test findet ein Nachtragstest statt, der eine Wiederholungsmöglichkeit bietet, falls der reguläre Test negativ beurteilt oder versäumt wurde.

Freies Üben: Nicht jede Form des Lernens ist bis ins Detail organisiert. Wir werden auch außerhalb des Rahmens von Hausübungen und geleiteten Übungen lernen und üben müssen, beispielsweise um uns neue Kompetenzen anzueignen oder auf geleitete Übungen und Tests vorzubereiten. Vor allem das Üben in kleinen Lerngruppen hat sich als erfolgversprechend erwiesen.

Die Beurteilung der Lehrveranstaltung erfolgt nach den in TISS verlautbarten Regeln. Vereinfacht gilt Folgendes:

- Eine positive Gesamtbeurteilung setzt die positive Beurteilung der Übungen und beider regulärer Tests (oder entsprechender Nachtragstests) voraus.
- Ein Teil (Übungen oder Test 1 oder Test 2) ist positiv, wenn mindestens 50% der in diesem Teil erreichbaren Leistungen erbracht wurden.
- Bei positiver Gesamtbeurteilung ergibt sich die Note gleichverteilt aus dem Durchschnitt der Beurteilungen für die drei Teile (Übungen und Test 1 und Test 2).
- Die Beurteilung des Übungsteils (mit insgesamt maximal 100 Prozentpunkten) setzt sich zusammen aus
  - den Beurteilungen der Hausübungen,
  - der Mitarbeit in geleiteten Übungen,
  - und den Beurteilungen der während der geleiteten Übungen gelösten Programmieraufgaben.

Eine beachtliche Zahl an Studierenden orientiert sich beim Lernen in erster Linie an Beurteilungsregeln, kaum am Angebot an Lernaktivitäten, inhaltlichen Interessen oder zielführenden Methoden des Lernens. Daher sind die Regeln so aufgebaut, dass die mangelhafte Mitarbeit in einigen (aber nicht allen) Lernaktivitäten automatisch zu einer schlechteren oder sogar negativen Beurteilung führt, unabhängig von Kompetenzen. Dennoch liegt ein eindeutiger Schwerpunkt der Beurteilung auf den Tests sowie kompetenzorientierten Leistungen in den Übungen. Damit die Teilnahme an EP2 zu einem Erfolg wird, müssen wir uns

beim Lernen auf die Aneignung der oben beschriebenen Kompetenzen konzentrieren und uns gleichzeitig an die Vorgaben in den Beurteilungsregeln halten, dürfen die beurteilungsrelevanten Lernaktivitäten aber nicht als für die Aneignung der Kompetenzen hinreichend betrachten.

1.3.3 Hinweise zur Teilnahme und zum Lernen

Bevor wir uns Details zum Lernen in EP2 überlegen, müssen wir klären, ob wir überhaupt an EP2 teilnehmen sollen. Hier sind einige diesbezügliche Empfehlungen:

- Wer EP1 schon positiv absolviert hat, soll unbedingt an EP2 teilnehmen. EP1 und EP2 werden am besten in kurzem Abstand hintereinander absolviert. Außerdem hängen viele weitere Lehrveranstaltungen sowohl von EP1 als auch EP2 ab. — so früh wie möglich
- Es wird empfohlen, EP2 im gleichen Semester wie *Algorithmen und Datenstrukturen* zu besuchen, da diese Lehrveranstaltungen dafür ausgelegt sind, sich gegenseitig zu ergänzen. — mit AlgoDat
- Es ist unmöglich, sich bedeutende fehlende Kompetenzen von EP1 gleichzeitig mit denen von EP2 anzueignen.[2] Wer EP1 schon teilweise absolviert hat, aber nicht positiv beurteilt wurde, sollte noch nicht an EP2 teilnehmen. Die Teilnahme ist nicht sinnvoll, wenn vorausgesetzte Kompetenzen fehlen. — nicht ohne Voraussetzungen
- Wer schief (im Sommersemester) einsteigt und daher EP1 noch nicht absolvieren konnte, soll an EP2 (gemeinsam mit *Algorithmen und Datenstrukturen*) nur teilnehmen, wenn schon ausreichende Programmiererfahrungen vorhanden sind. Ohne Programmierkenntnisse ist EP2 nicht geeignet. — selbst einschätzen

Sobald eine Entscheidung für die Teilnahme getroffen wurde, ist es notwendig, sich ganz auf EP2 einzulassen, von Semesteranfang bis Semesterende. Es funktioniert nicht, „es nur einmal auszuprobieren“. Die Lernkurve ist in EP2 wesentlich steiler als in EP1.

Die Aneignung von Kompetenzen in der Programmierung ist indivi- — überall teilnehmen

[2] Programmieren lernen wir in Phasen, die hintereinander ablaufen. Kompetenzen späterer Phasen können wir uns erst aneignen, wenn schon Kompetenzen der früheren Phasen vorhanden sind. Wenn wir die Reihenfolge umzukehren versuchen, werden wir zwar Wissen aus späteren Phasen beschreiben können, aber nicht in der Lage sein, dieses Wissen zu verstehen und praktisch anzuwenden.

duell sehr unterschiedlich. Unzählige Faktoren beeinflussen den dafür nötigen Aufwand. Fortschritte ergeben sich oft ruckartig – manchmal große Fortschritte in kurzer Zeit, dann wieder lange Pausen, in denen scheinbar nichts weitergeht. Eine Lernaktivität, die einer Person zu einem großen Fortschritt verhilft, bleibt bei einer anderen Person ganz ohne Wirkung. Wie in Abschnitt 1.3.2 angeführt, wird eine breite Palette an Lernaktivitäten angeboten, in der Hoffnung, dass für alle Studierenden etwas dabei ist, das sie weiterbringt. Im Allgemeinen weiß niemand, welche Lernaktivität für welche Person am erfolgversprechendsten ist. Wir können es nur ausprobieren. Aus diesem Grund gibt es die Empfehlung, an allen Lernaktivitäten über den gesamten Zeitraum teilzunehmen. Wir können nicht vorhersehen, ob uns nicht gerade die Lernaktivität, die uns anfangs am wenigsten sinnvoll schien, irgendwann den größten Lernfortschritt verschafft.

Initiative ergreifen

Lehrenden ist es in Massenlehrveranstaltungen kaum möglich, individuell auf jeden einzelnen Menschen einzugehen. Stattdessen müssen sich Lehrende darauf verlassen, dass Studierende die Initiative ergreifen und sich Kompetenzen entsprechend ihrer individuellen Lernsituationen selbständig aneignen. Lehrende können nur für ein Umfeld sorgen, das dies erleichtert. Lehrende werden gelegentlich Feedback geben, z. B. in geleiteten Übungen, oder wenn Studierende im Tutorium gezielt danach fragen. Selbstverständlich müssen Lehrende auch für Beurteilungen sorgen und damit Druck auf Studierende ausüben, sich Kompetenzen zeitgerecht anzueignen.

Wir müssen uns die in Abschnitt 1.3.1 grob beschriebenen Kompetenzen erarbeiten. Beispielsweise reicht es nicht zu verstehen, wie man Programme nachvollzieht und schreibt. Wir müssen in der Lage sein, natürlichsprachige Programmieraufgaben richtig zu interpretieren und Java-Programme entsprechend der Inhalte zu schreiben. Aber auch das alleine reicht nicht. Wir müssen zusätzlich beschreiben können, wie wir beim Programmieren systematisch vorgehen, und mit anderen Personen darüber sprechen können, welche Eigenschaften unsere Programme haben. Neben fachlichen Kenntnissen brauchen wir praktische Fertigkeiten und gewisse soziale und persönliche Kompetenzen. Das ist eine Vielzahl von Fähigkeiten auf unterschiedlichen Gebieten, die mit dem Wort *Programmierenlernen* nur sehr unzureichend ausgedrückt sind. Ein typischer Fehler beim Erarbeiten von Kompetenzen besteht darin, sich die falschen Kompetenzen zu erarbeiten. Manchmal glauben wir, problemlos Programmieraufgaben in Programme umsetzen zu können, wenn wir viele solche Aufgaben und entsprechende Programme gesehen und verstanden haben. Tatsächlich ist es etwas ganz anderes, diese Umsetzung selbst zu machen. Es hilft zwar, wenn wir viele Beispiele gesehen haben, aber Aufgaben in Programme umzusetzen lernen wir nur, indem wir das wirklich auch selbst tun. Wir müssen das alleine genauso wie in Teams tun.

richtige Kompetenzen

Folgende Empfehlungen sind dafür hilfreich:

- In Gruppen lernt es sich meist besser. Die Gruppendynamik hilft beim Erarbeiten bestimmter Kompetenzen und ist auch motivierend. Wir dürfen trotzdem nicht vergessen, dass wir uns einige der nötigen Kompetenzen nur alleine erarbeiten können.

  Lerngruppe

- Die angebotenen Lernunterlagen sind zwar hoffentlich hilfreich, aber wahrscheinlich nicht ausreichend. Gelegentlich werden wir zusätzliche Informationen in Büchern oder im Internet nachschlagen. Auf die Empfehlung bestimmter Bücher oder Webseiten wird verzichtet, weil es individuell verschieden ist, welche Art uns anspricht. Informationen auf Webseiten sind von sehr unterschiedlicher Qualität. Das trifft ganz besonders auf typische Programmierbeispiele zu, die zu einem großen Teil von wenig erfahrenen Studierenden stammen oder auf einem anderen als dem in EP2 verfolgten didaktischen Konzept beruhen. Diese Beispiele sollten daher mit Vorsicht genossen werden.

  Internet, Bücher

- Es ist gut, sich Informationen aus Büchern oder dem Internet zu besorgen. Aber wir müssen die dafür aufgewendete Zeit in Grenzen halten. Zeit, die wir in das praktische Lösen von Programmieraufgaben investieren, wird wahrscheinlich einen rascheren Fortschritt bringen (und auch mehr Spaß machen). Idealerweise finden wir selbst kleine Programmieraufgaben, die wir auch gleich selbst (oder in einer Lerngruppe) lösen. In den folgenden Kapiteln finden sich zahlreiche Aufgaben, die als Anhaltspunkte dienen können. Es ist nicht nötig, alle diese Aufgaben zu lösen. Oft ist es vorteilhaft, jene Aufgaben herauszupicken, die den individuell größten Erkenntnisgewinn erwarten lassen, und die Aufgaben bewusst an die jeweilige Situation anzupassen.

  praktisch arbeiten

- Programmieren ist spannend und macht Spaß. Wir müssen nicht immer darauf Bedacht nehmen, welche Kompetenz wir erlernen oder ausbauen wollen. Wir können ruhigen Gewissens ein Programm schreiben, nur weil wir wissen wollen, ob und wie etwas funktioniert, oder weil es gerade Spaß macht.

  Spaß haben

2 Datenabstraktion

Wie bereits in Abschnitt 1.2 erwähnt, ist *Datenabstraktion* ein zentraler Begriff in der Programmierung, vor allem in der Programmierung im Großen. Dabei werden mehrere Methoden und Variablen zu in sich konsistenten Einheiten zusammenzufasst, sodass diese Einheiten logische Bausteine (*abstrakte Datentypen*) bilden, aus denen wir ganze Programme zusammensetzen. Wir brauchen nur abstrakte Vorstellungen solcher Bausteine im Kopf zu haben, wenn wir sie aneinanderfügen, keine Implementierungsdetails – daher die Begriffe. Auf diese Weise hoffen wir die Programmerstellung so weit vereinfachen zu können, dass auch größere Programmieraufgaben handhabbar werden.

Wir betrachten zunächst das Prinzip hinter der Datenabstraktion sowie die entsprechenden Sprachkonstrukte in Java. Es folgen Beispiele, die den fließenden Übergang von Datenabstraktionen zu Datenstrukturen verdeutlichen – eine Unterscheidung, die vor allem in der Betrachtungsweise liegt. Im letzten Abschnitt dieses Kapitels verwenden wir Datenabstraktion zur Realisierung rekursiver Datenstrukturen.

2.1 Prinzip und Sprachunterstützung

Wir führen Objekte, Klassen und verwandte Konzepte in Java ein, nachdem wir die Verwendung einiger einfacher abstrakter Datentypen beispielhaft betrachtet haben.

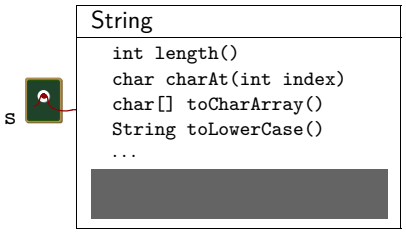
2.1.1 Verwendung abstrakter Datentypen

Erinnern wir uns an `String`, den Typ von Zeichenketten in Java. Das ist nicht nur ein Referenztyp, sondern auch ein abstrakter Datentyp: Wir wissen zwar, dass irgendwo in einer Zeichenkette Daten enthalten sind, aus denen sich die Länge und die einzelnen Zeichen der Zeichenkette rekonstruieren lassen, aber wir wissen nicht wo und in welcher Form diese Daten genau abgelegt sind. Für die Verwendung der Zeichenkette brauchen wir das nicht zu wissen:

Listing 2.1 Auf Daten in Zeichenketten wird nur über Methoden zugegriffen

```
1 private static void printFifthChar(String s) {
2     if (s != null && s.length() > 4) {
3         System.out.print(s.charAt(4));
4     }
5 }
```

Da der formale Parameter `s` wie jede Variable eines Referenztyps `null` enthalten kann, stellen wir sicher, dass `s != null` gilt und `s` eine ausreichende Länge hat um auf das fünfte Zeichen zugreifen zu können – jenes mit Index 4, da das erste Zeichen über Index 0 referenziert wird. Ausdrücke wie `s.length()` und `s.charAt(4)` stellen *Methodenaufrufe* in der Zeichenkette dar, die von `s` referenziert wird.[1] Als Ergebnisse bekommen wir die Länge und das Zeichen zurück. Wir veranschaulichen uns die Zeichenkette so:



Wir sehen, welche Methoden wir aufrufen können bzw. welche Nachrichten die Zeichenkette versteht, aber ein Teil der Zeichenkette ist hinter einem grauen Balken versteckt und für uns nicht einsehbar. Konkret sehen wir nicht, wie die Methoden implementiert sind und wie die Daten aussehen, welche die Zeichen der Zeichenkette darstellen. Das sind Implementierungsdetails. Wir können Zeichenketten problemlos verwenden, ohne uns über die Implementierungsdetails Gedanken machen zu müssen. Eine Voraussetzung dafür ist, dass die in `String` sichtbaren Teile klar beschrieben sind[2] und unsere Intuition unterstützen. Es geht also keineswegs darum, einfach nur komplexe Details wegzulassen. Vielmehr wird bewusst eine abstrakte Vorstellung geschaffen um die Verwendung zu erleichtern. Implementierungsdetails sind in der abstrakten Vorstellung irrelevant. Wir können Implementierungsdetails ändern, ohne die abstrakte Vorstellung zu beeinflussen.

Verwendung ohne Detailwissen

abstrakte Vorstellung

[1] Gleichbedeutend sagen wir auch, dass die *Nachrichten* `length()` und `charAt(4)` an die Zeichenkette in `s` *gesendet* oder *geschickt* werden.

[2] `http://docs.oracle.com/javase/8/docs/api/java/lang/String.html`

Wir kennen auch schon die Verwendung von `Scanner` als abstrakten Datentyp zum Einlesen von Daten:

Listing 2.2 Methoden von `Scanner` ändern unsichtbare Zustände

```
1 private static void echo() {
2     Scanner scanner = new Scanner(System.in);
3     while (scanner.hasNextLine()) {
4         System.out.println(scanner.nextLine());
5     }
6 }
```

Wie auf Zeichenketten wird auf Objekte vom Typ `Scanner` nur über Aufrufe von Methoden zugegriffen. Anders als Zeichenketten verfügen Scanner über interne Zustände, die durch Methodenaufrufe geändert werden: Durch Ausführung von `scanner.nextLine()` ändert sich die Zeile, die als nächste einzulesen ist. Das Weiterschalten von Zeilen können wir uns sehr anschaulich vorstellen, ohne zu wissen, was tatsächlich passiert. Vermutlich reicht es nicht, zum Weiterschalten nur irgendwelche Variablenwerte im Scanner zu ändern. Der Scanner wird auf für uns unbekannte Weise irgendwie mit dem Betriebssystem interagieren. Aufgrund von Datenabstraktion brauchen wir uns darum glücklicherweise nicht zu kümmern. Dieses Beispiel zeigt, dass das abstrakte Modell (Weiterschalten von Eingaben) recht weit von der Implementierung (komplexe Interaktion mit Betriebssystem) entfernt sein kann.

interner Zustand

abstraktes Modell

Listing 2.2 zeigt einen weiteren Aspekt: Bevor wir etwas einlesen können, müssen wir erst ein Objekt vom Typ `Scanner` erzeugen. Im entsprechenden `new`-Ausdruck wird `System.in` als Argument übergeben, das bestimmt, über welche Kanäle Daten einzulesen sind. Details zu solchen Parametern werden wir in Abschnitt 2.1.4 betrachten. Auch ohne Detailwissen ist klar, dass wir bei Verwendung abstrakter Datentypen entscheidenden Einfluss auf die Ausführung nehmen können, von der Erzeugung über den gesamten Verwendungszeitraum, ohne jemals Implementierungsdetails gesehen zu haben.

Verändern ohne Detailwissen

`String` und `Scanner` sind Beispiele für Tausende von vordefinierten, meist gut durchdachten abstrakten Datentypen in der Entwicklungsumgebung von Java. Wir wollen auch eigene abstrakte Datentypen erstellen, die wegen des stärkeren Anwendungsbezugs aber meist nicht so schön in sich konsistent sind wie die vordefinierten. Zur Demonstration entwickeln wir den abstrakten Datentyp `BoxedText` – informell beschrieben in Listing 2.3 durch Kommentare. Schon durch die Beschreibungen können wir verstehen, was hinter `BoxedText` steckt.

Beschreibung = abstrakter Datentyp

Listing 2.3 Kommentar zur Beschreibung von `BoxedText`

```
1  /*******************************************************
2  class BoxedText: Rectangular text within border lines.
3    Width and height of the text are initially 0.
4    Valid indexes for lines range from 0 to height - 1.
5
6  public methods:
7
8    void newDimensions(int width, int height):
9      Sets new text width and height to specified values.
10     Text existing so far is replaced with blank (' ').
11     Requires: width >= 0 and height >= 0.
12
13   void setLine(int index, String txt):
14     Fills line 'index' with 'txt'. If 'txt' is longer
15     than the text width, additional characters are
16     ignored. If 'txt' is shorter, the line is filled
17     with ' '. If 'txt' contains a line break, all text
18     in 'txt' following the line break is ignored.
19     Requires: index >= 0 and index < text height.
20
21   void print():
22     Prints boxed text to standard output. For example,
23     if width is 2, height is 1 and text at line 0 is
24     "ab", the output looks like:
25     ----
26     |ab|
27     ----
28
29   String toString():
30     Returns the same text as printed by 'print'
31     (without line break at the end).
32 *******************************************************/
```

Wir sehen, dass Details beschrieben sind, die für die Verwendung wichtig sind. Andere Details, etwa die Form, in der Text intern abgelegt wird, sind nicht erwähnt. Nachdem wir `BoxedText` implementiert haben, sollte eine Ausführung von

Listing 2.4 Verwendung von `BoxedText` aufgrund der Beschreibung

```
1 private static void testBoxedText() {
2     BoxedText t = new BoxedText();
3     t.newDimensions(10, 3);
4     t.setLine(1, "Das ist ein Text");
5     t.print();
6 }
```


folgende Ausgabe erzeugen:

```
----------
|        |
|Das ist ei|
|        |
----------
```

2.1.2 Datenkapselung

In Java sind abstrakte Datentypen durch Klassen implementiert. So wie Klassen in EP1 eingesetzt wurden, können wir jedoch kaum von abstrakten Datentypen sprechen. Eine Klasse sollten wir dann als Implementierung eines abstrakten Datentyps betrachten, wenn die Klasse

Klasse

- *Objektvariablen* deklariert, die in jedem Objekt der Klasse vorhanden sind – in jedem Objekt eigene Variablen (wobei Deklarationen von Objektvariablen denen lokaler Variablen ähneln, aber nicht in Methoden, sondern direkt in Klassen stehen),

Objektvariable

- *Objektmethoden* (in der Klasse, ohne `static`) definiert, die in Objekten der Klasse ausführbar sind und auf Objektvariablen dieser Objekte zugreifen,

Objektmethode

- nur durch Erzeugung von einem oder mehreren Objekten der Klasse sinnvoll verwendbar ist.

Objekt

Objektvariablen – also Variablen, die zu einem Objekt gehören und im Gegensatz zu lokalen Variablen während der gesamten Lebenszeit eines Objekts existieren – entstehen beim Erzeugen eines Objekts mittels `new`. In der Regel verwenden nur Objektmethoden diese Objektvariablen – siehe Abschnitt 2.1.3. Objektvariablen sind ohne Objektmethoden nur selten sinnvoll, genau wie Objektmethoden ohne Objektvariablen. Das Zusammenfügen von Variablen und Methoden zu Objekten nennen wir *Datenkapselung* – Objekte *kapseln* Daten und Methoden.

Kapselung

Listing 2.5 zeigt eine mögliche Implementierung des in Listing 2.3 beschriebenen abstrakten Datentyps. Eine Ausführung der Methode in Listing 2.4 produziert damit die oben dargestellte erwartete Ausgabe. Eigentlich sollten die Kommentare aus Listing 2.3 bei der Klassendefinition und den Methodendefinitionen stehen. Zwecks besserer Übersicht wurden sie in Listing 2.5 weggelassen.

Implementierung von abstraktem Datentyp

Listing 2.5 Implementierung von `BoxedText`

```
1  public class BoxedText {
2
3      private int textWidth = 0;
4      private int textHeight = 0;
5      private char[][] text = new char[0][];
6
7      public void newDimensions(int width, int height) {
8          textWidth = width;
9          textHeight = height;
10         text = new char[height][width];
11         for (char[] line : text) { fill(line, 0); }
12     }
13
14     public void setLine(int index, String txt) {
15         int i = 0;
16         if (txt != null) {
17             int m = Math.min(textWidth, txt.length());
18             while (i < m && txt.charAt(i) != '\n') {
19                 text[index][i] = txt.charAt(i++);
20             }
21         }
22         fill(text[index], i);
23     }
24
25     private void fill(char[] line, int i) {
26         for (; i < textWidth; i++) {
27             line[i] = ' ';
28         }
29     }
30
31     public void print() {
32         System.out.println(toString());
33     }
34
35     public String toString() {
36         String horizontal = "--";
37         for (int i = 0; i < textWidth; i++) {
38             horizontal += "-";
39         }
40         String result = horizontal + "\n";
41         for (char[] line : text) {
42             result += "|";
43             for (char c : line) { result += c; }
44             result += "|\n";
45         }
46         return result + horizontal;
47     }
48 }
```


In den Zeilen 3 bis 5 in Listing 2.5 werden Objektvariablen deklariert und initialisiert. Wie üblich beginnen die Deklarationen mit dem Wort `private` und stehen direkt in der Klasse (nicht in einer Methode), gleichen sonst aber syntaktisch lokalen Variablendeklarationen.

Java

Die in EP1 verwendeten Methoden begannen (außer `main`) meist mit `private static`. In Listing 2.5 beginnen viele mit `public` und enthalten kein `static`. Die Bedeutung von `private` und `public` werden wir in Abschnitt 2.1.3 analysieren. Zunächst ist das Weglassen von `static` wichtig: Methoden ohne `static` sind *Objektmethoden*. Eine Objektmethode kann auf Objektvariablen in gleicher Weise zugreifen wie auf formale Parameter und lokale Variablen, wie an vielen Stellen in Listing 2.5 zu sehen ist. Wenn ein Methodenkopf dagegen `static` enthält – wir sprechen von *statischen Methoden* oder *Klassenmethoden* – darf die Methode nicht auf Objektvariablen zugreifen. Eine Klasse kann sowohl Objektmethoden als auch Klassenmethoden enthalten; in `BoxedText` stehen aber nur Objektmethoden.

Java

Objektmethode verwendet Objektvariablen

Klassenmethode kennt keine Objektvariablen

Objektmethoden rufen wir anders auf als Klassenmethoden. Für den Aufruf einer Objektmethode benötigen wir ein Objekt der entsprechenden Klasse, für den Aufruf einer Klassenmethode nur die Klasse selbst. Wenn `o` als Objektmethode und `c` als Klassenmethode in Klasse `C` implementiert ist, können wir die Aufrufe so schreiben:

```
C.c(...);       // Aufruf von c in Klasse C
C x = new C();  // Erzeugen von Objekt x der Klasse C
x.o(...);       // Aufruf von o in Objekt x
```

Das hat den Grund, dass bei Ausführung von `o` auf die Objektvariablen des Objekts in `x` zugegriffen werden kann, während zur Ausführung von `c` keine Objektvariablen nötig sind. Konkrete Beispiele sind die Aufrufe von Objektmethoden in Listing 2.4 sowie der Aufruf der Klassenmethode `min` aus `Math` in Zeile 17 von Listing 2.5.

Innerhalb der Klasse `BoxedText` sehen wir das Objekt, das die zugegriffenen Objektvariablen enthält, nicht direkt (wir könnten aber über `this` darauf zugreifen – siehe Abschnitt 2.1.4). Bei Zugriffen schreiben wir einfach nur die Namen der Objektvariablen hin, ohne dazuzuschreiben, dass es sich z. B. um die Objektvariablen des Objekts handelt, das in der Variablen `t` von Listing 2.4 steht. Wie wir in Listing 2.6 sehen, kann sich ein Zugriff auf eine Objektvariable manchmal auf das Objekt in `x` beziehen, dann wieder auf das in `y`. So wie eine Ausführung von `x.print()` nur die Objektvariablen von `x` verwendet, greift eine Aus-

aktuelles Objekt

Listing 2.6 Überlappende Verwendungen mehrerer Objekte derselben Klasse

```
1  private static void testBoxedText2() {
2      BoxedText x = new BoxedText();
3      BoxedText y = new BoxedText();
4      x.newDimensions(10, 3);
5      y.newDimensions(20, 1);
6      x.setLine(1, "Das ist ein Text");
7      y.setLine(0, "Anderer Text");
8      x.print();
9      y.print();
10 }
```

führung von `y.print()` nur auf Objektvariablen von `y` zu. Es werden zwei unterschiedliche umrandete Texte ausgegeben.

In Zeile 32 von Listing 2.5 wird `toString()` aufgerufen, ohne anzugeben, in welchem Objekt die Methode ausgeführt werden soll (das Objekt, das normalerweise links von „.“ steht). Es wird die ab Zeile 35 definierte Objektmethode aufgerufen. Wir können in einer Objektmethode (nicht in einer Klassenmethode) eine andere Objektmethode desselben Objekts aufrufen, ohne dabei das Objekt anzugeben; es ist ja ohnehin dasselbe Objekt. Auf diese Weise wird auch `fill` in den Zeilen 11 und 22 von Listing 2.5 aufgerufen. Wenn wir eine Klassenmethode ohne Klasse und Punkt vor dem Methodennamen aufrufen, wird die Methode dieses Namens in der gleichen Klasse (unabhängig von einem Objekt) ausgeführt.

Aufruf im selben Objekt

Aufruf in selber Klasse

Klassenvariable

Neben Objektvariablen unterstützt Java Klassenvariablen, die mit `static` in einer Klasse deklariert sind. Eine Klassenvariable existiert nur einmal pro Klasse, nicht in jedem Objekt. Auf Klassenvariablen können Klassenmethoden und Objektmethoden zugreifen. Der sinnvolle Einsatz von Klassenvariablen mit veränderlichen Werten ist schwierig. Wir verwenden Klassenvariablen daher meist nur als Konstanten (`static final` Variablen). Gelegentlich greifen wir über vordefinierte Klassen auf Klassenvariablen zu, z. B. auf `in` und `out` in der Klasse `System`: In Zeile 32 von Listing 2.5 wird die Objektmethode `println` in dem Objekt aufgerufen, das in `out` von `System` steht.

Aufgabe 2.1 Aus didaktischen Gründen verzichten wir in Beispielprogrammen auf Kommentare; Programme sollen durch ihr Verhalten verstanden werden, nicht nur intuitiv durch Namen und Kommentare. Schreiben Sie zum Kopf jeder Methode in allen Beispielen, auch den noch folgenden, einen kurzen, aussagekräftigen Kommentar, der besagt,

wozu die Methode dient und worauf bei der Verwendung zu achten ist, sodass die *Außensicht* deutlich erkennbar wird. Schreiben Sie außerdem zur Deklaration jeder Objektvariablen einen kurzen Kommentar, der besagt, wozu die Variable dient.

Aufgabe 2.2 Beschreiben Sie Gemeinsamkeiten und Unterschiede zwischen *Klassen-* und *Objektmethoden.* Beschreiben Sie, anhand welcher Kriterien Sie sich für welche Art von Methode entscheiden.

Aufgabe 2.3 Veranschaulichen Sie sich, welche Variablen bei Ausführung der Methode in Listing 2.6 (auch in dabei erzeugten Objekten) angelegt werden und welche Werte sie nach jedem Ausführungsschritt enthalten. Beschreiben Sie die einzelnen Schritte.

2.1.3 Data-Hiding

Außensicht

Innensicht

Datenabstraktion ist die kombinierte Anwendung von Datenkapselung und *Data-Hiding,* das „Verstecken“ von Implementierungsdetails vor Zugriffen von außen. Durch Data-Hiding bekommen wir zwei verschiedene Sichten auf einen abstrakten Datentyp, die abstrakte *Außensicht* wie in Listing 2.3 und die Implementierungsdetails offenlegende *Innensicht* wie in Listing 2.5. Wir brauchen beide, die Außensicht für die Verwendung und die Innensicht für die Implementierung des abstrakten Datentyps. Diese Sichten unterscheiden sich meist wesentlich voneinander, weil verschiedene Interessen dahinterstecken:

- Die Außensicht ist die Anwendersicht. Als Zusammenfassung von Listing 2.3 wird man zuerst mit `newDimension` die Größe eines umrandeten Texts bestimmen, danach über `setLine` Textinhalte in die einzelnen Zeilen schreiben, über `print` ausgeben und bei Bedarf einzelne Schritte wiederholen, wobei zu bedenken ist, dass `newDimension` alte Textinhalte löscht.

  Außensicht für Anwender

- Die Innensicht konzentriert sich auf die Implementierung der Methoden mit den nötigen Objektvariablen. Variablen und Typen werden so gewählt, dass die Implementierungen möglichst einfach sind und dennoch den Beschreibungen entsprechen.

  Innensicht zum Implementieren

Wir müssen die Sichten in Einklang zueinander bringen. Die Außensicht entwerfen wir so, dass eine Implementierung mit vertretbarem Aufwand möglich scheint. Danach gestalten wir die Innensicht, sodass alle nötigen Aspekte der Außensicht bestmöglich erfüllt sind.

Java

public überall sichtbar

private nur in Innensicht

eher private

Bei Data-Hiding geht es darum, welche Methoden und Variablen[3] als `public` und welche als `private` definiert bzw. deklariert sind.[4] Mit `public` kennzeichnen wir alles, was in der Außensicht von Bedeutung ist, mit `private` jenes, was zur Implementierung der Innensicht benötigt wird und von außen nicht gesehen werden muss. Da Objekt- und Klassenvariablen fast immer Implementierungsdetails darstellen, sollten sie auch fast immer mit `private` deklariert sein. Bei Methoden unterscheiden wir zwischen den auch von außen aufrufbaren `public` Methoden und als `private` definierten *Hilfsmethoden.* Beispielsweise ist `fill` in Listing 2.5 eine Hilfsmethode, da sie in Listing 2.3 nicht vorkommt und daher als `private` definiert ist. Wir definieren `fill` als `private` weil wir `fill` von außen nicht verwenden *wollen.* Es stimmt zwar, dass wir `fill` wegen `private` von außen nicht verwenden *dürfen,* aber wenn wir dies wollten, hätten wir `fill` als `public` definiert.

Es ist vorteilhaft, Abstraktionen einfach zu halten. Das heißt, möglichst viel von der unvermeidbaren Komplexität soll in der Innensicht abstrakter Datentypen versteckt bleiben. Wir verwenden `private` daher so oft wie möglich, `public` nur dann, wenn die Verwendung in der Außensicht unverzichtbar ist. Dies wirkt sich positiv aus, wenn die Implementierung eines abstrakten Datentyps, beispielsweise zur Korrektur von Fehlern, geändert werden muss. Solange nur die Innensicht von der Änderung betroffen ist, bleibt die Änderung auf die Implementierung des abstrakten Datentyps beschränkt, andere Programmteile müssen nicht geändert werden. Ist die Außensicht betroffen, müssen auch viele Stellen, an denen der abstrakte Datentyp verwendet wird, geändert werden. Das können uns unbekannte Stellen sein. Indem wir die Außensicht einfach halten, erhöhen wir die Wahrscheinlichkeit dafür, dass nötige Änderungen nur die Innensicht betreffen. Würden wir Listing 2.5 so ändern, dass `text` vom Typ `String[]` wäre, müssten wir zwar fast jede Methode in Listing 2.5 umschreiben, da aber nur die Innensicht betroffen wäre, würden Aufrufe wie in Listing 2.4 und Listing 2.6 weiterhin funktionieren. Würden wir dagegen den Typ des formalen Parameters `txt` in `setLine` auf `char[]` ändern, wäre die Außensicht betroffen und wir müssten alle Listings 2.3 bis 2.6 ändern.

[3] Gemeint sind hier Objektmethoden, Klassenmethoden, Objektvariablen und Klassenvariablen. Wir sprechen vereinfacht von Methoden und Variablen, wenn es im entsprechenden Kontext keine Rolle spielt, ob `static` dabei steht oder nicht.

[4] Neben `public` und `private` gibt es noch `protected` und Default-Sichtbarkeit (ohne angegebenen Modifier für die Sichtbarkeit). Wir verwenden vorläufig nur `public` und `private`, da diese Sichtbarkeiten vollkommen ausreichen.

Listing 2.7 Klasse mit Getter-Methoden. Ein Parameter hat den Typ der Klasse

```
1  // A point is a position in a two-dimensional space.
2  public class Point {
3      private int x = 0, y = 0;  // coordinates of point
4
5      public void set(int newX, int newY) {
6          x = newX;
7          y = newY;
8      }
9
10     public int getX() {
11         return x;
12     }
13
14     public int getY() {
15         return y;
16     }
17
18     // let p be the new origin of this point
19     public void origin(Point p) {
20         x -= p.x;
21         y -= p.y;
22     }
23 }
```

Ohne entsprechende Erfahrung finden wir den Umgang mit abstrakten Datentypen oft schwierig, da wir von außen nicht direkt auf Objektvariablen zugreifen können. Das ist Gewohnheitssache. Sobald wir gute abstrakte Datentypen entwickeln können, sehen wir kaum noch Bedarf für Variablenzugriffe von außen. In der Zwischenzeit können wir uns mit *Setter-* und *Getter-Methoden* behelfen: Diese meist von außen sichtbaren Methoden setzen den Wert einer `private` Variablen bzw. geben den aktuellen Wert in der Variablen zurück. Listing 2.7 zeigt ein Beispiel: `getX` und `getY` sind ganz typische Getter-Methoden; `set` ist dagegen keine typische Setter-Methode, weil mehr als nur eine Variable gesetzt wird. Getter- und Setter-Methoden sind aus dem Blickwinkel der Wartung problematisch, aber oftmals besser als `public` Variablen.

Getter, Setter

Durch `public` und `private` wird der Unterschied zwischen der Außen- und Innensicht eines Objekts verdeutlicht. Aber es stimmt nicht ganz, dass wir nur innerhalb eines Objekts auf `private` Variablen und Methoden zugreifen dürfen. In der Methode `origin` in Listing 2.7 greifen wir direkt auf die Variablen `x` und `y` eines (wahrscheinlich) anderen Objekts vom Typ `Point` zu; wir schreiben direkt `p.x` statt `p.getX()`.

Java

Das ist möglich, weil der deklarierte Typ von `p` gleich der Klasse ist, in der wir uns befinden. Tatsächlich bedeutet `private`, dass nur innerhalb der *Klasse* auf mit diesem Modifier versehene Variablen und Methoden zugegriffen werden kann, unabhängig davon, zu welchem Objekt eine Objektvariable oder Objektmethode gehört. Diese Bedeutung von `private` ist praktisch und logisch, weil ohnehin alle Objekte derselben Klasse dieselbe Innensicht haben. In `origin` kommt die Außensicht von `p` zum Tragen (weil wir ein Objekt vom Typ `Point` *verwenden,* nicht implementieren), gleichzeitig aber auch die Innensicht von `Point` (`origin` ist in `Point` implementiert).

public Klasse

Auf Klassen bedeutet `public` etwas anderes. Jede `public` Klasse muss in einer Datei desselben Namens (abgesehen von der Erweiterung) stehen, und nur solche Klassen sind allgemein verwendbar. Vorläufig definieren wir jede Klasse mit `public class`. Wenn wir `public` weglassen, kann die Klasse nur als *Hilfsklasse* verwendet werden, die von einer `public` Klasse kontrolliert wird – vergleichbar mit Hilfsmethoden in Klassen. Verwendungsmöglichkeiten werden wir später sehen.

Aufgabe 2.4 Ändern Sie Listing 2.5, sodass `text` vom Typ `String[]` ist, aber dennoch alle Details von Listing 2.3 eingehalten werden. Beschreiben Sie anhand dieses Beispiels und allgemein, wie abstrakte Datentypen *Programmänderungen unterstützen* können.

Aufgabe 2.5 Beschreiben Sie, was wir unter *Datenkapselung* und *Data-Hiding* verstehen, wodurch sich diese beiden Begriffe voneinander unterscheiden und wie sie sich gegenseitig ergänzen.

Aufgabe 2.6 Beschreiben Sie, warum Objektvariablen und (sofern möglich) auch Methoden `private` sein sollen.

Aufgabe 2.7 Beschreiben Sie Zusammenhänge zwischen der Verwendung von `private` bzw. `public` und der *Innen-* bzw. *Außensicht* von Objekten. Machen Sie deutlich, welche `private` Variablen und Methoden auch über die Außensicht eines Objekts zugreifbar sind.

Aufgabe 2.8 Beschreiben Sie die Bedeutung von `public` auf Klassendefinitionen.

2.1.4 Objekterzeugung

Java

Ein neues Objekt erzeugen wir mit dem `new`-Operator. Beispielsweise liefert die Ausführung von `new Point()` ein neues Objekt vom Typ

Listing 2.8 Klasse mit Konstruktor zur Initialisierung von Objekten

```
1 public class Point {
2     private int x, y;
3
4     public Point(int initX, int initY) {
5         x = initX;
6         y = initY;
7     }
8     ...
9 }
```

`Point`. Jedes so erzeugte Objekt hat eine eigene *Identität.* Wir stellen uns das vereinfacht so vor, dass beim Erzeugen eines Objekts Speicherplatz reserviert wird, in dem Objektvariablen und zur Objektverwaltung nötige Daten abgelegt werden. Die Anfangsadresse des Speicherplatzes entspricht der Objektidentität. Variablen von Referenztypen enthalten solche Anfangsadressen und referenzieren damit Objekte. Zwei Anwendungen des `new`-Operators liefern zwei Objekte mit unterschiedlichen Speicheradressen, diese Objekte sind nicht identisch. Wenn jedoch zwei Variablen dieselbe Objektadresse enthalten, sind die von diesen Variablen referenzierten Objekte identisch – nur ein Objekt mit zwei Referenzen darauf.

Identität

Ein wesentlicher Schritt bei der Objekterzeugung ist die Initialisierung von Objektvariablen. Anders als bei lokalen Variablen gibt es für Objektvariablen keine klar erkennbare sequentielle Abfolge von Variablenzugriffen. Der Compiler kann Lesezugriffe unmittelbar nach der Objekterzeugung (noch vor der Initialisierung) nicht ausschließen. Daher werden alle Objekt- und Klassenvariablen automatisch mit `null` oder `0` bzw. dem Null-Wert des entsprechenden Typs vorinitialisiert bevor die Objekterzeugung abgeschlossen ist. Erst nach abgeschlossener Objekterzeugung erfolgt die eigentliche Initialisierung der Objektvariablen mit den Werten, die in den Deklarationen angegeben sind – siehe Zeile 3 in Listing 2.7 und die Zeilen 3 bis 5 in Listing 2.5.

