Informatics

Franz Puntigam

Programmier-paradigmen

2023/2024

Skriptum zur Lehrveranstaltung 194.023 an der TU Wien (Vorabversion September)

Inhaltsverzeichnis

1 Grundlagen und Zielsetzungen **9**

1.1 Berechnungsmodell 10

1.1.1 Formaler Hintergrund 10

1.1.2 Lambda-Kalkül 13

1.1.3 Praktische Realisierung 16

1.2 Evolution und feine Programmstrukturen 19

1.2.1 Widersprüche und typische Entwicklungen 19

1.2.2 Strukturierte Programmierung 22

1.2.3 Programmteile als Daten 25

1.3 Programmorganisation 28

1.3.1 Modularisierung 28

1.3.2 Parametrisierung 33

1.3.3 Ersetzbarkeit 36

1.4 Abstraktion 38

1.4.1 Prinzip der Abstraktion 38

1.4.2 Datenabstraktion 42

1.4.3 Abstraktionshierarchien 45

1.5 Daten und Datenfluss 47

1.5.1 Kommunikation über Variablen 47

1.5.2 Wiedererlangung der Kontrolle 52

1.5.3 Verteilung der Daten 54

1.6 Typisierung 57

1.6.1 Typkonsistenz, Verständlichkeit und Planbarkeit 57

1.6.2 Nominale und strukturelle Typen 61

1.6.3 Gestaltungsspielraum 63

2 Etablierte Denkmuster und Werkzeugkisten **67**

2.1 Prozedurale Programmierung 67

2.1.1 Alles im Griff 67

2.1.2 Totgesagte leben länger 71

2.2 Objektorientierte Programmierung 74

2.2.1 Modularisierungseinheiten und Abstraktion 74

2.2.2 Polymorphismus und Ersetzbarkeit 77

2.2.3 Typische Vorgehensweisen 79

2.2.4 Worum es geht 82

2.3 Funktionale Programmierung 85

2.3.1 Sauberkeit aus Prinzip 85

2.3.2 Streben nach höherer Ordnung 90

2.3.3 Friedliche Koexistenz 96

2.4 Parallele Programmierung 100
2.4.1 Parallelität und Ressourcenverbrauch 100
2.4.2 Formen der Parallelität 103
2.4.3 Streben nach Unabhängigkeit 108
2.5 Nebenläufigkeit 111
2.5.1 Threads und Mutual-Exclusion 112
2.5.2 Warten bis es passiert 116

3 Ersetzbarkeit und Untertypen 121
3.1 Ersetzbarkeitsprinzip 121
3.1.1 Untertypen und Schnittstellen 121
3.1.2 Untertypen und Codewiederverwendung 126
3.1.3 Dynamisches Binden 129
3.2 Ersetzbarkeit und Objektverhalten 132
3.2.1 Client-Server-Beziehungen 132
3.2.2 Untertypen und Verhalten 139
3.2.3 Abstrakte Klassen 143
3.3 Vererbung versus Ersetzbarkeit 145
3.3.1 Reale Welt, Vererbung, Ersetzbarkeit 145
3.3.2 Vererbung und Codewiederverwendung 147
3.3.3 Fehlervermeidung 152
3.4 Klassen und Vererbung in Java 154
3.4.1 Klassen in Java 154
3.4.2 Vererbung und Interfaces in Java 157
3.4.3 Pakete und Zugriffskontrolle in Java 160
3.5 Ausnahmebehandlungen und Ersetzbarkeit 164
3.5.1 Ausnahmebehandlung in Java 164
3.5.2 Einsatz von Ausnahmebehandlungen 167
3.5.3 Zusicherungen und Ausnahmen 171

4 Dynamische Typinformation und statische Parametrisierung 175
4.1 Generizität 175
4.1.1 Wozu Generizität? 175
4.1.2 Einfache Generizität in Java 176
4.1.3 Gebundene Generizität in Java 180
4.2 Verwendung von Generizität 185
4.2.1 Richtlinien für die Verwendung von Generizität 186
4.2.2 Arten der Generizität 190
4.3 Typabfragen und Typumwandlungen 193
4.3.1 Verwendung dynamischer Typinformation 193
4.3.2 Typumwandlungen und Generizität 197
4.3.3 Kovariante Probleme 202
4.4 Überladene Methoden und Multimethoden 206
4.4.1 Deklarierte versus dynamische Argumenttypen 207
4.4.2 Simulation von Multimethoden 210

4.5 Annotationen und Reflexion 212
4.5.1 Annotationen und Reflexion in Java 212
4.5.2 Anwendungen von Annotationen und Reflexion 216
4.6 Aspektorientierte Programmierung 219
4.6.1 Konzeptuelle Sichtweise 219
4.6.2 AspectJ 223

5 Applikative Programmierung und Parallelausführung 229
5.1 Lambdas und Java-8-Streams 229
5.1.1 Anonyme innere Klassen und Lambdas 229
5.1.2 Java-8-Streams 233
5.1.3 Applikative Programmierung in der Praxis 239
5.2 Funktionen höherer Ordnung 243
5.2.1 Nachbildung typischer Kontrollstrukturen 243
5.2.2 Funktionale Elemente in Java 248
5.3 Nebenläufige Programmierung in Java 254
5.3.1 Thread-Erzeugung und Synchonisation in Java 254
5.3.2 Nebenläufigkeit in der Praxis 257
5.3.3 Synchronisation und die objektorientierte Sicht 261
5.4 Prozesse und Interprozesskommunikation 264
5.4.1 Erzeugen von Prozessen in einer Shell 264
5.4.2 Umgang mit Dateien und I/O-Strömen in Java 269
5.4.3 Beispiel zu parallelen Prozessen 274

6 Entwurfsmuster und Entscheidungshilfen 281
6.1 Grundsätzliches und Muster für Verhalten 281
6.1.1 Aufbau von Entwurfsmustern 282
6.1.2 Visitor 284
6.1.3 Iterator 287
6.1.4 Template-Method 290
6.2 Erzeugende Entwurfsmuster 292
6.2.1 Factory Method 292
6.2.2 Prototype 295
6.2.3 Singleton 298
6.3 Entwurfsmuster für Struktur 301
6.3.1 Decorator 301
6.3.2 Proxy 304
6.4 Entscheidungshilfen 306
6.4.1 Entwurfsmuster als Entscheidungshilfen 306
6.4.2 Richtlinien in der Entscheidungsfindung 309

Vorwort

Programmierparadigmen ist eine Lehrveranstaltung für Studierende der Informatik an der TU Wien. Neben einem generellen Überblick über verschiedene Paradigmen in der Programmierung sowie deren Zielsetzungen und Bedeutungen in der Softwareentwicklung liegen Schwerpunkte auf der objektorientierten, funktionalen, nebenläufigen (engl. concurrent) und parallelen Programmierung mit einigen dafür typischen Techniken und Denkmustern. Konzepte für Modularisierung, Parametrisierung, Ersetzbarkeit, Typisierung und den Umgang mit Abhängigkeiten zwischen Daten werden angesprochen. Als Programmiersprache wird einheitlich Java verwendet, einerseits weil die Programmierung in Java als bekannt vorausgesetzt werden kann, andererseits um Gemeinsamkeiten und Unterschiede zwischen den Paradigmen deutlicher erkennen zu können, als dies mit unterschiedlichen, besser an die jeweiligen Paradigmen angepassten Sprachen möglich wäre.

Neal Ford schreibt in [11]: „Learning a new programming language is easy. . . . But learning a new paradigm is difficult – you must learn to see different solutions to familiar problems.“ Das trifft wohl auf alle Programmierparadigmen zu. Diese Sätze verdeutlichen, warum es sinnvoll ist, viel Aufwand in das Erlernen von Paradigmen zu stecken, ohne für jedes Paradigma eine eigene Sprache zu verwenden: Paradigmen beeinflussen die Art unseres Denkens beim Programmieren ganz wesentlich. Wenn wir in mehreren Paradigmen denken, erkennen wir ohne Schwierigkeiten ganz unterschiedliche Lösungswege für eine gegebene Problemstellung und können den erfolgversprechendsten Weg wählen. Damit können wir effizienter und in höherer Qualität programmieren.

Das vorliegende Skriptum wurde als Lehrbuch zur Begleitung der Lehrveranstaltung konzipiert. Durch das Studium des Skriptums alleine sind die Lernziele nicht erreichbar. Es ist auch die Lösung der umfangreichen, fast wöchentlich herausgegebenen Programmieraufgaben in Kleingruppen notwendig.

Die Lehrveranstaltung ist als Teil der Programmier- und Programmiersprachausbildung an der TU Wien anzusehen, die auf die vorangegangenen Lehrveranstaltungen *Einführung in die Programmierung 1* und *2* aufbaut. Entsprechende Vorkenntnisse werden vorausgesetzt: Es wird erwartet, dass Studierende vor der Teilnahme an der Lehrveranstaltung in der Lage sind,

- systematische Vorgehensweisen beim Programmieren und wichtige Konzepte einer aktuellen alltagstauglichen Programmiersprache (vorzugsweise Java) zu beschreiben,
- in natürlicher Sprache beschriebene Programmieraufgaben in ausführbare Programme umzusetzen,

- Übliche (vor allem prozedurale) Vorgehensweisen und Werkzeuge beim Programmieren systematisch anzuwenden,
- in natürlicher Sprache beschriebene Datenabstraktionen, Algorithmen und Datenstrukturen zu implementieren,
- Techniken der objektorientierten Modellierung anzuwenden,
- Programmieraufgaben selbständig zu lösen und ebenso in Zweierteams zusammenzuarbeiten.

Sind alle Voraussetzungen erfüllt, sollten die Lernziele bei intensiver Mitarbeit mit einem Aufwand von etwa 150 Stunden (6 ECTS) zu erreichen sein.

Nach positiver Absolvierung sind Studierende in der Lage

- die wichtigsten Ziele und einige typische Anwendungsbereiche und Techniken in der objektorientierten, funktionalen, nebenläufigen und parallelen Programmierung (Paradigmen) sowie der Modularisierung, Parametrisierung, Ersetzbarkeit und Typisierung (Konzepte) unter Verwendung fachspezifischer Terminologie zu beschreiben,
- diese Paradigmen und Konzepte und einige ihrer Ausprägungen durch ihre wesentlichen Eigenschaften klar voneinander zu unterscheiden,
- ausgewählte, für diese Paradigmen typische Vorgehensweisen und Techniken sowie die genannten Konzepte in kleinen Teams in einer alltagstauglichen Programmiersprache (Java) praktisch anzuwenden,
- in natürlicher Sprache in unterschiedlichen Details (auch unvollständig) beschriebene Programmieraufgaben in ausführbare Programme in einer alltagstauglichen Programmiersprache umzusetzen, die typische Merkmale vorgegebener Programmierstile aufweisen,
- eigene Programme nach vorgegebenen Kriterien kritisch zu beurteilen.

Viel Spaß beim Lesen und Lernen!

1 Grundlagen und Zielsetzungen

Im Zusammenhang mit der Programmierung verstehen wir unter dem Begriff *Paradigma* eine bestimmte Denkweise oder Art der Weltanschauung. Entsprechend entwickeln wir Programme in einem gewissen Stil. Nicht jeder individuelle Programmierstil ist gleich ein eigenes Paradigma, sondern nur solche Stile, die sich grundlegend voneinander unterscheiden. Einander ähnliche Stile mit gemeinsamen charakteristischen Eigenschaften fallen unter dasselbe Paradigma. Diese Eigenschaften betreffen ganz unterschiedliche Aspekte, etwa das zugrundeliegende Berechnungsmodell, die Strukturierung des Programmablaufs oder Datenflusses, die Aufteilung großer Programme in überschaubare Einzelteile und so weiter. Daraus ergibt sich eine Vielzahl an Paradigmen, die sich im Laufe der Zeit entwickelt und eine mehr oder weniger große Verbreitung gefunden haben. In engem Zusammenhang mit dem Erfolg von Paradigmen steht die Verfügbarkeit und Qualität entsprechender Programmiersprachen und Entwicklungswerkzeuge. Nicht selten definiert sich ein Paradigma durch seine Sprachen und Werkzeuge. Entscheidend sind die verfolgten Ziele. Unterschiedliche Zielsetzungen führen zu unterschiedlichen Paradigmen, auch wenn gleiche Sprachen und Werkzeuge eingesetzt und ähnliche Aspekte berücksichtigt werden.

Häufig wird das auf Maschinenbefehlen aufbauende *imperative* vom auf formalen Modellen beruhenden *deklarativen* Paradigma unterschieden. Das imperative Paradigma wird nach der vorherrschenden Programmstrukturierung in das *prozedurale* und *objektorientierte* Paradigma unterteilt, das deklarative nach dem formalen Modell in das *funktionale* und *logikorientierte* Paradigma. Diese etablierte, aber heute nicht mehr hinreichende Einteilung lässt zahlreiche Paradigmen unberücksichtigt, die in der praktischen Softwareentwicklung weit verbreitet oder in bestimmten Nischen von Bedeutung sind, etwa verschiedene Formen der parallelen und verteilten Programmierung. Diese werden (mehr oder weniger willkürlich) einem der etablierten Paradigmen zugeschlagen.

Wir wollen in diesem Kapitel etwas Struktur in die Paradigmenvielfalt bringen, ohne Paradigmen auszuschließen, die nicht in eine althergebrachte Untergliederung passen. Dazu versuchen wir, Schwerpunkte auf Aspekte und Zielsetzungen hinter den Paradigmen zu legen, obwohl sich durch eine solche Einteilung ein nicht immer ganz orthogonales Gefüge ergibt. Auf diese Weise lässt sich der Gestaltungsspielraum entsprechender Sprachen und Werkzeuge am besten abschätzen. Außerdem gibt uns diese Vorgehensweise Gelegenheit, diverse Aspekte und Konzepte der Programmierung auf einer hohen Ebene zu betrachten und zu verschiedenen Zielen in Beziehung zu setzen. Quasi im Vorbeigehen führen wir einige Grundlagen ein, auf denen weitergehende Betrachtungen von Programmierparadigmen aufbauen. Wir sprechen die Bedeutung von Themen an, die in späteren Kapiteln genauer untersucht werden.

1.1 Berechnungsmodell

Hinter jedem Programmierparadigma steckt ein Berechnungsmodell. Berechnungsmodelle haben immer einen formalen Hintergrund. Sie müssen in sich konsistent und in der Regel so mächtig wie die Turing-Maschine sein, also alles ausdrücken können, was als berechenbar gilt. Ein Formalismus eignet sich aber nur dann als Grundlage eines Paradigmas, wenn die praktische Umsetzung ohne übermäßig großen Aufwand zu Programmiersprachen, Entwicklungsmethoden und Werkzeugen hinreichender Qualität führt. Es braucht Experimente und Erfahrungen, um die praktische Eignung festzustellen.

1.1.1 Formaler Hintergrund

Es entstehen ständig neue Theorien, Formalismen, etc., die in einem Zweig der Informatik von Bedeutung sind. Wir stellen eine Auswahl davon kurz vor und betrachten ihren Einfluss auf Programmierparadigmen. Details der Formalismen sind Gegenstand anderer Lehrveranstaltungen.

Funktionen: In fast jedem Programmierparadigma spielen Funktionen oder ähnliche Konzepte eine zentrale Rolle. Es gibt unterschiedliche Formalismen zur Beschreibung von Funktionen [31]. Am einfachsten zu verstehen sind *primitiv-rekursive Funktionen,* die von einer vorgegebenen Menge einfacher Funktionen ausgehen und daraus durch Komposition und Rekursion neue Funktionen bilden, so wie es in der Programmierung häufig gemacht wird. Primitiv-rekursive Funktionen können vieles berechnen, aber nicht alles, was berechenbar ist. Die Mächtigkeit der Turing-Maschine wird durch *μ-rekursive Funktionen* erreicht, wo ein hinzugefügter Fixpunkt-Operator μ angewandt auf partielle Funktionen das kleinste aller möglichen Ergebnisse liefert. Der genau so mächtige *λ-Kalkül* wurde unabhängig von primitiv- und μ-rekursiven Funktionen entwickelt und beschreibt in der untypisierten Variante Funktionen ohne Notwendigkeit für einen speziellen Fixpunkt-Operator.

Historisch gesehen ist nicht ganz klar, wie groß der Einfluss dieser Formalismen, vor allem des λ-Kalküls, auf die Entwicklung der Programmierparadigmen war. Aus praktischer Sicht interessant ist vor allem die Möglichkeit, neue Funktionen einfach aus bestehenden Funktionen zusammenzusetzen. Dazu braucht es keine Theorie. Frühe imperative Programmiersprachen wie Algol enthielten dennoch λ-Ausdrücke. Aber in der damaligen Form haben sie sich nicht bewährt und sind bald verschwunden. John McCarthy, der Entwickler von Lisp und wesentlicher Mitbegründer der funktionalen Programmierung, hatte sich nach eigenen Angaben zuvor kaum mit der Theorie der Funktionen beschäftigt [27]. Erst später wurden Lisp-Dialekte um λ-Ausdrücke erweitert. Moderne funktionale Sprachen verwenden den λ-Kalkül deutlich erkennbar als Basis. Erst in jüngerer Zeit werden vom λ-Kalkül inspirierte sogenannte Lambda-Ausdrücke oder Lambdas auch in der objektorientierten Programmierung verwendet.

Prädikatenlogik: Die Prädikatenlogik ist ein etabliertes, mächtiges mathematisches Werkzeug. Zu Beginn der Informatik wurde automatisches Beweisen in der Prädikatenlogik nicht selten als gleichbedeutend mit künstlicher Intelligenz angesehen. Es zeigte sich, dass *Horn-Klauseln,* eine Teilmenge der Prädikatenlogik, die Basis eines Berechnungsmodells mit der Mächtigkeit der Turing-Maschine bilden. Seit etwa 1965 sind mittels *Resolution* und *Unifikation* mehr oder weniger automatisch Beweise über Horn-Klauseln ableitbar [32]. Daraus entstand in den 1970er-Jahren die Programmiersprache *Prolog* [10]. Bis in die frühen 1990er-Jahre galt die logikorientierte Programmierung als der große Hoffnungsträger zur Lösung unzähliger Probleme, etwa die Überwindung der Kluft zwischen Softwareentwicklung und -anwendung, die Automatisierung der Programmierung und die Unterstützung verteilter und hochgradig paralleler Hardware. Man sprach von *Programmiersprachen der 5. Generation.* Umfangreiche Forschungsarbeiten führten zwar zu wichtigen Erkenntnissen, aber die viel zu hoch gesteckten Ziele wurden nicht einmal annähernd erreicht. Heute spielt die logikorientierte Programmierung im engeren Sinn nur mehr eine untergeordnete Rolle. Ihr Einfluss ist dennoch stark. So verwenden wir häufig relationale Datenbanken mit logikorientierten Abfragesprachen.

Constraint-Programmierung: Ein Zweig der Programmiersprachen der 5. Generation verwendete Randbedingungen auf Variablen („constraints“) wie etwa „$x < 5$“ zusätzlich zu Bedingungen wie „A oder B ist wahr“. Aktuelle Beweistechniken können Randbedingungen vergleichsweise effizient auflösen. Dieser Zweig hat sich verselbständigt. Constraint-Programmierung ist heute mit funktionalen und imperativen Sprachen kombinierbar, nicht nur mit logikorientierten. Dafür werden vorwiegend fertige Bibliotheken verwendet, die fast überall eingebunden werden können.

Temporale Logik und Petri-Netze: In temporaler Logik sind zeitliche Abhängigkeiten in logischen Ausdrücken recht einfach abbildbar. Beispielsweise wird festgelegt, dass eine Aussage nur vor oder nach einem bestimmten Ereignis gilt. Es gibt mehrere Arten der temporalen Logik. Häufig ist eine temporale Logik die erste Wahl, wenn zeitabhängige Aussagen oder Ereignisse formal zu beschreiben sind, etwa die Synchronisation in nebenläufigen Programmen oder die Steuerung von Maschinen.

In denselben Bereichen sind auch verschiedene Arten von *Petri-Netzen* verwendbar, die auf intuitiv einfach verständlichen Automaten aufbauen. Temporale Logiken sind in der Regel in Petri-Netze umwandelbar und Petri-Netze in temporale Logiken. Während Petri-Netze gute Möglichkeiten zur grafischen Veranschaulichung komplexer zeitlicher Abhängigkeiten bieten, ist die Beweisführung in temporalen Logiken einfacher.

Freie Algebren: Algebra ist ein sehr altes und etabliertes Teilgebiet der Mathematik, das sich mit Eigenschaften von Rechenoperationen befasst. Gleichzeitig ist eine *(universelle) Algebra* auch ein mathematisches Objekt, etwa

eine Gruppe, ein Ring, ein Körper, etc. Von besonderer Bedeutung für die Informatik sind sogenannte *freie Algebren.* Diese universellen Algebren sind, stark vereinfacht formuliert, die allgemeinsten Algebren innerhalb von Familien von Algebren mit gemeinsamen Eigenschaften. Freie Algebren erlauben uns die Spezifikation beinahe beliebiger Strukturen, beispielsweise von Datenstrukturen, über einfache Axiome. Auch wenn freie Algebren im Zusammenhang mit Programmierparadigmen nicht so dominant sind wie z. B. Funktionen, so spielen sie doch in einigen Bereichen eine wichtige Rolle, etwa im Zusammenhang mit Modulen und Typen. Auf freien Algebren basieren aber auch viele Spezifikationssprachen.

Prozesskalküle: Eine Familie speziell dafür entwickelter Algebren eignet sich gut zur Modellierung von Ausführungseinheiten (*Prozesse* genannt) in nebenläufigen und parallelen Systemen. Die bekanntesten Prozesskalküle sind *CSP (Communicating Sequential Processes)* [15] und π*-Kalkül* [30]. Primitive Operationen gibt es ausschließlich für das Senden und Empfangen von Daten. Sie sind durch Hintereinanderausführung, Parallelausführung sowie alternative Ausführung miteinander kombinierbar. Während die Mächtigkeit der Turing-Maschine im λ-Kalkül durch die Übergabe beliebiger Argumente bei Funktionsaufrufen erreicht wird, geschieht dies im π-Kalkül durch das Senden und Empfangen beliebiger Daten durch Prozesse. Dieser Unterschied hat wichtige Konsequenzen: Im λ-Kalkül lassen sich nur *transformatorische* Systeme gut beschreiben, die zum Zeitpunkt des Programmstarts vorliegende Eingabedaten in Ergebnisdaten transformieren. Dagegen eignet sich der π-Kalkül zur Beschreibung *reaktiver* Systeme, die auf Ereignisse in der Umgebung reagieren, wann immer diese auftreten. Endlosberechnungen im λ-Kalkül liefern keine Ergebnisse und sind sinnlos. Dagegen sind Endlosberechnungen im π-Kalkül wohldefiniert und sinnvoll, aber Prozesse können sich gegenseitig blockieren.

Automaten: Die klassische Automatentheorie wurde in der Frühzeit der Informatik entwickelt. Automaten unterschiedlicher Komplexität stellen gleichzeitig verschiedene Arten von Grammatiken und entsprechende Sprachklassen dar [16]. Obwohl Grammatiken nur Syntax beschreiben, sind die mächtigsten unter ihnen so mächtig wie die Turing-Maschine. Zur Syntaxbeschreibung und in der Implementierung von Programmiersprachen spielen Konzepte aus der Automatentheorie nach wie vor eine große Rolle, als Grundlage von Programmierparadigmen aber nur am Rande. Wegen ihrer anschaulichen Darstellung werden Automaten nicht selten zur Spezifikation des Systemverhaltens verwendet.

WHILE, GOTO und Co: So manches typische Sprachkonstrukt imperativer Sprachen hatte zum Zeitpunkt seiner Entstehung keinerlei formalen Hintergrund. Entsprechende Formalismen mussten erst geschaffen und analysiert werden. Beispiele dafür sind die *WHILE*- und die *GOTO*-Sprache, ganz einfache formale Sprachen, in denen es außer Zuweisungen, primitiven arithmetischen Operationen und bedingten Anweisungen nur entwe-

der eine While-Schleife oder eine Goto-Anweisung (Sprung an eine beliebige andere Programmstelle) gibt. Diese beiden Sprachen sind so mächtig wie die Turing-Maschine. Im Gegensatz dazu ist die *LOOP*-Sprache, in der es statt While bzw. Goto eine Schleife mit einer vorgegebenen Anzahl an Iterationen gibt, nur äquivalent zu primitiv-rekursiven Funktionen. *PRAM*-Sprachen (Parallel Random Access Memory) ändern obige Sprachen dahingehend ab, dass die dahinter stehenden Maschinenmodelle mehrere Operationen gleichzeitig auf unterschiedlichen Speicherzellen durchführen können. Es ist klar, dass derartige formale Sprachen eine starke Verbindung zu imperativen Programmierparadigmen haben.

Diese Aufzählung ist ganz und gar nicht vollständig. Es gibt viele weitere, oft eher exotische Modelle und Formalismen, die in dem einen oder anderen Paradigma eine größere Bedeutung erlangt haben.

1.1.2 Lambda-Kalkül

Wir betrachten eine Variante des λ-Kalküls etwas genauer. Gegeben sei eine unendliche Menge von *Variablen* V, wobei zur Vereinfachung auch Konstanten und Literale in V vorkommen. Die Menge E aller wohlgeformten (das heißt, syntaktisch korrekten) λ-Ausdrücke ist induktiv aufgebaut:

$$
\begin{array}{rll}
v \in E & \text{wenn } v \in V & \\
fe \in E & \text{wenn } e, f \in E & /\!*\ \text{Anwendung von } f \text{ auf } e\ *\!/ \\
\lambda v.e \in E & \text{wenn } v \in V; e \in E & /\!*\ \text{Funktion; Parameter } v\text{, Ergebnis } e\ *\!/
\end{array}
$$

Funktionen heißen auch λ-*Abstraktionen.* Wir fügen nach Bedarf Klammern hinzu, um die Struktur der λ-Ausdrücke eindeutig zu beschreiben. Beispielsweise steht $\lambda v.(v\,v)$ für eine λ-Abstraktion, die als Ergebnis $v\,v$ zurückgibt, wobei v durch das bei einer Anwendung an die Funktion übergebene Argument ersetzt wird. Entsprechend steht $(\lambda v.(v\,v))(a\,b)$ für die Anwendung der Funktion auf $a\,b$, was äquivalent zu $(a\,b)(a\,b)$ ist. Zur Vermeidung zu vieler Klammern wird $(e\,f)g$ üblicherweise als $e\,f\,g$ geschrieben, $\lambda v.(ef)$ als $\lambda v.ef$, $\lambda u.(\lambda v.e)$ als $\lambda u.\lambda v.e$ und so weiter. Wir verwenden dennoch immer Klammern, um Missverständnisse zu vermeiden. Im Wesentlichen beschreibt der λ-Kalkül eine *Äquivalenzrelation* (genauer eine *Kongruenzrelation*, also eine Äquivalenzrelation mit speziellen Eigenschaften) zwischen den Ausdrücken in E. Zwei λ-Ausdrücke werden als äquivalent betrachtet, wenn sie durch Anwendung einiger weniger Regeln ineinander umgewandelt werden können.

Wir benötigen die Menge $\mathrm{fv}(e)$ der in e *frei* vorkommenden Variablen:

$$
\begin{array}{rll}
\mathrm{fv}(v) = \{v\} & & (v \in V) \\
\mathrm{fv}(fe) = \mathrm{fv}(f) \cup \mathrm{fv}(e) & & /\!*\ \mathrm{fv}(f) \text{ vereinigt mit } \mathrm{fv}(e)\ *\!/ \\
\mathrm{fv}(\lambda v.e) = \mathrm{fv}(e) \setminus \{v\} & & /\!*\ \mathrm{fv}(e) \text{ ohne } v\ *\!/
\end{array}
$$

$\mathrm{fv}(e)$ enthält also alle in e vorkommenden Variablen, außer jenen, die als Parameter *gebunden* vorkommen. Wir benötigen auch die *Ersetzung*: Der Ausdruck

$[e/v]f$ mit $v \in V$ wird gelesen als „e ersetzt alle freien Vorkommen von v in f“ und steht für den λ-Ausdruck, der durch die Ersetzung entsteht:

$$\begin{aligned}
[e/v]v &= e \\
[e/v]u &= u && (u \in V; u \neq v) \\
[e/v](f\,g) &= ([e/v]f)([e/v]g) \\
[e/v](\lambda v.f) &= \lambda v.f \\
[e/v](\lambda u.f) &= \lambda u.[e/v]f && (u \neq v; u \notin \mathrm{fv}(e))
\end{aligned}$$

Der Parameter v einer Funktion wird nicht ersetzt, da der Parameter für einen anderen Ausdruck steht als v außerhalb der Funktion. Auffällig ist die Einschränkung $u \notin \mathrm{fv}(e)$ in der letzten Zeile: Die Ersetzung ist nicht möglich, wenn u in e frei vorkommt, weil u als Parameter in der Funktion für einen anderen Ausdruck steht als u außerhalb der Funktion. Wir sehen gleich, dass die erste Regel des λ-Kalküls solche Namenskonflikte beseitigen kann.

Hier sind die Äquivalenzregeln des λ-Kalküls (≡ steht für „ist äquivalent zu“):

$$\begin{aligned}
\lambda u.e &\equiv \lambda v.[v/u]e && (v \notin \mathrm{fv}(e)) && /*\ \alpha\text{-Konversion } */ \\
(\lambda v.f)e &\equiv [e/v]f && && /*\ \beta\text{-Konversion } */ \\
\lambda v.(e\,v) &\equiv e && (v \notin \mathrm{fv}(e)) && /*\ \eta\text{-Konversion } */
\end{aligned}$$

Die *α-Konversion* erlaubt uns, Parameter beliebig umzubenennen, solange dies zu keinen Namenskonflikten führt; daher die Einschränkung $v \notin \mathrm{fv}(e)$. Die wichtigste Regel ist die *β-Konversion*, die besagt, dass eine Funktion $\lambda v.f$ angewandt auf ein Argument e äquivalent zum Funktionsergebnis $[e/v]f$, also dem Ergebnis f, in dem alle freien Vorkommen des Parameters v durch das Argument e ersetzt sind. Dabei ist zu beachten, dass Vorkommen von v zwar in $\lambda v.f$ gebunden sind, aber nicht in f (außer für jene möglicherweise existierenden v, die Parameter einer anderen, in f enthaltenen Funktion sind). Die Ersetzung beschreibt die Parameterübergabe. Die relativ unwichtige *η-Konversion* sorgt dafür, dass alle Funktionen äquivalent sind, die für gleiche Argumente gleiche Ergebnisse liefern; sie kann für Optimierungen eingesetzt werden. Betrachten wir den λ-Ausdruck $(\lambda v.(f v))e$. Durch Anwendung der β-Konversion erkennen wir, dass dieser Ausdruck äquivalent zu fe ist. Aber auch die η-Konversion ist anwendbar und führt zu fe. Wenn beide Regeln anwendbar sind, spielt es keine Rolle, welche wir verwenden. Aber die η-Konversion ist auch anwendbar, wenn der Teilausdruck e nicht dabei steht. Die β-Konversion ist dagegen auch anwendbar, wenn das Ergebnis der Funktion nicht genau diese Form hat.

Obere Regeln sind ungerichtet, sie können von rechts nach links ebenso angewendet werden wie von links nach rechts. Das ist unpraktisch, weil wir viel Intuition benötigen, um herauszufinden, wie wir einen λ-Ausdruck in einen anderen, äquivalenten λ-Ausdruck umformen können. Zur Vereinfachung verwenden wir statt der Äquivalenzregeln meist *Reduktionsregeln*, die immer nur von links nach rechts anwendbar sind und damit λ-Ausdrücke vereinfachen (*reduzieren*):

$$\begin{aligned}
(\lambda v.f)e &\to [e/v]f && && /*\ \beta\text{-Reduktion } */ \\
\lambda v.(e\,v) &\to e && (v \notin \mathrm{fv}(e)) && /*\ \eta\text{-Reduktion } */
\end{aligned}$$

Der einzige Unterschied ist die vorgegebene Richtung, abgesehen davon, dass es für die α-Konversion keine gerichtete Variante gibt, da eine Umbenennung in jede Richtung gleich funktioniert. Wir verwenden β- und η-Reduktion zusammen mit α-Konversion. Wir sagen, ein λ-Ausdruck sei in *Normalform*, wenn er maximal reduziert ist, darauf also keine β- oder η-Reduktion mehr anwendbar ist. Reduktionen und Normalformen haben einige wichtige Eigenschaften:

- Nicht zu jedem λ-Ausdruck gibt es einen äquivalenten λ-Ausdruck in Normalform. Es gibt λ-Ausdrücke, die wir beliebig oft reduzieren können, ohne sie dabei kleiner werden zu lassen. Ein Beispiel ist $(\lambda v.(v\,v))(\lambda v.(v\,v))$: β-Reduktion ist anwendbar und führt wieder zu genau dem gleichen λ-Ausdruck – eine Endlosreduktionsfolge.

- Auch wenn es zu einem λ-Ausdruck einen äquivalenten λ-Ausdruck in Normalform gibt, kann es dennoch unendlich lange Reihenfolgen von Reduktionen geben, die nicht zu einer Normalform führen. Das ist dann der Fall, wenn Regeln immer wieder auf den gleichen Teilausdrücken angewandt werden, die nichts zur Erreichung der Normalform beitragen.

- Jede Reihenfolge von Anwendungen der Regeln auf einen λ-Ausdruck, die zu einer Normalform führt, führt (bis auf α-Konversion) zur gleichen Normalform. Z. B. führt die Ableitung $(\lambda u.(\lambda v.(u\,v))b)a \to (\lambda v.(a\,v))b \to a\,b$ zum gleichen Ausdruck wie $(\lambda u.(\lambda v.(u\,v))b)a \to (\lambda u.(u\,b))a \to a\,b$. Wenn wir eine Normalform finden, wissen wir, dass diese (bis auf Umbenennungen) die einzige ist. Umbenennungen durch α-Konversion sind immer möglich, wenn Funktionen vorkommen.

- Wenn Anwendungen der Regeln auf zwei λ-Ausdrücke zu gleichen λ-Ausdrücken führen, wissen wir, dass die λ-Ausdrücke äquivalent zueinander sind. Um Äquivalenz zu zeigen, versuchen wir auf Normalformen zu reduzieren. Nur bei Äquivalenz sind die jeweiligen Normalformen gleich (bis auf α-Konversion).

Wir haben schon erwähnt, dass der λ-Kalkül alles berechnen kann, was als berechenbar gilt. Berechnen heißt, dass wir λ-Ausdrücke auf Normalformen reduzieren. Aufgrund des bekannten Halteproblems der Turing-Maschine muss es, wie oben gezeigt, λ-Ausdrücke ohne Normalform geben. Überraschend ist eher, mit welch einfachen Mitteln die Mächtigkeit der Turing-Maschine erreicht wird. Im Vergleich zu Programmiersprachen kann der λ-Kalkül fast nichts. Es gibt keine vorgegebenen Zahlen und Grundrechenoperationen, keine Schleifen, keine `if`-Anweisungen, nicht einmal Funktionen mit mehreren Parametern. Einzig und alleine ganz einfache Funktionen mit je einem einzigen Parameter existieren. Daraus können wir alles andere aufbauen, auch die grundlegenden Objekte der Mathematik wie Mengen und Zahlensysteme. Ermöglicht wird das dadurch, dass im λ-Kalkül alle λ-Ausdrücke als Argumente an Funktionen übergeben und als Ergebnisse zurückgegeben werden können, auch Funktionen selbst. Der Umgang mit dem λ-Kalkül auf niedrigem Niveau ist zwar oft schwierig und ineffizient, die Verwendung von Funktionen darin aber vergleichsweise einfach.

Betrachten wir einige Beispiele für grundlegende Funktionalität im λ-Kalkül. Zunächst stellen wir Funktionen mit mehreren Parametern dar. Das gelingt, indem wir eine Funktion mit einem Parameter verwenden, die eine weitere Funktion mit einem Parameter zurückgibt, die eine weitere Funktion mit einem Parameter zurückgibt, ... (so viele Parameter wie nötig), die dann das eigentliche Ergebnis zurückgibt. Folgende Funktion nimmt zwei Parameter:

$$\lambda u.(\lambda v.((u\,v)(v\,u)))$$

Angewandt auf a und b sind folgende β-Reduktionen ausführbar:

$$((\lambda u.(\lambda v.((u\,v)(v\,u))))a)b \rightarrow (\lambda v.((a\,v)(v\,a)))b \rightarrow (a\,b)(b\,a)$$

Statt a und b könnten wir beliebig komplexe λ-Ausdrücke verwenden.

Folgende drei Funktionen zeigen den Umgang mit Wahrheitswerten:

$\lambda u.(\lambda v.u)$ /∗ `true`: nimm erstes Argument ∗/
$\lambda u.(\lambda v.v)$ /∗ `false`: nimm zweites Argument ∗/
$\lambda b.(\lambda u.(\lambda v.((b\,u)v)))$ /∗ `if_then_else` ∗/

Durch Anwendung von `if_then_else` auf `true` (Teilausdruck dafür zur besseren Übersicht unterstrichen), a und b ergeben sich folgende β-Reduktionen:

$$\begin{aligned}
&(((\lambda b.(\lambda u.(\lambda v.((b\,u)v))))(\underline{\lambda u.(\lambda v.u)}))a)b \rightarrow\\
&((\lambda u.(\lambda v.(((\underline{\lambda u.(\lambda v.u)})u)v)))a)b \rightarrow\\
&(\lambda v.(((\underline{\lambda u.(\lambda v.u)})a)v))b \rightarrow\\
&((\underline{\lambda u.(\lambda v.u)})a)b \rightarrow\\
&(\lambda v.a)b \rightarrow\\
&a
\end{aligned}$$

Die Variablen u und v im unterstrichenen Teilausdruck sind gebunden und von Ersetzungen gleichnamiger Variablen weiter außen nicht betroffen. Bei Verwendung von `false` statt `true` wäre der Ausdruck zu b reduzierbar.

Zur Darstellung von Zahlen gibt es viele Möglichkeiten. Beispielsweise ist die Zahl 0 einfach durch 0 (mit $0 \in V$) darstellbar und darauf ein s (irgendein Name mit $s \in V$) anwendbar, um die Zahl um 1 zu erhöhen. Damit entspricht $s\,0$ der Zahl 1, $s(s\,0)$ der Zahl 2, $s(s(s\,0))$ der Zahl 3 und so weiter. Darauf aufbauend können wir grundlegende Rechenoperationen als Funktionen definieren.

1.1.3 Praktische Realisierung

Beim Programmieren denken wir kaum bewusst an Berechnungsmodelle, sondern eher an Programmierwerkzeuge und diverse Details der Syntax und Semantik einer Sprache. Es geht um die Lösung praktischer Aufgaben. Im Idealfall unterstützen uns die Werkzeuge und Sprachen bei der Lösung der Aufgaben in einer zum Berechnungsmodell passenden Weise. Trotz der Unterschiedlichkeit der Aufgaben, Sprachen, Modelle und Werkzeuge bestimmen immer wieder dieselben Eigenschaften den Erfolg oder Misserfolg eines Programmierparadigmas zu einem großen Teil mit:

Kombinierbarkeit: Bestehende Programmteile sollen sich möglichst einfach zu größeren Einheiten kombinieren lassen, ohne dass diese größeren Einheiten dabei ihre einfache Kombinierbarkeit einbüßen. Funktionen liefern ein gutes Beispiel dafür: Eine Funktion setzt sich einfach aus Aufrufen weiterer Funktionen zusammen. Nicht alle Formalismen liefern eine so gute Basis für kombinierbare Sprachkonzepte. Beispielsweise entstehen aus der Kombination mehrerer Automaten zu größeren Automaten meist äußerst komplexe Strukturen. Mechanismen für die Kombination müssen gut *skalieren,* das heißt, auch große Einheiten müssen einfach kombinierbar sein, nicht nur kleine. Gerade wenn es um die Entwicklung großer Programme geht, ist die einfache Kombinierbarkeit extrem wichtig. Vor allem in der objektorientierten Programmierung reicht auch die gute Kombinierbarkeit von Funktionen nicht mehr aus und wir verwenden zusätzliche Sprachkonzepte wie Objekte zur Verbesserung der Kombinierbarkeit.

Konsistenz: Programmiersprachen und -paradigmen müssen zahlreiche Erwartungen hinsichtlich verschiedenster Aspekte erfüllen. Ein einziger Formalismus reicht als Grundlage dafür nicht aus. Mehrere sprachliche Konzepte – etwa solche zur Beschreibung von Algorithmen und andere zur Beschreibung von Datenstrukturen – müssen zu einer Einheit verschmolzen werden. Alle Konzepte sollen in sich und miteinander konsistent sein, also gut zusammenpassen. Nun sind Formalismen von Haus aus oft nicht oder zumindest nicht vollständig kompatibel zueinander. Wir würden kaum ein zufriedenstellendes Ergebnis erhalten, wenn wir die beste formale Grundlage für jeden Aspekt wählen und alle entsprechenden Formalismen zusammenfügen würden. Ein riesiges Konglomerat mit zahlreichen Widersprüchen würde entstehen. Gute Sprachen und Paradigmen kommen mit wenigen konsistenten Konzepten aus. Im Umkehrschluss bedeutet das aber auch, dass nicht in allen Aspekten auf den optimalen Grundlagen aufgebaut werden kann, sondern viele Kompromisse nötig sind. Konsistenz ist in der Praxis wichtiger als Optimalität. Nur so kann sich die nötige Einfachheit ergeben.

Abstraktion: Eines der ursprünglichsten und noch immer wichtigsten Ziele der Verwendung höherer Programmiersprachen ist die Abstraktion über Details der Hardware und des (Betriebs-)Systems. Programme sollen nicht von solchen Details abhängen, sondern möglichst *portabel* sein, also auf ganz unterschiedlichen Systemen laufen können. Praktisch alle derzeit verwendeten Paradigmen erreichen dieses Ziel recht gut, sofern nicht sehr große Systemnähe gefordert wird. Heute verstehen wir unter „Abstraktion“ nicht mehr nur die Abstraktion über das System, sondern abstrakte Sichtweisen vieler weiterer Aspekte. Wir können Abstraktionen von fast allem bilden und in Programmen darstellen. Im Laufe der Entwicklung von Programmierparadigmen hat der erreichbare Abstraktionsgrad ständig zugenommen und ist heute sehr hoch. In den Abschnitten 1.3 und 1.4 werden wir verschiedene Formen der Abstraktion näher betrachten.

Systemnähe: Programme müssen effizient auf realer Hardware ausführbar sein. Effizienz lässt sich scheinbar am leichtesten erreichen, wenn das Paradigma wesentliche Teile der Hardware und des Betriebssystems direkt widerspiegelt. Unverzichtbar ist Systemnähe dann, wenn Hardwarekomponenten direkt angesprochen werden müssen. Häufig wird ein systemnahes Paradigma bevorzugt, um die Möglichkeiten eines Systems direkt einbinden zu können. Gelegentlich forcieren Anbieter aus wirtschaftlichen Gründen eine starke Betriebssystemabhängigkeit auf Kosten der Portabilität. In anderen Bereichen möchten wir dagegen aus Sicherheits- und Portabilitätsgründen vor direkten Zugriffen auf das System geschützt sein. Ein Paradigma, das überall passt, wird es kaum geben können.

Unterstützung: Ein Paradigma hat keinen Wert, wenn entsprechende Programmiersprachen, Entwicklungswerkzeuge sowie Bibliotheken vorgefertigter Programmteile fehlen. So manches ursprünglich nur mittelmäßige Paradigma (etwa das prozedurale, funktionale und objektorientierte) hat sich wegen guter Unterstützung trotzdem durchgesetzt und wurde im Laufe der Zeit verbessert. Ohne Personen, die an den Erfolg glauben und viel Zeit und Geld in die Entwicklung investieren, kann sich kein Paradigma durchsetzen. Oft sind Kleinigkeiten und Zufälle für den Erfolg ausschlaggebend. So hat die fast schon tot geglaubte Programmiersprache Smalltalk zum Jahrtausendwechsel plötzlich eine Renaissance erlebt, weil keine ganzen Zahlen mit beschränktem Wertebereich unterstützt wurden und daher keine Zahlenüberläufe möglich waren, die in anderen Sprachen wegen zu klein gewählter Wertebereiche häufig zu Fehlern und Abstürzen führten. Manchmal ist die Unterstützung durch einflussreiche Unternehmen essentiell, wie die starke Verbreitung von Java und C# zeigt.

Beharrungsvermögen: Obwohl ständige Innovationen in der Softwareentwicklung üblich sind, werden größere Änderungen der Programmierparadigmen nur sehr langsam angenommen. Jeder Paradigmenwechsel bedeutet, dass so manches Wissen verloren geht und neue Erfahrungen erst gemacht werden müssen. Für einen Wechsel braucht es sehr überzeugende Gründe. Ein solcher Grund wäre, dass jemand den Paradigmenwechsel in einem vergleichbaren Bereich schon vollzogen hat und damit merklich erfolgreicher ist als im alten Paradigma. Wir sprechen von einer *Killerapplikation,* also erfolgreicher Software, die den Erfolg einer Technik oder eines Paradigmas so deutlich aufzeigt, dass das Potential zur Ersetzung althergebrachter Techniken oder Paradigmen in diesem Bereich unübersehbar ist. Ohne Killerapplikation kommt es zu keinem Paradigmenwechsel.

Wer danach sucht, wird viele weitere praktische Kriterien für den Erfolg von Paradigmen finden, vor allem solche, die für bestimmte Paradigmen bedeutend sind. Wir können die zukünftige Bedeutung vieler Kriterien nur schwer abschätzen. So manche Vermutung aus der Vergangenheit hat sich nicht bewahrheitet. Die oben genannten Kriterien waren jedoch bis jetzt stets von Bedeutung, so-

dass wir mit hoher Wahrscheinlichkeit davon ausgehen können, dass sie es auch in absehbarer Zukunft sein werden.

Die wichtigsten Werkzeuge in der Softwareentwicklung sind Compiler und Interpreter. Seit den ersten Programmiersprachen hat sich auf diesem Gebiet viel getan. Heute gibt es neue Implementierungstechniken und ausreichend Erfahrung, um manche Konzepte, die vor Jahrzehnten erfolglos versucht wurden, nun erfolgreich einzusetzen. Ein Beispiel dafür ist die JIT-Übersetzung (JIT = Just-In-Time); sie verbindet die Portabilität und einfache Bedienbarkeit von Interpretern mit der Effizienz von Compilern; auch wenn in diesen Aspekten gewisse Abstriche nötig sind, ergibt sich insgesamt ein guter Kompromiss. Solche technischen Entwicklungen beeinflussen Programmierparadigmen. Sprachkonzepte, die sich früher nicht durchgesetzt haben, können wegen neuer Implementierungstechniken plötzlich erfolgreich sein.

1.2 Evolution und feine Programmstrukturen

Vom *Programmieren im Großen, Groben* sprechen wir, wenn es um die Grobstruktur von Programmen, etwa die Modularisierung geht, wobei Programmdetails in der Regel unberücksichtigt bleiben. Entsprechend steht der Begriff der *Programmierung im Kleinen, Feinen* für die feinen Programmstrukturen, vor allem für alles, wo es auf Details ankommt. Keinesfalls geht es nur um kleine Programme oder unwichtige Aspekte; ganz im Gegenteil, es geht um den Kern der Programmierung. Zumindest der Umgang mit Variablen und Literalen sowie Ausdrücken, Anweisungen und Kontrollstrukturen bis hin zu Funktionen (bzw. Methoden und Prozeduren) gehört zur Programmierung im Feinen.

Es wird vorausgesetzt, dass grundlegende Sprachkonzepte bekannt sind. Wir betrachten, wie teilweise widersprüchliche Ziele hinter elementaren Sprachkonzepten zu unterschiedlichen Schwerpunktsetzungen und Programmierstilen führen. Danach sehen wir uns die *strukturierte Programmierung* und die Verwendung von Programmteilen als Daten etwas genauer an.

1.2.1 Widersprüche und typische Entwicklungen

Kaum jemand wird widersprechen, dass Programmiersprachen unter anderem folgende Eigenschaften aufweisen sollten:

- Flexibilität und Ausdruckskraft sollen in kurzen Programmtexten die Darstellung aller vorstellbaren Programmabläufe ermöglichen.
- Lesbarkeit und Sicherheit sollen Absichten hinter Programmteilen sowie mögliche Inkonsistenzen leicht erkennen lassen.
- Die Konzepte müssen verständlich bleiben und es muss klar sein, was einfach machbar ist und was nicht geht.

Diese Aussagen widersprechen sich, auch wenn das auf den ersten Blick nicht auffällt. Wenn wir uns sehr bemühen, können wir je zwei dieser Eigenschaften

so miteinander kombinieren, dass beide Eigenschaften recht gut erfüllt sind. Die technischen Möglichkeiten sind heute so weit fortgeschritten, dass Flexibilität, Ausdruckskraft, Lesbarkeit und Sicherheit gleichzeitig recht gut erfüllt sein können. Mehrere Jahrzehnte an Entwicklungsarbeit waren dafür nötig. Aber sobald wir den dritten Punkt, die einfache Verständlichkeit dazu nehmen, wird es schwierig: Die ersten beiden Punkte konnten nur dadurch unter einen Hut gebracht werden, dass immer feiner verästelte Fallunterscheidungen getroffen wurden, die außer wenigen Expert_innen kaum jemand mehr durchschaut.

Die althergebrachte dynamisch typisierte Programmierung bevorzugt den ersten gegenüber dem zweiten Punkt, während die althergebrachte statisch typisierte Programmierung den zweiten gegenüber dem ersten bevorzugt. Viele jüngere Sprachen erzielen einen hohen Grad an Flexibilität und Ausdruckskraft gleichzeitig mit Lesbarkeit und Sicherheit. Damit verbundene fortschrittliche Beweistechniken gehen aber auf Kosten des dritten Punkts. Neue Sprachen werden häufig mit verbesserter Einfachheit und Verständlichkeit beworben. Bei Erfolg verschwindet diese Zielsetzung (vor allem, wenn Sicherheit ein Thema ist) rasch aus dem Blickfeld, weil professionelle Entwickler_innen inhärent komplexe Aufgaben lösen müssen und dafür Unterstützung durch entsprechend komplexe Werkzeuge brauchen. Es ist auch erkennbar, dass statisch und dynamisch typisierte Programmierung zwar schon seit Beginn der Informatik zusammen existieren, aber gelegentlich Pendelbewegungen einmal hin zu eher statischer und dann wieder hin zu eher dynamischer Typisierung stattfinden.

Ob sich eher ein dynamischer oder statischer Ansatz zur Lösung einer Aufgabe eignet, hängt stark von der Art der Aufgabe ab. Kaum jemand kommt auf die Idee, ein sicherheitskritisches Betriebssystem in JavaScript oder Python (zwei dynamisch typisierten Sprachen) zu entwickeln. Niemand wird ein nur für den einmaligen Gebrauch bestimmtes Skript rasch in Rust (einer Sprache mit komplexen statischen Prüfungen) hinschreiben wollen. Aber es gibt einen großen Bereich an Anwendungen, für die sowohl statische als auch dynamische Ansätze vertretbar sind. Dafür wird häufig nur der vertrautere Ansatz gewählt, nicht notwendigerweise der beste. Die schwankende Anzahl an Personen, die mit der Entwicklung in bestimmten Ansätzen vertraut sind, ist wahrscheinlich der wichtigste Grund für die Pendelbewegungen. Technische Weiterentwicklungen können auch eine Ursache sein, aber meist nur in der Form, dass sich viele Personen von den Entwicklungen ansprechen lassen und sie ausprobieren wollen.

Aus dem eben Gesagten könnte geschlossen werden, dass ein Programmierparadigma oder -stil vor allem eine Modeerscheinung auf einer nur schwer technisch begründbaren Basis darstellt – hinsichtlich vieler einander widersprechender Aspekte, nicht nur wenn es um statisch versus dynamisch typisierte Programmierung geht. Als seriöse Techniker_innen lehnen wir Modeerscheinungen ab und verlassen uns lieber auf technische Hintergründe. Wahrscheinlich sind es jedoch gerade die kaum begründbaren Modeerscheinungen, die als Triebfedern technische Entwicklungen voranbringen. Auch Programmierparadigmen ändern sich mit der Zeit und werden besser. Wer heute auf die Anfänge z. B. der objektorientierten Programmierung zurückblickt, wird kaum noch Ähnlichkeiten zur damaligen Denkweise erkennen. Es gibt also einen starken evolutionären

Fortschritt, obwohl es sich um das gleiche Paradigma handelt. Viele Programmierer_innen haben die Entwicklung vom Beginn bis heute mitgemacht, immer wieder etwas Neues ausprobiert, vieles davon wieder verworfen, aber einiges ist hängen geblieben und hat das Paradigma nachhaltig verändert. Modeerscheinungen haben die Blickwinkel dabei immer wieder neu ausgerichtet und dazu geführt, dass das Paradigma stabiler geworden ist und heute der kritischen Betrachtung aus sehr vielen verschiedenen Blickwinkeln standhält.

Es stellt sich die Frage, wie sich ein Programmierparadigma verändern kann und nicht einfach ein neues, anderes Paradigma entsteht. Das hat mit den Menschen zu tun, die ein Paradigma weiterentwickeln. Jeder Mensch wird einen eigenen Stil innerhalb des Paradigmas finden, der sich von den Stilen der anderen unterscheidet. Jeder Mensch wird diesen Stil auch ständig weiterentwickeln. Wenn viele Menschen ihre Stile in eine ähnliche Richtung weiterentwickeln, entwickelt sich auch das Paradigma ohne starke Brüche weiter. Menschen arbeiten nicht isoliert voneinander, sondern passen ihre Stile einander an. Die Notwendigkeit der Zusammenarbeit ist eine sehr starke Kraft, die der Aufspaltung eines Paradigmas entgegenwirkt und gleichzeitig evolutionäre Änderungen zulässt. Ohne technisch zwingende Ursache könnte sich eine Aufspaltung in mehrere Paradigmen nur ergeben, wenn Menschen sich zu isolierten Gruppen zusammenfinden würden, die nicht an gemeinsamen Projekten arbeiten. Das passiert nur äußerst selten und ist im globalisierten Umfeld bei einem stark verbreiteten Paradigma, insbesondere im Kern der Programmierung praktisch unmöglich.

Auch in der persönlichen Entwicklung des Programmierstils eines Menschen sind typische Muster erkennbar. Nehmen wir als Beispiel die Namensgebung. Zu Beginn steht eine Unsicherheit, weil einerseits das praktische Analogon zur α-Konversion (Namen sind beliebig austauschbar) verinnerlicht wird, andererseits aber klar ist, dass es bestimmte Erwartungen gibt. Wir suchen nach starren, immer gültigen Regeln für Benennungen. Bald werden Namen lang und als natürlichsprachige Texte lesbar. Danach entsteht manchmal eine gewisse „Trotzphase“, in der lange Namen und starre persönliche Regeln vorherrschen, gleichzeitig aber vorgegebene Regeln gebrochen werden, um eine persönliche Note hineinzubringen. Schön langsam entsteht ein angepasster Stil, der Unterschiede zwischen den zu benennenden Einheiten macht und dort, wo Suggestion beim Lesen des Programms nötig ist, auf beschreibende Namen setzt, an anderen Stellen aber kurze, einheitliche Namen bevorzugt. Das setzt ein gutes Verständnis dafür voraus, was durch die Syntax oder Struktur des Programms offensichtlich ist und wo zusätzliche Suggestion durch Namen hilft. Gleichzeitig wird verstärkt auf Verwechslungsgefahren aufgepasst, also ähnlich klingende Namen ebenso gemieden wie solche, wo etwa 0 (Ziffer) mit O (Buchstabe) verwechselt werden könnte. Weit fortgeschrittene Personen achten darauf, dass Bedeutungen aus der Programmstruktur ablesbar sind. Namen können kürzer sein, folgen aber einem an das Projekt angepassten Schema. Regeln sind nicht mehr auf persönlicher Ebene wichtig, sondern projektbezogen. An einem Projekt arbeiten unterschiedlich weit fortgeschrittene Menschen. Programmiersprachen müssen so flexibel sein, dass sie mit allen Formen und Entwicklungsstufen in der Namensgebung umgehen können.

In gewisser Weise spiegeln sich persönliche Entwicklungen von Menschen auch in Programmiersprachen wider. Nehmen wir die Urform von Fortran. Namen wurden so gewählt, wie es in der Mathematik üblich ist. Variablen beginnend mit `i` bis `n` enthielten ganze Zahlen, alle anderen Gleitkommazahlen. Texte mussten (wie bei Lochkarten) in einer bestimmten Spalte beginnen. Schon bald wurde die Programmstruktur und Namensgebung (vor allem durch die Einführung von Typen) flexibler, gleichzeitig wurden aber auch einige Freiheiten (wie etwa Leerzeichen in Namen) aufgegeben. Namen wurden länger und beschreibender. Vor allem in Cobol waren Namen, auch syntaktisch vorgegebene, durchwegs lang und es wurde versucht, Programmtexte als gut lesbare englische Texte darzustellen. Damit sollte bewirkt werden, dass jeder Mensch, auch ohne Programmiererfahrung, Programme richtig lesen kann. Es stellte sich bald heraus, dass lange Programmtexte Entwicklungszeiten in die Höhe treiben, ohne Programme tatsächlich lesbarer zu machen. Als Gegenreaktion entstanden Sprachen wie APL, die sehr kompakte Programmtexte ermöglichten, die rasch schreibbar, aber kaum lesbar waren. Heute herrschen Sprachen vor, die sowohl kurze als auch lange Namen gut unterstützen, aber hinsichtlich sonstiger syntaktischer Elemente zurückhaltend sind, etwa Textblöcke in wenige überschaubare Klammern einschließen oder nur über Einrückungen klar vom Rest absetzen. Die Programmstruktur ist visuell leicht erkennbar, ohne Details lesen zu müssen. Treibende Faktoren solcher Entwicklungen sind stets neu gewonnene Erfahrungen vor dem Hintergrund widersprüchlicher Ziele.

1.2.2 Strukturierte Programmierung

Visuell leicht erkennbare Programmstrukturen sind durch *strukturierte Programmierung* möglich, einer großen Erfolgsgeschichte der Programmierung. Sie bringt Struktur in die prozedurale Programmierung. Jedes Programm und jeder Rumpf einer Prozedur ist nur aus drei einfachen Kontrollstrukturen aufgebaut:

- Sequenz (ein Schritt nach dem anderen)
- Auswahl (Verzweigung im Programm, z. B. `if` und `switch`)
- Wiederholung (Schleife oder Rekursion)

Heute erscheinen diese Kontrollstrukturen nicht nur in der prozeduralen Programmierung als so selbstverständlich, dass wir uns kaum mehr etwas anderes vorstellen können. Nur durch die genaue Ausformung ergeben sich noch Unterschiede. Das mächtige und früher allgegenwärtige „Goto“, das einen Gegenpol zur strukturierten Programmierung bildet, wird heute fast gar nicht mehr verwendet. Die Goto-Anweisung spezifiziert eine beinahe beliebige Stelle in einer Prozedur oder im Programm, an der die Programmausführung fortgesetzt werden soll. Diese Anweisung bildet Sprungbefehle, auf die Mikroprozessoren auf Hardware-Ebene fast immer aufbauen, recht direkt in die Programmiersprache ab. Dennoch gilt Goto wegen hoher Fehleranfälligkeit heute als verpönt.

Zuerst wollen wir klären, was eine *Prozedur* überhaupt ist. Sie ähnelt einer Funktion, abgesehen davon, dass die Konzentration auf Seiteneffekten liegt. Bei

einer Prozedur kommt es hauptsächlich auf Zustandsänderungen an, die in der Prozedur durch Zuweisung neuer Werte an Variablen erreicht werden, nicht auf zurückgegebene Ergebnisse. Im Gegensatz zu Funktionen sind Prozeduren nie frei von Seiteneffekten. Eine Prozedur verändert Werte von Variablen, die außerhalb der Prozedur definiert sind. Die Goto-Anweisung wird erst durch nicht-lokale Effekte von Zuweisungen mächtig. Goto ist daher untrennbar mit prozeduraler Programmierung, der Programmierung mit Prozeduren verknüpft.

Viele Kontrollstrukturen und Programmabläufe lassen sich grafisch darstellen. Dafür gibt es mehrere Ansätze. Beispielsweise ist der Programmfluss in Form eines Graphen visualisierbar, oder in Form von Kästchen, die wie Klötze eines Baukastens neben- und untereinander angeordnet die Blockstruktur des Programms widerspiegeln. Visualisierungen können dabei helfen, sich den Programmablauf vorzustellen. Heute werden Programme nur noch selten tatsächlich grafisch dargestellt, weil wir uns beim Lesen des Programmtexts die Struktur direkt, ohne Umweg über eine Visualisierung gut vorstellen können. Strukturierte Programmierung sorgt dafür, dass Visualisierungen der Struktur sehr einfach werden, so einfach, dass wir sie direkt im Code „erkennen“. In der strukturierten Programmierung hat jede Kontrollstruktur nur je einen wohldefinierten Einstiegs- und Ausstiegspunkt. Diese Punkte lassen sich in jeder denkbaren Kombination einfach miteinander verbinden und erleichtern so das Verfolgen des Programmpfads, wie in folgenden Methoden in Java:

```
long factRec(int x) {
    long f = x;
    if (x > 2) {
        f *= factRec(x - 1);
    }
    return f;
}
```

```
long factLoop(int x) {
    long f = x;
    while (--x > 1) {
        f *= x;
    }
    return f;
}
```

Einrückungen und Klammern machen den Kontrollfluss visuell sichtbar, moderne Entwicklungsumgebungen verwenden zusätzlich Farben und Symbole.

Programme mit Goto können, diszipliniert verwendet, zwar auch eine „sichtbare“ Struktur haben, diese spiegelt aber nicht den gesamten Kontrollfluss wider, weil Ein- und Ausstiege an vielen Punkten erfolgen können, was schwierig zu visualisieren und „erkennen“ ist. Noch schwieriger ist es, alle möglichen Programmpfade gedanklich nachzuvollziehen. Einige Kontrollstrukturen in aktuellen Sprachen erfüllen die Prinzipien der strukturierten Programmierung nicht vollständig und haben Eigenschaften, die eher mit Goto als der strukturierten Programmierung assoziiert werden. Betrachten wir Beispiele dazu:

```
for (int i=1; i<10; i++) {
    System.out.println(i);
    if (i == j) {
        break;
    }
}
```

```
for (int i=1; i<10; i++) {
    if (i == j) {
        continue;
    }
    System.out.println(i);
}
```

Oberflächlich betrachtet scheint der Kontrollfluss hier visuell „sichtbar“ zu sein, aber die Anweisungen `break` und `continue` fügen zusätzliche Ein- und Ausgänge hinzu, die nicht „sichtbar“ sind und möglicherweise in die Irre leiten. Es ist bekannt, dass Schleifen mit `break` oder `continue` häufig fehlerhaft sind. Die Ursache liegt wahrscheinlich darin begründet, dass die Verletzung der strukturierten Programmierung, obwohl nur lokal auf einen kleinen Programmteil begrenzt, nicht in das gewohnte Konzept passt und (vor allem routinierte Programmier_innen) mögliche Pfade im Programmablauf übersehen lässt. Das allgemeine Goto ist fehleranfälliger als die in ihrer Wirkung eingeschränkten Goto-Varianten `break` und `continue`, weil sich viel mehr Programmpfade ergeben können, die leicht übersehen werden. Problematisch an allen Varianten ist nicht die eigentlich sehr einfache Semantik der Befehle, sondern die Tatsache, dass Denkmuster, die wir uns durch die strukturierte Programmierung angewöhnt haben, durchbrochen sind. Ähnliche Effekte ergeben sich im Zusammenhang mit Ausnahmebehandlungen. Weniger gravierend sind die Auswirkungen von irgendwo in Programmtexten vorkommenden `return`-Anweisungen:

```
long factRec2(int x) {
    if (x <= 2) {
        return (long)x;  // zusätzlicher Ausgang
    }
    return x * factRec2(x - 1);
}
```

Solche `return`-Anweisungen stellen zwar zusätzliche Ausgänge dar (und verletzen damit Prinzipien der strukturierten Programmierung), aber es entstehen keine zusätzlichen Eingänge, wodurch die Anzahl der zu betrachtenden Programmpfade überschaubar bleibt.

Damit kommen wir auf den Kern der strukturierten Programmierung: Sequenz, Auswahl und Wiederholung mit je einem Ein- und Ausgang bilden die essenziellen Denkmuster, über die wir jedes Programm aufbauen können. Mehr als diese wenigen Denkmuster müssen wir uns auf der Ebene der feinen Programmstrukturen nicht verinnerlichen. Diese Denkmuster sind einfach miteinander kombinierbar und skalieren sehr gut bis auf die höchsten von uns handhabbaren Ebenen der Komplexität. Strukturierte Programmierung gibt uns die Einfachheit, die große Komplexität erst handhabbar macht.

Einfachheit ist auch eine wesentliche Eigenschaft des λ-Kalküls. Bei der strukturierten Programmierung geht es um Einfachheit auf einer anderen, abstrakteren Ebene. Der λ-Kalkül beinhaltet zwar einfache Konzepte, über die sich alles aufbauen lässt, aber die Umsetzung von Aufgabenstellungen in Programme kann sehr fordernd sein. Die strukturierte Programmierung gibt uns zusätzlich als Denkmuster noch klare Hinweise darauf, wie eine Aufgabenstellung in ein Programm umsetzbar ist. Wir müssen in der Aufgabe nur geeignete Sequenzen, Fallunterscheidungen und Wiederholungen finden, schon ergibt sich der Kontrollfluss fast von alleine.

Es stellt sich die Frage, warum es noch Anweisungen wie `break` und `continue` gibt, obwohl sie nicht mit strukturierter Programmierung vereinbar sind. Solche

Anweisungen erhöhen weder die Ausdruckskraft, noch tragen sie (wesentlich) zur Effizienz bei. Die Antwort liegt vermutlich im Wunsch nach Kontinuität und Freiheit. Viele Programmierer_innen kennen diese Konzepte von früher und wollen die Freiheit haben, sie einzusetzen. Weil der Wunsch danach groß genug ist, kommen Sprachentwickler_innen ihm entgegen. Die Frage nach dem tieferen Sinn hat hier keinen Platz. Nur mit der reinen Goto-Anweisung waren die Erfahrungen so eindeutig negativ, dass kaum jemand sie haben will.

Die strukturierte Programmierung löst nicht alle Probleme. Neben dem Kontrollfluss müssen wir auch den Datenfluss im Auge behalten, der von der strukturierten (und auch der prozeduralen) Programmierung gänzlich ignoriert wird. In Abschnitt 1.5 werden wir uns näher damit beschäftigen.

Wir sind implizit davon ausgegangen, dass die strukturierte Programmierung eng mit der prozeduralen Programmierung verknüpft ist. Tatsächlich spielt die strukturierte Programmierung fast überall eine bedeutende Rolle, auch in der objektorientierten, funktionalen und parallelen Programmierung, wahrscheinlich sogar noch stärker als in der prozeduralen. Neuere Programmierstile tun sich leichter damit, Überbleibsel aus vergangener Zeit zu entsorgen. Eine absolut reine strukturierte Programmierung ist dennoch kaum zu erreichen, weil wir auf Konzepte wie Ausnahmebehandlung nicht verzichten möchten, die sich von Natur aus nicht der strukturierten Programmierung unterordnen lassen.

1.2.3 Programmteile als Daten

Funktionen, Prozeduren, Methoden und ähnliche Konzepte gibt es in fast allen Paradigmen. Dennoch spielen Funktionen in funktionalen Sprachen eine wichtigere Rolle, die unter anderem dadurch sichtbar wird, dass Funktionen wie Daten verwendet werden. Wir können Funktionen zur Laufzeit erzeugen (solche Funktionen werden heute häufig als „Lambdas“ oder „Lambda-Ausdrücke“ bezeichnet, obwohl es sich eher um eingeschränkte Formen von λ-Abstraktionen aus dem λ-Kalkül handelt), in Variablen ablegen, als Argumente an andere Funktionen übergeben und als Ergebnisse von Funktionen zurückbekommen. Methoden in Java sind dagegen nicht (oder nur sehr eingeschränkt, jedenfalls nicht vollwertig) als Daten verwendbar. Wir können sie zwar aufrufen, aber nicht an Variablen zuweisen oder als Argumente übergeben.[1] Objekt-ähnliche Konzepte gibt es ebenso in vielen Paradigmen, z. B. in Form von Modulen (siehe Abschnitt 1.3), aber im Wesentlichen nur in der objektorientierten Programmierung sind Objekte wie alle anderen Daten zur Laufzeit erzeug- und verwendbar. Anscheinend bekommt so manches Sprachkonzept erst durch die gleiche Behandlung wie alle anderen Daten jene überragende Bedeutung, die notwendig ist, um ein Paradigma darauf aufzubauen.

In Abschnitt 1.1 haben wir gesehen, dass der λ-Kalkül seine Mächtigkeit dadurch bekommt, dass alle λ-Ausdrücke, auch Funktionen, an Funktionen übergeben und von Funktionen zurückgegeben werden können, wodurch Funktionen

[1] Das gilt auch in neueren Java-Versionen, in denen z. B. `Math::max` scheinbar als Methode an Variablen zugewiesen werden kann. Tatsächlich wird nicht die Methode selbst zugewiesen, sondern ein Objekt, in dem die Methode definiert ist.

als Daten behandelt werden. Primitiv-rekursiven Funktionen fehlt diese Möglichkeit, weshalb sie für sich (ohne zusätzlichen Fixpunkt-Operator μ) nicht so mächtig sind. Zumindest Funktionen verleiht die Verwendbarkeit als Daten also tatsächlich formal nachweisbar große Mächtigkeit.

Aus solchen Überlegungen könnte geschlossen werden, dass die Verwendbarkeit von Programmteilen als Daten das Wesentliche in der Programmierung wäre. Leider sind die Zusammenhänge komplizierter, wie wir am Vergleich des λ-Kalküls mit primitiv-rekursiven Funktionen erkennen können. Primitiv-rekursive Funktionen sind das, was der Name suggeriert: Funktionen (oder z. B. auch Methoden in Java), die sich gegenseitig aufrufen und Zahlen (oder andere elementare Objekte) als Parameter übergeben und Ergebnisse zurückgeben. Beim Programmieren bevorzugen wir diese Funktionen wegen ihrer Einfachheit. Der gesamte Kontrollfluss ist nach den Prinzipien der strukturierten Programmierung vollständig visualisierbar. Viele Aufgaben sind anhand der Denkmuster hinter der strukturierten Programmierung ohne große intellektuelle Anstrengung lösbar. Aber nicht alle Aufgaben sind so lösbar, weil primitiv-rekursive Funktionen nicht die Mächtigkeit des λ-Kalküls haben. Würden wir Funktionen zu Daten machen, hätten wir zwar das Problem mit der Vollständigkeit auf einen Schlag beseitigt, aber Aufgaben, die mit primitiv-rekursiven Funktionen alleine nicht lösbar sind, könnten wir nicht durch strukturierte Programmierung lösen. Das liegt daran, dass die Parameterübergabe zum Datenfluss, nicht zum Kontrollfluss zählt und damit von der strukturierten Programmierung nicht erfasst wird. Anders ausgedrückt: Die Mächtigkeit von Funktionen als Daten steht im Gegensatz zur Einfachheit in der Verwendung. Glücklicherweise gibt es auch andere Wege, um Vollständigkeit zu erreichen. Wir können in Übereinstimmung mit der strukturierten Programmierung einen Fixpunkt-Operator hinzufügen, bzw. einfacher ausgedrückt, essentielle Kontrollstrukturen einbauen. Wenn wir schon dabei sind, können wir ganz in Übereinstimmung mit der strukturierten Programmierung auch weitere Kontrollstrukturen wie einen `if_then_else`-Ausdruck und `case`-Ausdruck (wie die `if`- bzw. `switch`-Anweisung in Java, aber als Ausdruck, nicht als Anweisung) einführen, um die Bequemlichkeit beim Programmieren weiter zu erhöhen.

Das heißt, um die Mächtigkeit der Turing-Maschine zu erreichen, sind keine Programmteile als Daten nötig; in vielen Bereichen des üblichen Programmieralltags verzichten wir gerne auf sie. In manchen Fällen sind Programmteile als Daten dennoch sehr wertvoll: Sie ermöglichen es, innerhalb einer Sprache selbst neue Konstrukte mit der Ausdruckskraft von Kontrollstrukturen einzuführen. Auch wenn wir das nur selten brauchen, ist es vorteilhaft, die Möglichkeit dazu zu haben. Es ist damit nicht nötig, alle Kontrollstrukturen schon im Sprachdesign vorzusehen, weil `if_then_else`-Ausdrücke, `case`-Ausdrücke und so weiter auch später jederzeit hinzugefügt werden können, abgesehen von speziell an die jeweiligen Kontrollstrukturen angepasster Syntax. Die Entwicklung solcher Kontrollstrukturen kann aufwändig sein, aber wenn wir sie einmal haben, sind sie einfach verständlich. Es ist nicht schwer, dafür zu sorgen, dass diese Kontrollstrukturen die Prinzipien der strukturierten Programmierung berücksichtigen. In Abschnitt 2.3.2 werden wir uns mit einem Programmierstil

beschäftigen, in dem vorwiegend Kontrollstrukturen zum Einsatz kommen, die auf die Verwendung von Funktionen als Daten aufbauen und entweder vorgefertigt aus umfangreichen Bibliotheken stammen oder selbst geschrieben werden. Eine Voraussetzung dafür, dass das gut funktioniert, ist ein passendes Umfeld; das gesamte System muss darauf ausgelegt sein.

Ein weiterer Grund spricht gegen die Einführung vieler als Daten verwendbarer Programmteile: Die uneingeschränkte Verwendbarkeit als Daten führt häufig zu einer großen Zahl an Sonderfällen, die in der Implementierung einer Programmiersprache berücksichtigt werden müssen. Das lässt den Implementierungsaufwand steigen und die Effizienz der Programme sinken. Außerdem verlagert die Verwendung von Programmteilen als Daten einige Tätigkeiten, die sonst ein Compiler statisch erledigen könnte, hin zur Laufzeit, wodurch einige Typprüfungen durch den Compiler nicht mehr machbar sind; darunter leidet auch die Sicherheit. Es will gut überlegt sein, was als Daten verwendbar sein soll und was nicht. Einfache Argumentationsketten sind als Entscheidungsgrundlagen ungeeignet. Nur mit viel Expertenwissen in mehreren Bereichen wie der Theorie der Programmierung, Typtheorie und Compilertechnologie können fundierte Entscheidungen getroffen werden. Viele Zusammenhänge haben sich als praktische Erfahrungen über mehrere Jahrzehnte langsam herauskristallisiert. Die Entscheidung dafür, einige Programmteile als Daten verwendbar zu machen, führt aufgrund inhärenter Zusammenhänge nicht selten automatisch dazu, dass andere Programmteile nicht sinnvoll als Daten verwendbar sind. Unterschiedliche Entscheidungen führen damit zu unterschiedlichen Programmierstilen. Die Verwendung von Funktionen als Daten führt zu funktionalen Stilen, die von Objekten als Daten zu objektorientierten. Natürlich wurden zahllose Versuche unternommen, diese beiden Paradigmen zu vereinen. Bis zu einem gewissen Grad gelingt das ganz gut, sodass in der gleichen Sprache sowohl funktional als auch objektorientiert programmiert werden kann. Aber eine vollständige Vereinigung ist wegen inhärenter Widersprüche nicht möglich. Funktionale Programmteile können nicht gleichzeitig objektorientiert sein. Beispielsweise sind Objekte in Java ganz selbstverständlich als Daten verwendbar und Lambdas können seit kurzem als Daten betrachtet werden, die für vollwertige Funktionen stehen. Es sind jedoch nur dafür vorgesehene Funktionen (solche, hinter denen funktionale Abstraktionen stecken) sinnvoll als Daten verwendbar, nicht beliebige Methoden. Dass das objektorientierte und funktionale Paradigma nicht zu einem Paradigma zusammenwachsen, hat prinzipielle Ursachen; es geht nicht um den Willen zur Zusammenarbeit.

Wie Lambdas in Java zeigen, ist es gar nicht notwendig, dass Funktionen als Daten verwendbar sind. Das Gleiche lässt sich durch Objekte erreichen, die in Variablen abgelegt, als Parameter übergeben oder als Ergebnisse zurückgegeben werden können: Als Typ eines solchen Objekts wird ein Interface mit nur einer Methode verwendet. Das Objekt muss diese Methode implementieren und über das Interface ist die Methode (und sonst nichts) aufrufbar. Wir müssen zwar um ein Eck herum denken, aber dieser Mechanismus funktioniert in jeder objektorientierten Sprache. Leider ist dieser Mechanismus nur für Methoden geeignet, die auf objektorientierte Formen der Abstraktion verzichten.

Wir können einen Schritt weiter gehen und uns fragen, was eine Methode (oder Funktion) von einem Objekt unterscheidet. Eine für viele von uns wahrscheinlich provokant klingende, aber bei entsprechender Interpretation richtige Antwort ist: Ein Objekt entspricht der Ausführung einer Methode. Einige eher experimentelle Programmiersprachen, aber auch JavaScript machen das deutlich: Eine Ausführungsinstanz einer Methode, also der *Activation-Record*[2], der neben Verwaltungsinformation die Parameter und lokalen Variablen der gerade ausgeführten Methode enthält, kann als Objekt betrachtet werden. Die lokalen Variablen und Parameter in der Ausführungsinstanz entsprechen Objektvariablen, wenn von einer anderen Methode aus darauf zugegriffen wird (in Java nicht möglich). Ein Methodenaufruf (ebenso wie ein Konstruktoraufruf) ist also gleichzeitig eine Objekterzeugung und umgekehrt. Solche Überlegungen lassen den Unterschied zwischen verschiedenen Arten als Daten verwendbarer Programmteile noch weiter schwinden.

1.3 Programmorganisation

Bei der *Programmierung im Groben* geht es um die Zerlegung großer Programme in überschaubare Einheiten. Nicht einzelne Variablen, Typen, Funktionen, etc. stehen im Blickpunkt, sondern größere Gruppen davon, sowie Beziehungen zwischen diesen Gruppen. Das wesentliche Ziel ist die *Modularisierung* von Programmen (auch *Faktorisierung* genannt). Sie soll so erfolgen, dass größtmögliche Flexibilität und Wartbarkeit über einen langen Zeitraum erzielt wird. Durch *Parametrisierung* und *Ersetzbarkeit* der durch die Modularisierung entstehenden Einheiten bekommen wir die für die langfristige Wartung nötige Flexibilität.

1.3.1 Modularisierung

Durch Modularisierung bringen wir größere Strukturen in Programme. Wir zerlegen das Programm in einzelne *Modularisierungseinheiten,* die nur lose voneinander abhängen und daher relativ leicht austauschbar sind. Verschiedene Formen von Modularisierungseinheiten sind unterscheidbar:

Modul: Darunter verstehen wir eine *Übersetzungseinheit*, also die Einheit, die ein Compiler in einem Stück bearbeitet, z. B. in Java ein Interface oder eine Klasse. Ein Modul enthält vor allem Deklarationen bzw. Definitionen von zusammengehörenden Variablen, Typen, Prozeduren, Funktionen, Methoden und Ähnlichem. Getrennt voneinander übersetzte Module werden von einem *Binder* (*Linker*) oder zur Laufzeit zum ausführbaren Programm verbunden. Wenn

[2] Alle Daten in einem Programm liegen entweder auf einem *Stack* (von wo sie am Lebensende leicht durch „pop“ entfernt werden können) oder am *Heap* (mit eher langlebigen Daten, die am Lebensende durch eine vergleichsweise aufwändige Speicherbereinigung entfernt werden). Die Ausführungsinstanz wird auch *Stack-Frame* genannt, weil sie meist am Stack liegt. Wenn jedoch nicht zwischen Objekt und Ausführungsinstanz unterschieden wird, liegt die Ausführungsinstanz am Heap, sodass Stack-Frame kein passender Name ist.

einzelne Module geändert werden, sind nur diese Module (sowie möglicherweise von ihnen abhängige Module, siehe unten) neu zu übersetzen, nicht alle Module. Diese Vorgehensweise beschleunigt die Übersetzung großer Programme wesentlich. Einzelne Module sind in wenigen Sekunden oder Minuten übersetzt, während die Übersetzung aller Module Stunden und Tage dauern kann.

Beim Programmieren zeigt sich ein weiterer Vorteil: Module lassen sich relativ unabhängig voneinander entwickeln, sodass mehrere Leute oder mehrere Teams gleichzeitig an unterschiedlichen Stellen eines Programms arbeiten können, ohne sich gegenseitig zu stark zu behindern.

Unter der *Schnittstelle* eines Moduls verstehen wir zusammengefasste Information über Inhalte des Moduls, die auch in anderen Modulen verwendbar sind. Nicht nur Java-Interfaces sind Schnittstellen. Klar definierte Schnittstellen sind überall hilfreich. Einerseits braucht der Compiler Schnittstelleninformation, um Inhalte anderer Module verwenden zu können, andererseits ist diese Information auch beim Programmieren wichtig, um Abhängigkeiten zwischen Modulen besser zu verstehen. Meist wird nur ein kleiner Teil des Modulinhalts in anderen Modulen benötigt. Schnittstellen unterscheiden klar zwischen Modulinhalten, die für andere Module zugreifbar sind, und solchen, die nur innerhalb des Moduls gebraucht werden. Erstere werden *exportiert,* letztere sind *privat.* Private Modulinhalte sind von Vorteil: Sie können vom Compiler im Rahmen der Programmiersprachsemantik beliebig optimiert, umgeformt oder sogar weggelassen werden, während für exportierte Inhalte eine Zugriffsmöglichkeit von außen bestehen muss, die gewisse Regeln einhält. Änderungen privater Modulinhalte wirken sich hinsichtlich der Konsistenz nicht auf andere Module aus. Änderungen exportierter Inhalte machen hingegen oft entsprechende Änderungen in anderen Modulen nötig, die diese Inhalte verwenden. Zumindest müssen Module, die geänderte Inhalte verwenden, neu übersetzt werden. Um Abhängigkeiten deutlicher zu machen, wird in Modulen häufig auch angegeben, welche Inhalte anderer Module verwendet werden. Diese Inhalte werden *importiert.*

Der explizite Import ermöglicht getrennte *Namensräume.* Innerhalb eines Moduls sind nur die Namen der in diesem Modul deklarierten bzw. definierten sowie importierten Inhalte sichtbar und nur diese Namen müssen eindeutig sein. Der gleiche Name kann in einem anderen Modul (also in einem anderen Namensraum) eine andere Bedeutung haben. Beim Programmieren sind Namen in anderen Modulen ignorierbar. Allerdings kann es vorkommen, dass aus unterschiedlichen Modulen unterschiedliche Inhalte des gleichen Namens zu importieren sind. Solche Namenskonflikte sind durch *Umbenennung* während des Importierens oder durch *Qualifikation* des Namens (das ist das Voranstellen des Modulnamens vor den importierten Namen) auflösbar.

Module im ursprünglichen Sinn können nicht zyklisch voneinander abhängen. Wenn ein Modul B Inhalte eines Moduls A importiert, kann A keine Inhalte von B importieren. Das hat mit der getrennten Übersetzung zu tun: Modul A muss vor B übersetzt werden, damit der Compiler während der Übersetzung von B bereits auf die übersetzten Inhalte von A zugreifen kann. Würde A auch Inhalte von B importieren, könnten A und B nur gemeinsam übersetzt werden, was der Definition und Idee von Modulen widerspricht. Manchmal ist

die gemeinsame Übersetzung voneinander zyklisch abhängiger Module dennoch erlaubt. Wesentliche Vorteile ergeben sich aber nur bei getrennter Übersetzung.

Zyklen in den Abhängigkeiten lassen sich durch Aufspaltung in je zwei getrennte Module auflösen, wobei eines Schnittstelleninformation und das andere die Implementierung enthält. Beispielsweise sind in Java Interfaces und Klassen, welche die Interfaces implementieren, getrennte Übersetzungseinheiten. Schnittstelleninformationen hängen in der Regel nicht zyklisch voneinander ab. Dagegen hängen Implementierungen häufig gegenseitig von Schnittstellen anderer Module ab. Die Trennung ermöglicht getrennte Übersetzungen: Zuerst werden die Schnittstellen getrennt voneinander übersetzt. Da Implementierungen nicht direkt auf andere Implementierungen, sondern auf Schnittstellen zugreifen, sind danach die Implementierungen getrennt voneinander übersetzbar.

Die in Java 9 hinzugefügten sogenannten Java-Module (Dateien, deren Inhalte mit `module` beginnen) sind selbst keine Module im hier beschriebenen Sinn, sondern verdeutlichen Schnittstellenspezifikationen von Klassen und Interfaces aus Modulsicht und klären damit Abhängigkeiten zwischen Modulen.

Objekt. Anders als Module sind Objekte keine Übersetzungseinheiten und werden erst zur Laufzeit erzeugt. Daher gibt es keine derart starken Einschränkungen hinsichtlich zyklischer Abhängigkeiten wie bei Modulen. Abgesehen davon haben Objekte eine ähnliche Zielsetzung wie Module: Sie kapseln Variablen und Methoden zu logischen Einheiten (*Kapselung*) und schützen private Inhalte vor Zugriffen von außen (*Data-Hiding*). Wie Module stellen sie Namensräume dar und sorgen dafür, dass sich Änderungen privater Teile nicht auf andere Objekte auswirken. Kapselung und Data-Hiding zusammen nennen wir *Datenabstraktion.* Dieses Wort deutet an, wozu wir Objekte einsetzen: Wir betrachten ein Objekt als rein abstrakte Einheit, die ein in unserer Vorstellung existierendes „Etwas“ auf der Ebene der Software realisiert. Zwar ist eine solche Abstraktion auch mit Modulen erreichbar, aber durch die Vermischung mit dem zusätzlichen Verwendungszweck als Übersetzungseinheit nur sehr eingeschränkt.

Anders als Module sind Objekte immer als Daten verwendbar. Zu den wichtigsten Eigenschaften von Objekten zählen *Identität, Zustand* und *Verhalten.* Im Prinzip haben auch Module diese Eigenschaften. Jedoch ist die Identität eines Moduls mit dessen eindeutigem Namen und Verhalten gekoppelt, wodurch unterschiedliche Module immer unterschiedliches Verhalten aufweisen. Objekte haben keinen eindeutigen Namen und es kann mehrere Objekte mit dem gleichen Verhalten geben. Erst dadurch gewinnen Begriffe wie Identität und Gleichheit Bedeutung: Zwei durch verschiedene Variablen referenzierte Objekte sind *identisch,* wenn es sich um ein und dasselbe Objekt handelt. Zwei Objekte sind *gleich,* wenn sie den gleichen Zustand und das gleiche Verhalten haben, auch wenn sie nicht identisch sind. Dann ist ein Objekt eine *Kopie* des anderen.

Es gibt eine breite Palette an Möglichkeiten zur Festlegung der Details. So ist Data-Hiding in jeder objektorientierten Sprache etwas anders realisiert und neue Objekte werden auf ganz unterschiedliche Weise erzeugt und initialisiert. Beispielsweise entstehen neue Objekte in der Programmiersprache Self [33] nur

durch Kopieren bereits bestehender Objekte und es sind keine Klassen und ähnliche Konzepte nötig. Meist verwenden wir aber Klassen und Konstruktoren für die Initialisierung.

Klasse. Eine Klasse wird häufig als Schablone für die Erzeugung neuer Objekte beschrieben. Sie gibt die Variablen und Methoden des neuen Objekts vor und spezifiziert ihre wichtigsten Eigenschaften, jedoch nicht die Werte der Variablen. Alle Objekte der gleichen Klasse haben gleiches Verhalten. Objekte unterschiedlicher Klassen verhalten sich unterschiedlich, sodass nur Objekte der gleichen Klasse gleich oder identisch sein können. Der Begriff *Klasse* kommt von der Klassifizierung anhand des Verhaltens. In diesem Sinn betrachten wir auch Java-Interfaces als Klassen.

Meist ist es möglich, Klassen von anderen Klassen abzuleiten und dabei Methoden zu erben. Auf eine bestimmte Art angewandt ergeben sich durch Klassenableitungen auf vielfältige Weise strukturierbare Klassifizierungen von Objekten, in denen ein einzelnes Objekt gleichzeitig mehrere Typen haben kann. Zusammen mit abgeleiteten Klassen sind auch *abstrakte Klassen* sinnvoll, von denen zwar andere Klassen ableitbar, aber keine Objekte erzeugbar sind. Interfaces wie in Java sind einfach nur eine Spezialform von abstrakten Klassen.

Zwecks Datenabstraktion sollten Objekte zwischen exportierten und privaten Inhalten unterscheiden. Da diese Unterscheidung für alle Objekte der gleichen Klasse gleich ist, wird das meist auf der Klassenebene spezifiziert. Allerdings sind die Kriterien häufig aufgelockert. Beispielsweise können auch andere Objekte der gleichen Klasse auf private Inhalte zugreifen, und es gibt mehrere Stufen der Sichtbarkeit. Dadurch, dass neue Objekte nur über Klassen (oder durch Kopieren) erzeugt werden, sind trotz größerer Flexibilität die Ziele des Data-Hiding dennoch erreichbar.

Wie in Java ist eine Klasse oft auch ein Modul und damit eine Übersetzungseinheit. Statische Variablen und Methoden sowie Konstruktoren entsprechen Modulinhalten und zyklische Abhängigkeiten zwischen Klassen sind verboten. Durch Ableitung von (abstrakten) Klassen oder Interfaces lassen sich zyklische Abhängigkeiten für nicht-statische Methoden immer auflösen. Statische Methoden sind in Java nicht in getrennten Schnittstellen beschreibbar, sodass diese Technik dafür nicht nutzbar ist.

Komponente. Eine Komponente ist ein eigenständiges Stück Software, das in ein Programm eingebunden wird. Für sich alleine ist eine Komponente nicht lauffähig, da sie die Existenz anderer Komponenten voraussetzt und deren Dienste in Anspruch nimmt.

Komponenten ähneln Modulen: Beides sind Übersetzungseinheiten und Namensräume, die Datenkapselung und Data-Hiding unterstützen. Komponenten sind flexibler: Während ein Modul Inhalte ganz bestimmter, namentlich genannter anderer Module importiert, importiert eine Komponente Inhalte von zur Übersetzungszeit nicht genau bekannten anderen Komponenten. Erst beim Einbinden in ein Programm werden diese anderen Komponenten bekannt. So-

wohl bei Modulen als auch Komponenten ist offen, wo exportierte Inhalte verwendet werden, aber bei Komponenten ist zusätzlich offen, von wo importierte Inhalte kommen. Letzteres verringert die Abhängigkeit der Komponenten voneinander. Deswegen gibt es bei der getrennten Übersetzung kein Problem mit zyklischen Abhängigkeiten.

Das Einbinden von Komponenten in Programme nennen wir *Deployment*. Es ist aufwändiger als das Einbinden von Modulen, da auch die Komponenten festgelegt werden müssen, von denen etwas importiert wird. Oft werden zuerst die einzubindenden Komponenten zum Programm hinzugefügt und erst in einem zweiten Schritt festgelegt, von wo importiert wird. Diese Vorgehensweise ermöglicht die Einführung zyklischer Abhängigkeiten zwischen Komponenten, so wie zyklische Abhängigkeiten zwischen Objekten erst nach der Objekterzeugung entstehen können. Das Deployment kann statisch vor der Programmausführung oder dynamisch zur Laufzeit erfolgen.

Namensraum. Jede oben angesprochene Modularisierungseinheit bildet einen eigenen Namensraum und kann damit Namenskonflikte abfedern. Das gilt jedoch nicht für globale Namen, die außerhalb dieser Modularisierungseinheiten stehen, etwa für die Namen von Modulen, Klassen und Komponenten. Wir müssen auch globale Namen verwalten. Häufig werden Modularisierungseinheiten in andere Modularisierungseinheiten gepackt, z. B. innere Klassen in äußere. Dies ist zwar für die Datenabstraktion sinnvoll, aber nicht für die Verwaltung globaler Namen, da das Ineinanderpacken die getrennte Übersetzung behindert.

Manche Sprachen bieten keine Unterstützung für die globale Namensverwaltung. Beim Anwenden von Werkzeugen, z. B. Compilern, sind alle Dateien anzuführen, die benötigte Modularisierungseinheiten enthalten. Dieser vor allem aus C bekannte Ansatz ist flexibel, aber recht unsicher.

Etwas fortgeschrittener sind Modularisierungseinheiten, die wir Namensräume nennen. Sie fassen mehrere Modularisierungseinheiten zu einer Einheit zusammen, ohne die getrennte Übersetzbarkeit zu stören. Meist entstehen dabei hierarchische Strukturen, die Verzeichnisstrukturen ähneln und manchmal tatsächlich auf Verzeichnisse abgebildet werden, z. B. Pakete in Java. Namen werden dadurch komplexer. Beispielsweise bezeichnet `a.b.C` die Klasse `C` im Namensraum `b` welcher im Namensraum `a` steht.

Trotz der Verwendung von Namensräumen sind solche Namen immer relativ zu einer Basis und daher nur in einer eingeschränkten Sichtweise global. In letzter Zeit verwenden wir vermehrt tatsächlich global eindeutige Namen zur Adressierung von Modularisierungseinheiten, vor allem für öffentlich sichtbare. Meist werden diese Einheiten wie Webseiten durch ihre URI-Adressen bezeichnet. Solche Adressen wurden ja extra dafür geschaffen, Ressourcen in einem globalen Umfeld eindeutig zu bezeichnen.

Die Art einer Modularisierungseinheit hängt nicht von der Größe ab. Von jeder Art gibt es kleine und große Exemplare. Auch bei der Schachtelung von Modularisierungseinheiten sind viele Varianten denkbar. So können etwa Module in Komponenten enthalten sein, aber auch Komponenten in Modulen.

1.3.2 Parametrisierung

Ein bekanntes Schlüsselkonzept zur Steigerung der Flexibilität von Modularisierungseinheiten ist deren Parametrisierung. Darunter verstehen wir im weitesten Sinn, dass in den Modularisierungseinheiten belassene Lücken erst später befüllt werden. Ein Beispiel dafür haben wir schon betrachtet: Aus einem Modul wird eine Komponente, wenn zunächst offen bleibt, von welchen anderen Komponenten etwas importiert wird; erst beim Zusammensetzen des Systems werden diese Komponenten bestimmt. Im Allgemeinen bleiben beliebige Teile offen, und das Befüllen der Lücken erfolgt zu unterschiedlichen Zeitpunkten aus verschiedenen Quellen.

Befüllen zur Laufzeit. Am einfachsten ist das Befüllen der Lücken zur Laufzeit, wenn gewöhnliche Daten (elementare Werte oder als Daten verwendbare Programmteile) einzufüllen sind. In diesem Fall werden die Lücken durch einfache Variablen dargestellt. Beim Befüllen werden ihnen Werte zugewiesen. Allerdings befinden sich die Variablen üblicherweise nicht an der Stelle im Programm, an der die zuzuweisenden Werte bekannt sind. Die Werte können auf unterschiedliche Weise zu den Variablen gebracht werden, unter anderem so:

Konstruktor: Beim Erzeugen eines Objekts wird ein Konstruktor ausgeführt, der die Objektvariablen initialisiert. Der Konstruktor hat Parameter. An der Stelle, an der die Objekterzeugung veranlasst wird, werden Werte als Argumente an den Konstruktor übergeben, die zur Initialisierung verwendet werden. Das ist die häufigste und einfachste Form der Parametrisierung, nicht nur in objektorientierten Sprachen.

Initialisierungsmethode: In einigen Fällen sind Konstruktoren nicht verwendbar, beispielsweise wenn Objekte durch Kopieren erzeugt werden oder zwei zu erzeugende Objekte voneinander abhängen; wir können ja das später erzeugte Objekt nicht an den Konstruktor des zuerst erzeugten Objekts übergeben. Solche Probleme sind durch Methoden lösbar, die unabhängig von der Objekterzeugung zur Initialisierung eines bereits bestehenden Objekts aufgerufen werden. Objekte werden also in einem ersten Schritt erzeugt und in einem zweiten initialisiert, bevor sie verwendbar sind. Diese Technik funktioniert nur in imperativen Paradigmen.

Zentrale Ablage: Eine weitere Möglichkeit besteht darin, Werte an zentralen Stellen (etwa in globalen Variablen oder als Konstanten) abzulegen, von wo sie bei der Objekterzeugung oder erst bei der Verwendung abgeholt werden. In letzterer Variante ist diese Technik auch für statische Modularisierungseinheiten verwendbar, die bereits zur Übersetzungszeit feststehen. Zum Abholen der Werte wird direkt auf die Variablen oder Konstanten zugegriffen oder es werden Methoden verwendet. Klassen können Konstanten z. B. von einem Interface erben, um Werte „abzuholen“.

Von diesen Techniken sind unzählige Verfeinerungen vorstellbar, die gerade in der objektorientierten Programmierung häufig ausgereizt werden.

Diese Techniken eignen sich zur *Dependency-Injection* (Einbringen von Abhängigkeiten). Dabei wird die Verantwortung für das Erzeugen und Initialisieren von Objekten an eine zentrale Stelle (z. B. eine Klasse) übertragen, von der aus die Abhängigkeiten zwischen den Objekten überblickbar und steuerbar sind.

Generizität. Unter Generizität verstehen wir eine Form der Parametrisierung, bei der Lücken zumindest konzeptuell bereits zur Übersetzungszeit befüllt werden. Daher können alle Arten von Modularisierungseinheiten außer Objekten generisch sein (also Lücken enthalten, die mittels Generizität befüllt werden), aber beispielsweise auch Funktionen und Ähnliches. Die Lücken werden zunächst durch generische Parameter bezeichnet. In allen Lücken, die mit demselben befüllt werden sollen, steht auch derselbe generische Parameter. Später, aber noch vor der Programmausführung werden die generischen Parameter durch das Einzufüllende ersetzt. Bevorzugt stehen generische Parameter nicht für gewöhnliche Werte, sondern für Konzepte, die keine Daten sind. Häufig sind das Typen. In diesem Fall nennen wir generische Parameter auch *Typparameter*. Generizität ist also vorwiegend für solche Fälle gedacht, wo das Befüllen der Lücken zur Laufzeit nicht funktioniert.

Grundsätzlich ist Generizität einfach und z. B. in Ada und C++ schon lange erprobt. In der Praxis ergeben sich aber Schwierigkeiten. So muss der Compiler mehrere Varianten des Codes verwalten, den generischen Quellcode sowie eine oder mehrere durch Füllen der Lücken generierte Variante(n). Fehlermeldungen, die sich auf generierten Code beziehen, sind für Programmierer_innen kaum verständlich auszudrücken. Oft sind Einschränkungen auf generischen Parametern nötig, weil das dafür Einzufüllende bestimmte Bedingungen erfüllen muss. Da es sich nicht um Daten handelt, lassen sich derartige Einschränkungen nur schwer ausdrücken. Besonders schwierig wird es, wenn für mehrere Vorkommen desselben generischen Parameters unterschiedliche Einschränkungen gelten.

Annotationen. Annotationen sind optionale Parameter, die an unterschiedlichste Sprachkonstrukte anheftbar sind. Ein Beispiel dafür ist die Annotation überschriebener Methoden in Java mittels `@Override`. Annotationen werden von Werkzeugen verwendet oder einfach ignoriert. So gibt ein Compiler, der `@Override` versteht, in manchen Situationen Warnungen aus, während andere Compiler und Werkzeuge, die die Annotation nicht kennen, diese unberücksichtigt lassen. Diese Form der Annotationen wirkt sich statisch, also zur Übersetzungszeit aus. Annotationen sind häufig auch dynamisch, also zur Laufzeit abfragbar. Über spezielle Funktionen oder Ähnliches lässt sich erfragen, mit welchen Annotationen ein Sprachkonstrukt versehen ist. Alles funktioniert so, als ob keine Annotationen vorhanden wären, solange die Annotationen nicht explizit abgefragt und entsprechende Aktionen gesetzt werden. Annotationen ähneln also Kommentaren, die aber bei Bedarf auch zur Steuerung des Programmablaufs herangezogen werden können.

Wie Generizität eignen sich Annotationen nur für statisch bekannte Informationen. Die Lücken, die durch Annotationen befüllt werden, sind im Gegensatz

zur Generizität nirgends im Programm festgelegt. Daher ist die Art und Weise, wie die mitgegebenen Informationen zu verwenden sind, ebenso unterschiedlich wie die Anwendungsgebiete. In der Praxis werden Annotationen oft in Situationen eingesetzt, wo Informationen nicht nur von lokaler Bedeutung sind, sondern auch System-Werkzeuge (wie einen Compiler oder das Betriebssystem) steuern oder zumindest beeinflussen. Über Annotationen werden auch häufig Spracherweiterungen eingeführt. In diesem Fall ist das Programm nicht sinnvoll, wenn Annotationen nicht verstanden werden, aber die Erweiterungen sind mit relativ kleinem Aufwand durchführbar, ohne vorgegebene Standards zu verletzen.

Aspektorientierte Programmierung. Auch bei der aspektorientierten Programmierung ist in der Regel keine Spezifikation von Lücken im Programm nötig. Stattdessen werden zu einem bestehenden Programm von außen sogenannte *Aspekte* hinzugefügt. Ein Aspekt spezifiziert eine Menge von Punkten im Programm, etwa alle Stellen, an denen bestimmte Methoden aufgerufen werden, sowie das, was an diesen Stellen passieren soll, etwa vor oder nach dem Aufruf bestimmten zusätzlichen Code ausführen oder den Aufruf durch einen anderen ersetzen. Ein *Aspect-Weaver* genanntes Werkzeug angewandt auf das Programm und die Aspekte modifiziert das Programm entsprechend der Aspekte. Meist geschieht dies vor der Übersetzung des Programms, manche Aspect-Weaver erledigen diese Aufgabe erst zur Laufzeit. Beispielsweise ist über Aspekte recht einfach erreichbar, dass bestimmte Aktionen im Programm zuverlässig in einer Log-Datei protokolliert werden, ohne den Quellcode des Programms dafür ändern zu müssen. Die Aspekte lassen sich vor der Übersetzung des Programms leicht austauschen oder weglassen, sodass auf einfache Weise ganz unterschiedliches Programmverhalten erreicht wird. Über Aspekte wird etwa die Generierung bestimmter Debug-Information veranlasst und später wieder weggenommen.

Bestimmte Aufgaben lassen sich durch Aspekte überzeugend rasch und einfach lösen. Für andere Aufgaben sind Aspekte kaum geeignet. Ein Problem besteht darin, dass die Bestimmung der betroffenen Punkte im Programm Wissen über Implementierungsdetails voraussetzt. Wenn sich solche Details ändern, müssen auch die Aspekte angepasst werden.

Parametrisierung steigert zwar die Flexibilität von Modularisierungseinheiten, aber ein Problem bleibt bei allen Formen der Parametrisierung bestehen: Die Änderung einer Modularisierungseinheit macht mit hoher Wahrscheinlichkeit auch Änderungen an allen Stellen nötig, an denen diese Modularisierungseinheit verwendet wird. Konkret: Wenn die Lücken sich ändern, dann muss sich auch das ändern, was zum Befüllen der Lücken verwendet wird. Solche notwendigen Änderungen behindern die Wartung gewaltig. Vor allem müssen für die Änderungen alle Stellen bekannt sein, an denen eine Modularisierungseinheit verwendet wird. Bei Modularisierungseinheiten, die in vielen unterschiedlichen Programmen über die ganze Welt verstreut zum Einsatz kommen, ist das so gut wie unmöglich. Nachträgliche Änderungen der Lücken in solchen Modularisierungseinheiten sind dadurch praktisch kaum durchführbar.

1.3.3 Ersetzbarkeit

Eine Möglichkeit zur praxistauglichen Änderung von Modularisierungseinheiten verspricht der Einsatz von *Ersetzbarkeit* statt oder zusätzlich zur Parametrisierung: Eine Modularisierungseinheit A ist durch eine andere Modularisierungseinheit B ersetzbar, wenn ein Austausch von A durch B keine Änderungen an Stellen nach sich zieht, an denen A (nach dessen Ersetzung B) verwendet wird.

Leider ist es recht kompliziert, im Detail festzustellen, unter welchen Bedingungen A durch B ersetzbar ist. Diese Bedingungen hängen nicht nur von A und B selbst ab, sondern auch davon, was, von außen betrachtet, von A und B erwartet wird. Daher ist Ersetzbarkeit nur für Modularisierungseinheiten anwendbar, die alle erlaubten Betrachtungsweisen von außen klar festlegen. Das geht Hand in Hand mit klar definierten Schnittstellen. Ersetzbarkeit zwischen A und B ist dann gegeben, wenn die Schnittstelle von B das Gleiche beschreibt wie die von A. Jedoch kann die Schnittstelle von B mehr Details festlegen als die von A, also etwas festlegen, was in A noch offen ist. Entsprechende Schnittstellen sind auf verschiedene Weise spezifizierbar:

Signatur: In der einfachsten Form spezifiziert eine Schnittstelle nur, welche Inhalte der Modularisierungseinheit von außen zugreifbar sind. Diese Inhalte werden über ihre Namen und gegebenenfalls die Typen von Parametern und Ergebnissen beschrieben. Die Bedeutung der Inhalte bleibt offen. Wir nennen eine solche Schnittstelle *Signatur* der Modularisierungseinheit. In Kapitel 3 werden wir sehen, dass Ersetzbarkeit für Signaturen einfach und klar definiert ist und auch von einem Compiler überprüft werden kann. Im Wesentlichen muss B alles enthalten und von außen zugreifbar machen, was auch in A von außen zugreifbar ist, kann aber mehr enthalten als A. Wenn wir uns hinsichtlich der Ersetzbarkeit jedoch nur auf Signaturen verlassen, kommt es leicht zu Irrtümern. Ein Inhalt von B könnte eine ganz andere Bedeutung haben als der gleichnamige Inhalt von A. Es passiert etwas Unerwartetes, wenn statt des Inhalts von A der entsprechende Inhalt von B verwendet wird. Dennoch verlassen wir uns eher auf Signaturen, als ganz auf Ersetzbarkeit zu verzichten.

Abstraktion realer Welt: Schnittstellen werden neben Signaturen auch durch Namen und informelle Texte beschrieben, welche die Modularisierungseinheiten charakterisieren. Diese Schnittstellen entsprechen abstrakten Sichtweisen von Objekten aus der realen Welt. Aufgrund von Alltagserfahrungen können wir recht gut abschätzen, ob eine solche Abstraktion als Ersatz für eine andere angesehen werden kann. Beispielsweise ist ein Auto genauso wie ein Fahrrad ein Fahrzeug; wir können ein Fahrzeug durch ein Auto oder Fahrrad ersetzen, aber ein Auto ist kein Fahrrad und das Fahrrad nicht durch ein Auto ersetzbar. In der objektorientierten Programmierung spielen solche Abstraktionen eine wichtige Rolle: Ersetzbarkeit haben wir nur, wenn sowohl die Signaturen als auch die Abstraktionen passen, sodass Irrtümer unwahrscheinlich sind. Jedoch beruht dieser Ansatz auf Intuition und kann in die Irre führen.

Zusicherungen: Um Fehler auszuschließen, ist eine genaue Beschreibung der erlaubten Erwartungen an eine Modularisierungseinheit nötig. Diese Beschreibung bezieht sich auf die Verwendungsmöglichkeiten aller nach außen sichtbaren Inhalte. In der objektorientierten Programmierung hat sich für solche Beschreibungen der Begriff *Design-by-Contract* etabliert. Dabei entspricht die Schnittstelle einem Vertrag zwischen einer Modularisierungseinheit (als Server) und ihren Verwendern (Clients). Der Vertrag legt in *Zusicherungen* fest, was sich der Server von den Clients erwarten kann (das sind *Vorbedingungen*), was sich die Clients vom Server erwarten können (*Nachbedingungen*), welche Eigenschaften in konsistenten Programmzuständen immer erfüllt sind (*Invarianten*), wie sich Zustände ändern und in welchen Aufruf-Reihenfolgen Clients mit dem Server interagieren können (*History-Constraints*). Theoretisch sind über diese Arten von Zusicherungen die erlaubten Erwartungen beliebig genau beschreibbar. Es ist auch klar geregelt, wie sich Zusicherungen aus unterschiedlichen Schnittstellen zueinander verhalten müssen, damit Ersetzbarkeit gegeben ist. In der Praxis ergeben sich jedoch Probleme. Häufig sind Zusicherungen nur informell und nicht präzise. Vor allem komplexere Vorbedingungen stehen nicht selten in Konflikt zu Data-Hiding, weil sie von Programmzuständen abhängen, die eigentlich nach außen nicht sichtbar werden sollten. Meist wird die Einhaltung von Zusicherungen, wenn überhaupt, erst zur Laufzeit überprüft; dann ist es dafür eigentlich schon zu spät.

Überprüfbare Protokolle: In jüngerer Zeit wurden Techniken entwickelt, die formale Beschreibungen erlaubter Erwartungen auf eine Weise ermöglichen, dass bereits der Compiler deren Konsistenz überprüfen kann. Genau genommen spezifizieren solche Schnittstellen Kommunikationsprotokolle zwischen Modularisierungseinheiten. Die Protokolle unterscheiden sich darin, ob nur die Beziehung zwischen einem Client und Server geregelt wird, oder zwischen mehreren Einheiten gleichzeitig. Natürlich können die Protokolle auch auf ganz unterschiedliche Weise ausgedrückt sein. Es gibt nicht nur einen sinnvollen Ansatz, sondern viele verschiedene, die sich in ihren Eigenschaften grundsätzlich voneinander unterscheiden. Generell dürfen die Protokolle nicht beliebig komplex sein, da die Konsistenz sonst nicht mehr entscheidbar ist. Für praktische Anwendungen würde die Mächtigkeit jedenfalls ausreichen. Allerdings sind alle solchen Ansätze noch recht neu und weit von einer praktischen Realisierung in einer etablierten Programmiersprache entfernt. Ob die Erwartungen erfüllt werden, kann erst die fernere Zukunft zeigen.

In der objektorientierten Programmierung steht die Ersetzbarkeit ganz zentral im Mittelpunkt. Programme werden so gestaltet, dass jeder Programmteil möglichst problemlos durch einen anderen Programmteil ersetzbar ist. Einerseits ist auf Ersetzbarkeit von Objekten innerhalb eines Programms zu achten. Programmteile sind vielfältig einsetzbar, wenn sie zwar Objekte einer bestimmten Art erwarten, aber trotzdem auf allen Objekten operieren können, durch die die erwarteten Objekte ersetzbar sind. Dies ermöglicht einfache Erweiterungen

des Programms, ohne dabei schon existierenden Code ständig anpassen zu müssen. Andererseits bietet die Ersetzbarkeit (nicht nur in der objektorientierten Programmierung) eine Grundlage für die Erzeugung neuer Programmversionen, die mit Ihrer Umgebung trotz Erweiterungen kompatibel bleiben.

Als Basis für Schnittstellenbeschreibungen verwenden wir in der objektorientierten Programmierung Abstraktionen der realen Welt, in jüngerer Zeit meist gepaart mit Zusicherungen. Abstraktionen werden häufig über Klassen realisiert. Ersetzbarkeit wird nur dann als gegeben angesehen, wenn es auch eine Klassenableitung gibt. Klassenableitungen bekommen dadurch eine doppelte Bedeutung: Sie sind Grundlage sowohl für Vererbung als auch Ersetzbarkeit. Obwohl Vererbung aufgrund der Definition nichts mit Ersetzbarkeit zu tun hat, werden diese Begriffe als zusammengehörig betrachtet. Dennoch ist es in der praktischen Programmierung notwendig, Vererbung klar von Ersetzbarkeit zu unterscheiden, da andernfalls unvermeidliche Zielkonflikte zu versteckten schweren Fehlern führen.

In anderen Programmierparadigmen ist Ersetzbarkeit zur Erzeugung neuer Programmversionen genauso wichtig, aber meist fehlen Sprachmechanismen, um Ersetzbarkeit sicherzustellen. Im besten Fall gibt es noch eine Unterstützung zur Überprüfung von Signaturen.

1.4 Abstraktion

In der Programmierung spielen Abstraktionen eine sehr wichtige Rolle. Allerdings wird dieser Begriff für viele Dinge verwendet, die zwar irgendetwas miteinander zu tun haben, sich aber doch voneinander unterscheiden. Wir haben in Abschnitt 1.1.2 erfahren, dass λ-Abstraktion ein anderer Begriff für Funktion ist, also jede Funktion, Prozedur, Methode und Ähnliches eine Abstraktion darstellt. In Abschnitt 1.1.3 haben wir Abstraktion als einen Erfolgsfaktor für Programmierparadigmen beschrieben, wobei es vorwiegend um die Abstraktion über Details der Hardware und des Betriebssystems geht, mit Portabilität als Ziel. In Abschnitt 1.3.3 haben wir Abstraktion über die reale Welt als ein Konzept zur Beschreibung von Schnittstellen eingeführt, das intuitive Beziehungen zwischen unterschiedlichen Einheiten herstellt. Im Folgenden vertiefen wir die Sichtweise von Abstraktion als Konzept, das intuitive Vorstellungen in die ansonsten unpersönliche, formale Welt der Programmierung bringt. Gut gewählte Abstraktionen können die Komplexität einer Aufgabe erheblich reduzieren.

1.4.1 Prinzip der Abstraktion

Beim Programmierenlernen wird uns immer wieder das Abstraktionsprinzip[3] gepredigt: Duplikate von Programmtexten sind zu vermeiden, stattdessen sind Funktionen, Prozeduren, Methoden und Ähnliches einzusetzen, welche die an mehreren Stellen benötigten Programmtexte nur einmal enthalten, diese sind

[3] *Abstraktionsprinzip* ist auch ein Begriff aus dem Wirtschaftsrecht, der jedoch nichts mit dem hier verwendeten Prinzip zu tun hat.

an den entsprechenden Stellen aufzurufen. Diese Form der Abstraktion hängt eindeutig mit der λ-Abstraktion zusammen. Natürlich ist es vorteilhaft, wenn nur ein Programmtext statt vieler Duplikate des gleichen Programmtexts gewartet werden muss. Der Begriff „Abstraktion“ deutet an, dass damit intuitive Vorstellungen in die Programmierung gebracht werden: Die Funktionen, Prozeduren, Methoden, etc. haben Namen und sollten von Kommentaren begleitet sein, welche die Absichten dahinter klar machen. Zwar existieren Abstraktionen auch ohne Namen und Beschreibungen, sind dann aber nicht leicht greifbar.

In gewisser Weise wird mit dem Abstraktionsprinzip das Pferd von hinten aufgezäumt: Es sollte nicht so sein, dass zuerst Duplikate eingeführt werden, die danach durch Abstraktion beseitigt werden. Besser wäre es, gleich abstrakt zu denken, wodurch Duplikate von vorne herein vermieden werden und uns gar nicht in den Sinn kommt, dass es um die Vermeidung von Duplikaten geht. Unter Duplikaten sind diesbezüglich auch nicht nur genau gleiche Programmtexte zu verstehen, sondern solche mit gleicher beabsichtigter Wirkung. Es geht also um die Absicht dahinter, die einem Werkzeug (z. B. dem Compiler) nicht bekannt ist. Betrachten wir dazu einige Beispiele:

```
// swap a[i] and a[j]; i != j
void swap(int[] a,int i,int j){
    int h = a[i];
    a[i] = a[j];
    a[j] = h;
}
```

```
// swap a[i] and a[j]; i != j
void swap(int[] a,int i,int j){
    a[i] ^= a[j];
    a[j] ^= a[i];
    a[i] ^= a[j];
}
```

Die beiden `swap`-Varianten enthalten unterschiedliche Programmtexte, aber die Absicht dahinter dürfte gleich sein – links die übliche Vertauschung mit einer Hilfsvariable, rechts eine Vertauschung mit einem bitweise auf ganze Zahlen angewandten XOR. Diese beiden Methoden bewirken nicht das Gleiche, weil die rechte Variante nur funktioniert, wenn `i` und `j` voneinander verschieden sind. Trotzdem eignen sich die beiden Methoden für den im Kommentar beschriebenen Einsatzzweck, stellen also die gleiche Abstraktion dar. Wir sehen auch, dass Kommentare eine wesentliche Rolle spielen, wenn sie Absichten ausdrücken, die von der Semantik einer Programmiersprache nicht erfasst werden.

Umgekehrt kann es auch so sein, dass unterschiedliche Abstraktionen durch den gleichen Programmtext ausdrückbar sind:

```
// x smaller y if result <  0
// x equals y  if result == 0
// x larger y  if result >  0
int compare(int x, int y) {
    return x - y;
}
```

```
// difference between x and y
int subtract(int x, int y) {
    return x - y;
}
```

Wir wollen `compare` nicht als äquivalent zu `subtract` betrachten, da hinter diesen Methoden unterschiedliche Absichten stecken. Es hat sich bewährt, unterschiedliche Dinge auch bei zufällig gleichem Programmtext als unterschiedlich zu betrachten, weil damit gerechnet werden muss, dass sich Abstraktionen

im Laufe der Zeit weiterentwickeln, bei unterschiedlichen Absichten auf unterschiedliche Weise. Vielleicht stellt es sich irgendwann als günstig heraus, Ergebniswerte von `compare` auf `-1`, `0` und `1` zu beschränken, was die Absichten unverändert lässt, aber zu anderem Programmtext als in `subtract` führt.

Gute Abstraktionen beruhen immer auf bestimmten Absichten und wirken damit auch auf einer intuitiven Ebene. Eine restriktive Verfolgung des Abstraktionsprinzips zur Vermeidung von Duplikaten ist dabei nicht immer hilfreich. Stattdessen wäre es sinnvoll, sich eine abstrakte Denkweise anzueignen, also von vorne herein in Abstraktionen, nicht in konkreten Programmtexten zu denken. Natürlich müssen wir konkrete Programmtexte lesen können, sollten dabei aber stets auch versuchen, die Abstraktionen zu sehen, die beim Schreiben der Programmtexte maßgebend waren.

Eine Frage, die sich immer wieder stellt, ist die nach der richtigen Granularität, also Größe der Abstraktionseinheiten (Funktionen, Prozeduren, Methoden, etc.). Abstraktionen existieren auf allen Granularitätsstufen, vom Einzeiler bis zu sehr vielen Zeilen. Eine (außer aus einem pragmatischen Gesichtspunkt) durch nichts belegbare Regel besagt, dass eine Einheit auf einer Bildschirmseite Platz haben soll. Genau genommen ist diese Antwort wenig sinnvoll, weil es nicht um die Anzahl der Zeilen geht, sondern um die Komplexität der abstrakten Vorstellung, der Absicht hinter dieser Einheit; es gibt sehr komplexe Einzeiler genauso wie lange und trotzdem simple Programmteile. Es ist auch nicht sinnvoll zu fragen, ab welcher Komplexität Abstraktionen eingesetzt werden sollen und bis zu welcher Komplexität die elementaren, durch die Programmiersprache vorgegebenen Operationen reichen, weil jede noch so elementare Operation schon von vorne herein eine Abstraktion ist. Zumindest abstrahieren elementare Operationen von der darunter liegenden Hardware und verstecken dabei unnötige Details. Sinnvoller ist eher die Frage danach, ob allgemeine Abstraktionen auf der Sprachebene reichen, oder ob spezifisch auf den Anwendungsbereich zugeschnittene Abstraktionen eingeführt werden sollen. Da meist umfangreiche Bibliotheken zur Verfügung stehen, hängt die Antwort darauf kaum von der Komplexität ab, sondern davon, wie gut bereits vorhandene allgemeine Abstraktionen das abdecken, was gebraucht wird. Wir versuchen, Abstraktionen so gut es geht auf einer allgemeinen Ebene zu halten, müssen aber auch den Mut haben, auf spezifisch zugeschnittene Abstraktionen auszuweichen, wenn sich Lösungsansätze auf allgemeiner Ebene als zu umständlich erweisen.

Es wird heute als selbstverständlich empfunden, dass Programmiersprachen von Details der Hardware und des Betriebssystems abstrahieren. In einigen Bereichen stehen jedoch keine passenden Abstraktionen zur Verfügung. Besonders fällt das dort auf, wo Eigenschaften bestimmter, wenig standardisierter Hardware, etwa einer GPU, eingesetzt werden sollen. Dafür benötigen wir spezielle Abstraktionen, die sowohl von Hardware-Details als auch Details des Anwendungsbereichs und der eingesetzten Algorithmen abhängen. Wir haben gelernt, vorgefertigte Abstraktionen einzusetzen und diese miteinander zu kombinieren. Wenn vorgefertigte Abstraktionen fehlen, brauchen wir neue, ungewohnte Denkansätze und viel Wissen über die Hardware. Sobald wir eine brauchbare Lösung gefunden haben, kommt die nächste Generation an Hardware, die

wieder neue Lösungsansätze verlangt; trotz aller Mühe ist unsere Lösung nicht ausreichend portabel.

An solchen Beispielen lässt sich erahnen, wie die heute üblichen allgemeinen Abstraktionen entstanden sind. Anfangs mussten Programme immer wieder an sich ständig ändernde Hardware angepasst werden. Über die Jahre änderte sich zweierlei: Einerseits wurde Hardware einheitlicher und Änderungen aus Sicht der Programmierung immer weniger gravierend, auch weil Hardware besser an die Bedürfnisse der Software angepasst wurde. Andererseits wurden durch Herumprobieren Abstraktionen gefunden, die viele Bereiche der Programmierung recht gut abdecken, aber nicht alle (z. B. GPU-Programmierung). Die Existenz mehrerer Programmierparadigmen und vieler -stile nebeneinander verdeutlicht, dass es nicht eine ideale Menge an Abstraktionen gibt, sondern viele unterschiedliche Varianten, die sich evolutionär weiterentwickeln. Die Mengen solcher Abstraktionen sind fragile Gebilde, die nur durch die Notwendigkeit der Zusammenarbeit vieler Menschen eine ausreichende Stabilität finden.

Heute noch innovative Techniken werden, wenn sie allgemein gebraucht werden, zusammen mit spezieller Hardware sehr wahrscheinlich auch bald einen Weg in übliche Hardwareausstattungen und Programmbibliotheken finden und dann nicht mehr als außergewöhnlich gelten. Allerdings werden sich auch immer wieder neue innovative Techniken entwickeln, für die erst mühsam passende Abstraktionen gefunden werden müssen. Das braucht alles viel Zeit.

„Abstraktion“ ist ein vielschichtiger Begriff. Fassen wir zusammen, was Abstraktion auf der Ebene der Programmierung im Feinen bedeuten kann:

- Eine Funktion, Prozedur oder Methode fasst Programmteile zu einer Einheit zusammen, sodass diese Programmteile nicht mehrfach geschrieben werden müssen. Der Name der Einheit wird zur Referenzierung der Einheit verwendet und ist eventuell suggestiv, hat darüber hinaus aber keine Bedeutung. Er ist austauschbar. Jede semantisch wirksame Änderung der Einheit ändert auch das dahinter stehende abstrakte Verständnis. Es handelt sich um eine λ-*Abstraktion* im ureigensten Sinn.

- Wenn wir eine Variable verwenden, die eine Funktion, Prozedur oder Methode enthält, bleibt uns der dahinter stehende Programmcode ohne zusätzliche Information gänzlich verborgen. Im abstrakten Verständnis gehen wir davon aus, dass die Variable für beliebigen, austauschbaren Programmtext steht. Nur die Signatur ist uns gegebenenfalls bekannt. Da die Signatur die Struktur eines möglichen Aufrufs bestimmt, können wir von *struktureller Abstraktion* sprechen.

- Wie bei der λ-Abstraktion fasst eine Funktion, Prozedur oder Methode Programmteile zu einer Einheit zusammen, aber deren Name und Signatur wird zusammen mit Kommentaren zur Beschreibung der Einheit als für Menschen verständliche Spezifikation betrachtet. Jede inhaltliche Änderung des Namens, der Signatur oder eines Kommentars ändert das dahinter stehende abstrakte Verständnis, auch wenn die beschriebenen Programmteile unverändert bleiben. Eine Änderung der Programmteile

lässt das abstrakte Verständnis unberührt, solange die geänderte Semantik inhaltlich noch immer das widerspiegelt, was im Namen, in der Signatur und in den Kommentaren ausgedrückt wird. Aufgrund der Bedeutung des Namens können wir von einer *nominalen Abstraktion* sprechen. Wir haben es mit der gleichen Form nominaler Abstraktion zu tun, wenn eine Variable eine entsprechende Funktion oder Prozedur enthält, da nur Name, Signatur und Beschreibung der Variablen die abstrakte Vorstellung bestimmt, nicht der beschriebene Programmtext.

- In jeder Sprache ist ein Gefüge von Basisbegriffen nötig, die zusammengenommen von der Hardware und dem Betriebssystem abstrahieren. Hinter jedem dieser Begriffe steckt eine *Basisabstraktion*. Auch vorgegebene Kontrollstrukturen und elementare Anweisungen sind Basisabstraktionen. Da das abstrakte Verständnis auf von der Sprache vorgegebenen Namen und syntaktischen Elementen beruht, ähnelt jede Basisabstraktion einer nominalen Abstraktion. Allerdings lässt die Sprachdefinition keine Änderungen von Basisabstraktionen zu – der wesentliche Unterschied zu nominalen Abstraktionen. Name und abstraktes Verständnis sind untrennbar miteinander verbunden. Wenn wir eine Basisabstraktion in einer Variablen ablegen, verlieren wir den Zusammenhang zwischen Name und abstraktem Verständnis, wodurch die Variable nur mehr als entweder strukturelle oder nominale Abstraktion gesehen werden kann.

Wir können zahlreiche weitere Arten von Abstraktionen unterscheiden, etwa zwischen selbst entwickelten und in einer Bibliothek gefundenen vorgefertigten Abstraktionen. Jede Bedeutung von „Abstraktion“ hat in einem bestimmten Kontext seine Berechtigung, keine ist besser oder richtiger als eine andere. Wann immer dieser Begriff verwendet wird, müssen wir darauf achten, welche Bedeutung im Fokus steht. Häufig schwingen mehrere Bedeutungen mit.

1.4.2 Datenabstraktion

Datenabstraktion verlagert die Abstraktion von der Ebene der feinen Programmstrukturen auf die Ebene der Programmierung im Groben. Modularisierungseinheiten werden als Abstraktionseinheiten betrachtet, die häufig Konzepte der realen Welt simulieren und von ihnen abstrahieren. Modularisierungseinheiten bilden auch die Grundbausteine in der Programmorganisation, die wir zu ganzen Programmen zusammenfügen. Datenabstraktion beruht auf der Kombination folgender zwei Konzepte (wie in Abschnitt 1.3.1 angerissen):

Datenkapselung: Eine Modularisierungseinheit fasst eine Menge von Funktionen, Prozeduren oder Methoden und eine Menge von Variablen zu einer untrennbaren Einheit zusammen. Diese Teile hängen stark voneinander ab: Die Bedeutungen der Variablen sind nur diesen Funktionen, Prozeduren oder Methoden bekannt, wären ohne sie also sinnlos. Die Funktionen, Prozeduren oder Methoden verwenden die Daten in den Variablen gemeinsam, würden ohne die Variablen also nicht funktionieren und müssen

bei der Verwendung der Variablen koordiniert vorgehen. Die abstrakte Sichtweise kommt aus dem abstrakten Verständnis jedes einzelnen Teils sowie dem oft von der realen Welt motivierten Zusammenspiel der Teile.

Data-Hiding: Hier geht es um die Trennung der Innenansicht von der Außenansicht einer Modularisierungseinheit. Innerhalb der Modularisierungseinheit sind alle Teile einander bekannt und unbeschränkt verwendbar. Von außerhalb sind nur die von der Modularisierungseinheit exportierten Teile sichtbar, wobei Variablen im Normalfall privat (also von außen nicht sichtbar) bleiben, weil ihre Bedeutungen ja nur innerhalb bekannt sind. Die Betrachtung von außen steht damit notwendigerweise auf einer höheren Abstraktionsstufe als die Betrachtung von innen.

Die nach außen sichtbaren Inhalte bestimmen die Verwendbarkeit der Modularisierungseinheit. Private Inhalte bleiben bei der Verwendung unbekannt und die gesamte Modularisierungseinheit daher auf gewisse Weise abstrakt. Wir sprechen von einer „undurchsichtigen Schachtel“ (black box) oder häufiger „semitransparenten Schachtel“ (grey box), weil manches von außen sichtbar ist. Natürlich stellen auch die Funktionen, Prozeduren und Methoden Abstraktionen dar, meist im Sinn von nominalen Abstraktionen (in allen Paradigmen, vor allem dem objektorientierten), manchmal als λ-Abstraktionen (in der funktionalen Programmierung wenn Funktionen als Spezifikationen gesehen werden – *denotationale Semantik*), selten als strukturelle Abstraktionen (etwa abstrakte Methoden in „funktionalen Interfaces“ in Java). Implizit steht hinter jeder Modularisierungseinheit ein nicht formal festgelegtes, also abstraktes Konzept, das im Idealfall (vor allem in der objektorientierten Programmierung, aber nicht nur dort) eine Analogie in der realen Welt hat, um besser verständlich zu sein. Private und nach außen sichtbare Inhalte zusammen müssen diesem Konzept entsprechen. Solange das Konzept erhalten bleibt, sind private Inhalte problemlos änderbar, ohne dabei die Verwendbarkeit der Modularisierungseinheit zu beeinträchtigen. Üblicherweise sind auch Modularisierungseinheiten selbst, nicht nur ihre Inhalte, durch Kommentare beschrieben, um die Absichten dahinter (also die Abstraktionen) klar darzulegen.

Ein *abstrakter Datentyp* ist im Wesentlichen eine Schnittstelle einer Modularisierungseinheit[4] und entspricht damit genau der abstrakten Sichtweise. Wie in Abschnitt 1.3.3 beschrieben, können wir die Signatur der Modularisierungseinheit als Schnittstelle betrachten. In diesem Fall sprechen wir von einem *strukturellen Typ,* weil er nur von den Namen, Parametertypen und Ergebnistypen

[4]In einer ursprünglichen Definition entspricht ein abstrakter Datentyp einer freien Algebra (siehe Abschnitt 1.1) und besteht damit neben der Signatur auch aus einer Menge an algebraischen Gesetzen (Axiomen), die Elemente der Signatur miteinander in Beziehung setzen. Diese ursprüngliche Definition wird in reinen Formen der funktionalen Programmierung verwendet, wo man auch von *algebraischen Datentypen* spricht. Das passt, weil Modularisierungseinheiten in der rein funktionalen Programmierung keine destruktiv änderbaren Variablen enthalten. Zur Beschreibung von durch Zuweisungen änderbaren Programmzuständen sind Axiome weniger gut geeignet. Aus pragmatischen Gründen wurden die Axiome daher einfach weggelassen (für strukturelle Typen) oder durch abstraktere Formen zur Festlegung der Beziehungen ersetzt (für nominale Typen).

der nach außen sichtbaren Inhalte abhängt – quasi von der nach außen sichtbaren Struktur. Alle Funktionen, Prozeduren und Methoden in einem strukturellen Typ werden als strukturelle Abstraktionen gesehen. Wenn unterschiedliche Modularisierungseinheiten die gleiche Signatur haben, dann haben sie auch den gleichen strukturellen Typ. Genau diese Eigenschaft lässt strukturelle Typen als abstrakte Datentypen in der Praxis wenig sinnvoll erscheinen, weil dies bedeuten würde, dass hinter allen Modularisierungseinheiten mit gleicher Signatur auch die gleiche Abstraktion steckt. Das trifft im Allgemeinen nicht zu. In einigen speziellen Einsatzgebieten sind aber genau solche Abstraktionen sinnvoll; eine Abstraktion entspricht dabei genau der Signatur ohne Annahme zusätzlicher Eigenschaften, es zählt nur das Vorhandensein öffentlich sichtbarer Funktionen, Prozeduren und Methoden, ohne Rücksicht darauf, was sie machen.

Meist gehen wir davon aus, dass neben Signaturen abstrakte Vorstellungen zu den Schnittstellen gehören. Das ergibt die übliche Bedeutung eines abstrakten Datentyps. Für die praktische Verwendung ist es notwendig, dass ein solcher abstrakter Datentyp mit einem eindeutigen Namen bezeichnet wird (etwa `Stack` oder `Queue`), es handelt sich also um einen *nominalen Typ*. Zwei abstrakte Datentypen mit unterschiedlichen Namen werden als verschieden betrachtet, auch wenn sie die gleiche Signatur haben. Erst dadurch können wir diese abstrakten Datentypen mit voneinander verschiedenen Abstraktionen in Verbindung bringen. Der Name steht stellvertretend für alle Absichten und Vorstellungen, die damit verbunden sind, egal ob über Kommentare beschrieben oder aus der Bedeutung von Namen und einer Analogie zur realen Welt intuitiv erfasst. Auf solche Techniken zur Spezifikation aufbauend sind Funktionen, Prozeduren und Methoden in einem nominalen Typ meist nominale Abstraktionen.

In rein funktionalen Sprachen werden bestimmte abstrakte Datentypen als *algebraische Datentypen* spezifiziert, wobei alle Beziehungen zwischen Daten und Funktionen klar festgelegt sind. In diesem Fall werden eindeutige Namen verwendet, wodurch es sich um nominale Typen handelt, aber die Programmstruktur legt wesentlich mehr als nur Signaturen fest. Zusätzliche Beschreibungen sowie Bedeutungen von Namen und Analogien zur realen Welt sind weniger wichtig. Funktionen werden manchmal als λ-Abstraktionen gesehen. Das heißt nicht, dass es in funktionalen Sprachen keine strukturellen Typen gibt. Etwa in ML spielen „Structure“ genannte strukturelle Typen eine bedeutende Rolle.

Abstrakte Datentypen können beliebig genau und auf unterschiedlichste Arten beschrieben sein. In vielen Fällen verwenden wir Zusicherungen entsprechend Design-by-Contract zur Beschreibung, siehe Abschnitt 1.3.3. Das ändert nichts daran, dass wir es mit nominalen Typen zu tun haben. Nominale Typen implizieren, dass es nur genau eine Stelle geben kann, an der ein Typ eingeführt wird; an dieser Stelle wird der Name mit der Modularisierungseinheit in Verbindung gebracht. Das ist auch die Stelle, an der die Zusicherungen zu finden sind. Strukturelle Typen mit Zusicherungen ergeben auch deswegen wenig Sinn, weil einander entsprechende Typen an mehreren Stellen stehen würden und die Zuordnung von Zusicherungen zu Typen im Allgemeinen mehrdeutig wäre.

Theoretisch lässt sich auch mit strukturellen Typen so arbeiten, als ob es sich um nominale Typen handeln würde: Jede Modularisierungseinheit bekommt zu-

sätzliche, eigentlich nicht verwendete Inhalte, deren Namen als Teil der Signatur das abstrakte Konzept dahinter beschreiben (etwa eine Methode namens `iBelongToStack` oder `iBelongToQueue`). Damit werden gleiche Signaturen für unterschiedliche Konzepte ausgeschlossen. Wegen dieser theoretischen Möglichkeit betrachten wir in der Theorie fast nur strukturelle Typen, arbeiten in der Praxis aber dennoch fast ausschließlich mit nominalen Typen.

Heute ist es in allen etablierten Programmierparadigmen üblich, vorgefertigte Funktionen, Prozeduren und Modularisierungseinheiten nur über abstrakte Datentypen bereitzustellen, die man häufig *Bibliotheken* nennt. Die Zuordnung der Inhalte zu Bibliotheken sorgt nicht nur für eine schönere Gliederung, sondern verbessert auch die Verständlichkeit der Abstraktionen. Alleine schon die Zugehörigkeit zu einer bestimmten Bibliothek verrät einiges über dahinter stehende Absichten. Ein der Bibliothek innewohnender logischer Aufbau und dessen Beschreibung trägt noch wesentlich mehr zum Verständnis bei. Auch elementare Typen wie `int` können als abstrakte Datentypen betrachtet werden. Als Basisabstraktionen abstrahieren diese Typen etwa von der Repräsentation in der Hardware. Operationen darauf, etwa die ganzzahlige Addition, werden als mit dem Typ (`int`) in Zusammenhang stehend angesehen, so als ob die Addition in der Bibliothek `int` auffindbar wäre.

1.4.3 Abstraktionshierarchien

Gerade in der objektorientierten Programmierung, aber nicht nur dort, spielen ganze Hierarchien an Abstraktionen eine große Rolle. Ein und dieselbe Modularisierungseinheit kann gleichzeitig auf mehreren Abstraktionsebenen betrachtet werden. Das heißt, mehrere voneinander verschiedene abstrakte Datentypen, die Abstraktionen des gleichen Konzepts mit unterschiedlichem Detailiertheitsgrad darstellen, sind gleichzeitig unterschiedliche Schnittstellen der gleichen Modularisierungseinheit. Ein abstrakter Datentyp kann auch gleichzeitig Schnittstelle mehrerer unterschiedlicher Modularisierungseinheiten sein. Daraus können sich recht komplexe Beziehungsstrukturen ergeben. Diese Strukturen lassen sich zum Teil formal in Programmen abbilden.

Abstraktionen sind häufig vage, nicht genau beschriebene oder beschreibbare Vorstellungen von irgendwelchen Konzepten. Weil Menschen gut darin sind, mit solchen vagen Vorstellungen zu arbeiten, ist das prinzipiell als Vorteil zu sehen; vage Vorstellungen können mit der Zeit konkreter werden und sich in ihrer Bedeutung wandeln, ohne die Vorstellungswelt zusammenbrechen zu lassen. Allerdings führen zu vage Vorstellungsgebilde dazu, dass Abstraktionshierarchien nicht eindeutig sind, unterschiedlich verstanden werden oder im Laufe der Zeit schwinden. Zumindest sind sie einer formalen Analyse kaum zugänglich und daher in Programmen kaum abbildbar. Der einzige Ausweg besteht darin, Beziehungen zwischen Abstraktionen klarer zu definieren, aus vagen Vorstellungen konkretere zu machen. Dabei passiert jedoch etwas Entscheidendes: Es reicht nicht mehr, einfach nur mit Vorstellungen zu arbeiten, sondern wir brauchen präzise Definitionen, die zwar so unbestimmt wie möglich bleiben, aber in jenen Bereichen, die die Struktur betreffen, keinen Interpretationsspielraum

lassen. Einfache Beziehungen zwischen Vorstellungen werden zu komplizierten Gebilden. Mehr noch: Es gibt nicht nur *eine* sinnvolle Art, Präzision in die Vorstellungswelt zu bringen, sondern zahlreiche, die sich teilweise widersprechen und zu unterschiedlichen Strukturen führen. Einige davon werden in der Programmierung tatsächlich eingesetzt, manchmal sogar gleichzeitig. Wir unterscheiden zumindest folgende Arten von Beziehungen zwischen Abstraktionen:

Beziehungen in der realen Welt: Dabei bleiben die Vorstellungen vage, werden aber durch Begriffe aus der realen Welt möglichst klar umrissen. Beziehungen zwischen Abstraktionen ergeben sich einfach daraus, ob die Begriffe auch in der realen Welt auf natürliche Weise in einer entsprechenden Beziehung stehen. Vor allem die „is-a"-Beziehung steht im Mittelpunkt. Da z. B. in der realen Welt ein Auto genau so wie ein Fahrrad ein Fahrzeug ist (is-a), wird ein abstrakter Datentyp zur Darstellung eines Autos als eine speziellere, kompatible Variante eines abstrakten Datentyps zur Darstellung eines Fahrzeugs betrachtet. Entsprechend ist auch ein Fahrrad eine Spezialisierung von Fahrzeug, aber Auto und Fahrrad stehen in keiner solchen Beziehung zueinander. Beziehungen aus der realen Welt werden vor allem in der objektorientierten Modellierung zum Entwurf von Systemen eingesetzt, in der Programmierung selbst aber nur zusammen mit zusätzlichen Einschränkungen.

Untertypbeziehungen: Das sind die in der objektorientierten Programmierung essenziellen Beziehungen, die Ersetzbarkeit ausdrücken. B ist Untertyp von A, wenn jede Instanz von B verwendet werden kann, wo eine Instanz von A erwartet wird, wobei A und B abstrakte Datentypen sind. Häufig werden während der Programmentwicklung Beziehungen in der realen Welt zu Untertypbeziehungen weiterentwickelt. Es kann durchaus sein, dass ein Auto und ein Fahrrad Untertypen eines Fahrzeugs sind, es hängt aber von vielen Details ab, ob das tatsächlich so ist. Neben den Beziehungen in der realen Welt (nötig um den weiterhin vage bleibenden Vorstellungen zu entsprechen) gelten viele weitere Einschränkungen auf den Inhalten der Modularisierungseinheiten, um Ersetzbarkeit zu ermöglichen. Eine der Einschränkungen bezieht sich z. B. darauf, dass ein Parameter einer Methode im Untertyp B nicht spezieller sein darf als der entsprechende Parameter im Obertyp A, um Typsicherheit zu gewährleisten; daraus folgt, dass keine binären Funktionen, also Funktionen, die jeweils mindestens zwei Argumente des gleichen Typs nehmen (etwa um Werte des gleichen Typs miteinander zu vergleichen), über mehrere Ebenen hinweg typsicher darstellbar sind. Wie in Abschnitt 1.3.3 argumentiert, ist Ersetzbarkeit in der Programmierung extrem wichtig, aber nicht ohne bedeutsame Kompromisse erreichbar.

Untertypbeziehungen höherer Ordnung: Dieser Begriff ist zwar aus einer formalen Definition verständlich, aber irreführend, weil entsprechende Beziehungen keine Ersetzbarkeit garantieren und daher keine Untertypbeziehungen sind. Der einzige Unterschied zu Untertypbeziehungen besteht

darin, dass binäre Funktionen unterstützt werden. Solche Beziehungen sind vor allem im Zusammenhang mit einigen Varianten der Generizität von Bedeutung, weil damit Einschränkungen der Werte, die generische Parameter ersetzen, spezifiziert werden können, ohne auf binäre Funktionen verzichten zu müssen. Sie kommen in einigen objektorientierten Sprachen (C++) ebenso zum Einsatz wie in funktionalen Sprachen (Haskell).

Vererbungsbeziehungen: Vererbung ist ein in der objektorientierten Programmierung historisch gewachsener Begriff. Es geht darum, Programmtexte aus einer Oberklasse direkt in eine Unterklasse zu übernehmen, was so erfolgen soll, dass die Unterklasse eine Spezialisierung der Oberklasse ist und Beziehungen aus der realen Welt übernimmt. Damit wird, im Gegensatz zu ursprünglichen Erwartungen, leider keine Ersetzbarkeit garantiert. Obwohl in der objektorientierten Programmierung nach wie vor viel von Vererbung gesprochen wird, ist es nicht mehr zeitgemäß, die Programmstruktur auf Vererbung aufzubauen.