Initialisierung

In `new Scanner(System.in)` wird ein Argument (aktueller Parameter) verwendet, das vermutlich irgendwie (auf unbekannte Weise) zur Initialisierung von Objektvariablen dient. Das Sprachkonzept dahinter sind *Konstruktoren.* Sie ähneln Objektmethoden, dienen aber nur zur Initialisierung neuer Objekte (Zeilen 4 bis 7 in Listing 2.8). Die Erzeugung eines Objekts von `Point` (entsprechend Listing 2.8) erfolgt z. B. mit `new Point(4, 5)`, wobei der Konstruktor unmittelbar

Konstruktor

nach Erzeugung des Objekts ausgeführt wird. Die (formalen) Parameter `initX` und `initY` enthalten die entsprechenden Argumente `4` und `5`, welche an `x` und `y` zugewiesen werden und dabei die vorinitialisierten Variableninhalte ersetzen. Syntaktisch unterscheidet sich ein Konstruktor von einer Objektmethode dadurch, dass der Ergebnistyp fehlt und der Name gleich dem Klassennamen ist. Der Rumpf des Konstruktors kann beliebige Anweisungen enthalten, genauso wie eine Objektmethode mit Ergebnistyp `void`. Üblicherweise dient ein Konstruktor nur zur Initialisierung von Objektvariablen, wobei die Berechnung der Initialisierungswerte auch aufwendig sein kann. Aufgrund dieser Verwendung versteht es sich von selbst, dass Konstruktoren nicht `static` sein können. Bezüglich der Sichtbarkeit ist alles möglich, wobei mittels eines als `private` deklarierten Konstruktors nur innerhalb der Klasse ein Objekt dieser Klasse erzeugt werden kann.

Nach `new` steht stets ein Konstruktoraufruf. Daher benötigt jede Klasse, von der es Objekte geben soll, zumindest einen Konstruktor. In bisherigen Beispielen haben wir jedoch keine Konstruktoren definiert. Das ist möglich, weil der Compiler für jede Klasse ohne definierten Konstruktor (und nur für solche Klassen) automatisch einen *Default-Konstruktor* einführt, das ist ein Konstruktor folgender Form:

Default-Konstruktor

```
public Klassenname() {}
```

Dieser Konstruktor ist also überall sichtbar, hat keine Parameter und macht bei der Ausführung nichts. Entsprechend Listing 2.8 müssen wir bei der Objekterzeugung unbedingt zwei Argumente übergeben, weil es einen expliziten Konstruktor mit zwei Parametern, aber keinen Default-Konstruktor gibt. Anders für die Klassen in Listing 2.5 und Listing 2.7: Hier gibt es nur den Default-Konstruktor, weswegen wir keine Argumente angeben.

überladene Konstruktoren

Eine Klasse kann mehrere Konstruktoren enthalten. In diesem Fall sind die Konstruktoren *überladen,* genauso wie Methoden überladen sein können. Der Compiler entscheidet anhand der Anzahl und der deklarierten Typen der Argumente, welcher Konstruktor auszuführen ist. Listing 2.9 zeigt eine Klasse mit mehreren Konstruktoren. Damit verwendet `new Point(2, 3)` den ersten Konstruktor der Klasse, `new Point()` den zweiten und `new Point(new Point())` den zweiten sowie dritten.

Weiterleitung zu Konstruktor

Listing 2.9 zeigt Anweisungen der Form `this(...);`. Sie rufen Konstruktoren derselben Klasse auf und führen sie auf gleiche Weise wie Methoden aus. Konkret führt `this(1, 1);` im Rumpf des zweiten

Listing 2.9 Klasse mit mehreren Konstruktoren, `this` und `this(...);`

```
1  public class Point {
2      private int x, y;
3
4      public Point(int x, int y) {
5          this.x = x;
6          this.y = y;
7      }
8      public Point() {
9          this(1, 1);
10     }
11     public Point(Point p) {
12         this(p.x, p.y);
13     }
14
15     public Point copy() {
16         return new Point(this);
17     }
18
19     ...
20 }
```

Konstruktors den ersten Konstruktor aus, `x` und `y` werden mit `1` initialisiert. Im dritten Konstruktor führt `this(p.x, p.y);` ebenso den ersten Konstruktor aus, sodass eine Kopie von `p` (also ein neues Objekt mit den gleichen Werten in den Objektvariablen) entsteht. Es gibt jedoch eine bedeutende Einschränkung: Anweisungen der Form `this(...);` dürfen nur ganz am Anfang eines Konstruktors stehen, sonst nirgends. Wenn Programmtexte wie Methoden aufgerufen werden sollen, müssen wir auch Methoden verwenden, nicht Konstruktoren. Methoden sind in Konstruktoren uneingeschränkt aufrufbar.

Selbstreferenz

In Listing 2.9 wird auch die *Pseudovariable* `this` verwendet. Pseudovariable bedeutet, dass `this` zwar wie eine normale Variable gelesen, ihr Wert von uns aber nicht selbst geschrieben werden kann. Der Wert von `this` ist stets eine Referenz auf das Objekt, in dem wir uns gerade befinden und zu dem die gerade direkt zugreifbaren Objektvariablen gehören. Innerhalb eines Konstruktors ist es das Objekt, das gerade initialisiert wird. Ist `x` eine Objektvariable, können wir statt `x` auch `this.x` schreiben um deutlich zu machen, dass das `x` des aktuellen Objekts gemeint ist – im Gegensatz zu `p.x` eines anderen Objekts `p`. In den Zeilen 5 und 6 von Listing 2.9 machen wir das aus einem ganz bestimmten Grund: Die Parameter des ersten Konstruktors heißen zufällig genau gleich wie die Objektvariablen. In einem solchen Fall *ver-*

verdeckte Variable

decken die Parameter die Objektvariablen, das heißt, mit `x` bezeichnen wir innerhalb des ersten Konstruktors den Parameter, nicht die gleichnamige Objektvariable. Mittels `this.x` greifen wir dagegen auf die Objektvariable zu, nicht den Parameter. Generell können Parameter und lokale Variablen Objektvariablen verdecken, wobei die Objektvariablen über `this` dennoch zugreifbar bleiben. In Zeile 16 verwenden wir die Pseudovariable `this` weil das Objekt, in dem wir uns gerade befinden, als Argument verwendet wird. Auch dieser Fall tritt häufig auf. Innerhalb von `static` Methoden ist `this` nicht verwendbar, da dort kein aktuelles Objekt zugreifbar ist und keine Objektvariablen sichtbar sind.

Aufgabe 2.9 Beschreiben Sie, warum und womit Objektvariablen automatisch *vorinitialisiert* werden.

Aufgabe 2.10 Beschreiben Sie, wozu und wie *Konstruktoren* verwendet werden (auch dann, wenn im Programmtext einer Klasse kein Konstruktor steht) und wodurch sie sich von Methoden unterscheiden.

Aufgabe 2.11 Beschreiben Sie, wozu `this` und `this(...)` dienen und wo diese Ausdrücke verwendbar sind.

2.2 Datenstrukturen und abstrakte Datentypen

Unter einer *Datenstruktur* verstehen wir eine Beschreibung der Art und Weise, wie die Daten dargestellt sind und wie sie zusammenhängen, sowie der Operationen, über die wir auf diese Daten zugreifen. Die Beschreibung legt das Verhalten der Operationen fest und *abstrahiert* von der konkreten Implementierung. Wir können allgemeine Eigenschaften der Datenstruktur gänzlich unabhängig von einer bestimmten Implementierung oder Programmiersprache betrachten.

Datenstruktur ≠ Datenabstraktion

Eine Datenstruktur wird meist als abstrakter Datentyp implementiert. Die Implementierung betrachten wir auf abstrakter Ebene aus zumindest zwei Blickwinkeln – als Datenstruktur und als Außensicht des abstrakten Datentyps. Diese Blickwinkel geben in der Regel unterschiedliche Details der Implementierung preis. Die Datenstruktur muss auch alle Zusammenhänge zwischen den Daten detailliert beschreiben, die für die Außensicht des abstrakten Datentyps bedeutungslos sind. Umgekehrt muss die Außensicht Typen formaler Parameter in `public` Methoden (das sind Operationen aus dem Blickwinkel der Datenstruktur) im Detail beschreiben, die für die Datenstruktur keine Rolle spielen. Übergänge sind fließend. Wir sprechen eher von Datenstrukturen, wenn wir uns auf Beziehungen zwischen Daten konzentrieren und von abstrakten Datentypen, wenn es um die Implementierung oder deren Außensicht geht.

Betrachten wir zum Beispiel ein Array. Als Datenstruktur ist ein Array eine Aufeinanderfolge direkt hintereinander stehender Arrayeinträge, auf die sowohl schreibend als auch lesend über einen Index zugegriffen werden kann. Die Größe ist beschränkt und unveränderlich. Für die Beschreibung des Arrays als Datenstruktur spielt es keine Rolle, welche Typen die Arrayeinträge und Indexe haben und welche Syntax für Arrayzugriffe verwendet wird. Als abstrakter Datentyp von außen betrachtet sind die Typen der Arrayeinträge und Indexe von großer Bedeutung, ebenso die Syntax der Zugriffe. Auch aus der Außensicht des abstrakten Datentyps ist die Arraygröße beschränkt und unveränderlich, und es kann sowohl lesend als auch schreibend zugegriffen werden. Aber es spielt keine Rolle, wie die Arrayeinträge angeordnet sind (direkt hintereinander oder sonst irgendwie), solange es eine fixe Zuordnung zwischen Index und Eintrag gibt.

2.2.1 Datensätze

Zu den einfachsten Datenstrukturen zählt der *Datensatz*. Er besteht aus einer vorgegebenen Menge zusammengehöriger Variablen, auf die lesend und bei Bedarf schreibend zugegriffen wird. Zum Beispiel ist ein durch `new Student(...)` erzeugtes Objekt ein Datensatz, wobei Listing 2.10 den entsprechenden abstrakten Datentyp implementiert.

Neben den `private` Variablen des Datensatzes enthält die Klasse nur einen `public` Konstruktor und `public` Getter- und Setter-Methoden entsprechend der Zugriffsoperationen der Datenstruktur. Das ist typisch für eine direkte Implementierung eines Datensatzes.

Ein Datensatz verbindet die darin enthaltenen Variablen zu einer Einheit. Wenn wir ein Objekt vom Typ `Student` als Argument an eine Methode übergeben, hat die aufgerufene Methode über die Getter- und Setter-Methoden Zugriff auf alle darin enthaltenen Variablen. Noch wichtiger ist, dass andere Datenstrukturen wie Arrays Datensätze enthalten und Methoden Datensätze als Ergebnisse zurückgeben können – siehe Listing 2.11. Auf diese Weise kann ein einziger Arrayeintrag in gewissem Sinn mehrere Variablen enthalten und ein einziger Methodenaufruf mehrere Werte als Ergebnis zurückgeben.

Kommentare wurden in Listing 2.10 weggelassen. Wir können uns dennoch gut vorstellen, was die Methoden machen: „Sets/Returns . . . “

Listing 2.10 Einfacher Datensatz mit einfachen Zugriffsoperationen

```
1  public class Student {
2      private final int regNumber;
3      private String name;
4      private String mail;
5
6      public Student(int regNumber, String name) {
7          this.regNumber = regNumber;
8          setName(name);
9          setMail("e"+regNumber+"@student.tuwien.ac.at");
10     }
11
12     public int regNumber() {
13         return regNumber;
14     }
15
16     public String getName() {
17         return name;
18     }
19
20     public void setName(String name) {
21         this.name = name;
22     }
23
24     public String getMail() {
25         return mail;
26     }
27
28     public void setMail(String mail) {
29         this.mail = mail;
30     }
31 }
```

Listing 2.11 Datensätze erleichtern den Umgang mit zusammenhängenden Daten

```
1  private static Student find(Student[] studs, int reg) {
2      for (Student stud : studs) {
3          if (stud.regNumber == reg) {
4              return stud;
5          }
6      }
7      return new Student(reg, "Max Mustermann");
8  }
```


Wir betrachten die Menge dieser gedachten Methodenbeschreibungen gleichzeitig als Außensicht des abstrakten Datentyps und als Datenstruktur. Bei einfachen Datensätzen sind die Daten unabhängig voneinander, sodass die Datenstruktur nicht mehr beschreiben muss als die Außensicht des abstrakten Datentyps. Aber die Außensicht gibt Details preis, über die die Datenstruktur abstrahiert: Die Typen `String` und `int` der Variablen werden in den Getter- und Setter-Methoden sichtbar und sind bedeutende Bestandteile der Außensicht, da Wissen über diese Typen für die Verwendung notwendig ist. Das Ergebnis von `regNumber()` kann etwa nur zwischen -2^{31} und $2^{31}-1$ liegen. Bei Betrachtung einer Datenstruktur abstrahieren wir über solche Details, damit mögliche Implementierungen nicht unnötig eingeschränkt werden. Es kann (sowohl bei Betrachtung als abstrakter Datentyp als auch als Datenstruktur) beabsichtigt sein, dass der Wertebereich von `regNumber()` uneingeschränkt bleibt. Wäre der Wertebereich vor einigen Jahren auf 0 bis 9999999 eingeschränkt worden, hätte die 2017 erfolgte Umstellung auf achtstellige Matrikelnummern Programmänderungen nötig gemacht. Auch durch die Verwendung von `int` ist Listing 2.10 so eingeschränkt, dass eine Umstellung auf zehnstellige Matrikelnummern Änderungen erfordern würde.

Details auch in Außenansicht

Die Verwendung eines Datensatzes wie in Listing 2.10 mag uns zunächst als sehr einfach erscheinen. Wenn wir jedoch `Student` in ein Programm einbetten, das Daten zu Studierenden verwaltet, werden wir bald bemerken, dass der größte Teil des Programms außerhalb von `Student` geschrieben werden muss. Alleine das Einlesen neuer, zu ändernder Daten benötigt viele Programmzeilen. Ein stärker mit Funktionalität angereicherter abstrakter Datentyp, wie in Listing 2.12 skizziert, verlagert den Programmtext ins Innere der Klasse.

einfacher Datensatz → externe Methoden

Die Grundstruktur bestehend aus Objektvariablen und Konstruktor ist im Vergleich zu Listing 2.10 gleich geblieben. Aber die Methoden sind wesentlich spezifischer auf eine bestimmte Anwendung zugeschnitten. Sie enthalten die komplette Funktionalität um persönliche Daten auszugeben und zu editieren sowie Mails an Studierende zu versenden. Damit ist es einfacher, ein Programm zur Verwaltung von Studierenden zu schreiben, weil ein großer Teil des nötigen Programmtexts bereits in `Student` enthalten ist. Außer für anwendungsneutrale abstrakte Datentypen, die man häufig in vorgefertigten Klassenbibliotheken findet, sollten wir generell anwendungsspezifische Operationen wie in Listing 2.12 bevorzugen und Getter- und Setter-Methoden meiden. Allerdings ist die Datenstruktur eines Datensatzes in Listing 2.12 schwer erkennbar,

spezifische Operationen in die Klasse

Listing 2.12 Datensatz mit anwendungsspezifischen Operationen

```
1  public class Student {
2      private final int regNumber;
3      private String name;
4      private String mail;
5
6      public Student(int regNumber, String name) {
7          this.regNumber = regNumber;
8          this.name = name;
9          mail = "e"+regNumber+"@student.tuwien.ac.at";
10     }
11
12     public void showPersonalData() { ... }
13     public void editPersonalData() { ... }
14     public void mail(String head, String text) { ... }
15 }
```

weil die Außensicht keine Möglichkeit zum direkten Zugriff auf die Daten bietet. Der Betrachtungswinkel liegt schwerpunktmäßig auf dem des abstrakten Datentyps, obwohl es sich auch um eine Datenstruktur handelt. Methoden wie `mail`, die sich nicht primär auf Daten beziehen, werden kaum als Operationen einer Datenstruktur angesehen.

Aufgabe 2.12 Vervollständigen Sie Listing 2.12 und schreiben Sie ein kleines Programm, das alle Methoden aus Listing 2.12 aufruft.

Aufgabe 2.13 Beschreiben Sie den Unterschied zwischen der Abstraktion durch *Algorithmen bzw. Datenstrukturen* und der Abstraktion durch *abstrakte Datentypen.* Erklären Sie, in welcher Rolle (z. B. beim Entwerfen, Implementieren, Anwenden, Analysieren und Warten von Programmtexten) wir Bedarf an welcher Form der Abstraktion haben.

Aufgabe 2.14 Wählen Sie zwei Gegenstände aus Ihrer Umgebung (z. B. Stift und Papier) und erstellen Sie entsprechende *Datensätze,* die typische Eigenschaften der Gegenstände durch Objektvariablen und Objektmethoden festlegen. Schreiben Sie eine Methode, die solche Datensätze ähnlich der realen Gegenstände miteinander agieren lässt. Begründen Sie, warum Sie Getter- und Setter-Methoden bzw. mit Funktionalität angereicherte Methoden verwendet haben.

2.2.2 Lineare Zugriffe

Im Gegensatz zu einem Datensatz kann eine *Queue* und ein *Stack* gleichzeitig beliebig viele Daten derselben Art enthalten. Allerdings

Listing 2.13 Vergleich zwischen Queue und Stack

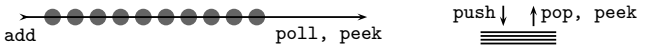
```
1  private static void testQueueAndStack() {
2      final SQueue q = new SQueue(); // Queue aus Listing 2.14
3      final SStack s = new SStack(); // Stack aus Listing 2.15
4      for (char c = 'A'; c <= 'Z'; c++) {
5          q.add("" + c);
6          s.push("" + c);
7      }
8      for (int i = 0; i < 27; i++) {
9          System.out.print(q.peek() + s.peek());
10         System.out.println(q.poll() + s.pop());
11     }
12 }
```

kann nur linear auf die Daten zugegriffen werden. Das heißt, im Gegensatz zu einem Array kann man nicht beliebige Einträge lesen oder schreiben, sondern es gibt jeweils nur genau eine Stelle, an der Daten eingefügt, und eine Stelle, von der Einträge gelesen werden. Einerseits schränkt diese Eigenschaft die Verwendbarkeit ein, andererseits kommen wir bei der Verwendung ohne komplizierte Indexberechnungen aus und brauchen beim Anlegen keine Größe anzugeben.

Eine Queue stellen wir uns als Perlenschnur vor, bei der Perlen (Daten) an einem Ende auf die Schnur gefädelt und am anderen Ende wieder von der Schnur genommen werden. Die erste aufgefädelte Perle wird auch als erste wieder heruntergenommen – *FIFO-Verhalten* (First-In, First-Out). Einen Stack stellen wir uns dagegen als einen Stapel von Tellern (Daten) vor, wo neu hinzukommende Teller oben auf den Stapel gelegt und benötigte Teller ebenso von oben vom Stapel genommen werden. Der zuletzt auf den Stapel gelegte Teller wird als erster vom Stapel genommen – *LIFO-Verhalten* (Last-In, First-Out).

Queue: FIFO

Stack: LIFO



Üblicherweise bezeichnet `push` die Operation zum Ablegen eines Eintrags auf einem Stack und `pop` die zum Lesen und gleichzeitigen Entfernen des obersten Stack-Eintrags. Häufig, aber nicht immer gibt es eine Operation (hier `peek` genannt) zum Lesen des obersten Stack-Eintrags ohne ihn zu entfernen. Entsprechende Operationen gibt es auch auf Queues, wobei die Benennung nicht einheitlich ist. Wir bezeichnen mit `add` die Operation zum Einfädeln, mit `poll` die zum Lesen und Entfernen, und mit `peek` die zum Lesen ohne Entfernen.

Listing 2.14 Wrapper-Implementierung einer Queue

```
1 public class SQueue {
2     private final DEQueue q = new DEQueue();
3     public void add(String e) { q.addLast(e); }
4     public String poll() { return q.pollFirst(); }
5     public String peek() { return q.peekFirst(); }
6     public int size() { return q.size(); }
7 }
```

Listing 2.15 Wrapper-Implementierung eines Stacks

```
1 public class SStack {
2     private final DEQueue s = new DEQueue();
3     public void push(String e) { s.addFirst(e); }
4     public String pop() { return s.pollFirst(); }
5     public String peek() { return s.peekFirst(); }
6 }
```

In Listing 2.13 ist die Verwendung einer Queue der eines Stacks gegenübergestellt, beide enthalten Zeichenketten. Eine Ausführung der Methode zeigt das FIFO- bzw. LIFO-Verhalten: Die Zeichenketten in der Queue werden in derselben Reihenfolge ausgelesen und ausgegeben wie sie eingefügt wurden, jene im Stack in umgekehrter Reihenfolge. Nach dem Lesen eines Eintrags mit `peek` wird (bei wiederholtem Lesen) immer wieder derselbe Eintrag zurückgegeben, nach dem Lesen eines Eintrags mit `poll` bzw. `pop` dagegen ein anderer.

Double Ended Queue

Die Listings 2.14 und 2.15 implementieren Queue und Stack. Methoden delegieren ihre Aufgaben im Wesentlichen nur an Methoden eines anderen Objekts. Klassen mit derartigen Methoden heißen *Wrapper*. Die eigentliche Implementierung steht in Listing 2.16. Wir betrachten `DEQueue` (abgekürzt für Double-Ended-Queue) als Verallgemeinerung von Queue und Stack, vorstellbar als Perlenschnur, bei der Perlen an beiden Enden aufgefädelt und weggenommen werden:

Beim Einfügen und Entfernen an unterschiedlichen Enden ergibt sich das Verhalten einer Queue, bei gleichen Enden das eines Stacks. Wir können aber auch beide Verhaltensweisen mischen, manchmal am einen und manchmal am anderen Ende einfügen oder entfernen.

Listing 2.16 Double-Ended-Queue als Array implementiert

```
1  public class DEQueue {
2      private int mask = (1 << 3) - 1;
3      private String[] es = new String[mask + 1];
4      private int head, tail;
5
6      public void addFirst(String e) {
7          es[head = (head - 1) & mask] = e;
8          if (tail == head) { doubleCapacity(); }
9      }
10     public String pollFirst() {
11         String result = es[head];
12         es[head] = null;
13         if (tail != head) { head = (head + 1) & mask; }
14         return result;
15     }
16     public String peekFirst() { return es[head]; }
17
18     public void addLast(String e) {
19         es[tail] = e;
20         tail = (tail + 1) & mask;
21         if (tail == head) { doubleCapacity(); }
22     }
23     public String pollLast() {
24         if (tail != head) { tail = (tail - 1) & mask; }
25         String result = es[tail];
26         es[tail] = null;
27         return result;
28     }
29     public String peekLast() {
30         return es[(tail - 1) & mask];
31     }
32
33     public int size() { return (tail - head) & mask; }
34
35     private void doubleCapacity() {
36         mask = (mask << 1) | 1;
37         String[] newes = new String[mask + 1];
38         int i = 0, j = 0;
39         while (i < head) { newes[j++] = es[i++]; }
40         j = head += es.length;
41         while (i < es.length) { newes[j++] = es[i++]; }
42         es = newes;
43     }
44 }
```

null

Eine der Feinheiten, die wir in abstrakten Datentypen (sowohl in der Innen- als auch Außensicht) berücksichtigen müssen, betrifft den Umgang mit `null`. Die Ergebnisse der mit `poll` und `peek` beginnenden Methoden sind `null`, wenn keine Daten vorhanden sind. Allerdings kann ein Ergebnis auch `null` sein, wenn mittels einer `add`-Variante `null` eingefügt wurde. Diese Fälle müssen unterscheidbar sein.

size wegen null

Listing 2.16 enthält wie Listing 2.14 die Methode `size`, mit der die Anzahl der vorhandenen Einträge abgefragt werden kann. Es ist nur dann kein Eintrag vorhanden, wenn `size` den Wert 0 liefert. In Listing 2.15 wurde dagegen auf `size` verzichtet, sodass wir bei Verwendung dieser Implementierung von Queue eine leere Queue nur richtig erkennen können, wenn `null` als Argument von `push` verboten ist.

Die Verwendung von `mask` stellt eine Besonderheit in der Implementierung dar, die voraussetzt, dass die Länge von `es` eine Zweierpotenz und der Wert von `mask` um 1 niedriger ist. Anfangs hat `es` die Länge 8 (binär 1000) und `mask` den Wert 7 (binär 111). Auch nach Verdoppeln der Kapazität ist die Arraylänge eine Zweierpotenz (binär 1 gefolgt von einer Anzahl n an Nullen) und der Wert von `mask` besteht binär aus genau n Einsern. Das vereinfacht Indexberechnungen: Für jede ganze Zahl x liegt x `&` `mask` im Indexbereich von `es` und entspricht x modulo der Arraylänge. Zweierpotenzen als Arraylängen stellen meist keine nennenswerte Einschränkung dar, da sich das ganz natürlich aus Verdopplungen der Kapazität ergibt. Durch Verdoppeln wird im Gegensatz zu linearem Vergrößern schon nach wenigen Vergrößerungen die maximale Länge erreicht, was den gesamten Aufwand deutlich reduziert. Falls die Arraylänge beliebig gewählt werden soll (nicht notwendigerweise Zweierpotenz), muss statt mit einer Bit-Maske mit Modulo-Rechnungen gearbeitet werden, also z. B. `head = (head + 1) % es.length`. Solche Berechnungen sind nur minimal aufwändiger. In Listing 2.16 wird vor allem aus didaktischen Gründen mit Masken gearbeitet: Das zum Verständnis nötige Wissen über Zusammenhänge zwischen Indexberechnungen und Bitmustern wird dabei helfen, bestimmte Eigenschaften einiger weiter unten behandelter Datenstrukturen zu erkennen.

Die Begriffe Stack, Queue und Double-Ended-Queue bezeichnen eher die Außensicht eines abstrakten Datentyps als eine Datenstruktur, weil keine bestimmte Darstellungsform der Daten beschrieben wird. Die Daten könnten statt durch ein Array auch durch eine lineare Liste (siehe Abschnitt 2.3.1) dargestellt sein. Aus dem Blickwinkel einer Datenstruktur sprechen wir beispielsweise von einer *mittels Array implementierten Queue* oder einem *als Queue verwendeten Array.*

Java

Standardbibliotheken von Java enthalten in `java.util` die Klasse `ArrayDeque<...>`, die `DEQueue` ähnelt, aber einen etwas größeren Funktionsumfang hat und nicht auf Zeichenketten eingeschränkt ist. Konkret schreiben wir `ArrayDeque<String>`, `ArrayDeque<char[]>`, `ArrayDeque<Integer>` und so weiter, wenn die Objekte Zeichenketten, Arrays vom Typ `char[]`, ganze Zahlen vom Typ `int` und so weiter enthalten sollen.[5] Obwohl diese Klasse `size` implementiert, dürfen Argumente von `addFirst` und `addLast` in `ArrayDeque<...>` nicht `null` sein. Die Klasse `LinkedList<...>` aus `java.util` implementiert die Methoden von `DEQueue` in Form einer linearen Liste (siehe Abschnitt 2.3.1) und ist auf dieselbe Art verwendbar. Die Klasse `Stack<...>` aus `java.util` bietet kaum mehr als den Funktionsumfang von Listing 2.15. Sie wird heute nur mehr selten eingesetzt, weil die Implementierung als ineffizient gilt. Es gibt ja genug Alternativen.

Aufgabe 2.15 Beschreiben Sie FIFO- und LIFO-Verhalten und erklären Sie, wie in Listing 2.13 solches Verhalten erkennbar ist.

Aufgabe 2.16 Beschreiben Sie in Bezug auf Listing 2.16

- wozu die Objektvariablen dienen, insbesondere `mask`,
- wodurch sichergestellt wird, dass alle Arrayzugriffe im erlaubten Indexbereich stattfinden,
- wodurch sichergestellt ist, dass alle `poll`- und `peek`-Varianten bei einer leeren Datenstruktur `null` zurückgeben,
- was die Methode `doubleCapacity` im Detail macht,
- wie oft `doubleCapacity` maximal aufgerufen wird.

Aufgabe 2.17 Schreiben Sie ein Programm zur Simulation eines *Verschubbahnhofs* mit zehn parallelen Gleisen, jeweils mit Platz für 10 Waggons, wobei jedes Gleis durch ein Objekt vom Typ `DEQueue` aus Listing 2.16 implementiert ist. Nur der erste und letzte Waggon auf jedem Gleis ist sichtbar, und nur diese Waggons können als neue erste bzw. letzte Waggons auf noch nicht volle andere Gleise verschoben werden. Organisieren Sie die Simulation als Spiel, in dem anfangs in zufälliger Reihenfolge auf den Gleisen abgestellte Waggons, die jeweils

[5] Das Konzept dahinter heißt *Generizität.* In EP2 gehen wir nicht näher als nötig auf Generizität ein, da man beim Einsatz von Generizität leicht in Situationen gerät, die Detailwissen über weit fortgeschrittene Programmierkonzepte verlangen.

mit einer Gleisnummer und fortlaufenden Nummern bezeichnet sind, entsprechend dieser Bezeichnungen sortiert werden müssen.

Aufgabe 2.18 Beschreiben Sie, anhand welcher Kriterien Sie entscheiden, ob Sie in Ihrem Programm eine Queue, einen Stack, ein Array oder eine assoziative Datenstruktur einsetzen.

2.2.3 Assoziative Datenstrukturen

Assoziative Datenstrukturen *assoziieren* (verknüpfen) Werte mit anderen Werten. Das bekannteste Beispiel dafür ist ein Array: Jeder Index, das ist ein Wert, ist mit dem entsprechenden Arrayeintrag, das ist auch ein Wert, assoziiert. Der Index ist eine ganze Zahl, mit deren Hilfe sich die Speicheradresse des entsprechenden Arrayeintrags leicht berechnen lässt. Ist der Index bekannt, kann sehr effizient auf den Arrayeintrag zugegriffen werden. Aus diesem Grund sind Arrays in vielen Programmiersprachen vordefiniert und werden häufig verwendet. Allerdings haben Arrays auch Nachteile: Als Indexe können nur fortlaufende ganze Zahlen (oder Werte eines Datentyps, die sich leicht in fortlaufende ganze Zahlen umwandeln lassen) verwendet werden. Beim Erzeugen eines Arrays müssen wir die Größe angeben und können sie nachträglich nicht mehr ändern. Über den Index sind Arrayeinträge lückenlos sortiert, und es ist nicht möglich, einen neuen Index zwischen zwei aufeinanderfolgenden Indexen einzufügen, ohne die Zuordnung zwischen Indexen und Arrayeinträgen zu ändern. Häufig benötigen wir assoziative Datenstrukturen, die diese Eigenschaften von Arrays vermeiden. Es gibt viele solche Datenstrukturen, die eines gemeinsam haben: Ohne fortlaufenden Index ist es schwieriger, aus einem Wert die Speicheradresse des assoziierten Wertes zu berechnen. Zugriffe sind aufwendiger und benötigen irgendeine Form der Suche. Durch eine gute, an die jeweilige Situation angepasste Wahl der assoziativen Datenstruktur lässt sich der Aufwand für die Suche sowie für andere Operationen wie Einfügen und Entfernen von Einträgen in einem vernünftigen Rahmen halten.

assoziative Datenstruktur → Suche ohne Index

Listing 2.17 zeigt eine recht einfache Implementierung einer assoziativen Datenstruktur. Inhalte sind in Arrays abgelegt, und nach Einträgen wird linear gesucht, also ein Eintrag nach dem anderen betrachtet, bis das Gesuchte gefunden ist. Wie in assoziativen Datenstrukturen üblich werden (statt der Indexe von Arrays) *Schlüssel* (englisch *Keys,* abgekürzt `k`) mit Werten (englisch *Values,* abgekürzt `v`) verknüpft. Ein Schlüssel zusammen mit dem assoziierten Wert ist ein *Eintrag.* Sowohl Schlüssel als auch Werte sind in Listing 2.17 vom Typ `String`, können

Schlüssel
Eintrag

allgemein aber von beliebigen, auch verschiedenen Typen sein. Für assoziative Datenstrukturen typisch sind Operationen wie `put`, `get` und `remove`. Zur Beschreibung dieser Operationen nehmen wir an, dass `x` ein Objekt vom Typ `SimpleAssoc` ist und `k` sowie `v` Strings sind:

- Ein Aufruf `x.put(k, v)` legt in `x` einen Eintrag mit Schlüssel `k` und Wert `v` an, falls in `x` noch kein Eintrag mit Schlüssel `k` existiert; der Rückgabewert ist in diesem Fall `null`. Sollte schon ein Eintrag mit Schlüssel `k` existieren, wird dieser Eintrag geändert, sodass danach `k` mit `v` assoziiert ist; der Wert, mit dem `k` zuvor assoziiert war, kommt als Ergebnis zurück.

- Ein Aufruf `x.get(k)` gibt als Ergebnis den Wert zurück, mit dem `k` in `x` assoziiert ist, falls es einen Eintrag mit Schlüssel `k` gibt. Andernfalls ist das Ergebnis `null`.

- Ein Aufruf `x.remove(k)` entfernt den Eintrag mit Schlüssel `k` aus `x` und gibt den mit `k` assoziierten Wert zurück, falls es einen solchen Eintrag gibt. Andernfalls ist das Ergebnis `null`.

Aufgrund dieses Verhaltens kann es in `x` nicht mehrere Einträge mit dem gleichen Schlüssel geben. Wenn `k` und `v` stets verschieden von `null` sind, können wir aus den Methodenergebnissen ablesen, ob es vor Ausführung schon einen entsprechenden Eintrag gegeben hat. Allerdings erlaubt die Implementierung von `SimpleAssoc` wegen des Vergleichs in Zeile 9 die Verwendung von `null` als Schlüssel und Werte, wodurch ein Ergebnis `null` nicht nur bei einem fehlenden Eintrag vorkommt. Über die Methode `containsKey` können wir dennoch feststellen, ob es einen Eintrag für einen bestimmten Schlüssel gibt. Analog dazu stellt die Methode `containsValue` fest, ob es einen Eintrag mit einem bestimmten Wert gibt. Beispielsweise können wir durch einen Aufruf `x.containsValue(null)` feststellen, ob irgendein Eintrag den Wert `null` hat. Aber wir können nicht feststellen, wieviele solche Einträge es gibt und welche Schlüssel Einträge mit diesem Wert haben.

Schlüssel eindeutig

Außer `size` ruft jede `public` Methode in Listing 2.17 `find` auf. Ist ein gesuchter Eintrag gefunden (oder bei `find(...) == top` nicht vorhanden), bleibt nicht mehr viel zu tun. Am meisten Programmtext brauchen wir in den Zeilen 16 bis 23 für das Vergrößern der Arrays, wenn nicht genug Platz frei ist. Einige Implementierungsdetails helfen, den Programmtext kurz zu halten. So enthalten `ks[top]` und `vs[top]`

Suche wichtig

Array vergrößern

Listing 2.17 Assoziative Datenstruktur als Arrays mit linearer Suche

```
1  public class SimpleAssoc {
2      private int top;
3      private String[] ks = new String[8];
4      private String[] vs = new String[8];
5
6      private int find(String s, String[] a) {
7          int i = 0;
8          while (i < top &&
9                  !(s==null ? s==a[i] : s.equals(a[i])))
10             i++;
11         return i;
12     }
13     public String put(String k, String v) {
14         int i = find(k, ks);
15         if (i == top && ++top == ks.length) {
16             String[] nks = new String[top << 1];
17             String[] nvs = new String[top << 1];
18             for (int j = 0; j < i; j++) {
19                 nks[j] = ks[j];  nvs[j] = vs[j];
20             }
21             ks = nks;  vs = nvs;
22         }
23         ks[i] = k;
24         String old = vs[i];
25         vs[i] = v;
26         return old;
27     }
28     public String remove(String k) {
29         int i = find(k, ks);
30         String old = vs[i];
31         if (i < top) {
32             ks[i] = ks[--top];  ks[top] = null;
33             vs[i] = vs[top];    vs[top] = null;
34         }
35         return old;
36     }
37     public String get(String k) {
38         return vs[find(k, ks)];
39     }
40     public boolean containsKey(String k) {
41         return find(k, ks) < top;
42     }
43     public boolean containsValue(String v) {
44         return find(v, vs) < top;
45     }
46     public int size() { return top; }
47 }
```


stets `null` damit großteils auf Sonderbehandlungen für nicht gefundene Einträge verzichtet werden kann. Daher und zur Verbesserung der Speicherverwaltung werden Arrayeinträge in den Zeilen 33 und 34 auf `null` gesetzt: Das Java-System entfernt nicht mehr referenzierte Objekte ab und zu automatisch, aber nicht z. B. in `ks` referenzierte Objekte, da Verwendungen möglich sind (etwa durch den Debugger). `auf null setzen`

Java-Standardbibliotheken enthalten mehrere unterschiedliche Implementierungen assoziativer Datenstrukturen, die im Grunde alle genau so wie `SimpleAssoc` (mit den gleichen Methodennamen) verwendet werden. Eine empfehlenswerte Implementierung durch Arrays mit linearer Suche ist jedoch nicht darunter, weil diese Suche ineffizient ist. Wir werden später effizientere Suchmethoden kennenlernen, die auf Hash-Tabellen oder Suchbäumen aufbauen. Zu den am häufigsten verwendeten vordefinierten Implementierungen zählen analog dazu `HashMap<K,V>` und `TreeMap<K,V>`, wobei `K` für den Typ der Schlüssel und `V` für den Typ der assoziierten Werte steht. **Java**

Aufgabe 2.19 Beschreiben Sie in Bezug auf Listing 2.17

- was die Objektvariablen beinhalten, insbesondere `top`,
- wieso `find` einen Index und nicht gleich den Eintrag an diesem Index zurückgibt,
- warum `find` den Parameter `a` verwendet, obwohl `ks` und `vs` direkt zugreifbar sind,
- wieso in Zeile 9 Zeichenketten mittels `==` verglichen werden,
- wie `put` und `remove` im Detail funktionieren,
- was passieren könnte, wenn in `remove` auf das Eintragen von `null` in die Arrays verzichtet würde.

Aufgabe 2.20 Schreiben Sie unter Verwendung von `SimpleAssoc` aus Listing 2.17 ein interaktives Programm, das als *elektronisches Telefonbuch* zu Namen entsprechende Telefonnummern findet. Auch das Anlegen und Löschen von Einträgen sowie Ändern von Telefonnummern soll möglich sein.

Aufgabe 2.21 Entwickeln Sie einen einfachen Datensatz `Pair`, der (ähnlich `Point` aus Listing 2.7) zwei Zeichenketten zueinander in Beziehung setzt, und ersetzen Sie die beiden Arrays in Listing 2.17 durch

ein einziges Array vom Typ `Pair[]`. Beschreiben Sie, welche Änderungen dadurch nötig werden.

Aufgabe 2.22 Beschreiben Sie, welche Werte Sie im Allgemeinen als *Schlüssel* assoziativer Datenstrukturen wählen.

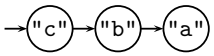
Aufgabe 2.23 Beschreiben Sie an assoziativen Datenstrukturen und Listen Unterschiede und Gemeinsamkeiten zwischen Datenstrukturen, Implementierungen und Außensichten abstrakter Datentypen, sowie die Blickwinkel (Datenstruktur, Implementierung, Außensicht), die in folgenden Begriffen hauptsächlich zum Tragen kommen: assoziative Datenstruktur, über zwei Arrays implementierte assoziative Datenstruktur, `SimpleAssoc`.

2.3 Rekursion über Daten

Eine Methode ist rekursiv, wenn im Zuge der Ausführung dieser Methode ein weiterer Aufruf dieser Methode vorkommen kann. Eine Klasse bzw. Datenstruktur ist rekursiv, wenn sie (direkt in einer Objektvariablen oder indirekt in einem referenzierten Objekt) eine Referenz auf ein Objekt dieser Klasse bzw. Datenstruktur enthalten kann. Solche Methoden, Klassen und Datenstrukturen sind *induktiv* aufgebaut.