Simulation: Der Begriff Simulation ist überladen. Einerseits verstehen wir darunter das Erstellen eines virtuellen Abbilds eines Konzepts aus der realen Welt in Software, wobei das virtuelle Abbild natürlich nicht die volle Komplexität der realen Welt übernehmen kann, sondern darüber abstrahieren muss. Andererseits wird damit auch eine Beziehung zwischen formalen Systemen, etwa Sprachen bezeichnet, die die relative Ausdrucksstärke vergleicht. Ein System simuliert ein anderes System, wenn das eine System jedes Konstrukt des anderen Systems abbilden kann; das eine System ist in einem gewissen Sinn so mächtig wie das andere. Wenn zwei formale Systeme sich gegenseitig simulieren, sprechen wir von Bisimulation.

1.5 Daten und Datenfluss

Während es mit der strukturierten Programmierung eine breite Akzeptanz für einen relativ einheitlichen Umgang mit dem Kontrollfluss in einem Programm gibt, gehen unterschiedliche Programmierparadigmen mit dem Datenfluss unterschiedlich um. Wir untersuchen zunächst, wie Daten fließen und wie der Datenfluss mit dem Kontrollfluss interagiert. Abhängigkeiten zwischen Daten können als ein mächtiges Instrument genutzt werden, erweisen sich aber auch als gefährlich, wenn sie Programme sehr rasch undurchschaubar kompliziert werden lassen. Wir betrachten mehrere Vorgehensweisen, um damit zurechtzukommen.

1.5.1 Kommunikation über Variablen

Variablen in einem Programm lassen sich in folgende Kategorien einteilen:

Lokale Variablen: Sie werden in einer Funktion, Prozedur oder Methode deklariert und sind nur dort sichtbar und zugreifbar. Bei jedem Aufruf wird neuer Speicherplatz für die lokalen Variablen angelegt, bei der Rückkehr

wird der Speicherplatz wieder freigegeben. Jede Ausführung verwendet daher unterschiedliche Instanzen lokaler Variablen.

Parameter: Sie werden innerhalb einer Funktion, Prozedur oder Methode (im Kopf) deklariert und wie lokale Variablen verwendet. Im Unterschied zu lokalen Variablen stellen Sie eine Verbindung zum Aufrufer her, der beim Aufruf *Argumente* (auch *aktuelle Parameter* genannt) übergibt; Parameter (zur klaren Unterscheidung von aktuellen Parametern auch *formale Parameter* genannt) sind interne Bezeichner für entsprechende Argumente. Java unterstützt nur Eingangsparameter, aber im Allgemeinen werden folgende Parameterarten unterschieden:

Eingangsparameter: Daten fließen auf oberster Ebene nur vom Aufrufer zum Aufgerufenen. Als Argumente werden beliebige Werte übergeben. Falls neue Werte an Parameter zugewiesen werden, gehen diese bei der Rückkehr verloren.

Durchgangsparameter: Daten fließen auch auf oberster Ebene in beide Richtungen. Als Argumente werden initialisierte Variablen übergeben, deren Inhalte über Parameter sichtbar sind. Falls neue Werte an Parameter zugewiesen werden, ändern sich auch die Werte in den Argumenten. Bei einer Realisierung, in der Parameter Referenzen auf Argumente enthalten, sprechen wir von *Referenzparametern*. Durchgangsparameter können aber auch so realisiert sein, dass Argumente beim Aufruf und bei der Rückkehr kopiert werden.

Ausgangsparameter: Daten fließen nur vom Aufgerufenen zum Aufrufer. Als Argumente werden Variablen übergeben. An die entsprechenden Parameter müssen Werte zugewiesen werden, die nach der Rückkehr in den Argumenten stehen.

Variablen in Modularisierungseinheiten: Das sind Variablen, die zu einer Modularisierungseinheit gehören, aber keine lokalen Variablen oder Parameter sind. Je nach Art der Modularisierungseinheit sprechen wir von Objektvariablen (gehören zu Objekten, auch wenn sie in einer Klasse deklariert sind), Klassenvariablen und so weiter. Speicherplatz für solche Variablen wird bei statischen Modularisierungseinheiten statisch (vom Compiler) reserviert, bei Objekten zum Zeitpunkt der Objekterzeugung. Der Speicher bleibt logisch gesehen bis zum Programmende reserviert, kann aber, sobald keine Zugriffe mehr möglich sind, schon früher durch Speicherbereinigung freigegeben werden.

Globale Variablen: Im engeren Sinn sind das Variablen auf globaler Ebene, also solche, die außerhalb von Modularisierungseinheiten deklariert sind und statisch (vom Compiler) verwaltet werden. In einem weiteren Sinn werden auch Variablen, die zu statischen Modularisierungseinheiten gehören, als global bezeichnet, aber nicht Objektvariablen.

Variablen, genauer die über Variablennamen angesprochenen Speicherbereiche, enthalten ihre Werte entweder direkt, oder *Zeiger* (bzw. *Referenzen*) auf

Speicherbereiche, in denen die Werte (oder wieder nur Zeiger) stehen. Wenn wir einen Zeiger *dereferenzieren,* erhalten wir das, worauf gezeigt wird. Das Dereferenzieren erfolgt explizit durch Anwendung eines dafür vorgesehenen Operators oder implizit, also automatisch beim Zugriff auf eine Variable, die eine Referenz enthält. Bei automatischer Dereferenzierung sprechen wir eher von Referenzen, bei expliziter eher von Zeigern, aber ganz einheitlich ist die Terminologie nicht.

Häufig enthalten zwei verschiedene Variablen Zeiger auf dieselben Daten. Die beiden Variablen sind dann *Aliase* voneinander, also im Wesentlichen unterschiedliche Namen für eine Sache; die Inhalte der beiden Variablen sind *identisch.* In einer Programmiersprache mit Zeigern (schließt Referenzen ein), also eigentlich in jeder Sprache, entstehen Aliase unvermeidlich, beispielsweise bei der Parameterübergabe: Wird eine Variable mit einem Zeiger übergeben, sind das Argument und der Parameter Aliase voneinander. Das spart Kopieraufwand, weil statt einer großen Datenmenge nur ein kleiner Zeiger übergeben wird. Nebenbei ergeben sich zusätzliche semantische Möglichkeiten, indem Veränderungen der durch den Parameter referenzierten Daten auch bei Eingangsparametern über das Argument sichtbar werden. Bei der Parameterübergabe fließen Daten auf übersichtliche Weise entlang Linien, die durch den Kontrollfluss vorgegeben sind. Aber auch ganz unterschiedliche Arten von Variablen in ganz unterschiedlichen Programmteilen können Aliase voneinander sein. Die Änderung der referenzierten Daten über eine der beiden Variablen wird sofort über die andere Variable sichtbar.

Über Variableninhalte können Programmteile miteinander kommunizieren:

```
void sort(int[] a) {
    boolean done;
    do { done = true;
         for (int i = 1; i < a.length; i++)
             if (a[i - 1] > a[i]) {
                 swap(a, i - 1, i);
                 done = false;
             }
    } while (!done);
}
```

In `sort` werden über die Variable `done` Informationen über erfolgte Vertauschungen gesammelt und am Ende der `do`-Schleife einmal überprüft. Das heißt, `done` dient als Kommunikationskanal zwischen Ausführungen der `if`-Anweisung und der Abbruchbedingung, wobei `done` eine lokale Variable ist. Im folgenden Beispiel geschieht Ähnliches, aber gut versteckt und nicht lokal:

```
boolean print(int[] a) {
    for (int i : a) System.out.println(i);
    return !System.out.checkError();
}
```

In `System.out` gibt es eine private Variable, die Informationen darüber enthält, ob Ein-/Ausgabefehler aufgetreten sind; `checkError()` prüft auf Fehler. Hier

wird Information unabhängig von der Aufrufstruktur über viele Aufrufe hinweg weitergegeben und nur einmal am Ende abgefragt. In diesen Beispielen sind für die Kommunikation über Variablen keine Aliase nötig.

Aliase erweitern die Möglichkeiten zur Kommunikation zwischen Programmteilen nicht nur, sie können sie auch sehr gut verschleiern. In obigem `print`-Beispiel ist noch erkennbar, dass die Kommunikation über `System.out` als zentrale Stelle läuft. In folgender Abwandlung ist das nicht mehr erkennbar:

```
boolean print(int[] a, PrintStream p, PrintStream q) {
    for (int i : a) p.println(i);
    return !q.checkError();
}
```

Das ist so, obwohl ein Aufruf `print(a, System.out, System.out)` nichts an der Ausführung ändert. Es ist sehr schwer und aufwändig, schon vor der Laufzeit zuverlässige Informationen über Alias-Beziehungen zu erhalten, also im Beispiel festzustellen, ob `p` und `q` immer oder niemals identisch sind. Zur Laufzeit können wir solche Information durch `if(p==q)...` leicht erhalten.

Das Hauptproblem an der Kommunikation zwischen Variablen besteht darin, dass Kommunikationsstrukturen die über die Kontrollstrukturen aufgebaute Programmstruktur unterlaufen, damit auch die bewusst gemachten Einschränkungen durch die strukturierte Programmierung. Die Verschleierung durch Aliase kann auf mehrerlei Weise gesehen werden: Aus einer Sicht verstärkt sie die Problematik und ist daher bestmöglich einzuschränken (eine herkömmliche, eher intuitive Vorgehensweise). Aus einer anderen Sicht ist sie nicht die Ursache, sondern nur ein Verstärkungsfaktor und die Ursache sollte gelöst werden (referentielle Transparenz in der funktionalen Programmierung). Aus einer dritten Sicht sind Programme so zu gestalten, dass stets maximale Verschleierung angenommen wird, also darauf verzichtet wird, den genauen Ablauf zu verstehen und bei Bedarf zur Laufzeit Abfragen und Programmverzweigungen einzusetzen (eine professionelle Vorgehensweise in der objektorientierten Programmierung). Jede dieser Sichtweisen ist gerechtfertigt.

Die Verwendung von Programmteilen als Daten erschwert das Verständnis von Programmen zusätzlich: Werden Funktionen oder Modularisierungseinheiten als Parameter übergeben (das ist eine Kurzform für „als Argumente an Parameter übergeben"), dann werden implizit auch Kontrollstrukturen übergeben, die über den Parameter beliebig benutzbar sind. Wie wir schon gesehen haben, ist das ein sinnvolles und sehr mächtiges Instrument. Ein Nachteil besteht jedoch darin, dass der zur Ausführung gebrachte (dynamische) Kontrollfluss nicht mehr vollständig mit dem im Programmtext direkt ersichtlichen (statischen) Kontrollfluss übereinstimmt. Um den Programmablauf auf einer niedrigen Ebene im Detail zu verstehen, müssten wir den Datenfluss, also die Weitergabe der Kontrollstrukturen als Daten in allen Einzelheiten nachvollziehen. Da alle Arten von Variablen indirekt auch Kontrollstrukturen enthalten können, ist das äußerst schwierig oder sogar unmöglich. Die übliche Vorgehensweise besteht darin, gar nicht zu versuchen, den Programmablauf auf dieser Detailebene zu verstehen. Die Lösung liegt in der Abstraktion: Wir verknüpfen

eine abstrakte Vorstellung mit einer Variablen, ohne den Inhalt der Variablen zu betrachten. Wir versuchen, das Programm nur anhand der abstrakten Vorstellung zu verstehen, sodass jeder mögliche bzw. erwartete oder erlaubte Variableninhalt zum gleichen Programmverständnis führt. Diese Vorgehensweise hat sich in mehreren unterschiedlichen Programmierparadigmen (vor allem in der objektorientierten und funktionalen Programmierung) als sehr erfolgreich erwiesen. Allerdings handelt es sich um eine komplexe, professionelle Vorgehensweise, die viel Programmiererfahrung und -disziplin voraussetzt. Das bewusste Zulassen eines gewissen Kontrollverlusts, bzw. das gezielte Ersetzen der Kontrolle durch intuitive Vorstellungen, war vielleicht einer der wichtigsten Paradigmenwechsel in der Geschichte der Programmierung, der die Entwicklung einiger hochkomplexer heutiger Systeme erst ermöglicht hat. Ein bedeutender Teil dieses Skriptums wird sich mit Techniken beschäftigen, die zeigen, wie wir mit abstrakten Vorstellungen umzugehen haben, um flexibel genug zu bleiben und trotzdem die Kontrolle nicht ganz zu verlieren.

Programmteile als Daten in Variablen, die auf irgendeine Weise zur Ausführung gebracht werden können, sind ein potenzielles Sicherheitsrisiko. Wenn wir Programmtexte ausführen, die wir nicht kennen oder kontrollieren können, könnte Unerwartetes oder Gefährliches passieren. Mehrere typische Techniken zum Eindringen in geschützte Systeme beruhen darauf. Wir müssen also besondere Sorgfalt walten lassen. Aliase verstärken diese Gefahr. Über ein Alias in einem weniger gut geschützten Programmbereich könnte gefährlicher Code untergejubelt werden, der durch die Kommunikation über Variablen in einem geschützten Bereich automatisch sichtbar wird und zur Ausführung kommt.

Ein vielfältiger Einsatz von Programmteilen als Daten lässt die Bedeutung von Kontrollstrukturen schwinden. Das bringt uns zur Frage, ob es überhaupt noch zeitgemäß ist, den Programmablauf auf Kontrollstrukturen zu stützen, oder ob wir den Programmablauf ganz auf den Fluss von Daten aufbauen können. *Lazy-Evaluation* zeigt uns, dass Letzteres nicht nur machbar, sondern in einigen Bereichen sogar sehr sinnvoll ist. Herkömmliche Programmabläufe, manchmal als *Eager-Evaluation* bezeichnet, wenden Operationen wie im Kontrollfluss beschrieben sofort (also so früh wie möglich) auf Operanden an, z. B. wird `2+5` sofort zu `7` evaluiert. Bei Lazy-Evaluation erfolgen die eigentlichen Berechnungen dagegen so spät wie möglich, `2+5` bleibt unverändert, solange das Ergebnis nicht verwendet wird; erst wenn die Auswertung unumgänglich ist, wird sie durchgeführt, etwa in `System.out.print(2+5)` unmittelbar vor der Ausgabe. Das heißt, die Programmabarbeitung entsprechend des Kontrollflusses baut nur ein Netz von Abhängigkeiten zwischen Operationen, Werten und Variablen auf, ohne die Operationen auszuführen. Erst wenn eine abschließende Operation das Ergebnis benötigt, wird das Netz der Abhängigkeiten so weit reduziert, bis das Ergebnis vorliegt, jedoch nicht weiter. Der Begriff *Reduktion* verdeutlicht, dass es um Reduktionen wie im λ-Kalkül geht: Für Lazy-Evaluation wählen wir in jedem Reduktionsschritt einen am weitesten außen liegenden reduzierbaren λ-Ausdruck, für Eager-Evaluation, einen am weitesten innen liegenden. Auf den ersten Blick mag die Vorgehensweise bei Lazy-Evaluation als umständlich erscheinen, die eigentlichen Berechnungen werden ja nur hinaus-

gezögert. Aber genau das ist einer der Vorteile: Berechnungen werden vielleicht so lange hinausgezögert, bis ihre Ergebnisse (und damit auch die Berechnungen) gar nicht mehr nötig sind. In der Praxis zeigt sich, dass viele berechnete Teilergebnisse nie verwendet werden und sich der Zusatzaufwand daher möglicherweise lohnt. Wichtiger ist, dass Lazy-Evaluation neue Programmierstile ermöglicht. Beispielsweise ist es ganz einfach, mit potenziell unendlich langen Listen zu arbeiten, weil ohnehin nur die gebrauchten Listenelemente tatsächlich berechnet werden, der Rest wird nicht berechnet. Einige Programmiersprachen wie Haskell beruhen ganz auf Lazy-Evaluation, viele anderen Sprachen, einschließlich Java, verwenden Lazy-Evaluation nur in bestimmten Bereichen.

Eine Schwierigkeit besteht in der unterschiedlich langen Lebenszeit von Variablen. Würden wir einen Zeiger in einer globalen Variable auf den Inhalt einer lokalen Variable zeigen lassen, würde dieser Zeiger ins Leere gehen, sobald die lokale Variable nicht mehr existiert. Einige Programmiersprachen überlassen es ganz den Programmierer_innen, mit dieser Problematik umzugehen (z. B. C und C++). Aktuellere Sprachen sorgen in der Regel dafür, dass nur von kurzlebigen Variablen auf längerlebige gezeigt werden kann, nicht umgekehrt. Das hat jedoch einen Preis: Viel mehr Daten als unbedingt nötig werden in potenziell langlebigen Variablen in Modularisierungseinheiten, insbesondere Objekten abgelegt, nicht in lokalen Variablen, wodurch ein verstärkter Bedarf an Speicherbereinigung entsteht.

1.5.2 Wiedererlangung der Kontrolle

Kommen wir zurück auf den Kern des Problems: Der Datenfluss unterläuft Programmstrukturen, die durch den Kontrollfluss vorgegeben sind. Aliase und die Verwendung von Kontrollstrukturen als Daten verstärken den Effekt, sodass es insgesamt sehr schwierig wird, den Überblick zu behalten; die Kontrolle geht verloren. Wir haben zwar schon einige Lösungsansätze angedeutet, müssen diese aber noch genauer betrachten und vor allem mit konkreten Zielsetzungen in Verbindung bringen. Das gelingt am besten, indem wir die entsprechenden Lösungsansätze in unterschiedlichen Paradigmen einander gegenüberstellen.

Prozedurale Programmierung. Das Ziel besteht darin, die Programmstruktur möglichst klar durch den Kontrollfluss abzubilden. Es wird versucht, unvermeidliche Störfaktoren klein zu halten. Auch wenn die Ursprünge in die Zeit von „Goto“ zurückgehen, steht die strukturierte Programmierung auf der Basis von Kontrollstrukturen, die ihre Wirkung hauptsächlich über Seiteneffekte erzielen, zentral im Mittelpunkt. Globale Variablen sind aus Effizienzgründen zugelassen und spielen praktisch gesehen eine wichtige Rolle, gleichzeitig wird vor ausufernden Verwendungen globaler Variablen gewarnt, weil ihr störerischer Einfluss auf die Programmstruktur bekannt ist. Auch vor gefährlichen Verwendungen von Aliasen wird gewarnt, obwohl ihr Auftreten kaum zu verhindern ist. Modularisierungseinheiten sind, falls vorhanden, bewusst stark eingeschränkt, jedenfalls sind sie nicht als Daten verwendbar. Auch Prozeduren werden meist nicht als Daten angesehen, jedenfalls spielen sie als Daten keine große Rolle.

Manchmal können Prozeduren als Eingangsparameter an Prozeduren übergeben werden, aber in die umgekehrte Richtung geht das eher nicht, was auch durch die beschränkte Lebenszeit lokaler Variablen begründet ist.

Funktionale Programmierung. Anders als in der prozeduralen Programmierung stehen Funktionen, die als Daten verwendbar sind, zentral im Mittelpunkt. Von Anfang an war klar, dass dies einen auf Seiteneffekten beruhenden Kontrollfluss untergräbt, weshalb auf entsprechende Kontrollstrukturen verzichtet wurde. Das Ziel ist, stattdessen Kontrollstrukturen aus der Verwendung von Funktionen als Daten aufzubauen. Strukturierte Programmierung ist nicht von der Sprache vorgegeben, aber viele Programme halten sich dennoch an die entsprechenden Prinzipien. Ursprünglich waren Seiteneffekte erlaubt (etwa in Lisp und Scheme), sie galten aber bald als unerwünscht, weil sich Aliase zu bedeutenden Störfaktoren entwickelten. Inzwischen hat sich *referentielle Transparenz* als Ziel neuerer funktionaler Sprachen durchgesetzt, was bedeutet, dass nicht zwischen dem Original und einer Kopie unterschieden wird, das heißt, kein Unterschied zwischen gleichen und identischen Dingen gemacht wird. Das wird durch die gänzliche Vermeidung von Seiteneffekten erreicht (außer für die Ein- und Ausgabe, aber auch dafür gibt es trickreiche Lösungen). Damit verliert der Umgang mit Aliasen den Schrecken und Modularisierungseinheiten werden problemlos einsetzbar. Der offensichtliche Nachteil, dass mangels Seiteneffekten keine Kommunikation über Variablen außerhalb der durch Funktionsaufrufe gegebenen Programmstruktur möglich ist, wird als Vorteil gesehen. Paradoxerweise hat sich durch Verzicht auf vorgegebene Kontrollstrukturen und Seiteneffekte etwas ergeben, was die ursprünglichen Ziele hinter Kontrollstrukturen sehr gut erreicht. Beim praktischen Programmieren müssen wir uns jedoch daran gewöhnen, dass die Einschränkungen restriktiv und nicht umgehbar sind, sowie daran, dass neben λ-Abstraktion und nominaler Abstraktion auch strukturelle Abstraktion eingesetzt wird (also viele Variablen für – abgesehen von Typinformation – unbekannte ausführbare Funktionen stehen).

Objektorientierte Programmierung. Die objektorientierte Programmierung folgt der prozeduralen Herangehensweise, stellt aber Objekte als Daten in den Mittelpunkt. Zunächst ergeben sich alle angesprochenen Schwierigkeiten, die durch ein Bündel an Maßnahmen in den Griff gebracht werden:

- Jedes Objekt (allgemeiner jede Modularisierungseinheit) wird als nominaler abstrakter Datentyp betrachtet und ist damit auf einer abstrakten Ebene, ohne Kenntnis von Implementierungsdetails verständlich.

- Variablen enthalten ausschließlich (Referenzen auf) Objekte, wobei aus pragmatischen Gründen manche Objekte, etwa Zahlen, speziell behandelt werden. Damit ist jeder Variableninhalt als Instanz eines nominalen abstrakten Datentyps verwendbar.

- Auf Variablen in Modularisierungseinheiten wird von außen nicht zugegriffen, innerhalb schon. Das ermöglicht die örtlich eingegrenzte einfache

Kommunikation über Variablen, aber auch die von konkreten Daten unabhängige abstrakte Sichtweise von außen (als abstrakter Datentyp). Aliase ermöglichen dennoch die uneingeschränkte Kommunikation über Variablen, ohne Rücksicht auf Grenzen von Modularisierungseinheiten.

- Einzelne Methoden sind entsprechend der strukturierten Programmierung über prozedurale Kontrollstrukturen aufgebaut. Neben lokalen Variablen und Parametern wird auch großzügig auf Variablen der eigenen Modularisierungseinheit zugegriffen; das gilt nicht als verpönt.

- Methoden aus anderen Objekten werden über dynamisches Binden aufgerufen, wodurch nicht von vorne herein bekannt ist, welche Methoden zur Ausführung kommen. Damit ist ein statisches Verständnis der genauen Methodenausführungen auf Detailebene ausgeschlossen. Es wird ein Verständnis auf einer abstrakten Ebene erzwungen.

- Mit Aliasen wird offensiv umgegangen, sie gelten nicht als Problem. Es wird streng zwischen dem Original und seiner Kopie unterschieden, auch Gleichheit und Identität werden klar voneinander unterschieden.

Zusammengefasst: Innerhalb einer Modularisierungseinheit gibt es große Freiheit mit allen damit verbundenen Gefahren, über Grenzen von Modularisierungseinheiten hinweg besteht die größtmögliche Abgrenzung. Aliase bleiben auf globaler Ebene deutlich sichtbar. Ihre Auswirkungen werden dadurch reduziert, dass alle Variableninhalte als abstrakte Objekte mit klar definierter Identität betrachtet werden. Beim praktischen Programmieren steht der Umgang mit nominalen Abstraktionen ganz zentral im Mittelpunkt.

Häufig wird von paradigmenübergreifenden Sprachen und Systemen gesprochen. Dahinter steckt das Ziel, die Grenzen zwischen den Paradigmen zu überwinden. Wenn wir allerdings sehen, wie starke Unterschiede zwischen den Paradigmen sich schon bei vergleichsweise elementaren Aspekten ergeben, können wir erahnen, wie schwierig die Überwindung der Grenzen sein muss.

1.5.3 Verteilung der Daten

Unterschiedliche Herangehensweisen bewirken unterschiedliche Datenstrukturen und Verteilungen der Daten im Speicher. Nehmen wir als Beispiel die Daten zu einer Lehrveranstaltung, die Studierenden Beurteilungen einzelner Lehrveranstaltungsteile zuordnet. Die prozedurale Programmierung versucht Aliase zu vermeiden und führt daher häufig, aber nicht immer zu Strukturen, wo jedes Datenelement auf nur eine einheitliche Weise auffindbar ist. Das Beispiel könnte etwa über assoziative Datenstrukturen gelöst sein, je eine für Studierendendaten (Namen, Mailadressen, Studienkennzahlen, etc.) und jeden einzeln beurteilten Lehrveranstaltungsteil, die jeweils Matrikelnummern als Schlüssel verwenden.

In der funktionalen Programmierung ist es schwierig, einzelne Daten in großen Strukturen zu ändern, weil dabei die ganzen Strukturen neu aufgebaut werden

müssen. Das führt häufig zu einer Strukturierung, in der stabile (sich kaum ändernde) Daten von solchen getrennt werden, die ständigen Änderungen unterliegen. Gleichzeitig sollen zu viele voneinander getrennte Datenstrukturen vermieden werden, weil Funktionen dafür jeweils einen eigenen Parameter benötigen. Beispielsweise könnte es eine stabile Datenstruktur mit Studierendendaten (wie im prozeduralen Fall) und eine davon getrennte, sich ständig erweiternde Datenstruktur mit Beurteilungen geben, aber keine Trennung zwischen Lehrveranstaltungsteilen. Häufig geänderte Beurteilungsdaten können auf stabile Studierendendaten verweisen, umgekehrt ergäben sich Effizienzprobleme.

Die objektorientierte Programmierung fasst häufig ganz unterschiedliche Daten zu einer Einheit zusammen, um Variablen zur Kommunikation zu nutzen. Im Beispiel würden alle Daten zu einem Studierenden in einem einzelnen Objekt gekapselt, einschließlich der Beurteilungsdaten.

Während es in der prozeduralen, funktionalen und objektorientierten Programmierung vorwiegend um den Überblick über den Programmablauf geht und sich die Verteilungen der Daten eher zufällig ergeben, gehen einige andere Paradigmen von bestimmten vorgegebenen Datenverteilungen aus.

Verteilte Programmierung. Dabei wird von einer Verteilung der Daten über mehrere oder viele Rechner ausgegangen, wobei gleiche Daten redundant auf mehreren Rechnern vorhanden sein können. Gründe für die Verteilung sind vielfältig. Häufig ist die Datenmenge und die Anzahl der Zugriffe zu groß, um sie auf einem Rechner handhaben zu können, manchmal ist eine Datenauslagerung zwecks Datenschutz oder aus wirtschaftlichen oder rechtlichen Gründen erforderlich. In jedem Fall müssen der Ort der Daten und der Ort der Berechnungen zusammengebracht werden. Das lässt sich bewerkstelligen, indem Rohdaten, oder wenn Berechnungen am Ort der Daten erfolgen, erst Ergebnisse weitergegeben werden. Es ergibt sich eine gewaltige Zahl an Möglichkeiten und Optimierungen zur Verringerung des Datenverkehrs, wenn Berechnungen nicht nur lokal, sondern über ein Netzwerk verteilt stattfinden. Das Ziel der verteilten Programmierung besteht im Finden einer guten Strategie für die Verteilung der Daten und Berechnungen. In diesem Kontext werden Überlegungen hinter der prozeduralen, funktionalen und objektorientierten Programmierung als engstirnig betrachtet, weil die Problematik auf anderen Gebieten liegt; ein Verständnis des Kontrollflusses ist vergleichsweise einfach. Dennoch gibt es Querverbindungen, weil etwa eine Strukturierung von Daten wie in der objektorientierten Programmierung den Austausch von (im Beispiel) Daten zu einzelnen Studierenden unterstützt, während eine Strukturierung wie in der funktionalen Programmierung dabei hilft, mehrere Kopien der Daten auf einfache Weise zu verwalten.

Parallele Programmierung. Das Ziel ist die Erreichung kurzer Laufzeiten. Meist ist eine große Menge eher homogener (gleichartiger) Daten in kurzer Zeit zu verarbeiten, wobei häufig auch spezielle Hardware zum Einsatz kommt. Daten müssen so strukturiert sein, wie es der Hardwareeinsatz verlangt. Meist ist es notwendig, die Daten in mehrere Bereiche zu untergliedern, die unabhän-

gig voneinander auf unterschiedlichen Recheneinheiten verarbeitbar sind. Eine Datenstrukturierung wie in der objektorientierten Programmierung ist dabei hinderlich. Im Beispiel sollen etwa Daten über Beurteilungen unterschiedlicher Lehrveranstaltungsteile unabhängig voneinander verarbeitbar sein, was aber nicht geht, wenn Studierendendaten zu Objekten zusammengefügt sind. Umgekehrt entspricht eine für die parallele Programmierung optimierte Datenstruktur nur selten auch den Prinzipien der objektorientierten Programmierung. Datenstrukturierungen wie in der funktionalen Programmierung sind ebenso nicht von vorne herein ideal, aber eine im Hinblick auf die Parallelverarbeitung optimierte Datenstruktur lässt sich meist auch ganz gut mit der funktionalen Programmierung kombinieren. Oft setzt die parallele Programmierung nach wie vor auf der prozeduralen Programmierung auf, weil einige der Zielsetzungen übereinstimmen, etwa die bestmögliche Vermeidung von Aliasen.

Nebenläufige Programmierung. Es soll auf Ereignisse reagiert werden, die zu unvorhersehbaren Zeitpunkten auftreten. Meist bestehen gleichzeitig viele, logisch in sich geschlossene Handlungsstränge, die jedoch nicht in einem Stück ausführbar sind, weil auf das Eintreffen angeforderter Daten gewartet werden muss. Nehmen wir als Beispiel einen Web-Server, der gleichzeitig viele offene Verbindungen bearbeitet, aber für jede Verbindung den Großteil der Zeit mit dem Warten auf das Eintreffen ausgefüllter Formulare verbringt. Die Verbindungen sind logisch voneinander getrennt, z. B. wegen unterschiedlicher Zugangsdaten und Zugriffsrechte. Am einfachsten lässt sich das durch *nebenläufige Programmierung* handhaben, indem jeder Handlungsstrang etwa als Java-*Thread* (Sprachkonstrukt für Handlungsstränge) abgebildet wird. Der englische Begriff *concurrent programming* drückt nicht so klar aus, worum es geht. Es können ähnliche Techniken wie in der Parallelprogrammierung eingesetzt werden, aber die Zielsetzung ist eine andere, weil eine kurze Laufzeit nur eine untergeordnete Rolle spielt; die meiste Zeit wird ja mit Warten verbracht. Wichtiger ist es, viele Handlungsstränge handhaben zu können. Gleichzeitige Zugriffe auf gleiche Daten durch unterschiedliche Handlungsstränge sind nur mittels Synchronisation verhinderbar. Die Daten müssen so strukturiert sein, dass unterschiedliche Handlungsstränge möglichst auf unterschiedliche Daten zugreifen. Die ideale Strukturierung hängt von Details ab. Manchmal ist eine Strukturierung wie in der objektorientierten Programmierung gut geeignet, in vielen Fällen nicht.

Echtzeitprogrammierung. Dabei geht es darum, dass auf Ereignisse von außen innerhalb einer vorgegebenen, meist kurzen Maximalzeit reagiert werden muss. Häufig sind Garantien für die Einhaltung der Maximalzeit nötig. Diese Forderung macht die Echtzeitprogrammierung sehr speziell und unterscheidet sie von allen anderen Programmierparadigmen und -stilen. Daten müssen so vorliegen, dass Zugriffe darauf in einer vorgegebenen Maximalzeit abgeschlossen sind. Auf Zufall beruhende Datenstrukturen wie etwa Hashtabellen sind dafür kaum geeignet, ebenso wenig wie Datenstrukturierungen, die üblicherweise in der funktionalen und objektorientierten Programmierung entstehen.

Metaprogrammierung. Dabei werden Programme selbst als Daten angesehen. Je nach Ausformung werden Programme zur Laufzeit erstellt, verändert oder nur gelesen und Programmtexte so ausgeführt, wie es im ursprünglichen Programm nicht vorgesehen war. Alle Bestandteile eines Programms müssen in Form von Daten vorliegen. Metaprogrammierung ist mit allen Paradigmen kombinierbar (außer vielleicht Echtzeitprogrammierung), kann dabei aber einen beträchtlichen Einfluss auf die Strukturierung der Daten haben.

1.6 Typisierung

Die Bedeutung von Typen in Programmiersprachen nimmt stetig zu, obwohl Typen Einschränkungen darstellen: Wir können nicht beliebige Werte, Ausdrücke, etc. verwenden, sondern nur solche, die vorgegebenen Typen entsprechen. Als Gegenleistung erhalten wir bessere Planbarkeit – weil wir früher wissen, womit wir es zu tun haben – und bestimmte Formen der Abstraktion. Beides verbessert die Lesbarkeit und Zuverlässigkeit von Programmen.

1.6.1 Typkonsistenz, Verständlichkeit und Planbarkeit

Berechnungsmodelle kennen nur je eine Art von Werten, z. B. nur Zahlen, Symbole oder Terme. In Programmiersprachen unterscheiden wir mehrere Arten von Werten, z. B. ganze Zahlen, Gleitkommazahlen und Zeichenketten. Typen helfen bei der Klassifizierung der Werte. Viele Operationen sind nur für Werte bestimmter Typen definiert. Beispielsweise sind nur ganze Zahlen oder Gleitkommazahlen miteinander multiplizierbar, aber keine Zeichenketten. Wenn die Typen der Operanden mit der Operation zusammenpassen, sind die Typen *konsistent.* Andernfalls tritt ein *Typfehler* auf. Ohne Typen kommen wir nur aus, wenn alle Operationen auf alle Operanden anwendbar sind, was praktisch nie der Fall ist. Daher gibt es in jeder Programmiersprache Typen.

Typprüfungen. Es gibt erhebliche Unterschiede zwischen Sprachen hinsichtlich des Umgangs mit Typen. Nur wenige hardwarenahe Sprachen prüfen Typkonsistenz gar nicht, etwa Assembler oder Forth; bei einem Typfehler werden Bit-Muster falsch interpretiert. Einige Sprachen prüfen Typkonsistenz in vielen, aber nicht allen Fällen; etwa bleiben in C Arraygrenzen und Typumwandlungen unüberprüft. Die meisten Sprachen prüfen Typkonsistenz vollständiger.

Sprachen unterscheiden sich hinsichtlich des Zeitpunkts, an dem Typen bekannt sind und überprüft werden. In dynamisch typisierten Sprachen ergeben sich Typen erst zur Laufzeit und werden bei der Anwendung einer Operation dynamisch überprüft. In statisch typisierten Sprachen sind genaue Typen aller Ausdrücke bereits dem Compiler bekannt, der sie für statische Prüfungen nutzt. Bei statischer Prüfung kann zur Laufzeit kein Typfehler auftreten. Bei dynamischer Prüfung müssen wir immer mit Typfehlern rechnen. Allerdings ist die statisch verfügbare Information begrenzt. Das heißt, für manche Aspekte (etwa Arraygrenzen) ist dynamische Prüfung nötig, oder die Flexibilität ist

so eingeschränkt, dass statische Prüfung ausreicht. Dank ausgefeilter statischer Typanalysen gibt es heute hinreichend flexible Sprachen ausschließlich auf Basis statischer Typprüfungen, z. B. Haskell. Viele Sprachen bevorzugen dennoch einen Mix aus dynamischer und statischer Prüfung. Zwischen dynamisch und statisch typisierten Sprachen liegen *stark typisierte* Sprachen, in denen der Compiler statisch Typsicherheit garantiert, obwohl nicht alle Typinformation zur Compilezeit vorliegt.[5] Objektorientierte Sprachen wie Java sind häufig, bis auf Aspekte wie Arraygrenzen, stark typisiert; deklarierte Typen reichen für statische Typprüfungen, obwohl viel Typinformation durch dynamisches Binden erst zur Laufzeit bekannt wird.

Der Hauptgrund für statische Typprüfungen scheint die verbesserte Zuverlässigkeit der Programme zu sein. Das stimmt zum Teil, aber nicht auf direkte Weise. Typkonsistenz bedeutet ja nicht Fehlerfreiheit, sondern nur die Abwesenheit bestimmter, eher leicht auffindbarer Fehler. Manchmal hört man die Meinung, dynamisch typisierte Sprachen seien sogar sicherer, weil Programme besser getestet und dabei neben Typfehlern auch andere Fehler entdeckt würden. Es bleibt offen, ob dahinter ein wahrer Kern steckt. Viel wichtiger ist die Tatsache, dass explizit hingeschriebene Typen, die es in dynamisch typisierten Sprachen meist nicht gibt, das *Lesen und Verstehen* der Programme erleichtern. Die statische Prüfung sorgt dafür, dass die Information in den Typen zuverlässig ist – die einzig mögliche Form zuverlässiger Kommentare. Besseres Verstehen erhöht die Zuverlässigkeit der Programme, die Typprüfungen selbst aber kaum.

Auch in statisch typisierten Sprachen müssen nicht alle Typen hingeschrieben sein. Viele Typen kann ein Compiler aus der Programmstruktur herleiten; wir sprechen von *Typinferenz*. Beispielsweise müssen wir in Haskell keinen Typ hinschreiben, obwohl der Compiler statisch Typkonsistenz garantiert. Zur besseren Lesbarkeit sollen wir Typen dennoch hinschreiben. Bis zu einem gewissen Grad erhöht Typinferenz die Verständlichkeit, obwohl keine Typen dabei stehen. Das hat mit reduzierter Flexibilität zu tun: Es sind keine so komplexen Programmstrukturen verwendbar, dass der Compiler bei der Prüfung der Typkonsistenz überfordert wäre. Einfachere Programmstrukturen sind nicht nur für Compiler, sondern auch für Menschen einfacher verständlich.

Typen und Entscheidungsprozesse. Bedeutende Entscheidungen hinsichtlich Programmstruktur und Programmablauf müssen schon früh getroffen werden. Hier ist eine Einteilung von Entscheidungszeitpunkten:

- Einiges ist bereits in der *Sprachdefinition* oder in der *Sprachimplementierung* festgelegt. Beispielsweise ist festgelegt, dass Werte von `int` ganze Zahlen in einer 32-Bit-Zweierkomplementdarstellung sind und vordefinierte Operationen darauf nicht auf Über- bzw. Unterläufe prüfen. Programme können daran nichts ändern.

[5] Der Begriff *stark typisiert* ist schon lange in der hier genannten Bedeutung etabliert. Allerdings haben Programmiersprachentwickler mehrfach versucht, ihn für sich zu reklamieren und umzudefinieren. Daher hat er heute in einigen Kreisen eine etwas andere Bedeutung.

- Zum Zeitpunkt der *Erstellung von Übersetzungseinheiten* werden die meisten wichtigen Entscheidungen getroffen, auf die wir beim Programmieren Einfluss haben. Hierzu braucht es viel Flexibilität.

- Manche wichtige Entscheidungen werden durch Parametrisierung erst bei der *Einbindung* vorhandener Module, Klassen oder Komponenten getroffen. Dies geht jedoch nur, wenn die eingebundenen Modularisierungseinheiten dafür vorgesehen sind.

- Vom *Compiler* getroffene Entscheidungen sind eher von untergeordneter Bedeutung und betreffen meist nur Optimierungen. Alles Wichtige ist bereits im Programmcode festgelegt oder liegt erst zur Laufzeit vor.

- Zur Laufzeit ist die *Initialisierungsphase* von der *eigentlichen Programmausführung* zu unterscheiden. Erstere dient auch der Einbindung von Modularisierungseinheiten und der Parametrisierung, wo dies erst zur Laufzeit möglich ist. Zur Laufzeit getroffene Entscheidungen folgen einem fixen, im Programm festgelegten Schema.

Typen verknüpfen die zu unterschiedlichen Zeitpunkten vorliegenden Informationen miteinander. Vor allem statisch geprüfte Typen helfen dabei, einmal getroffene Entscheidungen über den gesamten folgenden Zeitraum konsistent zu halten. Einige Beispiele in Java sollen dies verdeutlichen:

- Angenommen, bei der Erstellung einer Klasse bekommt eine Variable den Typ `int`. Der Compiler reserviert so viel Speicherplatz, wie in der Sprachdefinition für `int` (32 Bit) vorgesehen ist. Zur Laufzeit wird ebenso wie bei der Programmerstellung fix davon ausgegangen, dass die Variable eine Zahl im entsprechenden Wertebereich enthält.

- Ist der Typ der Variablen ein Typparameter, reserviert der Compiler den für eine Referenz nötigen Speicherplatz. Bei der Programmerstellung und zur Laufzeit wird innerhalb der Klasse, in der die Variable deklariert ist, von einer Referenz unbekannten Typs ausgegangen. Wird der Typparameter durch `Integer` ersetzt (`int` darf in Java keine Typparameter ersetzen), kann davon ausgegangen werden, dass die Variable ein Objekt dieses Typs referenziert. Vor dem Ablegen einer Zahl vom Typ `int` in der Variablen wird ein `Integer`-Objekt erzeugt und beim Auslesen wieder die `int`-Zahl extrahiert. Code dafür erzeugt der Compiler automatisch.

- Hat die Variable einen allgemeinen Typ wie `Object`, reserviert der Compiler Platz für eine Referenz. Zur Laufzeit muss bei Verwendung der Variablen in der Regel eine Fallunterscheidung getroffen werden, da unterschiedliche Arten von Werten unterschiedliche Vorgehensweisen verlangen. Diese Fallunterscheidungen stehen im Programmcode innerhalb einer Klasse oder verteilt auf mehrere Klassen.

Je früher Entscheidungen getroffen werden, desto weniger ist zur Laufzeit zu tun und desto weniger Programmtext für Fallunterscheidungen ist nötig. Ohne

statisch geprüfte Typen wäre mehrfacher Aufwand nötig: Sogar wenn wir wissen, dass die Variable eine ganze Zahl enthält, muss der Compiler eine beliebige Referenz annehmen und Code für eine dynamische Typprüfung erzeugen.

Frühe Entscheidungen haben einen weiteren Vorteil: Typfehler und andere damit zusammenhängende Fehler zeigen sich früher, wo ihre Auswirkungen noch nicht so schwerwiegend sind. Vom Compiler entdeckte Fehler sind meist einfach zu beseitigen. Auf Fehler, die zur Laufzeit in einer Initialisierungsphase erkannt werden, kann effektiver reagiert werden als auf Fehler, die während der eigentlichen Programmausführung auftreten. Auch in dynamisch typisierten Sprachen können wir Typfehler schon in der Initialisierungsphase zu erkennen versuchen. Dazu gibt es die Möglichkeit, den Typ eines Werts abzufragen.

Planbarkeit. Frühe Entscheidungen erleichtern die Planung weiterer Schritte. Ist eine Variable vom Typ `int`, ist klar, welche Werte die Variable enthalten kann. Statt auf Spekulationen bauen wir auf Wissen auf. Um uns auf einen Typ festzulegen, müssen wir voraussehen (also planen), wie bestimmte Programmteile im fertigen Programm verwendet werden. Wir werden zuerst jene Typen festlegen, bei denen kaum Zweifel an der künftigen Verwendung bestehen. Frühe Entscheidungen sind daher oft stabil. Unsichere Entscheidungen werden eher nach hinten verschoben und zu einem Zeitpunkt getroffen, an dem bereits viele andere damit zusammenhängende Entscheidungen getroffen wurden und der Entscheidungsspielraum entsprechend kleiner ist. Probleme können sich durch notwendige Programmänderungen ergeben: Wenn in Typen dokumentierte Entscheidungen revidiert werden müssen, sind alle davon abhängigen Programmteile anzupassen. Bei statischer Typprüfung hilft der Compiler, die betroffenen Programmstellen zu finden; an diesen Stellen werden Typen nicht mehr konsistent verwendet und müssen geändert werden.

Wahrscheinlich sind Verbesserungen der Lesbarkeit und Verständlichkeit sowie der Unterstützung früher Entscheidungen zusammen mit der Planbarkeit die wichtigsten Gründe für die Verwendung statisch geprüfter Typen. Mit Abstrichen können wir Ähnliches auch in dynamischen Sprachen erreichen. Häufig finden wir Variablennamen, welche die Art der enthaltenen Werte beschreiben.[6] Damit wird nicht nur bessere Lesbarkeit bezweckt, sondern es werden auch getroffene Entscheidungen festgehalten. Bei Einhaltung entsprechender Konventionen funktioniert das gut. Dazu braucht es viel Programmierdisziplin. Gelegentlich geht die Disziplin genau dann verloren, wenn es darauf ankommt – dann, wenn in Variablennamen dokumentierte Entscheidungen revidiert werden müssen. Es ist schwer, alle von einer frühen Entscheidung abhängigen Stellen zu finden. Oft bleiben aus Bequemlichkeit bestehende Namen erhalten, obwohl die durch die Namen suggerierten Informationen nicht mehr stimmen.

[6] Dabei geht jedoch viel von der Flexibilität dynamisch typisierter Sprachen verloren, weil gerade die dynamische Typisierung nicht genutzt wird. Das gleiche Programm hätte auch in einer stark oder statisch typisierten Sprache geschrieben werden können.

1.6.2 Nominale und strukturelle Typen

Wie wir in Abschnitt 1.4.2 gesehen haben, ermöglichen nominale Typen einige Formen der Abstraktion, die mit strukturellen Typen nicht möglich sind. Der einzige Unterschied zwischen nominalen und strukturellen Typen besteht darin, dass nominale Typen in expliziten Typdefinitionen eingeführt werden und dabei Namen bekommen. Hinsichtlich der Struktur, also des inneren Aufbaus, gibt es keinen Unterschied. Die Einführung von Namen alleine reicht schon aus, um zu bewirken, dass alle nominalen Typen vom Compiler als voneinander verschieden angesehen werden, auch wenn sie die gleiche Struktur haben. Dazu kommt der praktische Vorteil, dass die Definition eines Typs nur an einer einzigen Stelle im Programm erfolgt und alle Informationen dazu an dieser Stelle gesammelt aufzufinden sind.

Allerdings haben nominale Typen nicht nur Vorteile gegenüber strukturellen Typen, sondern auch Nachteile. Strukturelle Typen sind vor allem einfacher und flexibler verwendbar, weil keine speziellen Typdefinitionen nötig sind. Außerdem verringern sie die Gefahr von Namenskonflikten. Der einzige Nachteil struktureller Typen liegt in Schwierigkeiten im Umgang mit Abstraktionen. Wir wollen uns vor Augen führen, was das für einige Aspekte praktisch bedeutet.

Untertypen. Untertypbeziehungen werden in der objektorientierten Programmierung durch das Ersetzbarkeitsprinzip [8, 25] definiert:

> Ein Typ U ist Untertyp eines Typs T wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.

Ohne Ersetzbarkeit (siehe Abschnitt 1.3.3) gibt es also keine Untertypen.

Wie wir in Kapitel 3 sehen werden, sind Untertypbeziehungen durch das Ersetzbarkeitsprinzip für strukturelle Typen ganz eindeutig definiert. Die Theorie lässt keinen Spielraum: Sind zwei strukturelle Typen gegeben, kann ein Compiler durch Anwendung fixer Regeln automatisch ermitteln, ob ein Typ Untertyp des anderen ist [2]. Untertypbeziehungen auf strukturellen Typen sind damit implizit; wir müssen nicht angeben, dass U Untertyp von T ist.

Für nominale Typen reichen einfache Regeln nicht aus. Abstrakte und daher den Regeln nicht zugängliche Konzepte lassen sich nicht automatisch vergleichen. In der Praxis müssen wir beim Programmieren explizit hinschreiben, welcher Typ Untertyp von welchem anderen ist. Meist verwenden wir abgeleitete Klassen: Wird eine Klasse U von einer Klasse T abgeleitet, so nehmen wir an, dass U Untertyp von T ist. Dazu müssen auch die entsprechenden strukturellen Typen, also die Signaturen von U und T, in einer durch die Regeln überprüfbaren Untertypbeziehung stehen. U soll nur dann von T abgeleitet werden, wenn das Konzept hinter T durch das Konzept hinter U vollständig ersetzbar ist, sodass statt eines Objekts von T stets auch ein Objekt von U verwendbar ist. Beim Programmieren liegt die Einhaltung dieser Bedingung zur Gänze in unserer Verantwortung. Kein Compiler oder anderes Werkzeug kann sie uns abnehmen. Fälschlich angenommene Untertypbeziehungen zählen zu den folgenschwersten Fehlern in der objektorientierten Programmierung. Die Einführung

nominaler Typen ermöglicht nicht nur Abstraktion, sondern verpflichtet uns zu einem bewussten Umgang damit, weil andernfalls die gesamte Vorstellungswelt der objektorientierten Programmierung in sich zusammenbrechen würde.

Zusicherungen werden häufig in die Entscheidung von Untertypbeziehungen einbezogen. Einige wenige Programmiersprachen wie Eiffel [28] integrieren Zusicherungen als Sprachkonstrukte. Allerdings zeigt die Erfahrung, dass Zusicherungen praktisch nie so präzise und vollständig sind, dass dadurch abstrakte Konzepte bei der Entscheidung von Untertypbeziehungen außer Acht gelassen werden könnten. Das heißt, die Einbeziehung von Zusicherungen ändert nichts daran, dass Untertypbeziehungen explizit hingeschrieben und damit die Verantwortung für die Kompatibilität der Konzepte übernommen werden muss. Daher kennen die meisten Programmiersprachen keine Zusicherungen, die zum geprüften Teil der Typen gehören.[7] Trotzdem empfiehlt es sich, beim Programmieren die wichtigsten Zusicherungen zumindest in Form von Kommentaren hinzuschreiben und dadurch die gröbsten Fehler bei Untertypbeziehungen zu verhindern. Kommentare werden damit zu einem Teil der Typen, wodurch Änderungen der Kommentare ähnliche Auswirkungen wie Änderungen der Typen (konkreter, der öffentlich sichtbaren Schnittstellen) nach sich ziehen. Natürlich versteht ein Compiler Inhalte von Kommentaren genau so wenig wie die von dynamisch geprüften Zusicherungen. Explizit angegebene Untertypbeziehungen sind für den Compiler aber prüfbar und falsche Abstraktionen ergeben echte Fehler im Programmablauf.

Obwohl die Regeln zur Entscheidung von Untertypbeziehungen struktureller Typen einfach sind, implizieren sie eine bedeutende Einschränkung, die auch für nominale Typen gilt: Typen von Funktions- bzw. Methoden-Parametern (als Eingangsparameter) dürfen in Untertypen nicht stärker werden. Enthält ein Obertyp `T` z. B. eine Methode `int compareTo(T x)` (die `x` mit `this` vergleicht), kann sie im Untertyp `U` nicht durch `int compareTo(U x)` überschrieben sein. Derartiges wird in der Praxis häufig benötigt, nicht nur (allerdings häufig) für binäre Methoden wie `compareTo`. Aber es gibt keine Möglichkeit, entsprechende Typen statisch zu prüfen. Dynamische Prüfungen sind natürlich möglich.

Generizität. Während Subtyping den objektorientierten Sprachen vorbehalten ist, wird Generizität in Sprachen aller Paradigmen mit statischer Typprüfung verwendet. Da es darum geht, Typen als Parameter einzusetzen und diese Parametrisierung schon zur Übersetzungszeit aufzulösen, muss Generizität sehr tief in das Typsystem integriert sein. Für dynamisch typisierte Sprachen ist Generizität aus diesem Grund nicht sinnvoll.

[7] Für viele Sprachen, auch für Java, gibt es Bibliotheken, die Funktionalität zur dynamischen Prüfung von Zusicherungen anbieten. Praktisch werden solche Bibliotheken kaum eingesetzt, weil dynamische Prüfungen von Zusicherungen mit erheblichen Nachteilen verbunden sind. Gründe dafür sind zum Teil sehr verzwickt und nicht in wenigen Zeilen erklärbar. Ein Problem liegt darin, dass zur Laufzeit für Prüfungen verfügbare Information häufig nicht statisch antizipierbar ist, wodurch dynamisch geprüfte Zusicherungen im Gegensatz zu abstrakten Vorstellungen oft nur wenig zum statischen Programmverständnis beitragen und keinesfalls das explizite Hinschreiben von Untertypbeziehungen ersetzen können.

Einfache Generizität ist leicht zu verstehen und auch vom Compiler leicht handzuhaben. Die Komplexität steigt jedoch rasch an, wenn Einschränkungen auf Typparametern zu berücksichtigt sind, das heißt, wenn Typparameter nur durch Typen mit bestimmten Eigenschaften ersetzbar sein sollen. Im Wesentlichen gibt es zwei etwa gleichwertige formale Ansätze dafür: *F-gebundene Generizität* [7] nutzt Untertypbeziehungen zur Beschreibung von Einschränkungen und wird z. B. in Java eingesetzt. *Higher-Order-Subtyping* [1] (siehe Abschnitt 1.4.3) ist dafür ebenso geeignet und unterstützt binäre Methoden direkt. Dieser Ansatz wird auf unterschiedliche Weise beispielsweise in C++, aber auch in der funktionalen Sprache Haskell verwendet. Beide Ansätze können gut mit Fällen umgehen, mit denen Untertypen nicht oder nur schwer zurechtkommen. Deswegen unterstützen die meisten stark typisierten objektorientierten Sprachen Generizität. Jedoch ist Generizität hinsichtlich der Wartung prinzipiell kein vollwertiger Ersatz für Ersetzbarkeit.

Zur Beschreibung von Einschränkungen auf Typparametern wären strukturelle Typen sinnvoll, da es dabei nicht um Abstraktionen geht, sondern um das Vorhandensein von Funktionen oder Methoden mit bestimmten Signaturen. Allerdings wäre es für viele Menschen kaum nachvollziehbar, wenn Namen manchmal eine Rolle spielen und in anderen Fällen nicht, sodass z. B. in Java auch für solche Einschränkungen nominale Typen mit nominalem Subtyping zum Einsatz kommen. Dahinter stecken pragmatische Überlegungen. Es soll vermieden werden, mehrere einander ähnliche Hierarchien nebeneinander anzubieten, selbst wenn mehrere Hierarchien echte Vorteile hätten. Higher-Order-Subtyping basierend auf strukturellen Typen in C++ hat sich durchgesetzt, weil entsprechende Typhierarchien implizit sind, also nur vom Compiler, nicht von Menschen gesehen werden. Versuche, diese Hierarchien für Menschen sichtbar zu machen, sind bislang wegen fehlender Akzeptanz großteils gescheitert. Dagegen ist Higher-Order-Subtyping in Haskell ohne Probleme explizit sichtbar, weil dort mangels Untertypbeziehungen keine Verwechslungsgefahr besteht.

1.6.3 Gestaltungsspielraum

Zahlreiche Entscheidungen im Entwurf einer Sprache spiegeln sich in den Typen wider. Der Gestaltungsspielraum scheint endlos. Aber die Theorie zeigt klare Grenzen auf. Beispielsweise haben wir schon angedeutet, dass es im Zusammenhang mit Untertypen eine bedeutende Einschränkung gibt. Im Folgenden betrachten wir kurz einige weitere Bereiche, teilweise mit für Uneingeweihte überraschenden Möglichkeiten und Grenzen.

Rekursive Datenstrukturen. Werte vieler Typen wie Listen und Bäume sind in ihrer Größe unbeschränkt. Diese Typen sind rekursiv. Nicht selten sprechen wir von potentiell unendlichen Strukturen. Aber tatsächlich unendlich große Strukturen sind im Computer niemals darstellbar. Würden wir versuchen, Werte rekursiver Typen als möglicherweise unendlich zu betrachten, würden wir scheitern, weil Typprüfungen in eine Endlosschleife geraten.

Der einzig sinnvolle Weg zur Beschreibung von Werten nicht beschränkter Größe führt über eine induktive Konstruktion: Wir beginnen mit einer endlichen Menge M_0, die nur einfache Werte enthält, z. B. $M_0 = \{\texttt{end}\}$ wenn wir nur einen einzigen Wert `end` brauchen. Dann beschreiben wir, wie über endlich viele Möglichkeiten aus einer Menge M_i die Menge M_{i+1} ($i \geq 0$) generiert wird, wobei M_{i+1} zumindest alle Elemente von M_i enthält, z. B.

$$M_{i+1} = M_i \cup \{\texttt{elem}(n, x) \mid n \in \texttt{Int}; x \in M_i\}.$$

Die meist unendliche Menge $M = \bigcup_{i=0}^{\infty} M_i$, also die Vereinigung aller M_i, enthält dann alle beschriebenen Werte. Die als Beispiel konstruierte Menge enthält die Elemente `end`, `elem(3,end)`, `elem(7,elem(3,end))` und so weiter, also alle Listen ganzer Zahlen. Vor allem in neueren funktionalen Sprachen können Typen tatsächlich auf diese Weise konstruiert werden. Der Typ `IntList`, dessen Werte die Elemente der oben beschriebenen Menge sind, wird in Haskell so definiert (wobei „`|`" Alternativen trennt):

```
data IntList = end | elem(Int, IntList)
```

Aus der Syntax geht auf den ersten Blick hervor, dass `IntList` ein rekursiver Typ ist, weil `IntList` sowohl links als auch in zumindest einer Alternative rechts von `=` vorkommt. Rekursion beschreibt die Mengenkonstruktion nur in leicht abgewandelter Form, so wie M_i in M_{i+1} verwendet wird.

Nicht alle unendlichen Mengen sind auf diese Weise konstruierbar: Wir können z. B. Listen konstruieren, in denen alle Listenelemente vom gleichen Typ sind oder in denen sich die Typen der Listenelemente zyklisch wiederholen, aber es ist unmöglich, Listen zu konstruieren, in denen alle Listenelemente unterschiedliche Typen haben. Die Art der Konstruktion ist nicht willkürlich, sondern unterscheidet handhabbare Strukturen von solchen, deren Typen nicht in endlicher Zeit prüfbar sind. In der Praxis brauchen wir sicher nicht mehr, als auf obige Weise konstruierbar ist.

Wir müssen klar zwischen M_0 (nicht-rekursiv) und der Konstruktion aller M_i mit $i > 0$ (rekursiv) unterscheiden, wobei M_0 nicht leer sein darf. Letztere Eigenschaft heißt *Fundiertheit*. In jeder Typdefinition muss es zumindest eine nicht-rekursive Alternative (z. B. `end` in `IntList`) geben.

Viele Sprachen verwenden einen einfacheren Ansatz für nicht-elementare Typen: Die Menge M_0 ist etwa in Java schon in der Sprachdefinition vorgegeben und enthält nur den speziellen Wert `null`; Klassen beschreiben die M_i mit $i > 0$. Da statt eines Objekts immer auch `null` verwendet werden kann, ist Fundiertheit immer gegeben. Der Nachteil dieser Vereinfachung liegt auf der Hand: Wir müssen beim Programmieren stets damit rechnen, statt eines Objekts nur `null` zu erhalten, auch wenn es gar nicht um rekursive Datenstrukturen geht.

Typinferenz. Die Techniken hinter der Typinferenz sind heute weitgehend ausgereift. Für bestimmte Aufgaben wird Typinferenz sehr breit eingesetzt, auch in Java (etwa für Typparameter im Zusammenhang mit Generizität). Ein prinzipielles Problem ist bis heute jedoch ungelöst: Typinferenz funktioniert

nicht, wenn gleichzeitig, also an derselben Programmstelle, Ersetzbarkeit durch Untertypen verwendet wird. Aus diesem Grund wird Typinferenz in neueren funktionalen Sprachen, in denen keine Untertypen verwendet werden, in größtmöglichem Umfang eingesetzt, aber nur lokal und eingeschränkt in objektorientierten Sprachen.

Typinferenz erspart das Hinschreiben von Typen, ändert aber nichts an der durch statische Typprüfungen reduzierten Flexibilität. Weggelassene Typangaben schwächen obendrein die Dokumentation (Typangaben als in Programmcode gegossene Kommentare). Aber gerade im lokalen Bereich vermeidet Typinferenz das mehrfache Hinschreiben des gleichen Typs an nahe beieinander liegenden Stellen, ohne den Dokumentations-Effekt zu zerstören. Das kann die Lesbarkeit sogar verbessern.

Propagieren von Eigenschaften. Statisch geprüfte Typen sind sehr gut dafür geeignet, statische Information von einer Stelle im Programm an andere Stellen zu propagieren (= weiterzuverbreiten). Beispielsweise kann eine Funktion nur aufgerufen werden, wenn der Typ des Arguments mit dem des Parameters übereinstimmt; die Kompatibilität zwischen Operator und Operanden sicherzustellen ist ja eine wesentliche Aufgabe von Typen. Dabei wird Information über das Argument an die aufgerufene Funktion propagiert. Entsprechendes gilt auch für das Propagieren von Information von der aufgerufenen Funktion zur Stelle des Aufrufs unter Verwendung des Ergebnistyps und bei der Zuweisung eines Werts an eine Variable. Genau diese Art des Propagierens von Information funktioniert nicht nur für Typen im herkömmlichen Sinn, sondern für alle statisch bekannten Eigenschaften. Wenn wir z. B. wissen, dass das Argument ein Objekt ungleich `null`, eine Zahl zwischen 1 und 9 oder eine Primzahl ist, dann gilt genau diese Eigenschaft innerhalb der Funktion auch für den Parameter. Erst in jüngster Zeit wird das Propagieren beliebiger solcher Information in Programmiersprachen unterstützt.

Typen werden in der Regel als unveränderlich angesehen: Eine Variable, deren deklarierter Typ `int` ist, enthält immer eine entsprechende ganze Zahl, auch nachdem der Variablen ein neuer Wert zugewiesen wurde. Auch beliebige Information ist so handhabbar: Wenn der Typ eine Eigenschaft (wie ungleich `null`) impliziert, dann muss bei jeder Zuweisung an die Variable statisch sichergestellt sein, dass der zugewiesene Wert diese Eigenschaft erfüllt. Das ist der schwierige Teil. Es muss irgendwie festgelegt sein, welche Werte die gewünschten Eigenschaften besitzen. Beispielsweise sind neue Objekte genauso ungleich `null` wie Variablen nach einer expliziten Prüfung – etwa `x` als Argument von `use` in `if(x!=null)use(x);`. Unnötige dynamische Typprüfungen dieser Art möchten wir vermeiden.

Manche sinnvolle Information ist nicht beliebig propagierbar. Beispielsweise wäre es unsinnig, die Information „das ist die einzige Referenz auf das Objekt“ durch Zuweisung an mehrere Variablen zu propagieren, weil dies die Richtigkeit der Information zerstören würde; danach gäbe es ja mehrere Referenzen. Um mit solcher Information umzugehen, muss das Propagieren kontrolliert werden:

Die Information wird zwar weitergegeben, etwa vom Argument zum Parameter, aber nicht dupliziert; das Argument hätte die Information durch die Weitergabe an den Parameter verloren (wodurch das Argument nach dem Aufruf nicht mehr sinnvoll verwendbar ist). Derartiges funktioniert gut, wenn Typen nicht unveränderlich, sondern zustandsbehaftet sind; je nach Zustand des Typs ist die Information vorhanden oder nicht. Solche Typen gibt es in experimentellen Sprachen. Diese Typen sind sehr mächtig, weil sie auch Eigenschaften ausdrücken können, die für die Synchronisation benötigt werden.

Zusammenfassend gilt, dass mit statisch geprüften Typen heute sehr viel möglich ist, was noch vor wenigen Jahren unmöglich schien. Umgekehrt kennen wir aber auch unerwartete Einschränkungen, die es notwendig machen, manche Aspekte erst zur Laufzeit zu prüfen.

2 Etablierte Denkmuster und Werkzeugkisten

In diesem Kapitel nähern wir uns einigen etablierten Programmierparadigmen auf herkömmliche Weise. Wir versuchen zu klären, welche Denkmuster dominieren, welche Sprachkonzepte in entscheidendem Ausmaß zum Einsatz kommen und was hinter dem Erfolg entsprechender Programmierstile steckt.

2.1 Prozedurale Programmierung

Die prozedurale Programmierung ist das älteste Paradigma. Es wird auch heute noch sehr häufig eingesetzt. Fast alle, die jemals mit Programmierung in Kontakt gekommen sind, sind zuerst mit prozeduraler Programmierung in Kontakt gekommen. Entsprechende Denkmuster sind daher weit verbreitet und stehen für viele als Synonym für „Programmieren“. Die Vertrautheit erschwert es, über das Wesen der prozeduralen Programmierung nachzudenken, weil gefühlsmäßig fast alles darunter fällt. Wir versuchen dennoch eine klare Abgrenzung.

2.1.1 Alles im Griff

Namensgebend in der prozeduralen Programmierung sind die Prozeduren, die sich gegenseitig aufrufen und einen Programmfortschritt durch destruktive Zuweisungen, also Änderungen des Programmzustands über Seiteneffekte erzielen. Programme sind daher nicht nur über eine Menge an Prozeduren beschrieben, sondern auch durch (in der Regel globale) Daten bzw. Datenstrukturen, die es erst ermöglichen, von einem Programmzustand zu sprechen. Obwohl Goto mit der prozeduralen Programmierung kompatibel ist, werden heute vorwiegend Prozeduren nach dem Prinzip der strukturierten Programmierung geschrieben. Die Konzentration liegt auf der Programmierung im Feinen, Programmierung im Groben beschränkt sich, wenn überhaupt thematisiert, auf Module zur getrennten Übersetzung, eventuell unterstützt durch Generizität zur Parametrisierung. Weder Objekte noch Prozeduren sind als Daten im engeren Sinn verwendbar, obwohl manchmal Prozeduren als Parameter erlaubt sind.

Aus dieser Zusammenfassung sind folgende typische Eigenschaften ableitbar:

- Durch Weglassen komplexer Sprachkonzepte (Prozeduren als Daten, viele Formen von Modularisierungseinheiten und Parametrisierung, Ersetzbarkeit) ergibt sich eine überschaubare, anfängerfreundliche Menge sprachlicher Ausdrucksmittel, die sich vergleichsweise einfach und widerspruchsfrei miteinander kombinieren und jeden erdenklichen Programmablauf

darstellen lassen. Eine große Zahl an Programmieraufgaben scheint gut lösbar zu sein, sobald Variablen, Parameterübergaben, Hintereinanderausführungen, Verzweigungen und Schleifen verstanden wurden. Wenige zusätzliche Konzepte auf Seiten der Datenstrukturen wie Arrays, Records und Zeiger vervollständigen das Bild.

- Der Programmablauf wird fast zur Gänze durch Kontrollstrukturen festgelegt, die Strukturierung der Daten durch wenige elementare Datenstrukturen wie Arrays, Records und Zeiger. Diese Eigenschaft gibt uns sehr viel Kontrolle. Jedes Detail wird kontrolliert, nicht nur der Programmablauf, sondern auch die genaue Strukturierung der Daten.

- Wegen der guten Kontrollmöglichkeiten ist die prozedurale Programmierung besonders gut für die hardwarenahe Programmierung geeignet. Auch unübliche Hardware lässt sich meist vergleichsweise einfach ansprechen.

- Kontrolle wird nicht nur ermöglicht, sondern erzwungen. Wir müssen sogar in jenen Bereichen jedes feine Detail festlegen, wo es auf die Details gar nicht ankommt. Diese Eigenschaft lässt auch Programme mittlerer Größe und von mittlerem Schwierigkeitsgrad schon sehr komplex wirken, weil der Überblick über die vielen Details rasch verloren geht.

- Abstraktion erfolgt nur über Prozeduren. Abstrakte Denkweisen spielen deshalb eine eher untergeordnete Rolle. Obwohl nominale Abstraktion sinnvoll einsetzbar ist, wird dennoch häufig nur λ-Abstraktion eingesetzt.

- Vollständige Kontrolle und niedrige Abstraktionsgrade sind die besten Voraussetzungen für die Entwicklung von Programmen, die formal auf Korrektheit überprüft werden können. Prozedurale Programmierung ist gut mit formalen Korrektheitsbeweisen kombinierbar.

- Rekursive Prozeduraufrufe sind heute fast überall möglich, aber Rekursion wird meist nur dann angewandt, wenn äquivalente Schleifenkonstruktionen deutlich mehr Aufwand verursachen würden. Das wird meist mit besserer Laufzeiteffizienz und besserer Kontrolle begründet, insbesondere im Hinblick auf die Auslastung des Stacks für Prozeduraufrufe.

- Programme sind entweder nur dynamisch oder (bis auf Arrayzugriffe und Ähnliches) nur statisch typisiert. Mischformen wie in der objektorientierten Programmierung kommen nicht zum Einsatz. Dynamisch typisierte Programmierung herrscht vor allem in einem semiprofessionellen Bereich vor, wo Programme eher einfach sind und die Anzahl sprachlicher Ausdrucksmittel auf ein absolutes Mindestmaß reduziert bleiben soll. Statische Typisierung herrscht dort vor, wo es auf Hardwarenähe, Laufzeiteffizienz und sehr gute Kontrollmöglichkeiten ankommt, die Einfachheit der sprachlichen Ausdrucksmittel dagegen nicht so wichtig ist.

Zusammengefasst überzeugt die prozedurale Programmierung einerseits durch einfach handhabbare Werkzeuge, andererseits durch gute Kontrollmöglichkeiten. Beides hat auch Nachteile: Die Notwendigkeit der Kontrolle und das Fehlen fortgeschrittener Werkzeuge für den Umgang mit Abstraktionen lässt den Aufwand für das Erstellen und Warten umfangreicher, komplexer Software gewaltig ansteigen. Wer prozedural programmiert, möchte und muss jedes Detail im Griff haben. Geht der Überblick verloren, entstehen unweigerlich Fehler, die üblicherweise nicht durch sprachliche Mittel eingegrenzt oder verziehen werden.

Typische prozedurale Programme in Java verwenden statische Methoden als Prozeduren, Klassen als Module, Klassenvariablen als globale Variablen und Objekte von Klassen ohne Methoden als Records. Das sind im Großen und Ganzen jene Programmiertechniken, von deren Verwendung in der Java-Programmierung ständig abgeraten wird. Der Grund ist einfach: Zuerst lernen wir meist die Grundkonzepte der prozeduralen Programmierung; sobald wir sie beherrschen, sollen wir in Java weg von der prozeduralen hin zur objektorientierten Programmierung geführt werden. Im Kontext der prozeduralen Programmierung verwenden wir jedoch genau diese in der objektorientierten Programmierung verpönten Techniken:

```
import java.io.*;
import java.util.Scanner;

public class ProceduralCourse {
    private final static int MAX = 1000;
    private final static Student[] studs = new Student[MAX];
    private final static int[] exercises = new int[MAX];
    private final static int[] test1 = new int[MAX];
    private final static int[] test2 = new int[MAX];
    private static int used = 0;

    private static void readStudents(InputStream stream) {
        Scanner in = new Scanner(stream);
        while (in.hasNextInt()) {
            if (used >= MAX)
                throw new IndexOutOfBoundsException("too much");
            studs[used] = new Student();
            studs[used].regNo = in.nextInt();
            for (int i = 0; i < used; i++)
                if (studs[i].regNo == studs[used].regNo)
                    throw new RuntimeException("double regNo");
            studs[used].curr = in.nextInt();
            studs[used].name = in.nextLine().strip();
            used++;
        }
    }
```

```
    private static void readP(int[] xs, InputStream stream) {
        Scanner in = new Scanner(stream);
        while (in.hasNextInt())
            set(xs, in.nextInt(), in.nextInt());
    }
    private static void set(int[] xs, int r, int p) {
        if (p <= 0 || p > 100)
            throw new RuntimeException("inappropriate points");
        for (int i = 0; i < used; i++)
            if (studs[i].regNo == r) {
                if (xs[i] != 0)
                    throw new RuntimeException("contradictory");
                xs[i] = p;
                return;
            }
        throw new RuntimeException("unknown regNo");
    }
    private static void printStatus() {
        for (int i = 0; i < used; i++)
            System.out.println(studs[i].regNo + ", "
                + studs[i].name + " (" + studs[i].curr + "): "
                + exercises[i] + "+"
                + test1[i] + "+" + test2[i] + " = "
                + (exercises[i] + test1[i] + test2[i]));
    }
    public static void main(String[] args) throws IOException {
        if (args.length != 4)
            throw new IllegalArgumentException("wrong params");
        readStudents(new FileInputStream(args[0]));
        readP(exercises, new FileInputStream(args[1]));
        readP(test1, new FileInputStream(args[2]));
        readP(test2, new FileInputStream(args[3]));
        printStatus();
    }
}

class Student {
    public int regNo;
    public int curr;
    public String name;
}
```

Das Beispiel ist nicht rein prozedural, weil vordefinierte Klassen wie `Scanner` und `FileInputStream` zum Einsatz kommen, die eine Verwendung im objektorientierten Stil verlangen. In Java ist es schwer, alle Objekte auszublenden. Aber das Wesentliche kommt doch zum Ausdruck: Es wird viel mit globalen

Daten gearbeitet, obwohl das auch in der prozeduralen Programmierung eher vermieden werden soll. Die Alternative wäre, Prozeduren mit vielen Parametern zu verwenden, was aber oft auf Kosten der Verständlichkeit geht. Klassen, die als Module nur Klassenvariablen und statische Methoden enthalten, können durchaus über Data-Hiding Inhalte vor Zugriffen von außen schützen. Bei Verwendung als Record wie für `Student` ist jedoch eine Objekterzeugung nötig (das Objekt ist ein Record) und für Zugriffe von außen müssen Objektvariablen `public` sein. Es wäre möglich, ein Objekt von `Student` als abstrakte Einheit zu betrachten, darauf wird aber weitgehend verzichtet; die Variablen in `Student` werden beinahe so verwendet, als ob sie unabhängig voneinander wären.

Das Beispiel ist klein gehalten, indem Plausibilitätsprüfungen eingelesener Daten minimalistisch sind und Fehler nur über Ausnahmen gemeldet werden. Genauere Prüfungen und informativere Fehlermeldungen würden die Komplexität des Programms ansteigen lassen. Die Einfachheit ist also relativ zu sehen.

2.1.2 Totgesagte leben länger

Wer die objektorientierte oder funktionale Programmierung gewohnt ist, empfindet die prozedurale Programmierung vielleicht als längst überholt. Dabei wird übersehen, dass die prozedurale Programmierung zwar nicht überall, aber in bestimmten Bereichen durchaus vorteilhaft ist und ganz bewusst eingesetzt wird. Wir wollen einige dieser Bereiche kurz ansprechen.

Hardwarenahe Programmierung. Wenn es notwendig ist, spezielle Hardware anzusprechen, ist Kontrolle über den Programmablauf und die Position im Speicher, an der Daten abgelegt werden, unerlässlich. Diese Forderungen lassen sich in der prozeduralen Programmierung am besten erfüllen. Höhere Formen der Abstraktion wären nur hinderlich. Allerdings kommen hauptsächlich Programmiersprachen zum Einsatz, die für die hardwarenahe Programmierung ausgelegt sind. Java wird kaum verwendet, weil es auf herkömmliche Weise nicht möglich ist, bestimmte Speicheradressen anzusprechen. Es kommt also nicht nur auf den Programmierstil an, sondern auch darauf, was in der Sprache einfach ausdrückbar ist. Wenn Laufzeiteffizienz von großer Bedeutung ist und die Hardware ausreichend Speicher hat, sind stark typisierte Sprachen wie C ideal. Auf sehr kleinen Mikrocontrollern ist Speicherplatz oft zu begrenzt für umfangreiche C-Programme. Dafür kommen auch interpretierte, dynamisch typisierte oder sogar untypisierte Sprachen wie etwa Forth und Assembler zum Einsatz, die sehr sparsam mit Speicher umgehen können. Details der Programmierstile richten sich ganz nach den Einschränkungen der Hardware. Der Zusatzaufwand durch das notwendige Ansprechen spezieller Hardware ist in der Regel so groß, dass wirklich große Programme sowieso kaum erstellt werden und die Nachteile der prozeduralen Programmierung daher kaum wirksam werden.

Echtzeitprogrammierung. Hierbei kommt es darauf an, dass maximale Antwortzeiten eingehalten werden. Ein prozeduraler Programmierstil erleichtert die

Kontrolle der Programmabläufe und Datenauslegungen. Obwohl Java ursprünglich nicht dafür entwickelt wurde, ist es durchaus möglich, Echtzeitprogramme in Java zu schreiben, falls es nicht zu problematisch ist, wenn Antwortzeiten manchmal nicht eingehalten werden (sogenannte weiche Echtzeitsysteme). Trotzdem sind zur möglichst guten Erreichung der zeitlichen Ziele gut kontrollierte, also prozedurale Programmierstile nötig. Häufig reicht es aber, wenn einschränkende prozedurale Stile nur in bestimmten Programmteilen verwendet werden, während andere Teile z. B. objektorientierte Stile verwenden.

Ganz anders ist die Situation bei harten Echtzeitsystemen, wo die Einhaltung der Antwortzeiten absolut notwendig ist, insbesondere bei sicherheitskritischen Echtzeitsystemen. Dafür wird häufig eine spezielle Form der hardwarenahen Programmierung eingesetzt, manchmal mit redundanter Auslegung der Hard- und Software. Alle Abläufe und auch die Datenstrukturen müssen ganz im Hinblick auf die Abschätzbarkeit der Antwortzeiten ausgelegt sein. In der Praxis bedeutet das zahlreiche Einschränkungen in den verwendbaren Kontrollstrukturen und Befehlen genauso wie in den einsetzbaren Datenstrukturen und Speicherauslegungen. Im sicherheitskritischen Bereich kann auch der Einsatz von formalen Beweistechniken nötig sein. Daher kommen nur sehr eingeschränkte prozedurale Programmierstile in Frage.

Scripting. Speziell unter Unix ist ein Shell-Skript ein kleines, prozedurales Programm, das als Befehle Programmaufrufe verwendet. Daraus hat sich ein eigenständiger Zweig der Programmierung entwickelt, in dem mittels meist interpretierter und dynamisch typisierter Sprachen Programme geschrieben werden, die bereits bestehende Programme aufrufen, miteinander kombinieren und in einen neuen Kontext einbetten. Solche Programme werden in der Regel in kurzer Zeit im Hinblick auf eine bestimmte Aufgabe entwickelt und sind nur für den einmaligen Gebrauch (also nicht für eine langfristige Wartung) oder die wiederholte Anwendung in einem ganz bestimmten, sehr eng gefassten Kontext (also ohne Portabilität) ausgelegt, wobei Entwickler_innen und Anwender_innen häufig die gleichen Personen sind (also wenig Aufwand betrieben wird, um hohe Qualität zu erreichen). Ohne dafür ausgelegt zu sein, werden solche Programme in der täglichen Praxis dennoch wiederholt zum Einsatz gebracht (weil sie schon da sind) und immer wieder an sich ändernde Situationen und Kontexte angepasst (weil das, was da ist, doch nicht ganz passt). Meist bleiben die Programme relativ klein. Die meisten derartigen Programme entsprechen ganz der prozeduralen Programmierung, obwohl entsprechende Sprachen oft Konzepte für die objektorientierte Programmierung anbieten. Das ist verständlich, weil es unsinnig wäre, den Zusatzaufwand der objektorientierten Programmierung zu tragen, wenn die Software nicht für die langfristige Wartung ausgelegt ist. Scripting kommt heute in einem weiten Feld zum Einsatz. Typische Anwendungsbereiche sind die regelmäßige Überprüfung und Wartung von Rechnernetzwerken, wiederholt durchzuführende Datenaufbereitungen, sowie die Bereitstellung von Daten aus Datenbanken auf Webseiten. Oft ist Scripting die einzig mögliche, aus der Not heraus geborene Vorgehensweise, wenn Software rasch benötigt wird

und nicht genug Zeit für die Entwicklung einer qualitativ hochwertigeren und für die längerfristige Wartung ausgelegten Lösung zur Verfügung steht. In solchen Fällen wird häufig nicht nur die Wartbarkeit der Software vernachlässigt, sondern auch die Robustheit und der Datenschutz.

Micro-Services. Immer wieder entstehen neue Softwarearchitekturen und entsprechende Programmiertechniken, vor allem in der verteilten Programmierung. In letzter Zeit wurden Micro-Services populär. Ein Micro-Service ist ein auf einem bestimmten Rechner laufender Prozess (also eine Programmausführung), der eine bestimmte, relativ kleine Aufgabe erledigt, etwa einen Bestellvorgang durchführt. Eine klar vorgegebene Schnittstelle verbindet das Micro-Service mit der Außenwelt. Komplexe Anwendungssoftware ist aus vielen Micro-Services zusammengesetzt. Jedes Micro-Service für sich genommen ist klein genug, um trotz hoher Qualität mit vertretbarem Zeitaufwand entwickelt (und bei Bedarf durch ein anderes Micro-Service ersetzt) werden zu können. Die Komplexität eines Micro-Services bleibt dadurch überschaubar. Es wird immer wieder betont, dass jedes Micro-Service mit unterschiedlicher Technologie in unterschiedlichen Programmierstilen realisiert sein kann. Dennoch fördert diese Vorgehensweise prozedurale Programmierstile, einfach deswegen, weil wegen der Überschaubarkeit keine komplexeren Mechanismen eingesetzt werden müssen.

Micro-Services stehen hier stellvertretend für viele Software-Architekturen und Vorgehensweisen, deren Ziel es ist, hochkomplexe und umfangreiche Software so in Einzelteile herunterzubrechen, dass jeder Teil für sich überschaubar bleibt und weitgehend unabhängig von anderen Teilen entwickelt werden kann. Das ermöglicht prozedurale Programmierung für die Einzelteile. Die Weiterführung dieses Gedankens lässt erkennen, dass die prozedurale Programmierung im Kern praktisch jedes Programmierparadigmas steckt.

Hobby-Programmierung und Anwendungsprogrammierung. Der Einstieg in das Programmieren erfolgt fast immer über die prozedurale Programmierung. In der Regel steht am Anfang statische Typisierung, wenn längerfristig eine professionelle Form der Softwareentwicklung angestrebt wird. Dagegen steht am Anfang dynamische Typisierung wenn erwartet wird, dass die Programmierung im Hobby-Bereich oder in der Anwendungsprogrammierung bleibt. Diese Unterscheidung ist historisch gewachsen und entstammt der Erfahrung, dass dynamisch typisierte Programmierung mit einfachen Sprachen rasche Erfolgserlebnisse beschert, es aber erschwert, sich später komplexe Programmiertechniken anzueignen. Heute gibt es eine riesige, ständig wachsende Zahl an Menschen unterschiedlicher beruflicher oder fachlicher Ausrichtungen, die gelernt haben, in einem eher hobbymäßigen Bereich zu programmieren. Viele davon betreiben die Programmierung gerne und auf durchaus hohem Niveau. Sie sind in der Lage und willens, ihre Fähigkeiten auch in einem professionellen Bereich einzusetzen. Aber sie bleiben häufig bei der prozeduralen Programmierung (oft trotz Verwendung einer objektorientierten Sprache). Solche Menschen sind für die Softwareentwicklung wertvoll, weil sie neben ihren Programmierfähigkeiten tiefgehende

Fähigkeiten in anderen Bereichen mitbringen, die als *Domain-Wissen* auf Anwendungsgebieten der zu erstellenden Software gebraucht werden. Es wird von *Anwendungsprogrammierung* gesprochen, wenn Domain-Wissen auf dem Gebiet der Anwendung einen bedeutenden Aspekt in der Softwareerstellung ausmacht. Allerdings steht diese Form der Programmierung häufig vor dem Problem, dass Wege gefunden werden müssen, wie auch komplexe Software auf Basis prozeduraler Programmierung erstellt werden kann. Es braucht Expert_innen, die entsprechende Softwarearchitekturen entwerfen können. Dafür reicht Wissen über prozedurale Programmierung nicht aus, sondern es ist nötig, die Ideen hinter unterschiedlichen Paradigmen zu verstehen und so miteinander zu kombinieren, dass sich ein Gefüge ergibt, in dem alle beteiligten Menschen das Maximum aus ihren jeweiligen Fähigkeiten herausholen können. Ein pragmatischer Umgang mit Programmierparadigmen ist dabei ein entscheidender Faktor.

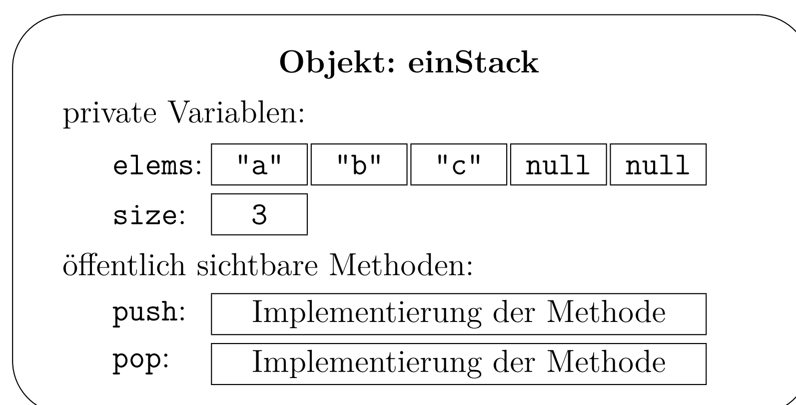
2.2 Objektorientierte Programmierung

Die objektorientierte Programmierung zählt zu den erfolgreichsten Paradigmen der Informatik. Der Inhalt dieses Abschnitts sollte großteils schon aus anderen Lehrveranstaltungen bekannt sein. Durch die Wiederholung soll vor allem die verwendete Terminologie wieder in Erinnerung gerufen werden.

2.2.1 Modularisierungseinheiten und Abstraktion

Die objektorientierte Programmierung will Softwareentwicklungsprozesse unterstützen, die auf inkrementeller Verfeinerung aufbauen. Bei diesen Prozessen spielt die leichte Wartbarkeit eine große Rolle. Im Wesentlichen geben uns objektorientierte Sprachen Werkzeuge in die Hand, die wir zum Schreiben leicht wiederverwendbarer und änderbarer Software brauchen.

Objekt. Ein Objekt ist eine grundlegende Modularisierungseinheit – siehe Abschnitt 1.3.1. Zur Laufzeit besteht die Software aus einer Menge von Objekten, die einander teilweise kennen und untereinander *Nachrichten* (*Messages*) austauschen. Das Schicken einer Nachricht entspricht dem Aufruf einer Methode. Folgende Abbildung zeigt ein Objekt:



Dieses Objekt mit der Funktionalität eines Stacks fügt zwei Variablen und zwei Methoden zu einer Einheit zusammen und grenzt sie vom Rest des Systems weitgehend ab. Eine Variable enthält ein Array mit dem Stackinhalt, die andere die Anzahl der Stackelemente. Das Array kann höchstens fünf Stackelemente halten. Zur Zeit sind drei Einträge vorhanden.

Jedes Objekt besitzt folgende Eigenschaften [35]:

Identität (Identity): Das Objekt ist durch seine unveränderliche Identität eindeutig bestimmt. Über seine Identität kann ihm eine Nachricht geschickt werden. Vereinfacht stellen wir uns die Identität als die Speicheradresse des Objekts vor. Dies ist eine Vereinfachung, da die Identität erhalten bleibt, wenn sich die Adresse ändert, etwa bei einer Speicherbereinigung oder beim Auslagern in eine Datenbank. Jedenfalls sind gleichzeitig durch zwei Namen bezeichnete Objekte identisch, wenn sie am selben Speicherplatz liegen, es sich also um ein Objekt mit zwei Namen handelt.

Zustand (State): Die Werte der Variablen im Objekt bilden den Zustand. Er ist meist änderbar. In obigem Beispiel ändert sich der Zustand durch Zuweisung neuer Werte an `elems` und `size`.

Zwei Objekte sind *gleich*, wenn sie den gleichen Zustand und das gleiche Verhalten haben. Identische Objekte sind trivialerweise gleich. Zwei Objekte können auch gleich sein, ohne identisch zu sein; dann sind sie *Kopien* voneinander. Zustände gleicher Objekte können sich unabhängig voneinander ändern; die Gleichheit geht dadurch verloren. Identität geht durch Zustandsänderungen nicht verloren. In der Praxis verwenden wir viele Varianten des Begriffs Gleichheit. Oft werden nur manche, nicht alle Variableninhalte in den Vergleich einbezogen. Gelegentlich betrachten wir nur identische Objekte als gleich.

Verhalten (Behavior): Das Verhalten des Objekts beschreibt, wie sich das Objekt beim Empfang einer Nachricht verhält. In obigem Beispiel wird die Methode `push` beim Empfang der Nachricht `push("d")` das Argument `"d"` in den Stack einfügen (falls es einen freien Platz gibt), und `pop` wird beim Empfang von `pop()` ein Element entfernen (falls eines vorhanden ist) und an den Absender zurückgeben.

Schnittstelle und Implementierung. Unter der *Implementierung* einer Methode verstehen wir den Programmcode, der festlegt, was bei Ausführung der Methode zu tun ist. Die Implementierungen aller Methoden eines Objekts, die Deklarationen der Objektvariablen und die Konstruktoren bilden zusammen die Implementierung des Objekts. Die Implementierung beschreibt das Verhalten bis ins kleinste Detail. Für die Programmausführung ist die genaue Beschreibung essentiell; sonst wüsste der Computer nicht, was zu tun ist.

Für die Wartung ist es günstiger, wenn die Beschreibung des Verhaltens nicht jedes Detail widerspiegelt. Statt der Implementierung haben wir eine abstrakte Vorstellung im Kopf, die Details offen lässt. Beispielsweise gibt

`push` fügt beim Empfang der Nachricht `push("d")` das Argument `"d"` in den Stack ein (falls es einen freien Platz gibt)

eine abstrakte Vorstellung davon wider, was tatsächlich passiert. Es bleibt offen, wie und wo `"d"` eingefügt wird und wann Platz frei ist. Menschen können mit solchen (nominalen) Abstraktionen einfacher umgehen als mit Implementierungen. Bei Computern ist es genau umgekehrt. Daher wollen wir Beschreibungen des Objektverhaltens so weit wie möglich abstrakt halten und erst dann zur Implementierung übergehen, wenn dies für den Computer notwendig ist.

Wir fordern eine weitere Objekteigenschaft, die den Abstraktionsgrad des Verhaltens nach Bedarf steuern lässt:

Schnittstelle (Interface): Eine Schnittstelle eines Objekts beschreibt das Verhalten des Objekts in einem Abstraktionsgrad, der für Zugriffe von außen notwendig ist. Ein Objekt kann mehrere Schnittstellen haben, die das Objekt aus den Sichtweisen unterschiedlicher Verwendungen beschreiben. Manchmal entsprechen Schnittstellen nur Signaturen (strukturelle Abstraktion), meist ergeben sich durch Verwendung nominaler Typen begleitet von Kommentaren nominale Abstraktionen – siehe Abschnitte 1.4.2 und 1.6.2. Über Zusicherungen kann das Verhalten beliebig genau beschreiben werden. Ein Objekt *implementiert* seine Schnittstellen; das heißt, die Implementierung des Objekts legt das in den Schnittstellen unvollständig beschriebene Verhalten im Detail fest. Jede Schnittstelle kann das Verhalten beliebig vieler Objekte beschreiben. In Sprachen mit statischer Typprüfung entsprechen Schnittstellen den Typen des Objekts.

Beim Zugriff auf ein Objekt müssen wir nur eine Schnittstelle kennen, nicht den Objektinhalt. Nur das, was in den Schnittstellen beschrieben ist, ist von außen sichtbar. Schnittstellen trennen also die Innen- von der Außenansicht und sorgen für Data-Hiding. Kapselung zusammen mit Data-Hiding ergibt Datenabstraktion: Die Daten sind nicht direkt sichtbar und manipulierbar, sondern abstrakt. Im Beispiel sehen wir die Daten des Objekts nicht als Array von Elementen zusammen mit der Anzahl der Einträge, sondern als abstrakten Stack, der über Methoden zugreifbar und manipulierbar ist. Die Abstraktion bleibt unverändert, wenn wir das Array gegen eine andere Datenstruktur, etwa eine Liste, austauschen. Datenabstraktionen helfen bei der Wartung: Details von Objekten sind änderbar, ohne deren Außenansichten und damit deren Verwendungen zu beeinflussen. Außerdem ist die abstrakte Außenansicht viel einfacher verständlich als die Implementierung mit all ihren Details.

Klasse. Viele, aber nicht alle objektorientierten Sprachen beinhalten ein Klassenkonzept: Jedes Objekt gehört zu der Klasse, in der das Objekt implementiert ist. Die Klasse beschreibt auch *Konstruktoren* zur Initialisierung neuer Objekte. Alle Objekte, die zur Klasse gehören, wurden durch Konstruktoren dieser Klasse initialisiert. Wir nennen diese Objekte auch *Instanzen* der Klasse. Genauer gesagt sind die Objekte Instanzen der durch die Klasse beschriebenen Schnittstellen bzw. Typen. Die Klasse selbst ist die spezifischste Schnittstelle mit der genauesten Verhaltensbeschreibung. Ein Stack könnte so implementiert sein:

```
public class Stack {
    private String[] elems;
    private int size = 0;
    public Stack(int sz) { // new stack of size sz
        elems = new String[sz];
    }
    // push pushes elem onto stack if not yet full
    public void push(String elem) {
        if (size < elems.length) elems[size++] = elem;
    }
    // pop returns element taken from stack, null if empty
    public String pop() {
        if (size > 0) return elems[--size]; else return null;
    }
}
```

Alle Objekte einer Klasse haben dieselbe Implementierung und auch dieselben Schnittstellen. Aber unterschiedliche Objekte haben ihre eigenen Identitäten und Objektvariablen, obwohl diese Variablen gleiche Namen und Typen tragen. Auch die Zustände können sich unterscheiden.

Die Kommentare sind Zusicherungen, also mehr als nur Erläuterungen zum besseren Verständnis. Sie beschreiben das abstrakte Verhalten, soweit dies für Aufrufer relevant ist. Wer einen Aufruf in das Programm einfügt, soll sich auf die Zusicherungen in Form von Kommentaren verlassen und nicht die Implementierung betrachten müssen. Details der Implementierung können geändert werden, solange die unveränderten Kommentare auf die geänderte Implementierung zutreffen.

Anmerkung: Häufig wird gesagt, ein Objekt gehöre zu mehreren Klassen, der spezifischsten und deren Oberklassen. Im Skriptum verstehen wir unter der „Klasse des Objekts“ immer die spezifischste Schnittstelle und sprechen allgemein von „Schnittstelle“, wenn wir eine beliebige meinen.

2.2.2 Polymorphismus und Ersetzbarkeit

Im Zusammenhang mit Programmiersprachen sprechen wir von *Polymorphismus,* wenn eine Variable oder Methode gleichzeitig mehrere Typen haben kann. An einen formalen Parameter einer polymorphen Methode können Argumente von mehr als nur einem Typ übergeben werden. Objektorientierte Sprachen sind polymorph. Im Gegensatz dazu sind statisch typisierte Sprachen wie C und Pascal *monomorph*: Jede Variable oder Funktion hat einen eindeutigen Typ.

Wir unterscheiden verschiedene Arten des Polymorphismus [8]:

- Polymorphismus
  - universeller Polymorphismus
    - Generizität
    - Untertypbeziehungen
  - Ad-hoc-Polymorphismus
    - Überladen
    - Typumwandlung

Nur beim universellen Polymorphismus haben die Typen, die zueinander in Beziehung stehen, eine gleichförmige Struktur. Generizität erreicht die Gleichförmigkeit durch gemeinsamen Code, der über Typparameter mehrere Typen haben kann. Bei Verwendung von Untertypen wird Gleichförmigkeit durch gemeinsame Schnittstellen für unterschiedliche Objekte erzielt. Überladene Methoden müssen, abgesehen vom gemeinsamen Namen, keinerlei Ähnlichkeiten in ihrer Struktur besitzen und sind daher ad-hoc-polymorph. Auch bei Typumwandlungen (Casts), insbesondere auf elementaren Typen wie von `int` auf `double`, gibt es keine gemeinsame Struktur; `int` und `double` sind intern ja ganz unterschiedlich dargestellt. Typumwandlungen auf Referenztypen fallen dagegen eher in die Kategorie der Untertypbeziehungen.

In der objektorientierten Programmierung sind Untertypen von überragender Bedeutung, die anderen Arten des Polymorphismus existieren eher nebenbei. Daher wird alles, was mit Untertypen zu tun hat, oft auch *objektorientierter Polymorphismus* oder kurz nur Polymorphismus genannt.

In einer objektorientierten Sprache hat eine Variable (in Java jede Referenz-Variable beliebiger Art, auch ein Parameter) gleichzeitig folgende Typen:

Deklarierter Typ: Das ist der Typ, mit dem die Variable deklariert wurde. Dieser existiert natürlich nur in Sprachen mit expliziter Typdeklaration.

Dynamischer Typ: Das ist der *spezifischste Typ,* den der in der Variablen gespeicherte Wert hat. Dynamische Typen sind oft spezifischer als deklarierte und können sich mit jeder Zuweisung ändern. Der Compiler kennt dynamische Typen im Allgemeinen nicht. Dynamische Typen werden unter anderem für dynamisches Binden eingesetzt.

Statischer Typ: Dieser Typ wird vom Compiler (statisch) ermittelt und liegt irgendwo zwischen deklariertem und dynamischem Typ. In vielen Fällen ordnet der Compiler derselben Variable an unterschiedlichen Stellen verschiedene statische Typen zu. Solche Typen werden beispielsweise für Programmoptimierungen verwendet. Es hängt von der Qualität des Compilers ab, wie spezifisch der statische Typ ist. In Sprachdefinitionen kommen statische Typen (wenn nicht gleich den deklarierten) daher nicht vor.

Eine Konsequenz aus der Verwendung von Untertypen ist *dynamisches Binden:* Wenn eine Nachricht an ein Objekt geschickt wird, kennt der Compiler nur den statischen und deklarierten Typ des Empfängers der Nachricht. Die Methode soll aber entsprechend des dynamischen Typs ausgeführt werden. Daher kann die auszuführende Methode erst zur Laufzeit bestimmt werden. In manchen Situationen kennt der Compiler die auszuführende Methode, etwa wenn die Methode als `static`, `final` oder `private` deklariert ist, und kann *statisches Binden* verwenden, sodass zur Laufzeit keine Bestimmung der passenden Methode nötig ist. Durch mögliche Optimierungen ist statisches Binden etwas effizienter als dynamisches. Dennoch ist das dynamische Binden einer der wichtigsten Eckpfeiler der objektorientierten Programmierung. Ein sinnvoller Umgang mit dynamischem Binden kann die Wartbarkeit erheblich verbessern, ohne dass dies zu Effizienzverlusten führen muss.

Vererbung. Die *Vererbung* (*Inheritance*) ermöglicht es, neue Klassen aus bereits existierenden Klassen abzuleiten. Dabei werden nur die Unterschiede zwischen der *abgeleiteten Klasse* (*Derived-Class, Unterklasse, Subclass*) und der *Basisklasse* (*Base-Class, Oberklasse, Superclass*), von der abgeleitet wird, angegeben. Vererbung erspart uns etwas Schreibaufwand. Außerdem werden einige Programmänderungen vereinfacht, da sich Änderungen von Klassen auf alle davon abgeleiteten Klassen auswirken. Diese Vorteile werden jedoch vielfach deutlich überschätzt.

In populären objektorientierten Programmiersprachen können bei der Vererbung Unterklassen im Vergleich zu Oberklassen nicht beliebig geändert werden. Eigentlich gibt es nur zwei Änderungsmöglichkeiten:

Erweiterung: Die Unterklasse erweitert die Oberklasse um neue Variablen, Methoden und Konstruktoren.

Überschreiben: Methoden der Oberklasse werden in der Unterklasse durch neue Methoden überschrieben, die jene aus der Oberklasse ersetzen.

Diese beiden Änderungsmöglichkeiten sind beliebig kombinierbar.

In Sprachen wie Java besteht ein enger Zusammenhang zwischen Vererbung und Untertypen: Ein Objekt einer Unterklasse kann, soweit es vom Compiler prüfbar ist, überall verwendet werden, wo ein Objekt einer Oberklasse erwartet wird. Änderungsmöglichkeiten bei der Vererbung sind eingeschränkt, um die Ersetzbarkeit von Objekten der Oberklasse durch Objekte der Unterklasse zu ermöglichen. Es besteht eine direkte Beziehung zwischen Klassen und Typen: Die Klasse eines Objekts ist gleichzeitig der spezifischste Typ bzw. die spezifischste Schnittstelle des Objekts. Dadurch entspricht eine Vererbungsbeziehung begleitet von Verhaltensbeschreibungen einer Untertypbeziehung. Verhaltensbeschreibungen der Methoden in Unterklassen müssen denen in Oberklassen entsprechen, können aber genauer sein. Das ist eine Voraussetzung für Untertypbeziehungen, obwohl der Compiler Kommentare nicht überprüfen kann.

Java unterstützt nur *Einfachvererbung* (*Single-Inheritance*) zwischen Klassen im engeren Sinn. Das heißt, jede Unterklasse wird von genau einer anderen Klasse abgeleitet. Die Verallgemeinerung dazu ist *Mehrfachvererbung* (*Multiple-Inheritance*), wobei jede Klasse mehrere Oberklassen haben kann. Mehrfachvererbung gibt es zum Beispiel in C++. Neben Klassen gibt es in Java *Interfaces*, auf denen es auch Mehrfachvererbung gibt. Interfaces sind eingeschränkte Klassen (im weiteren Sinn), die keine Objektvariablen enthalten und von denen keine Instanzen erzeugt werden können. Wir verwenden in diesem Skriptum für das Sprachkonstrukt in Java den englischen Begriff, während wir mit dem gleichbedeutenden deutschen Begriff Schnittstellen im Allgemeinen bezeichnen.

2.2.3 Typische Vorgehensweisen

Faktorisierung. Zusammengehörende Programmteile (wie Variablen und Methoden) sollen zu Modularisierungseinheiten zusammengefasst werden. Wir sprechen eher von *Faktorisierung* (*Factoring*) statt Modularisierung, weil sich nicht

nur Module, sondern vorwiegend Klassen und zur Laufzeit den Klassen entsprechende Objekte ergeben. Gute Faktorisierung bewirkt, dass Programmänderungen lokal an jeweils nur einer Stelle durchzuführen sind. Bei schlechter Faktorisierung müssen für eine einzige Änderung viele, einander ähnliche Programmstellen gefunden und geändert werden. Gute Faktorisierung verbessert auch die Lesbarkeit des Programms, weil bestimmte Programmteile an nur einer einzigen Stelle zu finden sind, nicht verteilt auf viele Programmstellen.

Objekte dienen durch Kapselung zusammengehöriger Programmteile der Faktorisierung. Durch Zusammenfügen von Daten mit Methoden ergeben sich mehr Freiheiten zur Faktorisierung als in anderen Paradigmen, bei denen Daten und Funktionen voneinander getrennt sind. Gute Faktorisierung kann die Wartbarkeit verbessern. Die objektorientierte Programmierung bietet viele Möglichkeiten zur Faktorisierung. Bei Weitem nicht jede Faktorisierung ist gut. Es ist unsere Aufgabe, die Qualität von Faktorisierungen zu beurteilen.

Die Lesbarkeit eines Programms wird erhöht, wenn die Zerlegung in Objekte so erfolgt, wie es der Erfahrung in der realen Welt entspricht. Software-Objekte sollen die reale Welt simulieren, soweit dies zur Erfüllung der Aufgaben sinnvoll erscheint. Namen für Software-Objekte sollen üblichen Bezeichnungen realer Objekte entsprechen. Wir dürfen die Simulation aber nicht zu weit treiben. Vor allem sollen keine Eigenschaften der realen Welt simuliert werden, die für die entwickelte Software bedeutungslos sind. Die Einfachheit ist wichtiger.

Entwicklungsprozesse. Die Qualität eines Programms wird von vielen Faktoren bestimmt. Einflussgrößen sind zu Beginn der Entwicklung unbekannt oder nicht kontrollierbar. Erfahrungen gibt es erst, wenn das Programm existiert.

Traditionell wird Software nach dem *Wasserfallmodell* entwickelt, in dem die üblichen Entwicklungsschritte (Analyse, Design, Implementierung, Verifikation) einer nach dem anderen ausgeführt werden. Solche Entwicklungsprozesse eignen sich gut für kleine Projekte mit klaren Anforderungen. Aber das Risiko ist groß, dass etwas Unbrauchbares herauskommt, weil es erst ganz am Ende Feedback gibt. Daher werden für größere Projekte eher *zyklische Prozesse* verwendet, die die einzelnen Schritte ständig auf jeweils einem anderen Teil der gesamten Aufgabe wiederholen. So ergibt sich schon recht früh Feedback. Aber der Fortschritt eines Projekts ist nur schwer planbar. Es kann leicht passieren, dass sich die Programmqualität zwar ständig verbessert, das Programm aber nie zum Einsatz gelangt, da die Mittel vorher schon erschöpft sind. Zyklische Prozesse verkraften Anforderungsänderungen besser, aber Zeit und Kosten sind schwerer planbar, auch wenn es heute Berechnungsmodelle dafür gibt.

Erfahrung. In den vergangenen Jahrzehnten hat sich in der Programmierung ein umfangreicher Erfahrungsschatz darüber entwickelt, mit welchen Problemen in Zukunft gerechnet werden muss, wenn eine Aufgabe auf eine bestimmte Art gelöst wird. Mit dem dafür nötigen Wissen setzen wir Erfahrungen gezielt ein. Eine Garantie für den Erfolg eines Projekts gibt es natürlich trotzdem nicht, aber die Wahrscheinlichkeit für einen Erfolg steigt.

In der objektorientierten Programmierung ist der gezielte Einsatz von Erfahrungen essenziell. Es gibt viele unterschiedliche Möglichkeiten zur Lösung von Aufgaben, jede mit anderen Eigenschaften. Nur mit Erfahrung wird jene Möglichkeit gewählt, deren Eigenschaften später am ehesten hilfreich sind. Mit wenig Erfahrung wird nur die Möglichkeit gewählt, die zufällig zuerst in den Sinn kommt und damit auf einen wichtigen Vorteil verzichtet.

Zusammenhalt und Kopplung. Beim Entwickeln von Software müssen wir in *jeder* Phase wissen, wie wir vorgehen, um ein möglichst hochwertiges Ergebnis zu produzieren. Vor allem eine gute Faktorisierung ist entscheidend, aber leider erst gegen Ende der Entwicklung beurteilbar. Es gibt Faustregeln, die uns beim Finden guter Faktorisierungen unterstützen. Wir wollen hier einige Faustregeln betrachten, die in vielen Fällen einen Weg zu guter Faktorisierung weisen [5]:

Verantwortlichkeiten (Responsibilities): Die Verantwortlichkeiten einer Klasse können wir durch drei w-Ausdrücke beschreiben:

- „was ich weiß“ – Beschreibung des Zustands der Objekte
- „was ich mache“ – Verhalten der Objekte
- „wen ich kenne“ – sichtbare Objekte, Klassen, etc.

Das Ich steht dabei jeweils für die Klasse. Wenn etwas geändert werden soll, das in den Verantwortlichkeiten einer Klasse liegt, dann sind dafür die Personen zuständig, die die Klasse entwickelt haben.

Klassen-Zusammenhalt (Class-Cohesion): Darunter verstehen wir den *Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse.* Dieser Grad ist zwar nicht einfach messbar, aber intuitiv einfach fassbar. Der Zusammenhalt ist hoch, wenn alle Variablen und Methoden eng zusammenarbeiten und durch deren Namen und den Klassennamen gut beschrieben sind. Einer Klasse mit hohem Zusammenhalt fehlt Wichtiges, wenn Variablen oder Methoden entfernt werden. Der Zusammenhalt wird niedriger, wenn Klasse, Variablen oder Methoden sinnändernd umbenannt werden.

Objekt-Kopplung (Object-Coupling): Das ist die *Abhängigkeit der Objekte voneinander.* Die Objekt-Kopplung ist stark, wenn

- viele Methoden und Variablen nach außen sichtbar sind,
- im laufenden System Nachrichten und Variablenzugriffe zwischen unterschiedlichen Objekten häufig auftreten
- und die Anzahl der Parameter dieser Methoden groß ist.

Der Klassen-Zusammenhalt soll hoch sein. Ein hoher Klassen-Zusammenhalt deutet auf eine gute Zerlegung des Programms in einzelne Klassen beziehungsweise Objekte hin – gute Faktorisierung. Bei guter Faktorisierung ist die Wahrscheinlichkeit kleiner, dass bei Programmänderungen auch die Zerlegung in Klassen und Objekte geändert werden muss (*Refaktorisierung*, *Refactoring*).

Natürlich ist es bei hohem Zusammenhalt schwierig, bei Refaktorisierungen den Zusammenhalt beizubehalten oder noch weiter zu erhöhen.

Die Objekt-Kopplung soll schwach sein. Schwache Kopplung deutet darauf hin, dass Objekte weitgehend unabhängig voneinander sind. Dadurch beeinflussen Programmänderungen wahrscheinlich weniger Objekte unnötig. Beeinflussungen durch unvermeidbare Abhängigkeiten sind unumgänglich.

Klassen-Zusammenhalt und Objekt-Kopplung stehen in enger Beziehung zueinander. Wenn der Klassen-Zusammenhalt hoch ist, dann ist oft, aber nicht immer, die Objekt-Kopplung schwach und umgekehrt. Das ergibt sich nicht automatisch, sondern ist das Ergebnis eines bewussten Umgangs mit diesen Kriterien. Da Menschen auch dann gut im Assoziieren zusammengehöriger Dinge sind, wenn sie Details noch gar nicht kennen, ist es relativ leicht, bereits in einer frühen Entwicklungsphase zu erkennen, wie ein hoher Klassen-Zusammenhalt und eine schwache Objekt-Kopplung erreichbar sein könnte. Die Simulation der realen Welt hilft dabei vor allem zu Beginn der Softwareentwicklung.

Wenn eine Entscheidung zwischen Alternativen getroffen werden muss, können Klassen-Zusammenhalt und Objekt-Kopplung Beiträge zur Entscheidungsfindung liefern. Sowohl Klassen-Zusammenhalt als auch Objekt-Kopplung lassen sich im direkten Vergleich einigermaßen sicher prognostizieren. In manchen Fällen können jedoch andere Faktoren ausschlaggebend sein.

Auch mit viel Erfahrung können wir kaum auf Anhieb einen optimalen Entwurf liefern. Normalerweise muss die Faktorisierung einige Male geändert werden; wir sprechen von *Refaktorisierung*. Sie ändert die Struktur eines Programms, lässt aber dessen Funktionalität unverändert. Es wird dabei also nichts hinzugefügt oder weggelassen, und es werden auch keine inhaltlichen Änderungen vorgenommen. Solche Refaktorisierungen sind in einer frühen Projektphase ohne größere Probleme und Kosten möglich und werden durch eine Reihe von Werkzeugen unterstützt. Einige wenige Refaktorisierungen führen meist rasch zu einer stabilen Zerlegung der betroffenen Programmteile, die später kaum noch refaktorisiert werden müssen. Wir müssen nicht von Anfang an einen optimalen Entwurf haben, sondern nur alle nötigen Refaktorisierungen durchführen, bevor sich Probleme über das ganze Programm ausbreiten. Natürlich darf nicht so häufig refaktorisiert werden, dass bei der inhaltlichen Programmentwicklung kein Fortschritt mehr erkennbar ist. Ein vernünftiges Maß rechtzeitiger Refaktorisierungen führt häufig zu gut faktorisierten Programmen.

2.2.4 Worum es geht

Im Gegensatz zu jener der prozeduralen Programmierung ist die Werkzeugkiste der objektorientierten Programmierung gut gefüllt, wie die Länge obiger Beschreibungen zeigt. Erst im komplexen Zusammenspiel entfalten die Werkzeuge ihre Wirkung. Diese Wirkung haben wir in Abschnitt 1.5.2 angerissen: Wir arbeiten mit Abstraktionen auf hohem Niveau, die uns ein Verständnis des Programms und seiner Teile ermöglichen, ohne alle Details des Programmablaufs verstehen zu müssen. Das Ziel ist fast konträr zu dem der prozeduralen Programmierung. Ebenso verhält es sich mit dem Haupteinsatzgebiet: Die ob-

jektorientierte Programmierung eignet sich für die Entwicklung und Wartung langlebiger, komplexer Systeme durch Profis. Es wäre unsinnig, überschaubar kleine oder für den einmaligen Gebrauch bestimmte Programme objektorientiert zu entwickeln, weil dies den Aufwand ohne Gegenleistung in die Höhe treiben würde. Wir müssen leider auch davon ausgehen, dass in der Programmierung noch wenig erfahrene Leute von der objektorientierten Programmierung anfangs überfordert sind, weil das Zusammenspiel der Werkzeuge komplex ist. Die ernsthaft betriebene objektorientierte Programmierung ist sehr fordernd, setzt viel Erfahrung voraus und ist aufwändiger als andere Paradigmen. Andererseits bietet uns die objektorientierte Programmierung auch dann noch vertretbare Erfolgsaussichten, wenn Lösungsversuche in allen anderen Paradigmen aufgrund der Größe und Komplexität der Aufgabe fehlzuschlagen drohen.

Der Schlüssel zur objektorientierten Programmierung liegt in der Abstraktion (siehe Abschnitt 1.4), vor allem der Abstraktion auf der Ebene der Programmierung im Groben. Es geht um Modularisierungseinheiten, vorwiegend Objekte, die als abstrakte Datentypen gesehen werden. Wir stellen uns Objekte als Software-Abbilder von (nicht nur materiellen) Dingen aus der realen Welt vor und übertragen uns bekannte Eigenschaften dieser Dinge auf Software-Objekte. Natürlich ist jedes Abbild nur eine unvollständige Simulation des realen Dings und übernimmt nicht alle Eigenschaften. Hier kommt die Notwendigkeit von Erfahrung ins Spiel: Es muss ein Verständnis dafür entwickelt werden, welche Eigenschaften in der Software abgebildet werden und welche nicht. Außerdem müssen die erwartbaren Eigenschaften auf eine Weise beschrieben werden, sodass alle an der Entwicklung beteiligten Personen diese auf annähernd gleiche Weise verstehen. Es braucht eine gemeinsame Sprache. Es müssen auch Details auf gleiche Weise verstanden werden, auf die es in natürlichen Gesprächen in der Regel nicht ankommt. Beispielsweise verwenden wir Zusicherungen entsprechend Design-by-Contract, um Eigenschaften in einer mehr oder weniger standardisierten Art auszudrücken. Im Wesentlichen entwickelt sich die gemeinsame Sprache aber erst in der Zusammenarbeit in einem Team und unterliegt ständigen Weiterentwicklungen, weil auch Design-by-Contract große Interpretationsspielräume lässt. Darüber hinaus ergeben sich Abstraktionen auf immer höherem Niveau: Nicht nur ein Ding aus der realen Welt wird in Software simuliert, sondern auch Software-Objekte selbst oder nur in unserer Vorstellungswelt existierende Objekte werden für die Programmierung als reale Dinge angesehen und durch Objekte auf einer höheren Ebene simuliert. Mit der entsprechenden Erfahrung wirkt der Umgang mit derart abstrakten Objekten ganz einfach und natürlich, aber für Außenstehende ist es schwer, die für ein Verständnis nötigen Beziehungen zwischen den Begriffen zu entwickeln.

In der objektorientierten Programmierung leben wir in einer Phantasiewelt. Allerdings gibt es reale Bezüge zur echten Welt. Sie kommen nicht nur von den simulierten Dingen, sondern auch aus Einschränkungen, die uns die Werkzeuge auferlegen. Beispielsweise müssen wir ständig auf Ersetzbarkeit achten, ein Konzept mit klaren, aus der Theorie zweifelsfrei ableitbaren Einschränkungen. Die Theorie (Ersetzbarkeitsprinzip) müssen wir ebenso wie die Einschränkungen kennen und die Abstraktionen konform dazu entwickeln. Nachdem die Werkzeu-

ge kein Verständnis für unsere nominalen Abstraktionen haben, müssen wir uns selbst um die Einhaltung der wesentlichen Einschränkungen kümmern; Werkzeuge können nur wenige Eigenschaften automatisch prüfen. Solche notwendigen Bezüge zur echten Welt erschweren die Programmierung deutlich.

Menschen schaffen es oft schon nach kurzer Zeit, Objekte als Abstraktionen von Dingen aus der realen Welt zu sehen und häufig verwendete Abstraktionen aus vorgegebenen Klassenbibliotheken (trotz hohen Abstraktionsgrads) gut zu verstehen. Es dauert deutlich länger, bis Menschen sich klar genug in der gemeinsamen Sprache ausdrücken können und ihre Feinheiten verstehen. Meist dauert es auch sehr lange, sich an die vorgegebenen Einschränkungen zu gewöhnen. Das hat leider gravierende Konsequenzen: Werden Einschränkungen an nur einer Stelle im Programm ignoriert oder Eigenschaften falsch verstanden, kann das gesamte Gefüge wie ein Kartenhaus in sich zusammenbrechen und auch an ganz anderen Programmstellen zu Fehlern führen. Es ist also gefährlich, wenig erfahrene Personen unkontrolliert an größeren Projekten mitarbeiten zu lassen. Aber auch erfahrene Personen können unkonzentriert sein und etwas übersehen. Daher haben sich Vorgehensweisen wie *Pair-Programming* entwickelt, wobei jeder einzelne Schritt durch eine zweite Person begleitet und überwacht wird. Großer Wert liegt auf Teambesprechungen, Schulungen, Code-Reviews, etc., um vermeidbare Fehler zu unterbinden. Die objektorientierte Programmierung erfordert einen hohen Personaleinsatz und ist keinesfalls billig.

Die objektorientierte Programmierung hat einen großen evolutionären Wandel durchlebt. Bis in die 1990er-Jahre war wenig darüber bekannt, was Ersetzbarkeit genau bedeutet. Jede aus der realen Welt ableitbare „Is-a“-Beziehung wurde als mögliche Ersetzung missverstanden. Programmiert wurde auf einer intuitiven Ebene. Oft traten unerwartete Fehler auf, die pragmatisch beseitigt wurden, ohne deren Ursachen zu verstehen. Obwohl stark typisierte Sprachen zur Verfügung standen, wurden dynamische Sprachen bevorzugt, weil sie intuitiv einfacher und flexibler schienen. Im Hobby-Bereich war diese Art der objektorientierten Programmierung populär. Mit dem Jahrtausendwechsel trat ein Umdenkprozess ein, weil inhärente Widersprüche, die zu den unerwarteten Fehlern führten, als Verletzungen des Ersetzbarkeitsprinzips erkannt waren und nicht mehr ignoriert werden konnten. Vererbung wurde großteils verbannt und durch Untertypbeziehungen auf Basis des Ersetzbarkeitsprinzips ersetzt. Das erhöhte die Anforderungen an Programmierer_innen deutlich, was eine Professionalisierung zur Folge hatte. Es wurde versucht, durch Begriffe wie „Duck-Typing“ und mit Slogans den früheren objektorientierten Stil als eigenständiges Paradigma aufrecht zu erhalten, aber ohne Ersetzbarkeit (durchbrochene Werkzeugkette) sind viele Abstraktionen unwirksam und das Paradigma damit auf die prozedurale Programmierung zurückgeführt. Im Hobby-Bereich herrscht heute wieder die prozedurale Programmierung vor, auch wenn objektorientierte Sprachen und einige objektorientierte Werkzeuge eingesetzt werden. Objektorientierte Programmierung im heutigen Sinn ist Profis vorbehalten, die meist stark typisierte Sprachen bevorzugen und den Umgang mit allen Werkzeugen in der Werkzeugkiste beherrschen, nicht nur einigen wenigen. Profis sind teuer. Daher gehen aktuelle Trends in Richtung einer immer stärker werdenden Ein-

beziehung anderer Paradigmen in die objektorientierte Programmierung. Profis geben zwar die Architektur der zu entwickelnden Software vor und kümmern sich um kritische Bereiche, aber wo immer es geht, werden Programmteile ausgelagert und im prozeduralen oder funktionalen Paradigma erstellt.

2.3 Funktionale Programmierung

In der funktionalen Programmierung werden Funktionen als Daten verwendet. Diese einfache Eigenschaft ist schon fast die Definition des Paradigmas. Aber nur fast. Funktionen als Daten ergeben Möglichkeiten, die genutzt werden wollen. Das hat Konsequenzen, etwa jene, dass auf destruktive Zuweisungen verzichtet wird. So ergibt sich eine logische Konsequenz aus der anderen. Am Ende stehen funktionale Programmierstile, die mit den prozeduralen Programmierstilen, aus denen sie hervorgegangen sind, nur mehr wenig zu tun haben.

2.3.1 Sauberkeit aus Prinzip

Im Mittelpunkt der imperativen Programmierung steht die *destruktive Zuweisung,* also die Zuweisung eines neuen Werts an eine Variable, wobei der alte Wert der Variablen verloren geht. Vertreter „sauberer“ Programmierstile sehen jede solche Zuweisung als zerstörerischen Akt, etwas Unerwünschtes, Schmutziges, was durch Begriffe wie „Seiteneffekt“ und „Nebenwirkung“ zum Ausdruck kommt. Ein Ziel der funktionalen Programmierung liegt in der Zurückdrängung und schließlich Eliminierung jeder Art zerstörerischer Akte.

Unter einem *Programmzustand* verstehen wir die Gesamtheit der in allen Variablen enthaltenen Werte, unter einer *Zustandsänderung* die Änderung dieser Werte. Eine destruktive Zuweisung bewirkt eine Zustandsänderung, durch die etwas, was wir vorher über den Programmzustand gewusst haben, nachher nicht mehr gilt. Diese zeitlich begrenzte Gültigkeit unseres Wissens ist das, was an der destruktiven Zuweisung stört. Wir möchten, dass Wissen, das wir erworben haben, von Dauer ist. Aus formalen Modellen wissen wir, dass das geht. Etwa aus verschiedenen Varianten der monotonen Logik können wir Aussagen ableiten, die gültig bleiben; darauf können wir die logikorientierte Programmierung aufbauen. Wir konzentrieren uns in diesem Abschnitt jedoch auf die funktionale Programmierung (eingebettet in Java), die auf Modellen wie dem λ-Kalkül aufbaut. Durch das wiederholte Reduzieren von Ausdrücken sammeln wir immer mehr Wissen, müssen aber gesammeltes Wissen nie für ungültig erklären, bis der maximal reduzierte Ausdruck, die Normalform erreicht ist. Alle Informationen waren zwar auch schon im ursprünglichen (unreduzierten) Ausdruck enthalten, aber für uns nicht leicht zugänglich. Reduktionen bringen die Information in eine besser fassbare Form. Funktionsaufrufe stellen Reduktionen dar, viele weitere Sprachelemente sind leicht auf Reduktionen zurückführbar.

In reinen Formen der funktionalen Programmierung verzichten wir auf destruktive Zuweisungen. Das heißt nicht, dass wir auf jede Form der Zuweisung verzichten, weil die erstmalige Zuweisung eines Werts an eine Variable nicht

destruktiv und daher problemlos ist. Statt den Wert einer schon zuvor initialisierten Variablen zu verändern, führen wir eine neue Variable für einen neuen Wert ein; eine Variable ist damit nur mehr ein Name für einen bestimmten Wert. Am einfachsten geht das durch die Parameterübergabe bei einem Funktionsaufruf: Jeder Parameter fungiert als neuer, bisher nicht verwendeter Name, der für das Argument steht. Innerhalb einer Funktion ist es meist nicht schwer, jedem berechneten Teilergebnis einen neuen Namen zu geben (entspricht dem Ablegen in einer neuen lokalen Variable). Würden wir jedoch eine Schleife verwenden, hätten wir ein Problem: Eine Zuweisung in einem Schleifenrumpf bezieht sich in jeder Iteration auf die gleiche Variable und ist daher destruktiv. Dieses Problem lässt sich nur durch gänzlichen Verzicht auf Schleifen lösen. Das ist keine übermäßig große Einschränkung, weil jede Schleife durch Rekursion, also durch Funktionsaufrufe ersetzbar ist. Insgesamt bieten Funktionen den einfachsten und natürlichsten Weg zur Vermeidung destruktiver Zuweisungen.

Zur Illustration betrachten wir ein Java-Beispiel in einem einfachen funktionalen Stil. Inhaltlich ist es an das Beispiel zur prozeduralen Programmierung in Abschnitt 2.1 angelehnt:

```
import java.io.*;
import java.util.Scanner;

public class FunctionalCourse {
    public static void main(String[] args) throws IOException {
        if (args.length != 4)
            throw new IllegalArgumentException("wrong params");
        System.out.println(status(
            scan(new PointsF(2), args[3],
                 scan(new PointsF(1), args[2],
                      scan(new PointsF(0), args[1],
                           scan(new StudsF(), args[0], null))))));
    }
    private static Points scan(Func f, String file, Points p)
                throws IOException {
        return f.apply(new Scanner(new FileInputStream(file)), p);
    }
    private static String status(Points p) {
        if (p == null) return "";
        return p.stud.regNo+", "+p.stud.name+" ("+p.stud.curr+"): "
                + p.pnt[0]+"+"+p.pnt[1]+"+"+p.pnt[2]+" = "
                + (p.pnt[0]+p.pnt[1]+p.pnt[2])+"\n"+status(p.next);
    }
}

interface Func {
    Points apply(Scanner in, Points p);
}
```

```
class PointsF implements Func {
    final int part;
    public PointsF(int part) {
        this.part = part;
    }
    public Points apply(Scanner in, Points p) {
        if (!in.hasNextInt()) return p;
        final int regNo = in.nextInt();
        final int pnt = in.nextInt();
        if (pnt <= 0 || pnt > 100)
            throw new RuntimeException("inappropriate points");
        return apply(in, modify(p, regNo, pnt));
    }
    private Points modify(Points p, int regNo, int pnt) {
        if (p == null) throw new RuntimeException("unknown regNo");
        if (p.stud.regNo == regNo)
            return new Points(p.stud, add(p.pnt, pnt), p.next);
        return new Points(p.stud, p.pnt, modify(p.next, regNo, pnt));
    }
    private int[] add(int[] old, int pnt) {
        if (part == 0 && old[0] == 0)
            return new int[]{pnt, old[1], old[2]};
        if (part == 1 && old[1] == 0)
            return new int[]{old[0], pnt, old[2]};
        if (old[2] == 0)
            return new int[]{old[0], old[1], pnt};
        throw new RuntimeException("contradictory");
    }
}
class StudsF implements Func {
    public Points apply(Scanner in, Points p) {
        if (!in.hasNextInt()) return p;
        final int regNo = in.nextInt();
        if (exists(regNo, p))
            throw new RuntimeException("double regNo");;
        final int curr = in.nextInt();
        final String name = in.nextLine().strip();
        final Stud stud = new Stud(regNo, curr, name);
        return apply(in, new Points(stud, new int[3], p));
    }
    private boolean exists(int regNo, Points p) {
        if (p == null) return false;
        if (p.stud.regNo == regNo) return true;
        return exists(regNo, p.next);
    }
}
```

```
class Points {
    public final Stud stud;
    public final int[] pnt;
    public final Points next;
    public Points(Stud stud, int[] pnt, Points next) {
        this.stud = stud;
        this.pnt = pnt;
        this.next = next;
    }
}
class Stud {
    public final int regNo, curr;
    public final String name;
    public Stud(int regNo, int curr, String name) {
        this.regNo = regNo;
        this.curr = curr;
        this.name = name;
    }
}
```

Im Gegensatz zum Beispiel in Abschnitt 2.1 verwenden wir hier eher lineare Listen statt Arrays. Arrays sind in der funktionalen Programmierung weniger wertvoll, weil Arrayeinträge nicht destruktiv verändert werden dürfen. Im Beispiel werden Arrays, statt sie zu ändern, stets neu erzeugt. Beim Einfügen wird die Liste mit Knoten vom Typ `Points` zum Teil immer wieder neu aufgebaut. Im funktionalen Stil ist es einfach, in jedem Punkte-Eintrag eine direkte Referenz auf die entsprechenden Studierendendaten zu halten, die beim Neuaufbau unverändert übernommen werden. Viele Variablen wurden als `final` deklariert; bei formalen Parametern wurde aus Platzgründen darauf verzichtet, obwohl auch die Parameter so verwendet werden, als ob sie `final` wären. Das ist eine typische Eigenschaft von funktionalen Programmen in Java. Häufig kommen tief verschachtelte Ausdrücke vor, die durch die Schachtelung helfen, die Anzahl der nötigen Variablen klein zu halten. Die Klassen `Points` und `Stud` enthalten zwar `public` Variablen und deren Instanzen sind eher Records als Objekte im Sinne der objektorientierten Programmierung, aber dennoch haben die Records andere Eigenschaften als in der prozeduralen Programmierung, weil Variablen nach der Initialisierung nur lesbar sind. Funktionen werden nicht direkt als Daten verwendet, aber als Instanzen von `Func` sind über dieses Interface beschliebene Methoden fast wie Funktionen als Daten verwendbar. Objekte von `Func` enthalten neben `apply` auch als lokal zu betrachtende Hilfsfunktionen und Objektvariablen, die Werte aus den Umgebungen der Funktionen enthalten. Wie zu erwarten benötigt das funktionale Programm mehr Methoden und Parameter als das prozedurale. Im Gegenzug werden keine Schleifen benötigt. Trotz dieser Unterschiede bestehen viele Gemeinsamkeiten mit dem prozeduralen Beispiel aus Abschnitt 2.1. Insbesondere bilden beide Programme die Ausführung recht klar über den Kontrollfluss ab, der Datenfluss spielt keine große Rolle.

Der in obigem Beispiel dargestellte Stil bildet nur einen ersten Schritt in Richtung funktionaler Programmierung. Tatsächlich sind wir hauptsächlich mit Sprachkonzepten der prozeduralen Programmierung in Java ausgekommen und haben kaum Funktionen als Daten verwendet. Die gesamte Ein- und Ausgabe erfolgt über prozedurale Seiteneffekte und verlangt in dieser Form nach viel Kontrolle. Java ist keine funktionale Sprache und unterstützt die funktionale Programmierung daher nur unzureichend. Dennoch hat diese einfache Form der funktionalen Programmierung schon einige wichtige Eigenschaften:

- Auf destruktive Zuweisungen wird verzichtet. Daher gibt es (außer bei der Parameterübergabe) keine Möglichkeit, dass Programmteile über gemeinsame Variablen miteinander kommunizieren. Ohne Funktionen als Daten bestimmt nur der Kontrollfluss den Programmablauf.

- Gleiche Werte (auch Instanzen von Klassen) bleiben immer gleich, die Gleichheit geht nicht durch destruktive Zuweisungen verloren. Diese Eigenschaft bildet die Basis für *referentielle Transparenz:* Wenn wir Objekte ausschließlich über `equals` (oder `compareTo` und ähnliche Methoden) vergleichen, nicht mittels `==`, ist der Unterschied zwischen Gleichheit und Identität aufgehoben. Es gibt keinen Unterschied mehr zwischen einem Original und dessen Kopie.

- Trotz der Verwendung von Rekursion sind Programme strukturell einfacher, weil keine Schleifen verwendet werden.

Einige weitere Eigenschaften einfacher funktionaler Programme kommen nur zum Tragen, wenn speziell für die funktionale Programmierung ausgelegte Sprachen zum Einsatz kommen. Java-Programme können nicht davon profitieren:

- In Java wirkt sich tiefe Rekursion negativ auf den Speicherverbrauch für den Stack der Methodenaufrufe aus. Funktionale Sprachen verwenden Techniken wie „Tail-End-Recursion-Optimization“, um die Stackbelastung klein zu halten und Funktionsaufrufe effizienter zu machen. In vielen Fällen wird fast die gleiche Effizienz wie durch Schleifen erreicht. Einige Optimierungen setzen die semantische Einfachheit funktionaler Programme voraus und sind nicht direkt auf imperative Sprachen übertragbar.

- Die strukturelle Einfachheit funktionaler Programme kann in der Syntax der Sprache genutzt werden, um Programme kompakter, mit kürzeren Programmtexten darzustellen. In obigem Java-Programm ist das nicht zu erkennen. Das kommt daher, dass wir dem Programm in einer Sprache, die für die objektorientierte Programmierung ausgelegt ist, einen fremden Stil aufgezwungen haben. Alleine schon die vielen Verwendungen der in einer funktionalen Sprache unnötigen Wörter `private`, `public`, `static` und `final` erhöhen den Schreibaufwand deutlich.

- Ist durchgehend referentielle Transparenz gegeben, können Vergleichsmethoden wie `equals` leicht automatisch generiert werden, was den Programmieraufwand reduziert.

- Referentielle Transparenz ermöglicht einen effizienten Umgang mit Konstanten. Häufig wird in funktionalen Sprachen *Pattern-Matching* eingesetzt. Dabei erfolgt, ähnlich wie in einer `switch`-Anweisung, eine Verzweigung abhängig von vorhandenen Daten, wobei aber auch komplexe Datenmuster herangezogen werden, nicht nur Zahlen und Zeichenketten. Prinzipiell wäre Pattern-Matching auch in der imperativen Programmierung sinnvoll, entfaltet aber erst zusammen mit referentieller Transparenz das volle Potential zur Verringerung der Programmkomplexität.

- Funktionale Programme eignen sich bestens für die Typinferenz, bei der ein Compiler statische Typen aus dem Programm errechnet und Typkonsistenz garantiert, ganz ohne Typdeklarationen im Programm. Das ermöglicht kurze und zugleich statisch typsichere Programme. In objektorientierten Sprachen ist uneingeschränkte Typinferenz an Stellen, an denen Ersetzbarkeit gegeben sein muss (also an allen wichtigen Stellen), prinzipiell unmöglich. Im Umkehrschluss gilt auch, dass funktionale Sprachen mit Typinferenz keine Ersetzbarkeit garantieren können.

- Einfache lineare Datenstrukturen, etwa Listen, sind in der funktionalen Programmierung besonders wichtig. Häufig werden auch komplexe Listenoperationen durch die Syntax der Sprache direkt unterstützt.

2.3.2 Streben nach höherer Ordnung

Auch ohne wesentliche Verwendung von Funktionen als Daten hat die funktionale Programmierung schon interessante Eigenschaften. Funktionale Programmierstile ändern sich deutlich, wenn Funktionen häufig als Daten verwendet werden. Funktionen, die Funktionen als Parameter nehmen bzw. als Ergebnisse zurückgeben, heißen *Funktionen höherer Ordnung* oder *funktionale Formen.* Sie dienen in gewisser Weise als Ersatz für Kontrollstrukturen. Werden Funktionen höherer Ordnung ernsthaft eingesetzt, dann nicht nur an einzelnen Programmstellen, sondern meist in großem Stil und fast überall. Der Einsatz von Funktionen höherer Ordnung reduziert den sichtbaren Bedarf an Rekursion deutlich, im (nicht so gut sichtbaren) Hintergrund ist Rekursion nach wie vor wesentlich.

Der auf Funktionen höherer Ordnung aufbauende funktionale Programmierstil heißt auch *applikative Programmierung* um deutlich zu machen, dass es nicht darauf ankommt, für jede Kleinigkeit eine eigene Funktion zu schreiben, sondern darauf, bestehende (großteils vorgefertigte und mit dem System ausgelieferte) Funktionen geschickt anzuwenden und miteinander zu kombinieren, um eine Aufgabe zu lösen. Es ergibt sich eine Steigerung der Produktivität der Programmierer_innen, weil auch komplexe Algorithmen in vergleichsweise kurzer Zeit ausführbar gemacht werden können. Im Prinzip kann zwar jede erdenkliche Funktion höherer Ordnung entwickelt werden. Letztendlich kommt es aber darauf an, eine überschaubar kleine Menge an Funktionen höherer Ordnung in hoher Qualität zur Verfügung zu haben, die in Kombination zur Lösung der meisten Aufgaben ausreicht. Sollte ausnahmsweise einmal keine passende Funktion zur Verfügung stehen, können wir sie problemlos hinzufügen.