2.3.1 Lineare Liste

Eine *lineare Liste* oder kurz *Liste* ist eine sehr einfache rekursive Datenstruktur, die sich beim Entwurf eines Programms häufig ganz natürlich ergibt. In der Darstellung als gerichteter Graph sehen wir, dass jeder als Kreis gezeichnete *Knoten* der Liste (abgesehen vom letzten) genau einen anderen Knoten (den *Nachfolger*) referenziert und (abgesehen vom ersten, auch *Kopf* genannt) von genau einem Knoten (dem *Vorgänger*) referenziert wird:



Innerhalb der Kreise stehen die *Listeneinträge,* genau einer pro Knoten. Die hier dargestellte Liste hat als Einträge die Zeichenketten `"c"`, `"b"` und `"a"` (auch Elemente oder schlicht Werte genannt), im Allgemeinen können sie von beliebigen Typen sein – jedoch jeder Eintrag der gleichen

Liste vom gleichen deklarierten Typ.[6] Jeder Knoten ist beispielsweise durch folgenden Datensatz implementiert:

Listing 2.18 Implementierung eines Listenknotens als Datensatz

```
1  public class ListNode {
2      private String value;
3      private ListNode next;
4
5      public ListNode(String v, ListNode n) {
6          value = v;
7          next = n;
8      }
9      public String value() { return value; }
10     public ListNode next() { return next; }
11     public void setNext(ListNode n) { next = n; }
12     public void setValue(String v) { value = v; }
13 }
```

Ein solcher Datensatz enthält nur den Listeneintrag `value` und die Referenz auf den Nachfolger `next` als Objektvariablen, entsprechende Getter- und Setter-Methoden sowie einen Konstruktor. Da `next` den gleichen Typ `ListNode` hat wie der Datensatz (Rekursion auf Daten), können wir ein einzelnes Objekt dieses Typs als Datensatz betrachten, aber gleichzeitig auch die Gesamtheit aller über `next` miteinander verbundener Objekte als lineare Liste. Mit diesem Ansatz liegt die gesamte inhaltliche Komplexität leider beim Verwender von `ListNode`:

Listing 2.19 Beispiel für die Verwendung von `ListNode`

```
1  private static void useListNode() {
2      ListNode list = null;
3      for (char c = 'a'; c <= 'z'; c++) {
4          list = new ListNode("" + c, list);
5      }
6      while (list != null) {
7          System.out.println(list.value());
8          list = list.next();
9      }
10 }
```

Bei Ausführung von `useListNode` werden die Zeichenketten in umgekehrter Reihenfolge ausgegeben (und aus der Liste entfernt) als in die Liste eingefügt – LIFO-Verhalten. Im Unterschied zur üblichen Verwendung linearer Datenstrukturen wie in Listing 2.13 kann die Variable `list` in Listing 2.19 nicht als `final` deklariert werden: Ihr Wert

einfache Liste: LIFO

[6] Einträge können unterschiedliche *dynamische Typen* haben – siehe Kapitel 3.

ändert sich bei jeder Änderung der Liste. Das „Vergessen“ einer solchen Änderung ist eine häufige Fehlerursache. Meist verwenden wir Klassen, die solche Variablenänderungen verbergen und damit den Umgang mit Listen auf eine höhere Abstraktionsebene bringen:

Listing 2.20 Liste als Stack verwendbar, inhaltliche Komplexität verborgen

```
1  public class ListStack {
2      private ListNode head;
3
4      public void push(String v) {
5          head = new ListNode(v, head);
6      }
7      public String pop() {
8          if (head != null) {
9              String result = head.value();
10             head = head.next();
11             return result;
12         }
13         return null;
14     }
15     public String peek() {
16         return head == null ? null : head.value();
17     }
18 }
```

Länge unbeschränkt

Im Gegensatz zu Arrays gibt es bei Listen keine Einschränkung der Listenlänge. Wir können weitere Knoten hinzufügen, bis der Speicher des Computers gänzlich aufgebraucht ist.

Einfügen am Ende: FIFO

Eine Liste ist als Queue verwendbar, indem neue Knoten am Ende der Liste eingefügt werden. In der Implementierung in Listing 2.21 unterscheidet sich `poll` nur durch den Namen von `pop`, aber `add` durchläuft zur Suche nach dem letzten Knoten vor dem Einfügen die gesamte Liste. Diese einfache Implementierung von `add` ist für lange Listen sehr ineffizient. Listing 2.22 zeigt, dass das Einfügen am Ende auch ohne Durchlaufen der Liste möglich ist: Die Variable `last` wird aus der Methode `add` auf die Ebene von Objektvariablen geschoben, wobei `last` stets auf den letzten Knoten verweist. Die zusätzliche Objektvariable wirkt sich nicht nur auf `add`, sondern auch auf `poll` aus: Aus Gründen der Programmhygiene (für bessere Wartbarkeit und Speichereffizienz) ist es sinnvoll, `last` bei leeren Listen auf `null` zu setzen, obwohl Methoden auch bei Verzicht darauf richtige Ergebnisse liefern.

Suche nach Listenende vermeidbar

Index möglich, aber aufwendig

Die Verwendung von Listen ist nicht auf Stacks und Queues beschränkt. Z.B. ermöglichen Indexe wie in Listing 2.23 Zugriffe auf beliebige Einträge. Mit den hier verwendeten *einfachverketteten Listen*

Listing 2.21 Liste als Queue verwendbar, Einfügen aufwendig

```
1  public class ListQueueSimple {
2      private ListNode head;
3
4      public void add(String v) {
5          if (head == null) {
6              head = new ListNode(v, null);
7          } else {
8              ListNode last = head;
9              while (last.next() != null) {
10                 last = last.next();
11             }
12             last.setNext(new ListNode(v, null));
13         }
14     }
15     public String poll() {
16         if (head != null) {
17             String result = head.value();
18             head = head.next();
19             return result;
20         }
21         return null;
22     }
23     public String peek() {
24         return head == null ? null : head.value();
25     }
26 }
```

ist das Entfernen des letzten Knotens ähnlich aufwendig wie `add` in Listing 2.21, sodass solche einfachverketteten Listen für Double-Ended-Queues nicht sinnvoll sind. Dafür werden *doppeltverkettete Listen* verwendet, die auch vom Nachfolger auf den Vorgänger referenzieren – siehe Abschnitt 2.3.3. Häufig kommen auch *sortierte Listen* vor, bei denen die Listeneinträge stets sortiert gehalten werden.

Aufgabe 2.24 Entwickeln Sie eine *lineare Liste,* deren Einträge vom Typ `Pair` (aus Aufgabe 2.21) sind, und ersetzen Sie die Arrays in Listing 2.17 durch diese Liste. Beschreiben Sie, welche Änderungen dadurch nötig werden.

Aufgabe 2.25 Erweitern Sie `ListQueue` aus Listing 2.22 zusammen mit Listing 2.23 um die Methode `void add(String v, int i)` zum Einfügen von `v` am Index `i` und die Methode `String remove(int i)` zum Entfernen und Zurückgeben des Listeneintrags am Index `i`. Be-

Listing 2.22 Liste als Queue verwendbar, Einfügen einfach

```
1  public class ListQueue {
2      private ListNode head, last;
3
4      public void add(String v) {
5          if (head == null) {
6              head = last = new ListNode(v, null);
7          } else {
8              last.setNext(last = new ListNode(v, null));
9          }
10     }
11     public String poll() {
12         if (head != null) {
13             String result = head.value();
14             head = head.next();
15             if (head == null) { last = null; }
16             return result;
17         }
18         return null;
19     }
20     public String peek() {
21         return head == null ? null : head.value();
22     }
23 }
```

Listing 2.23 Index-Operationen stehen in `ListQueue` – siehe Listing 2.22

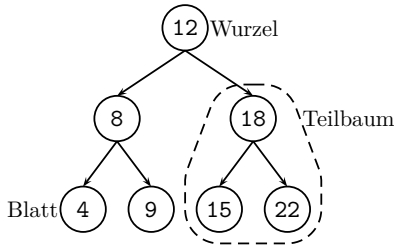
```
1  public int indexOf(String v) {
2      int i = 0;
3      ListNode n = head;
4      while (n != null && !(v == null ? v == n.value()
5                                      : v.equals(n.value()))) {
6          i++;
7          n = n.next();
8      }
9      return n == null ? -1 : i;
10 }
11 public String valueAt(int i) {
12     for (ListNode n=head; n != null; n=n.next()) {
13         if (i-- == 0) { return n.value(); }
14     }
15     return null;
16 }
```


schreiben Sie, wie Sie nach passenden Stellen in Listen suchen, neue Knoten einhängen und bestehende Knoten entfernen.

Aufgabe 2.26 Ändern Sie `ListQueue` aus Listing 2.22 zusammen mit Listing 2.23 so ab, dass die Listeneinträge stets lexikographisch aufsteigend sortiert bleiben, wobei die Methode `int add(String v)` das Argument `v` an einer passenden Stelle einfügt und den Index des neuen Eintrags zurückgibt. Beschreiben Sie dazu nötige Änderungen.

2.3.2 Binärer Baum

In einem binären Baum referenziert jeder Knoten bis zu zwei Nachfolger. Es gibt genau eine *Wurzel* ohne Vorgänger, jeder andere Knoten hat genau einen Vorgänger. Jeder Knoten ohne Nachfolger ist ein *Blatt.* Jeder Knoten im Baum ist die Wurzel eines *Teilbaums,* der alle von diesem Knoten aus erreichbaren weiteren Knoten enthält.



Oft verwenden wir den binären Baum als *Suchbaum,* bei dem die Suche nach Einträgen im Vordergrund steht. Zwecks effizienterer Suche sind die Einträge sortiert. Im Beispiel sind die Einträge ganze Zahlen: Für jeden Teilbaum ist jeder über den linken Nachfolger erreichbare Eintrag kleiner als der Eintrag an der Wurzel und der Eintrag an der Wurzel kleiner als jeder über den rechten Nachfolger erreichbare Eintrag. So wissen wir stets, ob wir bei der Suche nach einem bestimmten Eintrag den linken oder rechten Teilbaum betrachten müssen. Listing 2.24 zeigt einen Ausschnitt aus einer Implementierung eines Knotens. Im Gegensatz zu den Listenknoten in Listing 2.18 verzichten wir in Listing 2.24 auf Getter- und Setter-Methoden und implementieren wesentliche Teile der Zugriffsoperationen direkt in den Baumknoten. Das hat einen didaktischen Grund – alternative Implementierungstechnik – aber auch einen praktischen: Operationen auf Bäumen *traversieren*[7] die Bäume

Suchbaum: Einträge sortiert

Bäume meist rekursiv traversiert

[7] Unter *Traversieren* verstehen wir das Durchwandern einer (meist rekursiven) Datenstruktur, von einem Knoten zum anderen, bis das Ziel erreicht ist.

Listing 2.24 Ausschnitt aus der Implementierung eines Baumknotens

```
1  public class TreeNode {
2      private int value;
3      private TreeNode left, right;
4
5      public TreeNode(int v) { value = v; }
         ... hier folgt der Programmtext aus Listing 2.26
   }
```

Listing 2.25 Implementierung eines binären Suchbaums

```
1  public class STree {
2      private TreeNode root;
3
4      // add v if not already in the tree
5      public void add(int v) {
6          if (root == null) { root = new TreeNode(v); }
7          else { root.add(v); }
8      }
9      // is v in the tree?
10     public boolean contains(int v) {
11         return root != null && root.contains(v);
12     }
13 }
```

häufig mit rekursiven Methoden, die zwecks direkten Zugriffs auf Objektvariablen in den Baumknoten stehen, während Listenzugriffe oft nicht-rekursiv sind. Wie bei Listen rufen wir die Methoden in den Knoten auch bei Bäumen nicht direkt auf, sondern verwenden eine Klasse wie in Listing 2.25, die Details vor Anwendern verbirgt. Das Traversieren erfolgt nicht in `STree`, sondern in `TreeNode` (Listing 2.26). Sowohl `add` als auch `contains` suchen nach der Stelle im Baum, an der `v` stehen sollte. Genau an der Stelle, an der der Eintrag erwartet wird, aber nicht gefunden wurde, trägt `add` einen neuen Knoten ein.

Rekursive Methoden folgen ganz natürlich der induktiven Struktur der Daten. Viele einfach rekursive Methoden lassen sich ohne besonderen Aufwand durch Schleifen ersetzen. Bei Mehrfachrekursion ist das nur selten der Fall. In Bäumen werden häufig rekursive Methoden verwendet, da sich durch mehrere Nachfolger oft auf natürliche Weise Mehrfachrekursion (baumartige Rekursion) ergibt.

Suchbäume eignen sich wegen der relativ effizienten Suche gut zur Organisation der Schlüssel in assoziativen Datenstrukturen. Im Bei-

Listing 2.26 Methoden in `TreeNode` – Fortsetzung von Listing 2.24

```
6  public void add(int v) {
7      if (v < value) {
8          if (left != null) { left.add(v); }
9          else { left = new TreeNode(v); }
10     } else if (v > value) {
11         if (right != null) { right.add(v); }
12         else { right = new TreeNode(v); }
13     }
14 }
15 public boolean contains(int v) {
16     return v == value ||
17             (v < value ? left!=null && left.contains(v)
18                     : right!=null && right.contains(v));
19 }
```

spiel in Listing 2.27 und 2.28 werden Zeichenketten mit Zeichenketten assoziiert. Zur Sortierung werden lexikographische Vergleiche von Zeichenketten `x` und `y` mittels `x.compareTo(y)` verwendet, wobei das Ergebnis vom Typ `int` kleiner 0 ist falls `x` lexikographisch vor `y` steht, größer 0 falls `y` vor `x` steht, und 0 wenn `x` und `y` gleich sind. Wir nehmen an, dass `null` lexikographisch vor Zeichenketten steht. Die Sortierung gibt es nur auf Schlüsseln, nicht auf damit assoziierten Werten. Daher ist z. B. `containsKey` relativ effizient, aber für `containsValue` muss der gesamte Baum traversiert werden. Um die Implementierung nicht zu unübersichtlich werden zu lassen, verzichten wir auf `remove`.

compareTo

Fassen wir einige Beobachtungen zu Implementierungen und Traversierungen rekursiver Datenstrukturen zusammen:

- Rekursive Datenstrukturen sind als gerichtete Graphen darstellbar. Knoten sind meist als Objekte einer Klasse (wobei jeder Knoten als Datensatz die Daten des Knotens mit Referenzen auf weitere Knoten verbindet) und Kanten durch Referenzen auf die Knoten implementiert.

- Häufig, aber nicht immer, wird nur über einen abstrakten Datentyp (z. B. `STree`) auf die Implementierung der Knoten (z. B. `TreeNode`) mit den eigentlichen rekursiven Daten zugegriffen, damit Anwender von der Komplexität im Umgang mit rekursiven Datenstrukturen möglichst gut abgeschirmt werden. Ein weiterer Grund ist die einfache Darstellung leerer Strukturen, was aber auch über Einträge mit Wert `null` in Knoten erreichbar wäre.

Listing 2.27 Knoten eines Baums als Teil einer assoziativen Datenstruktur

```
1  public class TANode {
2      private String key, value;
3      private TANode left, right;
4
5      public TANode(String k, String v) {
6          key = k;
7          value = v;
8      }
9
10     private int compare(String k) {
11         if (k == null) { return key == null ? 0 : -1; }
12         if (key == null) { return 1; }
13         return k.compareTo(key);
14     }
15
16     public String put(String k, String v) {
17         int cmp = compare(k);
18         if (cmp < 0) {
19             if (left!=null) { return left.put(k,v); }
20             left = new TANode(k,v);
21         } else if (cmp > 0) {
22             if (right!=null) { return right.put(k,v); }
23             right = new TANode(k,v);
24         } else {
25             String result = value;
26             value = v;
27             return result;
28         }
29         return null;
30     }
31     public TANode find(String k) {
32         int cmp = compare(k);
33         if (cmp == 0) { return this; }
34         TANode node = cmp < 0 ? left : right;
35         if (node == null) { return null; }
36         return node.find(k);
37     }
38     public boolean hasValue(String v) {
39         return (v == null ? value == v
40                           : v.equals(value)) ||
41                 (left != null && left.hasValue(v)) ||
42                 (right != null && right.hasValue(v));
43     }
44     public String value() { return value; }
45 }
```


Listing 2.28 Assoziative Datenstruktur auf der Basis von binärem Suchbaum

```
1  public class TreeAssoc {
2      private TANode root;
3
4      public String put(String k, String v) {
5          if (root != null) { return root.put(k,v); }
6          root = new TANode(k,v);
7          return null;
8      }
9      public String get(String k) {
10         if (root == null) { return null; }
11         TANode node = root.find(k);
12         return node == null ? null : node.value();
13     }
14     public boolean containsKey(String k) {
15         return root != null && root.find(k) != null;
16     }
17     public boolean containsValue(String v) {
18         return root != null && root.hasValue(v);
19     }
20 }
```

- Statt Referenzen auf Knoten können Variablen `null` enthalten, wodurch `null`-Vergleiche und Programmpfade zur Behandlung von `null` einen großen Teil der Implementierung ausmachen.
- Weil `this` nicht `null` sein kann, müssen Empfänger von Nachrichten *vor* Methodenaufrufen auf `null` geprüft werden.
- Das Fehlen gültiger Einträge wird häufig über Ergebnisse wie `null` angezeigt. Sind solche Werte auch gültige Einträge, muss die Unterscheidbarkeit zwischen gültigen und ungültigen Einträgen auf andere Weise sicher gestaltet werden.
- Die natürlichste Form der Traversierung induktiver Datenstrukturen erfolgt über rekursive Methoden. Nur Einfachrekursion ist ohne großen Aufwand durch Schleifen ersetzbar.
- Folgende Fehler sind typisch bei der Implementierung rekursiver Datenstrukturen:
  - Überprüfung auf `null` vergessen,
  - Laufvariable nicht auf nächsten Knoten weitergeschaltet,

- beim Einfügen oder Entfernen eines Knotens nicht alle nötigen Referenzen angepasst oder Datenstruktur zerstört,
- nicht alle zu betrachtenden Knoten traversiert oder unnötig Knoten besucht,
- den Überblick verloren durch schlechte Programmstruktur, zu viele Referenzen oder Optimierungen von Details,
- Randfälle nicht behandelt (z.B. leere Liste als Sonderfall).

Aufgabe 2.27 Beschreiben Sie, wozu jede Anweisung in `STree` und `TreeNode` (Listing 2.24 bis Listing 2.26) dient.

Aufgabe 2.28 Ändern Sie `TreeAssoc` und `TANode` (Listing 2.27 und 2.28) so ab, dass nicht nur nach Schlüsseln, sondern auch nach damit assoziierten Werten über einen binären Suchbaum gesucht wird. Dazu sind in `TANode` neben `left` und `right` zwei weitere Objektvariablen nötig, über die Inhalte von `value` über einen Suchbaum gefunden werden können. Beschreiben Sie die nötigen Änderungen.

Aufgabe 2.29 Beschreiben Sie typische Fehler bei der Implementierung und Traversierung rekursiver Datenstrukturen sowie Vorgehensweisen um solchen Fehlern vorzubeugen bzw. um sie zu erkennen.

2.3.3 Fundierung, Zyklen, doppelte Verkettung


Fundierung

Die große Bedeutung von `null` ist nicht zu übersehen. So wie in rekursiven Methoden Abbruchbedingungen für die Termination nötig sind, so müssen rekursive Datenstrukturen ihre Enden anzeigen, meist mit `null`. Rekursive Datenstrukturen werden immer auf gleiche Weise aufgebaut: Am Beginn steht eine leere Datenstruktur (*Fundierung*), zu der nach und nach Knoten hinzugefügt werden. Es entstehen Strukturen, deren Größe nur durch den verfügbaren Speicher beschränkt ist. Die Strukturen können (auch bei theoretisch unbegrenztem Speicher) niemals unendlich groß werden, sodass sich in Methoden zum Traversieren natürliche Abbruchbedingungen ergeben. Sind Strukturen zyklisch, können beim Traversieren endlos immer wieder die gleichen Knoten besucht werden. Dann wird das Ende nicht durch `null` angezeigt und wir müssen auf andere Weise erkennen, ob schon alle Knoten bearbeitet wurden.

Das Beispiel einer Ringliste zeigt, dass die Fundierung einer rekursiven Datenstruktur auch ohne `null` möglich ist:

Listing 2.29 Queue als Ringliste mit speziellem Knoten `nil` statt `null`

```
1  public class RingQueue {
2      private ListNode nil;    // siehe Listing 2.18, Seite 61
3
4      public RingQueue() {
5          nil = new ListNode(null, null);
6          nil.setNext(nil);
7      }
8
9      public void add(String v) {
10         nil.setValue(v);
11         nil.setNext(nil=new ListNode(null,nil.next()));
12     }
13     public String poll() {
14         ListNode n = nil.next();
15         nil.setNext(n.next());
16         return n.value();
17     }
18     public String peek() { return nil.next().value(); }
19     public boolean element(String v) {
20         ListNode n = nil.next();
21         while (n != nil && !(v == null ? v == n.value()
22                                        : v.equals(n.value()))) {
23             n = n.next();
24         }
25         return n != nil;
26     }
27     public boolean isEmpty(){ return nil.next()==nil; }
28 }
```

Die Objektvariable `nil` referenziert einen speziellen, eigentlich nicht zur Liste gehörenden Knoten, der das Listenende darstellt. Dieser Knoten referenziert den ersten echten Listenknoten. Alle Knoten sind zyklisch in einer niemals leeren Struktur miteinander verbunden. Auf diese Weise entfallen viele `null`-Abfragen und Fallunterscheidungen. In Randfällen greifen wir auf `nil` zu, und `nil.value()` weist mit dem Ergebnis `null` auf einen ungültigen Eintrag hin. Wir erkennen das Listenende durch einen Vergleich mit `nil`. Trotz zyklischer Struktur gibt es keine Endlosschleifen in korrekt implementierten Methoden.

Effizienz durch Einfachheit

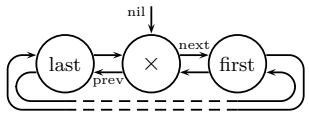
Listen – gleichgültig ob zyklisch oder nicht – können leicht in eine Richtung traversiert werden. Aber in umgekehrter Richtung ist dies sehr aufwendig. Abhilfe schaffen *doppelt verkettete Listen*, die in jedem Knoten neben einer Referenz auf den Nachfolger auch eine auf den Vorgänger enthalten. Die Listings 2.30 und 2.31 implementieren eine Double-Ended-Queue als doppelt verkettete Ringliste. Methoden zum

Listing 2.30 Listenknoten für doppelte Verkettung

```
1  public class DLNode {
2      private String value;
3      private DLNode prev, next;
4
5      public DLNode(String v) {
6          value = v;
7          prev = next = this;
8      }
9      private DLNode(String v, DLNode p, DLNode n) {
10         value = v;  prev = p;  next = n;
11     }
12
13     public String value() { return value; }
14     public DLNode next() { return next; }
15     public DLNode previous() { return prev; }
16     public void setValue(String v) { value = v; }
17     public DLNode add(String v) {
18         return next = next.prev =
19                         new DLNode(v, this, next);
20     }
21     public DLNode remove() {
22         next.prev = prev;
23         prev.next = next;
24         return this;
25     }
26 }
```

Einfügen und Löschen stehen neben Getter-Methoden und einer Setter-Methode direkt in den Knoten, sodass sich alle Operationen einfach implementieren lassen. Die Semantik ist dennoch komplex, weil in jedem Knoten statt nur einer Referenz auf einen weiteren Knoten zwei solche Referenzen gehandhabt werden müssen. Folgende Zeichnungen veranschaulichen (a) die generelle Struktur, (b) das Einfügen eines Knotens und (c) das Entfernen des letzten Knotens:

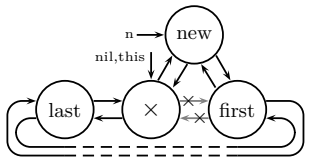
(a)



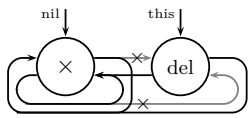
Listing 2.31 Doppelt verkettete Ringliste mit speziellem Knoten `nil`

```
1  public class RingDEQueue {
2      private DLNode nil;
3
4      public RingDEQueue() { nil = new DLNode(null); }
5
6      public void addFirst(String v) { nil.add(v); }
7      public void addLast(String v) {
8          nil.setValue(v);
9          nil = nil.add(null);
10     }
11     public String pollFirst() {
12         return nil.next().remove().value();
13     }
14     public String pollLast() {
15         return nil.previous().remove().value();
16     }
17     public String peekFirst() {
18         return nil.next().value();
19     }
20     public String peekLast() {
21         return nil.previous().value();
22     }
23     public boolean isEmpty(){ return nil.next()==nil; }
24 }
```

(b)



(c)



Aufgabe 2.30 Ringlisten sind sinnvoll, wenn sie Fallunterscheidungen einsparen. Vergleichen Sie `RingQueue` aus Listing 2.29 mit `ListQueue` aus Listing 2.22 und beschreiben Sie, durch welche Implementierungsdetails Fallunterscheidungen in `RingQueue` eingespart wurden.

Aufgabe 2.31 Beschreiben Sie, wie sich das Einfügen und Entfernen von Knoten in/aus Listen ändert, wenn statt einer einfach verketteten Liste eine doppelt verkettete Liste verwendet wird.

Aufgabe 2.32 Ersetzen Sie in Ihrer Lösung von Aufgabe 2.25 die einfach verkettete Liste durch eine doppelt verkettete Liste, aber keine Ringliste. Beschreiben Sie die dazu nötigen Änderungen.

3 Abstraktionshierarchien

Die Implementierung (die *eine* Innensicht) eines abstrakten Datentyps kann von außen gleichzeitig auf vielerlei Weise – mit unterschiedlichen Schwerpunktsetzungen und in unterschiedlichem Detailliertheitsgrad – betrachtet werden (*viele* Außensichten). Wenn eine Betrachtungsweise eine andere einschließt, sprechen wir von Abstraktionen auf unterschiedlichen *Abstraktionsebenen.* So kann eine Double-Ended-Queue (niedrigere Ebene) als Queue (höhere, abstraktere Ebene) betrachtet werden. Sind zwei Betrachtungsweisen verschieden voneinander, ohne dass eine die andere einschließt (z.B. Queue und Stack), abstrahieren sie über unterschiedliche Aspekte. Auch solche Betrachtungsweisen können miteinander *kompatibel* sein, sich also gegenseitig ergänzen, so wie sich eine Queue und ein Stack in einer Double-Ended-Queue ergänzen. Umgekehrt kann eine Betrachtungsweise auf viele unterschiedliche Implementierungen passen, so wie eine Double-Ended-Queue etwa durch ein Array oder eine doppelt verkettete Liste implementiert sein kann. Wenn wir Betrachtungsweisen und Implementierungen in Beziehung zueinander setzen, entstehen Strukturen, die in der Praxis auf vielfältige Weise verwendbar sind. Wir nennen sie *Abstraktionshierarchien.*

In diesem Kapitel betrachten wir zunächst die wichtigsten Sprachkonzepte von Java für den Umgang mit Abstraktionshierarchien. Wir werden sehen, welche Möglichkeiten sich ergeben und worauf wir bei der Verwendung achten müssen. Am meisten profitieren wir von Abstraktionshierarchien mit einer möglichst einheitlichen, vollständigen und logisch aufgebauten Struktur. Als Beispiel bauen wir schrittweise eine Abstraktionshierarchie über häufig verwendete abstrakte Datentypen auf. Das gibt uns die Gelegenheit, den Umgang mit rekursiven Datenstrukturen zu üben und Details zur Implementierung einiger typischer Datenstrukturen und Algorithmen mit ihren wesentlichen Eigenschaften näher zu untersuchen. Zur effizienten Verwendung von Abstraktionshierarchien müssen wir, trotz Abstraktion, Eigenschaften bestimmter Implementierungsansätze kennen. Wir werden anhand von Beispielen Wissen über Implementierungen mit abstrakten Betrachtungsweisen in Einklang bringen.

3.1 Klassifizieren

Java

Klassen in Java klassifizieren Objekte und bilden eine Grundlage für den Aufbau von Abstraktionshierarchien. Einige von uns denken dabei vielleicht an *Vererbung,* ein Konzept zur Übernahme von Programmtexten aus anderen Klassen. Tatsächlich ignorieren wir Vererbung weitgehend, müssen sie wegen ihrer tiefen Verwurzelung in Java aber erwähnen. Stattdessen führen wir das wichtigere Konzept der *Untertypbeziehungen* und damit im Zusammenhang stehende Operatoren (`instanceof` und Casts auf Referenztypen) ein und beschreiben Methoden, die in jeder Klasse implementiert sind. Auf eine Beschreibung der Sprachkonzepte folgen Anwendungsbeispiele, viele davon werden am Ende für die Implementierung einer Hash-Tabelle verwendet.

3.1.1 Objektschnittstellen in Java

Java

Die Außensicht einer Klasse bezeichnen wir auch als *Schnittstelle* der Klasse. Sie setzt sich aus den *Signaturen* (das sind die Köpfe bzw. Deklarationen) aller `public` Methoden, Variablen und Konstruktoren mit den dazugehörenden Kommentaren zusammen. Informell können wir Listing 2.3 als Schnittstelle von Listing 2.5 (`BoxedText`) ansehen.

Signatur

Interface

Java stellt *Interfaces* bereit, die ähnlich den Klassen definiert werden und wie Klassen in jeweils eigenen Dateien stehen:

Listing 3.1 Ein Java-Interface als Abstraktion von Listing 2.5 auf Seite 36

```
1 public interface SetBoxed {  // Kommentare wie in Listing 2.3
2     void newDimensions(int width, int height);
3     void setLine(int index, String txt);
4 }
```

Ein Interface enthält Signaturen von `public` Objektmethoden[1] (auch ohne Modifier `public`). Das Interface in Listing 3.1 enthält einen Ausschnitt aus Listing 2.3 (Seite 34), wobei Kommentare nicht nocheinmal hingeschrieben wurden. Wie Klassen sind auch Interfaces als Referenztypen verwendbar. Im Gegensatz zu Klassen können von Interfaces keine Objekte erzeugt werden; `new SetBoxed()` ist verboten. Um die Verbindung zwischen Klasse und Interface herzustellen, enthält die Definition der Klasse eine `implements`-Klausel, und die Klasse implementiert jede im Interface spezifizierte Methode:

implements

[1] Es dürfen auch `public final static` Variablen (Konstanten) enthalten sein, aber keine Konstruktoren, Objektvariablen und nichts, das nicht `public` ist.

Listing 3.2 `BoxedText` implementiert alle Methoden des Interfaces in Listing 3.1

```
public class BoxedText implements SetBoxed {
    ...  hier steht der gesamte Klasseninhalt aus Listing 2.5
}
```

Durch `implements` wird `BoxedText` zu einem *Untertyp* von `SetBoxed` und `SetBoxed` zu einem *Obertyp* von `BoxedText`. Objekte, die der Schnittstelle von `BoxedText` entsprechen, entsprechen auch der Betrachtungsweise von `SetBoxed` – Boxinhalte können gesetzt werden. Objekte vom Typ `BoxedText` sind auch Objekte vom Typ `SetBoxed`, die Objektmenge von `SetBoxed` schließt jene von `BoxedText` ein. An eine Variable vom deklarierten Typ `SetBoxed` kann auch ein Objekt vom Typ `BoxedText` zugewiesen werden. Umgekehrt geht das nicht: An eine Variable vom deklarierten Typ `BoxedText` darf kein Objekt vom deklarierten Typ `SetBoxed` zugewiesen werden:

Unter- und Obertyp

```
SetBoxed x = new BoxedText(); // kein Fehler
BoxedText y = x; // Compiler meldet Fehler
```

Wir verwenden für `x` den *deklarierten Typ* `SetBoxed`, obwohl diese Variable ein Objekt vom Typ `BoxedText` referenziert. Das ist wegen der Untertypbeziehung erlaubt. Wir sagen, `x` hat den *dynamischen Typ* `BoxedText`. Die Zuweisung von `x` an `y` ist dennoch nicht erlaubt, weil der Compiler für die Überprüfung der Typkompatibilität nur deklarierte Typen verwendet und der deklarierte Typ von `x` kein Untertyp vom deklarierten Typ von `y` ist. Ähnliches gilt auch für Methodenaufrufe: Ein Aufruf von `x.setLine(0, ".")` ist möglich, weil `SetBoxed` als deklarierter Typ von `x` diese Methode enthält. Aber `x.print()` ist nicht aufrufbar, weil `SetBoxed` diese Methode nicht beschreibt, obwohl Sie in `BoxedText` implementiert ist.

deklarierter Typ

dynamischer Typ

Eine Klasse kann mehrere Interfaces implementieren:

Listing 3.3 Ein Java-Interface mit einer vielseitig verwendbaren Methode

```
1 public interface Print {
2     void print();  // Print 'this' to standard output.
3 }
```

Listing 3.4 `BoxedText` implementiert mehrere Interfaces

```
public class BoxedText implements SetBoxed, Print {
    ...  hier steht der gesamte Klasseninhalt aus Listing 2.5
}
```

Weil die Beschreibung von `print` in Listing 3.3 allgemeiner gehalten ist als jene in Listing 2.3, entsprechen alle Objekte von `BoxedText` auch der Betrachtungsweise von `Print`, und `BoxedText` ist Untertyp von sowohl `SetBoxed` als auch `Print`. Die Interfaces `SetBoxed` und `Print` stehen in keiner Untertypbeziehung zueinander; sie abstrahieren über ganz unterschiedliche Aspekte. Dennoch sind sie kompatibel zueinander; sie haben einen gemeinsamen Untertyp.

Interfaces können andere Interfaces erweitern:

Listing 3.5 Interface erweitert zwei andere Interfaces

```
1 public interface AbstrBoxed extends SetBoxed, Print {
2     String toString();
3 }
```

extends

Alle Inhalte der rechts von `extends` stehenden Interfaces werden automatisch in das gerade definierte Interface übernommen, sodass neben `toString` auch `newDimensions`, `setLine` und `print` Methoden von `AbstrBoxed` sind.[2] Außerdem wird `AbstrBoxed` zu einem Untertyp von `SetBoxed` und `Print`. Eine neuerliche Änderung von `BoxedText` lässt eine Reihe von Untertypbeziehungen entstehen:

Listing 3.6 `BoxedText` wird zu Untertyp von zahlreichen Typen

```
public class BoxedText implements AbstrBoxed { ... }
```

`BoxedText` wird zu einem Untertyp von `AbstrBoxed`, `SetBoxed` und `Print`. Werte vom deklarierten Typ `BoxedText` können an alle Variablen zugewiesen werden, die mit einem dieser Typen deklariert sind. Obwohl `AbstrBoxed` alle `public` Methoden von `BoxedText` enthält, ist `AbstrBoxed` nicht äquivalent zur Schnittstelle von `BoxedText`; beispielsweise fehlt eine Beschreibung des Konstruktors.

Mit Interfaces lassen sich alle nötigen Abstraktionshierarchien aufbauen, wobei die konkreteste Abstraktionsebene aus Klassen besteht. Dennoch ist ein weiterer Mechanismus tief in Java verwurzelt:

Listing 3.7 Ableitung einer neuen Klasse von einer bestehenden Klasse

```
1 public class BoxedTextReset extends BoxedText {
2     public void reset() { newDimensions(0, 0); }
3 }
```

[2] Wenn wir möchten, können wir die Signaturen einer beliebigen Auswahl dieser Methoden auch in `AbstrBoxed` hinschreiben, ohne die Semantik zu verändern. Dabei müssen wir auf gleiche Parametertypen achten, da der Compiler die Signaturen andernfalls als verschieden und damit als überladen betrachten würde.

Die Implementierung von `BoxedTextReset` gleicht der von `BoxedText` abgesehen von der zusätzlichen Methode `reset`. `BoxedTextReset` ist eine *Unterklasse* von `BoxedText` und `BoxedText` die *Oberklasse* von `BoxedTextReset`. Gleichzeitig ist `BoxedTextReset` auch ein Untertyp von `BoxedText`.

Unter- und Oberklasse

Im Unterschied zu Untertypbeziehungen beziehen sich Unterklassenbeziehungen auf Implementierungen, nicht nur Objektschnittstellen. Steht eine Methode der Oberklasse nicht im Programmtext der Unterklasse, wird diese Methode automatisch aus der Oberklasse in die Unterklasse übernommen (*geerbt*).

erben

Enthält die Unterklasse eine Methode mit derselben Signatur wie die Oberklasse, *überschreibt* die Methode der Unterklasse jene der Oberklasse und die Methode aus der Oberklasse wird nicht geerbt. Während eine Klasse beliebig viele Interfaces implementieren kann (`implements`-Klausel), darf sie nur Unterklasse von genau einer anderen Klasse sein (`extends`-Klausel in einer Klasse).

überschreiben

Da wir auf das Erben von Methoden weitgehend verzichten wollen, müssen wir Unterklassenbeziehungen nur aus einem Grund kennen: Jede Klasse (außer `Object` in `java.lang`) hat genau eine Oberklasse. Ist keine `extends`-Klausel angegeben, wird als Oberklasse automatisch `Object` aus der Klassenbibliothek `java.lang` angenommen. Das bedeutet, dass jede Java-Klasse direkt oder indirekt von `Object` abgeleitet ist und alle Methoden enthält, die in `Object` implementiert und nicht überschrieben sind. `Object` enthält einige in der Praxis wichtige Methoden, etwa `equals`. Um sinnvoll verwendbar zu sein, sollen manche dieser Methoden in Unterklassen überschrieben werden – siehe Abschnitt 3.1.2 und Abschnitt 3.1.3.

genau eine Oberklasse

Eine Objektmethode in `Object` ist `getClass()`. Sie ist in jedem Objekt aufrufbar und liefert als Ergebnis eine interne Repräsentation des dynamischen Typs, also der Klasse `X`, wenn das Objekt durch `new X(...)` erzeugt wurde. Diese interne Repräsentation vom Typ `Class` gibt zur Laufzeit durch Methodenaufrufe viel Information über die Klasse preis. Wir benötigen `getClass()` vor allem in Vergleichen: Für zwei Referenzen `x` und `y` ist `x.getClass() == y.getClass()` wahr, wenn `x` und `y` denselben dynamischen Typ haben.

getClass

Class

Jeder Typ (nicht nur ein Referenztyp) hat eine interne Repräsentation vom Typ `Class`, die wir durch Anhängen von `.class` an den Typnamen erhalten, etwa `String.class`, `Print.class`, `int.class` und `int[].class`. So ist `x.getClass() == BoxedText.class` wahr, wenn `x` den dynamischen Typ `BoxedText` hat.

class

instanceof

Auch über `instanceof` können wir zur Laufzeit Typinformation abfragen: `x instanceof T` ist wahr, wenn `x != null` gilt und der dynamische Typ von `x` ein Untertyp vom Typ `T` ist.

Cast

Typumwandlungen (Casts) ändern den deklarierten Typ:

```
SetBoxed x = new BoxedText();
BoxedText y = (BoxedText)x;
```

Das funktioniert fehlerfrei, weil der deklarierte Typ von `x` vor der Zuweisung in `BoxedText` umgewandelt wird. Allerdings würde die Typumwandlung einen Laufzeitfehler generieren, wenn der dynamische Typ von `x` kein Untertyp von `BoxedText` wäre.[3]

Aufgabe 3.1 Beschreiben Sie Gemeinsamkeiten und Unterschiede zwischen *Klassen* und *Interfaces* in Java. Beschreiben Sie auch mögliche Beziehungen zwischen zwei Klassen, zwei Interfaces sowie einer Klasse und einem Interface.

Aufgabe 3.2 Beschreiben Sie, was wir unter *Unter-* und *Obertypen* verstehen und unter welchen Bedingungen solche Beziehungen bestehen.

Aufgabe 3.3 Beschreiben Sie, was wir unter einem *deklarierten Typ* und *dynamischen Typ* einer Variablen verstehen und in welchen Fällen sich der deklarierte vom dynamischen Typ unterscheidet.