Java unterstützt seit Version 8 die applikative Programmierung mit speziell dafür eingeführten Spracherweiterungen, insbesondere den Java-8-Streams und Lambda-Ausdrücken (kurz Lambdas) sowie vielen Erweiterungen vordefinierter Klassen. Hier ist das schon bekannte Beispiel in einem applikativen Stil:

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class ApplicativeCourse {
    public static void main(String[] args) {
        if (args.length != 4)
            throw new IllegalArgumentException("wrong params");
        List<Stream<String>> ins = Stream.of(args).map(file -> {
              try {return Files.lines(Paths.get(file));}
              catch(IOException x) {throw new RuntimeException(x);}
            }).collect(Collectors.toList());
        Map<Integer, Stud> studs = ins.get(0)
            .collect(HashMap::new, (smap, line) -> {
                Scanner sc = new Scanner(line);
                int regNo = sc.nextInt();
                Stud stud = new Stud(regNo, sc.nextInt(),
                                     sc.nextLine().strip());
                if (smap.put(regNo, stud) != null)
                    throw new RuntimeException("double regNo");
            }, HashMap::putAll);
        List<Map<Integer, Integer>> lm = ins.stream().skip(1)
            .map(in -> in.collect(HashMap::new,
                   (Map<Integer, Integer> pmap, String line) -> {
                Scanner sc = new Scanner(line);
                int regNo = sc.nextInt();
                int points = sc.nextInt();
                if (points <= 0 || points > 100)
                    throw new RuntimeException("inappropriate points");
                if (studs.get(regNo) == null)
                    throw new RuntimeException("unknown regNo");
                if (pmap.put(regNo, points) != null)
                    throw new RuntimeException("contradictory");
            }, Map::putAll))
            .collect(Collectors.toList());
        System.out.println(
                studs.values().stream().map(s -> line(s, lm))
                        .collect(Collectors.joining("\n")));
    }
```

```
    private static String line(Stud s,
                               List<Map<Integer, Integer>> lm) {
        List<Integer> ps = lm.stream()
                .map(m -> m.getOrDefault(s.regNo, 0))
                .collect(Collectors.toList());
        return s.regNo + ", " + s.name + " (" + s.curr + "): "
                + ps.stream().map(i -> i.toString())
                  .collect(Collectors.joining("+"))
                + " = " + ps.stream().reduce(0, Integer::sum);
    }
}

class Stud {
    public final int regNo, curr;
    public final String name;

    public Stud(int regNo, int curr, String name) {
        this.regNo = regNo;
        this.curr = curr;
        this.name = name;
    }
}
```

Es fällt trotz ausladender Java-Syntax auf, dass das Programm kürzer ist, mit weniger Funktionen auskommt und nirgends sichtbar Rekursion verwendet. Außerdem werden keine eigenen Listen angelegt, sondern vorgefertigte Container-Klassen zusammen mit Generizität eingesetzt. Das sind (bis auf die Java-Syntax) typische Eigenschaften der applikativen Programmierung.

Das Beispiel verwendet Lambdas, die so ähnlich wie λ-Abstraktionen einsetzbar sind. Bei genauerer Betrachtung sind Lambdas Objekte, die nur eine einzige Methode enthalten. Sehen wir uns `s -> line(s, lm)` aus obigem Programmtext an: Um diesen Ausdruck besser verstehen zu können, gehen wir davon aus, dass `s` vom Typ `Stud` ist. Damit ist das Lambda einfach lesbar: `s` ist der einzige Parameter einer Funktion, die den Teil nach `->` durch einen Aufruf von `line(s, lm)` berechnet und das Ergebnis des Aufrufs zurückgibt, wobei `lm` eine weiter oben initialisierte Variable ist. Der Ausdruck entspricht dieser Methode:

```
String apply(Stud s) { return line(s, lm); }
```

Genau genommen wird an die Methode `map`, an die das Lambda als Argument übergeben wird, ein neues Objekt übergeben, das nur diese eine Methode enthält. `map` hat (nach Auflösung der Generizität) folgende Signatur:

```
Stream<String> map(Function<Stud, String> mapper)
```

Dabei ist `Function<Stud, String>` (nach Auflösung der Generizität) ein Interface, das nur eine abstrakte Methode beschreibt:

```
String apply(Stud t);
```

Lambdas können nur dort verwendet werden, wo das entsprechende Interface (etwa als Parametertyp oder bei zurückgegebenen Lambdas der Ergebnistyp) genau eine abstrakte Methode ohne Default-Implementierung enthält; solche Interfaces heißen *funktionale Interfaces*. Das Lambda entspricht der Implementierung dieser einen Methode, von dieser Methodensignatur kommt auch der Methodenname. Innerhalb von `map` kann das übergebene Lambda ganz normal wie jede andere Methode aufgerufen werden: `mapper.apply(...)` wobei die Punkte für das Argument vom Typ `Stud` stehen. Im Rumpf des Lambdas können Variablen aus der Umgebung vorkommen, etwa `lm`. Allerdings gibt es die Einschränkung, dass solche Variablen als `final` deklariert sein müssen, oder nur so verwendet werden, als ob sie `final` wären. Damit wird verhindert, dass auf undurchschaubare oder gefährliche Weise über Variablen kommuniziert wird.

Bei anderen Lambdas im Beispiel verhält es sich ähnlich, wobei jedoch mehrere syntaktisch verschiedene Varianten vorkommen. Links von `->` kann statt eines Namens eine Liste von Namen in runden Klammern stehen; dabei entspricht jeder Name einem Parameter. Damit können Lambdas beliebig viele Parameter haben. Zwecks besserer Lesbarkeit können auch Typen der Parameter dabei stehen. In einigen Fällen sind Typen bei solchen Parametern sogar notwendig, wenn der Compiler in der lokalen Umgebung nicht ausreichend Information findet, um die Typen durch Typinferenz berechnen zu können. Wie jede objektorientierte Sprache kann Java ja keine uneingeschränkte Typinferenz anbieten, sondern nur eine auf die lokale Umgebung eingeschränkte Form. Rechts von `->` können auch Anweisungen und Methodenrümpfe (mehrere Anweisungen in geschwungenen Klammern) stehen, nicht nur Ausdrücke zur Berechnung von Rückgabewerten. Im Beispiel werden mehrere solche Lambdas verwendet, auch solche mit umfangreichen Rümpfen.

Nicht jede als Argument oder Ergebnis verwendete Methode muss in oben beschriebener Lambda-Syntax definiert sein, es sind auch schon existierende Methoden verwendbar. So ist `HashMap::putAll` die Methode `putAll` aus der Klasse `HashMap`. Genauer: Als Argument wird ein neues Objekt übergeben, dessen einzige Methode sich genau so wie `putAll` aus `HashMap` verhält. Dementsprechend steht `HashMap::new` für ein Objekt, dessen einzige Methode ein neues Objekt von `HashMap` erzeugt. Parameterzahl und -typen der Methode oder des Konstruktors hängen vom funktionalen Interface ab.

Java-8-Streams stellen eine strukturierte Sammlung von Funktionen höherer Ordnung dar, die vor allem für die Abarbeitung großer Datenmengen konzipiert sind. Wie der Name sagt, entspricht jedes Objekt von `Stream` einem linearen Strom an gleichartigen Daten, wobei der Typ der Daten über Generizität festgelegt wird. Bei Datenströmen (kurz Strömen) der Arten `IntStream`, `LongStream` und `DoubleStream` sind Daten vom jeweiligen primitiven Typ. Die Abarbeitung erfolgt in drei Stufen: Zuerst wird ein Strom in einer Operation *erzeugt*, danach werden die Daten im Strom über beliebig viele *modifizierende Operationen* gefiltert oder geändert und am Ende steht eine *abschließende Operation*, die Daten meist in einer anderen Form außerhalb des Stroms verfügbar macht.

Im Beispiel wird die Variable `ins` mit einer vierelementigen Liste von Stömen initialisiert, die jeweils die Zeilen in den vier in den Kommandozeilenargumenten `args` genannten Dateien enthalten. `Stream.of(args)` erzeugt einen Strom, der die vier Dateinamen in `args` als Zeichenketten enthält. Die darauf angewandte Methode `map` wendet auf jede Zeichenkette ein Lambda an, das für jeden Dateinamen einen Strom an in der Datei enthaltenen Zeilen erzeugt. Das Lambda bewerkstelligt das durch Anwendung der vordefinierten Methode `Files.lines` auf je einen Dateipfad, der mittels `Paths.get(file)` aus dem Dateinamen `file` erzeugt wird. Wie bei jeder Ein- oder Ausgabe kann dabei eine `IOException` auftreten, die über einen `try-catch`-Block abgefangen werden muss. Ein Weiterreichen dieser überprüften Ausnahme nach außen ist durch das Lambda nicht möglich, weil die Signatur keine `throws`-Klausel enthält – ein Hinweis auf eine andere Form der Abstraktion als in der objektorientierten Programmierung. Die Ausnahme wird in eine unüberprüfte Ausnahme vom Typ `RuntimeException` umgewandelt, die weitergereicht werden kann. Das Ergebnis der modifizierenden Operation `map` ist also ein Strom, der vier Ströme enthält. Der äußere Strom wird durch die abschließende Operation `collect(Collectors.toList())` abgeschlossen, wobei die Inhalte des Stroms in einer Liste in der gleichen Reihenfolge wie im Strom abgelegt werden. Das Argument von `collect` ist ein Objekt, das für die passende Umwandlung zuständig ist. In `Collectors` sind Methoden definiert, die solche Objekte zurückgeben. Das von `toList()` zurückgegebene Objekt erzeugt eine Liste (hier vom Typ `List<Stream<String>>`), ohne Details über die Implementierung der Liste bekanntzugeben.

Die Ströme in `ins` werden auf zwei verschiedene Weisen weiterverwendet; der erste Strom enthält Studierendendaten, die weiteren Ströme Informationen zu „Punkten“ für drei verschiedene Lehrveranstaltungsteile. Der erste dieser Ströme wird direkt durch `collect` angewandt auf `ins.get(0)` abgeschlossen. Hier wird jedoch eine Variante von `collect` mit drei Parametern verwendet, über die wir das Erzeugen der resultierenden Datenstruktur genauer steuern können. Der erste Parameter wird zur Erzeugung einer Datenstruktur verwendet, der zweite zum Einfügen eines Datenelements aus dem Strom und der dritte zum Zusammenfassen mehrerer Datenstrukturen zu einer. Der dritte Parameter wird für den Fall benötigt, dass der Datenstrom parallel abgearbeitet wird, wobei mehrere gleichartige Datenstrukturen erzeugt und am Ende zusammengefügt werden. Im Beispiel wird über den ersten Parameter `HashMap::new` eine neue Hashtabelle erzeugt und über den dritten `HashMap::putAll` zwei Hashtabellen zusammengefügt. Das zweite Argument ist ein Lambda mit zwei Parametern, der Hashtabelle, in die eingefügt werden soll, sowie die zu bearbeitende Zeile aus dem Strom. Über einen Scanner werden die Daten aus der Zeile gelesen und das entsprechende neue Objekt von `Stud` in die Hashtabelle eingefügt, wobei es zu einer Ausnahme kommt, wenn schon Daten für die gleiche Matrikelnummer (`regNo`) vorhanden sind. Zur Bearbeitung der restlichen Ströme wird ein neuer Strom über die Einträge in `ins` erzeugt, wobei `skip(1)` gleich einen Eintrag, den ersten, der schon abgearbeitet ist, aus diesem Strom entfernt. Die restlichen drei Ströme im Strom werden auf ähnliche Weise wie der erste Strom behandelt: Aus jedem Strom werden Daten mit Punktebewertungen gelesen (jeweils eine

Matrikelnummer mit einer Punktezahl) und nach diversen Prüfungen Hashtabellen erzeugt, die jeweils Matrikelnummern auf Punktezahlen abbilden. Durch `collect(Collectors.toList())` wird eine Liste der Hashtabellen erzeugt. Alle Ströme in `ins` sind damit abgearbeitet.

Schließlich wird aus den Daten über Studierende in `studs` und Daten über gesammelte Punkte in `lm` die Ausgabe generiert. Dazu wird ein Strom über den Werten in `studs` (die Schlüssel werden nicht mehr benötigt) erzeugt, `map` wandelt, wie oben angesprochen, durch Aufrufe von `line` jedes Objekt von `Stud` im Strom in eine Ausgabezeile um und `collect` schließt den Strom ab. Die Methode `joining` aus `Collectors` ist nur auf einen Strom mit Zeichenketten anwendbar und sorgt dafür, dass alle Zeichenketten zu einer einzigen zusammengefügt werden, wobei die als Argument übergebene Zeichenkette `"\n"` als Trennsymbol zwischen je zwei Zeichenketten gesetzt wird. Aus dem Strom an Ausgabezeilen entsteht so der gesamte auszugebende Text.

Auch die Methode `line` ist über Ströme implementiert. Zunächst werden in der Liste `ps` die pro Lehrveranstaltungsteil gesammelten Punkte für den/die Studierende_n `s` abgelegt, wobei ein Strom über `lm` pro Lehrveranstaltungsteil eine Map enthält, ein Aufruf der modifizierenden Operation `map` durch `getOrDefault` jeweils die Punkteanzahl für `s` aus der Map holt (oder 0 falls keine Punkte für `s` eingetragen sind) und `collect` den resultierenden Strom an ganzen Zahlen abschließt. Die Ausgabezeile wird aus allgemeinen Daten in `s`, den einzelnen Werten in `ps` und der Summe der Werte in `ps` zusammengesetzt. Die Werte in `ps` werden dabei über einen Strom über `ps` erfasst, wobei über `map` zunächst die Zahlen in Zeichenketten umgewandelt werden, die danach über `joining` ähnlich wie oben beschrieben zusammengesetzt werden. Die Summe wird ebenfalls aus einem Stream über `ps` gebildet. Dieser Strom wird jedoch nicht über `collect` abgeschlossen, sondern über `reduce`, eine Methode, die alle Werte im Stream zu einem einzigen Wert reduziert. Diese Methode nimmt als ersten Parameter den Anfangswert, also das Ergebnis, das bei einem leeren Strom zurückkommen soll, in unserem Fall 0. Der zweite Parameter ist ein Lambda, das den bisher reduzierten Wert und den nächsten Wert im Strom zu einem neuen Wert reduziert. Hier wird dafür `Integer::sum` verwendet, eine Funktion, die einfach nur zwei ganze Zahlen addiert; stattdessen hätten wir gleichbedeutend das Lambda `(x,y) -> x + y` verwenden können. Das Ergebnis ist also einfach die Summe aller Zahlen in `ps`.

Java-8-Streams verwenden Lazy-Evaluation (siehe Abschnitt 1.5.1). Das heißt, Daten in einem Strom werden erst berechnet, wenn eine abschließende Operation diese Daten benötigt. In diesem Beispiel verwenden die abschließenden Operationen `collect` und `reduce` alle verfügbaren Daten. Dadurch wirkt sich Lazy-Evaluation semantisch kaum aus. Jedoch muss im Beispiel `ins` nie wirklich alle Zeilen aus allen eingelesenen Dateien enthalten. Daten werden erst aus den Dateien gelesen, wenn diese Daten verarbeitet werden, nach Abarbeitung können sie durch Speicherbereinigung entfernt werden. Lazy-Evaluation kann also tatsächlich die Effizienz steigern, in diesem Fall den Speicherverbrauch senken.

Wir haben gesehen, dass sich die applikative Programmierung in Java aus dem Zusammenspiel einiger nicht-trivialer Sprachkonzepte ergibt. Funktionen

höherer Ordnung in Java-8-Streams und Collections, Lambdas, funktionale Interfaces und Generizität spielen hinein. Ein Verständnis dieses Zusammenspiels ist unerlässlich, der applikative Programmierstil in Java daher eher für Menschen mit Programmiererfahrung geeignet. Entsprechend erfahrene Menschen können applikative Programme trotz geforderter hoher Qualitätsstandards in deutlich kürzerer Zeit schreiben als in allen anderen Programmierstilen. Leider sind applikative Programme auch von erfahrenen Menschen nicht so gut lesbar wie viele Programme in anderen erfolgreichen Stilen, was die Wartung auf lange Sicht erschwert. Die Effizienz beim Schreiben kommt unter anderem von der Typinferenz, die das Hinschreiben vieler Typen erspart, aber die Lesbarkeit vermindert. Noch wichtiger ist der höhere Grad an Abstraktion, weil hinter jeder Funktion höherer Ordnung komplexe Denkmuster stehen, die den genauen Programmablauf teilweise verschleiern. Die Idee hinter einem Lösungsansatz bleibt im fertigen Programm oft verborgen.

2.3.3 Friedliche Koexistenz

Wir wollen nun untersuchen, welche Rolle die funktionale Programmierung, insbesondere in der applikativen Form, in der Praxis spielen kann. Fassen wir einige diesbezügliche Beobachtungen zusammen:

- Die Programmerstellung durch erfahrene Leute kann in der funktionalen Programmierung sehr effizient sein. Der *ICFP Programming Contest,* bei dem es darum geht, eine algorithmisch komplexe Aufgabe möglichst rasch und effizient zu lösen, wurde häufig (aber nicht immer) von einem Team gewonnen, das eine funktionale Sprache in einem applikativen Stil verwendete. Das ist nicht überraschend, wenn man bedenkt, dass ICFP die „International Conference on Functional Programming“ ist. Aber alleine die Existenz dieses Bewerbs zeigt deutlich, wo die Stärken der funktionalen Programmierung vermutet werden: rasche Entwicklung algorithmisch komplexer Aufgaben, aber nicht deren langfristige Wartung,

- Der Schwerpunkt der funktionalen Programmierung liegt in der Programmierung im Feinen, nicht in der Programmorganisation. Damit steht die funktionale Programmierung in Konkurrenz zur prozeduralen Programmierung, aber kaum zur objektorientierten Programmierung.

- Die funktionale Programmierung bietet nicht so gute Kontrollmöglichkeiten von Details des Programmablaufs wie die prozedurale Programmierung, dafür aber einen viel höheren Abstraktionsgrad, der die Programmiereffizienz verbessert. Die funktionale Programmierung eignet sich daher kaum in Bereichen, in denen es auf bestmögliche Kontrolle ankommt (hardwarenahe Programmierung und Echtzeitprogrammierung).

- Eine für die parallele Programmierung vorteilhafte Strukturierung von Daten ist oft auch gut für die funktionale Programmierung geeignet, jedoch nicht für die objektorientierte. In Teilbereichen der parallelen Programmierung ist die funktionale Programmierung stark vertreten.

- Bei einigen Personengruppen im Bereich der Hobby- und Anwendungsprogrammierung ist, auch historisch bedingt, viel Wissen über funktionale Programmierung vorhanden. Es gibt offensichtlich keinen Widerspruch dazu, dass die effiziente funktionale Programmierung Erfahrung voraussetzt. Auch in der Hobby-Programmierung kann viel Erfahrung vorhanden sein, wenn auch häufig eingeschränkt auf enge Themenbereiche.

Die funktionale Programmierung in Java und ähnlichen Sprachen erfreut sich immer größerer Beliebtheit, was sich anhand obiger Beobachtungen nur unvollständig erklären lässt. Die Anzahl an Programmen, in denen die funktionale Programmierung ihre Vorteile voll ausspielen kann, ist ja begrenzt. Aber es geht gar nicht darum, ganze Programme in einem funktionalen Stil zu schreiben. Vielmehr gibt es in großen Programmen häufig Teile, die sich perfekt für einen funktionalen Programmierstil eignen würden, obwohl sich andere Programmteile nicht dafür eignen. Das heißt, die funktionale Programmierung hat vor allem in Kooperation mit anderen Programmierparadigmen viel Potential. Die Kombination mehrerer Paradigmen ist aber schwierig, weil die grundlegenden Annahmen, Vorgehensweisen und Werkzeuge nicht zusammenpassen. Referentielle Transparenz verträgt sich nicht mit imperativen Veränderungen des Programmzustands.

Trotz aller Widersprüche ist bekannt, wie die funktionale Programmierung in Kombination mit anderen Paradigmen einsetzbar ist. Da die objektorientierte Programmierung sich auf die Programmierung im Groben bezieht, die funktionale und prozedurale Programmierung aber auf die Programmierung im Feinen, scheint es oberflächlich betrachtet einfach zu sein, den prozeduralen Anteil an der objektorientierten Programmierung durch einen funktionalen Anteil zu ersetzen. So einfach geht das aber nicht, weil es rein auf Basis der funktionalen Programmierung keine veränderlichen Objektzustände geben könnte, die in der Praxis wesentlich sind. Eine sinnvolle Kombination setzt sowohl prozedurale als auch funktionale Anteile voraus. Das ist möglich, indem Prozeduren Funktionen (entsprechend der funktionalen Programmierung) aufrufen, aber Funktionen keine Prozeduren aufrufen und keinen Zugriff auf veränderbare Variablen bekommen. Daten fließen über Parameter von den Prozeduren zu den Funktionen und als Funktionsergebnisse zu den Prozeduren. So bleiben charakteristische Eigenschaften der Paradigmen erhalten, aber das Zusammenspiel der Programmteile in unterschiedlichen Paradigmen ist eingeschränkt. Die größte praktische Schwierigkeit besteht darin, an der Grenze zwischen prozeduralem und funktionalem Teil die Art des Denkens beim Programmieren umzustellen.

Auch in der rein funktionalen Programmierung stehen wir vor dem Dilemma, dass Ein- und Ausgabe notwendigerweise Seiteneffekte darstellen. In funktionalen Sprachen wie Haskell gibt es eine elegante Lösung: Die Ein- und Ausgabe wird in *Monaden* verpackt, das sind mathematische bzw. rein funktionale Gebilde, welche die eigentlichen Inhalte in Form von Funktionen kapseln und vom Rest des Systems abtrennen. Monaden bewirken, dass die darin enthaltenen Funktionen erst bei Vorliegen von Eingabedaten in Ausgabedaten abgebildet werden, unabhängig vom Ablauf des restlichen Programms. Das klingt kompli-

ziert und ist es auch, wenn wir versuchen, den Programmablauf genau nachzuvollziehen. Wegen des hohen Abstraktionsgrads durch referentielle Transparenz müssen wir den Ablauf aber nicht verfolgen, um das Programm zu verstehen. Im Endeffekt ergibt sich nämlich das, was wir oben beschrieben haben: Die gesamte Ein- und Ausgabe erfolgt in einem eher prozeduralen Programmteil, die Prozeduren rufen reine Funktionen auf, aber diese Funktionen rufen niemals Prozeduren auf; veränderbare Variablen sowie die gesamte Ein- und Ausgabe sind den reinen Funktionen (durch das Typsystem geprüft) nicht zugänglich.

Andere funktionale Sprachen wie OCAML sorgen direkt, ohne Umweg über Monaden, für eine Trennung zwischen Prozeduren mit Seiteneffekten und reinen Funktionen, wobei referentielle Transparenz auf Funktionen gewahrt bleibt. Das „O“ in OCAML steht für „Object“, es wird also ein objektorientierter Programmierstil mit Seiteneffekten unterstützt. Das funktioniert auf herkömmliche Weise: Methoden verändern den Zustand von Objekten durch destruktive Zuweisungen, aber Methoden können auch reine Funktionen aufrufen.

Lambdas in Java funktionieren nur zum Teil nach diesem Schema: Es ist zwar nicht erlaubt, innerhalb von Lambdas auf veränderbare Variablen aus der Umgebung direkt zuzugreifen, aber Methoden, die als Prozeduren zu betrachten sind, können aufgerufen werden. Durch den Aufruf von Prozeduren verlieren wir referentielle Transparenz und damit eine Form der funktionalen Abstraktion. Das heißt, wir müssen die Ausführungsreihenfolge nachvollziehen, um das Programm zu verstehen. Die Ausführungsreihenfolge ist die Reihenfolge, in der Methoden tatsächlich ausgeführt werden, nicht nur der logische Zusammenhang zwischen Ausdrücken. Bei Lazy-Evaluation wie in Java-8-Streams kann sich die Ausführungsreihenfolge erheblich von den logischen Zusammenhängen unterscheiden. Betrachten wir dazu ein Beispiel:

```
import java.util.stream.Stream;
public class Elem {
    private static int num = 0;
    private int id;
    private Elem() { System.out.print((id = num++)); }
    public static Stream<Elem> stream() {
        return Stream.generate(Elem::new);
    }
    public String toString() { return "-" + id; }
}
```

Objekte von `Elem` enthalten eine Variable `id` mit jeweils unterschiedlichen Zahlen. Die Methode `Elem.stream()` erzeugt einen unbegrenzten Strom von `Elem`-Objekten mit aufsteigenden Zahlenwerten. Da Java-8-Streams auf Lazy-Evaluation beruhen, stellt die unbeschränkte Strom-Länge kein Problem dar. Zur Demonstration des Ablaufs wird bei der Objekterzeugung die jeweilige Zahl ausgegeben, ebenso wie durch `toString` (nach einem Trennsymbol). Folgender Aufruf gibt 10 Zeilen aus, jede von der Form `0-0`, `1-1`, `2-2` und so weiter:

```
Elem.stream().limit(10).forEach(System.out::println);
```

Die Operation `limit` auf dem Strom lässt die gegebene Anzahl an Elementen durch und schließt den Strom danach. Die abschließende Operation `forEach` führt auf jedem Element das übergebene Lambda, in diesem Fall `println` aus. Wer gewohnt ist, prozedural zu denken, würde erwarten, dass zuerst durch den Konstruktor die Zahlen `0` bis `9` ausgegeben werden und erst danach die Ausgaben durch `println` erfolgen. Aber so funktionieren Ströme nicht. Erst bei Aufruf von `println` wird das Strom-Element benötigt und erst dann wird es (lazy) erzeugt. Nur dadurch ist es möglich, zuerst mit einem unbeschränkten Strom zu arbeiten und diesen erst später durch `limit` zu begrenzen. Abschließende Operationen auf Strömen haben häufig Seiteneffekte wie das Ausgeben von Werten oder das Hinzufügen von Werten zu Datenstrukturen, wobei die Datenstrukturen selbst nicht funktional aufgebaut sind. Modifizierende Operationen[1] sind dagegen meist frei von Seiteneffekten.

Es stellt sich die Frage, ob die Programmierung mit Lambdas und Java-8-Streams überhaupt funktional ist. Im Vergleich zu aktuellen funktionalen Sprachen wie Haskell ist Java ganz anders. Es gibt allerdings auch Sprachen wie *Lisp,* der Sprache, in der die funktionale Programmierung entwickelt wurde, in der destruktive Zuweisungen zwar unerwünscht, aber möglich sind. Wie in jeder objektorientierten oder prozeduralen Sprache ist es auch in Java möglich, auf Seiteneffekte weitgehend zu verzichten und daher funktional zu programmieren. Lambdas können dabei hilfreich sein. Es ist auch möglich, einen objektorientierten, prozeduralen und funktionalen Stil zu mischen, wobei Prozeduren Funktionen aufrufen, aber nicht umgekehrt. Es bleibt fraglich, ob das auch bei Java-8-Streams möglich ist, wenn abschließende Operationen als prozedural zu betrachten sind. Die Antwort ist ein vorsichtiges „vielleicht ja“, wenn wir die Ausführungsreihenfolge betrachten: Abschließende Operationen stehen in gewisser Weise am Anfang, weil sie Ausführungen anderer Operationen im Strom erst anstoßen; für alle anderen Operationen im Strom ist Seiteneffektfreiheit wichtiger, weil nur schwer vorhersehbar ist, wann sie, wenn überhaupt, ausgeführt werden, sodass die Auswirkungen von Seiteneffekten kaum abschätzbar sind. In obigem Beispiel ist das `print` im Konstruktor sicher ein größeres Problem als das `println` in der abschließenden Operation. In sehr eingeschränkter Form, etwa `num++` im Beispiel, sind Seiteneffekte vertretbar, wenn ihr Effekt örtlich gut abgegrenzt bleibt. Natürlich dürfen wir im Beispiel nicht davon ausgehen, dass alle zurückgegebenen Zahlen fortlaufend sind, weil ja auch außerhalb des betrachteten Stroms Objekte von `Elem` erzeugt werden könnten.

Generell können wir sagen, dass die applikative Programmierung zwar wesentlich von der funktionalen Programmierung beeinflusst ist, aber nicht auf funktionale Programmierung beschränkt bleibt. Es gibt durchaus auch prozedurale Elemente und die objektorientierte Datenkapselung kommt zum Tragen. Trotzdem müssen wir in der applikativen Programmierung vorsichtig und sparsam mit Seiteneffekten umgehen, weil Effekte durch kaum vorhersehbare Programmabläufe andernfalls rasch undurchschaubar werden – siehe Kapitel 5.

[1] „Modifizierend“ bedeutet hier nicht, dass Variableninhalte destruktiv verändert werden. Modifizierende Operationen wie `map` verändern den Strom durch Einführung neuer Werte.

Prinzipiell ist die applikative Programmierung eine wertvolle Ergänzung zur objektorientierten Programmierung, gerade wegen ihrer Verschiedenartigkeit. Applikative Programmteile können innerhalb eines großen objektorientierten Programms klein genug bleiben, damit die Auswirkungen der erschwerten Lesbarkeit in Grenzen gehalten werden. Trotzdem können diese Teile groß genug sein, um die Effizienz der Programmerstellung zu verbessern. Selbstverständlich wird applikative Programmierung vorwiegend dort eingesetzt werden, wo eine gewisse algorithmische Komplexität gegeben ist, damit die Vorteile voll zum Tragen kommen; gerade für solche Aufgaben zeigt die objektorientierte Programmierung ohnehin Schwächen. Die Faktorisierung eines Programms erfolgt in dieser Kombination meist wie in der objektorientierten Programmierung, sodass applikative Programmteile nur Ergänzungen sind, die den Charakter der objektorientierten Programmierung kaum ändern.

2.4 Parallele Programmierung

Im Gegensatz zur prozeduralen, funktionalen und objektorientierten Programmierung liegt der Fokus in der parallelen Programmierung nicht auf einer bestimmten Sammlung an Werkzeugen, sondern in einer sehr konkreten Zielsetzung: Eine (meist auf einer großen Datenmenge beruhende) nicht-triviale Aufgabe soll unter Einbeziehung mehrerer oder vieler gleichzeitig arbeitender Recheneinheiten so schnell wie möglich gelöst werden. Zur Erreichung des Ziels ist ein gutes Verständnis der verschiedenen Formen der Parallelverarbeitung nötig, von der Ebene der Hardware über die Betriebssystem- und Programmiersprachunterstützung bis zu typischen Programmiertechniken. Insgesamt stehen durch die vielen Ausprägungen der Parallelität sehr viele Werkzeuge zur Verfügung, aber die Werkzeugkiste, die alle in einem bestimmten Projekt eingesetzten Werkzeuge enthält, bleibt kleiner. Das heißt, viele Werkzeuge kommen nicht gemeinsam, sondern alternativ zum Einsatz.

2.4.1 Parallelität und Ressourcenverbrauch

Bringen wir zunächst mit wenig Aufwand Parallelität in ein Programm. Im Beispielprogramm zur applikativen Programmierung in Abschnitt 2.3.2 ersetzen wir den Ausdruck `ins.stream()` durch `ins.parallelStream()` und Java sorgt dafür, dass unterschiedliche Stromabschnitte von mehreren Recheneinheiten gleichzeitig bearbeitet werden, konkret Dateien mit Bewertungsdaten gleichzeitig eingelesen werden. Java-8-Streams sind für die Parallelverarbeitung ausgelegt. Durch Ausprobieren werden wir vielleicht (abhängig von vielen Details) feststellen, dass das parallele Programm trotz höheren Ressourcenverbrauchs (Speicher- und Prozessorauslastung) langsamer läuft als das sequentielle. Das Ziel der parallelen Programmierung wird damit in vielen Fällen verfehlt.

Das bringt uns zu einem wesentlichen Grundsatz der Parallelprogrammierung: Die Einführung von Parallelität, gleichgültig in welcher Form, erhöht auf jeden Fall den Ressourcenverbrauch. Größerer Ressourcenverbrauch wird sich

häufig in längerer Laufzeit niederschlagen. Nur wenn es gelingt, ausreichend viel Rechenarbeit so effizient auf die vorhandenen Recheneinheiten zu verteilen, dass dies den zusätzlichen Ressourcenverbrauch kompensiert, wird die Laufzeit verkürzt. Es ist aber immer so, dass alle verwendeten Recheneinheiten zusammengenommen mehr zu tun haben als in einem sequentiellen Programm. Wir müssen deshalb ausreichend viel parallelisierbare Rechenarbeit finden, um die Recheneinheiten auszulasten und zugleich verstehen, woher der zusätzliche Ressourcenverbrauch kommt und wie wir ihn klein halten können.

Gehen wir zunächst zur Vereinfachung davon aus, dass wir auf einem Rechner mit einem Prozessor mit mehreren Prozessorkernen arbeiten. Das ist heute üblich, am Smartphone genauso wie am Laptop und PC. Jeder Prozessorkern hat eine kleine Menge eigener Register und kann eine eigene sequentielle Abfolge von Befehlen abarbeiten. Alle Prozessorkerne teilen sich einen gemeinsamen Speicher, auf den in der Regel über mehrere Ebenen an Cache zugegriffen wird. Parallelverarbeitung heißt in diesem Fall, dass wir jedem freien Prozessorkern Aufgaben zuteilen, die unabhängig von Aufgaben auf anderen Kernen bearbeitet werden können. Mit folgenden Schwierigkeiten müssen wir dabei umgehen:

- Es ist einiges zu tun, um einen Prozessorkern von einer Aufgabe auf eine andere Aufgabe umzuschalten. Die aktuellen Registerwerte müssen abgespeichert und neue Registerwerte geladen werden. Außerdem müssen Cache-Inhalte für ungültig erklärt und frisch aus dem langsameren Speicher geladen werden. Das dauert alles zusammen nicht besonders lange, aber lange genug, um ein Faktor zu sein, der zu berücksichtigen ist. Wir müssen dafür sorgen, dass die bearbeiteten Aufgaben groß genug sind, um die Anzahl der Umschaltvorgänge klein zu halten.

- Die Prozessorkerne greifen alle auf den gleichen Speicher zu. Trotzdem müssen wir dafür sorgen, dass unterschiedliche Aufgaben nur auf unterschiedliche Speicherbereiche zugreifen, da sich die Berechnungen andernfalls gegenseitig in die Quere kommen und keine vernünftigen Ergebnisse liefern. Wir sprechen von einer *Race-Condition,* wenn Ergebnisse von Berechnungen davon abhängen, wie schnell einzelne parallele Programmteile ausgeführt werden. Race-Conditions sind unbedingt zu vermeiden.

- Wenn Abhängigkeiten zwischen einzelnen Aufgaben bestehen, indem die eine Aufgabe von Daten abhängt, die von der anderen Aufgabe produziert werden, oder wenn zwei Aufgaben auf gleiche Speicherbereiche (nicht nur lesend) zugreifen, dann müssen diese Aufgaben miteinander *synchronisiert* werden. Jede Form der Synchronisation bedeutet einen Verlust an sinnvoller Parallelabarbeitung, weil die Aufgaben nicht mehr gleichzeitig ausführbar sind.[2] Wir müssen dafür sorgen, dass die Abhängigkeiten zwischen den Aufgaben so klein wie möglich bleiben.

[2] Es gibt *optimistische* Formen der Synchronisation, bei denen die Aufgaben so ausgeführt werden, als ob sie unabhängig wären. Erst nach den Berechnungen wird geprüft, ob es durch Abhängigkeiten zu Problemen gekommen ist. In diesem Fall müssen Ergebnisse verworfen und Berechnungen erneut durchgeführt werden. Bei diesen Formen der Synchronisation haben wir zwar Recheneinheiten ausgelastet, dabei aber mit einer bestimmten

- Oft haben wir einen Vorrat an Aufgaben, die mehr oder weniger unabhängig voneinander abarbeitbar sind. Wir brauchen einen Algorithmus, den *Scheduler*, der die einzelnen Aufgaben den verfügbaren Prozessorkernen zuordnet. Dabei müssen Synchronisationsbedingungen berücksichtigt werden. Die verwendete Scheduling-Strategie kann durch Vorziehen wichtiger Arbeiten einen großen Einfluss auf die Gesamteffizienz haben.

- Manchmal ist Synchronisation unvermeidbar. Schlecht gewählte Synchronisationsbedingungen und Scheduling-Strategien können (häufig in Kombination) dazu führen, dass sich der Fortschritt in den Berechnungen stark verlangsamt oder sogar ganz zum Erliegen kommt. Wir sprechen von *Liveness*-Problemen. Gefürchtet sind

  Starvation, wobei eine für den Berechnungsfortschritt wichtige Aufgabe nicht oder nicht in ausreichendem Umfang zur Ausführung gelangt (also „verhungert“), während andere, für den Fortschritt weniger wichtige Aufgaben zu viele Ressourcen bekommen.

  Deadlock, wobei zwei oder mehr Aufgaben sich durch Synchronisation gegenseitig blockieren, weil sie von anderen Aufgaben benötigte Ressourcen nicht freigeben und gleichzeitig auf die Freigabe von Ressourcen durch andere Aufgaben warten.

  Livelock, wobei mehrere Aufgaben zwar fleißig „Scheinarbeiten“ leisten, aber keinen inhaltlichen Fortschritt erzielen, häufig weil sie sich gegenseitig den Zugriff auf benötigte Ressourcen wegnehmen.

- Die größte Schwierigkeit besteht darin, ein Programm so in einzelne Aufgaben passender Größe zu zerlegen, dass diese Aufgaben nicht oder kaum voneinander abhängen. Für manche Programme ist das relativ einfach möglich, vor allem wenn viele Daten unabhängig voneinander verarbeitet werden sollen und jeder Bearbeitungsschritt hinreichend groß ist. Für andere Programme ist das praktisch unmöglich. Häufig ist nicht das ganze Programm gut parallelisierbar, sondern nur einzelne Teile davon bzw. einzelne hintereinander ablaufende Phasen. Das ist unproblematisch, solange der weitaus größte Berechnungsaufwand in den gut parallelisierbaren Teilen steckt. Es ist allerdings zu bedenken, dass die sequenziellen, also nicht parallelisierten Programmteile die Laufzeit im Wesentlichen bestimmen, weil während der sequenziellen Phasen nur einer der Prozessorkerne zum Programmfortschritt beitragen kann.

- Idealerweise werden alle Teile des Systems gleichmäßig gut ausgelastet. Meist ergibt sich jedoch irgendwo ein *Engpass* (*Bottleneck*). Manchmal gibt es nicht genug Aufgaben, um alle Prozessorkerne auszulasten, dann gibt es wieder zu viele davon, sodass deren Verwaltung einen größeren Aufwand nach sich zieht. Manchmal muss auf so viele Daten gleichzeitig

---

Wahrscheinlichkeit keine sinnvolle Arbeit geleistet, sodass Abhängigkeiten einen Verlust an „sinnvoller“ Parallelberechnung bedeuten.

zugegriffen werden, dass die Caches überlastet sind und die langsamen Zugriffe auf den Hauptspeicher zum Engpass werden, wodurch die Prozessorkerne viel Zeit mit Warten verbringen. Manchmal kann das Netzwerk oder die Festplatte nicht schnell genug Daten liefern oder übernehmen.

- Das von uns betrachtete Programm existiert nicht alleine auf dem Rechner. Wenn wir das Programm optimal parallelisieren, sodass es alle Teile des Systems bestmöglich auslastet, wird das System wahrscheinlich überlastet sein, wenn andere Programme unvorhergesehen gleichzeitig laufen.

- Wir dürfen nicht vergessen, dass der Umgang mit diesen Schwierigkeiten einen erheblichen Programmieraufwand verursacht. Der Aufwand für das Erstellen, Testen und Warten paralleler Programme kann um ein Vielfaches größer sein als für entsprechende sequenzielle Programme.

Im eingangs gebrachten Beispiel wird Parallelität durch `parallelStream` so genutzt, dass die betroffenen Daten nicht voneinander abhängen und für jedes Element im Strom relativ viel zu tun ist. Aber nur ein Teil der gesamten Aufgabe kann davon profitieren. Es kann trotz geschickter Auswahl sein, dass die parallel ablaufenden Aufgaben zu klein sind, um den Mehraufwand in der Verwaltung kompensieren zu können, oder die Eingabe wird zum Engpass. Abhängig von der Hardware und dem Umfang der Daten kann die Laufzeit kürzer oder länger sein als in der nicht-parallelen Variante.

Das Ziel der Parallelisierung ist immer die Verkürzung der Laufzeit. Solange die angestrebte Programmeffizienz durch geeignete Wahl der eingesetzten Algorithmen erreicht werden kann, ist die Parallelisierung nicht sinnvoll. Wenn ein Programm wegen nicht ausreichender Effizienz parallelisiert werden muss, ist die Auswahl der Algorithmen neu zu überdenken. Der für ein sequenzielles Programm ideale Algorithmus muss nicht auch für die parallele Variante ideal sein. Im parallelen Programm ist es wesentlich, eine Aufteilung in ausreichend viele unabhängige Aufgaben geeigneter Größe zu finden. Es kann vorteilhaft sein, gleiche Berechnungsschritte in den einzelnen Aufgaben zu wiederholen, wenn die Aufgaben dadurch unabhängig voneinander werden. Aber die Anzahl an Prozessorkernen ist beschränkt. Daher ist kein beliebig großer Mehraufwand in den Berechnungen durch Parallelisierung kompensierbar.

2.4.2 Formen der Parallelität

Ein Prozessor mit mehreren Kernen stellt nur eine von vielen Formen der Parallelität dar. Auf der Ebene der Hardware begegnen wir häufig folgenden Formen:

- Ein Prozessorkern arbeitet einen sequenziellen Strom an Maschinenbefehlen ab. Aktuelle Prozessorkerne können mehrere Befehle gleichzeitig ausführen. Bei einer *VLIW*-Architektur (*Very Long Instruction Word*) sorgt der Compiler für parallel ausführbare Befehlssequenzen. Bei einer *Superscalar*-Architektur analysiert die Hardware aufeinander folgende Befehle dahingehend, ob sie voneinander abhängen, also beispielsweise ein Befehl

Daten produziert, die ein anderer zu seiner Durchführung benötigt. Unabhängige Befehle können parallel ausgeführt werden. Wenn die Ausführung komplexerer Befehle mehrere Prozessorzyklen dauert, kann die Wartezeit mit der Ausführung nachfolgender Befehle gefüllt werden, solange die Befehle unabhängig sind. Teilweise werden Befehle sogar dann vorgezogen, wenn die Unabhängigkeit nicht im Vorhinein garantiert ist; wird später festgestellt, dass Abhängigkeiten zu Fehlern geführt haben, müssen die fehlerhaften Berechnungen wiederholt werden (*Spekulative Ausführung*). Schön daran ist, dass sich der Compiler (etwa durch Umreihung von Befehlen) und die Hardware ohne unser Zutun um diese Formen der Parallelausführung kümmern. Wir sprechen von *impliziter Parallelität.*

- Ein Prozessor mit mehreren Kernen ist eine Form von *MIMD*-Parallelität (Multiple-Instructions-Multiple-Data). Es gibt mehrere sequenzielle Ströme von Befehlen, die unabhängig voneinander auf unterschiedlichen Daten arbeiten. Mehrere Kerne auf einem Prozessor teilen sich in der Regel eine gemeinsame Schnittstelle zum Hauptspeicher und gemeinsame Caches. Diese gemeinsame Schnittstelle kann leicht zu einem Engpass werden, weshalb die Anzahl der Kerne pro Prozessor recht klein, häufig einstellig ist. Es ist auch möglich, mehrere Prozessoren (mit jeweils mehreren Kernen) zu einer Einheit zusammenzuschließen und auf einen gemeinsamen Speicher zugreifen zu lassen. Wir sprechen dann von *Symmetric Shared-Memory Multiprocessors (SMP)* oder *Uniform-Memory-Access (UMA).* Die Komplexität der Schnittstelle zum gemeinsamen Speicher steigt mit der Anzahl an Prozessoren rasch an, weil Vorkehrungen dagegen getroffen werden müssen, dass die Schnittstelle zum Engpass wird. Außerdem müssen die Caches auf den unterschiedlichen Prozessoren konsistent gehalten werden. Obwohl die in Abschnitt 2.4.1 genannten Schwierigkeiten gelöst werden müssen, ist die SMP-Programmierung eine der einfachsten Formen der parallelen Programmierung.

- Wir sprechen von *Distributed-Shared-Memory (DSM)* oder *Non-Uniform-Memory-Access (NUMA)* wenn jeder Prozessor seinen eigenen Speicher hat, aber alle Prozessoren und ihre Speicher über ein Verbindungsnetzwerk miteinander verbunden sind. Alle Speicher liegen in einem gemeinsamen Adressraum. Der wesentliche Unterschied zu SMP besteht darin, dass Speicher, der zum eigenen Prozessor gehört, effizienter zugreifbar ist als Speicher, der zu einem anderen gehört. Das ist in der Programmierung zusätzlich zu beachten. Programmausführungen können nicht ohne Effizienzverlust von einem Prozessor zu einem anderen verschoben werden.

- Wenn mehrere oder viele Rechner ohne gemeinsamen Adressraum für Speicherzugriffe eingesetzt werden, geht es meist um *verteilte Programmierung.* Die Kommunikation über ein vergleichsweise langsames Netzwerk ist dabei in der Regel ein Engpass, dar die parallele Programmierung erschwert. Kürzestmögliche Ausführungszeiten stehen für die meisten Programme auf Rechnernetzwerken nicht zentral im Fokus, eher das Verkraf-

ten einer hohen Last an Arbeitsaufträgen. Es wird also darauf geachtet, jeden Auftrag mit möglichst wenig Ressourcen zu bearbeiten, nicht darauf, alle vorhandenen Ressourcen zur Erreichung einer möglichst kurzen Antwortzeit einzusetzen. Für manche Problemstellungen mit sehr hohem Rechenaufwand bei gleichzeitig geringem Kommunikationsaufwand sind Rechnernetzwerke auch für die parallele Programmierung gut geeignet. In diesem Fall werden alle verfügbaren Ressourcen so eingesetzt, dass in möglichst kurzer Zeit ein Ergebnis erzielt wird, ohne auf einen sparsamen Umgang mit Ressourcen zu achten.

- Für Datenparallelität (viele Daten, die auf gleiche Art zu bearbeiten sind) eignen sich *SIMD*-Instruktionen (Single-Instruction-Multiple-Data), wobei eine einzige Instruktion gleichzeitig auf ein ganzes Array an Daten angewandt wird. Typischerweise wird eine Instruktion auf etwa 128 bis 512 oder mehr Bits gleichzeitig angewandt, was klarerweise effizienter ist, als sie nur auf 64 Bits anzuwenden. Die Bits sind zu Worten mit jeweils 8, 16 oder 32 Bits zusammengefasst. Um bedingte Anweisungen auszuführen, ist häufig wählbar, auf welchen Worten eine Instruktion wirken soll und auf welchen nicht. SIMD-Instruktionen kommen in Vektoreinheiten vieler üblicher Prozessoren, vor allem aber in GPUs (Graphikprozessoren) zum Einsatz, weil sie sich ideal zum Bearbeiten von Bilddaten eignen. Eine GPU ist zwar langsamer getaktet als eine CPU (Hauptprozessor), enthält aber viel mehr Prozessorkerne, von denen jeder einfacher als in der CPU ist, aber SIMD-Anweisungen ausführt. Die Bandbreite für Speicherzugriffe muss für eine GPU deutlich größer sein als für eine CPU, damit die vielen parallelen Einheiten versorgt werden können. Während CPUs dafür ausgelegt sind, jeden Befehl so rasch wie möglich abzuschließen (kurze und häufig unsichtbare Pipelines) und Wartezeiten durch Vorziehen nachfolgender Befehle nach Möglichkeit zu füllen, sind Pipelines auf GPUs meist länger und sichtbar, das heißt, die durch Ausführung eines Befehls erzeugten Daten stehen erst einige Zyklen später für die Weiterverarbeitung zur Verfügung. In der Zwischenzeit müssen davon unabhängige Befehle ausgeführt werden, in jedem Zyklus einer. Manche GPUs sind als billige, aber dennoch recht leistungsfähige Parallelrechner einsetzbar. Bei anderen GPUs ist das schwierig, weil viele Kerne auf ganz bestimmte, nur für die Bildverarbeitung notwendige Aufgaben spezialisiert sind.

Unter einem *Prozess*[3] verstehen wir die Ausführung eines Programms auf einem Rechner gesteuert durch ein Betriebssystem. Häufig wird jede parallel zu

[3] Das ist ein vielfach überladener Begriff. Wenn wir umgangssprachliche Bedeutungen und jene in anderen technischen Bereichen und im Wirtschafts- und Rechtswesen ausklammern, bleibt noch die Bedeutung im Softwareengineering, wo ein Prozess die Vorgehensweise zur Steuerung der Softwareentwicklung von der Konzeption bis zum Ende des Software-Lebenszykluses ist. Sogar im Sinn einer Ausführung eines Programmstücks ist der Begriff überladen. Wir verwenden *Prozess* nur zur Bezeichnung der Ausführung eines Programms durch ein Betriebssystem, nicht die Ausführung eines Ausführungsstrangs (*Thread*) innerhalb eines Programms. Diverse vor allem theoretische Modelle (etwa die in Abschnitt 1.1.1 erwähnten Prozesskalküle) verwenden diesen Begriff auch für Ausführungsstränge.

lösende Aufgabe in Form eines Programms beschrieben, natürlich viele Aufgaben durch das gleiche Programm, jedoch auf jeweils andere Daten angewandt. Das Betriebssystem startet für jede Aufgabe einen eigenen Prozess. Jeder Prozess bekommt eine eigene Ablaufumgebung, die alle für die Ausführung nötigen Ressourcen bereitstellt und den Prozess gleichzeitig vor unerwünschten Einflüssen durch andere Prozesse abschirmt. Insbesondere ist kein Zugriff auf Speichersegmente anderer Prozesse möglich, da jeder Prozess nur seinen eigenen *virtuellen Speicher* sieht, der durch eine *Memory-Management-Unit (MMU)* im Hintergrund auf den von allen Prozessen gemeinsam verwendeten *physikalischen Speicher* abgebildet wird. Virtuelle und physikalische Speicheradressen unterscheiden sich voneinander. Die MMU sorgt dafür, dass jeder Prozess immer eine in sich konsistente Sichtweise auf seinen virtuellen Speicher bekommt, obwohl im Hintergrund möglicherweise bei Speichermangel ganze Seiten vom Hauptspeicher in einen externen Speicher ausgelagert und später wieder an einer anderen physikalischen Adresse aus dem externen Speicher geladen werden. Die starke Abschottung der Prozesse voneinander hat auch Nachteile, weil oft mehrere Prozesse auf gemeinsame Daten im Speicher zugreifen oder Daten von einem Prozess zum anderen weiterreichen müssen, voneinander abhängige Prozesse aufeinander warten und diesbezügliche Information austauschen müssen und die gemeinsame Nutzung von Systemressourcen (etwa Ein- und Ausgabe) koordiniert werden muss. Wir sprechen von *Interprozesskommunikation,* wenn es um den Austausch von Daten zwischen Prozessen geht. Dafür kommen einige Verfahren häufig zum Einsatz:

- Am bekanntesten ist der Datenaustausch über das Schreiben und Lesen von Dateien. Die Synchronisation kann z. B. durch das Anlegen und Löschen von Dateien erfolgen. Betriebssysteme bieten zur Vereinfachung auch spezialisierte Konzepte (etwa *Locks*) zur Synchronisation an.

- Dateien werden (nicht nur in Java) über *Ein- und Ausgabe-Ströme* (nicht zu verwechseln mit Java-8-Streams) gelesen und geschrieben. Es spielt dafür keine Rolle, woher ein Strom kommt oder wohin er geht. Eine *Pipeline* nutzt das aus: Das Betriebssystem verknüpft einen Ausgabe-Strom mit einem Eingabe-Strom, sodass die Daten ohne Umweg über eine Datei weitergereicht werden. Synchronisiert wird dadurch, dass beim Lesen so lange gewartet wird, bis die Daten auf der anderen Seite geschrieben wurden. In Linux können Pipelines bei Programmaufrufen sehr einfach angelegt werden. Z. B. startet `a|b` neue Prozesse für die Programme `a` und `b` gleichzeitig und verbindet die Standardausgabe von `a` mit der Standardeingabe von `b`. Neue Prozesse sind auch von einem bestehenden Prozess aus erzeugbar und dabei können Verbindungen über Pipelines aufgebaut werden. Aber über anonyme Pipelines ist es nicht möglich, im Nachhinein Verbindungen zwischen schon bestehenden Prozessen aufzubauen. Für diesen Zweck gibt es *benannte Pipelines,* die wie Dateien angesprochen und geöffnet werden können.

- Etwas komplexer ist die Kommunikation über *Sockets.* Erzeugt und geöff-

net werden Sockets, also die Endpunkte der Kommunikationsverbindungen, durch Aufrufe entsprechender Betriebssystemfunktionen. Auch zum Lesen und Schreiben stehen Betriebssystemfunktionen bereit. Sockets sind vor allem dann geeignet, wenn die darüber kommunizierenden Prozesse auf verschiedenen Rechnern liegen, es also keinen gemeinsamen physikalischen Speicher gibt. Häufig werden Sockets zusammen mit TCP, das ist ein relativ zuverlässiges Netzwerkprotokoll, sowie IP zur Adressierung im Internet eingesetzt; für manche Aufgaben ist aber UDP, ein einfaches, effizientes Netzwerkprotokoll besser geeignet. Im Bereich des Internets ist die Verwendung von Sockets heute praktisch unumgänglich.

- Auf einem Rechner mit gemeinsamem Speicher ist *Shared-Memory* die wahrscheinlich effizienteste Form der Interprozesskommunikation. Wie der Name schon sagt, bekommen dabei mehrere Prozesse Zugang zum gleichen (physikalischen) Speicherbereich. Ein Prozess reserviert durch Aufruf einer entsprechenden Betriebssystemfunktion Shared-Memory, andere Prozesse können danach Zugriff auf dieses Shared-Memory nehmen. Ein guter Teil der Arbeit wird von der MMU erledigt, wodurch der Mechanismus im Detail jedoch stark vom Betriebssystem und der Hardware abhängt und wenig portabel ist. Shared-Memory sorgt nur für den gemeinsamen Speicher, nicht für die Synchronisation der Zugriffe darauf. Dafür müssen andere Mechanismen, etwa Semaphore eingesetzt werden.

Viele Programmiersprachen unterstützen eigene Mechanismen zur Parallelausführung, die bei der Erzeugung und Verwaltung weniger Aufwand verursachen als Prozesse. Java bietet *Threads* an. Im Gegensatz zu einem Prozess hat ein Thread keinen eigenen virtuellen Speicher, sondern greift auf den gleichen Speicher zu wie das restliche Programm. Threads sind nicht voneinander abgeschirmt. Daher sind auch keine Mechanismen zur Interprozesskommunikation nötig. Kommuniziert wird über den gemeinsamen Speicher im gemeinsamen Adressraum. Besonderer Wert muss auf die Synchronisation gelegt werden, weil natürliche Formen der Synchronisation wie durch Dateien, Pipelines oder Sockets für Threads nicht zur Verfügung stehen. Dennoch sind Dateien, Pipelines und Sockets auch in Java verfügbar, weil die Ausführung eines Java-Programms durch einen Java-Interpreter auch ein Prozess ist, der möglicherweise mit anderen Prozessen kommunizieren muss. Threads bieten zusätzliche Parallelausführungen innerhalb von Prozessen, ersetzen Prozesse aber nicht.

Der elementarste Synchronisationsmechanismus ist das *Semaphor*. Das ist im Wesentlichen eine nicht-negative ganzzahlige Variable s, auf der es nur zwei Operationen gibt, die historisch P und V heißen (holländische Kürzel), in Programmiersprachen und Betriebssystemen aber verschiedene Namen haben können. $P(s)$ verringert den Wert von s um 1, wenn s dabei nicht negativ wird, andernfalls wird der Aufrufer von $P(s)$ (ein Prozess oder Thread) suspendiert. $V(s)$ weckt einen suspendierten Prozess oder Thread wieder auf, falls ein solcher existiert, andernfalls wird der Wert von s um 1 erhöht. Einsatzzwecke sind vielfältig, am bekanntesten ist der Einsatz zur Erreichung von *Mutual-Exclusion*,

also um zu gewährleisten, dass ein bestimmter Programmtext, der *kritische Abschnitt,* niemals von zwei Prozessen oder Threads gleichzeitig ausgeführt wird. Bei Erreichen des kritischen Abschnitts wird $P(s)$ ausgeführt, an dessen Ende $V(s)$, wobei s zuvor mit 1 initialisiert wurde. So müssen stets alle außer einem Prozess oder Thread am Eingang des kritischen Abschnitts warten, bis der eine Prozess oder Thread diesen wieder verlässt. Jede komplexere Form der Synchronisation lässt sich durch ein Semaphor oder das Zusammenspiel mehrerer Semaphore beschreiben. Java verwendet für die Synchronisation von Threads vorwiegend das Monitor-Konzept, das in Abschnitt 2.5 beschrieben ist.

Von großer praktischer Bedeutung ist die verwendete Scheduling-Strategie. Jedes Betriebssystem hat eine solche Strategie für Prozesse, meist mit der Möglichkeit durch Ändern von Prioritäten darauf Einfluss zu nehmen, wie viel Rechenzeit dem Prozess zugestanden werden soll. Ähnliches gibt es auch für Threads in Programmiersprachen. Da Betriebssysteme und Programmiersprachen praktisch nichts über die Charakteristika der Prozesse und Threads wissen, sind diese Strategien kaum auf den Anwendungszweck abgestimmt. Häufig verlassen sich Programme nicht nur darauf. In vielen Fällen wird einfach eine für den Einsatzzweck und die verfügbare Anzahl an Prozessorkernen optimale Anzahl an Prozessen oder Threads erzeugt, die von ihnen ausgeführten Aufgaben werden ihnen dynamisch je nach Bedarf innerhalb des Programms zugewiesen. Beispielsweise wird ein zentraler Pool an Aufgaben verwaltet; jeder Prozess oder Thread, der gerade nichts zu tun hat, holt sich eine Aufgabe aus dem Pool und erledigt diese. In Wahrheit ist es etwas komplexer, weil ein zentraler Pool leicht zu einem Engpass werden könnte, aber die Idee bleibt gleich. Die Auswahl der nächsten zu erledigenden Aufgabe liegt somit in unserer Hand.

2.4.3 Streben nach Unabhängigkeit

Der Erfolg in der parallelen Programmierung lässt sich im Wesentlichen an einer Zahl ablesen: dem *Speedup.* Gemessen wird er durch $S_p = T_1/T_p$, wobei S_p für den Speedup bei Lösung einer Aufgabe auf einem System mit p Prozessoren steht, T_p für die dafür aufgewendete Zeit und T_1 für die Zeit zur Lösung der gleichen Aufgabe auf einem System mit nur einem Prozessor. Auf Basis dieser Definition lassen sich Diagramme zeichnen, die den Speedup für verschiedene Anzahlen an Prozessoren veranschaulichen. Durch die vielen verschiedenen Formen der Parallelität dürfen wir die Definition von Speedup nicht immer wörtlich nehmen, sondern müssen sie auf die Art der Parallelität beziehen, etwa statt auf die Anzahl der Prozessoren auf die Anzahl der Prozessorkerne oder auf Wortlängen bei SIMD-Parallelität. Jedenfalls kann der Speedup aus theoretischen Überlegungen niemals größer als p sein. Tatsächlich bleibt er immer kleiner als p, weil es in jedem Programm auch Teile gibt, die nicht parallelisierbar sind. Gar nicht so selten wird stolz von erreichten Speedups berichtet, die p übersteigen. Dahinter stecken immer unzulässige Vergleiche, T_p bezieht sich etwa nicht auf die gleiche Aufgabe wie T_1, oder der Speedup gilt nur für bestimmte Daten, nicht für andere (etwa, wenn in großen Datenmengen aufgeteilt auf die Prozessoren nach einem bestimmten Wert gesucht wird, der bei einem einzigen

Prozessor erst recht weit hinten gefunden wird, bei einer Aufteilung auf eine bestimmte Anzahl von Prozessoren von einem der Prozessoren vielleicht schon im ersten Schritt). Eine faire Messung des Speedups ist aus solchen Gründen schwierig. Auch verantwortungsvoll gemessene Zahlen spiegeln nicht die ganze Wahrheit wider. Meist sind nur Durchschnittswerte aus vielen Messungen unter Variation aller möglichen Einflussgrößen aussagekräftig.

Die wesentliche Frage bei der Programmerstellung besteht darin, durch welche Vorgehensweise für eine gegebene Aufgabe der größtmögliche Speedup erreicht werden kann. Es gibt keine immer und überall anwendbare Vorgehensweise, die sicher zum Ziel führt. Die größten Erfolgsaussichten verspricht eine Reduktion der Abhängigkeiten zwischen Daten. Es ist relativ problemlos möglich, dieselben Daten von verschiedenen Prozessen gleichzeitig lesen zu lassen, solange garantiert werden kann, dass diese Daten nicht überschrieben werden. Die Ergebnisse von Berechnungen müssen in der Regel irgendwo abgelegt, also geschrieben werden. Für jedes erzeugte Datenelement muss ein Platz gefunden werden, auf dem kein anderes Datenelement abgelegt wird, ohne sich für die Findung des Platzes mit anderen Threads oder Prozessen koordinieren zu müssen. Sockets und Pipelines sind bestens dafür geeignete Mechanismen, Shared-Memory und Dateiinhalte aber nicht. Insbesondere für die Koordination von Threads steht im Wesentlichen nur gemeinsamer Speicher mit ausreichend guter Effizienz zur Verfügung. In Java gibt es durch Java-8-Streams einen Mechanismus, der den Pipelines ähnliche Strukturen im Speicher nachbildet. Tatsächlich werden die Schwierigkeiten bei Verwendung von Pipelines oder Java-8-Streams nur auf eine andere Ebene verlagert, weil noch immer eine Organisationsstruktur gefunden werden muss, sodass jede Pipeline und jeder Strom nur an einer Stelle gelesen und an einer anderen geschrieben wird.

Folgendes Beispiel soll demonstrieren, wie in den zu verarbeitenden Daten Abhängigkeiten zwischen Lesen und Schreiben vermieden werden können. Alle diesbezüglichen Überlegungen hängen stark von der Problemstellung ab und können nicht verallgemeinert werden. Im Beispiel wollen wir zählen, wie viele Primzahlen kleiner einer gegebenen Obergrenze `MAX_NUM` existieren. Algorithmisch bietet sich dafür das bekannte „Sieb des Eratosthenes“ an, bei dem jede Zahl n versuchsweise durch alle kleineren Primzahlen bis zur Wurzel von n dividiert werden. Das Problem dabei: Wir müssen die Primzahlen in aufsteigender Reihenfolge finden, wobei immer wieder auf schon gefundene Zahlen zugegriffen werden muss. Es besteht also eine natürliche Abhängigkeit zwischen dem Schreiben und nachfolgenden Lesen gefundener Primzahlen. Glücklicherweise müssen Primzahlen, die schon gefunden wurden, nachträglich nicht mehr geändert werden und die Wurzel aus n ist recht klein im Vergleich zu n. Diese Eigenschaften machen wir uns zu Nutze: Wir geben die kleine Zahl an Primzahlen bis 16 vor und berechnen daraus alle Primzahlen bis 256, weil die Wurzel aus 256 gleich 16 ist, sodass während dieser Berechnung nicht schon auf die neu berechneten Primzahlen zugegriffen werden muss. Dann fassen wir die schon bekannten mit den gefundenen Primzahlen zusammen. In der nächsten Phase berechnen wir aus den Primzahlen bis 256 jene bis $256 * 256$, dann jene bis $256 * 256 * 256 * 256$ und so weiter. Nach wenigen Phasen sind alle `long`-Zahlen abgearbeitet.

```
import java.util.Arrays;
import java.util.stream.LongStream;

public class Par {
    public static long MAX_NUM = 1L << 28;  // Wert anpassbar
    private static long[] prims = { 2L, 3L, 5L, 7L, 11L, 13L };
    public static void main(String[] args) {
        long low = 16L, high = 256L;
        do {
            int size = prims.length;
            long[] nPrims = inRange(low, high);
            prims = Arrays.copyOf(prims, size + nPrims.length);
            System.arraycopy(nPrims,0,prims,size,nPrims.length);
            low = high;
            high = high * high;
            if (high > MAX_NUM)
                high = MAX_NUM;
        } while (low < MAX_NUM);
        System.out.println(prims.length);
    }
    private static long[] inRange(long low, long high) {
        return LongStream.range(low, high).parallel()
                         .filter(Par::isPrime).toArray();
    }
    private static boolean isPrime(long n) {
        long sqrt = (long) Math.sqrt(n);
        return Arrays.stream(prims).takeWhile(v -> v <= sqrt)
                                   .allMatch(v -> n % v != 0);
    }
}
```

Für die eigentliche Parallelisierung des Programms wurde wenig Aufwand betrieben. Einzig und alleine in `inRange` wird auf einen Strom von `long`-Zahlen zwischen `low` (inklusive) und `high` (exklusive) die Methode `parallel()` angewandt, die automatisch eine passende Zahl an Threads erzeugt, wovon jeder einen Ausschnitt aus dem Wertebereich verarbeitet. Ein Aufruf von `filter` lässt nur die Primzahlen durch, die von `isPrime` als solche identifiziert wurden und die danach in einem Array passender Größe abgelegt werden. In `main` werden die einzelnen so entstandenen Arrays zusammenkopiert. Die Methode `isPrime` ist selbst nicht parallel, sondern verwendet einen sequentiellen Strom von Arrayinhalten (das sind die schon bekannten Primzahlen), beendet den Strom wenn alle Primzahlen bis zur Wurzel von n abgearbeitet sind und gibt `true` zurück, wenn sich n durch keine dieser Zahlen ohne Rest dividieren lässt.

Es ist häufig so wie im Beispiel, dass der Umgang mit Parallelität selbst nicht schwer ist, diesen Teil können wir Werkzeugen überlassen. Die Suche nach einer geeigneten Zerlegung der Aufgabenstellung und der Daten in unabhängige

Teile ist nicht so leicht automatisierbar. Natürlich ist das Beispielprogramm noch nicht optimal. Zur Verbesserung könnten wir an vielen Schräubchen drehen (etwa andere Aufteilung der zu untersuchenden Wertebereiche), was die Programmkomplexität aber vermutlich erhöhen würde.

In der Praxis wird Parallelprogrammierung vor allem dort eingesetzt, wo riesige Datenmengen mittels komplexer Algorithmen verarbeitet werden müssen. Häufig kommen immer wieder gleiche oder ähnliche grundlegende Algorithmen zum Einsatz, die von bestimmten Strukturierungen der Daten ausgehen. Beispielsweise beruhen viele technische oder physikalische Problemstellungen auf der Lösung umfangreicher Gleichungssysteme mittels linearer Algebra. Für solche Aufgaben wird überwiegend *LAPACK* eingesetzt, eine Bibliothek mit fertigen Lösungen für zahlreiche Aufgaben der linearen Algebra. LAPACK beruht wiederum auf *BLAS,* einer für viele Hardwarearchitekturen und Sprachen optimierten Bibliothek mit grundlegender Funktionalität für lineare Algebra. In solchen Bibliotheken stecken über viele Jahrzehnte gesammelte Erfahrungen in der Optimierung paralleler Algorithmen, die ständig an neue Entwicklungen angepasst wurden. Für Aufgaben, für die diese Bibliotheken geeignet sind, wird es ohne Zuhilfenahme von Bibliotheken nur mit sehr großem Aufwand und viel Wissen gelingen können, eine ähnlich gute Effizienz zu erreichen. LAPACK-Programmierung gilt daher heute praktisch schon als Synonym für das Lösen von Aufgaben im Bereich der linearen Algebra. Generell werden heute viele, auch sehr komplexe Aufgaben in der parallelen Programmierung durch Verwendung spezialisierter und gut optimierter fertiger Bibliotheken erledigt.

2.5 Nebenläufigkeit

Es mag überraschen, dass Threads und die Synchronisation von Threads im Abschnitt über parallele Programmierung nur am Rande erwähnt wurden. Tatsächlich kommen Threads vorwiegend im Bereich der nebenläufigen Programmierung (*concurrent programming*) zum Einsatz. Jeder Thread beschreibt einen in sich logisch konsistenten Handlungsstrang. Meist laufen viele Handlungsstränge gleichzeitig ab. Die Anzahl der Threads richtet sich vorwiegend nach dem Bedarf an Handlungssträngen, nicht nur nach der Zahl verfügbarer Recheneinheiten. Häufig wird die Ausführung eines Handlungsstrangs unterbrochen, etwa weil auf eine Eingabe, das Eintreffen von Daten von einem entfernten Rechner oder das Freiwerden geteilter Ressourcen gewartet werden muss. Im Gegensatz zur parallelen Programmierung steht der rasche Abschluss einer aufwändigen Rechenaufgabe nicht zentral im Fokus. Das heißt nicht, dass die Programmeffizienz vernachlässigt wird, sondern nur, dass ein sparsamer Umgang mit Rechenzeit, Speicherverbrauch und anderen Ressourcen eher im Mittelpunkt steht als ein möglichst rasch vorliegendes Endergebnis. Wie z. B. bei einem Betriebssystem gibt es häufig gar kein „Endergebnis“, sondern es wird ständig etwas gemacht, etwa auf Aufträge gewartet, die abzuarbeiten sind. Beispielsweise wartet eine Telefonanlage auf Anrufe und leitet Datenpakete weiter, viele gleichzeitig; daneben werden Abrechnungen erstellt und Zusatzdienste an-

geboten. Das Ziel besteht manchmal in einem möglichst großen Durchsatz, also der Erledigung möglichst vieler Aufträge pro Zeiteinheit. Manchmal gibt es Threads, deren einzige Aufgabe es ist, lange auf das Eintreten eines Ereignisses zu warten, um dann eine kleine Aufgabe zu erledigen, etwa zu einer bestimmten Uhrzeit einen Weckton erklingen zu lassen. Aber auch Simulationen zeitabhängiger Abläufe lassen sich damit gut realisieren. Es gibt also eine breite Palette an Einsatzmöglichkeiten.

2.5.1 Threads und Mutual-Exclusion

Folgendes Beispiel setzt Nebenläufigkeit zur Simulation der Ausbreitung einer Bakterienkultur in einer quadratischen Schale ein. Ein zweidimensionales Array stellt die Schale dar, in jedem Eintrag befindet sich ein Objekt vom Typ `State`, das angibt, wie viel Nahrung ein Bakterium an dieser Stelle finden kann (positive Zahl) bzw. ob sich an der Stelle schon ein Bakterium befindet und den Platz gegen andere verteidigt. Jedes Bakterium wird durch einen Thread dargestellt. Bakterien können sich abhängig vom vorhandenen Nahrungsvorrat nach gewissen Wartezeiten vermehren und Nachbarfelder besiedeln. Nach einer gewissen Zeit ohne Nahrung sterben sie, ebenso wenn sie ein Feld besiedeln wollen, das schon besiedelt ist. Simpler Ausgabetext zeigt an, in welcher Reihenfolge Bakterien an welchen Stellen entstehen und sterben. In diesem Beispiel dient Nebenläufigkeit dazu, mit durch einen Zufallszahlengenerator gewählten Wartezeiten zwischen den Schritten im Leben eines Bakteriums umzugehen. Zufall spielt an mehreren Stellen eine Rolle und bewirkt, dass jede Programmausführung zu anderen Simulationsergebnissen führt. Dennoch entstehen immer wieder einander ähnliche Muster, die genauer untersucht werden können.