Aufgabe 3.4 Beschreiben Sie, was `getClass`, `instanceof`, ein Cast auf Referenztypen und `X.class` (für geeignete `X`) bedeutet.

3.1.2 Ersetzbarkeit, dynamisches Binden, toString

Eigenschaften von Untertypbeziehungen

Wir fassen einige allgemeine Eigenschaften von Untertypbeziehungen zusammen, wobei R, S und T für beliebige Referenztypen stehen:

- Untertypbeziehungen bilden eine *Halbordnung* auf Typen:
  - T ist Untertyp von T (reflexiv).
  - Ist R Untertyp von S und S Untertyp von T, dann ist auch R Untertyp von T (transitiv).
  - Ist S Untertyp von T und T Untertyp von S, dann sind S und T gleich (antisymmetrisch).
- Interfaces sind nicht Untertypen von Klassen.

[3] Es würde aber kein Fehler gemeldet, wenn `x` gleich `null` wäre.

- Ist T eine Klasse, dann ist T Untertyp von `Object`. Methoden aus `Object` sind in jedem Objekt eines Referenztyps aufrufbar.
- Jeder dynamische Typ ist eine Klasse (*die* Klasse des Objekts).
- `null` ist kein Objekt von T. Dennoch kann jede Variable eines Referenztyps statt einer Referenz `null` enthalten.
- Ist S ein Untertyp von T wobei T eine public Methode deklariert, dann deklariert auch S eine public Methode mit der (bis auf kleine Details) gleichen Signatur. Eine solche Methode ist in einem Objekt von S genau so wie in einem von T aufrufbar.
- Ist S ein Untertyp von T, kann ein Objekt vom Typ S überall dort verwendet werden, wo ein Objekt vom Typ T erwartet wird (insbesondere bei einer Zuweisung oder Parameterübergabe).

Ersetzbarkeit

Die letzte dieser Eigenschaften beschreibt *Ersetzbarkeit:* An eine Variable (oder einen formalen Parameter) des deklarierten Typs T ist ein Wert des deklarierten Typs S zuweisbar, wenn S ein Untertyp von T ist. Ersetzbarkeit wird in der objektorientierten Programmierung intensiv genutzt. Das ist nur aufgrund der vorletzten Eigenschaft in obiger Auflistung möglich: Methodenaufrufe, die in Objekten vom deklarierten Typ T erlaubt sind, müssen auch in Objekten vom Typ S erlaubt sein. Nur die Signaturen der Methoden müssen gleich sein, nicht die Implementierungen. Sind R und S zwei voneinander verschiedene Klassen, die dasselbe Interface T implementieren, dann müssen sowohl R als auch S alle Methoden von T implementieren, können das aber auf verschiedene Weise tun. Angenommen, `x` ist eine Variable vom deklarierten Typ T. Dann kann der dynamische Typ von `x` (bzw. des durch `x` referenzierten Objekts) R oder S sein, nach mehrfachen Zuweisungen an `x` manchmal R und manchmal S. Wenn wir eine Methode in `x` aufrufen, wird die entsprechende Implementierung der Methode ausgeführt, manchmal jene von R, manchmal jene von S, je nach dem aktuellen dynamischen Typ von `x`. Wir sprechen von *dynamischem Binden,* wenn die auszuführende Methodenimplementierung vom dynamischen Typ des Empfängers einer Nachricht abhängt (nicht nur vom deklarierten Typ), andernfalls von *statischem Binden.*

dynamisches und statisches Binden

Hier ist als Beispiel ein Interface einer assoziativen Datenstruktur, das von zwei uns schon bekannten Klassen implementiert wird, eine Implementierung beruhend auf Arrays und eine auf Suchbäumen:

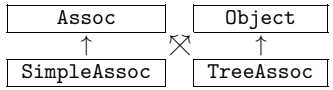
Listing 3.8 Interface für assoziative Datenstrukturen auf Zeichenketten

```
1 public interface Assoc {
2     String put(String k, String v);
3     String get(String k);
4     boolean containsKey(String k);
5     boolean containsValue(String v);
6 }
```

Listing 3.9 Zwei Klassen implementieren dasselbe Interface

```
public class SimpleAssoc implements Assoc {wie Listing 2.17}
public class TreeAssoc implements Assoc {wie Listing 2.28}
```

Dabei ergeben sich folgende Untertypbeziehungen:



Wir wollen die Laufzeiteffizienz der beiden Implementierungen durch folgende Methoden vergleichen:

Listing 3.10 Dynamisches versus statisches Binden

```
1  private static void testAssoc(Assoc assoc) {
2      final String name = assoc.getClass().getName();
3      System.out.println("Einfügen in " + name);
4      for (int i = 1; i <= 20000; i += 2) {
5          assoc.put("Key" + i, "Value" + i);
6      }
7      System.out.println("Auslesen aus " + name);
8      for (int i = 1; i <= 20000; i += 3) {
9          assoc.containsKey("Key" + i);
10         assoc.get("Key" + i);
11         assoc.containsValue("Value" + i);
12     }
13     System.out.println(name + " fertig getestet");
14 }
15 private static void testBoth() {
16     testAssoc(new SimpleAssoc());
17     testAssoc(new TreeAssoc());
18 }
```

Der formale Parameter `assoc` von `testAssoc` ist vom Typ `Assoc`. In `testBoth` wird diese Methode einmal mit einem Argument vom Typ `SimpleAssoc` und einmal mit einem vom Typ `TreeAssoc` aufgerufen. Der dynamische Typ von `assoc` ist daher von Aufruf zu Aufruf

Listing 3.11 Implementierung von `toString` als Erweiterung von Listing 2.17

```
1 public String toString() {
2     if (top <= 0) { return "{}"; }
3     String s = "(" + ks[0] + ", " + vs[0] + ")";
4     for (int i = 1; i < top; i++) {
5         s += ", (" + ks[i] + ", " + vs[i] + ")";
6     }
7     return "{" + s + "}";
8 }
```

verschieden, und Methodenaufrufe in `assoc` (z. B. `assoc.put(...)`) werden dynamisch gebunden. Das Ergebnis von `assoc.getClass()` (aus `Object`, daher aufrufbar) ist die interne Repräsentation des dynamischen Typs, in der die Methode `getName()` den Klassennamen des dynamischen Typs zurückgibt. Bei Ausführung von `testBoth` erkennen wir nicht nur anhand der Klassennamen im Output, sondern auch anhand der unterschiedlichen Laufzeiten, dass einmal die Methoden von `SimpleAssoc` und einmal die von `TreeAssoc` verwendet werden. Aufrufe von Klassenmethoden wie `testAssoc` oder von `private` Objektmethoden erfolgen dagegen durch statisches Binden.

Während `getClass` nicht überschrieben werden kann, sollte die Methode `String toString()` aus `Object` in vielen Klassen überschrieben, also implementiert sein. Diese Methode erzeugt eine textuelle Darstellung eines Objekts. Aufrufe von `toString` sind tief in Java integriert. So ist für jedes Objekt `x` der Ausdruck `System.out.print(x)` semantisch gleichbedeutend mit `System.out.print(x.toString())` und `"x=" + x` mit `"x=" + x.toString()`. Immer wenn eine Textdarstellung eines Objekts nötig ist, wird `toString()` aufgerufen. Das funktioniert aufgrund von dynamischem Binden. Während im `Assoc`-Beispiel durch dynamisches Binden nur zwischen zwei möglichen Klassen gewählt wurde, wird bei Aufrufen von `toString()` zwischen einer sehr großen Zahl möglicher Implementierungen gewählt.

Ist `toString()` nicht überschrieben, besteht die Textdarstellung nur aus dem Klassennamen mit einer angehängten Hexadezimalzahl. In der Regel wollen wir Objektinhalte darstellen. Listing 3.11 zeigt eine entsprechende Implementierung von `toString()` in `SimpleAssoc`, Listing 3.12 eine in Baumknoten und Listing 3.13 eine in `TreeAssoc`.

In den Zeilen 3 und 4 von Listing 3.12 wird `toString()` durch die Umwandlung von `left` und `right` in Zeichenketten rekursiv aufgerufen, obwohl im Programmtext kein Aufruf zu sehen ist. In Zeile 2

Listing 3.12 Implementierung von `toString` als Erweiterung von Listing 2.27

```
1 public String toString() {
2     String s = "(" + key + ", " + value + ")";
3     if (left != null) { s = left + ", " + s; }
4     if (right != null) { s += ", " + right }
5     return s;
6 }
```

Listing 3.13 Implementierung von `toString` als Erweiterung von Listing 2.28

```
1 public String toString() {
2     return root == null ? "{}" : "{" + root + "}";
3 }
```

von Listing 3.13 wird `toString()` aus Listing 3.12 aufgerufen falls `root` ungleich `null` ist. Die dynamischen Typen von `root`, `left`, `right`, etc. bestimmen, welche Implementierungen von `toString()` ausgeführt werden. Die Behandlung von `null` erfolgt nicht über dynamisches Binden, weil in `null` keine Methode ausführbar ist, sodass explizite `null`-Abfragen nötig sind. Trotz stark unterschiedlicher Implementierungen gibt es eine gemeinsame Außensicht auf `toString()`.

Aufgabe 3.5 Beschreiben Sie, was wir unter *Ersetzbarkeit* und *dynamischem Binden* verstehen und wie Ersetzbarkeit und dynamisches Binden mit Untertypbeziehungen zusammenhängen.

Aufgabe 3.6 Beschreiben Sie, warum in Listing 3.10 der Parameter `assoc` vom Typ `Assoc` und nicht entweder vom Typ `SimpleAssoc` oder `TreeAssoc` ist. Beschreiben Sie detailliert, welche Methoden aus welcher Klasse in Listing 3.10 ausgeführt werden, und was das mit Ersetzbarkeit und dynamischem Binden zu tun hat.

Aufgabe 3.7 Beschreiben Sie, warum `toString` in `Object` definiert ist und warum diese Methode in vielen Klassen zu überschreiben ist.

Aufgabe 3.8 Beschreiben Sie die Implementierungen von `toString` aus Listing 3.11 bis Listing 3.13, insbesondere jene Stellen, in denen die Besonderheiten von `toString` zum Tragen kommen.

3.1.3 Gleichheit und Hash-Werte

Die Methode `boolean equals(Object o)` sollte in den meisten Klassen ebenfalls überschrieben werden. Als Argument kann jedes beliebige

Objekt oder `null` verwendet werden. Wie von `String` bekannt dient diese Methode dazu, zwei Objekte inhaltlich miteinander zu vergleichen – im Gegensatz zu Identitätsvergleichen zweier Objekte mittels `==`. In `Object` ist diese Methode jedoch so vordefiniert:

equals

```
public boolean equals(Object o) { return this == o; }
```

Daher müssen wir `equals` überschreiben, wenn sich inhaltliche Vergleiche von Identitätsvergleichen unterscheiden sollen. Was bei inhaltlichen Vergleichen genau gleich sein soll, ist nicht vorgeschrieben. Meist werden zwei Objekte als inhaltlich gleich betrachtet, wenn alle Objektvariablen gleiche Inhalte haben. Objektvariablen, deren Inhalte nur der organisatorischen Vereinfachung dienen, bleiben jedoch unberücksichtigt. Manchmal (z. B. bei einem binären Baum) kommt es nur auf die enthaltenen Einträge an, nicht auf die Sortierung. Hier ist ein typisches Beispiel als Erweiterung von `BoxedText` aus Listing 2.5:

Listing 3.14 Implementierung von `equals` als Erweiterung von Listing 2.5

```
1  public boolean equals(Object o) {
2      if (this == o)
3          { return true; }
4      if (o == null || o.getClass() != BoxedText.class)
5          { return false; }
6      BoxedText b = (BoxedText)o;
7      if (textHeight != b.textHeight
8              || textWidth != b.textWidth)
9          { return false; }
10     for (int i = 0; i < textWidth; i++) {
11         for (int j = 0; j < textHeight; j++) {
12             if (text[i][j] != b.text[i][j])
13                 { return false; }
14         }
15     }
16     return true;
17 }
```

Da der Parameter vom Typ `Object` ist und `null` sein kann, ist in jeder sinnvollen Implementierung ein `null`-Vergleich, ein Vergleich der dynamischen Typen und eine Typumwandlung (Cast) nötig. Der Vergleich mit `this` in Zeile 2 dient der Effizienzsteigerung bei Identität. Die ersten 6 Zeilen sind für das Überschreiben von `equals` meist nach dem Vorbild in diesem Beispiel aufgebaut. Der Rest richtet sich nach den zu vergleichenden Werten.

Wie in der Klasse `Object` beschrieben, muss jede Implementierung von `equals` folgende Bedingungen erfüllen:

Eigenschaften von equals

- Für `x != null` liefert `x.equals(null)` stets `false`.
- `equals` ist eine Äquivalenzrelation:
  - Für `x != null` liefert `x.equals(x)` stets `true` (reflexiv).
  - Für `x != null` und `y != null` liefert `x.equals(y)` stets das gleiche Ergebnis wie `y.equals(x)` (symmetrisch).
  - Wenn `x.equals(y)` und `y.equals(z)` beide `true` liefern, dann tut dies auch `x.equals(z)` (transitiv).
- Solange `x` und `y` nicht geändert werden, liefern wiederholte Aufrufe von `x.equals(y)` stets gleiche Ergebnisse.

Eine Implementierung, die nicht alle dieser Bedingungen erfüllt, muss als fehlerhaft betrachtet werden, auch wenn weder der Compiler noch das Laufzeitsystem einen Fehler meldet.

hashCode

Zusammen mit `equals` müssen wir immer auch die in `Object` vordefinierte Methode `int hashCode()` überschreiben. Diese Methode gibt eine fast beliebige ganze Zahl zurück. Wird die Methode nicht überschrieben, gibt sie einen Ausschnitt aus der Objektadresse zurück. Jede Implementierung muss folgende Bedingungen erfüllen:

Eigenschaften von hashCode

- Hat `x.equals(y)` als Ergebnis `true`, dann gibt `x.hashCode()` dieselbe Zahl zurück wie `y.hashCode()`.
- Solange `x` nicht geändert wird, liefern wiederholte Aufrufe von `x.hashCode()` stets gleiche Ergebnisse.

Es gibt also einen Zusammenhang zwischen `hashCode` und `equals`. Beim Implementieren von `hashCode` achten wir darauf, dass unterschiedliche Objekte zu möglichst unterschiedlichen Ergebnissen von `hashCode` führen, obwohl gleiche Ergebnisse nicht auszuschließen sind:

Listing 3.15 Implementierung von `hashCode` als Erweiterung von Listing 2.5

```
1  public int hashCode() {
2      int h = 7654321 + 7 * textWidth + textHeight;
3      for (int i = 0; i < textWidth; i++) {
4          for (int j = 0; j < textHeight; j++) {
5              h = 1234567 * h + text[i][j];
6          }
7      }
8      return h;
9  }
```


Häufige Fehler bestehen darin, Ergebnisse der Methoden von `Object` auf unzulässige Weise miteinander zu verknüpfen. Es stimmt zwar, dass `x.hashCode() == y.hashCode()` aus `x.equals(y)` folgt, aber aus `x.hashCode() == y.hashCode()` folgt nicht `x.equals(y)`. Gilt `x.equals(y)`, dann oft auch `x.toString().equals(y.toString())`. Jedoch dürfen wir aus `x.toString().equals(y.toString())` keinesfalls `x.equals(y)` folgern. Die Zeichenketten und Hash-Werte sind oft sogar dann gleich, wenn die Objekte unterschiedliche Typen haben.

Hash-Tabelle

Hash-Werte wie die Ergebnisse von `hashCode` verwenden wir vor allem in *Hash-Tabellen* wie in Listing 3.16:

Listing 3.16 Assoziative Datenstruktur implementiert als Hash-Tabelle

```
1  public class HashTab implements Assoc {
2      private String[] ks = new String[65];
3      private String[] vs = new String[65];
4      private int count = 0;
5
6      private int find(String k) {
7          if (k == null) return ks.length - 1;
8          int i = k.hashCode() & (ks.length - 2);
9          while (ks[i] != null && ! ks[i].equals(k))
10             i = (i + 1) & (ks.length - 2);
11         return i;
12     }
13
14     public String put(String k, String v) {
15         if (k == null) { return null; }
16         int i = find(k);
17         String old = vs[i];
18         vs[i] = v;
19         if (ks[i] == null) {
20             ks[i] = k;
21             if (++count >= 0.75 * ks.length) {
22                 String[] oks = ks, ovs = vs;
23                 ks = new String[(oks.length << 1) - 1];
24                 vs = new String[(oks.length << 1) - 1];
25                 for (int j = 0; j < oks.length; j++) {
26                     if (oks[j] != null) {
27                         ks[i = find(oks[j])] = oks[j];
28                         vs[i] = ovs[j];
29                     }
30                 }
31             }
32         }
33         return old;
34     }
```

```
35      public String get(String k) { return vs[find(k)]; }
36      public boolean containsKey(String k) {
37          return ks[find(k)] != null;
38      }
39      public boolean containsValue(String v) {
40          for (int i = 0; i < vs.length - 1; i++) {
41              if (v == null ? ks[i] != v && vs[i] == v
42                            : v.equals(vs[i]))
43                  return true;
44          }
45          return false;
46      }
47      public String toString() {
48          String s = "";
49          for (int i = 0; i < ks.length - 1; i++) {
50              if (ks[i] != null)
51                  s += (s.isEmpty() ? "(" : ", (")
52                        + ks[i] + ", " + vs[i] + ")";
53          }
54          return "{" + s + "}";
55      }
56      public boolean equals(Object o) {
57          if (this == o) { return true; }
58          if (o == null || o.getClass() != HashTab.class)
59              return false;
60          HashTab t = (HashTab)o;
61          if (count != t.count) return false;
62          for (int i = 0; i < ks.length - 1; i++) {
63              if (ks[i] != null
64                   && (vs[i] == null
65                        ? !t.containsKey(ks[i])
66                            || t.get(ks[i]) != null
67                        : !vs[i].equals(t.get(ks[i]))))
68                  return false;
69          }
70          return true;
71      }
72      public int hashCode() {
73          int h = count;
74          for (int i = 0; i < ks.length - 1; i++) {
75              if (ks[i] != null) {
76                  h += ks[i].hashCode();
77                  if (vs[i] != null)
78                      h += vs[i].hashCode();
79              }
80          }
81          return h;
82      }
83  }
```


Hash-Tabellen sind Datenstrukturen, die Einträge in Arrays ablegen, wobei Indexe im Wesentlichen den aus den Einträgen errechneten Hash-Werten entsprechen. Damit können wir leicht feststellen, ob ein bestimmtes Objekt im Array enthalten ist: Wir berechnen den Hash-Wert des Objekts und schauen an diesem Index nach, ob dort der gesuchte Wert enthalten ist. Dieses Vorgehen hat einen Haken: Hash-Werte sind nicht eindeutig, mehrere Objekte können denselben Hash-Wert und damit denselben Index im Array haben. Es gibt zahlreiche Möglichkeiten um mit solchen *Kollisionen* umzugehen. In der Implementierung in Listing 3.16 verstehen wir den errechneten Hash-Wert als Anfangsindex für eine lineare Suche. Die Suche können wir abbrechen, sobald wir auf einen leeren Eintrag stoßen. Ein kleineres Problem besteht darin, dass Hash-Werte nicht an Arraylängen angepasst sind. Dieses Problem lässt sich leicht dadurch lösen, dass wir nur einen passenden Ausschnitt aus dem Bitmuster des Hash-Werts verwenden. Im Beispiel sind es die niederwertigsten Bits, oft werden aber höherwertige Bits gewählt.

Hash-Tabelle

Kollision

Aufgabe 3.9 Beschreiben Sie die Funktionsweise von Hash-Tabellen wie in Listing 3.16, insbesondere

- wie `find` aus dem Hash-Wert den Index berechnet,
- warum `ks` und `vs` um 1 länger sind als eine Zweierpotenz und welche Bedeutung der letzte Arrayeintrag hat,
- wozu `count` dient und warum der Wert von `count` stets kleiner ist als die Arraylängen,
- warum bei `k == null` kein Eintrag erfolgt,
- warum beim Vergrößern der Arrays für alle Einträge mittels `find` neue Indexe berechnet werden statt die Einträge nur zu kopieren,
- warum `containsValue` alle Einträge von `vs` durchläuft und in Vergleiche manchmal auch Einträge von `ks` einbezieht,
- warum `containsValue`, `toString`, `equals` und `hashCode` jeweils den letzten Arrayeintrag ignorieren,
- wozu `s.isEmpty()` in `toString` dient,
- warum `equals` nicht einfach nur Einträge am selben Index vergleicht, sondern stattdessen `get` verwendet,

- warum in Zeile 65 ein Aufruf von `containsKey` nötig ist,
- warum es sinnvoll ist, in `hashCode` möglichst alle Einträge einzubeziehen, obwohl die Berechnung dadurch aufwendiger wird,
- wie mit Kollisionen umgegangen wird,
- was passiert, wenn für viele unterschiedliche `k` das Ergebnis von `k.hashCode() & (ks.length - 2)` gleich ist.

Aufgabe 3.10 Beschreiben Sie den Unterschied zwischen *Identität* und *Gleichheit* sowie den Zusammenhang mit `==` und `equals` (sollte schon aus EP1 bekannt sein).

Aufgabe 3.11 Beschreiben Sie, wie eine Implementierung von `equals` üblicherweise aussieht und warum sie so aussieht.

Aufgabe 3.12 Beschreiben Sie die Bedingungen, die jede Implementierung von `equals` erfüllen muss, und was passieren kann, wenn eine Bedingung (jede getrennt für sich) nicht erfüllt ist.

Aufgabe 3.13 Beschreiben Sie, wie Implementierungen von `hashCode` von jenen von `equals` abhängen, welche Bedingungen `hashCode` erfüllen muss, und was passieren kann, wenn eine Bedingung (jede getrennt für sich) nicht erfüllt ist.

Aufgabe 3.14 Die Implementierungen von `equals` und `hashCode` müssen nicht von den Inhalten aller Objektvariablen abhängen. Beschreiben Sie, welche Variablen Sie einbeziehen würden und welche nicht. Beschreiben Sie auch, von welchen Variablen `hashCode` abhängen darf bzw. muss, wenn `equals` nicht von allen Variablen abhängt.

Aufgabe 3.15 Implementieren Sie eine Klasse zum Testen von `Assoc`. Befüllen Sie ein Objekt `x` von `Assoc` z. B. durch `x.put("a", "b");` und `x.put("c", "d");` und ein anderes Objekt `y` von `Assoc` durch `x.put("a, b), (c", "d");`. Beschreiben Sie anhand dieses Beispiels, warum es fast immer falsch ist, die Gleichheit zweier Objekte durch einen Vergleich der durch Anwendung von `toString` auf diese Objekte erzeugten Zeichenketten festzustellen. Finden Sie weitere Beispiele, in denen Vergleiche über `toString` falsche Ergebnisse liefern.

Aufgabe 3.16 Beschreiben Sie, wieso `x.hashCode()==y.hashCode()` nicht äquivalent zu `x.equals(y)` ist, auch dann nicht, wenn `y!=null` angenommen wird und `hashCode` (wie in `Object` implementiert) soweit wie möglich die Objektadresse zurückgibt.

Aufgabe 3.17 Wählen Sie einen beliebigen allgemeinen Begriff aus Ihrem Umfeld (z. B. „Schreibutensilien“), unterteilen Sie diesen Begriff in etwas konkretere (z. B. „Schreibstifte“, „Papierwaren“ und „Heftmaschinen“) und ganz konkrete Begriffe (z. B. „Kugelschreiber“, „Bleistift“, etc.), und suchen Sie eine Umgebung, in der diese Begriffe gemeinsam auftreten (z. B. „Schreibtischlade“). Erstellen Sie zu jedem Begriff eine Klasse oder ein Interface und setzen Sie diese soweit wie möglich in Untertypbeziehungen zueinander. Simulieren Sie typische Situationen in der Verwendung der Begriffe (z. B. „Aufsatz schreiben“) möglichst detailliert (z. B. „Schreibstift aus Ablage 1 und Blatt Papier aus Ablage 2 der Schreibtischlade nehmen, ...“) durch Programmtexte in den jeweiligen Klassen. Versuchen Sie, Variablen und Parameter mit Typen zu deklarieren, die möglichst weit oben in der Typhierarchie stehen (z. B. „Schreibstift“ statt „Kugelschreiber“), ohne dass dadurch der Einsatz von Typumwandlungen notwendig wird.

3.2 Einheitliche Konzepte

Bei Betrachtung der oben eingeführten abstrakten Datentypen im Zusammenhang mit Datenstrukturen fällt auf, dass zwischen ihnen viele Gemeinsamkeiten auf konzeptueller Ebene, aber bedeutende Unterschiede im Detail bestehen. Gemeinsamkeiten kommen daher, dass wir immer wieder dieselben grundlegenden Operationen ausführen – Einfügen, Suchen, Löschen und so weiter. Unterschiede kommen von unterschiedlichen Verwendungsmustern, gezielter Nutzung bestimmter Eigenschaften der Datenstrukturen oder schlicht historischen Entwicklungen – etwa unterschiedliche Benennung vergleichbarer Operationen. Wir bauen nun exemplarisch eine möglichst einheitliche Abstraktionshierarchie auf, berücksichtigen aber begründbare Unterschiede.

3.2.1 Iterierbare Abstraktionen

Iterator

Bei der Erstellung einer Hierarchie von Abstraktionen stehen Anwendungsfälle im Mittelpunkt der Überlegungen. Daraus ergeben sich Interfaces mit typischen Operationen für Stacks, Queues, Listen, Hash-Tabellen, etc. Das in Java wichtige Konzept des *Iterators* kommt hinzu:

Listing 3.17 Iterierbare Datenstruktur

```
1 public interface Iterable extends java.lang.Iterable<String>[4]
2 {   Iterator iterator();   }
```

Listing 3.18 Iterator lässt Inhalte von iterierten Datenstrukturen unverändert

```
1 public interface Iterator extends java.util.Iterator<String>[4]
2 {   String next();
3     boolean hasNext();
4 }
```

Ein Aufruf von `iterator()` in einer Datenstruktur vom Typ `Iterable` erzeugt einen Iterator, über den alle Einträge der Datenstruktur nacheinander gelesen werden. Ein Aufruf von `next()` gibt den nächsten Eintrag zurück, und das Ergebnis von `hasNext()` besagt, ob es weitere Einträge gibt – vergleichbar mit gleichnamigen Methoden von `Scanner`. Im Unterschied zu `poll()` in einer Queue lässt `next()` die Datenstruktur unverändert. Beliebig viele Iteratoren können gleichzeitig über dieselbe Datenstruktur iterieren. Hier ist eine einfache Anwendung:

Listing 3.19 Verwendung eines Iterators – (A) lange und (B) kurze Variante

```
1 private static void printAllA(Iterable data) {
2     Iterator i = data.iterator();
3     while (i.hasNext()) System.out.println)(i.next());
4 }
5 private static void printAllB(Iterable data) {
6     for (String s : data) System.out.println)(s);
7 }
```

Java

for each

Iteratoren sind so tief in Java integriert, dass sie durch spezielle Schleifen (wie auf Arrays) syntaktisch unterstützt werden: In Listing 3.19 ist `printAllB` semantisch äquivalent zu `printAllA`, da die ForEach-Schleife über einem Objekt von `Iterable` automatisch einen Iterator erzeugt und über diesen das Objekt von Anfang bis Ende durchläuft.[4]

Als Ergänzung zu Stacks und Queues erlauben Iteratoren das mehrfache Durchlaufen. Die Reihenfolge der Iterationen ist beliebig wählbar, muss also nicht immer LIFO- oder FIFO-Verhalten aufweisen.

Listing 3.20 Stack mit typischen Stack-Operationen

```
1 public interface Stack extends Iterable {
2     void push(String element);
3     String pop();
4     String peek();
5     int size();
6 }
```

[4] In `Iterable` und `Iterator` stellen `extends java.lang.Iterable<String>` und `extends java.util.Iterator<String>` Verbindungen zu gleichnamigen vordefinierten Interfaces her, damit ForEach-Schleifen funktionieren. Bei Verzicht auf ForEach-Schleifen sind diese Klauseln nicht nötig.

Listing 3.21 Queue mit typischen Queue-Operationen

```
1 public interface Queue extends Iterable {
2     void add(String element);
3     String poll();
4     String peek();
5     int size();
6 }
```

Listing 3.22 Double-Ended-Queue verallgemeinert Stack und Queue

```
1 public interface Deque extends Queue, Stack {
2     void addFirst(String element);
3     void addLast(String element);
4     String pollFirst();
5     String pollLast();
6     String peekFirst();
7     String peekLast();
8 }
```

Als Untertyp von `Queue` und `Stack` muss `Deque` die Methoden der Obertypen enthalten, z. B. `pop()` und `poll()`. Manche Methoden sind semantisch gleich, etwa `add` und `addLast`.

Listing 3.23 Assoziative Datenstruktur

```
1  public interface Map {
2      String put(String key, String value);
3      String get(String key);
4      String remove(String key);
5      boolean containsKey(String key);
6      boolean containsValue(String value);
7      Collection keys();
8      Collection values();
9      int size();
10 }
```

`Map` ist nicht Untertyp von `Iterable`, da nicht klar ist, worüber iteriert werden soll. Zum Ausgleich erzeugen die Methoden `keys()` und `values()` Sammlungen von Daten, über die iteriert werden kann:

Listing 3.24 Sammlung von Datenelementen

```
1 public interface Collection extends Iterable {
2     void add(String element);
3     int contains(String element);
4     int removeAll(String element);
5     void clear();
6     int size();
7     String[] toArray();
8 }
```

`clear()` löscht alle Einträge in einem Schritt, und `toArray()` bietet eine weitere Möglichkeit für den Zugriff auf alle Einträge. `Collection` schreibt keine bestimmte Reihenfolge der Einträge vor. Aber `Ordered` hält die Einträge sortiert, sodass sie sowohl im Ergebnis von `toArray()` als auch bei Zugriff über Iteratoren in sortierter Reihenfolge vorliegen:

Listing 3.25 Sortierte Sammlung von Datenelementen, keine weiteren Methoden

```
1 public interface Ordered extends Collection { }
```

Obwohl in `List` Einträge durch `sort()` sortierbar sind, kann die Sortierung durch `add` (neu einfügen) oder `set` (bestehenden Eintrag ersetzen) an einem bestimmten Index wieder verletzt werden:

Listing 3.26 Sammlung von über einen Index zugreifbaren Datenelementen

```
1 public interface List extends Collection {
2     int indexOf(String element);
3     String get(int index);
4     boolean add(int index, String element);
5     String set(int index, String element);
6     String remove(int index);
7     void sort();
8     Iterator sortedIterator();
9 }
```

Diese Interfaces sind auf viele Arten implementierbar. Für `Stack`, `Queue`, `Deque`, `Ordered` und `List` eignen sich Arrays und Listen. Wegen der nötigen linearen Ordnung sind übliche Hash-Tabellen ungeeignet und Bäume (abgesehen von `Ordered`) wenig geeignet. Bäume eignen sich für `Map`, `Collection` und `Ordered`, Hash-Tabellen ausgezeichnet für `Map` und `Collection`. Je nach Implementierung haben die abstrakten Datentypen unterschiedliche Eigenschaften, etwa andere Effizienz beim Suchen, Einfügen, Löschen, etc. Idealerweise sind alle geeigneten Implementierungsvarianten verfügbar um jene Variante wählen zu können, deren Eigenschaften für die jeweiligen Anwendungsfälle (je nach Häufigkeit von Suchen, Einfügen, Löschen, etc.) am besten passen. Oft werden Implementierungstechniken kombiniert, etwa eine Hash-Tabelle mit einem Baum zur Kollisionsbehandlung.[5]

Betrachten wir die Implementierung eines Iterators für eine Queue basierend auf einer Liste. Iterator-Implementierungen stehen in eigenen Klassen, und die Queue benötigt eine Methode zum Erzeugen eines speziell auf die Implementierung der Queue zugeschnittenen Iterators:

[5] Mit guten Hash-Funktionen ist diese Kombination aber nicht sehr sinnvoll.

Listing 3.27 Ausschnitt aus einer Implementierung von `Queue`

```
1 public class ListQueue implements Queue {
2     // Implementierung von ListQueue erweitert Listing 2.22, Seite 64
3     public Iterator iterator() {
4         return new ListNodeIter(head);
5     }
6 }
```

Listing 3.28 Implementierung eines Iterators über Listenknoten

```
1  public class ListNodeIter implements Iterator {
2      private ListNode n;  // Listing 2.18, Seite 61
3      public ListNodeIter(ListNode n) { this.n = n; }
4      public boolean hasNext() { return n != null; }
5      public String next() {
6          if (n == null) { return null; }
7          String result = n.value();
8          n = n.next();
9          return result;
10     }
11 }
```

Iteratoren brauchen Zugriff auf Implementierungsdetails, etwa das Array mit den Einträgen, weswegen Iteratoren meist gemeinsam mit Datenstrukturen implementiert werden müssen:

Listing 3.29 Bei Iteratorerzeugung wird Array weitergegeben

```
1 public class DEQueue implements Deque {
2     // Implementierung von DEQueue erweitert Listing 2.16, Seite 53
3     public Iterator iterator() {
4         return new ArrayIter(es, head, count);
5     }
6 }
```

Listing 3.30 Iterator mit direktem Zugriff auf Array aus `DEQueue`

```
1  public class ArrayIter implements Iterator {
2      private String[] es;
3      private int i, count;
4      public ArrayIter(String[] es, int i, int count) {
5          this.es = es;  this.i = i;  this.count = count;
6      }
7      public boolean hasNext() { return count > 0; }
8      public String next() {
9          if (count <= 0) { return null; }
10         count--;
11         return es[i++ & (es.length - 1)];
12     }
13 }
```

Aufgabe 3.18 Beschreiben Sie Gemeinsamkeiten und Unterschiede in der Verwendung eines Iterators und eines Scanners.

Aufgabe 3.19 Beschreiben Sie, wie die ForEach-Schleife auf einem Objekt von `Iterable` funktioniert.

Aufgabe 3.20 Beschreiben Sie, welche Methoden durch die Interfaces in Listing 3.20 bis Listing 3.26 jeweils spezifiziert werden, wozu diese Methoden wahrscheinlich verwendbar sind und was jede der Methoden Ihrer Meinung nach bewirkt oder bewirken sollte. Versehen Sie die Methoden in den Interfaces mit entsprechenden Kommentaren.

Aufgabe 3.21 `Map` ist kein Untertyp von `Iterable`. Beschreiben Sie, wie dennoch über alle Schlüssel und damit assoziierten Werte in einem Objekt vom Typ `Map` iteriert werden kann.

Aufgabe 3.22 Beschreiben Sie, warum ein Iterator fast immer in einer anderen Klasse implementiert ist (z. B. `ListNodeIter` in Listing 3.28) als der, in der ein Objekt davon erzeugt wird (z. B. `ListQueue` in Listing 3.27). Hinweis: Es muss möglich sein, dass mehrere Iteratoren gleichzeitig über dasselbe Objekt iterieren.

Aufgabe 3.23 Beschreiben Sie, wieso die Methode `iterator()` ein Ergebnis vom deklarierten Typ `Iterator` zurückgeben kann, obwohl ein Objekt eines Untertyps (z. B. `ListNodeIter`) erzeugt wird.

3.2.2 Baum als sortierte Datensammlung

Listing 3.31 zeigt zusammen mit Listing 3.32 und Listing 3.33 eine beispielhafte Implementierung von `Ordered`. Durch die Verwendung eines binären Suchbaums ergibt sich die Sortierung von selbst.

Listing 3.31 Implementierung von `Ordered` als binärer Suchbaum

```
1  import java.util.Arrays;
2  public class OrderedTree implements Ordered {
3      private OTNode root;
4      private int[] cnt = new int[1];
5
6      public OrderedTree() {}
7      public OrderedTree(String[] elems) {
8          for (String e : elems) { add(e); }
9      }
10     public OrderedTree(Iterable elems) {
11         for (String e : elems) { add(e); }
12     }
```



```
13     public void add(String e) {
14         cnt[0]++;
15         if (root != null) { root.add(e); }
16         else { root = new OTNode(e); }
17     }
18     public int contains(String e) {
19         if (root == null) { return 0; }
20         return root.contains(e);
21     }
22     public int removeAll(String e) {
23         int oldcnt = cnt[0];
24         if (root != null) { root = root.rmv(e, cnt); }
25         return oldcnt - cnt[0];
26     }
27     public void clear() { root = null; cnt[0] = 0; }
28     public int size() { return cnt[0]; }
29     public String[] toArray() {
30         String[] array = new String[size()];
31         toArray(array);
32         return array;
33     }
34     public int toArray(String[] array) {
35         return root == null ? 0 : root.fill(array, 0);
36     }
37     public Iterator iterator() {
38         OTIter iter = new OTIter();
39         if (root != null) { root.iter(iter, false); }
40         return iter;
41     }
42     public String toString() {
43         String res = "";
44         for (String s : this)
45             res += (res.isEmpty() ? "" : ", ") + s;
46         return "{" + res + "}";
47     }
48     public boolean equals(Object o) {
49         if (this == o) { return true; }
50         if (o == null || o.getClass() != getClass())
51             return false;
52         OrderedTree t = (OrderedTree)o;
53         return Arrays.equals(t.toArray(), toArray());
54     }
55     public int hashCode() {
56         return 57 - Arrays.hashCode(toArray());
57     }
58 }
```

Am intuitivsten wäre es, zwei Objekte von `OrderedTree` zu vergleichen, indem wir über beide Objekte iterieren und die Einträge in sor-

tierter Reihenfolge paarweise vergleichen. Um eine Alternative zu zeigen, betrachten wir im Beispiel zwei Objekte als gleich, wenn die daraus erzeugten Arrays gleich sind. Eine `import`-Anweisung in Zeile 1 macht die Klasse `Arrays` sichtbar, die unter anderem Methoden für Gleichheitsvergleiche und zur Berechnung von Hash-Werten von Arrays enthält. Wir verwenden diese Methoden, weil es auf Arrays (die direkt erzeugt werden, nicht über Klassen) keine Möglichkeit zum Überschreiben von `equals` und `hashCode` aus `Object` gibt.

Array für Änderbarkeit

Die Variable `cnt` enthält die Zahl der Einträge in der Datensammlung als Arrayeintrag. Ein Array wurde gewählt, damit es als Argument an eine Methode übergeben und in der Methode verändert werden kann, ohne dass die Änderung bei der Rückkehr verloren geht.[6]

Listing 3.32 Implementierung eines Baumknotens

```
1  public class OTNode {
2      private String elem;
3      private OTNode left, right;
4
5      public OTNode(String e) { elem = e; }
6
7      public int fill(String[] array, int i) {
8          if (left != null) { i = left.fill(array, i); }
9          if (i < array.length) {
10             array[i++] = elem;
11             if (right != null) i = right.fill(array,i);
12         }
13         return i;
14     }
15     private int compare(String e) {
16         if (e == null) { return elem == e ? 0 : -1; }
17         return elem == null ? 1 : e.compareTo(elem);
18     }
19     public void add(String e) {
20         int cmp;
21         OTNode p, n = this;
22         do {
23             cmp = (p = n).compare(e);
24             n = cmp > 0 ? p.right : p.left;
25         } while (n != null);
26         if (cmp > 0) { p.right = new OTNode(e); }
27         else { p.left = new OTNode(e); }
28     }
```

[6] Eine einfache `int`-Variable für `cnt` würde reichen, wenn `OTNode` und `OTIter` innere Klassen von `OrderedTree` wären. Darauf wurde verzichtet, damit Wesentliches nicht durch viele für uns neue Java-Sprachkonstrukte verschleiert wird.


```
29     public int contains(String e) {
30         int count = 0;
31         OTNode n = this;
32         do {
33             int cmp = n.compare(e);
34             if (cmp > 0) { n = n.right; }
35             else {
36                 n = n.left;
37                 if (cmp == 0) { count++; }
38             }
39         } while (n != null);
40         return count;
41     }
42     public OTNode rmv(String e, int[] cnt) {
43         int cmp = compare(e);
44         if (cmp > 0) {
45             if (right != null)
46                 right = right.rmv(e, cnt);
47         } else {
48             if (left != null) left = left.rmv(e, cnt);
49             if (cmp == 0) {
50                 cnt[0]--;
51                 if (left == null) { return right; }
52                 OTNode p = left;
53                 while (p.right != null) { p=p.right; }
54                 p.right = right;
55                 return left;
56             }
57         }
58         return this;
59     }
60     public String iter(OTIter iter, boolean next) {
61         OTNode n = next ? right : this;
62         while (n != null) {
63             new OTIter(n, iter);
64             n = n.left;
65         }
66         return elem;
67     }
68 }
```

Sortierung im Suchbaum

Die Sortierung ergibt sich durch den Aufbau des Suchbaums von selbst. Wir müssen nur zuerst den linken Teilbaum, dann den eigenen Eintrag, und am Ende den rechten Teilbaum betrachten. So sorgt auch der Iterator für die richtige Reihenfolge. Im Gegensatz zu `fill` kann der Iterator keine rekursiven Methodenaufrufe verwenden, weil nur ein Eintrag nach dem anderen gelesen wird. In gewisser Weise müssen wir im Iterator den bei Rekursion verwendeten Systemstack simulieren: Ein

keine Rekursion bei Iterator

Objekt vom Typ `OTIter` referenziert einen Baumknoten sowie ein weiteres Objekt vom Typ `OTIter`. Dabei ergibt sich eine lineare Liste, in der der erste Listenknoten den als nächsten zu behandelnden Baumknoten referenziert, der zweite dessen Vorgänger im Baum und so weiter. Die Liste enthält also in umgekehrter Reihenfolge den gesamten Pfad von der Baumwurzel bis zum aktuellen Baumknoten. Am Ende steht ein leeres Objekt vom Typ `OTIter`, das einen Iterator darstellt, in dem `hasNext` stets `false` liefert.

Listing 3.33 Iterator über Baum als rekursive Datenstruktur

```
1  public class OTIter implements Iterator {
2      private OTNode node;
3      private OTIter parent;
4
5      public OTIter() { }
6      public OTIter(OTNode n, OTIter p) {
7          node   = p.node;    p.node   = n;
8          parent = p.parent;  p.parent = this;
9      }
10     public boolean hasNext() { return node != null; }
11     public String next() {
12         if (node == null) { return null; }
13         OTNode todo = node;
14         node = parent.node;
15         parent = parent.parent;
16         return todo.iter(this, true); }
17     }
18 }
```

Aufgabe 3.24 Beschreiben Sie `OrderedTree` und die damit zusammenhängenden Klassen in Listing 3.31 bis Listing 3.33 unter anderem durch Beantwortung folgender Fragen:

- Wozu dient die Variable `cnt`, und was hat die Verwendung eines Arrays dafür damit zu tun, dass Methoden in Java nur *einen* Wert als Ergebnis zurückgeben können?
- Wozu dienen die zwei Konstruktoren von `OrderedTree`, die durch den gleichen Rumpf implementiert sind, und wodurch unterscheiden sie sich in der Ausführung?
- Was macht `compare` in Listing 3.32 im Detail?
- Warum wird die Schleife in `contains` (Listing 3.32) weiter ausgeführt, nachdem das gesuchte Element gefunden wurde?

- Was macht `rmv` aus Listing 3.32 im Detail, insbesondere die Zeilen 49 bis 56, und warum werden `root`, `left` und `right` nach einem entsprechenden Aufruf neu gesetzt?
- Was macht `fill` aus Listing 3.32 im Detail, wozu dient der Parameter `i` und wofür steht das Ergebnis?
- Wieso sind die Ergebnisse von `toArray` und `toString` sortiert, und durch welche kleinen Änderungen können wir die Sortierreihenfolgen umdrehen, ohne `add` oder `compare` zu ändern?
- Die Funktionsweise des Iterators ist schwer zu verstehen. Was machen die Methoden `iterator` und `iter` sowie die Klasse `OTIter`, und wie arbeiten diese Teile zusammen?
- Wieso vergleicht `equals` zwei aus den Bäumen erzeugte Arrays und nicht direkt die Struktur der Bäume? Warum wäre es falsch, wenn wir statt der Ergebnisse von `toArray` die Ergebnisse von `toString` vergleichen würden?
- Warum wird `toArray` auch in `hashCode` verwendet?

Aufgabe 3.25 Schreiben Sie ein Programm zum Testen von `Ordered` aus Listing 3.31 und den damit zusammenhängenden Klassen, insbesondere zum Testen des Iterators. Fügen Sie an geeigneten Stellen Anweisungen zur Erzeugung von Debug-Output ein, der Ihnen dabei hilft, die Funktionsweise des Iterators nachzuvollziehen. Beschreiben Sie anhand des Debug-Outputs, inwiefern ein Iterator intern einen Stack darstellt und wodurch die Einträge in den Stack zustandekommen.

Aufgabe 3.26 Ändern Sie Listing 3.31 bis Listing 3.33 so ab, dass `OTIter` nur eine Objektvariable vom Typ `java.util.Deque<OTNode>` enthält, wobei das Objekt als Stack verwendet wird. Beschreiben Sie, wie sich der Programmtext dadurch vereinfacht.

Aufgabe 3.27 Ändern Sie Listing 3.31 bis Listing 3.33 so ab, dass `OTNode` neben `elem`, `left` und `right` eine Objektvariable `parent` vom Typ `OTNode` enthält, die den Elternknoten (Vorgänger) im Baum referenziert. In Analogie zur doppelt verketteten Liste wäre dies ein doppelt verketteter Baum. Beschreiben Sie, wie sich die Änderung auf die Implementierungen der Methoden auswirkt. Ändern Sie `OTIter` so ab, dass die Iterator-Implementierung ohne einen Stack als Hilfsmittel auskommt. Beschreiben Sie, warum der doppelt verkettete Baum das ermöglicht, der einfache Baum jedoch nicht.

3.2.3 Sortieren von Listen

Bubblesort

Das Sortieren zählt zu den üblichen Operationen auf Datenstrukturen. Einfache Sortierverfahren auf Arrays sind uns schon aus EP1 bekannt. Diese Verfahren sind im Prinzip auch für rekursive Datenstrukturen verwendbar, wobei jedoch beachtet werden muss, dass nicht alle Einträge mit optimaler Effizienz zugreifbar sind. Eines der einfachsten Verfahren ist *Bubblesort.* Dieses Verfahren ist auch für einfache lineare Listen geeignet, weil nur benachbarte Listenknoten in der vorgegebenen Reihenfolge verglichen werden müssen:

Listing 3.34 Bubblesort in einer einfachverkettete Liste

```
1  public class LinearList implements List {
2      private ListNode head;
3      public void sort() {
4          if (head != null) {
5              boolean modified;
6              do {
7                  modified = false;
8                  ListNode m, n = head;
9                  while ((n = (m = n).next()) != null) {
10                     String x=m.value(), y=n.value();
11                     if (x != null && (y == null
12                                 || x.compareTo(y) < 0)) {
13                         m.setValue(y);
14                         n.setValue(x);
15                         modified = true;
16                     }
17                 }
18             } while (modified);
19         }
20     }
21     public Iterator sortedIterator() {
22         return new SortedListNodeIter(head);
23     }
24     // zahlreiche weitere Methoden zu implementieren ...
25 }
```

Bubblesort traversiert die Liste wiederholt, wobei Einträge in benachbarten Knoten miteinander verglichen und bei Bedarf vertauscht werden, bis sich bei einer Traversierung keine Änderung mehr ergibt. Der nötige Umgang mit `null` steigert die Komplexität, das Vertauschen von Knoteninhalten statt Knoten vereinfacht die Implementierung.

Sortieren durch Iterator

`List` enthält neben `iterator` eine Methode `sortedIterator`, die einen Iterator erzeugt, welcher in sortierter Reihenfolge über die Liste iteriert, ohne die Sortierung der Liste selbst zu ändern:

Listing 3.35 Iterator sucht nach dem kleinsten noch nicht bearbeiteten Eintrag

```
1  public class SortedListNodeIter implements Iterator {
2      private ListNode head, next;
3      public SortedListNodeIter(ListNode head) {
4          this.head = head;
5          setNext();
6      }
7      private static boolean prec(ListNode a,
8                                  ListNode b, int c) {
9          String x = a.value(), y = b.value();
10         if (y == null) { return x == null && c == 0; }
11         return x == null || y.compareTo(x) >= c;
12     }
13     private void setNext() {
14         int c = 0;
15         ListNode n = head, min = null;
16         while (n != null) {
17             if (n == next) { c = 1; }
18             else if ((next==null || prec(next,n,c))
19                       && (min==null || prec(n,min,0)))
20                 min = n;
21             n = n.next();
22         }
23         next = min;
24     }
25     public boolean hasNext() { return next != null; }
26     public String next() {
27         if (next == null) { return null; }
28         String result = next.value();
29         setNext();
30         return result;
31     }
32 }
```

Die Iterator-Klasse aus Listing 3.35 durchsucht vor jedem Aufruf von `next` die gesamte Liste nach dem kleinsten Eintrag, der noch nicht zurückgegeben wurde. Bis zum zuletzt zurückgegebenen Knoten `next` werden Einträge berücksichtigt, die größer oder gleich `next.value()` sind (bei `c == 0`), danach nur echt größere Einträge (`c == 1`). So wird bei gleichen Einträgen immer der am weitesten hinten stehende noch nicht zurückgegebene Eintrag gewählt. Sortiert wird schrittweise. Vergleiche (`prec`) sind wegen der Berücksichtigung von `null` komplex.

Mergesort

Die bisher betrachteten Sortierverfahren sind wenig effizient. *Mergesort* ist dagegen ein recht effizienter rekursiver Algorithmus, der sich besonders gut für Listen eignet. Zur Vermeidung unnötiger Komplexität bei Zugriffen steht folgende Implementierung in `ListNode`:

Listing 3.36 Mergesort auf Listenknoten

```
1  public class ListNode {   // erweitert Listing 2.18, Seite 61
2      private String value;
3      private ListNode next;
4      private static boolean cmp(ListNode a,ListNode b) {
5          return a.value == null || (b.value != null
6                  && a.value.compareTo(b.value) < 0);
7      }
8      public ListNode mergesort(boolean reverse) {
9          if (next == null) { return this; }
10         ListNode[] xs = new ListNode[2];
11         ListNode h, n = this;
12         for (int i = 0; n != null; i = (i + 1) & 1) {
13             h = n;
14             n = h.next;
15             h.next = xs[i];
16             xs[i] = h;
17         }
18         xs[0] = xs[0].mergesort(!reverse);
19         xs[1] = xs[1].mergesort(!reverse);
20         while (xs[0] != null) {
21             int i = (xs[1] == null
22               || cmp(xs[0], xs[1]) == reverse) ? 0 : 1;
23             h = xs[i];
24             xs[i] = h.next;
25             h.next = n;
26             n = h;
27         }
28         while (xs[1] != null) {
29             h = xs[1];
30             xs[1] = h.next;
31             h.next = n;
32             n = h;
33         }
34         return n;
35     }
36 }
```

Listing 3.37 Aufruf von `mergesort`

```
1  public class LinearList implements List {
2      private ListNode head;
3      // Alternative Implementierung von sort – siehe Listing 3.34
4      public void sort() {
5          if (head != null) {
6              head = head.mergesort(false);
7          }
8      }
9  }
```


Mergesort teilt eine Ansammlung von Daten (hier eine Liste, über `n` traversiert) zunächst in zwei annähernd gleich große Teile auf (in `xs[0]` und `xs[1]`), sortiert diese Teile durch rekursive Aufrufe und fügt die sortierten Teile zusammen, wobei nur die ersten Datenwerte der Teile verglichen werden müssen um festzustellen, von welchem Teil der nächste Wert (`n`) zu nehmen ist. Eine Schwierigkeit bei obiger Implementierung besteht darin, dass wir keine Referenz auf den letzten Listenknoten mitführen und daher das Einfügen am Listenende sehr aufwendig wäre. Zur Beibehaltung der Sortierung müssten wir aber am Ende einfügen (FIFO- statt LIFO-Verhalten). Um dennoch einfach am Listenanfang einfügen zu können, drehen wir mit jeder Rekursionsebene die Reihenfolge der Sortierung um. LIFO-Verhalten zusammen mit umgekehrter Sortierung liefert wieder die gewünschte Sortierung.

Vertauschen von Listenknoten

Anders als in Listing 3.34 vertauscht Listing 3.36 Listenknoten, nicht nur Einträge. Das ist vorteilhaft: Offensichtlich hängt der Aufwand nicht davon ab, wie viele Daten in welcher Form in einem Knoten gespeichert sind. Weniger offensichtlich ist die bessere Robustheit im Falle von Referenzen auf einzelne Knoten, die im Programm existieren, etwa in einem Iterator. Ändern wir essentielle Daten in solchen Knoten auf aus Sicht einer einzelnen Referenz unvorhersehbare Weise, wirkt sich dies mit höherer Wahrscheinlichkeit negativ auf die Korrektheit aus, als wenn wir nur Verwaltungsinformation (wie `next`) ändern.

Aufgabe 3.28 Beschreiben Sie die Funktionsweisen von `sort` aus Listing 3.34, `SortedListNodeIter` aus Listing 3.35 und `sort` bzw. `mergesort` aus Listing 3.37 und Listing 3.36, jeweils nicht nur den Algorithmus, sondern auch Implementierungsdetails.

Aufgabe 3.29 Schreiben Sie ein Programm, das die Laufzeiteffizienz von `sort` aus Listing 3.34, `SortedListNodeIter` aus Listing 3.35 und `sort` bzw. `mergesort` aus Listing 3.37 und Listing 3.36 miteinander vergleicht. Testen Sie sowohl mit zufällig gewählte Zeichenketten (z. B. die Wörter aus Texten) genauso wie schon vorsortierte Zeichenketten (z. B. diese Wörter nach dem Sortieren). Beschreiben Sie klar erkennbare Unterschiede in der Laufzeit sowie Abhängigkeiten der Laufzeiten von der Anzahl und Vorsortierung der Zeichenketten.

Aufgabe 3.30 Ändern Sie `mergesort` aus Listing 3.36 so ab, dass der Parameter `reverse` nicht mehr nötig ist, die Laufzeiteffizienz aber erhalten bleibt oder verbessert wird. Führen Sie dazu zusätzliche Referenzen auf die Listenenden ein. Vergleichen Sie die Laufzeiteffizienz der geänderten mit der ursprünglichen Variante.

3.2.4 Sichtweisen und Kopien

Listing 3.38 erweitert die Klasse `HashTab` aus Listing 3.16, sodass sie auch das Interface `Map` aus Listing 3.23 implementiert. In der Methode `remove` müssen wir mit Kollisionen rechnen: Es könnte sein, dass Einträge in die Hash-Tabelle am gleichen Index wie der zu entfernende Eintrag zu liegen gekommen wären, hätte es an diesem Index noch keinen Eintrag gegeben. Durch das Entfernen des Eintrags wären diese weiteren Einträge nicht mehr auffindbar. Daher werden in den Zeilen 17 bis 24 alle auf den zu entfernenden Eintrag direkt folgenden Einträge (bis zu einem leeren Index) ebenfalls entfernt und neu eingefügt.

Die Methoden `keys` und `values` geben Objekte von `Collection` aus Listing 3.24 zurück, welche die Schlüssel bzw. damit assoziierte Werte als Datensammlungen enthalten. Hier ist nur die Implementierung von `keys` gezeigt. Sie erzeugt ein Objekt von `KeySet` aus Listing 3.39. Eine wesentliche Eigenschaft von `KeySet` besteht darin, dass diese Klasse keine Kopien der Daten in der Hash-Tabelle enthält, sondern nur eine andere *Sichtweise* auf die Daten in der Hash-Tabelle ermöglicht. `KeySet` stellt zusätzliche Methoden zur Bearbeitung von Schlüsseln in Hash-Tabellen zur Verfügung, enthält jedoch außer einer Referenz auf die Hash-Tabelle keine Daten. Die Methoden in `KeySet` rufen Methoden der Hash-Tabelle auf. Die wesentliche Funktionalität ist in `HashTab` implementiert, unter anderem die Methoden `clear`, `size`, `keyArray` und `keyIterator`, die nur dafür gedacht sind, von `KeySet` aufgerufen zu werden. Eine Iterator-Implementierung über Schlüssel ist in Listing 3.40 gezeigt; sie greift direkt auf das Array mit den Schlüsseln in der Hash-Tabelle zu, nicht auf ein Objekt von `KeySet`.

andere Sichtweise

Die Methoden `toArray` in `KeySet` sowie `keyArray` in `HashTab` *kopieren* dagegen die Schlüssel aus einer Hash-Tabelle. Die Ausführung der Methode in Listing 3.41 verdeutlicht Gemeinsamkeiten und Unterschiede zwischen Sichtweisen und Kopien: Ausgaben durch die Zeilen 8 bis 10 sind jeweils unterschiedlich, abhängig von den dynamischen Typen von `map`, `coll` und `array`. Trotzdem beschreibt jede Ausgabe die gleiche Menge an Schlüsseln. In den Zeilen 11 bis 13 werden die über `map`, `coll` und `array` referenzierten Daten geändert. Neuerliche Ausgaben durch die Zeilen 14 bis 16 zeigen, dass sich Änderungen über `map` auch auf `coll` ausgewirkt haben, Änderungen über `coll` sich auch auf `map` ausgewirkt haben, aber Änderungen von `array` keine Auswirkungen auf `map` und `coll` zeigen, genauso wie Änderungen von `map` und `coll` keine Auswirkungen auf `array` zeigen. Zusammengefasst:

Kopie

Listing 3.38 Hash-Tabelle implementiert `Map`

```
1  public class HashTab implements Assoc, Map {
2      private String[] ks, vs;
3      private int count;
4      // übernimmt Inhalte von Listing 3.16 sofern sie hier nicht stehen
5      public HashTab() { clear(); }
6      public void clear() {
7          ks = new String[65];
8          vs = new String[65];
9          count = 0;
10     }
11     public String remove(String k) {
12         int i = find(k);
13         String result = vs[i];
14         if (ks[i] != null) {
15             ks[i] = vs[i] = null;
16             count--;
17             for (i = (i + 1) & (ks.length - 2);
18                  ks[i] != null;
19                  i = (i + 1) & (ks.length - 2)) {
20                 String ki = ks[i], vi = vs[i];
21                 ks[i] = vs[i] = null;
22                 count--;
23                 put(ki, vi);
24             }
25         }
26         return result;
27     }
28     public int size() { return count; }
29     public Collection keys() {return new KeySet(this);}
30     public int keyArray(String[] keys) {
31         int i = 0, j = -1;
32         while (i < keys.length && ++j < ks.length-1) {
33             if (ks[j] != null) { keys[i++] = ks[j]; }
34         }
35         return i;
36     }
37     public Iterator keyIterator() {
38         return new HashTabKeyIter(ks);
39     }
40     public Collection values() { ... }
```

- Durch das Kopieren von Daten (egal in welcher Darstellungsform) ergeben sich unabhängige Datenmengen, sodass spätere Änderungen der einen Datenmenge sich nicht auf die andere auswirken.

- Hinter unterschiedlichen Sichtweisen können gemeinsame Daten

Listing 3.39 Implementierung einer bestimmten Sichtweise auf `HashTab`

```
1  public class KeySet implements Collection {
2      private HashTab tab;
3      public KeySet(HashTab t) { tab = t; }
4      public void add(String element) {
5          if (!tab.containsKey(element))
6              tab.put(element, null);
7      }
8      public int contains(String e) {
9          return tab.containsKey(e) ? 1 : 0;
10     }
11     public int removeAll(String e) {
12         if (tab.containsKey(e)) {
13             tab.remove(e);
14             return 1;
15         }
16         return 0;
17     }
18     public void clear() { tab.clear(); }
19     public int size() { return tab.size(); }
20     public String[] toArray() {
21         String[] keys = new String[size()];
22         tab.keyArray(keys);
23         return keys;
24     }
25     public Iterator iterator() {
26         return tab.keyIterator();
27     }
28     public String toString() {
29         String s = "";
30         for (String e : this)
31             s += s.equals("") ? e : ", " + e;
32         return "{" + s + "}";
33     }
34     public boolean equals(Object o) { ... }
35     public int hashCode() { ... }
36 }
```

stehen, sodass sich Änderungen der Daten über eine Sichtweise auch durch andere Sichtweisen zeigen.

- Unterschiedliche Sichtweisen können unterschiedliche Darstellungen der gemeinsamen Daten nach außen bewirken.

`KeySet` zeigt, dass manchmal zusätzlicher Aufwand betrieben wird, um eine einheitliche Schnittstelle für Zugriffe auf eine Datenstruktur zu ermöglichen. So ist `add` in `KeySet` eigentlich unnötig, da ohnehin über

Listing 3.40 Iterator über die Schlüssel von `HashTab`

```
1  public class HashTabKeyIter implements Iterator {
2      private String[] ks;
3      private int i;
4      public HashTabKeyIter(String[] a) { ks = a; }
5      public boolean hasNext() {
6          for (int j = i; j < ks.length - 1; j++) {
7              if (ks[j] != null) { return true; }
8          }
9          return false;
10     }
11     public String next() {
12         while (i < ks.length - 1 && ks[i] == null)
13             i++;
14         return ks[i++];
15     }
16 }
```

Listing 3.41 `Collection`-Sichtweise versus kopierte Schlüssel einer Hash-Tabelle

```
1  private static void useHashTab {
2      Map map = new HashTab();
3      for (char c = 'A'; c <= 'Z'; c++) {
4          map.put("key" + c, "value" + c);
5      }
6      Collection coll = map.keys();
7      String[] array = coll.toArray();
8      System.out.println(map);
9      System.out.println(coll);
10     System.out.println(Arrays.toString(array));
11     map.put("newKey", "newValue");
12     coll.add("collKey");
13     array[0] = "newA";
14     System.out.println(map);
15     System.out.println(coll);
16     System.out.println(Arrays.toString(array));
17 }
```

`put` Einträge direkt in die Hash-Tabelle gemacht werden können. Bei `add` bestehen sogar die Schwierigkeiten, dass (1) kein mit dem Schlüssel assoziierter Wert gegeben ist und daher ein nicht spezifizierbarer Wert (`null`) eingetragen werden muss und (2) gar nichts eingetragen werden kann, falls der Schlüssel schon existiert. Jeder Schlüssel darf ja nur ein einziges Mal vorkommen, wodurch jedes Objekt von `KeySet` tatsächlich eine *Menge* (ohne Mehrfachvorkommen) beschreibt. Dennoch ist es zwecks einheitlicher Strukturen vorteilhaft, die Methode `add` anzubieten – keine Sonderlösung, sondern eine überall gleiche `Collection`.

Aufgabe 3.31 Beschreiben Sie im Detail, wie `remove` und `keyArray` aus Listing 3.38 funktionieren.

Aufgabe 3.32 Vervollständigen Sie die beiden Klassen `HashTab` aus Listing 3.38 (Methode `values`) und `KeySet` aus Listing 3.39 (`equals` und `hashCode`). Beschreiben Sie, welche Implementierungsprobleme dabei aufgetaucht sind und wie Sie sie gelöst haben.

Aufgabe 3.33 Beschreiben Sie anhand von Listing 3.41 den Unterschied zwischen unterschiedlichen Sichtweisen auf das gleiche Objekt und Kopien von Objekten. Beschreiben Sie auch jene Details im Programmablauf, durch die unterschiedliche Outputs zustandekommen.

Aufgabe 3.34 Schreiben Sie Varianten eines Programms, das Einträge in ein Objekt `x` vom Typ `OrderedTree`, `HashTab` oder `KeySet` (je nach Variante) macht und danach folgenden Programmtext ausführt:

```
Iterator i = x.iterator();
x.clear();
while(i.hasNext()) System.out.println(i.next());
```

Beschreiben Sie den Output und genaue Ursachen für das Zustandekommen dieses Outputs. Wiederholen Sie diese Aufgabe, wobei Sie jedoch statt aller Einträge durch `clear` nur manche Einträge entfernen.

3.3 Individuelle Eigenschaften und Effizienz

Programmieraufgaben lassen sich auf unterschiedlichste Arten lösen. Im Idealfall verringert sich der Programmieraufwand durch den Einsatz bereits vorhandener abstrakter Datentypen. Es stellt sich die Frage, welche abstrakte Datentypen wie miteinander kombiniert werden sollen um ein möglichst gutes Programm zu bekommen. Kriterien sind z. B. Laufzeit-, Speicher- und Energieeffizienz. Meist geht es nicht darum, die bestmögliche Effizienz zu bekommen, sondern darum, aus Benutzersicht inakzeptabel lange Wartezeiten oder zu kurze Akku-Laufzeiten zu vermeiden. Je genauer wir die verarbeiteten Daten kennen, um so leichter lassen sich diese Ziele erreichen. Häufig hängt die Effizienz vom Zufall ab. Das ist nicht notwendigerweise negativ, da eine große Zahl zufälliger Ereignisse in ihrer Gesamtheit recht zuverlässig abschätzbar ist, vorausgesetzt die Daten weisen tatsächlich den erwarteten Umfang und die angenommene Zufallsverteilung auf. Tun sie das nicht, müssen wir Vorkehrungen treffen, die sowohl den Programmieraufwand erhöhen als auch die Effizienz verringern können.

3.3.1 Effizienz und Zuverlässigkeit

Fassen wir typische Eigenschaften einiger Datenstrukturen zusammen:

- *Arrays* und übliche Arrayzugriffe sind bezüglich Laufzeit sowie Speicher- und Energieverbrauch sehr effizient (konstant), solange keine häufige lineare Suche in großen Arrays und kein häufiges Umkopieren nötig ist. Die binäre Suche in sortierten Arrays ist recht effizient (logarithmisch). Es gibt effiziente Sortierverfahren wie Quicksort. Der Programmieraufwand ist meist akzeptabel.

- Einfache *lineare Listen* sind mit sehr wenig Programmieraufwand verbunden. Zugriffe auf den Listenanfang einschließlich Einfügen und Löschen, je nach Implementierung auch auf das Listenende, sind sehr effizient (konstant). Es gibt effiziente Sortierverfahren wie Mergesort. Das Suchen und die meisten anderen Arten von Zugriffen sind ineffizient (linear). Der Speicher- und Energieverbrauch ist wegen des nötigen Umgangs mit Referenzen auch bei effizienten Operationen etwas höher als bei Arrays.

- *Binäre Suchbäume* halten die Einträge ständig sortiert und ermöglichen damit eine in der Regel (aber nicht immer) einigermaßen effiziente Suche (durchschnittlich logarithmisch, linear im schlechtesten Fall). Sortieren ist nicht nötig. Allerdings ist der Programmieraufwand höher als bei Listen, das Einfügen und Löschen ist weniger effizient (logarithmisch bzw. linear), und auch der Speicherverbrauch ist höher als bei Listen.

- *Hash-Tabellen* ermöglichen eine recht effiziente Suche und sind auch beim Einfügen und Löschen effizient, allerdings nur wenn kaum Kollisionen auftreten (konstant). Häufige Kollisionen machen Hash-Tabellen ineffizient (linear). Es gibt keine Reihenfolge auf Einträgen, wodurch auch das Sortieren sinnlos ist. Der Energieverbrauch ist bei wenigen Kollisionen recht niedrig, der Speicherverbrauch größer als bei Listen, weil Hash-Tabellen zur Vermeidung von Kollisionen nicht voll sein dürfen.

Der Speicher- und Energieverbrauch dieser Datenstrukturen ist relativ gering (im Wesentlichen linear) im Vergleich zu Datenstrukturen, die speziell für die parallele Programmierung entwickelt wurden. In EP2 werden wir aber nicht auf parallele Programmierung eingehen.

Obige Überlegungen führen zu diesen Empfehlungen: Wir verwenden

Empfehlungen

- Arrays, wenn nur einfache Arrayzugriffe nötig sind, aber auch für die binäre Suche bei seltenen Änderungen (erneute Sortierung);
- sonst lineare Listen, wenn Zugriffe nur auf den Listenanfang oder das Listenende erfolgen oder die Länge der Liste sehr kurz bleibt;
- sonst Hash-Tabellen, wenn keine Sortierung nötig ist;
- sonst diverse Arten von Bäumen, etwa binäre Suchbäume.

Zufall in Hash-Tabelle

Die Effizienz von Hash-Tabellen hängt stark vom Zufall ab. Wesentliche Operationen auf Hash-Tabellen (Suchen, Einfügen, Entfernen von Einträgen) sind effizient, wenn nur selten Kollisionsbehandlungen nötig sind. Das ist nur gegeben, wenn sich die Hash-Werte fast aller Einträge in den für die Indexberechnung verwendeten Bereichen voneinander unterscheiden. In einer Hash-Tabelle, die nur zu einem kleinen Teil befüllt ist, ist die Wahrscheinlichkeit für Einträge mit gleichen Hash-Werten natürlich niedriger als für eine beinahe volle Tabelle. Wir können den Zufall auf zwei Arten unterstützen: Hash-Funktionen, die gut streuende Hash-Werte berechnen, sowie einen nicht zu hohen maximalen Füllgrad von Tabellen (bis ca. 80%). Diese beiden Maßnahmen ergeben zusammen oft recht effiziente Operationen, aber Ausreißer sind möglich, das heißt, gelegentlich brauchen Zugriffe deutlich mehr Zeit als üblich.

Zufall in Suchbaum

Binäre Suchbäume sind ebenfalls stark datenabhängig. Die Struktur eines Baums hängt davon ab, in welcher Reihenfolge die Einträge eingefügt werden. Wenn beispielsweise Objekte in aufsteigender (oder absteigender) Reihenfolge eingefügt werden, das heißt, jeder neue Eintrag ist größer als der bisher größte Eintrag im Baum (oder kleiner als der bisher kleinste), dann wird stets nur eine der beiden Referenzen jedes Baumknotens verwendet (`left` oder `right`), die andere enthält `null`. Ein solcher Baum hat die Eigenschaften einer ganz einfachen linearen Liste, bei der zum Eintragen und Suchen die gesamte Liste durchlaufen wird. Wir sprechen von einem *zu einer Liste entarteten Baum,* der sehr ineffizient ist. Erfolgen die Einträge in zufälliger Reihenfolge, ist der Baum fast immer sehr gut strukturiert, sodass sowohl die Suche als auch das Eintragen effizient ablaufen. Entartete Bäume treten meist nur dann auf, wenn die Einträge in ihrer Reihenfolge nicht zufällig, sondern auf irgend eine Weise strukturiert sind – muss nicht unbedingt vollständig sortiert sein. Wenn die Daten, die in den Baum eingetragen werden sollen, strukturiert sind, müssen wir die Daten in einer von der Struktur abhängigen Reihenfolge so eintragen, dass ein gut geformter, sogenannter *balancierter* Baum entsteht – siehe Abschnitt 3.3.2.

entarteter Baum

In Abschnitt 3.3.2 werden wir Maßnahmen betrachten, die starke Ausreißer in der Effizienz von Hash-Tabellen und Bäumen vermeiden helfen. Diese Maßnahmen sind jedoch mit einem höheren Programmieraufwand verbunden, und auch die Effizienz kann im Durchschnitt verringert werden. Solange gelegentliche Ausreißer kein großes Problem darstellen, ist es durchaus vertretbar, dass wir uns auf den Zufall verlassen. Das macht Programme meist einfacher und effizienter. Sollten sich daraus in der Praxis zu häufig zu lange Wartezeiten ergeben, kann man noch immer nachbessern und Maßnahmen gegen Ausreißer einbauen.

bewusst schlechte Datenverteilung

Bei in der Praxis fast immer auftretenden zufälligen Datenverteilungen wird die gefühlte Effizienz meist recht gut sein. Die Gefahr, dass sich eine sehr schlechte Datenverteilung zufällig ergibt, ist vernachlässigbar. Aber eine sehr schlechte Datenverteilung kann in Ausnahmesituationen systematisch entstehen oder absichtlich herbeigeführt werden. Ein Beispiel ist ein Bestellsystem für Ersatzteile, wo nach einer Rückrufaktion in kurzer Zeit sehr viele Bestellungen für das gleiche Ersatzteil eintreffen – klingt unwahrscheinlich, hat in der Praxis aber schon zu Problemen geführt. Offensichtlicher ergeben sich Probleme bei einer *Denial-of-Service-Attack,* bei der ein System mit so vielen Anfragen gefüttert wird, dass es nicht mehr verwendbar ist; bei Kenntnis von Implementierungsdetails und gezielter Auswahl schlecht verteilter Daten ist dafür nur ein geringer Aufwand nötig. Programme, bei denen solche Gefahren bestehen, sollen stets gegen sehr starke Effizienzeinbrüche bei schlecht verteilten Daten gesichert sein. Die Vermeidung von Ausreißern ist in diesen Fällen aus Gründen der Sicherheit notwendig.

Sortieren und Zufall

Nicht nur Datenstrukturen und ihre elementaren Zugriffsoperationen, sondern auch sehr viele andere Algorithmen hängen vom Zufall ab. Beispielsweise ist das Sortierverfahren *Quicksort,* das vor allem auf Arrays häufig eingesetzt wird, zwar im Durchschnitt und in den meisten Fällen sehr effizient ($n \cdot \log n$ mit einem kleinen konstanten Faktor), kann im schlechtesten Fall (n^2) aber genau so schlecht sein wie *Bubblesort* im schlechtesten Fall. Um den schlechtesten Fall so gut wie möglich zu vermeiden, wird mit der Sortierung (Wahl des Pivot-Elements) meist nicht von vorne begonnen, sondern irgendwo in der Mitte. Auf Listen wird Quicksort kaum eingesetzt, weil man nicht effizient auf Einträge in der Mitte der Liste zugreifen kann und daher die Wahrscheinlichkeit für den schlechtesten Fall kaum verringerbar ist. *Mergesort* ist dagegen auch im schlechtesten Fall effizient ($n \cdot \log n$, mit einem etwas größeren konstanten Faktor als Quicksort). Allerdings ist Mergesort auch im durchschnittlichen und besten Fall nicht besser, weil Listen unabhängig

vom Inhalt immer auf die gleiche Weise geteilt werden. Das einfache, im Durchschnitt und im schlechtesten Fall ineffiziente Bubblesort (n^2), kann dagegen im besten Fall (n wenn Daten bereits sortiert) effizienter sein als Quicksort und Mergesort. Wenn wir die Daten genau kennen und z. B. wissen, dass die Datenmenge sehr klein ist oder die Daten bis auf wenige Vertauschungen schon fast sortiert sind, kann auch Bubblesort als effizientes Sortierverfahren angesehen werden. Wenn wir die Daten nicht kennen, werden wir auf Arrays eher Quicksort und auf Listen eher Mergesort verwenden. Ist die Gefahr von Ausreißern besonders groß, kann man auch auf Arrays Mergesort einsetzen. Je nach den Daten und Verwendungen können viele weitere Sortierverfahren sinnvoll sein. Beispielsweise ist der sortierte Iterator aus Listing 3.35 im Allgemeinen sehr ineffizient, weil bei der Suche nach dem nächsten Element immer die ganze Liste durchlaufen wird. In einem Spezialfall, in dem nur auf wenige Einträge am Anfang einer sortierten Liste zugegriffen wird, kann dieses Verfahren dennoch recht effizient sein; vollständiges Sortieren ist in diesem Fall nicht nötig.

Aufgabe 3.35 Beschreiben Sie, worauf Sie bei der Auswahl geeigneter Datenstrukturen und abstrakter Datentypen achten, wenn Sie ein Warenlager, ein Telefonbuch, eine vor dem Eingang zu einem Veranstaltungsort wartende Menschenmenge, eine Schreibtischlade, einen Aktenordner, etc. in Software simulieren sollen.

Aufgabe 3.36 Beschreiben Sie, wodurch starke Ausreißer in der Effizienz von Hash-Tabellen und Bäumen zustandekommen können, wie sie vermeidbar sind, und warum mögliche starke Ausreißer in der Effizienz meist zu vermeiden sind, auch wenn sie nur sehr selten auftreten.

3.3.2 Zufall versus Garantie

Zur Vermeidung entarteter Bäume gibt es verschiedene Arten *balancierter Bäume.* Wenn ein zu starkes Ungleichgewicht zwischen dem linken und rechten Teilbaum entsteht, werden einfach Knoten vom einen in den anderen Teilbaum verschoben, bis das Ungleichgewicht beseitigt ist. Das führt zwar zu einem Mehraufwand beim Einfügen und Löschen sowie zu einem etwas höheren Speicherverbrauch zur Verwaltung der Informationen bezüglich des Gleichgewichts, aber Effizienzprobleme durch das Entarten werden vermieden.

AVL-Baum

Ein Beispiel dafür ist der *AVL-Baum,* bei dem sich die Höhen der beiden Teilbäume unter jedem Knoten um höchstens 1 unterscheiden.

Listing 3.42 Einfügen in AVL-Baum als Variante von Listing 3.32 (Ausschnitt)

```
1  public class OTNode {
2      private String elem;
3      private OTNode left, right;
4      private int balance = 0;
5
6      public OTNode add(String e) {
7          if (compare(e) > 0) {
8              if (right == null) {
9                  right = new OTNode(e);
10                 ++balance;
11             } else {
12                 OTNode n = right.add(e);
13                 if (n != null) {
14                     right = n;
15                     return this;
16                 }
17                 if (++balance > 1) {
18                     if (right.balance > 0) {
19                         n = right;
20                         balance = 0;
21                     } else {
22                         n = right.left;
23                         balance = right.balance = 0;
24                         if (n.balance > 0) {
25                             balance = -1;
26                         } else if (n.balance < 0) {
27                             right.balance = 1;
28                         }
29                         right.left = n.right;
30                         n.right = right;
31                     }
32                     n.balance = 0;
33                     right = n.left;
34                     n.left = this;
35                     return n;
36                 }
37             }
38         } else {  // Symmetrischer Zweig für linken Teilbaum
39         }
40         return balance != 0 ? null : this;
41     }
42 }
```

Wie in Listing 3.42 gezeigt, enthält jeder Knoten eine zusätzliche Variable `balance`, die den Höhenunterschied zwischen linkem und rechtem

Teilbaum enthält. Nur die Werte 0, 1 und −1 sollten hier vorkommen. Tritt nach dem Einfügen oder Löschen doch kurzfristig ein größerer Unterschied auf, muss der Baum durch *Rotieren* umstrukturiert werden. Rotieren bedeutet, dass je nach Situation zwei oder drei miteinander verkettete Knoten (mit den darunter hängenden Teilbäumen) so gegeneinander vertauscht werden, dass die Höhenunterschiede danach ausgeglichen sind, wobei die Sortierung auf den Einträgen erhalten bleibt.

Hinsichtlich der Laufzeiteffizienz ist der Zusatzaufwand für das Rotieren nicht besonders groß. Aber aus Listing 3.42 ist leicht zu erkennen, dass der Entwicklungs- und Wartungsaufwand im Vergleich zum einfachen Einfügen wegen der vielen nötigen Fallunterscheidungen doch erheblich größer ist. In Java besteht eine zusätzliche Schwierigkeit darin, dass es keine Zeiger auf Objektvariablen wie in C gibt (wodurch der Elternknoten leicht geändert werden könnte) und jede Methode höchstens einen Ergebniswert zurückgeben kann (weil wir den Knoten, der `this` ersetzen soll, ebenso wie Informationen über die neue Höhe des Teilbaums zurückgeben möchten). Einer dieser Sprachmechanismen würde reichen um die Implementierung deutlich zu vereinfachen. In Listing 3.42 behelfen wir uns damit, dass `add` folgende Ergebnisse zurück gibt:

Java

- `null`, wenn Teilbaum gewachsen und `this` unverändert,
- Knoten, der `this` ersetzt (bzw. `this` für unveränderten Knoten), wenn Höhe unverändert.