```
public class BactSim implements Runnable {
    private static final int SIZE = 100;
    private static final State[][] field = new State[SIZE][SIZE];
    private int x, y, fitness;

    public static void main(String[] args) {
        for (int i = 0; i < SIZE; i++)
            for (int j = 0; j < SIZE; j++)
                field[i][j] = new State();
        new Thread(new BactSim(SIZE/2, SIZE/2)).start();
    }

    private BactSim(int x, int y) {
        this.x = x;
        this.y = y;
        fitness = field[x][y].occupy();
        System.out.println(x + "," + y + ": " + fitness);
    }
```

```
    public void run() {
        while (fitness-- > -(int)(Math.random() * 10)) {
            try { Thread.sleep((int)(Math.random() * 50));
            } catch (InterruptedException e) { break; }
            int dx, dy;
            do { dx = (int) (Math.random() * 3) - 1;
                 dy = (int) (Math.random() * 3) - 1;
            } while ((dx == 0 && dy == 0) || x+dx < 0 ||
                     x+dx >= SIZE || y+dy < 0 || y+dy >= SIZE);
            if (fitness > 0)
               new Thread(new BactSim(x + dx, y + dy)).start();
        }
        field[x][y].gone();
        System.out.println(x + "," + y + ": gone");
    }
}

class State {
    private int food = (int) (Math.random() * 20);
    public synchronized int occupy() {
        int res = food;
        food = -10;
        return res;
    }
    public synchronized void gone() {
        food = 0;
    }
}
```

Die Klasse `BactSim` implementiert das Interface `Runnable`, wodurch die Methode `run()` in jedem Objekt von `BactSim` in einem eigenen Thread ausführbar wird. Jedes Objekt von `Thread` stellt einen eigenen Thread dar. Der Konstruktor von `Thread` nimmt ein Objekt von `Runnable` als Parameter, in diesem Fall ein Objekt von `BactSim`. Durch einen Aufruf von `start()` beginnt der Thread zu laufen, der `run()` ausführt. In `main()` wird nach der Initialisierung des Arrays nur ein Thread (für ein Bakterium etwa in der Mitte der Schale) erzeugt und gestartet. Der Konstruktor von `BactSim` ruft `occupy()` im entsprechenden `State`-Objekt auf, um den Platz und die dort gefundene Nahrung in Besitz zu nehmen. Innerhalb von `run()` werden von gut genährten Bakterien weitere Threads auf Nachbarfeldern erzeugt und gestartet, sodass sich nach kurzer Zeit meist viele gleichzeitig laufende Threads ergeben. Sind die Nahrungsvorräte (zufallsabhängig) aufgebraucht, endet die Schleife in `run()` und der Arrayeintrag wird darüber informiert, dass der Platz nicht mehr besiedelt ist; mit dem Ende von `run()` endet auch die Ausführung des entsprechenden Threads. In der Schleife in `run()` wird `Thread.sleep(...)` ausgeführt, eine Methode, die den aktuellen Thread für die gegebene Anzahl an Millisekunden (das ist nur ein gro-

ber Richtwert) pausieren lässt. Während des Wartens können andere Threads ausgeführt werden. Es ist nicht ausgeschlossen, dass ein wartender Thread von außen durch die Methode `interrupt()` beendet wird. In einem solchen Fall wird eine `InterruptedException` ausgelöst, um dem beendeten Thread noch die Möglichkeit zu geben, den Zustand aufzuräumen. Diese Ausnahme muss behandelt werden, was sich im Beispiel wie aufgebrauchte Nahrung auswirkt.

Wenn mehrere Threads auf gleiche Variablen zugreifen oder sie sogar ändern, ist Synchronisation notwendig, um Race-Conditions zu vermeiden. Solche Variablenzugriffe erfolgen im Beispiel in den Methoden `occupy()` und `gone()`, da diese Methoden von mehreren Threads aus auch auf dem gleichen Objekt gleichzeitig aufrufbar sind. Diese Methoden sind als `synchronized` deklariert. Im gleichen Objekt ist zu jedem Zeitpunkt höchstens eine synchronized-Methode ausführbar; werden weitere Methoden aufgerufen, bevor die erste beendet ist, werden die Threads, die hinter den weiteren Aufrufen stehen, so lange blockiert, bis die erste und alle vorherigen Methoden auf diesem Objekt beendet sind. Java garantiert damit Mutual-Exclusion und schließt Parallelausführungen von synchronized-Methoden auf gleichen Objekten aus. Die Threads bekommen nacheinander *exklusiven Zugriff* auf das betrachtete *Synchronisationsobjekt.*

Mutual-Exclusion durch `synchronized` gibt es nur auf gleichen Objekten. Das heißt, im Beispiel können mehrere Bakterien gleichzeitig unterschiedliche Plätze in der Schale besetzen, ohne sich gegenseitig zu stören. Das geht, weil jeder Arrayeintrag als Synchronisationsobjekt dient, nicht das gesamte Array. Auch wenn der Schwerpunkt nicht auf kürzestmöglicher Laufzeit liegt, muss darauf geachtet werden, dass sich Threads möglichst wenig gegenseitig behindern und die kritischen Abschnitte (also die Zeiten für die Ausführungen von `occupy()` und `gone()`) möglichst kurz bleiben. Im Beispiel ist es zwar möglich, aber relativ unwahrscheinlich, dass zwei Bakterien gleichzeitig auf den gleichen Arrayeintrag zugreifen wollen. Das ist der Idealfall für Synchronisation. Noch besser wäre es aber, auf Synchronisation gänzlich verzichten zu können.

Neben synchronized-Methoden unterstützt Java synchronized-Blöcke, bei denen wir das zu verwendende Synchronisationsobjekt explizit angeben. Beispielsweise könnte der Aufruf von `occupy()` so aussehen, wenn wir `synchronized` in der Definition von `occupy()` weglassen:

```
synchronized (field[x][y]) { fitness = field[x][y].occupy(); }
```

Im Wesentlichen ist das eine syntaktische Variante mit gleicher Semantik. Durch Verwendung dieser Variante können wir das Synchronisationsobjekt beeinflussen. Beispielsweise könnten wir statt `field[x][y]` das gesamte Array `field` verwenden (natürlich konsistent, also auch für Aufrufe von `gone()`). Dadurch würden sich viel mehr Aufrufe gegenseitig behindern, das wäre also nicht gut. Von Vorteil wäre, dass wir das Array von `State`-Objekten durch ein `int`-Array ersetzen und Aufrufe der Methoden und des Konstruktors von `State` direkt in den Programmtext von `BactSim` verschieben könnten, eine Vereinfachung. Für feingranulare Synchronisation brauchen wir jedoch `State`-Objekte, weil `synchronized` als Synchronisationsobjekte nur Objekte, keine primitiven Datenelemente verwenden kann. In Java müssen wir den Zusatzaufwand der Ob-

jekterzeugung in Kauf nehmen. Den Schreibaufwand könnten wir etwas reduzieren, indem wir statt `State` die vordefinierte Klasse `AtomicInteger` verwenden; ein Aufruf von `getAndSet(-10)` wäre damit ein direkter Ersatz für einen Aufruf von `occupy()` und `set(0)` für `gone()`.

Synchronisation durch `synchronized` kann leicht zu Liveness-Problemen führen, insbesondere wenn gleichzeitig auf mehrere Synchronisationsobjekte zugegriffen wird. Nehmen wir an, dass ein Bakterium von einem Arrayeintrag auf einen anderen wandern kann. In `State` fügen wir folgende Methode ein:

```
public synchronized int enter(State from) {
    from.gone();
    return occupy();
}
```

In der Schleife von `run()`, z. B. als `else`-Zweig der `if`-Anweisung rufen wir diese Methode auf, um von `field[x][y]` auf `field[x+dx][y+dy]` zu wandern:

```
... = field[x+dx][y+dy].enter(field[x][y]);
```

Wahrscheinlich zeigt das geänderte Programm bei der Ausführung keine Auffälligkeiten. Dennoch haben wir einen potentiellen Deadlock eingebaut, der sich sehr selten zeigt: `enter(...)` ruft `gone()` auf, wobei ein Thread sowohl `field[x+dx][y+dy]` als auch das Nachbarfeld `field[x][y]` gleichzeitig als Synchronisationsobjekt benötigt. Ein anderer Thread will ungefähr gleichzeitig durch einen Aufruf von `enter(...)` von `field[x+dx][y+dy]` auf `field[x][y]` wandern. Dabei kann es passieren, dass der eine Thread exklusiven Zugriff auf `field[x+dx][y+dy]` bekommt, der andere auf `field[x][y]`. Nun benötigen beide Threads für den Aufruf von `gone()` exklusiven Zugriff auf das jeweils andere Objekt, können diesen aber nicht bekommen, weil schon der jeweils andere Thread exklusiven Zugriff auf dieses Objekt hat. Das ist ein Deadlock, bei dem zwei Threads endlos darauf warten, dass der jeweils andere die Ausführung einer Methode beendet. Ein Deadlock kann auch eine größere Zahl an über exklusive Objektzugriffe zyklisch miteinander verbundenen Threads umfassen, die jeweils auf das Ende des exklusiven Zugriffs durch den vorigen Thread warten.

Die Methode `enter(...)` ruft auch `occupy()` auf, etwas ausführlicher hingeschrieben `this.occupy()`. Dieser Aufruf ist problemlos, weil der Thread, der `enter(...)` ausführt, ohnehin schon exklusiven Zugriff auf `this` hat und daher `occupy()` direkt ausführen kann, ohne warten zu müssen.

Die größte Schwierigkeit besteht darin, mögliche Deadlock-Situationen zu erkennen. Ein hoher Abstraktionsgrad erschwert das gewaltig, weshalb nebenläufige Programmierung häufig auf prozeduraler Programmierung aufbaut, nicht auf objektorientierter, obwohl objektorientierte Sprachen eingesetzt werden. Sobald eine mögliche Deadlock-Situation erkannt ist, gibt es wenige Ansätze zu deren Beseitigung. In obigem Beispiel ist das Problem einfach zu beseitigen, indem kein Thread exklusiven Zugriff auf mehrere Objekte gleichzeitig bekommt, sondern nur hintereinander. Meist ist dieser einfache Ansatz nicht zielführend, weil es eben inhaltlich unvermeidbar ist, gleichzeitig exklusiv auf mehrere Objekte

zuzugreifen. In solchen Fällen bietet es sich an, die Reihenfolge einzuschränken, in der exklusiver Zugriff auf Objekte erlangt werden kann. Im Beispiel könnten wir verlangen, dass `field[u][v].enter[x][y]` nur für `u<x` oder bei `u==x` für `v<y` aufrufbar ist, sodass keine Zyklen in der Reihenfolge entstehen können. Die Folgen dieser Einschränkung lassen sich etwa durch Bereitstellung einer weiteren Methode lösen, in der die Bedeutungen von `this` und dem expliziten Parameter `from` vertauscht sind und für deren Aufruf genau die umgekehrten Einschränkungen gelten. Ein solcher Ansatz hat in der Regel gravierende Auswirkungen auf die gesamte Organisation des Programms. Nebenläufige Programmierung ist damit nicht nur eine Erweiterung eines prozeduralen oder funktionalen Programmierstils, sondern bekommt einen eigenen Charakter, der ganz wesentlich von den Notwendigkeiten zur Synchronisation bestimmt wird.

Livelocks sind oft noch schwieriger zu erkennen als Deadlocks. Häufig steht „Livelock“ für „aktives Warten“, also Situationen, in denen viel Zeit damit verbracht wird, immer wieder nachzufragen, ob eine bestimmte benötigte Ressource zur Verfügung steht. Damit können zyklische Abhängigkeiten wie bei Deadlocks entstehen, die jedoch nicht auffallen, weil nicht gewartet, sondern stets viel getan wird, allerdings ohne einen Fortschritt zu erzielen. Wegen der Gefahr von Livelocks sollte auf aktives Warten generell verzichtet werden.

2.5.2 Warten bis es passiert

Häufig muss auf Ereignisse reagiert werden, die zu unvorhersehbaren Zeitpunkten eintreten. Das typische Beispiel dafür ist das *Produzenten-Konsumenten-Problem,* bei dem einige „Produzenten“ wiederholt (meist zu nicht vorhersehbaren Zeitpunkten) Daten produzieren, die von einigen „Konsumenten“ verarbeitet werden. Der übliche Lösungsansatz besteht darin, die Kommunikation und Synchronisation zwischen Produzenten und Konsumenten über einen *Puffer* beschränkter Größe zu organisieren. Die Beschränkung der Größe verhindert, dass Produzenten ständig Daten produzieren, während Konsumenten „verhungern“ (Starvation). Produzenten und Konsumenten müssen aufeinander warten:

```
public class ProducerConsumer {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        for (char c = 'a'; c <= 'z'; c++) {
            new Thread(new Producer("" + c, buffer)).start();
            new Thread(new Consumer("" + c, buffer)).start();
        }
    }
}

class Buffer {
    private final static int SIZE = 1 << 3;
    private String[] buffer = new String[SIZE];
    private int in, out;
```

```
    public synchronized void put(String v) {
        while (buffer[in] != null)
            try { wait(); }
            catch (InterruptedException e) { return; }
        notifyAll();
        buffer[in] = v;
        in = (in + 1) & (SIZE - 1);
    }
    public synchronized String get() {
        while (buffer[out] == null)
            try { wait(); }
            catch (InterruptedException e) { return null; }
        notifyAll();
        String res = buffer[out];
        buffer[out] = null;
        out = (out + 1) & (SIZE - 1);
        return res;
    }
}

class Producer implements Runnable {
    private String id;
    private Buffer buffer;

    public Producer(String id, Buffer buffer) {
        this.id = id;
        this.buffer = buffer;
    }
    public void run() {
        for (int i = 1; i <= 100; i++) buffer.put(id + i);
    }
}

class Consumer implements Runnable {
    private String id;
    private Buffer buffer;

    public Consumer(String id, Buffer buffer) {
        this.id = id;
        this.buffer = buffer;
    }
    public void run() {
        for (int i = 1; i <= 100; i++)
            System.out.println(id + ": " + buffer.get());
    }
}
```

Produzent und Konsument sind irgendwelche Threads (entsprechende Klassen durch andere ersetzbar), die entweder über `put(...)` Daten an den gemeinsamen Puffer übergeben oder über `get()` Daten aus dem Puffer holen. Die Klasse `Buffer` enthält die gesamte Komplexität der Kommunikation und Synchronisation. Natürlich sind `put(...)` und `get()` als `synchronized` deklariert, um exklusiven Zugriff auf den Puffer und die darin vorkommenden Objektvariablen zu erhalten. Mutual-Exclusion ist für die Synchronisation jedoch nicht hinreichend, weil zwei Sonderfälle berücksichtigt werden müssen:

- Wenn der Puffer leer ist (erkennbar durch `buffer[out] == null`), muss jeder Thread, der `get()` ausführt, so lange blockiert werden, bis wieder etwas im Puffer ist.
- Wenn der Puffer voll ist (erkennbar durch `buffer[in] != null`), muss jeder Thread, der `put(...)` ausführt, so lange blockiert werden, bis wieder ein freier Platz im Puffer ist.

Genau für solche Zwecke gibt es in Java die Methode `wait()`, die nur in einem Synchronisationsobjekt mit exklusivem Zugriff aufgerufen werden darf (andernfalls wird eine Ausnahme ausgelöst). Ein Aufruf von `wait()` bewirkt, dass der aktuelle Thread seinen exklusiven Zugriff auf das Synchronisationsobjekt verliert und auf unbestimmte Zeit suspendiert, also von der weiteren Ausführung ausgeschlossen wird. Irgendwann (spätestens vor Beendigung des Programms) wird der suspendierte Thread wieder aufgeweckt und bekommt erneut exklusiven Zugriff auf das Synchronisationsobjekt, wobei die Programmausführung an der Stelle direkt nach dem Aufruf von `wait()` fortgesetzt wird. Wie bei `sleep(...)` muss die `InterruptedException` behandelt werden, die zu jedem Zeitpunkt auftreten könnte.[4] Der Zustand des Puffers könnte während des Wartens verändert worden sein, vielleicht aber auch nicht. Daher müssen wir die Bedingung nach der Rückkehr aus `wait()` in einer Schleife erneut überprüfen und wieder `wait()` aufrufen, wenn der Zustand für die Fortsetzung der Methodenausführung noch nicht passt. Wenn der Zustand passt, wird die Methode normal fortgesetzt und entweder ein neuer Eintrag in das Array gemacht oder ein bestehender Eintrag entfernt und zurückgegeben. Zusätzlich wird durch einen Aufruf von `notifyAll()` jeder Thread aufgeweckt, der zuvor durch einen Aufruf von `wait()` auf dem gleichen Synchronisationsobjekt suspendiert wurde. Aufgeweckte Threads dürfen nicht sofort weiterlaufen, sondern bekommen erst nacheinander, nachdem die gerade ausgeführte `synchronized`-Methode beendet ist, wieder exklusiven Zugriff auf das Synchronisationsobjekt. In unserem Fall bewirkt `notifyAll()` daher, dass jeder auf einen Puffer-Zugriff wartende Thread erneut die Gelegenheit bekommt zu überprüfen, ob der Puffer jetzt in einem für ihn passenden Zustand ist.

[4] Abgesehen von dieser Gemeinsamkeit unterscheiden sich `wait()` und `sleep(...)` grundlegend voneinander. So darf `wait()` nur aufgerufen werden, während exklusiver Zugriff auf ein Synchronisationsobjekt besteht, aber `sleep(...)` würde große Probleme verursachen, wenn der Aufruf während des exklusiven Zugriffs auf ein Synchronisationsobjekt erfolgen würde. Darüber hinaus wartet `wait()` für unbestimmte Zeit, `sleep(...)` jedoch nur eine bestimmte, ungefähr angegebene Zeit.

Neben `notifyAll()` gibt es auch die Methode `notify()`, die nicht alle, sondern maximal einen suspendierten Thread aufweckt. Für unsere Zwecke ist `notify()` nicht brauchbar, weil wir nicht bestimmen können, welcher suspendierte Thread aufgeweckt werden soll. Beispielsweise könnte `notify()` einen Thread aufwecken, der gleich wieder `wait()` aufrufen muss, weil der Objektzustand nicht passt, obwohl andere suspendierte Threads ausführbar wären. Dadurch könnten wichtige Threads verhungern. Eingeführt wurde `notify()` für Fälle, in denen der Objektzustand nach dem Aufwecken immer passt. Es gibt viele Gründe, aus denen suspendierte Threads unvorhersehbar aufgeweckt werden können. Sind zufällig gerade alle Threads aufgeweckt worden, wirkt sich `notify()` auf keinen Thread aus und hat damit keinen Effekt.

Das Produzenten-Konsumenten-Problem ist leicht verallgemeinerbar. Als Produzenten verwenden wir beispielsweise einen Thread, der auf einem Socket auf das Eintreffen von Nachrichten wartet und diese entsprechend ihrer Art sowie auf Basis der Systemauslastung an einen von mehreren Puffern weiterleitet. Weitere Produzenten könnten an anderen Datenquellen sitzen und eintreffende Nachrichten auf die gleiche Weise an Puffer weiterleiten. An den anderen Enden der Puffer sitzen Konsumenten, die eintreffende Nachrichten (unabhängig von ihrer Quelle, aber nach ihrer Art sortiert) bearbeiten und die Ergebnisse als Antworten verschicken. Auf diese Weise sind hochkomplexe Systeme aufbaubar, die die Rechenlast auf viele Rechner verteilen und mit einer hohen Auslastung zurechtkommen. Zwischengeschaltete Puffer sorgen dafür, dass eintreffende Nachrichten nicht sofort bearbeitet werden müssen, sondern erst dann, wenn Kapazitäten dafür frei werden. Durch Beschränkung der Größe der Puffer kann dennoch dafür gesorgt werden, dass die Wartezeiten nicht beliebig lang werden, sondern mangels Kapazität nicht mehr bearbeitbare Nachrichten gleich zurückgewiesen werden. Aus Anwendersicht ist es besser, wenn das System einen Auftrag wegen Überlastung ablehnt (und so gegen mögliche Denial-of-Service-Attacken auftritt), als nach übermäßig langer Wartezeit den dann vielleicht nicht mehr aktuellen Auftrag auszuführen oder gar nicht zu reagieren.

Das Konzept von Java, in dem über `synchronized`-Methoden sowie `wait()` und `notifyAll()` synchronisiert wird, ist eine Variante der schon sehr lange bekannten *Monitore* [14]. Über Semaphore hinausgehend wurden im Laufe der Zeit viele Sprachkonzepte für die Synchronisation entwickelt. Bekannt ist das in Ada integrierte *Rendezvous*-Konzept, bei dem Nachrichten an Thread-ähnliche Einheiten (als *Tasks* bezeichnet) geschickt werden, durchaus vergleichbar mit dem Schicken von Nachrichten an Objekte in der objektorientierten Programmierung. Allerdings gibt es keine Methoden zum Abarbeiten der Nachrichten. Stattdessen enthalten Tasks, die so wie Java-Threads meist in Endlosschleifen laufen, `accept`-Anweisungen, welche die Nachrichten bearbeiten. Das heißt, der Task, der eine Nachricht schickt, muss warten, bis der Task, an den die Nachricht geschickt wird, eine passende `accept`-Anweisung ausführt und die `accept`-Anweisung muss warten, bis eine Nachricht da ist (synchrone Kommunikation). Während der Ausführung der `accept`-Anweisung treffen sich die beiden Tasks (daher der Begriff „Rendezvous“), nach Abarbeitung dieser Anweisung laufen beide Tasks wieder getrennt voneinander weiter.

Noch direkter als das Rendezvous-Konzept verdeutlicht das *Actor*-Modell [3], wie ein nebenläufiges Vorgehen als Zusammenarbeit Thread-ähnlicher Einheiten (*Actors* genannt) durch den Austausch von Nachrichten verstanden werden kann: Jeder Actor ist ein eigener Handlungsstrang mit eigenen, nicht von außen zugreifbaren Variablen und einer eigenen Warteschlange an eingehenden Nachrichten. Der Actor kann Nachrichten aus seiner Warteschlange lesen und verarbeiten, dabei kann er auch Nachrichten an andere Actors schicken, die dann in der Warteschlange des Empfängers landen. Das Senden von Nachrichten ist asynchron, das heißt, es wird nicht gewartet, bis die Nachricht abgearbeitet ist. Obwohl das Actor-Modell schon recht alt ist (ebenso wie das Monitor- und Rendezvous-Konzept), ist es seit kurzem wieder häufiger in aktuellen Programmiersprachen verfügbar (etwa in Scala, es gibt auch mehrere Actor-Implementierungen für Java).

Schon vor der Ausformulierung des Actor-Modells waren ähnliche Konzepte im Umlauf. Eines dieser frühen Konzepte stand hinter der Entwicklung von *Smalltalk,* einer der ersten objektorientierten Sprachen. Das Ziel bestand darin, Actor-ähnliche Objekte, die jeweils einen eigenen Handlungsstrang haben, durch den Austausch von Nachrichten miteinander kommunizieren zu lassen. Dieses Ziel konnte nicht erreicht werden, weil die damalige Hardware mit der benötigten großen Anzahl an Threads nicht ausreichend effizient umgehen konnte. Als Ausweg wurde das heute übliche Objektkonzept entwickelt, in dem Objekte keine eigenen Handlungsstränge besitzen. Die frühe Terminologie ist teilweise erhalten geblieben; ein Methodenaufruf wird als „Senden einer Nachricht“ bezeichnet, was ursprünglich wörtlich zu verstehen war. Es gibt also zumindest historische Gemeinsamkeiten zwischen objektorientierter und nebenläufiger Programmierung. Gemeinsamkeiten dürfen nicht darüber hinwegtäuschen, dass es heute sehr deutliche Unterschiede gibt. Klar ist, dass parallele und objektorientierte Programmierung kaum zusammenpassen können, weil in diesen Paradigmen von gänzlich unterschiedlichen Strukturierungen der Daten ausgegangen werden muss. Die Diskrepanzen zwischen nebenläufiger und objektorientierter Programmierung sind dagegen nicht so offensichtlich, weil es ein breit gefächertes Einsatzgebiet mit unterschiedlichen Ausprägungen gibt. Manchmal sind diese Paradigmen miteinander kombinierbar, in anderen Fällen nicht.

3 Ersetzbarkeit und Untertypen

In diesem Kapitel konzentrieren wir uns auf das zentrale Konzept der objektorientierten Programmierung. In Abschnitt 3.1 untersuchen wir Ersetzbarkeit als Grundlage von Untertypbeziehungen. Der Schwerpunkt liegt auf Ersetzbarkeit basierend auf Signaturen sowie die Realisierung über nominale Typen in Java, die Abstraktionen der realen Welt ermöglichen. In Abschnitt 3.2 gehen wir auf Beschreibungen des Objektverhaltens ein, die bei der Verwendung von Untertypen zu beachten sind. Es geht um die Berücksichtigung von Zusicherungen über Design-by-Contract sowie Liskov-Ersetzbarkeit. Danach betrachten wir in Abschnitt 3.3 die Vererbung im Zusammenhang mit direkter Codewiederverwendung und stellen Sie der Ersetzbarkeit gegenüber. In Abschnitt 3.4 beleuchten wir einige Details zu Klassen und Interfaces in Java. Schließlich behandeln wir in Abschnitt 3.5 den Umgang mit Exceptions unter Berücksichtigung von Untertypen und Ersetzbarkeit.

3.1 Ersetzbarkeitsprinzip

Untertypbeziehungen sind durch das *Ersetzbarkeitsprinzip* definiert:

> *Definition:* Ein Typ U ist Untertyp eines Typs T, wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.

Ein Objekt eines Untertyps ist per Definition verwendbar, wo ein Objekt eines Obertyps erwartet wird. Wir benötigen das Ersetzbarkeitsprinzip vor allem für

- einen Methodenaufruf mit einem Argument, dessen deklarierter Typ Untertyp des deklarierten Typs des entsprechenden Eingangsparameters ist,
- die Zuweisung eines Objekts an eine Variable, wobei der deklarierte Typ des Objekts ein Untertyp des deklarierten Typs der Variablen ist.

Beide Fälle kommen in der objektorientierten Programmierung häufig vor.

3.1.1 Untertypen und Schnittstellen

Die Frage danach, wann das Ersetzbarkeitsprinzip erfüllt ist, wurde in der Fachliteratur intensiv behandelt [2, 4, 24]. Wir wollen diese Frage hier nur so weit beantworten, als es in der Praxis relevant ist. Fast alles, was wir anhand von Java untersuchen, gilt auch für andere objektorientierte Sprachen, zumindest für solche mit statischer Typprüfung wie C# und C++. Wir gehen davon

aus, dass Typen Schnittstellen von Objekten sind, die in Klassen beziehungsweise Interfaces spezifiziert wurden. Es gibt in Java auch elementare Typen wie `int`, die keiner Klasse entsprechen. Auf solchen Typen haben wir keine Untertypbeziehungen. Deshalb werden wir sie hier nicht näher betrachten.

Eine Voraussetzung für das Bestehen einer Untertypbeziehung in Java ist eine Vererbungsbeziehung auf den entsprechenden Klassen und Interfaces. Die dem Untertyp entsprechende Klasse bzw. das Interface muss durch `extends` oder `implements` von der dem Obertyp entsprechenden Klasse oder dem Interface direkt oder indirekt abgeleitet sein. Anders ausgedrückt: Es werden nominale Typen verwendet – siehe Abschnitte 1.4.2 und 1.6.2. Damit eignen sich Typen gut für die Abstraktion.

Es ist möglich, Untertypbeziehungen auch ohne Vererbung nur auf Basis struktureller Typen zu definieren. Das geschieht hauptsächlich in der Theorie, um formal schwer nachvollziehbare, auf Intuition beruhende Abstraktionen unberücksichtigt lassen zu können.

Strukturelle Untertypbeziehungen. Unter den folgenden Bedingungen stehen strukturelle Typen in einer Untertypbeziehung. Diese Bedingungen gelten allgemein und sind nicht auf eine bestimmte Programmiersprache bezogen. Alle Untertypbeziehungen sind stets

- reflexiv – jeder Typ ist Untertyp von sich selbst,
- transitiv – ist ein Typ U Untertyp eines Typs A und ist A Untertyp eines Typs T, dann ist U auch Untertyp von T,
- antisymmetrisch – ist ein Typ U Untertyp eines Typs T und ist T außerdem Untertyp von U, dann sind U und T äquivalent.

Beliebige strukturelle Typen bezeichnen wir mit U und T sowie A und B. Es gilt „U ist Untertyp von T" wenn folgende Bedingungen erfüllt sind:

- Für jede *Konstante* (also jede Variable, die nach der Initialisierung nur lesende Zugriffe erlaubt) in T gibt es eine entsprechende Konstante in U, wobei der deklarierte Typ B der Konstante in U ein Untertyp des deklarierten Typs A der Konstante in T ist.[1]

  Begründung: Auf eine Konstante kann außerhalb des Objekts nur lesend zugegriffen werden. Wenn wir die Konstante in einem Objekt vom Typ T sehen und lesend darauf zugreifen, erwarten wir uns, dass wir einen Wert vom Typ A erhalten. Diese Erwartung soll auch erfüllt sein, wenn das Objekt vom Typ U ist, wenn also ein Objekt von U verwendet wird, wo wir ein Objekt von T erwarten. Aufgrund der Bedingung gibt es im Objekt vom Typ U eine entsprechende Konstante vom Typ B. Wir bekommen beim lesenden Zugriff einen Wert vom Typ B, wo wir ein Objekt vom Typ A erwarten. Da B ein Untertyp von A sein muss, ist das Objekt

[1] In Java ist es nicht möglich, eine in T eingeführte Konstante in U zu ändern, sodass A und B immer gleich sind. Wir betrachten hier aber nicht Java, sondern den allgemeinen Fall.

von B überall verwendbar, wo ein Objekt von A erwartet wird. Unsere Erwartungen sind daher erfüllt. Die Initialisierung der Konstante müssen wir nicht berücksichtigen, weil sie nur dort (etwa im Konstruktor) erfolgen kann, wo keine Ersetzbarkeit nötig ist (Konstante in T wird in T initialisiert, Konstante in U in U).

- Für jede nach außen sichtbare[2] Variable in T gibt es eine entsprechende Variable in U, wobei die deklarierten Typen der Variablen äquivalent sind.

  Begründung: Auf eine Variable kann lesend und schreibend zugegriffen werden. Ein lesender Zugriff entspricht der oben beschriebenen Situation bei Konstanten; der deklarierte Typ B der Variablen in U muss ein Untertyp des deklarierten Typs A der Variablen in T sein. Wird eine Variable eines Objekts vom Typ T außerhalb des Objekts geschrieben, erwarten wir uns, dass der Variablen jedes Objekt vom Typ A zugewiesen werden darf. Diese Erwartung soll erfüllt sein, wenn das Objekt vom Typ U und die Variable vom Typ B ist. Die Erwartung ist nur erfüllt, wenn A ein Untertyp von B ist. Bei gemeinsamer Betrachtung lesender und schreibender Zugriffe muss B ein Untertyp von A und A ein Untertyp von B sein. Das ist nur möglich, wenn A und B äquivalent sind.

- Für jede Methode in T gibt es eine entsprechende gleichnamige Methode in U, wobei
  - der deklarierte Ergebnistyp der Methode in U ein Untertyp des deklarierten Ergebnistyps der Methode in T ist,
  - die Anzahl der Parameter beider Methoden gleich ist,
  - der deklarierte Typ jeden Eingangsparameters[3] in U ein Obertyp des deklarierten Typs des entsprechenden Parameters in T ist (der auch ein Eingangsparameter sein muss),
  - der deklarierte Typ jeden Durchgangsparameters in U äquivalent zum deklarierten Typ des entsprechenden Parameters in T ist (der auch ein Durchgangsparameter sein muss)
  - und der deklarierte Typ jeden Ausgangsparameters in U ein Untertyp des deklarierten Typs des entsprechenden Parameters in T ist (der auch ein Ausgangsparameter sein muss).

  Begründung: Für die Ergebnistypen der Methoden gilt das Gleiche wie für Typen von Konstanten beziehungsweise lesende Zugriffe auf Variablen: Der Aufrufer einer Methode möchte ein Ergebnis des in T versprochenen

[2] Es wird dringend emfohlen, nur `private` Variablen zu verwenden, die nicht nach außen sichtbar sind. Eine `private` Variable ist nur innerhalb einer Klasse sichtbar, sodass das Ersetzbarkeitsprinzip nicht zur Anwendung kommt oder (anders formuliert) eine Verletzung dieser Regel gar nicht möglich ist. Aus Gründen der Vollständigkeit und zum besseren Verständnis müssen wir dennoch auch den Fall einer nicht-privaten Variable untersuchen. Außerdem kann in Java eine in T eingeführte Variable in U nicht verändert werden, was aber irrelevant ist, weil wir hier den allgemeinen Fall betrachten, nicht Java.

[3] Siehe Abschnitt 1.5.1 für die Unterscheidung zwischen den Parameterarten.

Ergebnistyps bekommen, auch wenn die entsprechende Methode in U ausgeführt wird. Für die Typen der Eingangsparameter gilt das Gleiche wie für schreibende Zugriffe auf Variablen: Der Aufrufer möchte alle Argumente der Typen an die Methode übergeben können, die in T deklariert sind, auch wenn die entsprechende Methode in U ausgeführt wird. Daher dürfen die Eingangsparametertypen in U nur Obertypen der Eingangsparametertypen in T sein. Durchgangsparameter werden vom Aufrufer vor dem Aufruf geschrieben und nach der Rückkehr gelesen; da gelesen und geschrieben werden muss, gelten die gleichen Bedingungen wie für Variablen. Ausgangsparameter werden vom Aufrufer nach der Rückkehr gelesen, wodurch die gleichen Bedingungen wie für Methodenergebnisse gelten. Das Schreiben des Rückgabewerts und Zugriffe auf Parameter innerhalb der aufgerufenen Methode müssen wir nicht betrachten, da innerhalb der Methode der genaue Typ (U oder T) immer bekannt ist; das Ersetzbarkeitsprinzip wird dafür nicht benötigt.

Alle diese Bedingungen gelten nur für Variablen und Methoden, die außerhalb eines Objekts sichtbar sind, die also zur Schnittstelle gehören. Private Inhalte haben keinen Einfluss auf Untertypbeziehungen.

Diese Bedingungen hängen nur von den Strukturen der Typen ab. Sie berücksichtigen das Verhalten in keiner Weise. Solche Untertypbeziehungen sind auch gegeben, wenn die Strukturen nur zufällig zusammenpassen.

Obigen Regeln entsprechend kann ein Untertyp einen Obertyp nicht nur um neue Elemente erweitern, sondern auch deklarierte Typen der Elemente (z. B. Parameter) ändern; das heißt, die deklarierten Typen der Elemente können *variieren.* Folgende Arten der Varianz werden unterschieden:

Kovarianz: Der deklarierte Typ eines Elements im Untertyp ist Untertyp des deklarierten Typs des entsprechenden Elements im Obertyp. Deklarierte Typen von Konstanten und von Ergebnissen der Methoden sowie von Ausgangsparametern sind kovariant. Typen und die betrachteten darin enthaltenen Elementtypen variieren in die gleiche Richtung.

Kontravarianz: Der deklarierte Typ eines Elements im Untertyp ist Obertyp des deklarierten Typs des Elements im Obertyp. Deklarierte Typen von Eingangsparametern sind kontravariant. Typen und die betrachteten darin enthaltenen Elementtypen variieren in entgegengesetzte Richtungen.

Invarianz: Der deklarierte Typ eines Elements im Untertyp ist äquivalent zum deklarierten Typ des entsprechenden Elements im Obertyp. Deklarierte Typen von Variablen und Durchgangsparametern sind invariant. Die betrachteten in den Typen enthaltenen Elementtypen variieren nicht.

Möglichkeiten und Grenzen. Folgendes Beispiel in einer Java-ähnlichen Sprache (Java kann es wegen zusätzlicher Einschränkungen nicht sein) zeigt einige Möglichkeiten von strukturellen Untertypbeziehungen auf:

```
public class A {
    public A meth(B par) { ... }
}
public class B {
    public B meth(A par) { ... }
    public void foo() { ... }
}
```

Entsprechend den oben angeführten Bedingungen ist `B` ein Untertyp von `A`. Die Methode `meth` in `B` kann an Stelle von `meth` in `A` verwendet werden: Der Ergebnistyp ist kovariant verändert, der Parametertyp kontravariant. Wäre die Methode `foo` in `B` nicht vorhanden, dann könnten `A` und `B` sogar als äquivalent betrachtet werden (der Unterschied zwischen `A` und `B` würde verschwinden).

Das alles gilt in einer Java-ähnlichen Sprache, die auf strukturellen Typen beruht. Java verwendet jedoch nominale Typen, und in Java ist `B` kein Untertyp von `A`. Wenn `B` mit der Klausel „`extends A`" deklariert wäre, würde `meth` in `B` die Methode in `A` nicht überschreiben; stattdessen würde `meth` von `A` geerbt und überladen, so dass es in Objekten von `B` beide Methoden nebeneinander gäbe.

Obige Bedingungen für Untertypbeziehungen sind notwendig und für strukturelle Typen auch vollständig. Man kann keine weglassen oder aufweichen, ohne mit dem Ersetzbarkeitsprinzip in Konflikt zu kommen. Die meisten dieser Bedingungen stellen keine praktische Einschränkung dar. Wir kommen kaum in Versuchung sie zu brechen. Nur eine Bedingung, nämlich die geforderte Kontravarianz der Eingangsparametertypen, möchten wir manchmal gerne umgehen. Sehen wir uns dazu ein Beispiel an:

```
public class Point2D {
    protected int x, y;  // von außen sichtbar
    public boolean equal(Point2D p) {
        return x == p.x && y == p.y;
    }
}

public class Point3D {
    protected int x, y, z;
    public boolean equal(Point3D p) {
        return x == p.x && y == p.y && z == p.z;
    }
}
```

Wegen der zusätzlichen von außen sichtbaren Variable in `Point3D` sind die beiden Typen nicht äquivalent und kann `Point2D` kein Untertyp von `Point3D` sein. Außerdem kann `Point3D` kein Untertyp von `Point2D` sein, da `equal` nicht die Kriterien für Untertypbeziehungen erfüllt. Der Parametertyp wäre ja kovariant und nicht, wie gefordert, kontravariant.

Eine Methode wie `equal`, bei der mindestens ein Eingangsparametertyp stets gleich der Klasse (oder dem Interface) ist, in der die Methode definiert ist, heißt

binäre Methode.[4] Die Eigenschaft *binär* bezieht sich darauf, dass der Name der Klasse (des Interfaces) in der Methode mindestens zweimal vorkommt – einmal als Typ von `this` und mindestens einmal als Typ eines Parameters. Binäre Methoden werden häufig benötigt, sind über Untertypbeziehungen (ohne dynamische Typabfragen und Casts) aber prinzipiell nicht realisierbar.

> *Faustregel:* Kovariante Eingangsparametertypen und binäre Methoden widersprechen dem Ersetzbarkeitsprinzip. Es ist sinnlos, in solchen Fällen Ersetzbarkeit anzustreben.

Untertypbeziehungen in Java setzen entsprechende Vererbungsbeziehungen voraus. Vererbung ist in Java so eingeschränkt, dass zumindest alle Bedingungen für Untertypbeziehungen auf strukturellen Typen erfüllt sind. Die Bedingungen werden bei der Übersetzung eines Java-Programms fast lückenlos überprüft. (Eine Ausnahme bezogen auf Arrays und Generizität werden wir in Abschnitt 4.1.3 kennen lernen.)

Untertypbeziehungen sind in Java nicht nur aufgrund nominaler Typen stärker eingeschränkt, als durch obige Bedingungen notwendig wäre. In Java sind alle Typen invariant, abgesehen von kovarianten Ergebnistypen ab Version 1.5. Der Grund dafür liegt darin, dass bei Zugriffen auf Variablen und Konstanten nicht dynamisch, sondern statisch gebunden wird (dafür also keine Ersetzbarkeit nötig ist) und darin, dass Methoden überladen sein können. Da überladene Methoden durch die Typen der Parameter unterschieden werden, wäre es schwierig, überladene Methoden von Methoden mit kontravariant veränderten Typen auseinanderzuhalten.

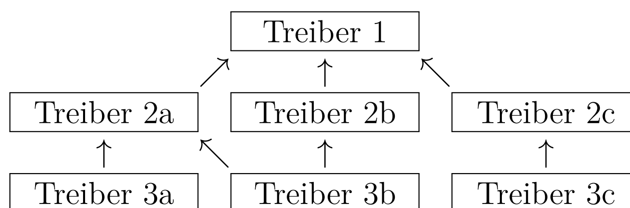
3.1.2 Untertypen und Codewiederverwendung

Die wichtigste Entscheidungsgrundlage für den Einsatz von Untertypen ist die erzielbare Wiederverwendung. Der richtige Einsatz eröffnet Möglichkeiten, die auf den ersten Blick nicht so leicht zu erkennen sind.

Nehmen wir als Beispiel die Treiber-Software für eine Grafikkarte. Anfangs genügt ein einfacher Treiber für einfache Ansprüche. Wir entwickeln eine Klasse, die den Code für den Treiber enthält und nach außen eine Schnittstelle anbietet, über die wir die Funktionalität des Treibers verwenden können. Letzteres ist der Typ des Treibers. Wir schreiben einige Anwendungen, welche die Treiberklasse verwenden. Daneben werden vielleicht auch von anderen Personen, die wir nicht kennen, Anwendungen erstellt, die unsere Treiberklasse verwenden. Alle Anwendungen greifen über dessen Schnittstelle beziehungsweise Typ auf den Treiber zu.

[4] Genau genommen ist das nur ein Spezialfall einer binären Methode. Es handelt sich auch dann um eine binäre Methode, wenn für mindestens zwei Eingangsparameter Argumente vom gleichen dynamischen Typ übergeben werden müssen, unabhängig von `this`. Es gibt einige objektorientierte Sprachen wie Ada und Eiffel, die das (durch dynamische Typprüfungen) unterstützen. In den meisten objektorientierten Sprachen sind binäre Methoden jedoch nur dadurch ausdrückbar, dass mindestens ein Eingangsparameter gleich dem Typ der Klasse sein muss, auch in Untertypen davon.

Mit der Zeit wird unser einfacher Treiber zu primitiv. Wir entwickeln einen neuen, effizienteren Treiber, der auch Eigenschaften neuerer Grafikkarten verwenden kann. Wir erben von der alten Klasse und lassen die Schnittstelle unverändert, abgesehen davon, dass wir neue Methoden hinzufügen. Nach obiger Definition ist der Typ der neuen Klasse ein Untertyp des alten Typs. Neue Treiber – das sind Objekte des Treibertyps – können überall verwendet werden, wo alte Treiber erwartet werden. Daher können wir in den vielen Anwendungen, die den Treiber bereits verwenden, den alten Treiber ganz einfach gegen den neuen austauschen, ohne die Anwendungen sonst irgendwie zu ändern. In diesem Fall haben wir Wiederverwendung in großem Umfang erzielt: Viele Anwendungen sind sehr einfach auf einen neuen Treiber umgestellt worden. Darunter sind auch Anwendungen, von deren Existenz wir nichts wissen. Das Beispiel können wir beliebig fortsetzen, indem wir immer wieder neue Varianten von Treibern schreiben und neue Anwendungen entwickeln oder bestehende Anwendungen anpassen, welche die jeweils neuesten Eigenschaften der Treiber nützen. Dabei kann es passieren, dass aus einer Treiberversion mehrere weitere Treiberversionen entwickelt werden, die nicht zueinander kompatibel sind. Folgendes Bild zeigt, wie die Treiberversionen nach drei Generationen aussehen könnten:



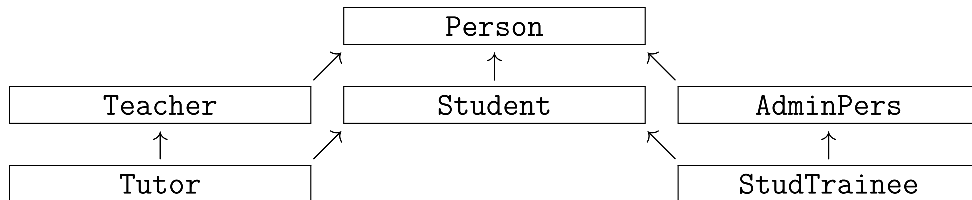
An dieser Struktur fällt Version 3b auf: Sie vereinigt die zwei inkompatiblen Vorgängerversionen 2a und 2b. Ein Untertyp kann mehrere Obertypen haben, die zueinander in keiner Untertypbeziehung stehen. Das ist ein Beispiel für Mehrfachvererbung, während in den anderen Fällen nur Einfachvererbung nötig ist. Diese Hierarchie kann in Java nur realisiert werden, wenn die Treiberschnittstellen Interfaces (keine Klassen) sind.

> *Faustregel:* Wir sollen auf Ersetzbarkeit achten, um Codewiederverwendung zwischen Versionen zu erreichen.

Die Wiederverwendung zwischen verschiedenen Versionen funktioniert nur dann gut, wenn die Schnittstellen bzw. Typen zwischen den Versionen *stabil* bleiben. Das heißt, eine neue Version darf die Schnittstellen nicht beliebig ändern, sondern nur so, dass die in Abschnitt 3.1.1 beschriebenen Bedingungen erfüllt sind. Im Wesentlichen kann die Schnittstelle also nur erweitert werden. Wenn die Aufteilung eines Programms in einzelne Objekte (also die Faktorisierung) gut ist, bleiben Schnittstellen normalerweise recht stabil.

> *Faustregel:* Schnittstellen sollen stabil bleiben. Eine gute Faktorisierung hilft dabei.

Das, was in obigem Beispiel für verschiedene Versionen funktioniert, ist auch innerhalb eines einzigen Programms nützlich, wie wir an einem modifizierten Beispiel sehen. Wir wollen ein Programm zur Verwaltung der Personen an einer Universität entwickeln. Die dafür verwendete Typstruktur könnte so aussehen:



Tutor_innen sind sowohl Lehrende als auch Studierende, Werkstudent_innen (`StudTrainee`) gehören zum Verwaltungspersonal (`AdminPers`) und sind Studierende. Wir benötigen im Programm eine Komponente, die Serienbriefe – Einladungen zu Veranstaltungen, etc. – an alle Personen adressiert. Für das Erstellen einer Anschrift benötigen wir nur Informationen aus der Klasse `Person`. Die entsprechende Methode muss nicht zwischen verschiedenen Arten von Personen unterscheiden, sondern funktioniert für jedes Objekt des deklarierten Typs `Person`, auch wenn es ein Objekt des dynamischen Typs `Tutor` ist. Diese Methode wird also für alle Arten von Personen (wieder)verwendet. Ebenso funktioniert eine Methode zum Ausstellen eines Zeugnisses für alle Objekte von `Student`, auch wenn es Tutor_innen oder Werkstudent_innen sind. Für dieses Beispiel müssen in Java ebenso Interfaces verwendet werden.

> *Faustregel:* Wir sollen auf Ersetzbarkeit achten, um interne Codewiederverwendung im Programm zu erzielen.

Solche Typstrukturen helfen, Auswirkungen nötiger Programmänderungen lokal zu halten. Ändern wir einen Typ, zum Beispiel `Student`, bleiben andere Typen, die nicht Untertypen von `Student` sind, unberührt. Von der Änderung betroffene Typen sind an der Typstruktur leicht erkennbar. Unter „betroffen" verstehen wir dabei, dass als Folge der Änderung möglicherweise weitere Änderungen in den betroffenen Programmteilen nötig sind. Die Änderung kann nicht nur diese Typen (Klassen oder Interfaces) selbst betreffen, sondern auch alle Programmstellen, die auf Objekte der Typen `Student`, `Tutor` oder `StudTrainee` zugreifen. Aber Programmteile, die auf Objekte des deklarierten Typs `Person` zugreifen, sollten von der Änderung auch dann nicht betroffen sein, wenn die Objekte tatsächlich vom dynamischen Typ `Student` sind. Diese Programmteile haben keinen Zugriff auf geänderte Objekteigenschaften.

> *Faustregel:* Wir sollen auf Ersetzbarkeit achten, um Programmänderungen lokal zu halten.

Falls bei der Programmänderung alle Schnittstellen unverändert bleiben, sind keine Programmstellen betroffen, an denen `Student` und dessen Untertypen verwendet werden. Lediglich diese Typen selbst sind betroffen. Auch daran können wir sehen, wie wichtig es ist, dass Schnittstellen und Typen stabil sind. Eine

Programmänderung führt möglicherweise zu vielen weiteren nötigen Änderungen, wenn dabei eine Schnittstelle geändert wird. Die Anzahl wahrscheinlich nötiger Änderungen hängt auch davon ab, wo in der Typstruktur die geänderte Schnittstelle steht. Eine Änderung ganz oben in der Struktur hat wesentlich größere Auswirkungen als eine Änderung ganz unten. Eine Schlussfolgerung aus diesen Überlegungen ist, dass wir möglichst nur von solchen Klassen ableiten sollen, deren Schnittstellen bereits – oft nach mehreren Refaktorisierungsschritten – recht stabil sind.

> *Faustregel:* Die Stabilität von Schnittstellen an der Wurzel der Typhierarchie ist wichtiger als an den Blättern. Wir sollen nur Untertypen von stabilen Obertypen bilden.

Aus obigen Überlegungen folgt auch, dass wir die Typen von Parametern möglichst allgemein halten sollen. Wenn in einer Methode von einem Parameter nur die Eigenschaften von `Person` benötigt werden, sollte der Parametertyp `Person` sein und nicht `StudTrainee`, auch wenn die Methode voraussichtlich nur mit Argumenten vom Typ `StudTrainee` aufgerufen wird. Wenn aber die Wahrscheinlichkeit hoch ist, dass nach einer späteren Programmänderung in der Methode vom Parameter auch Eigenschaften von `StudTrainee` benötigt werden, sollten wir gleich von Anfang an `StudTrainee` als Parametertyp verwenden, da nachträgliche Änderungen von Schnittstellen sehr teuer werden können.

> *Faustregel:* Wir sollen Parametertypen vorausschauend und möglichst allgemein wählen.

Trotz der Wichtigkeit stabiler Schnittstellen dürfen wir nicht bereits zu früh zu viel Zeit in den detaillierten Entwurf der Schnittstellen investieren. Zu Beginn haben wir häufig noch nicht genug Information, um stabile Schnittstellen zu erhalten. Schnittstellen werden trotz guter Planung oft erst nach einigen (wenigen) Refakorisierungsschritten stabil.

3.1.3 Dynamisches Binden

Bei Verwendung von Untertypen kann der dynamische Typ einer Variablen oder eines Eingangsparameters ein Untertyp des deklarierten Typs sein. Eine als `Person` deklarierte Variable kann etwa ein Objekt von `StudTrainee` enthalten. Häufig ist zur Übersetzungszeit der dynamische Typ nicht bekannt, das heißt, der dynamische Typ kann sich vom statischen Typ unterscheiden. Dann können Aufrufe einer Methode im Objekt, das in der Variablen steht, erst zur Laufzeit an die auszuführende Methode gebunden werden. In Java wird unabhängig vom deklarierten Typ immer die Methode ausgeführt, die im spezifischsten dynamischen Typ definiert ist. Dieser Typ entspricht der Klasse des Objekts in der Variablen.

Wir demonstrieren dynamisches Binden an einem kleinen Beispiel:

```
public class A {
    public String foo1() { return "foo1A"; }
    public String foo2() { return fooX(); }
    public String fooX() { return "foo2A"; }
}
public class B extends A {
    public String foo1() { return "foo1B"; }
    public String fooX() { return "foo2B"; }
}
public class DynamicBindingTest {
    public static void test(A x) {
        System.out.println(x.foo1());
        System.out.println(x.foo2());
    }
    public static void main(String[] args) {
        test(new A());
        test(new B());
    }
}
```

Die Ausführung von `DynamicBindingTest` liefert folgende Ausgabe:

```
foo1A
foo2A
foo1B
foo2B
```

Die ersten Zeilen sind einfach erklärbar: Nach dem Programmaufruf wird die Methode `main` ausgeführt, die `test` mit einem neuen Objekt von `A` als Argument aufruft. Diese Methode ruft zuerst `foo1` und dann `foo2` auf und gibt die Ergebnisse in den ersten beiden Zeilen aus. Dabei entspricht der deklarierte Typ `A` des Parameters `x` dem spezifischsten dynamischen Typ. Es werden daher `foo1` und `foo2` in `A` ausgeführt.

Der zweite Aufruf von `test` übergibt ein Objekt von `B` als Argument. Dabei ist `A` der deklarierte Typ von `x`, aber der dynamische Typ ist `B`. Wegen dynamischen Bindens werden diesmal `foo1` und `foo2` in `B` ausgeführt. Die dritte Zeile der Ausgabe enthält das Ergebnis des Aufrufs von `foo1` in einem Objekt von `B`.

Die letzte Zeile der Ausgabe lässt sich folgendermaßen erklären: Da die Klasse `B` die Methode `foo2` nicht überschreibt, wird `foo2` von `A` geerbt. Der Aufruf von `foo2` in `B` ruft `fooX` in der aktuellen Umgebung auf, das ist ein Objekt von `B`. Die Methode `fooX` liefert als Ergebnis die Zeichenkette `"foo2B"`, die in der letzten Zeile ausgegeben wird.

Bei dieser Erklärung müssen wir vorsichtig sein: Wir machen leicht den Fehler anzunehmen, dass `foo2` und daher auch `fooX` in `A` aufgerufen wird, da `foo2` ja nicht explizit in `B` steht. Tatsächlich wird aber `fooX` in `B` aufgerufen, da `B` der spezifischste Typ der Umgebung ist.

Dynamisches Binden ist mit `switch`-Anweisungen und geschachtelten `if`-Anweisungen verwandt. Wir betrachten als Beispiel eine Methode, die eine Anrede in einem Brief, deren Form auf konventionelle Weise über eine ganze Zahl bestimmt ist, in die Standardausgabe schreibt:

```
public void addressPerson(int form, String name) {
    switch(form) {
        case  1: System.out.print("Sehr geehrte Frau " + name);
                 break;
        case  2: System.out.print("Sehr geehrter Herr " + name);
                 break;
        default: System.out.print(name);
    }
}
```

In der objektorientierten Programmierung wird die Form der Anrede eher durch die Klassenstruktur zusammen mit dem Namen beschrieben:

```
public class Addressee {
    private String name;
    protected String name() {
        return name;
    }
    public void addressPerson() {
        System.out.print(name());
    }
    ... // Konstruktoren und weitere Methoden
}
public class FemaleAddressee extends Addressee {
    public void addressPerson() {
        System.out.print ("Sehr geehrte Frau " + name());
    }
}
public class MaleAddressee extends Addressee {
    public void addressPerson() {
        System.out.print ("Sehr geehrter Herr " + name());
    }
}
```

Durch dynamisches Binden wird automatisch die gewünschte Variante von `addressPerson()` aufgerufen. Statt einer `switch`-Anweisung wird also dynamisches Binden verwendet. Ein Vorteil des objektorientierten Ansatzes ist die bessere Lesbarkeit. Wir wissen anhand der Namen, wofür bestimmte Unterklassen von `Addressee` stehen. Die Zahlen 1 oder 2 bieten diese Information nicht. Außerdem ist die Form der Anrede mit dem auszugebenden Namen verknüpft, wodurch im Programm stets nur ein Objekt von `Addressee` anstatt einer ganzen Zahl und einem String verwaltet werden muss. Ein anderer Vorteil des

objektorientierten Ansatzes ist besonders wichtig: Wenn sich herausstellt, dass neben „Frau“ und „Herr“ noch weitere Formen von Anreden, etwa „Firma“, benötigt werden, können wir diese leicht durch Hinzufügen weiterer Klassen einführen. Es sind keine zusätzlichen Änderungen nötig. Insbesondere bleiben die Methodenaufrufe unverändert. Letzteres können wir durch Verwendung benannter Konstanten oder Enums statt der Zahlen nicht erreichen.

Auf den ersten Blick mag es scheinen, als ob der konventionelle Ansatz mit `switch`-Anweisung kürzer und auch einfach durch Hinzufügen einer Zeile änderbar wäre. Am Beginn der Programmentwicklung trifft das oft auch zu. Leider haben solche `switch`-Anweisungen die Eigenschaft, dass sie sich sehr rasch über das ganze Programm ausbreiten. Beispielsweise gibt es bald auch spezielle Methoden zur Ausgabe der Anrede in generierten e-Mails, abgekürzt in Berichten, oder über Telefon als gesprochener Text, jede Methode mit zumindest einer eigenen `switch`-Anweisung. Dann ist es schwierig, zum Einfügen der neuen Anredeform alle solchen `switch`-Anweisungen zu finden und noch schwieriger, diese Programmteile über einen längeren Zeitraum konsistent zu halten. Der objektorientierte Ansatz hat dieses Problem nicht, da alles auf die Klasse `Addressee` und ihre Unterklassen konzentriert ist. Es bleibt auch dann alles konzentriert, wenn zu `addressPerson()` weitere Methoden hinzukommen.

> *Faustregel:* Dynamisches Binden ist `switch`-Anweisungen und geschachtelten `if`-Anweisungen vorzuziehen.

3.2 Ersetzbarkeit und Objektverhalten

In Abschnitt 3.1.1 haben wir Bedingungen kennen gelernt, unter denen ein struktureller Typ Untertyp eines anderen ist. Für nominale Typen gilt daneben noch die Bedingung, dass der Untertyp explizit vom Obertyp abgeleitet sein muss. Die Erfüllung dieser Bedingungen wird vom Compiler überprüft. In Java und den meisten anderen Sprachen mit statischer Typprüfung werden sogar etwas strengere Bedingungen geprüft, die nicht für Untertypen, sondern z. B. für das Überladen von Methoden sinnvoll sind.

Jedoch sind alle prüfbaren Bedingungen nicht ausreichend, um die uneingeschränkte Ersetzbarkeit eines Objekts eines Obertyps durch ein Objekt eines Untertyps zu garantieren. Dazu müssen weitere Bedingungen hinsichtlich des Objektverhaltens erfüllt sein, die von einem Compiler nicht überprüft werden können. Wir müssen diese Bedingungen beim Programmieren selbst (ohne Werkzeugunterstützung) sicherstellen.

3.2.1 Client-Server-Beziehungen

Für die Beschreibung des Objektverhaltens ist es hilfreich, das Objekt aus der Sicht anderer Objekte, die auf das Objekt zugreifen, zu betrachten (siehe Abschnitt 1.3.3). Wir sprechen von *Client-Server-Beziehungen* zwischen Objekten. Einerseits sehen wir ein Objekt als *Server*, der anderen Objekten seine Dienste

zur Verfügung stellt. Andererseits ist ein Objekt ein *Client*, der Dienste anderer Objekte in Anspruch nimmt. Ein Objekt ist gleichzeitig Server und Client.

Für die Ersetzbarkeit von Objekten sind Client-Server-Beziehungen bedeutend. Ein Objekt ist gegen ein anderes ersetzbar, wenn das ersetzende Objekt als Server allen Clients zumindest dieselben Dienste anbietet wie das ersetzte Objekt. Um das gewährleisten zu können, brauchen wir eine Beschreibung der Dienste, also das Verhalten der Objekte.

Das *Objektverhalten* beschreibt, wie sich das Objekt beim Empfang einer Nachricht verhält, das heißt, was das Objekt beim Aufruf einer Methode macht. Diese Definition von Objektverhalten lässt etwas offen: Es ist unklar, wie exakt die Beschreibung des Verhaltens sein soll. Einerseits beschreibt die Signatur das Objekt nur sehr unvollständig. Eine genauere Beschreibung wäre wünschenswert. Andererseits enthält die Implementierung, also der Programmcode in der Klasse, oft zu viele Implementierungsdetails, die bei der Betrachtung des Verhaltens hinderlich sind. Im Programmcode gibt es meist keine Beschreibung, deren Detailiertheitsgrad zwischen dem der Signatur und dem der Implementierung liegt. Wir haben es beim Objektverhalten also mit einem abstrakten Begriff zu tun. Er wird vom Programmcode nicht notwendigerweise widergespiegelt.

Es hat sich bewährt, das Verhalten eines Objekts als einen Vertrag zwischen dem Objekt als Server und seinen Clients zu sehen – *Design-by-Contract.* Der Server muss diesen Vertrag ebenso einhalten wie jeder einzelne Client, in einigen Fällen auch die Gemeinschaft aller Clients zusammen. Generell sieht der Softwarevertrag folgendermaßen aus [25, 29]:

> *Vertrag:* Jeder Client kann Dienste des Servers in Anspruch nehmen, wenn die festgeschriebenen Bedingungen dafür erfüllt sind. Im Falle einer Inanspruchnahme setzt der Server die festgeschriebenen Maßnahmen und liefert dem Client ein Ergebnis, das die festgeschriebenen Bedingungen erfüllt.

Im einzelnen regelt der Vertrag für jeden vom Server angebotenen Dienst, also für jede aufrufbare Methode (unter der Annahme, dass auf Objektvariablen nur über Methoden des Servers zugegriffen wird), folgende Details:

Vorbedingung (Precondition): Für die Erfüllung der Vorbedingung einer Methode vor Ausführung der Methode ist jeder einzelne Client verantwortlich. Vorbedingungen beschreiben hauptsächlich, welche Eigenschaften die Argumente, mit denen die Methode aufgerufen wird, erfüllen müssen. Zum Beispiel muss ein bestimmtes Argument ein Array aufsteigend sortierter ganzen Zahlen im Wertebereich von 0 bis 99 sein. Vorbedingungen können auch den Zustand des Servers einbeziehen, soweit Clients diesen kennen. Z. B. ist eine Methode nur aufrufbar, wenn eine Objektvariable des Servers (etwa über einen Getter abfragbar) einen Wert größer 0 hat.

Nachbedingung (Postcondition): Für die Erfüllung der Nachbedingung einer Methode nach Ausführung der Methode ist der Server verantwortlich. Nachbedingungen beschreiben Eigenschaften des Methodenergebnisses und Änderungen beziehungsweise Eigenschaften des Objektzustands.

Als Beispiel betrachten wir eine Methode zum Einfügen eines Elements in eine Menge: Das Boolesche Ergebnis der Methode besagt, ob das Argument vor dem Aufruf bereits in der Menge enthalten war; am Ende muss es auf jeden Fall in der Menge sein. Diese Beschreibung kann man als Nachbedingung auffassen.

Invariante (Invariant): Für die Erfüllung von Invarianten auf Objektvariablen sowohl vor als auch nach Ausführung jeder Methode ist grundsätzlich der Server zuständig. Direkte Schreibzugriffe von Clients auf Variablen des Servers kann der Server aber nicht kontrollieren; dafür sind die Clients verantwortlich. Zum Beispiel darf das Guthaben auf einem Sparbuch nie kleiner 0 sein, egal welche Operationen darauf durchgeführt werden.

History-Constraint: Diese Bedingungen schränken die Entwicklung von Objekten im Laufe der Zeit ein. Wir unterscheiden zwei Unterarten:

Server-kontrolliert: Sie ähneln Invarianten, schränken aber zeitliche Veränderungen der Variableninhalte eines Objekts ein. Z. B. kann der ganzzahlige Wert einer Objektvariablen, die als Zähler verwendet wird, im Laufe der Zeit immer größer, aber niemals kleiner werden. Wie bei Invarianten ist für die Einhaltung der Server zuständig. Wenn jedoch Clients die betroffenen Variablen direkt schreiben können, sind auch die Clients verantwortlich.

Client-kontrolliert: Über History-Constraints ist auch die Reihenfolge von Methodenaufrufen einschränkbar. Beispielsweise ist eine Methode namens `initialize` in jedem Objekt nur einmal aufrufbar, davor sind keine Aufrufe anderer Methoden erlaubt. Methodenaufrufe erfolgen durch Clients. Nur Clients können die Aufrufreihenfolge bestimmen und sind für die Einhaltung der Bedingungen verantwortlich. Manchmal ist es gar nicht möglich, die Aufrufreihenfolge im Objektzustand abzubilden, z. B. wenn `initialize` in einem durch Kopieren (`clone`) erzeugten Objekt ausgeführt werden soll; der kopierte Objektzustand sagt darüber ja nichts aus.

History-Constraints werden derzeit in Softwareverträgen nicht so häufig verwendet wie die anderen Arten von Zusicherungen, hauptsächlich weil die Regeln hinter ihnen weniger einheitlich und nicht so leicht verständlich sind. Dennoch steckt in ihnen sehr viel Potenzial.

Vorbedingungen, Nachbedingungen, Invarianten und History-Constraints sind verschiedene Arten von *Zusicherungen* (*Assertions*).

Zum Teil sind Vorbedingungen und Nachbedingungen bereits in der Objektschnittstelle in Form von Parameter- und Ergebnistypen von Methoden beschrieben. Typkompatibilität wird vom Compiler überprüft. In der Programmiersprache Eiffel gibt es Sprachkonstrukte, mit denen man komplexere Zusicherungen schreiben kann [28]. Diese werden zur Laufzeit überprüft. Sprachen wie Java unterstützen überhaupt keine Zusicherungen – abgesehen von trivialen `assert`-Anweisungen, die sich aber nur beschränkt zur Beschreibung von

Verträgen eignen. Sogar in Eiffel sind viele sinnvolle Zusicherungen nicht direkt ausdrückbar. In diesen Fällen können und sollen wir Zusicherungen als Kommentare in den Programmcode schreiben und händisch überprüfen. Umgekehrt sollen wir fast jeden Kommentar als Zusicherung lesen.

Für Interessierte, nicht Prüfungsstoff. Ein Beispiel mit Zusicherungen in Eiffel:

```
class ACCOUNT feature {ANY}

    balance: Integer
    maxOverdraft: Integer

    payIn (amount: Integer) is
        require amount >= 0
        do balance := balance + amount
        ensure balance = old balance + amount
    end -{}- payIn

    payOut (amount: Integer) is
        require amount >= 0;
                balance + maxOverdraft >= amount
        do balance := balance - amount
        ensure balance = old balance - amount
    end -{}- payOut

    invariant balance >= -maxOverdraft
end -{}- class ACCOUNT
```

Bei jeder Methode[5] kann vor der eigentlichen Implementierung (`do`-Klausel) eine Vorbedingung (`require`-Klausel) und danach eine Nachbedingung (`ensure`-Klausel) stehen. Invarianten stehen ganz am Ende der Klasse, History-Constraints werden nicht unterstützt. In jeder Zusicherung steht eine Liste von implizit durch Und verknüpften Booleschen Ausdrücken. Sie werden zur Laufzeit zu Ja oder Nein ausgewertet. Bei Auswertung einer Zusicherung zu Nein wird eine Exception geworfen. In Nachbedingungen ist die Bezugnahme auf Variablen- und Parameterwerte zum Zeitpunkt des Methodenaufrufs erlaubt. So bezeichnet `old balance` den Wert von `balance` zum Zeitpunkt des Methodenaufrufs.

Diese Klasse sollte bis auf einige syntaktische und semantische Details selbsterklärend sein. Die Klausel `feature {ANY}` besagt, dass die danach folgenden Methodendefinitionen sowie automatisch generierte Getter für die Variablen (deren Aufrufe syntaktisch wie lesende Variablenzugriffe ausschauen) überall im Programm sichtbar sind; Variablen sind nur im eigenen Objekt schreibbar. Nach dem Schlüsselwort `end` und einem (in unserem Fall leeren) Kommentar kann zur besseren Lesbarkeit der Name der Methode oder der Klasse folgen. **Ende des Einschubs für Interessierte**

[5] Genau genommen kennt Eiffel keine Methoden. Stattdessen wird streng zwischen Prozeduren (die Seiteneffekte haben und keine Ergebnisse zurückgeben) und Funktionen (die Ergebnisse zurückgeben, aber keine Seiteneffekte haben) unterschieden. Durch die Freiheit von Seiteneffekten können Funktionen in Zusicherungen aufgerufen werden, ohne den eigentlichen Programmablauf zu stören.

Hier ist ein Java-Beispiel für Kommentare als Zusicherungen:

```
public class Account {
    private long balance, maxOverdraft;
    // balance >= -maxOverdraft

    // add amount to balance; amount >= 0
    public void payIn(long amount) {
        balance = balance + amount;
    }

    // subtract amount from balance;
    // amount >= 0; disposable() >= amount
    public void payOut(long amount) {
        balance = balance - amount;
    }
    public long disposable() { return balance+maxOverdraft; }
}
```

Beachten Sie, dass Kommentare in der Praxis (so wie im Beispiel) oft keine expliziten Aussagen darüber enthalten, ob und wenn Ja, um welche Arten von Zusicherungen es sich dabei handelt. Solche Informationen sind aus dem Kontext herauslesbar. Die erste Kommentarzeile kann nur eine Invariante darstellen, da allgemein gültige (das heißt, nicht auf einzelne Methoden eingeschränkte) Beziehungen zwischen Variablen hergestellt werden. Die zweite Kommentarzeile enthält gleich zwei verschiedene Arten von Zusicherungen: Die Aussage „add amount to balance“ bezieht sich darauf, wie die Ausführung der auf den Kommentar folgenden Methode den Objektzustand verändert. Das kann nur eine Nachbedingung sein. Nachbedingungen lesen sich häufig wie Beschreibungen dessen, was eine Methode tut. Aber die Aussage „amount ≥ 0“ bezieht sich auf eine erwartete Eigenschaft eines Parameters und ist daher eine Vorbedingung von `payIn`. Mit derselben Begründung ist „subtract amount from balance“ eine Nachbedingung und sind „amount ≥ 0“ und „disposable() $\geq$ amount“ Vorbedingungen von `payOut`. Die Methode `disposable` ist nötig, damit Clients die Bedingung prüfen können; die Objektvariablen sind ja nicht zugreifbar.

Nebenbei bemerkt sollen Geldbeträge wegen möglicher Rundungsfehler niemals durch Gleitkommazahlen dargestellt werden. Verwenden Sie lieber wie in obigem Beispiel ausreichend große ganzzahlige Typen oder noch besser spezielle Typen für Geldbeträge. Aufgrund komplexer Rundungsregeln sind in der Praxis fast immer spezielle Typen nötig.

Bisher haben wir die Begriffe Typ (bzw. Schnittstelle) und Signatur (bei nominalen Typen zusammen mit Namen) als im Wesentlichen gleichbedeutend angesehen. Ab jetzt betrachten wir Zusicherungen, unabhängig davon, ob sie durch eigene Sprachkonstrukte oder in Kommentaren beschrieben sind, als zum Typ (und zur Schnittstelle) eines Objekts gehörend. Ein nominaler Typ besteht demnach aus

- dem Namen einer Klasse, eines Interfaces oder elementaren Typs,
- der entsprechenden Signatur
- und den dazugehörenden Zusicherungen.

Der Name sollte eine kurze Beschreibung des Zwecks der Objekte des Typs geben und der Abstraktion dienen. Die Signatur enthält alle vom Compiler überprüfbaren Bestandteile des Vertrags zwischen Clients und Server. Zusicherungen enthalten alle über die Abstraktion durch Namen hinausgehenden Vertragsbestandteile, die nicht vom Compiler überprüft werden. Wir gehen hier davon aus, dass Zusicherungen Kommentare und alle Kommentare Zusicherungen sind.

In Abschnitt 3.1.2 haben wir gesehen, dass Typen wegen der besseren Wartbarkeit stabil sein sollen. Solange eine Programmänderung den Typ der Klasse unverändert lässt oder nur auf unbedenkliche Art und Weise erweitert (siehe Abschnitt 3.2.2), hat die Änderung keine Auswirkungen auf andere Programmteile. Das betrifft auch Zusicherungen. Eine Programmänderung kann sich sehr wohl auf andere Programmteile auswirken, wenn dabei eine Zusicherung (= ein Kommentar) geändert wird.

> *Faustregel:* Zusicherungen sollen stabil bleiben. Das ist für Zusicherungen in Typen nahe an der Wurzel der Typhierarchie ganz besonders wichtig.

Wir können die Genauigkeit der Zusicherungen selbst bestimmen. Dabei sind Auswirkungen der Zusicherungen zu beachten: Clients dürfen sich nur auf das verlassen, was in der Signatur und in den Zusicherungen vom Server zugesagt wird, und der Server auf das, was von den Clients zugesagt wird. Beispiele dafür folgen in Abschnitt 3.2.2. Sind die Zusicherungen sehr genau, können sich die Clients auf viele Details des Servers verlassen und auch der Server kann von den Clients viel verlangen. Aber Programmänderungen werden mit größerer Wahrscheinlichkeit dazu führen, dass Zusicherungen geändert werden müssen, wovon alle Clients betroffen sind. Steht hingegen in den Zusicherungen nur das Nötigste, sind Clients und Server relativ unabhängig voneinander. Der Typ ist bei Programmänderungen eher stabil. Aber vor allem die Clients dürfen sich nur auf Weniges verlassen. Wenn keine Zusicherungen gemacht werden, dürfen sich Clients auf nichts verlassen, was nicht aus der Signatur folgt.

> *Faustregel:* Zur Verbesserung der Wartbarkeit sollen Zusicherungen keine unnötigen Details festlegen.

Zusicherungen bieten viele Möglichkeiten zur Gestaltung der Client-Server-Beziehungen. Aus Gründen der Wartbarkeit sollen wir Zusicherungen aber nur dort einsetzen, wo tatsächlich Informationen benötigt werden, die über jene in der Signatur hinausgehen. Wir sollen Zusicherungen einsetzen, um den Klassenzusammenhalt zu maximieren und die Objektkopplung zu minimieren. In obigem Konto-Beispiel wäre es besser, die Vorbedingung „disposable() $\geq$ amount“ wegzulassen und die Einhaltung der Bedingung direkt in der Implementierung

von `payOut` durch eine `if`-Anweisung zu überprüfen. Dann ist nicht der Client für die Einhaltung der Bedingung verantwortlich, sondern der Server.

Die Vermeidung unnötiger Zusicherungen zielt darauf ab, dass Client und Server als relativ unabhängig voneinander angesehen werden können. Die Wartbarkeit wird dadurch natürlich nur dann verbessert, wenn diese Unabhängigkeit tatsächlich gegeben ist. Ein äußerst unerwünschter Effekt ergibt sich, wenn wir Zusicherungen (= nötige Kommentare) einfach aus Bequemlichkeit nicht in den Programmcode schreiben, der Client aber trotzdem bestimmte Eigenschaften vom Server erwartet (oder umgekehrt), also beispielsweise implizit voraussetzt, dass jeder Einzahlungsbetrag positiv ist. In diesem Fall haben wir die Abhängigkeiten zwischen Client und Server nur versteckt. Wegen der Abhängigkeiten können Programmänderungen zu unerwarteten Fehlern führen, die wir nur schwer finden, da die Abhängigkeiten nicht offensichtlich sind. Es sollen alle verwendeten Zusicherungen explizit im Programmcode stehen, außer solchen, die einschlägig geschulte Personen als ohnehin immer gültig betrachten. Andererseits sollen Client und Server so unabhängig wie möglich bleiben.

> *Faustregel:* Alle von geschulten Personen nicht in jedem Fall erwarteten, aber vorausgesetzten Eigenschaften sollen explizit als Zusicherungen im Programm stehen.

Sprechende Namen sagen viel darüber aus, wofür Typen und Methoden gedacht sind. Namen implizieren damit die wichtigsten Zusicherungen. Beispielsweise wird eine Methode `insert` in einem Objekt von `Set` ein Element zu einer Menge hinzufügen. Darauf werden sich Clients verlassen, auch wenn dieses Verhalten nicht durch explizite Kommentare spezifiziert ist. Trotzdem schadet es nicht, wenn das Verhalten zusätzlich als Kommentar beschrieben ist, da Kommentare den Detailiertheitsgrad viel besser angeben können als aus den Namen hervorgeht. Kommentare und Namen müssen in Einklang zueinander stehen.

Allgemein erwartete Eigenschaften gelten auch dann, wenn Namen und Kommentare nichts darüber aussagen. Diese Eigenschaften werden immer angenommen, außer Namen oder Kommentare besagen das Gegenteil. So darf keine Methode eine Datenstruktur zerstören oder beobachtbar verändern, wenn Namen oder Kommentare das nicht implizieren.

Meist finden wir bei Klassen und Interfaces Kommentare, die den dahinter stehenden abstrakten Datentyp erläutern und Beispiele für die Anwendung geben. Solche Kommentare sind häufig nicht direkt einzelnen Arten von Zusicherungen zuordenbar, hängen aber doch mit den Zusicherungen zusammen. Im Wesentlichen geben sie einen Kontext vor, in dem die Zusicherungen bei den Methoden und Variablendeklarationen zu verstehen sind. Abstrakte Datentypen abstrahieren über die reale Welt. Beschreibungen der abstrakten Datentypen legen auch die Beziehung zur realen Welt fest und klären, welche Aspekte der realen Welt gedanklich in die Softwarewelt übertragen werden. Implizit ergeben sich dadurch Zusicherungen: Die aus der realen Welt übernommenen Eigenschaften dürfen nicht verletzt werden. Sie gelten auch dann, wenn sie nicht explizit als Zusicherungen angeschrieben sind.

3.2.2 Untertypen und Verhalten

Zusicherungen müssen in Untertypen beachtet werden. Auch für Zusicherungen gilt das Ersetzbarkeitsprinzip bei der Feststellung, ob ein Typ Untertyp eines anderen Typs ist. Neben den Bedingungen, die wir in Abschnitt 3.1 kennen gelernt haben, müssen folgende Bedingungen gelten, damit ein Typ U Untertyp eines Typs T ist [24]:

Vorbedingung: Vorbedingungen auf einer Methode in T müssen Vorbedingungen auf der entsprechenden Methode in U implizieren. Vorbedingungen in Untertypen können schwächer, dürfen aber nicht stärker sein als entsprechende Vorbedingungen in Obertypen. Der Grund liegt darin, dass ein Aufrufer der Methode, der nur T kennt, nur die Erfüllung der Vorbedingungen in T sicherstellen kann, auch wenn die Methode tatsächlich in U statt T aufgerufen wird. Daher müssen die Vorbedingung in U automatisch erfüllt sein, wenn sie in T erfüllt sind. Werden Vorbedingungen in U aus T übernommen, können sie mittels Oder-Verknüpfungen schwächer werden. Ist eine Vorbedingung in T zum Beispiel „$x > 0$", kann sie in U auch „$x > 0$ oder $x = 0$", also abgekürzt „$x \geq 0$" lauten.

Nachbedingung: Nachbedingungen auf einer Methode in U müssen Nachbedingungen auf der entsprechenden Methode in T implizieren. Das heißt, Nachbedingungen in Untertypen können stärker, dürfen aber nicht schwächer sein als entsprechende Nachbedingungen in Obertypen. Der Grund liegt darin, dass ein Aufrufer der Methode, der nur T kennt, sich auf die Erfüllung der Nachbedingungen in T verlassen kann, auch wenn die Methode tatsächlich in U statt T aufgerufen wird. Daher müssen die Nachbedingungen in T automatisch erfüllt sein, wenn ihre Entsprechungen in U erfüllt sind. Werden Nachbedingungen in U aus T übernommen, können sie mittels Und-Verknüpfungen stärker werden. Lautet eine Nachbedingung in T zum Beispiel „`result` > 0", kann sie in U auch „`result` > 0 und `result` > 2", also „`result` > 2" sein.

Invariante: Invarianten in U müssen Invarianten in T implizieren. Das heißt, Invarianten in Untertypen können stärker, dürfen aber nicht schwächer sein als Invarianten in Obertypen. Der Grund liegt darin, dass ein Client, der nur T kennt, sich auf die Erfüllung der Invarianten in T verlassen kann, auch wenn tatsächlich ein Objekt von U statt einem von T verwendet wird. Der Server kennt seinen eigenen spezifischsten Typ, weshalb das Ersetzbarkeitsprinzip aus der Sicht des Servers nicht erfüllt zu sein braucht. Die Invarianten in T müssen automatisch erfüllt sein, wenn sie in U erfüllt sind. Wenn Invarianten in U aus T übernommen werden, können sie, wie Nachbedingungen, mittels Und-Verknüpfungen stärker werden.

Die Begründung geht davon aus, dass Objektvariablen nicht von außen verändert werden. Ist dies doch der Fall, so müssen Invarianten, die sich auf von außen änderbare Variablen beziehen, in U und T übereinstimmen. Beim Schreiben einer solchen Variable muss die Invariante vom Client

überprüft werden, was dem generellen Konzept widerspricht. Außerdem kann ein Client die Invariante gar nicht überprüfen, wenn in der Bedingung vorkommende Variablen und Methoden nicht öffentlich zugänglich sind. Daher sollen Objektvariablen nicht von außen änderbar sein.

Server-kontrollierter History-Constraint: Dafür gilt das Gleiche wie für Invarianten. Es ist jedoch nicht so einfach, von stärkeren oder schwächeren Bedingungen zu sprechen, da viel von der konkreten Formulierung der Bedingungen abhängt. Einfacher und klarer ist es, die Konsequenzen gegenüberzustellen. Für alle Objektzustände x und y in U und T (wobei ein Objektzustand die Werte aller gemeinsamen Variablen der Objekte von U und T widerspiegelt) soll gelten: Wenn Server-kontrollierte History-Constraints in T ausschließen, dass ein Objekt von T im Zustand x durch Veränderungen im Laufe der Zeit in den Zustand y kommt, dann müssen auch Server-kontrollierte History-Constraints in U ausschließen, dass ein Objekt von U im Zustand x durch Veränderungen im Laufe der Zeit in den Zustand y kommt. Einschränkungen auf T müssen also auch auf U gelten. Damit wird sichergestellt, dass ein Client sich auch dann auf die ihm bekannte Einschränkung in T verlassen kann, wenn statt einem Objekt von T eines von U verwendet wird. Es ist möglich, dass U die Entwicklung der Zustände stärker einschränkt als T, solange Clients betroffene Variablen nicht von außen schreiben können. Werden Variablen von außen geschrieben, müssen auch Clients für die Einhaltung der Server-kontrollierten History-Constraints sorgen, und die Bedingungen in U und T müssen übereinstimmen.

Client-kontrollierter History-Constraint: Dafür gilt das Gleiche wie für Vorbedingungen, jedoch bezogen auf Einschränkungen in der Reihenfolge von Methodenaufrufen. Eine Reihenfolge von Methodenaufrufen heißt *Trace,* die meist unendlich große Menge aller möglichen (= erlaubten) Traces ist ein *Trace-Set.* Jede entsprechend T erlaubte Aufrufreihenfolge muss auch entsprechend U erlaubt sein. Es ist jedoch möglich, dass U mehr Aufrufreihenfolgen erlaubt als T, wodurch die Einschränkungen in U schwächer sind als in T. Das durch Client-kontrollierte History-Constraints in T beschriebene Trace-Set muss also eine Teilmenge des durch Client-kontrollierte History-Constraints in U beschriebenen Trace-Sets sein. Wenn die Clients an ein Objekt Nachrichten in einer durch T erlaubten Reihenfolge schicken, so ist sichergestellt, dass das Objekt vom Typ U die entsprechenden Methoden auch in dieser Reihenfolge ausführen kann. Wir müssen bei der Überprüfung Client-kontrollierter History-Constraints im Allgemeinen die *Menge aller Clients* betrachten, nicht nur einen einzelnen Client, da der Server ja alle Methodenaufrufe nur in eingeschränkter Reihenfolge ausführen kann, nicht nur die Aufrufe, die von einem einzelnen Client kommen.

Im Prinzip lassen sich obige Bedingungen auch formal überprüfen. In Programmiersprachen wie Eiffel, in denen Zusicherungen formal definiert sind, wird

das tatsächlich (hinsichtlich Untertypbeziehungen vom Compiler) gemacht, abgesehen davon, dass Eiffel keine History-Constraints kennt. Aber bei Verwendung anderer Programmiersprachen sind Zusicherungen meist nicht formal, sondern nur umgangssprachlich als Kommentare gegeben. Unter diesen Umständen ist keine formale Überprüfung möglich. Daher müssen wir beim Programmieren alle nötigen Überprüfungen per Hand durchführen. Es muss gelten, dass

- obige Bedingungen für Untertypbeziehungen eingehalten werden,
- die Server alle Nachbedingungen, Invarianten und Server-kontrollierten History-Constraints erfüllen und nur voraussetzen, was in Vorbedingungen, Invarianten und History-Constraints festgelegt ist
- die Clients in Aufrufen alle Vorbedingungen und Client-kontrollierten History-Constraints erfüllen und nur voraussetzen, was durch Nachbedingungen, Invarianten und History-Constraints zugesichert wird.

Es kann sehr aufwändig sein, alle solchen Überprüfungen vorzunehmen. Einfacher geht es, wenn wir während der Code-Erstellung und bei Änderungen stets an die einzuhaltenden Bedingungen denken, die Überprüfungen also nebenbei machen. Wichtig ist darauf zu achten, dass die Zusicherungen unmissverständlich formuliert sind. Nach Änderung einer Zusicherung ist die Überprüfung besonders schwierig. Die Änderung einer Zusicherung ohne gleichzeitige Änderung *aller* betroffenen Programmteile ist eine häufige Fehlerursache in Programmen.

> *Faustregel:* Zusicherungen sollen unmissverständlich formuliert sein und während der Programmentwicklung und Wartung ständig bedacht werden.

Betrachten wir ein Beispiel:[6]

```
public class Set {
    public void insert(int x) {
        // inserts x into set iff not already there;
        // x is in set immediately after invocation
        ...;
    }
    public boolean inSet(int x) {
        // returns true if x is in set, otherwise false
        ...;
    }
}
```

Die Methode `insert` fügt eine ganze Zahl genau dann („iff“ ist eine übliche Abkürzung für „if and only if“) in ein Objekt von `Set` ein, wenn sie noch nicht

[6] Je nach Programmierstil stehen Zusicherungen vor oder nach Variablendeklarationen sowie vor oder nach Köpfen von Methoden oder Konstruktoren. In der Regel ist leicht erkennbar, wo sie dazugehören. Innerhalb eines Programms sollte der Stil möglichst einheitlich sein.

in dieser Menge ist. Unmittelbar nach Aufruf der Methode ist die Zahl in jedem Fall in der Menge. Die Methode `inSet` stellt fest, ob eine Zahl in der Menge ist oder nicht. Dieses Verhalten der Objekte von `Set` ist durch die Zusicherungen in den Kommentaren festgelegt. Wenn wir den Inhalt dieser Beschreibungen von Methoden genauer betrachten, sehen wir, dass es sich dabei um Nachbedingungen handelt. Da Nachbedingungen festlegen, was sich ein Client vom Aufruf einer Methode erwartet, lesen sich Nachbedingungen oft tatsächlich wie Beschreibungen von Methoden.

Folgende Klasse unterscheidet sich von `Set` nur durch einen zusätzlichen Server-kontrollierten History-Constraint:

```
public class SetWithoutDelete extends Set {
    // elements in the set always remain in the set
}
```

Eine Zahl, die einmal in der Menge war, soll stets in der Menge bleiben. Offensichtlich ist `SetWithoutDelete` ein Untertyp von `Set`, da nur ein vom Server kontrollierter History-Constraint dazugefügt wird, welcher die zukünftige Entwicklung des Objektzustands gegenüber `Set` einschränkt.

Sehen wir uns eine kurze Codesequenz für einen Client an:

```
Set s = new Set();
s.insert(41);
doSomething(s);
if (s.inSet(41)) { doSomeOtherThing(s); }
else { doSomethingElse(); }
```

Während der Ausführung von `doSomething` könnte `s` verändert werden. Es ist nicht ausgeschlossen, dass 41 dabei aus der Menge gelöscht wird, da die Nachbedingung von `insert` in `Set` ja nur zusichert, dass 41 unmittelbar nach dem Aufruf von `insert` in der Menge ist. Bevor wir die Methode `doSomeOtherThing` aufrufen (von der wir annehmen, dass sie ihren Zweck nur erfüllt, wenn 41 in der Menge ist), stellen wir sicher, dass 41 tatsächlich in der Menge ist. Dies geschieht durch Aufruf von `inSet`.

Verwenden wir ein Objekt von `SetWithoutDelete` anstatt einem von `Set`, ersparen wir uns den Aufruf von `inSet`. Wegen der stärkeren Zusicherung ist 41 sicher in der Menge:

```
SetWithoutDelete s = new SetWithoutDelete();
s.insert(41);
doSomething(s);
doSomeOtherThing(s); // s.inSet(41) returns true
```

Von diesem kleinen Vorteil von `SetWithoutDelete` dürfen wir uns nicht dazu verleiten lassen, generell starke Einschränkungen in Zusicherungen zu verwenden. Solche Einschränkungen erschweren die Wartung (siehe Abschnitt 3.2.1). Als triviales Beispiel können wir `Set` leicht um eine Methode `delete` (zum Löschen einer Zahl aus der Menge) erweitern:

```
public class SetWithDelete extends Set {
    public void delete(int x) {
        // deletes x from the set if it is there
        ...;
    }
}
```

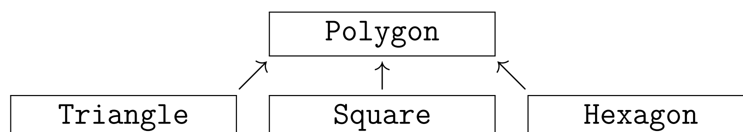
Aber `SetWithoutDelete` können wir, wie der Klassenname schon sagt, nicht auf diese Art erweitern. Jede vernünftige Nachbedingung von `delete` muss in Konflikt zum History-Constraint stehen. Wir dürfen nicht zu früh festlegen, dass es kein `delete` gibt, nur weil wir es gerade nicht brauchen. Invarianten wie in `SetWithoutDelete` sind nur sinnvoll, wenn wir wirklich darauf angewiesen sind. Andernfalls verbauen wir uns Wiederverwendungsmöglichkeiten.

Kommentare als Zusicherungen setzen voraus, dass wir Untertypbeziehungen explizit deklarieren, also nominale Typen verwenden. Damit bringen wir den Compiler dazu, beliebige weitere Bedingungen (gegenüber denen in Abschnitt 3.1.1) für eine Untertypbeziehung vorauszusetzen. Beispielsweise müssen wir explizit angeben, dass `SetWithoutDelete` ein Untertyp von `Set` ist, da sich diese Klassen für einen Compiler nur im Namen und in Kommentaren unterscheiden, deren Bedeutung der Compiler nicht kennt. Andernfalls wäre ein Objekt von `Set` auch verwendbar, wo eines von `SetWithoutDelete` erwartet wird. Es soll auch keine Untertypbeziehung zwischen `SetWithoutDelete` und `SetWithDelete` bestehen, obwohl dafür alle Bedingungen aus Abschnitt 3.1.1 erfüllt sind. Sonst wäre ein Objekt von `SetWithDelete` verwendbar, wo ein Objekt von `SetWithoutDelete` erwartet wird. Daher sind in vielen objektorientierten Sprachen Untertypen und Vererbung zu einem Konstrukt vereint: Vererbungsbeziehungen schließen zufällige Untertypbeziehungen aus, und wo eine Untertypbeziehung besteht, ist oft (nicht immer) auch Codevererbung sinnvoll.

3.2.3 Abstrakte Klassen

Klassen, die wir bis jetzt betrachtet haben, dienen der Beschreibung der Struktur ihrer Objekte, der Erzeugung und Initialisierung neuer Objekte und der Festlegung des spezifischsten Typs der Objekte. Im Zusammenhang mit Untertypen ist oft nur eine der Aufgaben nötig, nämlich die Festlegung des Typs. Das ist dann der Fall, wenn im Programm keine Objekte der Klasse selbst erzeugt werden sollen, sondern nur Objekte von Unterklassen. Aus diesem Grund unterstützen viele objektorientierte Sprachen *abstrakte Klassen*, von denen keine Objekte erzeugt werden können. Interfaces in Java sind eine besondere Form abstrakter Klassen, die keine Objektvariablen beschreiben, in der Fachliteratur häufig *Trait* genannt. Abgesehen davon gilt das hier gesagte auch für Interfaces.

Nehmen wir als Beispiel folgende Klassenstruktur:



Jede Unterklasse von Polygon beschreibt ein z. B. am Bildschirm darstellbares Vieleck. Polygon selbst beschreibt keine bestimmte Anzahl von Ecken, sondern fasst nur die Menge aller Vielecke zusammen. Wenn wir eine Liste unterschiedlicher Vielecke benötigen, werden wir den Typ der Vielecke in der Liste mit Polygon festlegen, obwohl in der Liste tatsächlich nur Objekte von Triangle, Square und Hexagon vorkommen. Es werden keine Objekte der Klasse Polygon selbst benötigt, sondern nur Objekte der Unterklassen. Polygon ist ein typischer Fall einer abstrakten Klasse.

In Java sieht die abstrakte Klasse etwa so aus:

```
public abstract class Polygon {
    public abstract void draw();
        // draw a polygon on the screen
}
```

Da die Klasse abstrakt ist, ist die Ausführung von new Polygon() nicht zulässig. Aber Unterklassen sind von Polygon ableitbar. Jede Unterklasse muss eine Methode draw enthalten, da diese Methode in Polygon deklariert ist. Genaugenommen ist draw als abstrakte Methode deklariert, als Signatur mit einem Kommentar als Zusicherung. Wir brauchen keine Implementierung in Polygon, da diese Methode ohnehin nicht ausgeführt wird; es gibt ja keine Objekte der Klasse Polygon, wohl aber Objekte des Typs Polygon. Nicht-abstrakte Unterklassen, das sind *konkrete Klassen*, müssen Implementierungen für abstrakte Methoden bereitstellen, diese also überschreiben. Abstrakte Unterklassen müssen abstrakte Methoden nicht überschreiben. Neben Variablen (außer in Java-Interfaces) und abstrakten Methoden dürfen abstrakte Klassen auch konkrete (also implementierte) Methoden enthalten, die wie üblich vererbt werden.

Die konkrete Klasse Triangle könnte so aussehen:

```
public class Triangle extends Polygon {
    public void draw() {
        // draw a triangle on the screen
        ...;
    }
}
```

Auch Square und Hexagon müssen die Methode draw implementieren.

So wie in diesem Beispiel kommt es vor allem in gut faktorisierten Programmen häufig vor, dass der Großteil der Implementierungen von Methoden in Klassen steht, die keine Unterklassen haben. Abstrakte Klassen sind eher stabil als konkrete. Zur Verbesserung der Wartbarkeit werden neue Klassen vor allem von stabilen Klassen abgeleitet. Außerdem werden möglichst stabile Typen für Parameter und Variablen verwendet. Da es leichter ist, abstrakte Klassen stabil zu halten, sind wir gut beraten, hauptsächlich solche Klassen, in Java vor allem Interfaces, für Parameter- und Variablentypen zu verwenden.

> *Faustregel:* Es ist empfehlenswert, als Obertypen und Parametertypen hauptsächlich abstrakte Klassen (in Java vor allem Interfaces) zu verwenden.

Parametertypen sollen keine Bedingungen an Argumente stellen, die nicht benötigt werden. Konkrete Klassen legen aber oft zahlreiche Bedingungen in Form von Zusicherungen und Methoden in der Schnittstelle fest. Dieser Konflikt ist lösbar, indem für die Typen der Parameter nur abstrakte Klassen verwendet werden. Es ist ja leicht, zu jeder konkreten Klasse eine oder mehrere abstrakte Klassen als Obertypen zu schreiben, die die benötigten Bedingungen möglichst genau angeben. Damit werden unnötige Abhängigkeiten vermieden.

3.3 Vererbung versus Ersetzbarkeit

Vererbung ist im Grunde einfach: Von einer Oberklasse wird scheinbar, meist nicht wirklich, eine Kopie angelegt, die durch Erweitern und Überschreiben abgeändert wird. Die resultierende Klasse ist die Unterklasse. Betrachten wir nur Vererbung und ignorieren Einschränkungen durch Untertypbeziehungen, haben wir vollkommene Freiheit in der Abänderung der Oberklasse. Vererbung ist zur direkten Wiederverwendung von Code einsetzbar und damit auch unabhängig vom Ersetzbarkeitsprinzip sinnvoll. Wir wollen zunächst einige Arten von Beziehungen zwischen Klassen aus Abschnitt 1.4.3 vergleichen und dann die Bedeutungen dieser Beziehungen für die Codewiederverwendung untersuchen.

3.3.1 Reale Welt, Vererbung, Ersetzbarkeit

In der objektorientierten Softwareentwicklung begegnen wir zumindest drei verschiedenen Arten von Beziehungen zwischen Klassen [22]:

Untertypbeziehungen: Diese Beziehungen, die auf dem Ersetzbarkeitsprinzip beruhen, haben wir bereits untersucht.

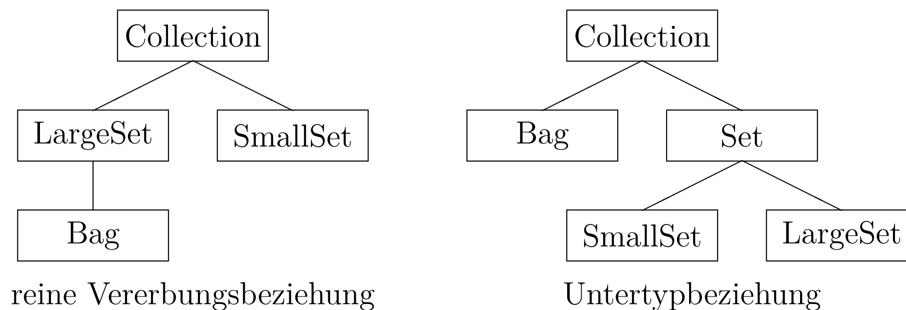
Vererbungsbeziehungen: Dabei entstehen neue Klassen durch Abänderung bestehender Klassen. Es ist nicht nötig, aber wünschenswert, dass dabei Code aus der Oberklasse in der Unterklasse direkt wiederverwendet wird. Für reine Vererbung ist das Ersetzbarkeitsprinzip irrelevant.

Beziehungen in der realen Welt: Bereits in frühen Entwicklungsphasen kristallisieren sich abstrakte Einheiten heraus, die später zu Klassen weiterentwickelt werden. Auch Beziehungen dazwischen existieren schon früh. Sie spiegeln *ist-ein-Beziehungen* („is a“) in der realen Welt wider. Zum Beispiel haben wir die Beziehung „Student is a Person“, wobei „Student“ und „Person“ abstrakte Einheiten sind, die später voraussichtlich zu Klassen weiterentwickelt werden. Durch die Simulation der realen Welt sind solche Beziehungen intuitiv klar, obwohl Details noch gar nicht feststehen. Normalerweise entwickeln sich diese Beziehungen während des Entwurfs zu (vor allem) Untertyp- und (gelegentlich) Vererbungsbeziehungen zwischen Klassen weiter. Es kann sich aber auch herausstellen, dass Details dem Ersetzbarkeitsprinzip widersprechen und Vererbung nicht sinnvoll einsetzbar ist. In solchen Fällen wird es zu Refaktorisierungen kommen, die in frühen Phasen noch einfach durchführbar sind.

Beziehungen in der realen Welt verlieren stark an Bedeutung, sobald genug Details bekannt sind, um sie zu Untertyp- und Vererbungsbeziehungen weiterzuentwickeln. Deshalb konzentrieren wir uns hier nur auf die Unterscheidung zwischen Untertypen und Vererbung.

In Java und ähnlichen objektorientierten Sprachen setzen Untertypen Vererbung voraus und sind derart eingeschränkt, dass die vom Compiler überprüfbaren Bedingungen für Untertypen erfüllt sind. Das heißt, als wesentliches Unterscheidungskriterium verbleibt nur die Frage, ob Zusicherungen zwischen Unter- und Oberklasse kompatibel sind. Diese Unterscheidung können nur Personen treffen, die Bedeutungen von Namen und Kommentaren verstehen. In allen anderen, von Compilern prüfbaren Kriterien sind in Java reine Vererbungs- von Untertypbeziehungen nicht unterscheidbar.

In der Regel ist leicht erkennbar, ob reine Vererbungs- oder Untertypbeziehungen angestrebt werden. Betrachten wir dazu ein Beispiel:



Das Ziel der reinen Vererbung ist es, so viele Teile der Oberklasse wie möglich direkt in der Unterklasse wiederzuverwenden. Angenommen, die Implementierungen von `LargeSet` und `Bag` zeigen so starke Ähnlichkeiten, dass sich die Wiederverwendung von Programmteilen lohnt. In diesem Fall erbt `Bag` große Teile der Implementierung von `LargeSet`. Für diese Entscheidung ist nur der pragmatische Gesichtspunkt, dass sich `Bag` einfacher aus `LargeSet` ableiten lässt als umgekehrt, ausschlaggebend; es würde sich auch `LargeSet` aus `Bag` ableiten lassen. Für `SmallSet` wurde eine von `LargeSet` unabhängige Implementierung gewählt, die bei kleinen Mengen effizienter ist als `LargeSet`.

Wenn wir uns von Konzepten und Typen leiten lassen, schaut die Hierarchie anders aus. Wir führen eine zusätzliche (abstrakte) Klasse `Set` ein, da die Typen von `LargeSet` und `SmallSet` die gleiche Bedeutung haben sollen. Wir wollen im Programmcode nur selten zwischen `LargeSet` und `SmallSet` unterscheiden. `Bag` und `Set` (bzw. `LargeSet`) stehen in keinem Verhältnis zueinander, da die Methoden für das Hinzufügen von Elementen einander ausschließende Bedeutungen haben, obwohl `Set` und `Bag` dieselbe Signatur haben können. Einander ausschließende Bedeutungen kommen daher, dass ein Objekt von `Set` höchstens ein Vorkommen eines Objekts enthalten kann, während in einem Objekt von `Bag` mehrere Vorkommen erlaubt sind. Entsprechend darf eine Methode nur dann ein Element zu einem Objekt von `Set` hinzufügen, wenn das Element noch nicht vorkommt, während die Methode zum Hinzufügen in ein Objekt von `Bag` jedes gewünschte Element akzeptieren muss.

Obiges Beispiel demonstriert unterschiedliche Argumentationen für die reine Vererbung im Vergleich zu Untertypbeziehungen. Die Unterschiede zwischen den Argumentationen sind wichtiger als jene zwischen den Hierarchien, da die Hierarchien selbst von Details und beabsichtigten Verwendungen abhängen. Bei reiner Vererbung ist es häufig so, dass sowohl A von B als auch B von A ableitbar zu sein scheint (konkret dann, wenn der Compiler die Signaturen von A und B nicht voneinander unterscheiden kann) und pragmatische Gesichtspunkte entscheiden, welche Variante bevorzugt wird. Bei Untertypbeziehungen ist die Richtung einer Ableitung immer eindeutig: Ist B Untertyp von A, wobei A und B ungleich sind, liegen immer klar benennbare und aus Zusicherungen ableitbare Gründe vor, weswegen A unmöglich Untertyp von B sein kann.

3.3.2 Vererbung und Codewiederverwendung

Manchmal ist durch reine Vererbungsbeziehungen, die Untertypbeziehungen unberücksichtigt lassen, ein höherer Grad an direkter Codewiederverwendung erreichbar, als wenn Untertypbeziehungen angestrebt werden. Natürlich möchten wir einen möglichst hohen Grad an Codewiederverwendung erzielen. Trotzdem ist es nicht günstig, Untertypbeziehungen unberücksichtigt zu lassen. Durch die Nichtbeachtung des Ersetzbarkeitsprinzips – das heißt, Untertypbeziehungen sind nicht gegeben – ist es nicht mehr möglich, ein Objekt eines Untertyps zu verwenden, wo ein Objekt eines Obertyps erwartet wird. Wenn wir trotzdem ein Objekt einer Unterklasse statt dem einer Oberklasse verwenden, wird sich ein unerwünschtes Programmverhalten zeigen (Fehler im Programm). Verzichten wir auf Ersetzbarkeit, wird die Wartung erschwert, da sich fast jede noch so kleine Programmänderung auf das ganze Programm auswirken kann. Viele Vorteile der objektorientierten Programmierung gehen damit verloren. Unter Umständen gewinnen wir zwar durch die reine Vererbung bessere direkte Codewiederverwendung in kleinem Umfang, tauschen diese aber gegen viele Möglichkeiten für die indirekte Codewiederverwendung in großem Umfang, die nur durch die Ersetzbarkeit gegeben sind.

> *Faustregel:* Wiederverwendung durch Ersetzbarkeit ist wesentlich wichtiger als direkte Wiederverwendung durch Vererbung.

Der allgemeine Ratschlag ist daher ganz klar: Ein wichtiges Ziel ist die Entwicklung geeigneter Untertypbeziehungen. Vererbung ist ein Mittel zum Zweck. Sie soll sich Untertypbeziehungen unterordnen. Im Allgemeinen soll es keine Vererbungsbeziehung geben, die nicht auch eine Untertypbeziehung ist, bei der also alle Zusicherungen kompatibel sind.

Nicht nur Personen mit wenig Programmiererfahrung, auch in der Programmierung sehr erfahrene Personen (allerdings häufig mit wenig Erfahrung in der Entwicklung von Softwarearchitekturen) vergessen allzu leicht das Ersetzbarkeitsprinzip und konzentrieren sich ganz und gar auf direkte Codewiederverwendung durch Vererbung. Daher soll noch einmal klar gesagt werden, dass die Menge des aus einer Oberklasse ererbten Codes für die Codewiederverwendung

nur sehr geringe Bedeutung hat. Viel wichtiger für die Wiederverwendung ist das Bestehen von Untertypbeziehungen.

Wir müssen aber nicht ganz auf direkte Codewiederverwendung durch Vererbung verzichten. In vielen Fällen lässt sich auch dann ein hoher Grad an direkter Codewiederverwendung erzielen, wenn das Hauptaugenmerk auf Untertypbeziehungen liegt. In obigem Beispiel gibt es vielleicht Programmcode, der sowohl in der Klasse `SmallSet` als auch in `LargeSet` vorkommt. Entsprechende Methoden können in der abstrakten Klasse `Set` implementiert sein, von der `SmallSet` und `LargeSet` erbt. Vielleicht gibt es sogar Methoden, die in `Set` und `Bag` gleich sind und in `Collection` implementiert werden können.

Direkte Codewiederverwendung durch Vererbung erspart uns nicht nur das wiederholte Schreiben gleicher Programmtexte, sondern hat auch Auswirkungen auf die Wartbarkeit. Wenn ein Programmteil nur einmal statt mehrmals implementiert ist, brauchen Änderungen nur an einer einzigen Stelle vorgenommen werden, wirken sich aber auf alle Programmteile aus, in denen der veränderte Code verwendet wird. Nicht selten müssen alle gleichen oder ähnlichen Programmteile gleichzeitig geändert werden, wenn sich die Anforderungen ändern. Gerade dabei kann Vererbung hilfreich sein.

Faustregel: Auch Vererbung kann die Wartbarkeit verbessern.

Es kommt vor, dass nach einer Änderung der Anforderungen nicht alle gleichen Programmteile geändert werden sollen, sondern nur einer oder einige wenige. Dann ist es nicht möglich, eine Methode unverändert zu erben. Glücklicherweise ist es in diesem Fall sehr einfach, eine geerbte Methode durch eine neue Methode zu überschreiben. Das Verlagern von Programmtexten von Unterklassen in Oberklassen ist also leicht revidierbar, ohne andere Klassen unnötig zu beeinflussen. In Sprachen wie Java ist es sogar möglich, die Methode zu überschreiben und trotzdem noch auf die überschriebene Methode in der Oberklasse zuzugreifen. Ein Beispiel soll das demonstrieren:

```
public class A {
    public void foo() { ... }
}
public class B extends A {
    private boolean b;
    public void foo() {
        if (b) { ... }
        else { super.foo(); }
    }
}
```

Der Programmcode in `A` ist trotz Überschreibens auch in `B` über `super` verwendbar. Diese Art des Zugriffs auf Oberklassen funktioniert allerdings nicht über mehrere Vererbungsebenen hinweg.

In komplizierten Situationen ist geschickte Faktorisierung notwendig, um direkte Codewiederverwendung zu erreichen:

```
public class A {
    public void foo() {
        if (...) { ... }
        else { ...; x = 1; ... }
    }
}
public class B extends A {
    public void foo() {
        if (...) { ... }
        else { ...; x = 2; ... }
    }
}
```

`foo` muss gänzlich neu geschrieben werden, obwohl der Unterschied minimal ist. Eine Aufspaltung von `foo` in mehrere Methoden kann helfen:

```
public class A {
    public void foo() {
        if (...) { ... }
        else { fooX(); }
    }
    protected void fooX() { ...; x = 1; ... }
}

public class B extends A {
    protected void fooX() { ...; x = 2; ... }
}
```

Das ist eine Anwendung der *Template-Method* (siehe Abschnitt 6.1.4). Wir müssen nur mehr einen Teil der Methode überschreiben. Solche Techniken setzen aber voraus, dass wir bereits beim Schreiben der Klasse `A` sehr klare Vorstellungen davon haben, welche Teile später überschrieben werden müssen. Direkte Codewiederverwendung ergibt sich nicht automatisch oder zufällig, sondern nur dort, wo dies gezielt eingeplant wurde.

Unterschiede zwischen Unter- und Oberklassen lassen sich auch durch Parameter beschreiben. Nach außen sichtbare Methoden setzen Parameterwerte:

```
public class A {
    public void foo() { fooY(1); }
    protected void fooY (int y) {
        if (...) { ... }
        else { ...; x = y; ... }
    }
}
public class B extends A {
    public void foo() { fooY(2); }
}
```

Der Code von `fooY` wird von `B` zur Gänze geerbt. Die überschriebene Methode `foo` muss nur ein Argument an `fooY` übergeben.

Die Vererbungskonzepte in objektorientierten Sprachen sind heute bereits auf viele mögliche Änderungswünsche vorbereitet. Alle Änderungswünsche können damit aber nicht erfüllt werden. Einige Programmiersprachen bieten mehr Flexibilität bei der Vererbung als Java, aber diese zusätzlichen Möglichkeiten stehen oft in Widerspruch zum Ersetzbarkeitsprinzip. Ein bekanntes Beispiel dafür ist private Vererbung in C++, bei der ererbte Methoden außerhalb der abgeleiteten Klasse nicht verwendbar sind. Wenn aus der Verwendung dieser Möglichkeiten klar wird, dass keine Ersetzbarkeit gegeben ist und der Compiler in solchen Fällen verbietet, dass ein Objekt einer Unterklasse verwendet wird, wo ein Objekt einer Oberklasse erwartet wird, ist dagegen auch nichts einzuwenden. Ganz im Gegenteil: Solche Möglichkeiten können die direkte Wiederverwendung von Code genauso verbessern wie die indirekte Wiederverwendbarkeit.

Für Interessierte, nicht Prüfungsstoff. In der Sprache *Sather* gibt es zwei komplett voneinander getrennte Hierarchien auf Klassen: die Vererbungshierarchie für direkte Codewiederverwendung und die Typhierarchie für indirekte Codewiederverwendung. Da die Vererbungshierarchie nicht den Einschränkungen des Ersetzbarkeitsprinzips unterliegt, gibt es zahlreiche Möglichkeiten der Codeveränderung bei der Vererbung, z. B. Einschränkungen der Sichtbarkeit von Methoden und Variablen (Kommentare beginnen mit `--`) :

```
class A is          -- Definition einer Klasse A
    ...;            -- Routinen und Variablen von A
end;
class B is          -- Definition einer Klasse B
    include A       -- B erbt von A
      a->b,         -- wobei a aus A in B b heisst,
      c->,          -- c aus A in B nicht sichtbar
      d->private d; -- und d aus A in B private ist
    ...;            -- Routinen und Variablen von B
end;
```

Neben den konkreten Klassen gibt es in Sather (wie in Java) auch abstrakte Klassen. Deren Namen müssen mit `$` beginnen:

```
abstract class $X is ...; end;
```

Abstrakte Klassen spielen in Sather eine ganz besondere Rolle, da nur sie als Obertypen in Untertypdeklarationen verwendbar sind:

```
abstract class $Y < $X is ...; end;
  -- $Y ist Untertyp von $X
class C < $Y, $Z is ...; end;
  -- C ist Untertyp von $Y und $Z
```

Damit sind Objekte von `C` überall verwendbar, wo Objekte von `$X`, `$Y` oder `$Z` erwartet werden. Anders als `extends` in Java bedeutet `<` in Sather jedoch nicht, dass

die Unterklasse von der Oberklasse erbt, sondern nur, dass der Compiler die statisch überprüfbaren Bedingungen für eine Untertypbeziehung prüft und dynamisches Binden ermöglicht. Für Vererbung ist eine separate `include`-Klausel notwendig.

Ende des Einschubs für Interessierte

Oft benötigen wir gar keine Vererbung für direkte Codewiederverwendung:

```
public class Impl {
    ...
    public void fooY (int y)  {
        if (...) { ... } else { ...; x = y; ... }
    }
}
public class A {
    private Impl delegate = ...
    public void foo() { delegate.fooY(1); }
}
public class B {
    private Impl delegate = ...
    public void foo() { delegate.fooY(2); }
}
```

Wir sagen, Aufrufe von `foo` in `A` und `B` werden an `Impl` bzw. ein Objekt von `Impl` *delegiert*. Delegation hat große Ähnlichkeiten mit Vererbung, ist aber im Detail ein anderer Mechanismus. Wir sehen die Unterschiede am ehesten bei Delegation an einen Obertyp:

```
public class A {
    public String foo1() { return fooX(); }
    public String foo2() { return fooY(); }
    public String fooX() { return "foo1A"; }
    public String fooY() { return "foo2A"; }
}
public class B extends A {
    A delegate = new A();
    public String foo1() { return delegate.foo1(); }
    public String fooX() { return "foo1B"; }
    public String fooY() { return "foo2B"; }
}
```

In `A` sind `foo1` und `foo2` gleich strukturiert, aber `B` delegiert `foo1` an `A` und erbt `foo2` von `A`. Die Ausführung von `foo1` in einem Objekt von `B` gibt als Ergebnis `"foo1A"` zurück, während die Ausführung von `foo2` in einem Objekt von `B` als Ergebnis `"foo2B"` zurückgibt. Das ist keine Überraschung. Durch Delegation wird `fooX` in einem Objekt vom Typ `A` ausgeführt, während `fooY` in einem Objekt vom Typ `B` ausgeführt wird. Würden `foo1` und `foo2` keine weiteren Methoden in `this` ausführen, könnten wir keinen Unterschied zwischen

Delegation und Vererbung erkennen. Überraschend ist eher, dass es in sehr vielen Fällen egal ist, ob Vererbung oder Delegation zum Einsatz kommt. Dort, wo die Unterschiede wichtig sind, ist nicht immer Vererbung die gewünschte Programmsemantik, oft ist Delegation vorteilhaft. Vererbung steht in direkter Konkurrenz zu Delegation, nicht zu Subtyping. Direkte Codewiederverwendung lässt sich ohne Vererbung erreichen, wodurch es keinerlei Grund gibt, Vererbung so einzusetzen, dass dadurch Untertypbeziehungen verletzt werden könnten.

3.3.3 Fehlervermeidung

Im Zusammenhang mit Untertypen und Vererbung passieren immer wieder schwere Fehler, die sich erst nach Programmänderungen äußern. Häufig besteht kein Bewusstsein dafür, etwas falsch zu machen. Die Fehler fallen bei einfachem Testen nicht auf und werden, wenn überhaupt, erst viel später entdeckt.

Der Kardinalfehler besteht darin, eine Untertypbeziehung anzunehmen, wo keine besteht. Nachdem der Compiler (oder das Laufzeitsystem) garantiert, dass die Bedingungen für strukturelle Untertypbeziehungen eingehalten werden, fallen Fehler in der Struktur fast immer rasch auf. Falsche Annahmen bezüglich des Objektverhaltens fallen aber nicht gleich auf. Insbesondere wird leicht übersehen, wenn von einem Objekt eines Untertyps anderes Verhalten erwartet wird als von einem Objekt eines Obertyps; das ist immer ein schwerwiegender Fehler. Dazu kommt noch, dass Anzahl und Komplexität der Typen, die bei der Überprüfung der Untertypbeziehungen zu berücksichtigen sind, oft recht groß werden. Ohne systematische Vorgehensweise fehlt der Überblick.

Die Einhaltung folgender Regel sollte viele Probleme vermeiden:

> *Faustregel:* Beim Programmieren ist *stets* zu prüfen, ob sich ein Objekt *immer* so verhält, wie in *jedem* Typ des Objekts beschrieben.

Leider enthält diese Regel einige Schwierigkeiten, nämlich die Wörter „stets“, „immer“ und „jedem“. Es sind sehr häufig recht komplexe und umfangreiche Prüfungen vorzunehmen. Das geht nur, wenn wir während des Programmierens ein recht genaues Modell des Objektverhaltens und der Verhaltensbeschreibungen (also der Typen) im Kopf haben – für alle Objekte und Typen, mit denen wir uns gerade beschäftigen. Dazu ist höchste Konzentration erforderlich. Es reicht ein Moment der Unachtsamkeit oder Ermüdung, schon ist ein schwerwiegender, kaum zu entdeckender Fehler eingebaut. Wir müssen uns sehr tief in die Verhaltensbeschreibungen einarbeiten, um keine Details zu übersehen. Es ist viel Übung erforderlich, bis wir uns die notwendigen Überprüfungen so gut eingeprägt haben, dass sie im Unterbewusstsein während des Programmierens automatisch ablaufen. Folgende Ratschläge fassen das zusammen:

- Gründlich in vorgegebene Verhaltensbeschreibungen einarbeiten.
- Auf eine Umgebung achten, in der wir uns gut konzentrieren können.
- Bei Ermüdung unverzüglich eine Pause einlegen.

- Viel selbst programmieren, auch komplizierte Aufgaben selbst lösen.

Diese Ratschläge werden dadurch relativiert, dass wir fast immer unter Zeitdruck arbeiten müssen. Es bleibt scheinbar nie genug Zeit für intensive Vorbereitungen, das Einrichten einer „Wohlfühlumgebung“, ausreichend häufige und umfangreiche Pausen und schon gar nicht für Programmierübungen. Trotzdem sollten wir uns, so gut es geht, an diese Ratschläge halten. Dann lernen wir nicht nur inhaltliche Fehler auszumerzen, sondern auch mit Zeitdruck umzugehen.

Ein bewusster Einsatz von Zusicherungen hilft dabei, die komplexen Beziehungen zwischen verschiedenen Beschreibungen des Objektverhaltens zu meistern. Es zahlt sich aus, Zusicherungen auf dem Obertyp in den Untertypen zu wiederholen. Das reduziert die Anzahl der Typen und Konzepte, die ständig im Kopf bleiben müssen. Außerdem werden wir eher auf Widersprüche aufmerksam, wenn widersprüchlichen Kommentare in derselben Klasse stehen.

> *Faustregel:* Durchdachte Zusicherungen sind bei der Überprüfung des Objektverhaltens sehr hilfreich.

Eine Ursache für Fehler in Untertypbeziehungen kann ein falsches Verständnis der Untertypbeziehungen sein, häufig die Verwechslung mit „Is-a-Beziehungen“ aus der objektorientierten Modellierung, gelegentlich auch die Verwechslung mit Vererbungsbeziehungen – siehe Abschnitt 3.3.1. Ein falsches oder noch nicht ganz verinnerlichtes Verständnis liegt mit hoher Wahrscheinlichkeit vor, wenn es als schwierig empfunden wird, die Richtung einer Untertypbeziehung sicher zu bestimmen, wenn also A gefühlsmäßig genausogut ein Untertyp von B sein könnte wie B von A. Solche Probleme lassen sich meist durch intensivere Beschäftigung mit dem Thema lösen. Manchmal ist das aber schwierig, wenn sich das falsche Verständnis über einen langen Zeitraum und viele Projekte zieht und wir uns so daran gewöhnt haben, dass wir die Problematik nicht mehr sehen. Das Umlernen ist alles andere als einfach, aber notwendig.

Eine andere häufige Fehlerursache ist das Übersehen scheinbarer Nebensächlichkeiten. Aufgrund der wichtigsten Bedingungen hätten wir zwar eine Untertypbeziehung, aber eine auf den ersten Blick unwichtige Bedingung zerstört diese Beziehung, ohne dass es uns auffällt. Diesem Problem können wir nur mit Aufmerksamkeit begegnen. Beispielsweise kann die störende Bedingung von Programmteilen kommen, die wir nur für das Testen brauchen und die wir daher nicht näher betrachtet haben. Aber diese Bedingung kann langfristig trotzdem zu schweren Fehlern führen. Ein weiterer, häufig übersehener Aspekt ist die Sichtbarkeit. Wenn wir auf eine ursprünglich private Methode doch von außen zugreifen wollen und sie daher sichtbar machen, müssen wir auch Untertypbeziehungen neuerlich überprüfen. Es könnte sein, dass diese Methode Untertypbeziehungen zerstört. Diese Aufzählung lässt sich endlos weiterführen.

Manchmal ist aber gerade die Konzentration auf Nebensächlichkeiten schuld an falschen Untertypbeziehungen. Wenn wir etwa zu sehr an einfache Änderbarkeit durch Vererbung denken, übersehen wir vielleicht wesentliche Kriterien für Untertypbeziehungen. Wir brauchen schon etwas Erfahrung, um uns stets auf das Wichtigste zu konzentrieren.

3.4 Klassen und Vererbung in Java

In den vorhergehenden Abschnitten haben wir einige wichtige Konzepte objektorientierter Sprachen im Allgemeinen betrachtet. In diesem Abschnitt konzentrieren wir uns auf einige Aspekte der konkreten Umsetzung in Java. Wir weisen auf empfohlene Verwendungen einiger Java-spezifischer Sprachkonstrukte hin und beseitigen häufige Unklarheiten und Missverständnisse. Personen mit viel Java-Programmiererfahrung mögen verzeihen, dass es zur Erreichung dieses Ziels notwendig ist, scheinbar ganz triviale Sprachkonstrukte zu wiederholen.

3.4.1 Klassen in Java

In Java wird streng zwischen Groß- und Kleinschreibung unterschieden, `A` und `a` sind daher verschieden. Namen von Klassen werden per Konvention mit großen Anfangsbuchstaben geschrieben, Namen von Konstanten dagegen oft nur mit Großbuchstaben und alle anderen Namen mit kleinen Anfangsbuchstaben. Zur besseren Lesbarkeit von Programmen sollten wir uns an diese Konventionen halten, auch wenn sie der Compiler nicht erzwingt. In manchen Programmierstilen beginnen Parameternamen mit „`_`“ (Underline), alle anderen Variablennamen mit Kleinbuchstaben. Andere Programmierstile verwenden dagegen mit Underline beginnende Namen ausschließlich für Objektvariablen. Meist werden mit Underline beginnenden Namen jedoch gänzlich vermieden.

Klassen können mehrere explizit definierte Konstruktoren enthalten:

```
public class Circle {
    private int r;
    public Circle(int r) { this.r = r; }       // 1
    public Circle(Circle c) { this.r = c.r; }  // 2
    public Circle() { r = 1; }                 // 3
    ...
}
```

Die Klasse `Circle` hat drei verschiedene Konstruktoren, die sich in der Anzahl oder in den Typen der Parameter unterscheiden. Das ist ein typischer Fall von Überladen. Beim Erzeugen eines neuen Objekts werden dem Konstruktor Argumente übergeben. Anhand der Anzahl und den deklarierten Typen der Argumente wird der geeignete Konstruktor gewählt:

```
    Circle a = new Circle(2);  // Konstruktor 1
    Circle b = new Circle(a);  // Konstruktor 2
    Circle c = new Circle();   // Konstruktor 3
```

In zwei Konstruktoren haben wir die *Pseudovariable* `this` wie den Namen einer Variable verwendet. Tatsächlich bezeichnet `this` immer das aktuelle Objekt der Klasse, das niemals `null` sein kann. In Konstruktoren ist dies das Objekt, das gerade erzeugt wurde. Im ersten Konstruktor benötigen wir `this`, um die Variable `r` im neuen Objekt, das ist `this.r`, vom Parameter `r` des Konstruktors

zu unterscheiden. Wie in diesem Beispiel können Parameter (oder lokale Variablen) Variablen im aktuellen Objekt der Klasse verdecken, die denselben Namen haben. Über `this` können wir dennoch auf die Objektvariablen zugreifen. Wie im zweiten Konstruktor gezeigt, ist `this` immer verwendbar, auch wenn es gar nicht nötig ist. Außerdem benötigen wir `this` bei der Verwendung des aktuellen Objekts der Klasse als Argument. Zum Beispiel liefert `new Circle(this)` innerhalb der Klasse `Circle` eine Kopie des aktuellen Objekts.

Falls in einer Klasse kein Konstruktor explizit definiert ist, enthält die Klasse automatisch einen Defaultkonstruktor:

```
public Klassenname() { super(); }
```

Dabei ruft `super()` den Konstruktor der Oberklasse auf. Ist keine Oberklasse angegeben, wird `Object` als Oberklasse verwendet. Es existiert kein Defaultkonstruktor wenn in der Klasse ein Konstruktor definiert ist.

Objektvariablen, auch *Instanzvariablen* genannt, sind Variablen, die zu Objekten (= Instanzen einer Klasse) gehören. Wenn in der Deklaration einer Objektvariablen keine Initialisierung angegeben ist, wird je nach Typ eine Defaultinitialisierung mit 0 bzw. 0.0 oder `null` vorgenommen; für lokale Variablen erfolgt dagegen keine Defaultinitialisierung. Jedes Objekt der Klasse enthält eigene Objektvariablen.

Manchmal benötigen wir Variablen, die nicht zu einem bestimmten Objekt einer Klasse gehören, sondern zur ganzen Klasse. Solche *Klassenvariablen* sind in Java einfach durch Voranstellen des Schlüsselwortes `static` deklarierbar. Klassenvariablen stehen nicht in Objekten der Klasse, sondern in der Klasse selbst. Auf eine in einer Klasse `A` deklarierte Klassenvariable `x` ist durch `A.x` zugreifbar. Auf Objektvariablen kann hingegen nur über ein Objekt der Klasse zugegriffen werden, wie z. B. in `c.r`, wobei `c` eine Variable vom Typ `Circle` ist. Ein Zugriff auf `Circle.r` ist nicht erlaubt.

Statische *Konstanten* stellen einen häufig verwendeten Spezialfall von Klassenvariablen dar. Sie werden durch `static final` gekennzeichnet. Auch Objektvariablen, Parameter und lokale Variablen können `final` sein. Werte solcher Variablen sind nach der Initialisierung nicht änderbar. Der Compiler kennt nur statische Konstanten (für Optimierungen).

Auch wenn es verlockend ist, sollten wir Klassenvariablen nicht als Variablen sehen, die allen Objekten einer Klasse gemeinsam gehören, da diese Sichtweise längerfristig zu unklaren Verantwortlichkeiten und damit zu Konflikten führen würde. Von einer nichtstatischen Methode aus sollten wir auf eine Klassenvariable nur mit derselben Vorsicht zugreifen, mit der wir auf Variablen eines anderen Objekts zugreifen – am besten nicht direkt, sondern nur über statische Zugriffsmethoden. Statische Konstanten haben dieses Problem nicht. Daher sind `static-final`-Variablen oft auch `public` und uneingeschränkt lesbar.

Eine Methode, die durch `static` gekennzeichnet wird, gehört ebenfalls zur Klasse und nicht zu einem Objekt. Ein Beispiel ist die Methode `main`:

```
static void main (String[] args) { ... }
```

Solche *statischen Methoden* werden über den Namen einer Klasse aufgerufen, nicht über ein Objekt – z. B. `A.x()` wenn `x` eine statische Methode der Klasse `A` ist. Daher ist während der Ausführung der Methode kein aktuelles Objekt der Klasse bekannt und es darf nicht auf Objektvariablen zugegriffen werden. Auch `this` ist in statischen Methoden nicht verwendbar.

Konstruktoren machen es uns leicht, komplexe Initialisierungen von Objektvariablen vorzunehmen. *Static-Initializers* bieten eine derartige Möglichkeit auch für Klassenvariablen:

```
static { ... }
```

Ein Static-Initializer besteht nur aus dem Schlüsselwort `static` und einer beliebigen Sequenz von Anweisungen in geschwungenen Klammern. Diese Codesequenz wird irgendwann vor der ersten Verwendung der Klasse ausgeführt; genaue Kontrolle über den Zeitpunkt haben wir nicht. Jede Klasse kann beliebig viele Static-Initializer enthalten. Obwohl die Ausführungsreihenfolge von oben nach unten klar geregelt ist, sollten wir Abhängigkeiten mehrerer Static-Initializer voneinander dennoch vermeiden.

Das Gegenteil von Konstruktoren sind *Destruktoren*, die festlegen, was unmittelbar vor der endgültigen Zerstörung eines Objekts gemacht werden soll. In Java sind Destruktoren Methoden mit Namen `finalize`, die keine Parameter haben und kein Ergebnis zurückgeben. Wir werden nicht näher auf Destruktoren eingehen, da sie auf Grund einiger Eigenschaften von Java kaum sinnvoll einsetzbar sind – verzögern die Freigabe von Speicher erheblich.

Geschachtelte Klassen sind innerhalb anderer Klassen definiert. Sie können überall definiert sein, wo Variablen deklariert werden dürfen und kommen vorwiegend wegen folgender Eigenschaft zum Einsatz: Innerhalb geschachtelter Klassen sind private Variablen und Methoden aus der Umgebung zugreifbar. Es gibt zwei Arten geschachtelter Klassen:

Statische geschachtelte Klassen: Sie werden mit dem Schlüsselwort `static` versehen und gehören zur umschließenden *Klasse* selbst:

```
class EnclosingClass {
    ...
    static class StaticNestedClass { ... }
    ...
}
```

Wie statische Methoden dürfen statische geschachtelte Klassen nur auf Klassenvariablen der umschließenden Klasse zugreifen und statische Methoden der umschließenden Klasse aufrufen. Dabei kann auch auf private statische Methoden und Variablen zugegriffen werden. Objektvariablen und nichtstatische Methoden der umschließenden Klasse sind nicht zugreifbar; es gibt ja kein entsprechendes Objekt. In Objekten statisch geschachtelter Klassen sind Objektvariablen und nichtstatische Methoden normal zugreifbar. Ein Objekt von `EnclosingClass.StaticNestedClass` wird durch `new EnclosingClass.StaticNestedClass()` erzeugt.

Innere Klassen: Jede innere Klasse wird ohne `static`-Modifier deklariert und gehört zu einem *Objekt* der umschließenden Klasse:

```
class EnclosingClass {
    ...
    class InnerClass { ... }
    ...
}
```

Objektvariablen und nichtstatische Methoden aus der umschließenden Klasse (`EnclosingClass`) können in `InnerClass` uneingeschränkt verwendet werden, auch private. Innere Klassen dürfen jedoch weder statische Methoden noch statische geschachtelte Klassen enthalten, da diese von einem Objekt der äußeren Klasse abhängen würden und dadurch nicht mehr statisch wären. Ein Objekt der inneren Klasse wird z. B. durch `a.new InnerClass()` erzeugt, wobei `a` vom Typ `EnclosingClass` ist.

Abgesehen von den oben beschriebenen Unterschieden entsprechen geschachtelte Klassen den nicht geschachtelten Klassen. Sie können abstrakt sein und von anderen Klassen erben. Es besteht immer eine sehr starke Kopplung zwischen geschachtelten und umgebenden Klassen bzw. ihren Objekten. Geschachtelte Klassen sollen nur verwendet werden, wenn auch alternative Implementierungsmöglichkeiten ähnlich starke Objekt-Kopplungen ergeben würden.

Java-Zwischencode kennt keine geschachtelten Klassen. Daher übersetzt der Compiler diese in normale nicht-public Klassen. Das führt zu einem Problem: Objektvariablen in geschachtelten Klassen sind nicht privat, auch wenn sie mit `private` deklariert wurden. Als `private` deklarierte Variablen sind so zugreifbar, als ob kein Modifier für die Sichtbakeit dabei stehen würde. Vor allem wenig erfahrene Personen setzen geschachtelte Klassen gerne auch dort ein, wo sie nicht angebracht sind, nur um eine etwaige formale Anforderung zu umgehen, dass Objektvariablen als `private` deklariert sein müssen. Damit wird nur die formale Anforderung ausgehebelt, ohne einen Vorteil zu bekommen. Tatsächlich öffnet eine derartige Vorgehensweise Tür und Tor für Fehler im Zusammenhang mit Sichtbarkeit und der Überprüfung von Zusicherungen.

3.4.2 Vererbung und Interfaces in Java

Klassen in Java unterstützen nur Einfachvererbung. Jede Klasse außer `Object` hat genau einen direkten Vorgänger in der Vererbungshierarchie.

Beim Erzeugen eines neuen Objekts wird nicht nur ein Konstruktor der entsprechenden Klasse aufgerufen, sondern auch mindestens ein Konstruktor jeder Oberklasse. Wenn die erste Anweisung in einem Konstruktor „`super(a,b,c);`“ lautet, wird in der Oberklasse, von der direkt geerbt wird, ein entsprechender Konstruktor mit den Argumenten `a`, `b` und `c` aufgerufen. Sonst wird automatisch ein Konstruktor der Oberklasse ohne Argumente aufgerufen. Eine Ausnahme stellen Konstruktoren dar, deren erste Zeile beispielsweise „`this(a,b,c)`;“ lautet. Solche Konstruktoren rufen einen Konstruktor der eigenen Klasse mit den

angegebenen Argumenten auf. Im Endeffekt werden auch in diesem Fall Konstruktoren aller Oberklassen aufgerufen, da irgendein Konstruktor nicht mehr mit `this(...)` beginnt; andernfalls hätten wir eine Endlosrekursion.

Eine Variable in der Unterklasse kann den gleichen Namen wie eine Variable der Oberklasse haben. Die Variable der Unterklasse *verdeckt* die Variable der Oberklasse, aber anders als bei überschriebenen Methoden existieren beide Variablen gleichzeitig. Die in der Unterklasse deklarierte Variable können wir in der Unterklasse direkt durch ihren Namen ansprechen. Die Variable in der Oberklasse, von der die Unterklasse direkt abgeleitet ist, kann über `super` angesprochen werden. Lautet der Name der Variablen `v`, dann ist `super.v` die in der Oberklasse deklarierte Variable. Namen, die bereits weiter oben in der Klassenhierarchie verdeckt wurden, sind durch Typumwandlungen ansprechbar. Z. B. ist `((Oberklasse)this).v` die Variable, die in `Oberklasse` den Namen `v` hat. Eine verdeckte Klassenvariable ist leicht über den Klassennamen ansprechbar, beispielsweise durch `Oberklasse.v`.

Überschriebene nichtstatische Methoden aus der Oberklasse, von der direkt abgeleitet wird, sind über `super` ansprechbar, wie wir in Abschnitt 3.3.2 gesehen haben. Mittels Typumwandlung sind überschriebene Methoden aus Oberklassen aber nicht ansprechbar: Eine Typumwandlung ändert nur den deklarierten Typ. Aufgrund von dynamischem Binden hat der deklarierte Typ (abgesehen von der Auswahl überladener Methoden) keinen Einfluss auf die Methode, die ausgeführt wird. Dynamisches Binden macht den Effekt der Typumwandlung wieder rückgängig. Statische Methoden aus Oberklassen sind wie Klassenvariablen durch Voranstellen des Klassennamens ansprechbar.

Eine Methode der Unterklasse überschreibt eine der Oberklasse nur wenn Name, Parameteranzahl und Parametertypen gleich sind; Ergebnistypen können (seit Java 1.5) kovariant sein. Sonst sind die Methoden überladen, das heißt, in der Unterklasse existieren beide Methoden gleichzeitig. Die deklarierten Typen der übergebenen Argumente entscheiden, welche überladene Methode aufgerufen wird.

Der Modifier `final` in einer Methodendefinition verhindert, dass die Methode in einer Unterklasse überschrieben wird. Ohne Überschreiben erfolgt der Aufruf wegen statischem Binden (meist unmerklich) schneller als durch dynamisches Binden. Trotzdem wird `final` eher selten verwendet, da oft gesagt wird, dass dies die Wartbarkeit vermindern kann. Es ist nicht klar, ob das stimmt, weil in der Praxis ohnehin fast ausschließlich nur jene Methoden sinnvoll überschrieben werden, die von Anfang an für das Überschreiben vorgesehen waren.

Auch ganze Klassen können mit dem Modifier `final` versehen sein. Solche Klassen haben keine Unterklassen. Dadurch ist es auch nicht möglich, die Methoden der Klassen zu überschreiben. Manche objektorientierte Programmierstile verwenden solche Klassen um klarzustellen, dass das Verhalten der Objekte durch die Implementierung festgelegt ist. Clients können sich auf alle Implementierungsdetails verlassen, ohne auf mögliche Ersetzungen Rücksicht nehmen zu müssen. Änderungen der Klassen können aber aufwändig sein, da alle Clients zu überprüfen und gegebenenfalls ebenfalls zu ändern sind. Abstrakte Klassen dürfen natürlich nicht `final` sein. Die Nachteile der Verwendung von `final`

Klassen sind durch den konsequenten Einsatz von abstrakten Klassen und Interfaces als Typen von Parametern und Variablen vermeidbar. In diesem Fall kann die konsequente Verwendung von `final` Klassen für alle nicht-abstrakten Klassen vorteilhaft sein. Damit ergibt sich ein Programmierstil ähnlich dem von Sather (siehe Abschnitt 3.3.2).

Interfaces sind in Java abstrakte Klassen, die (abgesehen von `static-final`-Konstanten) keine Variablen unterstützen, dafür aber Mehrfachvererbung erlauben. Interfaces unterscheiden sich von normalen abstrakten Klassen wie folgt:

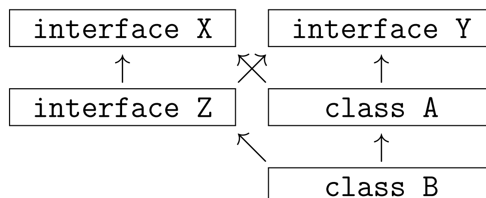
- Beginnen mit `interface` statt `abstract class`.
- Jede enthaltene Variable stellt eine `static-final`-Konstante dar.
- Der Modifier `abstract` ist nicht notwendig. Vor Java 8 waren alle Methoden implizit abstrakt und nicht `static`. Ab Java 8 können statische Methoden und mit dem Modifier `default` auch nicht-statische Methoden implementiert werden.
- Methoden und Konstanten sind `public`, auch ohne Modifier.
- Nach `extends` können mehrere, durch Komma getrennte Namen von Interfaces stehen (Mehrfachvererbung).

Auch Klassen können von mehreren Interfaces erben:

```
public interface X {
    double PI = 3.14159; // implizit public static final
    double fooX();
}
public interface Y {
    double fooY();
}
public interface Z extends X, Y {
    default double fooZ() { return fooX() + fooY(); }
}
public class A implements X, Y {
    private double factor = 2.0; // Objektvariable
    public double foo() { return PI; }
    public double fooX() { return factor * PI; }
    public double fooY() { return factor * fooX(); }
}
public class B extends A implements Z {
    public double fooY() { return 3.3 * foo(); }
}
```

Interface `Z` erbt von `X` und `Y`. Somit enthält `Z` die Konstante `PI` sowie die Methoden `fooX`, `fooY` und `fooZ` (mit Default-Implementierung). Die Klasse `A` erbt ebenfalls von `X` und `Y`. Interfaces, von denen eine Klasse erbt, stehen nach

`implements` um anzudeuten, dass die in den Interfaces deklarierten Methoden in der Klasse zu implementieren sind und die Klasse die durch die Interfaces spezifizierten Schnittstellen implementiert. In einer Klassendefinition kann nach `extends` nur eine Klasse stehen. Methoden mit Default-Implementierungen müssen nicht überschrieben werden. Das Beispiel hat diese Typstruktur:



Interfaces sind als Typen verwendbar. In dieser Hinsicht unterscheiden sie sich nicht von Klassen. Interfaces sind stabiler als Klassen mit Variablen. Durch Mehrfachvererbung sind sie flexibler einsetzbar als abstrakte Klassen. Daher sollten Interfaces immer verwendet werden, wo dies möglich ist, das heißt, wo keine Objektvariablen deklariert werden müssen.

Faustregel: Interfaces sind Klassen vorzuziehen.

Interfaces dienen fast ausschließlich der Festlegung von Untertypbeziehungen, die das Ersetzbarkeitsprinzip erfüllen. Reine Vererbung ist mit Interfaces wenig sinnvoll. Das ist ein weiterer Grund, warum Interfaces Klassen vorzuziehen sind: Wir kommen nicht so leicht in Versuchung, uns auf Vererbung statt Untertypbeziehungen zu konzentrieren. Wie bei Klassen gilt auch bei Interfaces, dass die entsprechenden Typen gemachte Zusicherungen einschließen. Wir sollten also stets Zusicherungen hinschreiben und die Kompatibilität der Zusicherungen händisch überprüfen. In Interfaces sind Zusicherungen wichtiger als in Klassen, da das Verhalten von Methoden ohne Default-Implementierungen höchstens aus gut gewählten Methodennamen, aber nicht aus einer Implementierung abgeleitet werden kann. Außerdem suchen wir mit entsprechender Erfahrung zuerst in Interfaces nach Zusicherungen und greifen nur dann auf weitere Zusicherungen in Klassen zurück, wenn das notwendig ist. Obiges Beispiel ist eigentlich unvollständig, da Zusicherungen fehlen.

3.4.3 Pakete und Zugriffskontrolle in Java

Jede compilierte Java-Klasse wird in einer eigenen Datei gespeichert. Der Dateiname entspricht dem Namen der Klasse mit der Endung `.class`. Das Verzeichnis, das diese Datei enthält, entspricht dem *Paket,* zu dem die Klasse gehört; das ist ein Namensraum (siehe Abschnitt 1.3.1). Der Name des Verzeichnisses ist der Paketname. Auch die Datei, die den Quellcode enthält, hat den gleichen Namen wie die Klasse, aber mit der Endung `.java`. Sie steht in einem Verzeichnis, dessen Name der Paketname ist, außer Klassen im *Default-Paket* an der Basis der Verzeichnisstruktur für Java-Klassen, dem sogenannten *Class-Path*. Es ist möglich, dass eine Quellcodedatei mehrere Klassen enthält. Von den nicht-geschachtelten Klassen in einer Datei darf nur die, deren Name

gleich dem Dateinamen ist, als `public` definiert sein. Bei der Übersetzung wird jedenfalls eine eigene Datei pro Klasse erzeugt.

Namen im Quellcode müssen den Namen des Pakets enthalten, in dem die Namen definiert sind, außer wenn sie im selben Paket definiert sind. Diese Namen sind relativ zum Class-Path, der vom System vorgegeben ist und über Parameter des Compilers bzw. Interpreters geändert werden kann. Nehmen wir an, wir wollen die statische Methode `foo` in einer Klasse `AClass` aufrufen, deren Quellcode in der Datei

```
myclasses/examples/test/AClass.java
```

relativ zum Class-Path steht. Dann lautet der Aufruf folgendermaßen:

```
myclasses.examples.test.AClass.foo();
```

Solche langen Namen bedeuten einen hohen Schreibaufwand und sind auch nur schwer lesbar. Daher bietet Java eine Möglichkeit, Klassen oder ganze Pakete zu importieren. Enthält der Quellcode zum Beispiel die Zeile

```
import myclasses.examples.test;
```

dann kann man `foo` durch „`test.AClass.foo();`“ aufrufen, da der Paketname `test` lokal bekannt ist. Enthält der Quellcode sogar die Zeile

```
import myclasses.examples.test.AClass;
```

können wir `foo` noch einfacher durch „`AClass.foo();`“ aufrufen. Häufig möchten wir alle Klassen in einem Paket auf einmal importieren. Das geht beispielsweise dadurch:

```
import myclasses.examples.test.*;
```

Auch nach dieser Zeile ist „`AClass.foo();`“ direkt aufrufbar.

Beliebig viele solche Zeilen mit dem Schlüsselwort `import` dürfen am Anfang einer Datei mit Quellcode stehen, sonst aber nirgends. Vor diesen Zeilen darf höchstens eine einzelne Zeile

```
package paketName;
```

stehen, wobei `paketName` den Namen und Pfad des Pakets bezeichnet, zu dem die Klasse in der Quelldatei gehört. Ist eine solche Zeile in der Quelldatei vorhanden, muss der Aufruf von `java` zur Ausführung der übersetzten Datei im Dateinamen den Pfad enthalten, der in `paketName` vorgegeben ist (wobei Punkte in `paketName` je nach Betriebssystem durch „`/`“ oder „`\`“ ersetzt sind). Wenn die compilierte Datei in einem anderen Verzeichnis steht, lässt sie sich nicht verwenden. Die Zeile mit dem Schlüsselwort `package` stellt also – zumindest zu einem gewissen Grad – sicher, dass die Datei nicht einfach aus dem Kontext gerissen und in einem anderen Paket eingesetzt wird.

Nun kommen wir zur Sichtbarkeit von Namen. Generell sind alle Einheiten wie Klassen, Variablen, Methoden, etc. in dem Bereich (Scope), in dem sie

definiert wurden, sichtbar und verwendbar, zumindest, wenn sie nicht durch eine andere Einheit mit dem gleichen Namen verdeckt sind. Einheiten, die mit dem Schlüsselwort `private` definiert wurden, sind sonst nirgends sichtbar, auch nicht in Unterklassen.

Es stimmt nicht, dass Einheiten, die mit dem Schlüsselwort `private` definiert sind, nicht vererbt werden. Im vom Compiler erzeugten Code müssen sie vorhanden sein, da aus einer Oberklasse ererbte Methoden darauf möglicherweise zugreifen. Aber `private` Einheiten sind (außer indirekt über ererbte Methoden) in der erbenden Klasse nicht zugreifbar. Deren Namen sind in der erbenden Klasse nicht definiert oder beziehen sich auf ganz andere Einheiten. Der Einfachheit halber sagen wir oft, dass `private` Einheiten nicht vererbt werden, da es beim Programmieren meist diesen Anschein hat, auch wenn es aus Sicht des Compilers nicht stimmt.

Einheiten, die mit dem Schlüsselwort `public` definiert wurden, sind dagegen überall sichtbar und werden vererbt. Wir können