Das funktioniert für `add`, weil diese Fälle nicht gleichzeitig auftreten können. Im Allgemeinen, etwa beim Löschen, sind aufwendigere Lösungen nötig, z. B. wie `cnt` in Listing 3.31 zum Zählen der Einträge. Aufrufer müssen den Knoten entsprechend ersetzen:

Listing 3.43 Variante von Listing 3.31 als AVL-Baum

```
1  public class OrderedTree implements Ordered {
2      ...
3      public void add(String e) {
4          cnt[0]++;
5          if (root != null) {
6              OTNode n = root.add(e);
7              if (n != null) { root = n; }
8          } else { root = new OTNode(e); }
9      }
10     public int removeAll(String e) { ... };
11 }
```


Bei Hash-Tabellen ist die Qualität der Hash-Funktionen entscheidend. Wenn deren Qualität nicht verbessert werden kann, lassen sich durch andere Formen der Kollisionsbehandlung starke Ausreißer vermeiden. Ein Lösungsansatz besteht darin, eine Hash-Tabelle mit balancierten Bäumen zu kombinieren: Das Array der Hash-Tabelle enthält Wurzeln von Bäumen, in denen alle Einträge enthalten sind, für die über die Hash-Funktion der gleiche Array-Index ermittelt wurde. Damit können beliebig viele Einträge am gleichen Index liegen. Statt der Kollisionsbehandlung durch Finden eines neuen Indexes haben wir den Zusatzaufwand durch das Einfügen in einen Baum. Der größte Vorteil dieses Lösungsansatzes liegt darin, dass eine schlecht streuende Hash-Funktion die Laufzeit-Effizienz zwar auf die Effizienz eines Baums (oder knapp darunter) reduziert, aber nicht dramatisch abstürzen lässt. Bei gut streuenden Hash-Funktionen liegt die Effizienz knapp unter der von einfachen Hash-Tabellen. Zusätzlich ist die Effizienz nicht mehr so stark vom Füllgrad der Tabelle abhängig. Manchmal ist die Anzahl der Einträge sogar viel größer als das Array. Durch die Wahl der Größe des Arrays können wir die charakteristische Effizienz beliebig zwischen der einer Hash-Tabelle und der eines Baums ansiedeln. Ein Nachteil ist jedoch der hohe Implementierungsaufwand und Speicherverbrauch. Es werden nicht nur die Vorteile von Hash-Tabellen und balancierten Bäumen miteinander kombiniert, sondern auch deren Nachteile.

Häufig enthalten Arrays in Hash-Tabellen lineare Listen von Einträgen. Listen sind einfacher handzuhaben als Bäume. Die Kollisionsbehandlung erfolgt durch Eintragen in eine Liste. Allerdings ist die Suche in einer langen Liste sehr aufwendig, sodass gut streuende Hash-Funktionen und ein nicht zu großer Füllgrad der Tabelle wesentlich sind. Ein Schutz gegen schlechte Datenverteilung besteht nicht. Die Anzahl der Einträge kann etwas größer sein als das Array.

Weil Zugriffe auf kurze Listen etwas effizienter sind als auf kleine Bäume, wird manchmal ein Mittelweg gewählt: Fallen nur wenige Einträge auf einen Index, enthält das Array an diesem Index eine lineare Liste, sonst einen Baum. Bei einer bestimmten Länge wird die Liste in einen Baum umgewandelt. Der Typ der Arrayeinträge ist z. B. ein Interface, das sowohl von der Liste als auch vom Baum implementiert wird. Über dynamisches Binden wird entschieden, ob auf eine Liste oder einen Baum zugegriffen wird.

Die Sicherheit lässt sich auch durch mehrere Hash-Funktionen erhöhen: Bei einer Kollision wird auf eine andere Hash-Funktion (oder einen anderen Bit-Muster-Bereich) mit anderer Streuung umgeschaltet.

Aufgabe 3.37 Vervollständigen Sie die Methode `add` aus Listing 3.42 und testen Sie sie. Beschreiben Sie, wodurch sich die Implementierung des linken Teilbaums von der des rechten unterscheidet.

schwierig

Aufgabe 3.38 Implementieren Sie die Methode `removeAll` des AVL-Baums aus Listing 3.43 entsprechend einer Beschreibung von AVL-Bäumen aus einem Buch oder dem Internet, wobei die Struktur von `OTNode` nicht wesentlich geändert werden soll. Beschreiben Sie, welche Schwierigkeiten dabei aufgetreten sind und wie Sie sie gelöst haben.

aufwendig

Aufgabe 3.39 Verwenden Sie einen doppelt verketteten Baum wie aus Aufgabe 3.27 als Basis für die Implementierung eines AVL-Baums (nach einer Beschreibung aus einem Buch oder einer Webseite). Beschreiben Sie, warum dies etwas einfacher ist als eine Implementierung auf Basis eines gewöhnlichen einfach verketteten Baums.

3.3.3 Angebot von Java

Java

Die am häufigsten verwendeten vordefinierten abstrakten Datentypen in Java gehören zum sogenannten *Java-Collections-Framework* und sind unter `java.util` zu finden. Neben anderen stehen folgende Interfaces für unterschiedliche Einsatzgebiete bereit:

`Collection<E>` ist der Typ von Datensammlungen, wobei `E` für den Typ der Einträge steht. Er kommt überall zum Einsatz, wo irgendeine nicht näher bestimmte iterierbare Sammlung von Daten benötigt wird. Die Methoden im Interface sind zwar zahlreich, aber nicht spezifisch für bestimmte Aufgaben. Einige Methoden wie `add`, `remove` und `clear` sind als „optional“ gekennzeichnet, das heißt, Implementierungen dürfen einen Laufzeitfehler melden statt diese Methoden wie beschrieben auszuführen. Daher ist bei der Verwendung Vorsicht angebracht. Dieses Interface ähnelt dem aus Listing 3.24, wobei jedoch der Typ der Einträge frei wählbar ist und die Methoden `add`, `contains` und `remove` (entspricht `removeAll`) Ergebnisse vom Typ `boolean` zurückgeben.

`Set<E>` enthält die gleichen Methoden wie `Collection<E>`, wobei jedoch in keinem Objekt vom Typ `Set<E>` gleiche Einträge mehrfach vorkommen dürfen. Es handelt sich also (beinahe) um eine Menge im mathematischen Sinn.

`SortedSet<E>` erweitert `Set<E>`, wobei alle Einträge vollständig sortiert sind. Um die Sortierbarkeit zu gewährleisten, dürfen für `E` nur Untertypen des Interfaces `Comparable<E>` verwendet werden, das eine Methode `compareTo` beschreibt, die wir für Vergleiche von Zeichenketten verwendet haben. Das Einsatzgebiet entspricht im Wesentlichen dem von `Ordered` aus Listing 3.25. Einträge dürfen nicht verändert werden, damit die Sortierung nicht durch Zustandsänderungen zerstört wird.

`List<E>` erweitert `Collection<E>`, wobei das gleiche Objekt mehrfach eingetragen sein darf. Wie in Listing 3.26 werden Zugriffe über einen Index unterstützt. Da die Implementierung dieser Zugriffe häufig ineffizient ist, wird `List<E>` eher selten verwendet.

`Queue<E>` erweitert `Collection<E>` um Methoden einer Queue, die in doppelter Ausführung vorhanden sind: `offer`, `poll` und `peek` geben spezielle Werte wie `false` und `null` zurück, wenn die Operationen nicht ausgeführt werden können, während die fast gleichbedeutenden Methoden `add`, `remove` und `element` in diesen Fällen zu einem Laufzeitfehler (Exception) führen.

`Deque<E>` erweitert `Queue<E>` um übliche Methoden einer Double-Ended-Queue sowie eines Stacks. Auch hier gibt es die Methoden in zwei Varianten, wenn nicht ausführbar mit speziellem Rückgabewert (z. B. `offerFirst` und `offerLast`) oder mit Laufzeitfehler (z. B. `addFirst` und `addLast`). Zusätzlich gibt es diese Methoden auch mit den Namen von Queue- und Stack-Methoden.

`Map<K,V>` beschreibt assoziative Datenstrukturen, wobei `K` der Typ der Schlüssel und `V` jener der assoziierten Werte ist. Wesentliche Methoden entsprechen denen von Listing 3.23, sind aber zahlreicher. Schlüssel dürfen nicht verändert werden, da Einträge nach Zustandsänderungen sonst nicht mehr auffindbar wären.

`SortedMap<K,V>` erweitert `Map<K,V>`, wobei alle Einträge vollständig sortiert sind. Schlüssel werden mittels `compareTo` verglichen.

Eine Reihe weiterer Interfaces werden in der nebenläufigen (parallelen) Programmierung und für spezielle Aufgaben eingesetzt.

Zahlreiche Implementierungen dieser Interfaces stehen zur Verfügung. Die meisten Implementierungen sind recht aufwendig und darauf ausgelegt, dass schlechte Hash-Funktionen oder Datenverteilungen nicht zu allzugroßen Effizienzeinbrüchen führen, aber dennoch auch im

Durchschnitt möglichst gute Effizienz erreicht wird. Implementierungsdetails dieser abstrakten Datentypen sind in der Regel nicht beschrieben. Aber der Quellcode kann betrachtet werden. Allerdings müssen wir davon ausgehen, dass unspezifizierte Details ständigen Änderungen (zwecks Verbesserung) unterliegen. Folgende Implementierungen werden häufig verwendet:

`HashSet<E>` ist eine Implementierung von `Set<E>` als Hash-Tabelle. Im Konstruktor kann die Anfangsgröße der Tabelle und der Füllgrad, ab dem die Tabelle vergrößert wird, angegeben werden. Meist wird darauf verzichtet, da die Default-Werte gut gewählt sind. Da es sich um eine Hash-Tabelle handelt, geben Iteratoren die Einträge in unvorhersehbarer Reihenfolge zurück. Eine Variante namens `LinkedHashSet<E>` verkettet alle Einträge zusätzlich in einer linearen Liste, sodass Iteratoren die Einträge in der Reihenfolge des Eintragens zurückgeben.

`HashMap<K,V>` ist eine Implementierung von `Map<K,V>` als eine Hash-Tabelle. Abgesehen davon gilt das gleiche wie für `HashSet<E>` mit der Variante `LinkedHashMap<K,V>`.

`ArrayList<E>` ist eine Implementierung von `List<E>` als Array, dessen Größe automatisch an die Verwendung (größter Index) angepasst wird. Übliche Arrayzugriffe erfolgen daher sehr effizient, während fast alle anderen Operationen eher ineffizient sind. Objekte von `ArrayList<E>` werden daher häufig anstelle normaler Arrays verwendet, wenn die benötigte Größe im Vorhinein unbekannt ist. Sie werden nur selten wie Listen eingesetzt.

`ArrayDeque<E>` ist eine effiziente Implementierung von `Deque<E>` als Array mit automatischer Anpassung der Größe. Da die Effizienz im Mittelpunkt steht, ist die Anwendbarkeit beschränkt. So ist `null` als Eintrag verboten.

`LinkedList<E>` ist eine Implementierung von `List<E>` und `Deque<E>` als doppelt verkettete Liste. Der Eintrag von `null` ist erlaubt.

`TreeSet<E>` ist eine Implementierung von `SortedSet<E>` als balancierter binärer Suchbaum (nicht AVL-Baum). Im Konstruktor kann bestimmt werden, auf welche Weise die Sortierung erfolgt.

`TreeMap<K,V>` ist eine Implementierung von `Map<K,V>` als balancierter Baum, der auf derselben Basis wie `TreeSet<E>` beruht.

`Arrays` beinhaltet häufig verwendete statische Methoden auf Arrays. Diese Klasse gehört zwar nicht zum Java-Collections-Framework, sollte aber dennoch als Alternative in Betracht gezogen werden, wenn Arrays als Datensammlungen verwendbar erscheinen.

Beim Einsatz vordefinierter abstrakter Datentypen sollten wir stets mit der Auswahl eines geeigneten Interfaces beginnen. Kommen aufgrund der benötigten Funktionalität mehrere Interfaces in Betracht, wählen wir in der Regel das einfachere, also eher den Ober- als den Untertyp. Danach entscheiden wir uns für eine Implementierung, wobei die in Abschnitt 3.3.1 genannten Kriterien zu berücksichtigen sind. Die Klasse verwenden wir üblicherweise nur dort, wo ein Objekt dieser Klasse erzeugt wird. An allen anderen Stellen, das sind vor allem die Variablendeklarationen, verwenden wir vorwiegend die entsprechenden Interfaces als Typen. Auf diese Weise können wir ganz einfach an nur einer Stelle, dort wo das Objekt erzeugt wird, eine Implementierung des Interfaces gegen eine andere Implementierung desselben Interfaces austauschen, falls dies notwendig wird. Die Methoden sind so weit wie möglich über alle Interfaces vereinheitlicht und haben sprechende Namen, sodass wir uns ihre Verwendungsweisen leicht merken können.

Aufgabe 3.40 Fassen Sie die Methoden in den Interfaces und Klassen des Java-Collections-Frameworks zusammen und beschreiben Sie Unterschiede zwischen Methoden vergleichbarer Funktionalität sowie wahrscheinliche Ursachen für diese Unterschiede.

Aufgabe 3.41 Betrachten Sie die Implementierungen von `HashMap`, `TreeMap` und `LinkedList` einer aktuellen JDK-Version in der IDE und versuchen Sie herauszufinden, welche Algorithmen und Datenstrukturen dahinterstecken und welche Implementierungsdetails gewählt wurden. Beschreiben Sie, was Sie herausgefunden haben.

4 Qualität in der Programmierung

Der Begriff *Qualität* hat viele unterschiedliche Facetten. Entsprechend breit sind die Themenbereiche, um die es in diesem Kapitel geht.

In Abschnitt 4.1 gehen wir davon aus, dass jedes Programm zur Laufzeit in einen Zustand kommen kann, in dem es nicht mehr so wie erwartet weitergeht. Ohne Vorkehrungen wird das Programm in diesem Fall mit einem Laufzeitfehler beendet. Wir können die Programmqualität verbessern, indem wir problematische Zustände im Programm selbst behandeln (etwa durch Ausnahmebehandlung) und dadurch Laufzeitfehler nicht nach außen sichtbar werden lassen. Die Ein- und Ausgabe von Daten betrachten wir genauer, weil in diesem Bereich häufig unvorhersehbare Zustände eintreten. Generell müssen wir überprüfen, ob Daten, die von außerhalb des Programms kommen (Eingabedaten im weitesten Sinn) alle erwarteten Eigenschaften erfüllen.

In Abschnitt 4.2 geht es dagegen um die statische Betrachtung von Programmtexten. Falsch verstandene Programmtexte führen bei der Wartung mit hoher Wahrscheinlichkeit zu schweren Fehlern. Wir lernen Techniken und Hilfsmittel zum Verstehen von Programmen kennen. Dazu gehört eine systematische Dokumentation genauso wie die Überprüfung durch Code-Reviews.

In Abschnitt 4.3 betrachten wir das Testen von Programmen und einiges, was damit zusammenhängt, sowie Ansätze zur Qualitätssteigerung. Das Verbessern von Programmtexten birgt in sich ein Gefahrenpotential, das wir kennen und mit dem wir umgehen müssen.

4.1 Umgang mit Fehlern zur Laufzeit

Java

Wir betrachten den Umgang mit Laufzeitfehlern in Java. Als Anwendungsbeispiel sehen wir uns die Ein- und Ausgabe über Streams an und stellen unterschiedliche Streams in Java einander gegenüber. Im Zusammenhang mit der Überprüfung von Eingabedaten betrachten wir ein Beispiel mit Pattern-Matching.

4.1.1 Ausnahmebehandlung in Java

Java

Ausnahme werfen

propagieren

Alle Laufzeitfehler werden in Java auf die gleiche Weise gehandhabt: Tritt eine Situation ein, in der das Programm nicht mehr normal weiterlaufen kann, wird der Programmfluss unterbrochen und eine *Ausnahme* (*Exception*) ausgelöst bzw. *geworfen*. Dabei wird Schritt für Schritt von jeder aufgerufenen Methode zurückgekehrt und der entsprechende Stack-Frame abgebaut (das heißt, die Ausnahme wird an den Aufrufer der Methode *propagiert*) bis auch eine Rückkehr aus dem allerersten Aufruf von `main` erfolgt ist. Dort wird schließlich das Programm beendet nachdem in einer Fehlermeldung neben der Art der Ausnahme ein *Stack-Trace* ausgegeben wurde; er enthält die Namen aller zur Zeit des Werfens ausgeführten Methoden (einer pro Stack-Frame) und die Programmstellen, an denen die Methodenaufrufe zu finden sind. Wie wir wissen, liefert ein Stack-Trace Hinweise zum Auffinden von Fehlerursachen im Programm. Abhängig vom Grund, aus dem das Programm nicht normal weiterlaufen kann, werden zahlreiche Arten von Ausnahmen unterschieden, und jede dieser Arten wird durch eine Klasse – den *Typ* der Ausnahme – dargestellt. Typische Namen solcher Klassen sind `NullPointerException` und `StackOverflowError`.

abfangen

Java bietet eine Möglichkeit, in den Prozess des Propagierens von Ausnahmen einzugreifen und die (weitere) Rückkehr aus Methodenaufrufen zu unterbinden. Wir sprechen dabei vom *Abfangen* der Ausnahme bzw. von *Ausnahmebehandlung* (*Exception-Handling*). Dies geschieht in einem `try-catch`-Block:

Listing 4.1 Abfangen einer Ausnahme – schlechter Programmierstil

```
1  public class HandledException {
2      private static void printLen(String s) {
3          System.out.print(s.length());
4      }
5      public static void main(String[] args) {
6          try {
7              printLen(null);
8              System.out.print("nicht ausgeführt");
9          }
10         catch(NullPointerException e) {
11             System.out.print("abgefangen ");
12         }
13         System.out.print("weiter");
14     }
15 }
```


Nach Aufruf von `main` wird der `try`-Block (Block vor dem `try` steht) ausgeführt und `printLen` aufgerufen. In `printLen` wird eine Ausnahme vom Typ `NullPointerException` geworfen, weil `s` den Wert `null` hat, wodurch `s.length()` nicht ausführbar ist. Diese Ausnahme wird an den Aufrufer `main` propagiert. Da der Aufruf in einem `try`-Block erfolgt ist, wird die Ausnahme nicht gleich weiterpropagiert, sondern nach einem `catch`-Block eines passenden Typs gesucht. Ein solcher steht gleich nach dem `try`-Block. Er wird als Ersatz für den durch die Ausnahme abgebrochenen `try`-Block ausgeführt. Danach ist die Ausnahme behandelt. Die Programmausführung wird nach dem `try-catch`-Block fortgesetzt. Es wird `"abgefangen weiter"` ausgegeben.

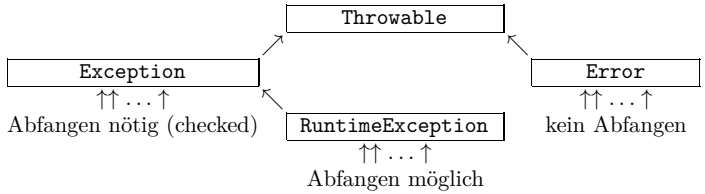
Nach einem `try`-Block können beliebig viele `catch`-Blöcke stehen. Tritt während der Ausführung des `try`-Blocks eine Ausnahme auf, werden die `catch`-Blöcke in gegebener Reihenfolge betrachtet. Der erste `catch`-Block, dessen Typ ein Obertyp des Typs der Ausnahme ist, wird ausgeführt, und die Ausnahme ist danach behandelt. Hat kein `catch`-Block einen passenden Typ, wird die Ausnahme weiter propagiert. Der Typ im `catch`-Block ist wie ein formaler Parameter angeschrieben, und der entsprechende Parameter (z. B. `e` in Listing 4.1) ist auch so verwendbar. Der Parameter steht für ein Objekt des Typs der Ausnahme.

Der Programmablauf in Listing 4.1 ist vorhersehbar. In solchen Fällen sollte man Ausnahmen vermeiden und übliche Konzepte wie bedingte Anweisungen einsetzen. Dagegen eignen sich Ausnahmebehandlungen ausgezeichnet um auf außergewöhnliche Ereignisse zu reagieren:

Listing 4.2 Abfangen von Ausnahmen zwecks freundlicherer Fehlermeldung

```
1  public static void main(String[] args) {
2      try { ... }
3      catch(Exception e) {
4          // store data and status information
5          System.out.println("Entschuldigung");
6      }
7  }
```

Folgende Untertypbeziehungen existieren auf Typen von Ausnahmen:



Alle Typen von Ausnahmen sind von der Klasse `Throwable` mit den beiden Untertypen `Exception` und `Error` abgeleitet. Von `Error` sind nur schwerwiegende Ausnahmen abgeleitet, nach denen eine weitere Programmausführung sinnlos erscheint, z. B. `OutOfMemoryError`. Alle anderen Typen sind von `Exception` abgeleitet. Daher fängt Listing 4.2 alle Ausnahmen ab, die einen sauberen Programmabschluss erlauben.

Die Typen aller vom Java-Laufzeitsystem geworfenen Ausnahmen sind Untertypen von `Error` oder `RuntimeException`, einem Untertyp von `Exception`. Das sind *nicht überprüfte* (*unchecked*) Ausnahmen. Wir müssen stets mit ihrem Auftreten rechnen. Alle anderen Ausnahmen (von `Exception`, aber nicht von `RuntimeException` abgeleitet) sind *überprüft* (*checked*). Überprüfte Ausnahmen sind in ihrer Verwendung eingeschränkt, sodass sie nicht beliebig weit propagiert werden können. Der Java-Compiler überprüft, dass jede Anweisung, bei deren Ausführung eine überprüfte Ausnahme auftreten könnte,

unchecked

checked

- in einem `try`-Block vorkommt und ein passender `catch`-Block existiert um diese Ausnahme abzufangen,
- oder in einer Methode vorkommt, die eine `throws`-Klausel für diese Ausnahme enthält (wodurch solche Überprüfungen auch bei jedem Aufruf dieser Methode durchgeführt werden).

throws

Wie wir in Abschnitt 4.1.2 sehen werden, kann beim Lesen einer Zeile aus einer Datei durch `readLine` eine überprüfte Ausnahme vom Typ `IOException` auftreten. Diese Methode ist folgendermaßen definiert:

```
public String readLine() throws IOException { ... }
```

Diese Methode dürfen wir nur innerhalb eines `try`-Blocks aufrufen, wenn es einen `catch`-Block für `IOException` gibt oder die Methode, in der der Aufruf erfolgt, selbst mit `throws IOException` definiert ist.

throw

Mit der Anweisung `throw x;` können wir das Werfen der Ausnahme `x` veranlassen, wobei `x` ein Objekt der Klasse `Throwable` ist:

Listing 4.3 Bei unzweckmäßigem Argument Ausnahme zum Aufrufer propagiert

```
1 public String get(int i) {
2     if (i < 0) throw new IllegalArgumentException();
3     ...   // following statements unreachable if i < 0
4 }
```

Für Ausnahmebehandlungen spielt es keine Rolle, ob eine Ausnahme vom Laufzeitsystem oder durch eine `throw`-Anweisung geworfen wurde.

Wir können auch eigene Arten von Ausnahmen definieren. Es reicht, eine neue Klasse zu schreiben, die Unterklasse eines bestehenden Ausnahmetyps ist. Meist werden eigene Ausnahmetypen von `Exception` abgeleitet, wodurch es sich um überprüfte Ausnahmen handelt.

Methoden in Ausnahmen sind wie üblich verwendbar:

Listing 4.4 Initialisieren und verwenden der Objektvariablen in Ausnahmen

```
1  public void exceptionThrowingMethod(int i) {
2      if (i < 0) {
3          String s = "i >= 0 required, " + i + " found";
4          throw new IllegalArgumentException(s);
5      }
6  }
7  public void exceptionCatchingMethod() {
8      try { exceptionThrowingMethod(-3); }
9      catch(Exception e) {
10         System.out.println(e.getMessage());
11         System.out.println(e);
12         e.printStackTrace();
13         throw new RuntimeException(e);
14     }
15 }
```

Eine Zeichenkette als Argument eines Konstruktors einer Ausnahme stellt in der Regel eine „Message“ dar, die in einer Objektvariable der Ausnahme abgelegt wird und durch `getMessage()` ausgelesen werden kann. Hier wird üblicherweise der Fehler beschrieben. Auch das Ergebnis von `toString()` in einer Ausnahme enthält die „Message“ falls ungleich `null`. Mittels `printStackTrace()` lässt sich der Stack-Trace in der Form ausgeben, die auch bei einem Programmabbruch erscheint, obwohl die Ausnahme abgefangen wurde. In Zeile 13 von Listing 4.4 wird im `catch`-Block neuerlich eine Ausnahme geworfen. Die ursprüngliche Ausnahme als Argument des Konstruktors bewirkt, dass die neue Ausnahme die ursprüngliche als „Cause“ beinhaltet. Der Stack-Trace davon enthält neben Informationen der neuen auch jene der ursprünglichen Ausnahme. Wir könnten die gerade abgefangene Ausnahme durch `throw e;` neuerlich werfen.

Message

Stack-Trace

Cause

Nach dem Werfen einer Ausnahme muss der Programmzustand *aufgeräumt* werden. Dabei hilft ein `finally`-Block, der auf den `try`-Block und die `catch`-Blöcke folgen kann: `finally {...}`. Anweisungen darin werden immer ausgeführt – ohne Ausnahme nach dem `try`-Block, bei abgefangener Ausnahme nach dem `catch`-Block, aber auch vor dem Propagieren einer nicht abgefangenen Ausnahme.

Aufräumen

Aufgabe 4.1 Beschreiben Sie die Konzepte für das Werfen und Abfangen von Ausnahmen in Java. Beschreiben Sie auch, wie und wozu diese Konzepte eingesetzt werden.

Aufgabe 4.2 Beschreiben Sie die unterschiedlichen Arten von Ausnahmen in Java sowie Gründe für die Unterscheidung und Auswirkungen auf die Verwendung dieser Ausnahmen.

Aufgabe 4.3 Beschreiben Sie, in welchen Fällen und wozu `throws`-Klauseln in Methoden-Köpfen nötig sind.

Aufgabe 4.4 Beschreiben Sie, wie nach Auftreten einer Ausnahme nach einem passenden `catch`-Block gesucht wird, welcher `catch`-Block schließlich ausgeführt wird (falls mehrere in Frage kommen) und wann welche `finally`-Blöcke ausgeführt werden.

Aufgabe 4.5 Beschreiben Sie, in welchen Fällen beim Werfen einer Ausnahme ein neues Objekt erzeugt werden muss und in welchen Fällen ein bereits existierendes Objekt verwendet werden kann.

Aufgabe 4.6 Beschreiben Sie anhand der Dokumentation der Klasse `Exception`, wozu die einzelnen Konstruktoren und Methoden in dieser Klasse verwendet werden können.

4.1.2 Ein- und Ausgabe über Streams

Java
Datenstrom

Die Ein- und Ausgabe beruht in Java auf *Streams*. Diese Datenströme gehen nur in eine Richtung – Lesen oder Schreiben. Solche Streams werden meist vom Betriebssystem verwaltet. Die Programmiersprache bietet Unterstützung für Aufrufe der Operationen des Betriebssystems sowie zusätzliche Funktionalität zur einfacheren Verwendung der Betriebssystemressourcen. In Java werden zahlreiche Arten von Streams[1] unterschieden. Das sind einige Unterscheidungsmerkmale:

gepuffert vs. ungepuffert

- Zugriffe auf *ungepufferte Streams* werden direkt an das Betriebssystem weitergeleitet, solche auf *gepufferte Streams* modifizieren Java-interne Puffer beschränkter Größe. Nur wenn bearbeitete Daten nicht im Puffer liegen oder keinen Platz haben, erfolgen teure Zugriffe auf das Betriebssystem. Gepufferte Streams sind meist effizienter als ungepufferte weil seltener auf das Betriebssystem zugegriffen wird. Betriebssysteme und ungepufferte Streams bieten meist nur minimale Funktionalität, gepufferte Streams erlauben einen größeren Funktionsumfang. Allerdings können sich bei gepufferten Streams Synchronisationsprobleme ergeben, wenn mehrere Streams die gleiche Datei bearbeiten. Bei ungepufferten Streams sorgt das Betriebssystem selbst für die Synchronisation.

Zeichen vs. Bytes

- Streams enthalten *Zeichen* oder *rohe Daten* (Bytes), im Betriebssystem immer nur rohe Daten. Bei Verwendung von Zeichen-Streams werden vom Betriebssystem erhaltene Daten automatisch in Zeichen (im UTF16-Format) konvertiert und Zeichen vor der Übergabe an das Betriebssystem in rohe Daten umgewandelt. Rohe Daten haben kein bestimmtes Darstellungsformat. Um die Umwandlung der Daten zu ermöglichen, kann bei der Erzeugung eines Zeichen-Streams angegeben werden, wie die rohen Daten zu interpretieren sind, z. B. als UTF-8, Latin1, ASCII oder nach zahlreichen weiteren Kodierungen.

In Java arbeiten Untertypen von `InputStream` und `OutputStream` mit rohen Daten. Darunter ist z. B. `FileInputStream` für die ungepufferte Eingabe von einer Datei und `BufferedInputStream` für die gepufferte Eingabe roher Daten (von wo auch immer). Dagegen arbeiten Unterklassen von `Reader` und `Writer` mit Zeichen. Z. B. lesen `InputStreamReader` (Setzen der Kodierung möglich) und `FileReader` (Datei wählbar) ungepufferte Zeichen. Für das gepufferte Lesen von Zeichen wird `BufferedReader` verwendet.

Streams werden stets so verwendet: Bei der Objekterzeugung (Ausführen des Konstruktors) wird der Stream geöffnet, danach über unterschiedliche Methoden davon gelesen oder darauf geschrieben. Nach Verwendung muss der Stream unbedingt durch Ausführung von `close()` geschlossen werden um gebundene Betriebssystemressourcen freizugeben. Treten beim Öffnen, Lesen, Schreiben oder Schließen Probleme auf, wird eine überprüfte Ausnahme vom Typ `IOException` geworfen.

close

IOException

Java

Mögliche Ausnahmen und die Notwendigkeit des Schließens machen den Umgang mit Streams ungewohnt komplex. Jedoch ist die Vorgehensweise immer die gleiche – siehe Listing 4.5.
Zur Vermeidung mehrfach gleichen Programmtexts für das Schließen von Streams auch bei Ausnahmen bietet sich ein `finally`-Block an. Um auch Ausnahmen während des Schließens abzufangen, werden zwei ineinander geschachtelte `try`-Blöcke verwendet. Nur existierende Streams können geschlossen werden. Da bereits beim Erzeugen eine Ausnahme auftreten kann, müssen wir überprüfen, ob `in` und `out` über-

[1] Wir verstehen unter *Streams* immer Ein-/Ausgabestreams in Java, nicht die in Java 8 neu hinzugekommenen Streams des Streams-Frameworks.

Listing 4.5 Einlesen einer Datei und Ausgeben derselben mit Zeilennummern

```
1  import java.io.*;
2  public class Numbered {
3      private static final String errMsg
4          = "Usage: java Numbered <in> <out>";
5
6      public static void main(String[] args) {
7          if (args.length != 2) {
8              System.err.println(errMsg);
9              return;
10         }
11         try {
12             BufferedReader in = null;
13             BufferedWriter out = null;
14             try {
15                 Reader r = new FileReader(args[0]);
16                 Writer w = new FileWriter(args[1]);
17                 in = new BufferedReader(r);
18                 out = new BufferedWriter(w);
19                 String line;
20                 for (int i = 1;
21                      (line = in.readLine()) != null;
22                      i++) {
23                     out.write(String.format(
24                                 "%6d: %s\n", i, line));
25                 }
26             } finally {
27                 if (in != null) { in.close(); }
28                 if (out != null) { out.close(); }
29             }
30         } catch (IOException e) {
31             System.err.println("I/O Error: "
32                                + e.getMessage());
33         }
34     }
35 }
```

Zustand überprüfen

haupt Streams referenzieren. Diese beiden Variablen werden mit `null` initialisiert um eine nicht erfolgte Objekterzeugung im `finally`-Block erkennen zu können. Entsprechend der allgemein gültigen Scoping-Regeln darf nicht auf Variablen zugegriffen werden, die zu einem bereits geschlossenen Block gehören. Wir dürfen im `finally`-Block (oder in einem `catch`-Block) keine Variablen verwenden, die im entsprechenden `try`-Block deklariert wurden. Aus diesem Grund wurden `in` und `out` schon vor dem inneren `try`-Block deklariert und initialisiert.

Listing 4.5 verwendet gepufferte Zeichen-Streams in `in` und `out`. Diese werden aus den ungepufferten Zeichen-Streams in `r` und `w` erzeugt. Konstruktoren von `FileReader` und `FileWriter` nehmen als Parameter die Namen der Dateien, die zum Lesen bzw. Schreiben geöffnet werden sollen. Beim Schließen von `in` bzw. `out` wird jeweils auch der entsprechende ungepufferte Stream geschlossen. Als Kodierung der Daten in diesen Dateien wird die Default-Kodierung angenommen, die nicht angegeben werden muss. Ist eine andere Kodierung nötig, müssen wir zuerst einen ungepufferten Stream mit rohen Daten erzeugen (`FileInputStream`, `FileOutputStream`), daraus unter Angabe der Kodierung einen ungepufferten Zeichen-Stream (`InputStreamReader`, `OutputStreamWriter`) und daraus den gepufferten Zeichen-Stream.

Ein Grund für die Verwendung gepufferter Streams ist auch die Verfügbarkeit praktischer Methoden wie `readLine`. Zur Erzeugung der auszugebenden Zeilen bietet sich `format` in `String` an. Diese Methode hat nach einer Steuer-Zeichenkette beliebig viele weitere Parameter. Das Ergebnis ist die Steuer-Zeichenkette, wobei darin enthaltene Steuerzeichenfolgen durch Werte der weiteren Parameter ersetzt sind (in der gegebenen Reihenfolge). So steht `%6d` für eine ganze Zahl dargestellt als sechsstellige Dezimalzahl und `%s` für eine Zeichenkette beliebiger Länge. Z. B. gibt `String.format("%6d: %s\n", 512, "X")` die Zeichenkette `"   512: X\n"` zurück.

format

Die Streams `System.in` und `System.out` sind uns schon aus vielen Beispielen bekannt. Es fällt jedoch auf, dass wir bisher nie Ausnahmen abfangen mussten. `System.in` vom Typ `InputStream` ist ein ungepufferter Stream roher Daten, in dem überprüfte Ausnahmen vom Typ `IOException` auftreten können. So wie bisher verwendet, fangen die Methoden von `Scanner` diese Ausnahmen ab, sodass wir sie nicht sehen. `System.out` vom Typ `PrintStream`, einem Untertyp von `OutputStream`, ist ebenso ein ungepufferter Stream roher Daten, der Methoden anbietet, die keine Ausnahmen werfen. Darunter sind `print` und `println`, aber auch die von erfahrenen Leuten bevorzugte Methode `printf`, die im Wesentlichen `print(String.format(...))` aufruft. Kann die Ausgabe wegen einer Ausnahme nicht erfolgen, wird ein Flag gesetzt und die aufgerufene Methode terminiert normal. Der Zustand des Flags ist über die Methode `checkError` auslesbar. Verzichtet man darauf, bleiben auftretende Probleme bei der Ausgabe unerkannt.

printf

checkError

Wie `System.out` hat die *Standard-Fehlerausgabe* `System.err` den Typ `PrintStream`. Um Fehlermeldungen klar von normalem Output unterscheiden zu können, werden sie nach `System.err` geschrieben.

Fehlerausgabe

Häufig sind die Streams so miteinander verknüpft, dass Fehlermeldungen dennoch neben normalem Output sichtbar werden. In einer IDE ist es beispielsweise oft so, dass Output und Fehlermeldungen zwar im selben Fenster angezeigt werden, aber in unterschiedlichen Farben.

Kommandozeilenargument

Eine Form der Eingabe kommt ohne Streams aus: *Kommandozeilenargumente*. Diese werden beim Aufruf des Java-Interpreters in einer Kommandozeile angegeben. Beispielsweise übergibt das Kommando `java Numbered x y` die beiden Argumente `x` und `y` als Zeichenketten an die Methode `Numbered.main`. In `main` sehen wir diese Argumente als Arrayinhalte des Parameters `args`. Listing 4.5 interpretiert auf diese Art eingegebene Zeichenketten als Dateinamen, liest den Inhalt der Datei `x` (falls vorhanden) und schreibt in die neue Datei `y` (und ersetzt dabei den Inhalt einer eventuell schon bestehenden Datei `y`).

anhängen oder überschreiben

`FileWriter` hat auch einen Konstruktor mit zwei Parametern. Der erste ist der Dateiname, der zweite vom Typ `boolean` gibt an, was bei einer schon bestehenden Datei gemacht werden soll: `false` löscht die Datei, `true` hängt das Geschriebene an das Ende der Datei.

automatisches Schließen

In aktuellen Java-Versionen implementieren alle Klassen für Streams das Interface `AutoCloseable`, das nur die Methode `close` enthält. Für Klassen, die dieses Interface implementieren, kann eine erweiterte Form von `try`-Blöcken am Ende `close` automatisch ausführen, nur dann wenn nötig und ohne dafür einen `finally`-Block zu brauchen:

Listing 4.6 Automatische Ressourcenverwaltung im `try`-Block

```
try ( BufferedReader in = new BufferedReader(...);
      BufferedWriter out = new BufferedWriter(...)
) { ... }
catch(IOException ex) { ... }
```

Am Ende des `try`-Blocks wird zuerst `out` und dann `in` geschlossen, auch wenn eine Ausnahme geworfen wurde.

Aufgabe 4.7 Zählen Sie unterschiedliche Arten von Java-Streams zur Ein- und Ausgabe auf und beschreiben Sie die Unterschiede zwischen ihnen. Beschreiben Sie auch, in welchen Situationen Sie welche Arten von Streams bevorzugen würden.

Aufgabe 4.8 Beschreiben Sie, wie Streams (unterschiedlicher Arten) in Java üblicherweise erzeugt, verwendet und geschlossen werden.

Aufgabe 4.9 Beschreiben Sie, wie in typischen Programmen mit Ein- und Ausgabe mögliche Ausnahmen gehandhabt werden, auch warum in Listing 4.5 zwei ineinander geschachtelte `try`-Blöcke nötig sind.

Aufgabe 4.10 Beschreiben Sie, auf welche Weise in Java die Ein- und Ausgabe ermöglicht wird, ohne Ausnahmen abfangen zu müssen.

Aufgabe 4.11 Beschreiben Sie, wie die Methode `printf` aus der Klasse `PrintStream` verwendet wird und warum diese Methode in der Praxis so häufig eingesetzt wird.

Aufgabe 4.12 Beschreiben Sie, wozu das Interface `AutoCloseable` dient und wie Sie daraus Nutzen ziehen können.

4.1.3 Validierung von Eingabedaten

Wir sprechen von einem *Fehler,* wenn Erwartungen, die wir als erfüllt annehmen, doch nicht erfüllt sind. Beim Programmieren achten wir stets darauf, nur erfüllte Annahmen zu treffen. Manchmal müssen wir aber unsichere Annahmen treffen. Beispielsweise nehmen wir an, dass ausreichend Speicher zur Verfügung steht, die Hardware funktioniert und das Betriebssystem Ressourcen zur Verfügung stellt. Sollte eine solche (fast sichere) Annahme doch einmal verletzt sein, ist es legitim, wenn die Programmausführung in einer Fehlermeldung endet.

Über eingegebene Daten dürfen wir keine unsicheren Annahmen treffen. Daten, die von außerhalb des Programms kommen, sind immer als beliebig anzusehen. Wir müssen davon ausgehen, dass auch unerwartete Eingaben getätigt werden, die alle Vorgaben ignorieren. Vom Programm aus gesehen gibt es keine *Eingabefehler* weil jede Eingabe erlaubt ist. Aus Anwendungssicht liegt ein Eingabefehler vor, wenn das Programm mit anderen Daten als erwartet gefüttert wird und daher unerwartete Ergebnisse liefert.

Benötigen wir von eingegebenen Daten zur Weiterverarbeitung bestimmte Eigenschaften, müssen wir die Daten überprüfen (das heißt *validieren*) und eigene Programmpfade vorsehen um auch mit Daten ohne diese Eigenschaften umzugehen. In Listing 4.5 erfolgen Dateizugriffe nur dann, wenn das Programm mit genau zwei Kommandozeilenargumenten aufgerufen wurde, andernfalls erfolgt eine Fehlermeldung – Fehler aus Anwendungssicht. Außerdem ist (ohne eigenen Programmtext) sichergestellt, dass die beiden Argumente alle Eigenschaften erfüllen, die wir von Dateinamen (mit Pfadangaben) erwarten. Sind diese Eigenschaften nicht erfüllt, werfen die Konstruktoren von `FileReader` und `FileWriter` Ausnahmen, die wir abfangen und weitermelden. Aus `in` können wir nur deswegen ohne Überprüfungen Daten lesen, weil wir alle Daten unabhängig von ihrer Struktur weiterverarbeiten.

Listing 4.7 Überprüfung von Eingaben, Fehlermeldungen über Ausnahmen

```
1  import java.io.*;
2  import java.util.*;
3  import java.util.regex.Pattern;
4
5  public class StudInfo {
6      private final static Pattern
7        SEP = Pattern.compile(";"),
8        PAT = Pattern.compile("\\d{8};\\w(\\w| )*;\\d+");
9      private String name;
10     private int points;
11
12     private StudInfo(String n) { name = n; }
13
14     public static void main(String[] args) {
15         try {
16             if (args.length < 2)
17                 throw new IOException(...);
18             Map<Long, StudInfo> map = new HashMap<>();
19             Set<String> used = new HashSet<>();
20             for (int i = 0; i < args.length-1; i++) {
21                 if (used.contains(args[i]))
22                     throw new IOException(...);
23                 used.add(args[i]);
24                 if (args[i].equals("-"))
25                     add(map, System.in);
26                 else try (InputStream s =
27                           new FileInputStream(args[i]))
28                     { add(map, s); }
29             }
30             if (args[args.length - 1].equals("-"))
31                 write(map, System.out);
32             else try (PrintStream p =
33                   new PrintStream(args[args.length-1]))
34                 { write(map, p); }
35         } catch (IOException e) {
36             System.err.println(... + e.getMessage());
37         }
38     }
39     private static void add(Map<Long, StudInfo> map,
40                             InputStream in)
41                         throws IOException {
42         Set<Long> used = new HashSet<>();
43         Scanner lines = new Scanner(in);
44         while (lines.hasNextLine()) {
45             String line = lines.nextLine();
46             if (!line.isEmpty()) {
47                 if (!PAT.matcher(line).matches())
48                     throw new IOException(...);
```



```
49                 Scanner s = new Scanner(line);
50                 s.useDelimiter(SEP);
51                 long n = s.nextLong();
52                 String name = s.next();
53                 int points = s.nextInt();
54                 if (used.contains(n))
55                     throw new IOException(...);
56                 used.add(n);
57                 StudInfo i = map.getOrDefault(n,
58                                   new StudInfo(name));
59                 if (!i.name.equals(name))
60                     throw new IOException(...);
61                 i.points += points;
62                 map.put(n, i);
63             }
64         }
65     }
66     private static void write(Map<Long, StudInfo> map,
67                               PrintStream s) {
68         Long[] ns = new Long[map.size()];
69         map.keySet().toArray(ns);
70         Arrays.sort(ns);
71         for (Long n : ns) {
72             StudInfo i = map.get(n);
73             s.printf("%08d;%s;%d\n",n,i.name,i.points);
74         }
75     }
76 }
```

Überprüfen der Struktur

Listing 4.7 zeigt eine Möglichkeit von vielen zur Überprüfung strukturierter Eingabedaten. Das Programm fasst die von Studierenden bei einzelnen Beurteilungsschritten (Übungen und Tests) erreichten Punkte zu Gesamtpunktezahlen zusammen. Für jeden Beurteilungsschritt gibt es eine Datei, die pro beurteilter Person eine solche Zeile enthält:

⟨Matrikelnummer⟩;⟨Name wie in TISS⟩;⟨Punktezahl⟩

Die Ergebnisdatei soll dieselbe Struktur haben und nach Matrikelnummern sortiert sein. Listing 4.7 erzeugt die Ergebnisdatei nur, wenn

- vor der Ergebnisdatei mindestens eine Eingabedatei als Kommandozeilenargument angegeben wurde, wobei „-“ als Dateiname für die Standardein- oder -ausgabe steht,
- mehrere Eingabedateien voneinander verschiedene Namen haben,
- alle Zeilen einer Datei unterschiedliche Matrikelnummern haben,

- Personen mit gleicher Matrikelnummer aus unterschiedlichen Eingabedateien den gleichen Namen haben,
- jede Zeile die vorgegebene Struktur hat, wobei Matrikelnummern aus acht Ziffern, Namen aus ASCII-Buchstaben und Leerzeichen, sowie erreichte Punkte aus ein oder mehreren Ziffern bestehen.

Ein großer Teil des Programmtexts ist nur für diese Überprüfungen zuständig. Derartiges ist auch in realen Programmen durchaus üblich.

plausibel

Wir sprechen von *Plausibilitätsprüfungen* weil wir nicht feststellen können, ob die Daten richtig sind, sondern nur, ob die Korrektheit plausibel scheint. Überprüfte Details können in einem weiten Bereich variieren. Wir könnten in Namen etwa auch Umlaute, Akzente und bestimmte Sonderzeichen zulassen oder die Punkte auf drei Stellen beschränken.

Java

Pattern

Listing 4.7 vereinfacht die Überprüfung mittels *Pattern-Matching:* Der *compilierte reguläre Ausdruck* in `PAT` beschreibt die gewünschte Struktur einer Zeile – siehe die Spezifikation der Klasse `Pattern`. Hat die Zeile nicht diese Form, wird eine Ausnahme geworfen. Nach nur einer Überprüfung (Programmzeile 47) ist garantiert, dass die Zeile keine andere Form haben kann. So ist es leicht, die Zeile ohne weitere Überprüfungen mittels `Scanner` in die einzelnen Teile mit den gewünschten Typen zu zerlegen, wobei die Teile nicht wie meist durch White-Space voneinander getrennt sind, sondern durch `";"`, wie in Programmzeile 50 über den compilierten regulären Ausdruck `SEP` angegeben.

In Listing 4.7 verwenden wir überprüfte Ausnahmen um Fehlermeldungen zu sammeln und an zentraler Stelle auszugeben. Das vereinfacht die Programmstruktur. Wir müssen aber nicht mit Ausnahmen arbeiten, sondern können auch direkt Fehlermeldungen nach `System.err` schreiben. Die zentralisierte Stelle funktioniert nur, wenn das Programm nach jeder Art von Fehler auf dieselbe Weise fortgesetzt wird – im Beispiel mit der Beendigung. Wenn für jede Art von Fehlermeldung eine unterschiedliche Programmfortsetzung nötig ist, wäre es besser, auf das Werfen von Ausnahmen zu verzichten.

Oft wird empfohlen, jede Annahme im Programm immer wieder zu überprüfen um Fehler rasch zu erkennen. In dieser Allgemeinheit geht die Empfehlung leider am wesentlichen Punkt vorbei. Wir müssen nur jene Annahmen prüfen, die unbekannte, von außen kommende Daten betreffen. Die Überprüfungen müssen sehr zuverlässig sein, möglichst nah an der Schnittstelle, über die Daten hineinkommen. Danach können wir uns auf die Annahmen verlassen. Beispielsweise wird in Listing 4.7, Zeilen 51–53 nicht überprüft, ob wirklich ein `long`-Wert, eine Zeichenkette und ein `int`-Wert aus einer Zeile lesbar ist. Wir wissen das ja schon. Unnötige Überprüfungen erhöhen die Komplexität und können dazu führen, dass wir Überprüfungen an Schnittstellen unterlassen weil wir uns fälschlich auf spätere Überprüfungen verlassen.

Manchmal ist nicht klar, was von außen kommt und was nicht. Beispielsweise haben wir in Listing 4.3 überprüft, ob der Parameterwert im erwarteten Bereich liegt. Das ist sinnvoll, wenn die Methode außerhalb unserer Kontrolle aufgerufen wird. Jedoch ist das Werfen einer Ausnahme nur eine Notlösung. Wir wollen schon vor der Programmausführung sicherstellen, dass die Methode nur erwartete Parameter bekommt. Dieses Thema behandeln wir im nächsten Abschnitt.

Aufgabe 4.13 Beschreiben Sie, was man unter der Validierung von Eingabedaten versteht und warum alle eingegebenen Daten vor der Verarbeitung validiert werden müssen.

Aufgabe 4.14 Beschreiben Sie, was man unter einer Plausibilitätsprüfung versteht und wozu solche Überprüfungen dienen.

Aufgabe 4.15 Beschreiben Sie, wozu die Klasse `Pattern` in Java dient, was ein regulärer Ausdruck beschreibt, was Pattern-Matching macht und wozu Pattern-Matching eingesetzt werden kann.

Aufgabe 4.16 Schreiben Sie ein Programm zur Verwaltung eines Warenlagers, einer Briefmarkensammlung, eines Möbeldepots, oder etwas Ähnlichem. Konzentrieren Sie sich auf die Ein- und Ausgabe. Entwerfen Sie ein passendes Format für erfasste Daten und sorgen Sie dafür, dass alle eingegebenen Daten vollständig und plausibel sind. Beschreiben Sie, wie Sie bei der Lösung dieser Programmieraufgabe vorgegangen sind und worauf Sie besonders geachtet haben.

4.2 Statisches Programmverstehen

Die wichtigste Kompetenz beim Programmieren besteht im Verstehen von Programmen. Ganz grob unterscheiden wir folgende Arten:

1. Verstehen der Syntax und Semantik von Sprachelementen sowie des Programmaufbaus durch Zusammensetzen von Elementen.
2. Nachvollziehen des Programmablaufs auch bei komplexen Programmen, Vorhersehen von Ergebnissen für bestimmte Eingaben.

3. Statisches Verstehen beim Programmlesen, Vorhersehen der Gesamtheit aller Ergebnisse ohne Annahme bestimmter Eingaben.

Die Reihenfolge spiegelt den Lernfortschritt wider. Grundvoraussetzung für eine effiziente, professionelle Softwareentwicklung ist das statische Programmverstehen, das wir uns leider nur mit viel Aufwand aneignen können. Große und komplexe Programme können auch sehr erfahrene Personen nur bei klar strukturiertem Vorgehen und bei Anwendung entsprechender Techniken statisch verstehen. Einige solche Vorgehensweisen und Techniken sehen wir in diesem Abschnitt.

4.2.1 Design-by-Contract

Softwarevertrag

Design-by-Contract ist eine weit verbreitete Vorgehensweise in der Programmentwicklung. Dabei betrachten wir jedes Objekt als *Server,* der Leistungen (*Services*) in Form aufrufbarer Methoden anbietet, und gleichzeitig als *Client,* der Leistungen von Servern durch Methodenaufrufe in Anspruch nimmt. Alle Leistungen werden durch *Verträge* (*Contracts*) zwischen Servern und Clients beschrieben. Ein Vertrag hält fest, was sich ein Client vom Server und der Server vom Client erwarten darf. Zuerst legen wir Verträge fest. Dann implementieren wir Klassen auf Basis der Verträge. Bei Auftreten eines Fehlers klären wir anhand der Verträge, welcher Programmteil wegen einer Vertragsverletzung daran Schuld ist; dort sollte der Fehler behoben werden. Durch das Einhalten aller Vertragsdetails schließen wir viele Fehler aus.

Zusicherung

Verträge halten sich an eine vorgegebene Struktur. Jeder Vertrag wird durch eine Menge von *Zusicherungen* spezifiziert. Jede Zusicherung ist eine Bedingung, die zu festgelegten Zeitpunkten erfüllt sein muss. Manchmal sind Zusicherungen formal angeschrieben, häufig nur informell als Kommentare. Wir unterscheiden folgende Arten:

- Eine *Vorbedingung* (*Precondition*) bezieht sich auf eine bestimmte Methode (ein Service); sie muss zu den Zeitpunkten aller Aufrufe der Methode erfüllt sein. Der Aufrufer (Client) muss für die Einhaltung sorgen. Der Server, genauer die Implementierung der aufgerufenen Methode darf sich darauf verlassen. Die meisten Vorbedingungen beschreiben erwartete Eigenschaften von Methodenparametern zu den Aufrufzeitpunkten.

- Eine *Nachbedingung* (*Postcondition*) bezieht sich ebenso auf eine bestimmte Methode; sie muss jedoch am Ende (bei der Rückkehr aus) der Methode erfüllt sein. Der Server, genauer die Implementierung der aufgerufenen Methode muss für die Einhaltung sorgen. Der Client (Aufrufer) darf sich darauf verlassen. Eine Nachbedingung beschreibt meist, was die Methode macht und welche Eigenschaften der Rückgabewert hat.

- Eine *Invariante* (*Invariant*) bezieht sich auf den Zustand eines Objekts bzw. die Inhalte von Objektvariablen; sie muss bei jedem Aufruf und am Ende jeder Ausführung jeder Methode des Objekts erfüllt sein. Der Server, genauer die gesamte Implementierung der Klasse muss für die Einhaltung sorgen. Jeder Client darf sich darauf verlassen. Vereinfacht gesehen ist eine Invariante eine Bedingung auf dem Objektzustand, die immer gelten muss.[2]

In Programmen finden wir Zusicherungen meist als Kommentare bei den Methodenköpfen (für Vor- und Nachbedingungen) oder bei den Deklarationen von Objektvariablen (für Invarianten). Solche Kommentare sind für uns nicht neu; wir haben sie schon zur Beschreibung abstrakter Datentypen verwendet. Was jetzt hinzukommt ist die klare Unterscheidung zwischen unterschiedlichen Arten von Zusicherungen und die Sichtweise, dass jeder solche Kommentar Bestandteil eines Vertrags ist. Aus dieser Sichtweise folgt recht eindeutig, wer wofür zuständig ist.

Verantwortung

Clients sind für die Einhaltung von Vorbedingungen verantwortlich, Server für die von Nachbedingungen und Invarianten. Aus diesem Grund müssen wir die unterschiedlichen Arten von Zusicherungen voneinander unterscheiden können: Invarianten beziehen sich auf Objektvariablen (hinsichtlich ihrer Konsistenz), Vor- und Nachbedingungen auf Methoden, wobei Vorbedingungen beim Methodenaufruf (meist hinsichtlich von Parameterwerten) und Nachbedingungen am Ende der Methodenausführung von Bedeutung sind. Wir ordnen Kommentare anhand der Inhalte den Zusicherungsarten zu.

Unterscheidung

Wegen der Vorbedingung in Zeile 8 von Listing 4.8 muss jeder Aufrufer von `payOut` sicherstellen, dass die Einlage hoch genug ist. Möglicherweise muss vor dem Aufruf von `payOut` das Argument mit dem Ergebnis eines Aufrufs von `deposit()` verglichen werden. Innerhalb von `payOut` ist keine Überprüfung nötig, weil der Client für die Erfüllung dieser Bedingung verantwortlich ist.

[2] Wenn `x` und `y` Objektvariablen vom Typ `int` sind, wobei `x < y` gelten muss (das ist eine Invariante), dann kann bei Ausführung von `x = ...; y = x + 2;` die Invariante zwischen den Anweisungen kurzzeitig verletzt sein. Das ist erlaubt, solange die kurzzeitige Verletzung von keinem Client gesehen werden kann.

Listing 4.8 Sparbuch mit Zusicherungen – viel Verantwortung beim Client

```
1  public class DepositAccount {
2      private long deposit; // deposit >= 0          Invariante
3
4      // amount > 0;                                  Vorbedingung
5      // deposit incremented by amount               Nachbedingung
6      public void payIn(long amount) {deposit += amount;}
7
8      // amount > 0; amount <= deposit;               Vorbedingung
9      // deposit decremented by amount               Nachbedingung
10     public void payOut(long amount) {deposit -= amount;}
11
12     public long deposit() {return deposit;}
13 }
```

Listing 4.9 Methode in `DepositAccount` – viel Verantwortung beim Server

```
1      // if true: deposit decremented by amount      Nachbed.
2      public boolean payOut(long amount) {
3          if (amount<=0 || amount>deposit) return false;
4          deposit -= amount;
5          return true;
6      }
```

In der Variante in Listing 4.9 liegt die Verantwortung ganz beim Server. Clients können `payOut` in Listing 4.9 ohne Überprüfung aufrufen, alle Überprüfungen erfolgen im Server. Es ist ratsam, Überprüfungen dort anzusiedeln, wo die dafür nötige Information vorhanden ist. Das ist sehr oft der Server. Die Variante in Listing 4.9 ist gegenüber der in Listing 4.8 vorteilhaft. Durch das Werfen einer Ausnahme kann die Verantwortung beim Server liegen, ohne über einen Ergebniswert Informationen an den Aufrufer zurückgeben zu müssen:

Listing 4.10 Werfen einer Ausnahme als Nachbedingung beschrieben

```
1      // deposit decremented by amount;              Nachbedingung
2      // throws IllegalArgumentException if
3      // amount <= 0 or amount > deposit             Nachbedingung
4      public void payOut(long amount) {
5          if (amount <= 0 || amount > deposit)
6              throw new IllegalArgumentException();
7          deposit -= amount;
8      }
```

Wir können Ausnahmen werfen, obwohl der Client verantwortlich ist:

Listing 4.11 Werfen einer Ausnahme trotz Vorbedingung

```
1      // amount > 0; amount <= deposit;              Vorbedingung
2      // deposit decremented by amount               Nachbedingung
3      public void payOut(long amount) {
4          if (amount <= 0 || amount > deposit)
5              throw new IllegalArgumentException();
6          deposit -= amount;
7      }
```

Nachdem laut Vertrag die Vorbedingung in der Methode nicht verletzt sein darf, ist es egal, was die Methode bei verletzter Vorbedingung macht. Durch das Werfen der Ausnahme wird die Zuverlässigkeit zusätzlich erhöht. Allerdings sollen wir unklare Verantwortungen in Verträgen wie in Listing 4.12 vermeiden:

Listing 4.12 Vorbedingung und Nachbedingung – Verantwortung nicht klar

```
1      // amount > 0; amount <= deposit;              Vorbedingung
2      // otherwise IllegalArgumentException;         Nachbedingung
3      // deposit decremented by amount               Nachbedingung
4      public void payOut(long amount) {
5          if (amount <= 0 || amount > deposit)
6              throw new IllegalArgumentException();
7          deposit -= amount;
8      }
```

Neben den in Verträgen beschriebenen Bedingungen gelten auch ganz allgemeine, nirgends genau festgelegte übliche Erwartungen an Programmtexte. Beispielsweise wird erwartet, dass eine Methode namens `insert` etwas einfügt und nicht etwa Daten löscht. Generell können wir erwarten, dass eine Methode Objekte nicht verändert, außer Methodennamen legen entsprechende Änderungen nahe oder Zusicherungen beschreiben Änderungen. Allgemeine Erwartungen müssen erfüllt sein, außer wenn Verträge die Erwartungen explizit einschränken. Folgende Methoden sind daher hinsichtlich Design-by-Contract fehlerhaft:

Erwartung

Listing 4.13 Implementierungen entsprechen nicht den Erwartungen

```
1      // if true: deposit incremented by amount      Nachbed.
2      public boolean payIn(long amount) { return false; }
3
4      // if true: deposit decremented by amount      Nachbed.
5      public boolean payOut(long amount) {
6          deposit -= amount;
7          return false;
8      }
```

Wir müssen Kommentare sehr genau lesen um sie als Zusicherungen zu verstehen. Betrachten wir Listing 2.3 auf Seite 34: Der erklärende Text in Zeile 1 führt in den Kontext ein; das ist keine der oben genannten Arten von Zusicherungen, sondern schränkt den Interpretationsspielraum für den gesamten Text ein. `width` und `height` in Zeile 3 beziehen sich auf den Objektzustand, aber es kann sich um keine Invariante handeln, weil die Bedingung nicht immer gelten muss, sondern nur nach der Objekterzeugung; Zeile 3 ist daher eine Nachbedingung für jeden Konstruktor (einschließlich der Initialisierung von Objektvariablen). Zeile 4 ist wie Zeile 2 eine Aussage, die sich auf den gesamten Text bezieht; wenn irgendwo ein Zeilenindex vorkommt, dann muss die genannte Bedingung gelten. Die Zeilen 9, 10, 14–18, 22–27, 30 und 31 enthalten Nachbedingungen für die davor genannten Methoden. Wie üblich machen Nachbedingungen den größten Teil der Beschreibungen aus. Vorbedingungen in den Zeilen 11 und 19 sind hier durch `Requires:` gekennzeichnet. Die Vorbedingung in Zeile 19 könnte weggelassen werden, weil sie nur eine Wiederholung der allgemeinen Aussage in Zeile 4 darstellt; auch ohne diesen Text wäre klar, dass der Wert von `index` nur in diesem Bereich liegen darf.

Zusicherungen müssen auch bei der Bildung von Untertypen berücksichtigt werden. Folgende Bedingungen sind unbedingt einzuhalten:

Zusicherungen und Untertypen

- Vorbedingungen auf Methoden in Untertypen dürfen nur gleiche oder schwächere Einschränkungen ausdrücken als auf entsprechenden Methoden in Obertypen – Verknüpfung mit ODER. Ist z. B. `x > 5` eine Vorbedingung auf der Methode im Obertyp, kann die Vorbedingung auf der Methode im Untertyp `x > 3` sein; `x > 5 || x > 3` entspricht `x > 3`.

- Nachbedingungen auf Methoden in Untertypen dürfen nur gleiche oder stärkere Einschränkungen ausdrücken als auf entsprechenden Methoden in Obertypen – Verknüpfung mit UND. Ist z. B. `x > 3` eine Nachbedingung auf der Methode im Obertyp, kann die Nachbedingung auf der Methode im Untertyp `x > 5` sein; `x > 3 && x > 5` entspricht `x > 5`.

- Invarianten in Untertypen dürfen nur gleiche oder stärkere Einschränkungen ausdrücken als in Obertypen – Verknüpfung mit UND. Ist z. B. `x > 3` eine Invariante im Obertyp, kann die Invariante im Untertyp `x > 5` sein; `x > 3 && x > 5` entspricht `x > 5`. Achtung: Das gilt nur, wenn `x` von außerhalb des Objekts nicht geschrieben wird – ein weiterer Grund, warum Objektvariablen generell `private` sein sollen. Kann eine Objektvariable von außerhalb geschrieben werden, muss eine Invariante, die sich auf diese Variable bezieht, in Unter- und Obertyp gleich sein.

Aufgabe 4.17 Beschreiben Sie, was man unter Design-by-Contract versteht, wie man dabei vorgeht und wozu Design-by-Contract dient.

Aufgabe 4.18 Beschreiben Sie, was ein Softwarevertrag entsprechend Design-by-Contract ausdrückt und wer die Vertragspartner sind.

Aufgabe 4.19 Beschreiben Sie, welche Arten von Zusicherungen in Design-by-Contract unterschieden werden, woran man die jeweilige Art erkennt und was durch sie ausgedrückt wird.

Aufgabe 4.20 Beschreiben Sie, wer für die Einhaltung von Zusicherungen (abhängig von der Art) verantwortlich ist und zu welchen Zeitpunkten welche Zusicherungen erfüllt sein müssen.

Aufgabe 4.21 Wählen Sie beliebige Klassen aus den Java-Standard-Bibliotheken, betrachten Sie die entsprechenden API-Dokumentationen und beschreiben Sie, welche Textbestandteile darin Zusicherungen darstellen und welcher Art diese Zusicherungen sind. Geben Sie zu Methodenbeschreibungen an, ob ein Großteil der Verantwortung eher beim Client oder beim Server liegt.

Aufgabe 4.22 Betrachten Sie von Ihnen selbst verfasste Kommentare aus früheren Programmieraufgaben. Beschreiben Sie, welche dieser Kommentare Zusicherungen darstellen und welcher Art diese Zusicherungen sind. Ändern Sie bei Bedarf die Kommentare so ab, dass sie unmissverständlich als Zusicherungen lesbar sind.

Aufgabe 4.23 Beschreiben Sie, welchen Einfluss Zusicherungen auf das Bestehen von Untertypbeziehungen haben. Welche Einschränkungen bestehen ganz konkret?

4.2.2 Statisches Verstehen des Programmablaufs

verstehen = beweisen

Das statische Verstehen eines Programmteils ähnelt dem formalen Beweisen der Korrektheit des Programmteils. Wir suchen nach einem Korrektheitsbeweis. Finden wir einen, haben wir den Programmteil verstanden; sonst ist der Programmteil nicht korrekt oder wir haben ihn noch nicht verstanden. Die wesentliche Frage ist, welche konkreten Ei-

genschaften zu beweisen sind, das heißt, unter welchen Bedingungen der Programmteil als korrekt gilt. Für Methoden gibt uns Design-by-Contract die Antwort: Wir müssen für jede Methode beweisen, dass am Ende der Methodenausführung die Nachbedingungen (der Methode) und die Invarianten (der Klasse) gelten, wobei wir annehmen, dass zu Beginn der Ausführung die Vorbedingungen und Invarianten erfüllt sind. Allerdings sind Zusicherungen meist nicht so präzise und vollständig formuliert, dass ein formaler Beweis durchführbar wäre. Wir suchen also vielmehr nach einem groben Beweisschema, das mit den Ungenauigkeiten zurechtkommt. Mit entsprechender Erfahrung entwickeln wir passende Beweisschemata gleich beim konzentrierten Lesen des Programmtexts, quasi nebenbei ohne Zusatzaufwand.

Design-by-Contract

Wir müssen aus am Methodenanfang angenommenen Bedingungen die am Methodenende geforderten Bedingungen herleiten. Dabei hilft der *Hoare-Kalkül,* ein formales Modell, das wir oberflächlich skizzieren: Ein *Hoare-Tripel* $\{V\}S\{N\}$ kombiniert ein Programmsegment S (Ausschnitt aus dem Programm) mit einer Vorbedingung V (gilt vor Ausführung von S) und einer Nachbedingung N (gilt nach Ausführung von S). Ist S eine elementare Anweisung, kennen wir die Semantik und leiten N aus V und S ab, zum Beispiel: $\{x = 1; y = 2\}$`x = y;`$\{x = 2; y = 2\}$. Ist S ein Methodenaufruf, dann ist die Auswirkung aus den Zusicherungen der Methode (Design-by-Contract) ableitbar: V muss die Vorbedingungen der Methode implizieren, N ergibt sich aus den Nachbedingungen der Methode. Hintereinanderausführung und bedingte Ausführung sind über einfache Regeln darstellbar. So ergibt $\{V\}S_1\{X\}$ und $\{X\}S_2\{N\}$ hintereinander ausgeführt $\{V\}S_1S_2\{N\}$. Schleifen sind komplizierter, aber handhabbar – siehe weiter unten. Solche Verknüpfungen ermöglichen Beweise über das gesamte Programm.

Hoare-Tripel

Java

Die `assert`-Anweisung drückt eine Zusicherung ähnlich wie in einem Hoare-Tripel aus, zwischen zwei Anweisungen sowohl als Vor- als auch Nachbedingung, siehe Listing 4.14. Meist haben solche Anweisungen, wie Kommentare, keine Auswirkungen auf das Programm.[3] Aber der Compiler prüft die Syntax. Für Zusicherungen, die nicht der Java-Syntax entsprechen (z.B. weil sie nach einer `return`-Anweisung stehen), können wir nur Kommentare verwenden.

`assert`

[3] Wir können über das Interpreter-Flag `-ea` (*enable assertions*) die Überprüfung von `assert`-Anweisungen einschalten. Dann werden die Ausdrücke zur Laufzeit ausgewertet und Ausnahmen vom Typ `AssertionError` geworfen, wenn sich `false` ergibt. Dennoch sollen wir `assert`-Anweisungen vor der Programmausführung überprüfen und statisch verstehen; im Programmeinsatz ist es zu spät.

Listing 4.14 Binäre Suche, semi-formale Zusicherungen zwischen Anweisungen

```
1  // pre:  sorted(a) && (l>h || (0<=l && h<a.length));
2  // post: result r with r>=0 && a[r]==v && l<=r && r<=h;
3  // post: result r==-1 if not(v,a,l,h); (a not modified)
4  static int searchR(int v, int[] a, int l, int h) {
5      assert sorted(a) && (l>h || (0<=l && h<a.length));
6      if (l > h) {
7          assert sorted(a) && l>h && not(v,a,l,h);
8          return -1;
9      }
10     assert sorted(a) && 0<=l && l<=h && h<a.length;
11     int r = (l + h) / 2;
12     assert sorted(a)&&0<=l&&l<=r&&r<=h&&h<a.length;
13     if (a[r] < v) {
14         assert not(v,a,l,r) && sorted(a) &&
15                 (r+1>h || (0<=r+1 && h<a.length));
16         return searchR(v,a,r+1,h); //not(v,a,l,h) if -1
17     } else if (a[r] == v) {
18         assert a[r]==v && l<=r && r<=h;
19         return r;
20     } else {
21         assert not(v,a,r,h) && sorted(a) &&
22                 (l>r-1 || (0<=l && r-1<a.length));
23         return searchR(v,a,l,r-1); //not(v,a,l,h) if -1
24     }
25 }
26
27 static boolean sorted(int[] a, int i) {
28     return i<=0 || (a[i-1]<=a[i] && sorted(a, i-1));
29 }
30 static boolean not(int v, int[] a, int l, int h) {
31     return l>h || (a[l]!=v && not(v,a,l+1,h));
32 }
```

Zeile 5 in Listing 4.14 wiederholt die Vorbedingung der Methode, die am Anfang stets gilt und zwecks Überprüfbarkeit als ausführbarer Programmtext (unter Verwendung von Hilfsmethoden) formuliert ist. Wenn S die `if`-Anweisung in den Zeilen 6–9 ist, dann steht die Bedingung in Zeile 5 für V und die in Zeile 10 für N in einem Hoare-Tripel. Wir kommen nur dann an das Ende der `if`-Anweisung, wenn die Bedingung `l > h` in Zeile 6 nicht erfüllt ist. In Zeile 10 gilt daher `l<=h` und alles, was daraus zusammen mit Zeile 5 folgt. In Zeile 7 gilt offensichtlich `l<=h` sowie `sorted(a)` direkt abgeleitet aus Zeile 5, aber es gilt auch `not(v,a,l,h)`, weil dies aus `l<=h` folgt (siehe Zeile 31). Letztere Eigenschaft brauchen wir als Nachbedingung für die Rückga-

be von `-1`. In Zeile 3 wird auch verlangt, dass `a` unverändert bleibt. Es wäre gar nicht notwendig, diese Bedingung anzuführen, weil ohne gegenteiligen Kommentar davon auszugehen ist, dass sich `a` nicht ändern darf. Da in der gesamten Methode und allen darin aufgerufenen Methoden nicht schreibend auf `a` zugegriffen wird, ist diese Bedingung erfüllt. Somit ist die Nachbedingung hergeleitet, die in Zeile 8 bei der Rückkehr aus der Methode gelten muss. Auf ähnliche Weise können wir auch Ausdrücke in anderen `assert`-Anweisungen sowie Nachbedingungen bei `return`-Anweisungen herleiten. Eine kleine Schwierigkeit ergibt sich bei der Rückkehr aus rekursiven Aufrufen (Zeilen 16 und 23): Im Fall eines Rückgabewertes von `-1` garantiert die Nachbedingung der rekursiven Aufrufe nur, dass `not(v,a,r+1,h)` bzw. `not(v,a,l,r-1)` gilt, nicht `not(v,a,l,h)`. Um Letzteres zu zeigen, benötigen wir auch noch die Bedingungen `not(v,a,l,r)` bzw. `not(v,a,r,h)`.

Die einzelnen Schritte im Beweis sind häufig einfach, sodass wir sie im Kopf behalten können und nicht anschreiben müssen. Dennoch empfiehlt es sich, beim Lesen des Programms bewusst an die Beweisschritte zu denken, bis man sich so daran gewöhnt hat, dass die Beweise beim Programmlesen automatisch unbewusst im Kopf ablaufen.

partielle Korrektheit

Auf die oben dargestellte Weise haben wir die *partielle Korrektheit* des Programms bewiesen. Wir haben also gezeigt, dass das Ergebnis und der Objektzustand am Ende jeder Methodenausführung den Nachbedingungen und Invarianten entspricht, wenn die Vorbedingungen und Invarianten zu Beginn der Methodenausführung erfüllt sind und die Methode terminiert. Wir haben aber noch nicht gezeigt, dass die Methode terminiert. Zum Beweis der Termination müssen wir für jede rekursive Methodenausführung und jede Schleife nachweisen, dass

Termination

- eine Abbruchbedingung existiert
- und in jedem Rekursionsschritt bzw. jeder Iteration ein Fortschritt erzielt wird, der den Programmzustand um ein fixes Mindestmaß näher an die Abbruchbedingung heranführt.

Fortschritt kann auf unterschiedliche Weise gemessen werden und ergibt sich meist natürlich aus dem verwendeten Algorithmus. Bei der binären Suche wie in Listing 4.14 wird der Suchbereich beispielsweise durch `r+1` in Zeile 16 sowie `r-1` in Zeile 23 um mindestens 1 kleiner. Die Halbierung des Suchbereichs wie in Zeile 11 ist dagegen zum Beweis der Termination ungeeignet, weil kein Mindestmaß des Fortschritts garantiert ist; unterscheiden sich `l` und `h` um höchstens 1 voneinander, dann hat `r` denselben Wert wie `l`, also kein Fortschritt.

Listing 4.15 Binäre Suche mit Schleife

```
1  // pre:  sorted(a);
2  // post: result r with r>=0 && r<a.length && a[r]==v;
3  // post: result r==-1 if not(v,a,0,a.length-1);
4  static int search(int v, int[] a) {
5      assert sorted(a);
6      int l = 0, h = a.length - 1;
7      assert inv(v,a,l,h);
8      while (l <= h) {
9          assert inv(v,a,l,h) && l<=h;
10         int r = (l + h) / 2;
11         assert inv(v,a,l,h) && l<=r && r<=h;
12         if (a[r] < v) {
13             assert inv(v,a,l,h) && l<=r && r<=h
14                    && not(v,a,0,r);
15             l = r + 1;
16         } else if (a[r] == v) {
17             assert r>=0 && r<a.length && a[r]==v;
18             return r;
19         } else {
20             assert inv(v,a,l,h) && l<=r && r<=h
21                    && not(v,a,r,a.length-1);
22             h = r - 1;
23         }
24         assert inv(v,a,l,h);
25     }
26     assert inv(v,a,l,h)&&l>h&&not(v,a,0,a.length-1);
27     return -1;
28 }
29
30 static boolean inv(int v, int[] a, int l, int h) {
31     return sorted(a) && not(v,a,0,l-1)
32                      && not(v,a,h+1,a.length-1);
33 }
```

Bei Schleifen ist der Nachweis der partiellen Korrektheit schwieriger. Wir müssen eine *Schleifeninvariante* finden, das ist eine Bedingung, die beschreibt, was sich in den Iterationen *nicht* ändert – quasi ein Gegenstück zu einer Beschreibung des Fortschritts für den Terminationsbeweis. Die Schleifeninvariante muss vor und nach der Schleife sowie am Beginn und Ende jeder Iteration erfüllt sein. Die Schleifenbedingung kann dadurch nicht Teil der Schleifeninvariante sein.

Schleifeninvariante

In Listing 4.15 steht `inv(v,a,l,h)` für die Schleifeninvariante, und `sorted` sowie `not` sind wie in Listing 4.14 definiert. Die Schleifeninvariante besagt, dass `a` stets sortiert bleibt und das gesuchte Element

an einem Index kleiner `l` oder größer `h` nicht enthalten kann. Daraus folgt auch, dass bei `l>h` das gesuchte Element in `a` nicht vorkommt. Abgesehen von der Schleifeninvariante underscheidet sich die Beweisführung in Listing 4.15 nicht wesentlich von der in Listing 4.14.

Das Finden einer Schleifeninvariante ist ein kreativer Prozess, der schwierig sein kann. Am besten suchen wir gleich beim Erstellen einer Schleife nach einer passenden Schleifeninvariante, Abbruchbedingung und Beschreibung des Fortschritts, weil diese zusammen den Charakter der Schleife vollständig festlegen. Kennen wir diese drei Teile, haben wir die Schleife verstanden, wodurch sich bei der Implementierung der Schleife keine Probleme ergeben werden.

Auf den ersten Blick mag es überraschen, dass wir uns hauptsächlich auf das verlassen, was sich in den Iterationen nicht ändert, obwohl Zustandsänderungen in Schleifen ganz wesentlich sind. Bei Konzentration auf Änderungen müssen wir alle Iterationen durchdenken, bevor wir Ergebnisse kennen. Das ist sehr aufwendig und bei einer unbekannten Anzahl an Iterationen sogar unmöglich. Wenn wir uns auf Schleifeninvarianten verlassen, müssen wir nur einen Schleifendurchlauf durchdenken ohne die Anzahl der Iterationen zu benötigen.

Hinsichtlich der Terminologie müssen wir aufpassen. Vor- und Nachbedingungen gibt es sowohl in Design-by-Contract als auch in Hoare-Tripeln. Vorbedingung heißt alles, was davor gilt, Nachbedingung alles was danach gilt, und alles zusammen sind Zusicherungen. Allerdings beziehen sich Zusicherungen in Design-by-Contract auf Methoden, in Hoare-Tripeln auf Programmsegmente, die z. B. Methodenaufrufe sein können, aber keine Methoden. Das müssen wir auseinanderhalten. Zusicherungen auf Methoden stehen meist vor Methodenköpfen. Zusicherungen im Inneren des Methodenrumpfs beziehen sich dagegen auf Hoare-Tripel. Eine Zusicherung zwischen zwei Anweisungen ist gleichzeitig Nachbedingung (nach der davor stehenden Anweisung) und Vorbedingung (vor der nachfolgenden Anweisung). Nur Zusicherungen entsprechend Hoare-Tripeln sind als `assert`-Anweisungen darstellbar. Schleifeninvarianten sind durch mehrere gleiche oder einander entsprechende Zusicherungen darstellbar (vor und nach Schleifen sowie am Anfang und Ende des Schleifenrumpfs), aber meist ist höchstens eine davon angeschrieben. Wir müssen Schleifeninvarianten strikt von Invarianten in Design-by-Contract unterscheiden, die bei Deklarationen von Objektvariablen stehen und nichts mit Schleifen zu tun haben.

In die Überprüfung von Methodenaufrufen müssen wir nicht nur die Vorbedingungen der Methoden, sondern auch *alle* uns bekannten Invarianten einbeziehen. Alle Invarianten müssen zu Beginn und am Ende von Methodenausführungen erfüllt sein, auch Invarianten auf Objekten, die scheinbar nichts mit der Methodenausführung zu tun haben. Das ist notwendig, weil wir die Abhängigkeiten zwischen Objekten in der Regel nicht kennen und daher nicht ausschließen können, dass während der Methodenausführung möglicherweise doch auf die Zustände scheinbar unbeteiligter Objekte zugegriffen wird. Wir nehmen an, dass alle Objektvariablen `private` und damit von außerhalb der Klasse nicht sichtbar sind. Daher kennen wir keine Zustände von Objekten anderer Klassen. Deren Invarianten brauchen wir nicht zu überprüfen; sie sind automatisch erfüllt, weil die Objektzustände seit Beendigung der letzten Methodenausführungen in diesen Objekten nicht verändert worden sein können. Wir müssen also sicherstellen, dass die Invarianten aller uns bekannten Objekte der eigenen Klasse erfüllt sind.

bei Aufruf Invarianten prüfen

Aufgabe 4.24 Beschreiben Sie, wie Sie ein Programm statisch verstehen können und was das mit Korrektheitsbeweisen zu tun hat?

Aufgabe 4.25 Beschreiben Sie, was ein Hoare-Tripel ist und was man damit machen kann.

Aufgabe 4.26 Beschreiben Sie, was eine `assert`-Anweisung in Java bewirkt, warum man sie verwendet und welcher Zusammenhang mit Hoare-Tripeln besteht.

Aufgabe 4.27 Beschreiben Sie, wie die Termination rekursiver Methodenaufrufe oder von Schleifen nachgewiesen werden kann.

Aufgabe 4.28 Beschreiben Sie, was eine Schleifeninvariante ist, wo sie gelten muss und wozu man sie verwendet.

Aufgabe 4.29 Beschreiben Sie, warum die (negierte) Abbruchbedingung niemals Teil einer Schleifeninvariante sein kann.

Aufgabe 4.30 Beschreiben Sie, worin Vor- und Nachbedingungen in Hoare-Tripeln den Vor- und Nachbedingungen bei Design-by-Contract ähneln und worin sie sich unterscheiden. Beschreiben Sie Gemeinsamkeiten und Unterschiede zwischen Invarianten und Schleifeninvarianten.

Aufgabe 4.31 Beschreiben Sie, warum alle Invarianten in allen Objekten vor jedem Methodenaufruf erfüllt sein müssen, nicht nur in dem Objekt, in dem die Methode aufgerufen wird.

4.2.3 Dokumentation und Code-Review

Im Bereich der Softwareentwicklung werden unzählige Arten von Dokumenten eingesetzt, die bei der Gestaltung und Wartung von Software hilfreich sind, oft unterstützt durch Werkzeuge. Eine besondere Rolle spielen Programmtexte. Sie haben nicht nur den Zweck, einer Maschine Anweisungen zu geben, sondern spiegeln auch unsere Gedanken bei der Programmentwicklung wider, unterstützen uns dabei, die Gedanken fokussiert, organisiert und nachprüfbar konsistent festzuhalten, und dienen dem Gedankenaustausch mit anderen Personen sowie als Gedächtnisstütze für uns selbst. Fast alle wichtigen Aspekte von Programmen werden heute direkt in Programmtexten ausgedrückt. Dazu gehören oft auch die Anforderungen an das System sowie Testfälle. Je umfangreicher und langlebiger Programme werden, desto größer wird die Bedeutung der Programmtexte als Kommunikationsmittel.

Kommentare werden manchmal fälschlich zur umgangssprachlichen Beschreibung komplexer Sprachkonstrukte verwendet. Tatsächlich sollen sie Lesern jene Informationen geben, die für die Konsistenz des Gesamtsystems wichtig und nicht aus Sprachkonstrukten herleitbar sind. Dazu gehören Zusicherungen entsprechend Design-by-Contract sowie komplexe Schleifeninvarianten. Ohne diese Informationen wären Programme und Programmteile kaum zu verstehen. Die Entwicklung guter Zusicherungen kann wesentlich aufwendiger sein als die des eigentlichen Programms, und meist beginnt die Entwicklung mit den Zusicherungen, die ständig aktuell gehalten werden. Oberflächliche Kommentare, die zur Wahrung des Scheins rasch hinzugefügt werden, sind unsinnig.

Unter *Design-Rules* verstehen wir die Gesamtheit aller Regeln, an die sich ein Softwareentwicklungsteam zu halten hat. Je nach Unternehmen und Projekt unterscheiden sich Design-Rules auch bei Kommentaren. Häufig müssen folgende Informationen vorhanden sein:

was Kommentare beschreiben

- Vor einer Klasse oder einem Interface werden Zweck, Grobstruktur und Besonderheiten beschrieben, ebenso die Verantwortlichen für die Erstellung und Wartung sowie die anvisierten Benutzer.
- Bei den Deklarationen von Objektvariablen stehen die Zwecke der Variablen sowie Invarianten.
- Vor Methoden und Konstruktoren stehen Vor- und Nachbedingungen. Letztere beschreiben oft auch den Zweck.
- In Methodenrümpfen stehen Schleifeninvarianten.

Überall können außerdem Hinweise auf erfolgte Änderungen beschrieben sein um zu verhindern, dass an kritischen Stellen Änderungen wiederholt gemacht und wieder zurückgenommen werden. Es fällt auf, dass der Großteil der Kommentare bei Deklarationen oder vor Definitionen steht, vergleichsweise wenig in den Rümpfen von Methoden und Konstruktoren. Die Qualität hängt nicht von der Menge an Kommentaren ab, sondern von der Aussagekraft und guten Platzierung. Manchmal deutet der Bedarf an übermäßig vielen Kommentaren auf unausgereifte, fehleranfällige Programmteile hin, während gut durchdachte und stabile Teile mit nur wenigen Kommentarzeilen auskommen.

Viele Beispiele für sehr hochwertige Dokumentationen von Klassen und Interfaces finden sich in den Java-Standardbibliotheken. Wir sollen ähnlich hohe Standards anstreben, obwohl wir sie wegen beschränkter zeitlicher Ressourcen kaum erreichen werden. Einiges können wir an diesen Musterbeispielen rasch erkennen:

hochwertige Kommentare

- Hochwertige Kommentare sind aus dem Blickwinkel der Anwendung geschrieben und geben Implementierungsdetails nur dort preis, wo dies unumgänglich ist.
- Kommentare sind als allgemeinverständliche Texte gestaltet. Es handelt sich trotzdem durchwegs um unmissverständliche und präzise Zusicherungen gemäß Design-by-Contract.
- Namen entsprechen einem einheitlichen Schema und sind so gewählt, dass sie Zusicherungen intuitiv unterstützen.
- Randfälle sind sehr genau beschrieben.
- Die Textlänge ist dennoch kurz gehalten.

Qualitativ hochwertige Programmteile sind ohne allzugroßen Aufwand statisch verstehbar. Das statische Verständnis ist eine unabdingbare Voraussetzung für den Korrektheitsbeweis und damit längerfristig gesehen für die Korrektheit, und ein hoher Aufwand für das Verstehen kann zu einem gewaltigen Kostenfaktor werden. Es gibt zwar ausgefeilte Werkzeuge um die Korrektheit von Programmen hinsichtlich vorgegebener Kriterien automatisch zu überprüfen, aber kein Werkzeug kann informelle Kommentare und die Intuition hinter Namen verstehen.

Eines der wichtigsten Hilfsmittel zur Beurteilung der Qualität von Programmteilen ist trotz vieler Werkzeuge nach wie vor die *Code-Review*. Dabei liest jemand den Programmtext und gibt Feedback.

Code-Review

Meist werden Code-Reviews von erfahrenen Personen durchgeführt, die nicht zum Entwicklungsteam gehören, gelegentlich lesen und beurteilen Teammitglieder Programmtexte gegenseitig. Das Feedback enthält neben Hinweisen auf allgemeine Auffälligkeiten Beurteilungen anhand einer vorgegebenen Liste von Kriterien, die sich von Fall zu Fall unterscheidet. Die Kriterienliste orientiert sich an den Design-Rules. Automatisch beurteilbare Kriterien kommen darin in der Regel nicht vor um die Konzentration auf das Wesentliche fokussiert zu halten. Hier sind einige Beispiele für zu überprüfende Kriterien:

Kriterien für Code-Review

- Sind Software-Verträge entsprechend Design-by-Contract so beschrieben, dass in der Softwareentwicklung kompetente Personen klar verstehen, was verlangt wird und was erwartet werden darf?
- Erfüllen Implementierungen von Methoden und Konstruktoren nachweislich alle ihre in den Software-Verträgen beschriebenen Pflichten, das heißt, werden unter der Annahme, dass Vorbedingungen und Invarianten erfüllt sind, alle Nachbedingungen und Invarianten eingehalten?
- Erfüllen Methodenaufrufe nachweislich alle ihre in den Software-Verträgen beschriebenen Pflichten, das heißt, sind alle Vorbedingungen aufgerufener Methoden erfüllt und sind zu den Zeitpunkten von Methodenaufrufen alle bekannten Invarianten erfüllt?
- Sind Kommentare konsistent mit dem Rest des Programms?
- Sind Untertypen wirklich Spezialisierungen ihrer Obertypen?
- Werden nur nötige Eigenschaften vorausgesetzt (etwa kein Untertyp verlangt, wo auch ein Obertyp reichen würde)?
- Sind alle Teile mit vertretbarem Aufwand statisch zu verstehen?
- Ist die Namensgebung in sich konsistent und intuitiv?
- Werden alle Daten validiert, die von außerhalb des Programms kommen könnten?
- Sind Überprüfungen von Daten ausreichend restriktiv, sodass ein Eindringen von außen nicht möglich ist?
- Werden Überprüfungen von Daten nicht unnötig wiederholt und sind sie in sich konsistent?

- Werden adäquate Algorithmen und Datenstrukturen eingesetzt?
- Kommen keine unnötigen, möglicherweise nach Änderungen übriggebliebenen Zusicherungen oder Programmteile vor?
- Ist die Sichtbarkeit von Programmteilen bestmöglich beschränkt?

Diese Liste ließe sich unendlich lange fortsetzen.

Code-Reviews sind extrem fordernd, weil hundertprozentige Konzentration über lange Zeiträume notwendig ist. Entsprechend langsam sind die Fortschritte. Dennoch zahlen sich Code-Reviews aus, weil nur durch sie eine hohe Programmqualität erreichbar ist. Code-Reviews bewirken unter anderem, dass bei der Programmerstellung noch mehr Wert auf die überprüften Kriterien gelegt wird als sonst.

Aufgabe 4.32 Beschreiben Sie, welche Informationen die Programmdokumentation in Form von Kommentaren geben soll und wo innerhalb einer Klasse meist welche Informationen gefunden werden können.

Aufgabe 4.33 Beschreiben Sie, was hochwertige Kommentare ausmacht, und geben Sie Beispiele dafür.

Aufgabe 4.34 Beschreiben Sie, was man unter Code-Review versteht, wie man dabei vorgeht und wozu man sie macht. Geben Sie auch Beispiele für Kriterien, die dabei überprüft werden.

Aufgabe 4.35 Wählen Sie ein vor längerer Zeit selbst geschriebenes Programm und prüfen Sie in einer Code-Review, wie gut die Kriterien aus Abschnitt 4.2.3 erfüllt sind.

Aufgabe 4.36 Tauschen Sie mit einer Studienkollegin oder einem Studienkollegen selbstgeschriebene Programme aus und beurteilen Sie die Programme gegenseitig durch Code-Reviews nach den Kriterien aus Abschnitt 4.2.3.

4.3 Testen und Verbessern

Wir wissen aus Erfahrung mit kleinen Programmen, wie wir über Testen wiederholt Fehler aufdecken, diese beseitigen und so Schritt für Schritt scheinbar für Korrektheit sorgen. Ganz so einfach ist die Sache jedoch nicht, wie wir leicht aus Fakten zu diesem Thema ersehen. Verbesserungen setzen voraus, dass wir Programme im Detail verstehen, was mit zunehmender Größe und Komplexität schwieriger wird.

4.3.1 Testen

Wissenswertes zum Testen

Unter dem *Testen* von Programmen verstehen wir in erster Linie das Ausprobieren. Das klingt einfach, ist es aber nicht, wenn es um nichttriviale Programme geht und gewisse Qualitätsstandards gefordert werden. Folgende Tatsachen sind, jede für sich, einfach nachvollziehbar, ergeben in der Summe jedoch ein Gesamtbild, das in Widerspruch zu Erfahrungen mit kleinen Programmen steht:

- Durch intensiveres Testen lassen sich mehr Fehler finden, aber es lassen sich niemals alle Fehler zuverlässig aufdecken.
- Die Anzahl der durch Testen (einer bestimmten Intensität) aufgedeckten Fehler ist häufig proportional zur Anzahl vorhandener Fehler. Werden in einem Programmteil mehr Fehler gefunden als in anderen, sind in dem einen Programmteil wahrscheinlich auch entsprechend mehr Fehler vorhanden als in anderen. Testergebnisse geben Hinweise auf die Qualität von Programmteilen.
- Decken wir einen Fehler durch Testen auf, kennen wir noch keine Fehlerursachen (häufig mehrere). Die Beseitigung einer scheinbaren Ursache bedeutet nicht, dass der Fehler beseitigt ist, da die Änderung zu neuen Fehlern führen kann – gängiger Spruch: „Ein Fehler ausgebessert, ein (bzw. zehn oder mehr) Fehler eingeführt."
- Es ist oft nicht klar, was ein Fehler ist. Ein aus einem bestimmten Betrachtungswinkel ungewöhnliches Verhalten der Software kann aus einem anderen Blickwinkel gewollt sein.
- Fehler aus einem den Design-Rules entsprechenden Blickwinkel können bewusst eingebaut worden sein um aus einem anderen Blickwinkel Vorteile zu erhalten. Beispielsweise werden bewusst Löcher in eine Passwortabfrage eingebaut um das automatisierte Testen (ohne Passworteingabe) zu erleichtern.
- Ein Fehler, der über die ganze Lebenszeit eines Programms unentdeckt bleibt, wirkt sich nicht störend aus. Allerdings wissen wir nie, ob der Fehler vielleicht nicht doch noch aufgedeckt wird.
- Fehler wirken sich ganz unterschiedlich aus, von harmlos bis katastrophal, manchmal sogar tödlich. Die Beurteilung der Schwere ist subjektiv. Beispielsweise wirkt eine vergessene Überprüfung von Eingaben harmlos, bis ein Angreifer diese Lücke zum Eindringen in das System nutzt. Gerade Fehler, die äußerst selten Auswirkungen zeigen, werden bewusst genutzt um in Systeme einzudringen und stellen damit eine besonders große Gefahr dar.

Der Grund, warum das wiederholte Korrigieren von Fehlern bei Programmen aus Übungsaufgaben scheinbar rasch zu korrekten Programmen führt, liegt in einer verzerrten Wahrnehmung: Die ersten Fehler, die gefunden werden, haben häufig triviale Ursachen, die sich ohne Schwierigkeiten finden und beseitigen lassen. Meist geben wir uns mit wenigen einfachen Testfällen zufrieden. Werden diese Testfälle richtig ausgeführt, haben wir eine für die Lehrveranstaltung ausreichende Programmqualität erreicht. In realen Programmen ist diese Qualität jedoch nicht ausreichend. Wir müssen das Programm weiter testen und werden zusätzliche Fehler finden, deren Ursachen nicht mehr leicht festzustellen sind. Je besser versteckt die Fehler sind, desto schwieriger wird die Beseitigung. Die Schwierigkeiten bei der Fehlerbeseitigung nehmen meist in größerem Ausmaß zu als die Schwierigkeiten beim Aufdecken der Fehler. Sobald ein bestimmter Schwierigkeitsgrad erreicht wurde, gelten die Tatsachen aus obiger Liste. Wir dürfen dann auf keinen Fall mehr so wie von kleinen Übungsprogrammen gewohnt vorgehen, sonst würde die Zahl der Fehler mit der Zeit steigen, nicht sinken.

Es braucht eine gezielte Vorgehensweise, ein *Qualitätsmanagement* um die Qualität durch Testen und Debuggen weiter verbessern zu können. Maßnahmen aus folgender unvollständigen Liste haben sich (in unterschiedlichen Kombinationen miteinander) bewährt:

- Testfälle werden nach einer Strategie festgelegt. Unterschiedliche Strategien haben unterschiedliche Schwerpunktsetzungen, aber jede Strategie generiert viel mehr und komplexere Testfälle als einfaches Ausprobieren. Diese Strategien sind am bekanntesten: Qualität durch Testen
  - *Black-Box-Testen* leitet Testfälle aus Anwendungsfällen ab. Testfälle und Programm entstehen unabhängig voneinander. Das kann auch Fehler in Situationen aufdecken, die bei der Programmentwicklung nicht bedacht wurden.
  - *White-Box-Testen* leitet Testfälle aus der Implementierung ab. Meist wird versucht, zumindest einen Testfall für jeden möglichen Programmpfad einzuführen. Damit wird eine gute *Coverage* aller Programmteile erreicht.

  - *Grey-Box-Testen* legt Testfälle schon vor der Implementierung als Spezifikation des Programms fest. Programme orientieren sich an den Testfällen. Ihr Charakter liegt häufig zwischen denen von Black-Box- und White-Box-Testen.

- Die Überprüfung der Testfälle wird automatisiert: Eigene Programmteile wenden Testfälle auf die zu testende Software an und überprüfen Ergebnisse. Dafür gibt es Werkzeuge, aber es geht auch ohne sie recht gut. Die Automatisierung ermöglicht *Regressions-Tests:* Nach einer Änderung werden die gleichen Testfälle noch einmal überprüft um festzustellen, ob es auch unbeabsichtigte Auswirkungen gibt. Regressions-Tests erhöhen die Wahrscheinlichkeit für das frühe Erkennen von Fehlern, die durch Änderungen eingeführt werden.

- Fehler werden nach der Dringlichkeit ihrer Behebung kategorisiert. Fehler, die ein hohes Sicherheitsrisiko darstellen, müssen sehr rasch beseitigt werden. Es folgen Fehler, die die Software für viele Anwenderinnen und Anwender unbenutzbar machen, sowie Fehler, die zu einem Sicherheitsrisiko werden könnten. Erst danach kommen alle übrigen Fehler. Die Einteilung erfolgt meist in Teambesprechungen, wobei vorher zu klären ist, ob ein Fehler im Programm oder beim Testen vorliegt.

- Fehler bei Sicherheit und Benutzbarkeit werden durch Testen der Funktionalität kaum aufgedeckt. Es muss gezielt danach gesucht werden. Neben unverzichtbaren Code-Reviews wird bei *Sicherheitstests* versucht, illegal in die Software einzudringen. Auch *Belastungstests* werden durchgeführt, häufig als *Stresstests* (wobei in jedem Aspekt gleichzeitig die schlechtestmögliche Annahme getroffen wird) und *Crashtests* (wobei versucht wird, das System funktionsunfähig zu machen oder zum Absturz zu bringen). *Benutzeroberflächentests* prüfen die Benutzbarkeit der Software.

- Das Finden von Fehlern ist einfach im Vergleich zum Erkennen der Ursachen. In fortgeschrittenen Entwicklungsstadien darf die Fehlerbehebung erst stattfinden, nachdem eine ausführliche Ursachenforschung betrieben wurde. Ein Vorgehen nach dem Trial-and-Error-Prinzip könnte die Qualität erheblich verschlechtern.

- Die Fehlerbeseitigung ist teuer. Je später ein Fehler gefunden wird, desto höher sind die Kosten. Daher wird mit dem Testen

früh begonnen. Am Beginn stehen *Modul-* bzw. *Komponententests,* bei denen Programmteile, etwa Klassen, unabhängig vom ganzen Programm getestet werden. *Schnittstellentests* überprüfen danach, ob mehrere Programmteile zusammenspielen. *Integrationstests* testen das Zusammenspiel aller Teile in einer realitätsnahen Umgebung, *Systemtests* in der realen Anwendung. Ein Systemtest heißt auch *Abnahmetest,* wenn dabei vorher vertraglich festgelegte Qualitätseigenschaften überprüft werden.

- Testergebnisse geben Hinweise auf fehleranfällige Programmstellen. Besonders fehlerträchtige Stellen müssen generell überarbeitet oder ausgetauscht werden statt einzelne Fehler zu korrigieren. Im Vorfeld sind Code-Reviews zur Ursachenforschung nötig.

- Zum Auffinden von Fehlerursachen kommen viele Hilfsmittel zum Einsatz, beispielsweise folgende:
  - Code-Review (unverzichtbar, egal was sonst gemacht wird),
  - Debugger (nur in einfachen Fällen geeignet),
  - Erstellen und Analysieren von Log-Dateien und Traces,
  - Einbeziehen von Personen, die Programmteile entwickelten,
  - Einbeziehen von Personen mit Expertenwissen auf besonders relevanten Gebieten wie Softwaresicherheit,
  - Einbeziehen von Personen, die die Software intensiv verwenden oder spezielles Domain-Wissen haben.

- Hinweise auf Fehler während der praktischen Anwendung sowie erkannte Fehlerursachen werden dahingehend analysiert, ob zusätzliche Testfälle eingeführt werden müssen.

- Die Qualität des Testens wird überprüft, indem absichtlich Fehler in Programmtexte eingeschleust werden und die Zuverlässigkeit der Erkennung dieser Fehler untersucht wird.

Im Bereich der Sicherheit kann gar nicht genug getestet werden. Es herrscht ein ständiger ungleicher Wettlauf gegen Angreifer, da Angreifer nur eine Lücke zum Eindringen benötigen, während wir jede mögliche Lücke schließen müssen. Wir müssen z. B. auf Folgendes achten:

- Eingegebene Daten dürfen nicht (unüberprüft) ausgeführt werden. Zum Beispiel darf ein eingegebener Name nicht ungefiltert in

Sicherheit

die SQL-Abfrage einer Datenbank eingebettet werden, da der Name aus Sicht der Datenbank etwas ganz anderes bedeuten könnte, etwa wenn statt des Namens ein komplexer SQL-Ausdruck eingegeben wurde, der Daten zerstört oder sichtbar macht (*SQL-Injection*).

- *Pufferüberläufe* bedeuten große Sicherheitsrisiken: Sind geschriebene Daten größer als dafür vorgesehene Speicherbereiche, überschreiben sie andere Speicherbereiche und beeinflussen so die Programmausführung. Das passiert z. B. bei Arrayzugriffen außerhalb erlaubter Indexgrenzen. Java überprüft Indexgrenzen zur Verringerung der Gefahr, aber auch die Prüfung kann ausgeschaltet oder fehlerhaft sein. Viel häufiger ist es jedoch so, dass die Arraygrenzen, die Java kennt und überprüft, nicht die gleichen sind, die aus Anwendersicht gelten. Das passiert beispielsweise, wenn Daten unterschiedlicher Art im gleichen Array liegen oder ein bereits bestehendes Array für einen anderen Zweck wiederverwendet wird. In solchen Fällen werden Programmierfehler zu einem Sicherheitsrisiko.

- Nur Programme höherer Sicherheitsstufen dürfen auf kritische Ressourcen zugreifen. Oft wird auch wenig vertrauenswürdigen Programmen die für sie nötige Sicherheitsstufe zuerkannt. Über solche Programme können sich weitere Programme Zugriff auf geschützte Daten verschaffen (*Privilege-Escalation*).

Aufgabe 4.37 Beschreiben Sie, warum ein Fehler in einem Programm, der in der Praxis nur mit extrem kleiner Wahrscheinlichkeit auftritt, dennoch schwerwiegende Folgen nach sich ziehen kann.

Aufgabe 4.38 Beschreiben Sie, nach welchen Strategien Testfälle festgelegt werden können und welche typischen Eigenschaften derart entstandene Testfälle haben.

Aufgabe 4.39 Beschreiben Sie, was man unter Regressions-Tests versteht, wie man dabei vorgeht und wozu sie benötigt werden.

Aufgabe 4.40 Beschreiben Sie die Begriffe Sicherheitstest, Belastungstest, Stresstest und Crashtest zusammen mit ihren Zielen.

Aufgabe 4.41 Beschreiben Sie die Begriffe Modul- bzw. Komponententest, Schnittstellentest, Integrationstest und Systemtest. Wozu dient die Unterscheidung zwischen diesen Testarten?

Aufgabe 4.42 Beschreiben Sie beispielhaft, wie Sie nach Aufdecken eines Fehlers vorgehen können um Ursachen dieses Fehlers zu finden.

Aufgabe 4.43 Beschreiben Sie Zusammenhänge zwischen Testen (bzw. Konsequenzen aus Testergebnissen) und Code-Reviews.

Aufgabe 4.44 Beschreiben Sie beispielhaft, worauf Sie beim Testen und in Code-Reviews achten müssen um Lücken zum Eindringen in die Software zu erkennen.

4.3.2 Für Qualität sorgen

Der intuitive Ansatz, die Programmqualität alleine durch ausgiebiges Testen und Korrigieren gefundener Fehler verbessern zu wollen, ist ab einem gewissen Punkt nicht mehr zielführend. Es stellt sich also die Frage, wie wir die Qualität sonst noch verbessern können. Zuvor müssen wir jedoch klären, was Qualität in der Programmierung überhaupt bedeutet. Das sind einige der vielen Kriterien:

Qualitätskriterien

- Das Programm ist möglichst gut *brauchbar:*
  - *Aufgabenerfüllung:* Nur die eigentlichen Aufgaben zählen; unnötige Programmeigenschaften bringen keine Vorteile.
  - *Bedienbarkeit:* Dazu gehört auch eine verständliche Sprache und intuitive Menüführung sowie ein flüssiger Ablauf.
  - *Konfigurierbarkeit:* Programme werden an die Bedürfnisse von Menschen angepasst, nicht umgekehrt.
  - *Attraktivität:* Die Anwendung macht Spaß und/oder liegt im Trend (Modeerscheinung, sozialer Druck, etc.).

- Das Programm funktioniert *zuverlässig* und *sicher:*
  - Ergebnisse sind korrekt und unmissverständlich.
  - Die Integrität der Daten bleibt stets erhalten.
  - Kann die Aufgabe ausnahmsweise nicht erfüllt werden, wird darauf klar hingewiesen.
  - Ausfälle kommen selten vor und sind rasch behoben.
  - Der Umgang mit dem Programm ist intuitiv.
  - Antwortzeiten liegen im erwarteten Bereich (wobei die Erwartung von der technischen Komplexität abhängt).

  - Es wird nichts Unerwartetes gemacht.
  - Es werden keine Daten an Unberechtigte weitergegeben oder von Unberechtigten entgegengenommen.
  - Erkannte Sicherheitslücken werden rasch geschlossen.
- Das Programm und die Programmierung laufen *effizient* ab:
  - Ressourcen wie Rechenzeit, Speicherbedarf, Energie, Netzwerkbelastung, etc. werden sparsam eingesetzt.
  - Andere Programme werden möglichst wenig gestört.
  - Interaktionen mit dem Programm sind auf das unbedingt nötige Ausmaß reduziert.
  - Entwicklungszeiten bleiben im vertretbaren Rahmen.
- Das Programm ist gut *wartbar:*
  - Programmtexte sind so einfach wie möglich.
  - Programmtexte sind gut lesbar und Softwareverträge klar.
  - Nötige Änderungen sind einfach durchführbar.
  - Abhängigkeiten zu anderer Software bleiben schwach.
  - Entwicklungsteams stehen auch für die Wartung bereit.

Natürlich möchten wir jedes Kriterium gut erfüllen. Allerdings stehen einige Kriterien in Widerspruch zueinander, und manche sind mit extrem hohem Aufwand verbunden. Wir müssen Kompromisse finden. Ein guter Kompromiss hängt von der Art des Programms ab. Beispielsweise wird von einer Software zur Steuerung eines Kraftwerks oder autonom fahrenden Fahrzeugs hohe Zuverlässigkeit und Sicherheit erwartet, während der Spaß an der Bedienung nicht wichtig ist. Von einem Computerspiel wird dagegen vor allem Attraktivität erwartet.

Tatsächlich ist eine lange Liste von Maßnahmen vorstellbar, die dabei helfen, eine gute Qualität der Programme zu erreichen:

Verbesserung der Qualität

- Ganz oben auf der Liste steht das *Bewusstsein* für Qualität bei der Programmentwicklung, das unter Anderem durch Design-Rules, Schulungen und Zertifizierungen gefördert wird. Alleine schon das Wissen darüber, dass Qualität viele Facetten hat, reicht aus um sich während der Entwicklung mit dem Thema zu beschäftigen und die Qualität zu steigern. Im Vergleich zum Korrigieren von Fehlern ist jede Investition in die Steigerung des Qualitätsbewusstseins kostengünstig und effektiv.

- So wie funktionale Anforderungen an eine zu entwickelnde Software im Vorfeld möglichst gut analysiert und dargestellt werden sollen, so sind auch Anforderungen an die Qualität im Rahmen der *Anforderungsanalyse* zu analysieren und darzustellen. Wenn wir bei der Programmerstellung wissen, auf welche Qualitätskriterien es am ehesten ankommt, haben wir eine reelle Chance, dafür zu sorgen, dass diese Kriterien auch erfüllt werden. Wenn im Gegensatz dazu alle Kriterien als gleich wichtig gelten, können wir uns nicht auf das Wesentliche konzentrieren und werden wegen beschränkter Entwicklungszeiten und widersprüchlicher Ziele höchstens zufällig eine brauchbare Qualität zustandebringen.

- Nach den Qualitätskriterien wird eine Liste von Coding-Rules erstellt, deren Einhaltung mittels *Code-Reviews* überprüft wird.

- Die ständige Kontrolle wird in den gesamten Entwicklungsprozess eingebaut, beispielsweise in Form von *Pair-Programming.* Dabei sitzen zwei Personen vor dem Bildschirm, eine die den Programmtext schreibt und eine andere, die Lösungsansätze strategisch durchdenkt und nebenbei Änderungen kontrolliert.

- Mitglieder des Entwicklungsteams treffen sich regelmäßig (etwa zu einem fixen wöchentlichen Termin) um aktuelle Probleme vor allem hinsichtlich der Qualität zu *besprechen.*

- Besprechungen und Konzepte wie Pair-Programming eignen sich auch sehr gut dazu, neue Teammitglieder intensiv und spezifisch auf das Projekt bezogen *einzuschulen.*

- Es gibt eine Vielzahl an *Analyse-Werkzeugen,* die Programmtexte automatisiert anhand von Software-Modellen analysieren, bewerten und Verbesserungen vorschlagen. Jedes ist ein automatisiertes Analogon zur Code-Review, das zwar sehr wertvoll sein kann, aber meist nicht das Gespür menschlicher Experten aufweist.

- Wenn bestimmte Qualitätskriterien vor allem im Bereich der Zuverlässigkeit und Sicherheit von sehr großer Bedeutung sind, wird man nicht um *formale Korrektheitsbeweise* herumkommen. Es gibt Werkzeuge zur automatischen Beweisführung. Damit können standardmäßige Überprüfungen (z. B. zum Nachweis der Deadlockfreiheit) mit relativ geringem Aufwand durchgeführt werden.

Dennoch verursachen formale Korrektheitsbeweise im Allgemeinen einen sehr hohen Aufwand. Einerseits müssen überprüfbare Kriterien formal beschrieben werden, andererseits muss der Programmtext so gestaltet sein, dass die Beweisführung gelingt. Häufig wird das Werkzeug einen Widerspruch finden, wobei unklar ist, ob er einen Fehler im Programm darstellt oder auf eine ungeeignete Beschreibung der Kriterien zurückzuführen ist. Oft liefern diese Werkzeuge kein Ergebnis innerhalb eines vernünftigen Zeitrahmens, sodass das Programm vereinfacht werden muss.

- Mitglieder eines Entwicklungsteams müssen sich hundertprozentig aufeinander verlassen können. Konflikte und Missverständnisse im Team führen fast sicher zu inkonsistenten Schnittstellen und schwerwiegenden Qualitätsmängeln. Neben *teambildenden Maßnahmen* sind auch rigorose Vorgehensweisen wie *Design-by-Contract* hilfreich beim frühen Erkennen und Beseitigen von Konflikten: Softwareverträge machen klar, wer wofür zuständig ist und wo welche Überprüfungen vorgenommen werden müssen. Während es beispielsweise absolut notwendig ist, dass alle von außen kommenden Daten genauestens überprüft werden, führt es zu einem Desaster, wenn Daten zigfach auf inkonsistente Weise überprüft werden weil Teammitglieder einander misstrauen.

- *Qualitätsmanagement* bedeutet auch, dass es für Teammitglieder vorteilhaft ist, ein waches Auge auf die Qualität zu richten und mögliche Mängel sofort aufzuzeigen. Aus Erfahrungen müssen Lehren gezogen und Entwicklungsprozesse angepasst werden. Wiederholte Anpassungen führen mit der Zeit zu einem funktionierenden, auf Qualität ausgerichteten Entwicklungsprozess.

Aufgabe 4.45 Zählen Sie Qualitätskriterien in der Programmierung auf. Beschreiben Sie beispielhaft, wie wichtig diese Kriterien für bestimmte Arten von Programmen jeweils sind.

Aufgabe 4.46 Zählen Sie einige Maßnahmen zur Verbesserung der Qualität von Programmen (abseits vom Testen) auf und beschreiben Sie, wodurch diese Maßnahmen die Qualität verbessern.

4.3.3 Optimieren

Es liegt in der Natur von Menschen insbesondere in der Softwareentwicklung, Aufgaben auf optimale Weise lösen zu wollen. Das zeugt von

einem hohen Qualitätsanspruch. Dennoch wird das Optimieren in der Programmierung als ungünstig angesehen, was beispielsweise durch folgende oft gehörte Empfehlungen zum Ausdruck kommt:

Optimierung negativ

- Für Nichtexperten: „Don't do it."
- Für Optimierungsexperten: „Don't do it now."[4]

Offensichtlich unterscheiden sich die Begriffe Qualität und Optimierung in ihrer Zielsetzung essenziell voneinander.

In der Regel bezeichnen wir Verbesserungen, die ein Compiler oder Laufzeitsystem zur Steigerung der Laufzeit-, Speicher- oder Energieeffizienz vornimmt, als Optimierungen. Derartige Optimierungen sind natürlich erwünscht, solange sie korrekt sind, also die Semantik des Programms unverändert lassen. Heute werden Compilertechniken mit ausgefeilten, aggressiven Optimierungen eingesetzt. Wenn wir beim Programmieren von Optimierungen sprechen, meinen wir Verbesserungen am Programmtext, die ebenfalls Steigerungen der Laufzeit-, Speicher- oder Energieeffizienz zum Ziel haben. Allerdings stehen wir in Konkurrenz zu den Compilertechniken. Tatsächlich kann eine scheinbare Optimierung am Programmtext die Programmqualität verringern:

was gegen Optimierung spricht

- Das Ziel der Optimierung liegt ausschließlich in einem effizienteren Umgang mit Ressourcen zur Laufzeit. Darunter leiden andere Qualitätskriterien, vor allem die Lesbarkeit und Einfachheit.
- Ohne Laufzeitvergleiche nur auf Verdacht hin durchgeführte Optimierungsversuche sind sinnlos. Häufig würde der Compiler ohne unser Zutun eine vergleichbare oder effektivere Optimierung anwenden, und manchmal machen unsere Änderungen Optimierungen durch den Compiler sogar unmöglich. Nur durch rigorose Laufzeitvergleiche und mit viel Wissen über Compilertechniken können wir Programmtexte so optimieren, dass darauf folgende Optimierungen des Compilers unterstützt werden.
- Jede noch so kleine Programmänderung kann frühere Optimierungen hinfällig machen oder in ihr Gegenteil verkehren.
- Da Optimierungen meist die Einfachheit und Verständlichkeit verschlechtern, führen sie auch häufig zu schwerwiegenden Fehlern. Insbesondere vergessen wir leicht auf Spezialfälle.

[4] Dabei wird selbstverständlich davon ausgegangen, dass komplexere Programme ständig verändert werden und daher niemals der richtige Zeitpunkt kommt.

Angesichts dieser Tatsachen ist klar, dass es nicht angebracht ist, Programmtexte in Konkurrenz zum Compiler optimieren zu wollen.

Nicht in Konkurrenz zu Compilertechniken stehend verstanden, liefern Optimierungen des Programmtexts dennoch einen wertvollen Beitrag zur Steigerung der Programmqualität. Das sind einige der vielen möglichen sinnvollen Optimierungsmaßnahmen:

sinnvolle Optimierungsmaßnahmen

- An erster Stelle steht die Auswahl geeigneter Algorithmen und Datenstrukturen. Ein gut gewählter Algorithmus kann im Vergleich zu einem schlecht gewählten einen gewaltigen Unterschied in der Programmeffizienz ausmachen. Optimierungen durch den Compiler sind dagegen fast vernachlässigbar.

- Wie jede Sprache hat auch Java Tücken, durch die scheinbar einfache Aufgaben komplexe Lösungen erfordern. Beispielsweise können wir nur umständlich erreichen, dass eine Methode mehrere Werte zurückgibt. Mit Geschick lassen sich komplexe Lösungen häufig vermeiden, etwa indem Methoden so gestaltet werden, dass mehrere Ergebniswerte nicht oder nur selten nötig sind.

- Data-Hiding dient in erster Linie der Datenabstraktion. Auch aus Sicht der Programmeffizienz ist es vorteilhaft, wenn alle Objektvariablen und möglichst viele Methoden `private` sind. Das gibt dem Compiler zusätzliche Ansatzpunkte für Optimierungen.

- Gelegentlich müssen wir über mehrere Ecken denken. Zunächst scheint die Verwendung von Interfaces die Programmeffizienz in geringem Ausmaß zu verringern. Allerdings ermöglichen Abstraktionshierarchien einen einfachen Austausch einer Datenstruktur gegen eine andere. Damit können wir mehrere Datenstrukturen ausprobieren und jene mit der höchsten Programmeffizienz verwenden. Auch ein späterer Tausch ist leicht durchführbar, wenn eine effizientere Datenstruktur verfügbar wird. So stellt sich der kleine Nachteil als großer Vorteil bezüglich der Effizienz heraus.

Optimierungen können die Programmeffizienz stark verbessern. Allerdings dürfen wir Optimierungen nur so einsetzen, dass sie andere Qualitätskriterien nicht mindern sondern im besten Fall unterstützen. Auf keinen Fall dürfen wir „kleinkariert“ denken, also alle minimalen Effizienzvorteile nutzen wollen und dabei die großen Brocken übersehen.

Das Wort „Optimierung“ suggeriert, dass das Ergebnis der Optimierung optimal ist. Optimalität würde ein Modell voraussetzen, das

Programme hinsichtlich der Effizienz vergleichbar macht und erkennen lässt, welches Programm das beste ist. Es ist kaum möglich, ein solches Modell zu finden. Die Laufzeiteffizienz müsste im Vergleich zur Speicher- und Energieeffizienz gewichtet sein. Außerdem können mehrere Ausführungen desselben Programms je nach Daten und Umgebung ganz unterschiedliche Effizienz zeigen. Tatsächlich haben Optimierungen also nichts mit Optimalität zu tun. „Optimieren“ wird eher in einem umgangssprachlichen Sinn verwendet, als Prozess, in dem schrittweise Änderungen in Richtung einer Verbesserung der Effizienz ausprobiert werden. Auf diese Weise gehen wir in der Praxis meist vor.

Programmoptimierungen sind nur sinnvoll, wenn sich zeigt, dass die geforderte Programmqualität hinsichtlich der Programmeffizienz noch nicht erreicht wurde. Dann muss nachgebessert werden. In diesem Szenario wissen wir, welche konkreten Ziele erreicht werden sollen und was konkret nachgebessert werden muss. Das gibt die Richtung für Verbesserungen vor. Nun führen wir folgende Schritte durch:

Schritte zur Optimierung

1. Wir analysieren, welche Programmteile wie stark zur unzureichenden Effizienz beitragen. Oft kommen dafür ohnehin nur wenige Programmstellen in Frage, manchmal sind aufwendigere Analysen wie Laufzeitmessungen von Programmteilen nötig.

2. Wir wählen einen gefundenen Programmteil, den mit dem größten Optimierungspotential, und versuchen dessen Effizienz durch eine der obigen Maßnahmen zu verbessern, in erster Linie durch den Austausch von Algorithmen und Datenstrukturen.

3. Wir überprüfen die Auswirkungen. Reicht die Effizienz, sind wir fertig. Sonst setzen wir mit Schritt 1 fort.

Diese Vorgehensweise wirkt einfach und selbstverständlich, lässt aber auch die Gefahr dahinter erkennen: Nach wenigen Iterationen ist ein Großteil des Optimierungspotentials ausgeschöpft. Jede zusätzliche Iteration ist sehr aufwendig, leistet aber, wenn überhaupt, nur einen sehr kleinen Beitrag zur weiteren Steigerung der Programmeffizienz. Irgendwann müssen wir uns eingestehen, dass wir die geforderte Programmqualität nicht erreichen können. Dann müssen wir den generellen Aufbau des Programms sowie die geforderten Qualitätskriterien neu überdenken, was wiederum einen gewaltigen Aufwand nach sich zieht. Übertriebene Forderungen an die Programmeffizienz sind daher gefährlich.

Potential begrenzt

Aufgabe 4.47 Beschreiben Sie, was man unter der Optimierung von Programmen versteht.

Aufgabe 4.48 Beschreiben Sie, warum viele Versuche zur Optimierung von Programmen die Programmqualität verschlechtern und welche Optimierungsmaßnahmen die Qualität verbessern können.

Aufgabe 4.49 Beschreiben Sie, wie man zur Programmoptimierung schrittweise vorgehen kann und welche Gefahr(en) diese Vorgehensweise in sich birgt.