```
myclasses.examples.test.AClass.foo();
```

aufrufen, wenn sowohl die Klasse `AClass` als auch die statische Methode `foo` mit dem vorangestellten Schlüsselwort `public` definiert wurde. In allen anderen Fällen dürfen wir `foo` nicht aufrufen.

Neben diesen beiden Extremfällen gibt es noch zwei weitere Möglichkeiten zur Steuerung der Sichtbarkeit. Dabei sind Einheiten zwar im gleichen Paket sichtbar, aber nicht in anderen Paketen. Einheiten, deren Definitionen mit `protected` beginnen, sind innerhalb des Paketes sichtbar und werden an alle Unterklassen vererbt, auch wenn diese in einem anderen Paket stehen. Einheiten, die weder `public` noch `protected` oder `private` sind, haben Default-Sichtbarkeit, auch „package-private“ genannt. Sie sind im gleichen Paket sichtbar, sonst aber nirgends. Einheiten mit Default-Sichtbarkeit sind in Unterklassen nur dann sichtbar, wenn die Unterklassen im gleichen Paket stehen.

Wir fassen diese Sichtbarkeitsregeln in einer Tabelle zusammen:

	`public`	`protected`	—	`private`
sichtbar im selben Paket	ja	ja	ja	nein
sichtbar in anderem Paket	ja	nein	nein	nein
ererbbar im selben Paket	ja	ja	ja	nein
ererbbar in anderem Paket	ja	ja	nein	nein

Andere Sichtbarkeitseigenschaften als die in der Tabelle angeführten werden von Java derzeit nicht unterstützt.

Es ist nicht leicht, stets die richtigen Sichtbarkeitseigenschaften zu wählen. Hier sind einige Ratschläge, die diese Wahl erleichtern sollen:

- Alle Methoden, Konstruktoren und Konstanten, die wir bei der Verwendung der Klasse oder von Objekten der Klasse benötigen, sollen `public` sein. Ganz selten gilt das auch für Variablen; das ist aber verpönt, weil sich Schreibzugriffe von außen negativ auf die Prüfung von Zusicherungen, vor allem Invarianten und History-Constraints auswirken.

- Wir wählen `private` für alles, was nur innerhalb der Klasse verwendet werden soll und außerhalb der Klasse nicht verständlich zu sein braucht. Die Bedeutung von Variablen ist außerhalb der Klasse in der Regel ohnehin nicht verständlich, `private` daher ideal.

- Wenn Methoden und Konstruktoren (gelegentlich auch Variablen) für die Verwendung einer Klasse und ihrer Objekte nicht nötig sind, aber in Unterklassen darauf zugegriffen werden muss, können wir `protected` verwenden. Aber `protected` Variablen sollten wir weitgehend meiden und stattdessen in Unterklassen nur über `protected` Methoden indirekt darauf zugreifen. Generell ist `protected` nur einzusetzen, wenn es kaum anders geht – schlechter Stil wegen schlechter Wartbarkeit.

- In einem Paket sollen alle Klassen stehen, die eng zusammenarbeiten. Methoden und Konstruktoren (und ganz selten Variablen), auf die von außerhalb der Klasse nur innerhalb eines Paketes zugegriffen wird, sollen außerhalb des Paketes auch nicht sichtbar sein. Wir verwenden dafür am besten Default-Sichtbarkeit. Außer bei einem echten Bedarf an enger Zusammenarbeit zwischen Klassen sollten wir `private` bevorzugen. Insbesondere Variablen mit Default-Sichtbarkeit sind zu vermeiden.

> *Faustregel:* Fast alle Variablen sollten `private` sein.

Es ist schwierig, geeignete Zusicherungen für Zugriffe auf Variablen anzugeben. Das ist ein wichtiger Grund für die Empfehlung, die Sichtbarkeit von Variablen so weit wie möglich einzuschränken. Statt einer Variablen können wir in der nach außen sichtbaren Schnittstelle eines Objekts immer auch eine Methode zum Abfragen des aktuellen Werts („Getter") und eine zum Setzen des Werts („Setter") schreiben. Obwohl solche Methoden weniger problematisch sind als Variablen, ist es noch besser, wenn sie gar nicht benötigt werden. Solche Methoden deuten, wie nach außen sichtbare Variablen, auf starke Objektkopplung und niedrigen Klassenzusammenhalt und damit auf eine schlechte Faktorisierung des Programms hin. Refaktorisierung ist angesagt.

> *Faustregel:* Methoden zum direkten Setzen bzw. Abfragen von Variablenwerten sind möglichst zu vermeiden.

Meist sind Klassen ganz unten in der Typhierarchie bestens für die Deklaration von Objektvariablen geeignet. Im Gegensatz zum Erben gemeinsamer Methoden bringt das Erben von Objektvariablen keine Vorteile. Typischerweise stehen Methoden, die direkt auf Objektvariablen zugreifen, ganz unten in der Typhierarchie und ererbte Methoden greifen nicht direkt auf Objektvariablen zu.

Wenn unklar ist, wo etwas sichtbar sein soll, verwenden wir zu Beginn die am stärksten eingeschränkte Variante. Erst wenn sich herausstellt, dass eine weniger restriktive Variante nötig ist, erlauben wir weitere Zugriffe. Es ist einfacher, Restriktionen der Sichtbarkeit aufzuheben, als neue Einschränkungen einzuführen. Auch beim Ausweiten der Sichtbarkeit können sich Probleme ergeben, etwa eine Untertypbeziehung verletzt werden.

Java bietet eine weitere Möglichkeit, die Sichtbarkeit von Klassen gezielt einzuschränken: Da jede übersetzte Java-Klasse in einer eigenen Datei steht, kann über die Zugriffsrechte des Dateisystems geregelt werden, wer darauf zugreifen darf. Leider sind diese Kontrollmöglichkeiten durch ganz unterschiedliche Dateisysteme nicht portabel und werden, auch wegen der umständlichen Realisierung, praktisch so gut wie nie verwendet.

In Java kann die Sichtbarkeit nur auf Klassen und Pakete eingeschränkt werden. Es gibt keine Möglichkeit der Einschränkung auf einzelne Objekte. Daher sind alle Variablen eines Objekts stets auch außerhalb des Objekts zugreifbar, zumindest von einem anderen Objekt derselben Klasse aus. Das bedeutet jedoch nicht, dass solche Zugriffe wünschenswert sind. Im Gegenteil: Direkte Zugriffe (vor allem Schreibzugriffe) auf Variablen eines anderes Objekts führen leicht zu inkonsistenten Zuständen und Verletzungen von Invarianten. Dieses Problem kann nur durch vorsichtige, disziplinierte Programmierung gelöst werden. Einschränkungen der Sichtbarkeit können aber helfen, den Bereich, in dem es zu direkten Variablenzugriffen von außen kommen kann, klein zu halten. Personen in der Softwareentwicklung sind ja stets für ganze Klassen bzw. Pakete, nicht nur für einzelne Objekte verantwortlich. Insofern sind Klassen und Pakete als Grundeinheiten für die Steuerung der Sichtbarkeit gut gewählt.

3.5 Ausnahmebehandlungen und Ersetzbarkeit

Ausnahmebehandlungen dienen vor allem dem Umgang mit unerwünschten Programmzuständen. Zum Beispiel werden in Java Ausnahmen ausgelöst, wenn das Objekt bei einer Typumwandlung keine Instanz des gegebenen Typs ist, eine Nachricht an `null` gesendet wird, etc. In diesen Fällen kann der Programmablauf nicht normal fortgeführt werden, da grundlegende Annahmen verletzt sind. Ausnahmebehandlungen geben uns die Möglichkeit, das Programm auch in solchen Situationen noch weiter ablaufen zu lassen. Obwohl der Fokus dieses Abschnitts auf Ausnahmen im Zusammenhang mit Ersetzbarkeit liegt, gehen wir doch etwas breiter auf Ausnahmen ein, damit das Thema in der Gesamtheit besser verständlich wird. Wir betrachten in Abschnitt 3.5.1 Ausnahmen in Java, geben in Abschnitt 3.5.2 einige Hinweise auf den sinnvollen Einsatz und gehen in 3.5.3 auf Zusammenhänge mit Zusicherungen ein.

3.5.1 Ausnahmebehandlung in Java

Eigentlich sollten wir mit den Grundlagen von Ausnahmen und deren Behandlung in Java schon vertraut sein. Da es diesbezüglich immer wieder Schwierigkeiten gibt und um Missverständnissen hinsichtlich der Terminologie vorzubeugen sind hier die wichtigsten Fakten zusammengefasst.

Ausnahmen sind in Java gewöhnliche Objekte, die über spezielle Mechanismen als Ausnahmen verwendet werden. Alle Objekte von `Throwable` sind dafür verwendbar. Praktisch verwenden wir nur Objekte der Unterklassen von `Error` und `Exception`, zwei Unterklassen von `Throwable`.

Unterklassen von `Error` werden hauptsächlich für vordefinierte, schwerwiegende Ausnahmen des Java-Laufzeitsystems verwendet und deuten auf echte Fehler hin, die während der Programmausführung entdeckt wurden. Es ist praktisch kaum möglich, solche Ausnahmen sinnvoll abzufangen; ihr Auftreten führt fast immer zur Programmbeendigung. Beispiele für Untertypen von `Error` sind `OutOfMemoryError` und `StackOverflowError`. Bei diesen Ausnahmen ist zu erwarten, dass jeder Versuch, das Programm fortzusetzen, wieder zu solchen Ausnahmen führt.

Unterklassen von `Exception` sind wieder in zwei Bereiche gegliedert – überprüfte Ausnahmen (häufig von uns selbst definiert) und nicht überprüfte, meist vom System vorgegebene. Letztere sind Objekte von `RuntimeException`, einer Unterklasse von `Exception`. Dazu zählen `IndexOutOfBoundsException` (Arrayzugriff außerhalb des Indexbereichs), `NullPointerException` (Senden einer Nachricht an `null`) und `ClassCastException` (Typumwandlung, wobei der dynamische Typ nicht dem gewünschten Typ entspricht). Oft ist es sinnvoll oder sogar notwendig, Ausnahmen, die Objekte von `Exception` sind, abzufangen und den Programmablauf an geeigneter Stelle fortzusetzen.

Vom Java-Laufzeitsystem werden nur Objekte der vordefinierten Unterklassen von `Error` und `RuntimeException` als Ausnahmen ausgelöst. Programme können explizit Ausnahmen auslösen, die Objekte jeder beliebigen Unterklasse von `Throwable` sind. Dies geschieht mit Hilfe der `throw`-Anweisung, wie im folgenden Beispiel:

```
public class Help extends Exception { ... }
    ...
    if (helpNeeded())
        throw new Help();
```

Falls `helpNeeded` als Ergebnis `true` liefert, wird ein neues Objekt von `Help` erzeugt und als Ausnahme verwendet. Bei Ausführung der `throw`-Anweisung (oder wenn das Laufzeitsystem eine Ausnahme auslöst) wird der reguläre Programmfluss abgebrochen. Das Laufzeitsystem sucht die nächste geeignete Stelle, an der die Ausnahme abgefangen und das Programm fortgesetzt werden kann. Wird keine solche Stelle gefunden, kommt es zu einem Programmabbruch.

Zum Abfangen von Ausnahmen gibt es `try-catch`-Blöcke:

```
    try { ... }
    catch(Help e) { ... }
    catch(Exception e) { ... }
```

Im Block nach dem Wort `try` stehen beliebige Anweisungen, die ausgeführt werden, wenn der `try-catch`-Block ausgeführt wird. Falls während der Ausführung dieses `try`-Blocks eine Ausnahme auftritt, wird eine passende `catch`-Klausel nach dem `try`-Block gesucht. Jede `catch`-Klausel enthält nach dem Schlüsselwort `catch` (wie eine Methode mit einem Parameter) genau einen Parameter. Ist die aufgetretene Ausnahme ein Objekt des Parametertyps, dann kann die `catch`-Klausel die Ausnahme abfangen. Das bedeutet, dass die Abarbeitung der

Befehle im `try`-Block nach Auftreten der Ausnahme endet, dafür aber die Befehle im Block der `catch`-Klausel ausgeführt werden. Im Beispiel können beide `catch`-Klauseln eine Ausnahme vom Typ `Help` abfangen, da jedes Objekt von `Help` auch ein Objekt von `Exception` ist. Wenn es mehrere passende `catch`-Klauseln gibt, wird die im Programmtext erste passende gewählt. Nach einer abgefangenen Ausnahme wird das Programm so fortgesetzt, als ob es keine Ausnahmebehandlung gegeben hätte. Das heißt, nach der `catch`-Klausel wird der erste Befehl nach dem `try-catch`-Block ausgeführt.

Normalerweise ist nicht klar, an genau welcher Stelle im `try`-Block die Ausnahme ausgelöst wurde. Wir wissen daher nicht, welche Befehle bereits ausgeführt wurden und ob die Werte in den Variablen konsistent sind. Wir müssen `catch`-Klauseln so schreiben, dass sie mögliche Inkonsistenzen beseitigen. Variablen, die in einem `try`-Block deklariert wurden, sind in entsprechenden `catch`-Blöcken nicht sichtbar.

Falls ein `try-catch`-Block eine Ausnahme nicht abfangen kann oder während der Ausführung einer `catch`-Klausel eine weitere Ausnahme ausgelöst wird, wird nach dem nächsten umschließenden `try`-Block gesucht. Wenn es innerhalb der Methode, in der die Ausnahme ausgelöst wurde, keinen geeigneten `try-catch`-Block gibt, so wird die Ausnahme von der Methode statt einem regulären Ergebnis zurückgegeben und die Suche nach einem passenden `try-catch`-Block im Aufrufer fortgesetzt, solange bis die Ausnahme abgefangen ist oder es (für die statische Methode `main`) keinen Aufrufer mehr gibt, an den die Ausnahme weitergereicht werden könnte. Letzterer Fall führt zum Programmabbruch.

Methoden dürfen nicht Ausnahmen beliebiger Typen zurückgeben, sondern nur Objekte von `Error` und `RuntimeException` sowie Ausnahmen von Typen, die im Kopf der Methode ausdrücklich angegeben sind – daher der Begriff *überprüfte Ausnahmen.* Soll eine Methode `foo` beispielsweise auch Objekte von `Help` als Ausnahmen zurückgeben können, so muss dies durch eine `throws`-Klausel deklariert sein:

```
void foo() throws Help { ... }
```

Alle Ausnahmen, die im Rumpf der Methode ausgelöst, aber mangels Eintrag in der `throws`-Klausel nicht zurückgegeben werden können, müssen im Rumpf der Methode durch einen geeigneten `try-catch`-Block abgefangen werden. Das wird vom Compiler überprüft.

Die im Kopf von Methoden deklarierten Ausnahmetypen sind für das Bestehen von Untertypbeziehungen relevant. Das Ersetzbarkeitsprinzip verlangt, dass die Ausführung einer Methode eines Untertyps nur solche Ausnahmen zurückliefern kann, die bei Ausführung der entsprechenden Methode des Obertyps erwartet werden. Daher dürfen Methoden in einer Unterklasse in der `throws`-Klausel nur Typen anführen, die auch in der entsprechenden `throws`-Klausel in der Oberklasse stehen. Selbiges gilt natürlich auch für Interfaces. Der Java-Compiler überprüft diese Bedingung. In Unterklassen dürfen Typen von Ausnahmen weggelassen werden, das heißt, Untertypen dürfen weniger Ausnahmen werfen als Obertypen, aber keinesfalls mehr. Das folgende Beispiel zeigt dies:

```
class A {
    void foo() throws Help, SyntaxError { ... }
}
class B extends A {
    void foo() throws Help { ... }
}
```

Der Compiler kann natürlich nur das Vorhandensein von Typnamen in `throws`-Klauseln prüfen, nicht das tatsächliche Werfen von Ausnahmen. Ersetzbarkeit verlangt, dass die Ausführung einer Methode im Untertyp niemals eine Ausnahme zurückpropagieren darf, die nicht auch bei Ausführung der entsprechenden Methode im Obertyp in derselben Situation zurückgegeben werden könnte und daher vom Aufrufer erwartet wird. Beschreibt eine Nachbedingung einer Methode im Obertyp beispielsweise, dass in einer konkreten Situation eine Ausnahme ausgelöst und propagiert wird, darf die entsprechende Methode im Untertyp diese Ausnahme nicht auch in gänzlich anderen Situationen auslösen.

Zum Abschluss seien `finally`-Blöcke erwähnt: Nach einem `try`-Block und beliebig vielen `catch`-Klauseln kann ein `finally`-Block stehen:

```
try { ... }
catch(Help e) { ... }
catch(Exception e) { ... }
finally { ... }
```

Wird der `try`-Block ausgeführt, so wird in jedem Fall auch der `finally`-Block ausgeführt, unabhängig davon, ob Ausnahmen aufgetreten sind oder nicht. Tritt keine Ausnahme auf, wird der `finally`-Block unmittelbar nach dem `try`-Block ausgeführt. Tritt eine Ausnahme auf, die abgefangen wird, erfolgt die Ausführung des `finally`-Blocks unmittelbar nach der der `catch`-Klausel. Tritt eine nicht abgefangene Ausnahme im `try`-Block oder in einer `catch`-Klausel auf, wird der `finally`-Block vor Weitergabe der Ausnahme ausgeführt. Tritt während der Ausführung des `finally`-Blocks eine nicht abgefangene Ausnahme auf, wird der `finally`-Block nicht weiter ausgeführt und die Ausnahme weitergegeben. Allenfalls vorher angefallene Ausnahmen werden in diesem Fall vergessen.

Solche `finally`-Blöcke eignen sich dazu, Ressourcen auch beim Auftreten von Ausnahmen freizugeben. Dabei ist aber Vorsicht geboten, da oft nicht klar ist, ob eine bestimmte Ressource bereits vor dem Auftreten einer Ausnahme angefordert war.

3.5.2 Einsatz von Ausnahmebehandlungen

Ausnahmen werden in folgenden Fällen eingesetzt:

Unvorhergesehene Programmabbrüche: Wird eine Ausnahme nicht abgefangen, kommt es zu einem Programmabbruch. Die dabei verursachte Bildschirmausgabe (Stack-Trace) enthält genaue Informationen über Art und Ort des Auftretens der Ausnahme. Damit lassen sich die Ursachen von Programmfehlern leichter finden.

Kontrolliertes Wiederaufsetzen: Nach aufgetretenen Fehlern oder in außergewöhnlichen Situationen wird das Programm an genau definierbaren Punkten weiter ausgeführt. Während der Programmentwicklung ist es vielleicht sinnvoll, einen Programmlauf beim Auftreten eines Fehlers abzubrechen, aber im praktischen Einsatz soll das Programm auch dann noch funktionieren, wenn ein Fehler aufgetreten ist. Ausnahmebehandlungen wurden vor allem zu diesem Zweck eingeführt: Man kann einen Punkt festlegen, an dem es auf alle Fälle weiter geht. Leider können Ausnahmebehandlungen echte Programmfehler nicht beheben, sondern nur den Benutzer darüber informieren und dann das Programm abbrechen, oder weiterhin (eingeschränkte) Dienste anbieten. Ergebnisse bereits erfolgter Berechnungen gehen dabei meist verloren.

Ausstieg aus Sprachkonstrukten: Ausnahmen sind nicht auf den Umgang mit Programmfehlern beschränkt. Sie erlauben ganz allgemein das vorzeitige Abbrechen der Ausführung von Blöcken, Kontrollstrukturen, Methoden, etc. in außergewöhnlichen Situationen. Das Auftreten solcher Ausnahmen wird erwartet (im Gegensatz zum Auftreten unbekannter Fehler). Es ist daher relativ leicht, entsprechende Ausnahmebehandlungen durchzuführen, die eine sinnvolle Weiterführung des Programms ermöglichen.

Rückgabe alternativer Ergebniswerte: In Java und vielen anderen Sprachen kann eine Methode nur Ergebnisse eines bestimmten Typs liefern. Wenn in der Methode eine unbehandelte Ausnahme auftritt, wird an den Aufrufer statt eines Ergebnisses die Ausnahme zurückgegeben, die er abfangen kann. Damit ist es möglich, dass die Methode an den Aufrufer in Ausnahmesituationen Objekte zurückgibt, die nicht den deklarierten Ergebnistyp der Methode haben.

Die ersten zwei Punkte beziehen sich auf fehlerhafte Programmzustände, die durch Ausnahmen eingegrenzt werden. Wir wollen solche Situationen beim Programmieren vermeiden. Es gelingt aber nicht immer. Die letzten beiden Punkte beziehen sich auf Situationen, in denen Ausnahmen und Ausnahmebehandlungen gezielt eingesetzt werden, um den üblichen Programmablauf abzukürzen oder Einschränkungen des Typsystems zu umgehen. Im Folgenden wollen wir uns den bewussten Einsatz von Ausnahmen genauer vor Augen führen.

> *Faustregel:* Aus Gründen der Wartbarkeit sollen Ausnahmen und Ausnahmebehandlungen nur in echten Ausnahmesituationen und sparsam eingesetzt werden.

Bei Auftreten einer Ausnahme wird der normale Programmablauf durch eine Ausnahmebehandlung ersetzt. Während der normale Kontrollfluss lokal sichtbar und durch Verwendung strukturierter Sprachkonzepte wie Schleifen und bedingte Anweisungen relativ einfach nachvollziehbar ist, sind Ausnahmebehandlungen meist nicht lokal und folgen auch nicht den gut verstandenen strukturierten Sprachkonzepten. Ein Programm mit vielen Ausnahmebehandlungen ist

schwer lesbar und Programmänderungen bleiben selten lokal, da immer auch eine nicht direkt sichtbare `catch`-Klausel betroffen sein kann. Das sind gute Gründe, um die Verwendung von Ausnahmen zu vermeiden.

> *Faustregel:* Wir sollen Ausnahmen nur einsetzen, wenn dadurch die Programmlogik vereinfacht wird.

Es gibt aber auch Fälle, in denen der Einsatz von Ausnahmen und deren Behandlungen die Programmlogik wesentlich vereinfachen kann, beispielsweise wenn viele bedingte Anweisungen durch eine einzige `catch`-Klausel ersetzbar sind. Wenn das Programm durch Verwendung von Ausnahmebehandlungen einfacher lesbar und verständlicher wird, ist der Einsatz durchaus sinnvoll. Das gilt vor allem dann, wenn die Ausnahmen lokal abgefangen werden. Oft sind aber gerade die nicht lokal abfangbaren Ausnahmen jene, die die Lesbarkeit am ehesten erhöhen können.

Wir wollen einige Beispiele betrachten, die Grenzfälle für den Einsatz darstellen. Im ersten Beispiel geht es um eine einfache Iteration:

```
while (x != null)
    x = x.getNext();
```

Die Bedingung in der `while`-Schleife ist vermeidbar, indem die Ausnahme, dass `x` gleich `null` geworden ist, abgefangen wird:

```
try {
    while(true)
        x = x.getNext();
}
catch(NullPointerException e) {}
```

Für sehr viele Iterationen kann die zweite Variante effizienter sein als die erste, da statt einem (billigen) Vergleich in jeder Iteration nur eine einzige (teure) Ausnahmebehandlung ausgeführt wird. Trotzdem ist von einem solchen Einsatz abzuraten, weil die zweite Variante trickreich und nur schwer lesbar ist. Außerdem haben die zwei Programmstücke unterschiedliche Semantik: Das Auftreten einer `NullPointerException` während der Ausführung von `getNext` wird in der ersten Variante nicht abgefangen, in der zweiten Variante aber (ungewollt) schon. Solche nicht-lokalen Effekte sind keineswegs offensichtlich, und darauf zu achten wird oft vergessen. Es passiert leicht, dass nicht-lokale Effekte erst später hinzukommen, z. B. bei Änderung der Implementierung von `getNext`.

> *Faustregel:* Bei der Verwendung von Ausnahmen müssen nicht-lokale Effekte beachtet werden.

Das nächste Beispiel zeigt geschachtelte Typabfragen:

```
if (x instanceof T1) { ... }
else if (x instanceof T2 { ... }
...
else if (x instanceof Tn { ... }
else { ... }
```

Diese sind durch eine trickreiche, aber durchaus lesbare Verwendung von `catch`-Klauseln ersetzbar, die einer `switch`-Anweisung ähnelt:

```
try { throw x; }
catch(T1 x) { ... }
catch(T2 x) { ... }
...
catch(Tn x) { ... }
catch(Throwable x) { ... }
```

Die zweite Variante funktioniert natürlich nur wenn `x` vom Typ `Throwable` ist. Da der `try`-Block nur eine `throw`-Klausel enthält und spätestens in der letzten Zeile jede Ausnahme gefangen wird, kann es zu keinen nicht-lokalen Effekten kommen. Nach obigen Kriterien steht einer derartigen Verwendung von Ausnahmebehandlungen nichts im Weg. Allerdings entspringen beide Varianten einem schlechten Programmierstil: Typabfragen sollen, soweit es möglich ist, vermieden werden, um die Wartbarkeit zu verbessern. Wenn, wie in diesem Beispiel, nach vielen Untertypen eines gemeinsamen Obertyps unterschieden wird, ist es sinnvoll, dynamisches Binden statt Typabfragen einzusetzen. Generell sollen komplexe bedingte Anweisungen vermieden werden.

Das folgende Beispiel zeigt einen Fall, in dem die Verwendung von Ausnahmen uneingeschränkt sinnvoll ist. Angenommen, die statische Methode `addA` addiert zwei beliebig große Zahlen, die durch Zeichenketten bestehend aus Ziffern dargestellt werden. Wenn eine Zeichenkette auch andere Zeichen enthält, gibt die Funktion die Zeichenkette `"Error"` zurück:

```
public static String addA(String x, String y) {
    if (onlyDigits(x) && onlyDigits(y)) { ... }
    else
        return "Error";
}
```

Diese Art des Umgangs mit Fehlern ist problematisch, da das Ergebnis jedes Aufrufs mit `"Error"` verglichen werden muss, bevor es weiter verwendet werden kann. Wird ein Vergleich vergessen, pflanzt sich der Fehler in andere Programmzweige fort. Ausnahmen lösen das Problem:

```
public static String addB(String x, String y)
        throws NoNumberString {
    if (onlyDigits(x) && onlyDigits(y)) {... }
    else
        throw new NoNumberString();
}
```

In dieser Variante kann sich der Fehler nicht leicht fortpflanzen. Wenn ein bestimmter Ergebniswert fehlerhafte Programmzustände anzeigt, ist es fast immer ratsam, statt diesem Wert eine Ausnahme zu verwenden. Diese Verwendung ist

zwar nicht lokal, aber spezielle Ergebniswerte würden ebenso nicht-lokale Abhängigkeiten im Programm erzeugen.

Wir könnten darüber diskutieren, ob es vernünftiger wäre, statt überprüfter Ausnahmen Untertypen von `RuntimeException` zu verwenden. Dadurch könnten wir `throws`-Klauseln vermeiden, mit denen gerade in den Fällen, in denen Ausnahmen besonders sinnvoll sind, nur schwer umgegangen werden kann. Vorteile bieten überprüfte Ausnahmen ja vor allem in jenen Fällen, in denen Ausnahmen generell vermieden werden sollten. Aus diesem Grund gibt es überprüfte Ausnahmen im Wesentlichen nur (mehr) in Java. Methoden ohne `throws`-Klauseln sind flexibler einsetzbar und einfacher wartbar.

3.5.3 Zusicherungen und Ausnahmen

Zusicherungen stehen auch in Verbindung mit Ausnahmen. Betrachten wir einige Beispiele von Methoden mit Vor- und Nachbedingungen:

```
public void a(int x) { /* x >= 0 */ use(x); }
public void b(int x) { /* x >= 0 */ assert(x >= 0); use(x); }
public void c(int x) { // throws exception if x < 0
    if (x < 0) throw new IllegalArgumentException();
    use(x);
}
public void d(int x) { // x >= 0, otherwise throws exception
    if (x < 0) throw new IllegalArgumentException();
    use(x);
}
```

Methode `a` enthält eine Vorbedingung `x >= 0`, für deren Einhaltung der Aufrufer verantwortlich ist. Die Implementierung der Methode darf sich auf die Einhaltung der Bedingung verlassen. Beim Aufruf von `use` kann davon ausgegangen werden, dass die Bedingung gilt.

In Methode `b` wird die gleiche Vorbedingung durch eine `assert`-Anweisung zusätzlich überprüft; diese Überprüfung wäre nicht nötig, da der Aufrufer die Vorbedingung ohnehin erfüllen muss, eine zusätzliche Überprüfung stört aber nicht. Die Überprüfung von `assert`-Anweisungen ist im Regelfall ausgeschaltet und kann über das Interpreter-Flag `-ea` eingeschaltet werden, was bei einer verletzten Vorbedingung zum Werfen einer Ausnahme führen würde. Das Werfen dieser Ausnahme selbst ist keine Verletzung von Design-by-Contract, auch wenn sich der Aufrufer dies nicht erwartet. Der Grund dafür liegt einfach darin, dass bereits der Aufrufer durch Übergabe eines Werts kleiner 0 die Vorbedingung verletzt hat (das ist der eigentliche Fehler) und der Server bei verletzter Vorbedingung nicht mehr an seinen Teil der Verpflichtungen gebunden ist. Wie bei `a` muss auch bei `b` ausschließlich der Client dafür sorgen, dass Verletzungen der Bedingung nicht vorkommen.

Methode `c` enthält dagegen eine Nachbedingung, weil das Werfen einer Ausnahme ganz eindeutig ein Teil des Programmverhaltens ist, das der Server verantworten muss. Es wird klar ausgedrückt, unter welchen Bedingungen sich

der Client eine Ausnahme erwarten darf bzw. muss; lediglich die Art der Ausnahme könnte genauer spezifiziert sein. Der Client darf `c` mit einem negativen Argument aufrufen und der Server muss dafür sorgen, dass in diesem Fall eine Ausnahme geworfen wird. Manchmal wird argumentiert, dass in einem Fall wie `c` eine Vorbedingung ausgedrückt werden würde, weil Ausnahmen ja nicht auftreten sollen und daher kein wesentlicher Unterschied zu *b* besteht. Dem muss entgegenhalten werden, dass die Art der Formulierung sehr wohl relevante Unterschiede ergibt. Der Bedeutungsunterschied liegt darin, wer für die Beseitigung etwaiger Fehler verantwortlich ist, Client oder Server.

Die Beschreibung des Auftretens einer Ausnahme wird manchmal fälschlich den Vorbedingungen zugerechnet, weil eine Ausnahme eine Verpflichtung an den Client darstellt, mit der Ausnahme umzugehen. Bei überprüften Ausnahmen wird das besonders deutlich – ein Grund, überprüfte Ausnahmen kritisch zu betrachten. Der aktive Teil (das, was alle Teile eines Ausnahmebehandlungs-Mechanismuses in Gang setzt) ist das Werfen der Ausnahme, nicht das mögliche Abfangen, weswegen es sich in `c` um eine Nachbedingung handelt.

Methode `d` ist ein schwieriger Fall. Einerseits ist `x >= 0` als Vorbedingung zu lesen, andererseits wird im „otherwise“ eine Nachbedingung ausgedrückt. Das bedeutet, der Aufrufer muss für passende Argumente sorgen und der Server im Falle einer Verletzung der Vorbedingung eine Ausnahme werfen. Sollte `use` mit einem Wert kleiner 0 aufgerufen werden (ohne zurückgegebe Ausnahme), hätte sowohl der Client als auch der Server einen Fehler gemacht und müsste jeweils für die Beseitigung seines Teils des Fehlers sorgen. Bei `d` dürfen wir nicht argumentieren, dass der Server nach einer Verletzung der Vorbedingung machen kann, was er will, weil die Nachbedingung sich ausdrücklich auf eine Verantwortlichkeit im Fall einer verletzten Vorbedingung bezieht.

Nehmen wir an, obige Methoden `a` bis `d` hätten alle den gleichen Namen. Es stellt sich die Frage, welche dieser Methoden durch welche andere ersetzbar wäre, sodass Untertypbeziehungen gewahrt bleiben. Konkret: Wenn eine dieser Methoden in einem Obertyp vorkommen würde, welche andere Methode könnte diese in einem Untertyp überschreiben? Bei `a` und `b` haben wir identische Zusicherungen. Daher ist sowohl `a` durch `b` als auch `b` durch `a` ersetzbar; diese beiden Methoden unterscheiden sich nicht in ihrem nach außen sichtbaren Verhalten, sondern nur in Implementierungsdetails. Die anderen Fälle sind komplizierter.

Angenommen, der Obertyp enthält eine Methode entsprechend `a` (oder äquivalent dazu `b`). Im Untertyp könnte diese Methode tatsächlich durch eine Methode entsprechend `c` überschrieben werden. Die Vorbedingung `x >= 0` kann im Untertyp schwächer sein, also auch ganz weggelassen werden. Nachbedingungen können im Untertyp stärker sein, es kann also auch eine neue Nachbedingung dazukommen. Vor- und Nachbedingungen sind also kein Hinderungsgrund. Allerdings gibt es noch die implizite Bedingung, dass eine Ausnahme nur in solchen Situationen geworfen werden darf, in denen der Client das erwartet. Bei zu grober Betrachtung scheint diese Bedingung verletzt zu sein. Tatsächlich wird die Ausnahme aber nur in Situationen geworfen, in denen aus Sicht des Obertyps

die Vorbedingung verletzt ist; diese Situation darf also gar nicht auftreten. Sollte sie doch auftreten, dann gilt das bei der Beschreibung von `b` Gesagte: Bei verletzter Vorbedingung kann der Server fast alles machen, was er will. Umgekehrt (`c` im Obertyp und `a` im Untertyp) gibt es keine Ersetzbarkeit, weil es sich sowohl bei der Vor- als auch der Nachbedingung spießen würde.

Nehmen wir nun an, der Obertyp enthält eine Methode entsprechend `a` und der Untertyp entsprechend `d`. Auch in diesem Fall wäre die Untertypbeziehung erfüllt. Der einzige Unterschied zur Beziehung zwischen `a` und `c` ist der, dass die Vorbedingung erhalten bleibt. Diese Argumentation macht es leicht, die Beziehung zwischen `c` und `d` zu betrachten: Wäre `c` im Obertyp und `d` im Untertyp, hätten wir eine Verletzung der Ersetzbarkeit durch eine hinzugekommene (also stärker werdende) Vorbedingung, das geht also nicht. Aber umgekehrt geht es – `d` im Obertyp und `c` im Untertyp – weil nur eine Vorbedingung wegfällt.

Diese Beispiele zeigen, dass Einschränkungen in den Beziehungen zwischen Unter- und Obertypen (z. B., Vorbedingungen dürfen in Untertypen nur schwächer werden) ein wertvolles Hilfsmittel sind, um zu entscheiden, ob Untertypbeziehungen vorliegen oder nicht. Das funktioniert auch in Situationen, in denen die Beziehungen nicht von vorne herein klar sind. Die Beurteilung von Beziehungen zwischen Einheiten, die Ausnahmen werfen, funktioniert nach dem gleichen Prinzip. Wir müssen nur aufpassen, dass wir die Zuordnung zu den einzelnen Arten von Zusicherungen richtig vornehmen. Es ist gefährlich, bei dieser Zuordnung schlampig vorzugehen und nur oberflächliche Inhalte einzubeziehen. Hierbei ist es nötig, genaue Wortlaute zu berücksichtigen und sich zu fragen, wer (Client oder Server) in der Lage ist, für die Erfüllung einer Bedingung (ganz wörtlich genommen, nicht in irgendeinem übertragenen Sinn) zu sorgen. Aus dem Ergebnis folgt ganz klar, bei wem die Verantwortung liegt. Wer das beherrscht, kann es kaum mehr falsch machen.

4 Dynamische Typinformation und statische Parametrisierung

Statisch typisierte prozedurale und funktionale Sprachen unterscheiden streng zwischen Typinformationen, die nur dem Compiler zum Zeitpunkt der Übersetzung zur Verfügung stehen und dynamischen Daten, die während der Programmausführung verwendet werden. Es gibt in diesen Sprachen keine dynamische Typinformation. Im Gegensatz dazu wird in objektorientierten Programmiersprachen dynamische Typinformation (ähnlich wie sie auch in dynamisch typisierten prozeduralen und funktionalen Sprachen existiert) für das dynamische Binden zur Ausführungszeit benötigt. Viele objektorientierte Sprachen erlauben den direkten Zugriff darauf. In Java gibt es zur Laufzeit Möglichkeiten, die Klasse eines Objekts direkt zu erfragen, zu überprüfen, ob ein Objekt Instanz eines bestimmten Typs ist, sowie zur überprüften Umwandlung des deklarierten Typs. Wir wollen in diesem Kapitel den Umgang mit dynamischer Typinformation untersuchen. Wir behandeln auch statische Formen der Parametrisierung von Modularisierungseinheiten, gleich zu Beginn Generizität, gegen Ende des Kapitels Annotationen und aspektorientierte Programmierung. Oberflächlich betrachtet scheint es keinerlei Gemeinsamkeiten zwischen diesen Themenbereichen zu geben. Tatsächlich bestehen, insbesondere in Java, starke Verbindungen dazwischen.

4.1 Generizität

Generische Klassen, Typen und Methoden enthalten Parameter, für die Typen eingesetzt werden. Andere Arten generischer Parameter unterstützt Java nicht. Daher nennen wir generische Parameter einfach *Typparameter*.

Generizität ist ein statischer Mechanismus, der von Java erst ab Version 1.5 unterstützt wird. Dynamisches Binden wie bei Untertypen ist nicht nötig. Dieses wichtige Unterscheidungsmerkmal zu Untertypen verspricht einen effizienten Einsatz in vielen Bereichen, schränkt uns beim Programmieren aber manchmal auch auf unerwartete Weise ein.

4.1.1 Wozu Generizität?

An Stelle expliziter Typen werden im Programm Typparameter verwendet. Das sind einfach nur Namen, die später konzeptuell (oft nur scheinbar) durch Typen ersetzt werden. Anhand eines Beispiels wollen wir zeigen, dass eine Verwendung von Typparametern und die spätere Ersetzung durch Typen sinnvoll ist:

> *Beispiel:* Wir entwickeln Programmcode für Listen von Zeichenketten. Bald stellt sich heraus, dass wir auch Listen mit Elementen vom Typ `Integer` brauchen. Da der existierende Code auf Zeichenketten eingeschränkt ist, müssen wir eine neue Variante schreiben. Untertypen und Vererbung sind dabei wegen der Unterschiedlichkeit der Typen nicht hilfreich. Aber Typparameter können helfen: Statt für `String` schreiben wir den Code für `Element`. Dabei ist `Element` kein existierender Typ, sondern ein Typparameter. Den Code für `String`- und `Integer`-Listen könnten wir daraus erzeugen, indem wir alle Vorkommen von `Element` durch diese Typnamen ersetzen.

Auf den ersten Blick schaut es so aus, als ob wir den gleichen Effekt auch erzielen könnten, wenn wir im ursprünglichen `String`-Listencode alle Vorkommen von `String` durch `Integer` ersetzen würden. Leider gibt es dabei ein Problem: Der Name `String` kann auch für ganz andere Zwecke eingesetzt sein, beispielsweise als Ergebnistyp der Methode `toString()`. Eine Ersetzung würde alle Vorkommen von `String` ersetzen, auch solche, die gar nichts mit Elementtypen zu tun haben. Daher wählen wir einen neutralen Namen wie `Element`, der im Code in keiner anderen Bedeutung vorkommt. Ein Programmstück kann auch mehrere Typparameter unterschiedlicher Bedeutungen enthalten.

Natürlich spart es Schreibaufwand, wenn wir eine Kopie eines Programmstücks anfertigt und darin alle Vorkommen eines Typparameters mit Hilfe eines Texteditors durch einen Typ ersetzen. Aber dieser einfache Ansatz bereitet Probleme bei der Wartung: Nötige Änderungen des kopierten Programmstücks müssen in allen Kopien gemacht werden, was einen erheblichen Aufwand verursacht. Leichter geht es, wenn das Programmstück nur einmal existiert: Wir schreiben das Programmstück einmal und kennzeichnen Typparameter als solche. Statt einer Kopie verwenden wir nur den Namen des Programmstücks zusammen mit den Typen, die an Stelle der Typparameter zu verwenden sind. Erst der Compiler erzeugt nötige Kopien oder verwendet eine andere Technik mit ähnlichen Auswirkungen. Änderungen sind nach dem nächsten Übersetzungsvorgang überall sichtbar, wo das Programmstück verwendet wird.

In Java erzeugt der Compiler keine Kopien der Programmstücke, sondern kann durch Typumwandlungen (Casts) den gleichen Code für mehrere Zwecke – etwa Listen mit Elementen unterschiedlicher Typen – verwenden; daher hängt Generizität (als rein statischer Mechanismus) mit dynamischer Typinformation zusammen. Generizität erspart damit nicht nur Schreibarbeit, sondern kann das übersetzte Programm auch kürzer und überschaubarer machen.

4.1.2 Einfache Generizität in Java

Generische Klassen und Interfaces haben ein oder mehrere Typparameter, die in spitzen Klammern, durch Beistriche voneinander getrennt, deklariert sind. Innerhalb der Klassen und Interfaces sind diese Typparameter beinahe wie normale Referenztypen verwendbar. Das erste Beispiel in Java verwendet zwei generische Interfaces mit je einem Typparameter `A`:

```
public interface Collection<A> {
    void add(A elem);       // add elem to collection
    Iterator<A> iterator(); // create new iterator
}
public interface Iterator<A> {
    A next();               // get the next element
    boolean hasNext();      // further elements?
}
```

Mit diesen Definitionen ist `Collection<String>` ein Interface, das durch Ersetzung aller Vorkommen des Typparameters `A` im Rumpf von `Collection<A>` generiert wird. So enthält `Collection<String>` die entsprechenden generierten Methoden `void add(String elem)` und `Iterator<String> iterator()`, wobei `Iterator<String>` die Methoden `String next()` und `boolean hasNext()` enthält. Der Typparameter kann durch Referenztypen ersetzt werden, aber nicht durch elementare Typen wie `int`, `char` oder `boolean`.

Hier ist eine Implementierung von `Collection<A>` als generische Liste:

```
public class List<A> implements Collection<A> {
    private Node<A> head = null, tail = null;
    public void add(A elem) {
        if (head == null) tail = head = new Node<A>(elem);
        else tail.setNext(tail = new Node<A>(elem));
    }
    private class ListIter implements Iterator<A> {
        private Node<A> p = head;
        public boolean hasNext() { return p != null; }
        public A next() {
            if (p == null)
                throw new java.util.NoSuchElementException();
            A elem = p.elem();
            p = p.next();
            return elem;
        }
    }
    public Iterator<A> iterator() { return new ListIter(); }
}
class Node<T> {
    private T elem;
    private Node<T> next = null;
    public Node(T elem) { this.elem = elem; }
    public T elem() { return elem; }
    public Node<T> next() { return next; }
    public void setNext(Node<T> next) { this.next = next; }
}
```

Die generische Klasse `List<A>` verwendet die Hilfsklasse `Node<T>` für Listenknoten und enthält die innere Klasse `ListIter` als Iterator-Implementierung.

Der Typparameter `A` der Liste ist auch in der inneren Klasse sichtbar und wie der Name eines Typs verwendbar. Da für die Listenknoten keine geschachtelte Klasse verwendet wird, ist dafür ein eigener Typparameter nötig. Wir könnten dafür ebenso `A` verwenden, aber auch jeden beliebigen anderen Namen. Wir verwenden `T`, um die Gültigkeitsbereiche sichtbar zu machen. Durch `interface Collection<A>` wird der Typparameter `A` eingeführt, der innerhalb dieses Interfaces sichtbar ist. Durch `interface Iterator<A>` wird ein anderer Typparameter (der auch `A` heißt) eingeführt, der nur innerhalb des anderen Interfaces sichtbar ist. Ebenso wird durch `class List<A>` ein weiterer Typparameter namens `A` eingeführt, der bis zum Ende der Klasse sichtbar ist. Durch `Node<T>` wird der Typparameter `T` eingeführt. An allen anderen Stellen im Beispiel, an denen `A` oder `T` vorkommt, werden keine Typparameter eingeführt, sondern zuvor eingeführte Typparameter verwendet. In `implements Collection<A>` sowie in der Variablendeklaration `Node<A> head` wird der durch `class List<A>` eingeführte Typparameter verwendet. Wenn wir irgendwo im Programm den Typ `List<String>` verwenden (wodurch der mit `List` eingeführte Typparameter durch `String` ersetzt wird), dann implementiert `List<String>` das Interface `Collection<String>` und hat die Variable `head` vom Typ `Node<String>`, also auch `T` wird durch `String` ersetzt. Mit dem Wort „ersetzt“ ist dabei eine Form der Parameterübergabe gemeint (`String` als Argument, `A` oder `T` als Parameter), genau so wie im λ-Kalkül die Parameterübergabe durch Ersetzung erfolgt. Wir machen nichts falsch, wenn wir das Ersetzen einfach umgangssprachlich verstehen und uns nicht weiter um eine formale Definition kümmern.

Konstruktoren haben wie in `Node<T>` die üblich Syntax `Node(...){...}`; es werden keine Typparameter angegeben. Bei der Objekterzeugung müssen aber Typen angegeben werden, die die Typparameter der Klasse ersetzen, etwa `new Node<A>(...)`; dabei ist `A` der Typ der Listenelemente.

Folgendes Programmstück zeigt den Umgang mit generischen Klassen:

```
class ListTest {
    public static void main(String[] args) {
        List<Integer> xs = new List<Integer>();
        xs.add(new Integer(0));   // oder xs.add(0);
        Integer x = xs.iterator().next();

        List<String> ys = new List<String>();
        ys.add("zerro");
        String y = ys.iterator().next();

        List<List<Integer>> zs = new List<List<Integer>>();
        zs.add(xs);
        // zs.add(ys);  !! Compiler meldet Fehler !!
        List<Integer> z = zs.iterator().next();
    }
}
```

An `ListTest` fällt auf, dass statt einfacher Werte von `int` Objekte der Standardklasse `Integer` verwendet werden müssen, da gewöhnliche Zahlen keine Referenzobjekte sind. In Java gibt es zu jedem elementaren Typ wie `int`, `char` oder `boolean` einen Referenztyp wie `Integer`, `Character` oder `Boolean`, weil in einigen Sprachkonstrukten nur Referenztypen erlaubt sind. Sie bieten die gleiche Funktionalität wie elementare Typen. Ein Nachteil ist der im Vergleich zu elementaren Werten weniger effiziente Umgang mit Objekten.

Java unterstützt *Autoboxing* und *Autounboxing*. Dabei erfolgt die Umwandlung zwischen Typen wie `int` und `Integer` bei Bedarf automatisch in beide Richtungen. Statt `xs.add(new Integer(0))` schreiben wir einfach `xs.add(0)`. Die automatische Umwandlung verringert nur den Schreibaufwand, nicht die dadurch bedingte Ineffizienz zur Laufzeit.[1]

Das Beispiel zeigt, dass Listen auch andere Listen enthalten können. Jedoch muss jedes Listenelement den durch den Typparameter festgelegten Typ haben. Der Compiler ist klug genug, um `List<Integer>` von `List<String>` zu unterscheiden. Diese beiden Listentypen sind nicht kompatibel zueinander.

Generizität bietet statische Typsicherheit. Bereits der Compiler garantiert, dass in ein Objekt von `List<String>` nur Zeichenketten eingefügt werden können. Der Versuch, ein Objekt eines inkompatiblen Typs einzufügen, wird als Fehler gemeldet. Wer den Umgang mit Collections und Ähnlichem ohne Generizität gewohnt ist, kennt die Probleme mangelnder statischer Typsicherheit, bei der Typfehler als Typkonvertierungsfehler zur Laufzeit auftreten. Generizität sorgt dafür, dass solche Typfehler schon zur Übersetzungszeit erkannt werden.

Auch Methoden können generisch sein, wie das nächste Beispiel zeigt:

```
public interface Comparator<A> {
    int compare(A x, A y);  // x <  y if result <  0
                            // x == y if result == 0
                            // x >  y if result >  0
}
public class CollectionOps {
    public static <A> A max(Collection<A> xs, Comparator<A> c) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (c.compare(w, x) < 0)
                w = x;
        }
        return w;
    }
}
```

[1] Es stimmt nicht, dass Autoboxing in jedem Fall ein neues Objekt erzeugt. Für häufig verwendete Zahlen in der Nähe von 0 stellt das Java-Laufzeitsystem schon vorher erzeugte Objekte bereit, die direkt übergeben werden. Das bedeutet jedoch, dass wir uns bei Autoboxing nicht auf die Objektidentität verlassen dürfen.

Die Methode `compare` in `Comparator<A>` vergleicht zwei Objekte des gleichen Typs und retourniert das Vergleichsergebnis als ganze Zahl. Unterschiedliche Komparatoren, also voneinander verschiedene Objekte mit einem solchen Interface, werden unterschiedliche Vergleiche durchführen. Die statische Methode `max` in `CollectionOps` wendet Komparatoren wiederholt auf Elemente in einem Objekt von `Collection<A>` an, um das größte Element zu ermitteln. Am vor dem Ergebnistyp von `max` eingefügten Ausdruck `<A>` ist erkennbar, dass `max` eine generische Methode mit einem Typparameter `A` ist. Dieser Typparameter kommt sowohl als Ergebnistyp als auch in der Parameterliste und im Rumpf der Methode vor. In den spitzen Klammern können auch mehrere, durch Komma voneinander getrennte Typparameter deklariert sein.

Generische Methoden haben den Vorteil, dass die für Typparameter zu verwendenden Typen nicht explizit angeben sein müssen:

```
    List<Integer> xs = ...;
    List<String>  ys = ...;

    Comparator<Integer> cx = ...;
    Comparator<String>  cy = ...;

    Integer rx = CollectionOps.max(xs, cx);
    String  ry = CollectionOps.max(ys, cy);
    // Integer rz = CollectionOps.max(xs, cy); !Fehler!
```

Der Compiler erkennt durch Typinferenz anhand der Typdeklarationen von `xs` und `cx` beziehungsweise `ys` und `cy`, dass beim ersten Aufruf von `max` für den Typparameter `Integer` und für den zweiten Aufruf `String` zu verwenden ist. Außerdem erkennt der Compiler statisch, wenn der Typparameter von `List` nicht mit dem von `Comparator` übereinstimmt.

Hier ist ein Beispiel für die Implementierung eines einfachen Komparators:

```
class IntComparator implements Comparator<Integer> {
    public int compare(Integer x, Integer y) {
        return x.intValue() - y.intValue();
    }
}
```

Aufgrund von Autounboxing kann die dritte Zeile auch einfach durch die Anweisung `return x - y;` ersetzt werden. Ein Komparator für Zeichenketten wird zwar etwas komplizierter, aber nach dem gleichen Schema aufgebaut sein.

4.1.3 Gebundene Generizität in Java

Die einfache Form der Generizität ist zwar elegant und sicher, aber für einige Verwendungszwecke nicht ausreichend: Im Rumpf einer einfachen generischen Klasse oder Methode ist über den Typ, der den Typparameter ersetzt, nichts bekannt. Insbesondere ist nicht bekannt, ob Objekte dieses Typs bestimmte Methoden oder Variablen haben.

Schranken. Über manche Typparameter benötigen wir mehr Information, um auf Objekte der entsprechenden Typen zugreifen zu können. Gebundene Typparameter liefern diese Information: In Java kann für jeden Typparameter eine Klasse und beliebig viele Interfaces als *Schranken* angegeben werden. Nur Untertypen der Schranken dürfen den Typparameter ersetzen. Damit ist statisch bekannt, dass in jedem Objekt des Typs, für den der Typparameter steht, die in den Schranken festgelegten öffentlich sichtbaren Methoden und Variablen verwendbar sind. Objekte des Typparameters können wie Objekte der Schranken (jede Schranke ist ein Typ der Objekte) verwendet werden:

```
public interface Scalable {
    void scale(double factor);
}
public class Scene<T extends Scalable> implements Iterable<T> {
    public void addSceneElement(T e) { ... }
    public Iterator<T> iterator() { ... }
    public void scaleAll(double factor) {
        for (T e : this)
            e.scale(factor);
    }
    ...
}
```

Die Klasse Scene hat einen Typparameter T mit einer Schranke. Jeder Typ, der T ersetzt, ist Untertyp von Scalable und unterstützt damit die Methode scale. Diese Methode wird in scaleAll aufgerufen (für jedes Element des aktuellen Objekts von Scene).

Schranken stehen nach dem Schlüsselwort extends innerhalb der spitzen Klammern. Das Schlüsselwort dafür ist immer extends, niemals implements. Pro Typparameter sind als Schranken eine Klasse sowie beliebig viele Interfaces erlaubt, jeweils durch & voneinander getrennt. Nur solche Typen dürfen den Typparameter ersetzen, die alle diese Interfaces erweitern bzw. implementieren. Ist ein Typparameter ungebunden, das heißt, ist keine Schranke angegeben, wird Object als Schranke angenommen, da jede Klasse von Object abgeleitet ist. Die in Object definierten Methoden sind daher immer verwendbar.

In obigem Beispiel erweitert Scene das in den Java-Bibliotheken vordefinierte Interface Iterable<T>, welches die Methode iterator zur Erzeugung eines Iterators beschreibt. In Scene wird der Iterator benötigt, um einfach mittels for-Schleife über alle Elemente des aktuellen Objekts von Scene zu iterieren.

Rekursion. Diese Beispiels-Variante verwendet Typparameter rekursiv:

```
public interface Comparable<A> {
    int compareTo(A that);  // this <  that if result <  0
                            // this == that if result == 0
                            // this >  that if result >  0
}
```

```
public class Integer implements Comparable<Integer> {
    private int value;
    public Integer(int value) { this.value = value; }
    public int intValue()  { return value; }
    public int compareTo(Integer that) {
        return this.value - that.value;
    }
}
public class CollectionOps2 {
    public static <A extends Comparable<A>>
                  A max(Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0)
                w = x;
        }
        return w;
    }
}
```

Diese Klasse `Integer` ist eine vereinfachte Form der in Java standardmäßig vorhandenen gleichnamigen Klasse. `Integer` wird von `Comparable<Integer>` abgeleitet. Der Name der Klasse kommt in der Schnittstelle vor, von der abgeleitet wird. Auf den ersten Blick mag eine derartige rekursive Verwendung von Klassennamen eigenartig erscheinen, sie ist aber klar definiert, einfach verständlich und in der Praxis sinnvoll. In der Schranke des Typparameters `A` von `max` in `CollectionOps2` kommt eine ähnliche Rekursion vor. Damit wird eine Ableitungsstruktur wie die von `Integer` beschrieben. Diese Form der Generizität mit rekursiven Typparametern nennen wir *F-gebundene Generizität* nach dem formalen Modell, in dem solche Konzepte untersucht wurden: System $F_{\leq}$, ausgesprochen „F-bound“ [7].

Keine impliziten Untertypen. Generizität unterstützt keine impliziten Untertypbeziehungen. So besteht zwischen `List<X>` und `List<Y>` keine Untertypbeziehung wenn `X` und `Y` verschieden sind, auch dann nicht, wenn `Y` Untertyp von `X` ist oder umgekehrt. Natürlich gibt es die expliziten Untertypbeziehungen, wie beispielsweise die zwischen `Integer` und `Comparable<Integer>`. Wir können Klassen wie üblich ableiten:

```
class MyList<A> extends List<List<A>> { ... }
```

Dann ist `MyList<String>` ein Untertyp von `List<List<String>>`. Jedoch ist `MyList<X>` kein Untertyp von `List<Y>` wenn `Y` möglicherweise ungleich `List<X>` ist. Die Annahme impliziter Untertypbeziehungen ist ein häufiger Anfängerfehler. Wir müssen stets bedenken, dass es weder in Java noch in irgendeiner anderen Sprache sichere implizite Untertypbeziehungen dieser Art geben kann.

In Java können bei Verwendung von Arrays Typfehler zur Laufzeit auftreten, da Arrays implizite Untertypbeziehungen unterstützen:

```
class Loophole {
    public static String loophole(Integer y) {
        String[] xs = new String[10];
        Object[] ys = xs; // no compile-time error
        ys[0] = y;        // throws ArrayStoreException
        return xs[0];
    }
}
```

Diese Klasse wird unbeanstandet übersetzt, da in Java für jede Untertypbeziehung auf Typen automatisch eine Untertypbeziehung auf Arrays von Elementen solcher Typen angenommen wird, obwohl Ersetzbarkeit verletzt sein kann. Im Beispiel nimmt der Compiler an, dass `String[]` Untertyp von `Object[]` ist, da `String` ein Untertyp von `Object` ist. Diese Annahme ist falsch. Generizität schließt solche Fehler durch das Verbot impliziter Untertypbeziehungen aus:

```
class NoLoophole {
    public static String loophole(Integer y) {
        List<String> xs = new List<String>();
        List<Object> ys = xs; // compile-time error
        ys.add(y);
        return xs.iterator().next();
    }
}
```

Wildcards. Die Sicherheit durch Nichtunterstützung impliziter Untertypbeziehungen hat auch einen Nachteil. Zum Beispiel kann die Methode

```
    void drawAll(List<Polygon> p) {
        ... // draws all polygons in list p
    }
```

nur mit Argumenten vom Typ `List<Polygon>` aufgerufen werden, nicht aber mit Argumenten vom Typ `List<Triangle>` oder `List<Square>` (entsprechend dem Beispiel aus Abschnitt 3.2.3). Dies ist bedauerlich, da `drawAll` nur Elemente aus der Liste liest und nie in die Liste schreibt, Sicherheitsprobleme durch implizite Untertypbeziehungen wie bei Arrays aber beim Schreiben auftreten. Für solche Fälle unterstützt Java *gebundene Wildcards* als Typen, die Typparameter ersetzen:

```
    void drawAll(List<? extends Polygon> p) { ... }
```

Das Fragezeichen steht für einen beliebigen Typ, der ein Untertyp von `Polygon` ist. Nun können wir `drawAll` auch mit Argumenten vom Typ `List<Triangle>` und `List<Square>` aufrufen. Der Compiler liefert eine Fehlermeldung, wenn

die Möglichkeit besteht, dass in den Parameter `p` geschrieben wird. Genauer gesagt erlaubt der Compiler die Verwendung von `p` nur an Stellen, für deren Typen in Untertypbeziehungen Kovarianz gefordert ist (Lesezugriffe, siehe Abschnitt 3.1.1). Durch diese Überprüfung ist die zweite Variante von `drawAll` genau so sicher wie die erste.

Gelegentlich gibt es auch Parameter, deren Inhalte in einer Methode nur geschrieben und nicht gelesen werden:

```
void addSquares(List<? extends Square> from,
                List<? super Square> to    ) {
    ... // add squares from 'from' to 'to'
}
```

In `to` wird nur geschrieben, von `to` wird nicht gelesen. Als Argument für `to` können wir daher `List<Square>`, aber auch `List<Polygon>` und `List<Object>` angeben. Als Schranke spezifiziert das Schlüsselwort `super`, dass jeder Obertyp von `Square` erlaubt ist. Der Compiler erlaubt die Verwendung von `to` nur an Stellen, für deren Typen in Untertypbeziehungen Kontravarianz gefordert ist; das sind Schreibzugriffe.

Nebenbei sei erwähnt, dass auch nur `?` als Wildcard verwendbar ist. Entsprechende Variablen und Parameter unterstützen nur das Lesen, aber gelesene Werte haben einen unbekannten Typ; `<?>` entspricht somit `<? extends Object>`.

Flexibilität durch Wildcards. In der Praxis kann die Verwendung solcher Wildcards recht kompliziert werden, wie folgendes Beispiel zeigt:

```
public class MaxList<A extends Comparable<? super A>>
        extends List<A> {
    public A max() {
        Iterator<A> i = this.iterator();
        A w = i.next();
        while (i.hasNext()) {
           A x = i.next();
           if (w.compareTo(x) < 0)
               w = x;
        }
        return w;
    }
}
```

Eine Schranke `A extends Comparable<A>` scheint auf den ersten Blick klarer auszudrücken, dass auf den Listenelementen `compareTo` benötigt wird. Aber mit der einfacheren Lösung haben wir ein Problem: Wir könnten einen Typ `MaxList<X>` zwar verwenden wenn `X` das Interface `Comparable<X>` implementiert, aber für einen von `X` abgeleiteten Typ `Y` wäre `MaxList<Y>` nicht erlaubt, da `Y` nur `Comparable<X>` implementiert, nicht `Comparable<Y>`. Ein Interface darf nicht mehrfach auf unterschiedliche Weise implementiert sein, sodass `Y` nicht sowohl `Comparable<X>` als auch `Comparable<Y>` implementieren

kann. Doch `<A extends Comparable<? super A>` erlaubt die Verwendung von `MaxList<Y>`: Es reicht, wenn `Y` nur `Comparable<X>` implementiert, da `X` Obertyp von `Y` ist und `?` für einen Obertyp von `A` steht.

Die Methode `compareTo` aus `Comparable` greift nur lesend auf ihr Argument zu. Für lesende Zugriffe verwenden wir jedoch meist `extends`-Wildcards, nicht wie im Beispiel ein `super`-Wildcard auf `Comparable`. Für diese Diskrepanz gibt es eine Erklärung: Entscheidend sind kovariante bzw. kontravariante Parameterpositionen. Die Richtung (ko- und kontravariant) dreht sich mit jeder *Klammerebene* um. Das heißt, in einem Wildcard in den äußersten spitzen Klammern stimmt „lesend" mit „kovariant" und „schreibend" mit „kontravariant" überein, innerhalb von zwei spitzen Klammern „lesend" mit „kontravariant" und „schreibend" mit „kovariant", innerhalb von drei spitzen Klammern wieder so wie in den äußersten Klammern und so weiter. Als Menschen können wir nach komplizierten Überlegungen zwar nachvollziehen, wie die Richtung mit den Klammerebenen zusammenhängt, aber intuitiv ist das nicht. Maschinen können damit problemloser umgehen. Am besten verlassen wir uns dabei auf den Compiler, der zuverlässig warnt, wenn wir einen falschen Wildcard verwenden, vorausgesetzt Warnungen für Generizität sind eingeschaltet.

Einschränkungen in Java. Generizität wurde mit minimalen Änderungen der Sprache zum ursprünglich nicht generischen Java hinzugefügt. Auf Grund von Kompatibilitätsbedingungen mussten Kompromisse gemacht werden, die die Verwendbarkeit von Generizität einschränken. Generell, also in anderen Sprachen als Java (und C#), treten beispielsweise keine Unterschiede in der Typsicherheit von Arrays und generischen Collections auf. Im Gegenteil: Der einfache und sichere Umgang mit Arrays ist ein Grund für die Einführung von Generizität. Als weitere Einschränkung in Java können Typparameter nicht zur Erzeugung neuer Objekte verwendet werden. Daher ist `new A()` illegal wenn `A` ein Typparameter ist. In der Praxis interessanter ist der Ausdruck `new A[n]`, der ein neues Array für `n` Objekte von `A` erzeugt. Dieser Ausdruck ist leider nicht typsicher wenn `A` ein Typparameter ist; der Compiler meldet einen Fehler. Weitere Einschränkungen der Generizität in Java gibt es bei expliziten Typumwandlungen und dynamischen Typvergleichen, wie wir später sehen werden.

Zwecks Kompatibilität zu älteren Java-Versionen werden zur Laufzeit Typprüfungen durchgeführt, obwohl der Compiler Typkonsistenz garantiert. Daher bedingt Generizität (sehr kleine) Einbußen an Laufzeiteffizienz. In anderen Programmiersprachen hat Generizität keinen negativen Einfluss auf die Laufzeit.

4.2 Verwendung von Generizität

Wir wollen nun betrachten, wie Generizität in der Praxis einsetzbar ist. Abschnitt 4.2.1 gibt einige allgemeine Ratschläge, in welchen Fällen sich die Verwendung auszahlt. In Abschnitt 4.2.2 beschäftigen wir uns mit möglichen Übersetzungen generischer Klassen und einigen Alternativen zur Generizität, um ein etwas umfassenderes Bild davon zu bekommen, was Generizität leisten kann.

4.2.1 Richtlinien für die Verwendung von Generizität

Generizität ist immer sinnvoll, wenn sie die Wartbarkeit verbessert. Aber oft ist nur schwer entscheidbar, ob diese Voraussetzung zutrifft. Wir wollen hier einige typische Situationen als Entscheidungshilfen bzw. Faustregeln anführen:

Gleich strukturierte Klassen und Methoden. Wir sollen Generizität immer verwenden, wenn es mehrere gleich strukturierte Klassen (oder Typen) beziehungsweise Methoden gibt. Typische Beispiele dafür sind Containerklassen wie Listen, Stacks, Hashtabellen, Mengen, etc. und Methoden, die auf Containerklassen zugreifen, etwa Suchfunktionen und Sortierfunktionen. Alle bisher in diesem Kapitel verwendeten Klassen und Methoden fallen in diese Kategorie. Wenn es eine Containerklasse für Elemente eines bestimmten Typs gibt, liegt immer der Verdacht nahe, dass genau dieselbe Containerklasse auch für Objekte anderer Typen sinnvoll sein könnte. Falls die Typen der Elemente in der Containerklasse gleich von Anfang an als Typparameter spezifiziert sind, ist es später leicht, die Klasse unverändert mit Elementen anderer Typen zu verwenden.

Faustregel: Containerklassen sollen generisch sein.

Es zahlt sich aus, Generizität bereits beim ersten Verdacht, dass eine Containerklasse auch für andere Elementtypen sinnvoll sein könnte, zu verwenden. Beim Erstellen der Klasse ist es leicht, zwischen Elementtypen und anderen Typen zu unterscheiden. Im Nachhinein, also wenn eine nichtgenerische Klasse in eine generische umgewandelt werden soll, ist diese Unterscheidung nicht immer so einfach. Generizität kann die Lesbarkeit in komplexen Fällen beeinträchtigen, zahlt sich aber trotzdem aus.

Wenn wir die Sinnhaftigkeit von Typparametern erst später erkennen, sollen wir das Programm gleich refaktorisieren, also mit Typparametern versehen. Ein Hinauszögern der Refaktorisierung führt leicht zu unnötigem Code.

Üblicher Programmcode definiert relativ wenige generische Containerklassen. Das liegt daran, dass Programmierumgebungen mit umfangreichen Bibliotheken ausgestattet sind, welche die am häufigsten verwendeten, immer wieder gleich strukturierten Klassen und Methoden bereits enthalten. Wir müssen diese Klassen und Methoden nicht neu schreiben.

Faustregel: Klassen und Methoden in Bibliotheken sind generisch.

In aktuellen Java-Versionen sind die Standardbibliotheken durchwegs generisch. Generizität ist in Java so gestaltet, dass der Umstieg auf Generizität möglichst leicht ist. Übersetzte generische Klassen können auch in älteren, nichtgenerischen Java-Versionen verwendet werden, und generisches Java kann mit nicht-generischen Klassen umgehen.

Erkennen gleicher Strukturen. Beispielsweise schreiben wir eine Klasse, die Konten an einem Bankinstitut repräsentiert. Konten können für unterschiedliche Währungen ausgelegt sein. In der Regel werden wir einen Typ `Currency`

einführen und für jede konkrete Währung wie Euro und US-Dollar einen Untertyp davon. Rechtliche Bedingungen für unterschiedliche Währungen werden sich wahrscheinlich so stark voneinander unterscheiden, dass wir für jede Währung tatsächlich einen eigenen Typ mit etwas unterschiedlichem Verhalten brauchen. Im Konto haben wir eine Variable, die ein Objekt des entsprechenden Währungstyps enthält. So weit können wir unsere Konto-Klasse sehr gut mittels Untertypbeziehungen ohne Generizität darstellen. Jetzt ergibt sich aber das Problem, dass wir in der Kontoklasse vielleicht mehrere Variablen mit Objekten von Währungstypen benötigen, von denen einige den gleichen Währungstyp haben müssen. Es stellt sich rasch heraus, dass das über Untertypbeziehungen nicht oder nur sehr umständlich darstellbar ist, etwa weil sich automatisch die Notwendigkeit kovarianter Eingangsparameter ergibt (kovariantes Problem). Subtyping ist dafür einfach nicht der richtige Ansatz. Mittels Generizität lässt sich diese Problematik relativ einfach in den Griff bekommen. Wir müssen nur für jede Art von Variable, die den gleichen Währungstyp haben muss, einen Typparameter anlegen. Diese Typparameter werden jeweils `Currency` als Schranke haben und können durch (möglicherweise unterschiedliche) Untertypen von `Currency` ersetzt werden. Trotzdem sind die notwendigen Konsistenzbedingungen zwischen den Variablen garantiert.

Ein Konto ist auch eine Art von Container. Es geht darum, dass dies nicht von vornherein sichtbar sein muss. Wir müssen unsere Augen dafür offen halten, wo sich Situationen ergeben, die so wie in einem Container handhabbar sind. Die Form der Beschreibung eines Typs macht es manchmal schwer, die Container-Eigenschaft zu erkennen. Ein deutlicher Hinweis darauf besteht darin, dass mehrere Variablen Inhalte nicht von vorne herein klar festgelegter, aber dennoch gleicher Typen enthalten.

> *Faustregel:* Generizität (oft gebundene Generizität) ist immer dort sinnvoll, wo mehrere Variablen vom gleichen (aber nicht von Anfang an fix festgelegten) Typ notwendig sind.

Abfangen erwarteter Änderungen. Mittels Generizität können wir erwartete Programmänderungen vereinfachen. Das gilt auch für Typen von formalen Parametern, die sich entsprechend dem Ersetzbarkeitsprinzip nicht beliebig ändern dürfen. Wir sollen Typparameter für die Typen formaler Parameter verwenden, wenn wir erwarten, dass sich diese Typen im Laufe der Zeit ändern. Es brauchen nicht gleichzeitig mehrere gleich strukturierte Klassen oder Methoden sinnvoll sein, sondern es reicht, wenn zu erwarten ist, dass sich Typen in unterschiedlichen Versionen voneinander unterscheiden.

> *Faustregel:* Wir sollen Typparameter als Typen formaler Parameter verwenden, wenn Änderungen der Parametertypen absehbar sind.

Wir können obiges Beispiel zu Bankkonten etwas abwandeln. Nehmen wir an, unsere Bank kennt anfangs nur Konten über Euro-Beträge, wodurch alle entsprechenden Variablen den Typ der Euro-Währung haben (keine Generizität nötig). Wenn zu erwarten ist, dass künftig auch Konten über US-Dollar-Beträge

gebraucht werden könnten, wäre es günstig, vor allem die Schnittstellen gleich von Anfang an so zu gestalten, dass die spätere Einführung anderer Währungen vereinfacht wird. Das heißt, wie oben werden gleich Typparameter eingeschränkt auf `Currency` eingeführt, auch wenn diese Typparameter anfangs alle nur durch den Typ der Euro-Beträge ersetzt werden.

Untertypbeziehungen und Generizität sind in Java miteinander verknüpft. Die sinnvolle Verwendung gebundener Generizität setzt das Bestehen geeigneter Untertypbeziehungen voraus. Eine weitere Parallele zwischen Generizität und Untertypbeziehungen ist erkennbar: Sowohl Generizität als auch Untertypbeziehungen helfen, notwendige Änderungen im Programmcode klein zu halten. Generizität und Untertypbeziehungen ergänzen sich dabei: Generizität ist auch dann hilfreich, wenn das Ersetzbarkeitsprinzip nicht erfüllt ist, während Untertypbeziehungen den Ersatz eines Objekts eines Obertyps durch ein Objekt eines Untertyps auch unabhängig von Parametertypen ermöglichen.

> *Faustregel:* Generizität und Untertyprelationen ergänzen sich. Wir sollen stets überlegen, ob wir eine Aufgabe besser durch Ersetzbarkeit, durch Generizität, oder (häufig sinnvoll) eine Kombination aus beiden Konzepten lösen.

Es sollte aber auch stets klar sein, dass nur Untertypen Ersetzbarkeit gewährleisten können. Bei Generizität bedingt eine Änderung im Server-Code in der Regel auch Änderungen im Code aller Clients. Sinnvoll eingesetzte Untertypen können das vermeiden.

Verwendbarkeit. Generizität und Untertypbeziehungen sind oft gegeneinander austauschbar. Das heißt, wir können ein und dieselbe Aufgabe mit Generizität oder über Untertypbeziehungen lösen. Es stellt sich die Frage, ob nicht generell ein Konzept das andere ersetzen kann. Das geht nicht, wie wir an folgenden zwei Beispielen sehen:

Generizität ist sehr gut dafür geeignet, wie in obigen Beispielen eine Listenklasse zu schreiben, wobei ein Objekt nur Elemente eines Typs enthält und ein anderes nur Elemente eines anderen Typs. Dabei ist statisch sichergestellt, dass alle Elemente in einer Liste den gleichen Typ haben. Solche Listen sind *homogen*. Ohne Generizität ist es nicht möglich, eine solche Klasse zu schreiben. Zwar können wir auch ohne Generizität Listen erzeugen, die Elemente beliebiger Typen enthalten, aber es ist nicht statisch sichergestellt, dass alle Elemente den gleichen Typ haben. Daher können wir mit Hilfe von Generizität etwas machen, was ohne Generizität, also nur durch Untertypbeziehungen, nicht machbar wäre.

Mit Generizität ohne Untertypbeziehungen ist es nicht möglich, eine Listenklasse zu schreiben, in der Elemente unterschiedliche Typen haben können. Solche Listen sind *heterogen*. Daher können wir mit Hilfe von Untertypbeziehungen etwas machen, was ohne sie, also nur durch Generizität, nicht machbar wäre. Generizität und Untertypbeziehungen ergänzen sich.

Diese Beispiele zeigen, was mit Generizität oder Untertypbeziehungen alleine nicht machbar ist. Sie zeigen damit auf, in welchen Fällen wir Generizität bzw. Untertypen zur Erreichung des Ziels unbedingt brauchen.

Untertypbeziehungen ermöglichen durch die Ersetzbarkeit Codeänderungen mit klaren Grenzen, welche Klassen davon betroffen sein können. Generizität kann die von Änderungen betroffenen Bereiche nicht eingrenzen, da im Client-Code Typparameterersetzungen spezifiziert werden müssen. Wenn es um die Entscheidung zwischen Untertypbeziehungen und Generizität geht, sollten wir daher Untertypbeziehungen vorziehen.

Laufzeiteffizienz. Die Verwendung von Generizität hat keine oder zumindest kaum negative Auswirkungen auf die Laufzeiteffizienz. Andererseits ist die Verwendung von dynamischem Binden im Zusammenhang mit Untertypbeziehungen immer etwas weniger effizient als statisches Binden. Aufgrund dieser Überlegungen kommt immer wieder jemand auf die Idee, stets Generizität einzusetzen, aber dynamisches Binden nur dort zuzulassen, wo es unumgänglich ist. Da Generizität und Untertypbeziehungen oft gegeneinander austauschbar sind, ist das im Prinzip machbar. Leider sind die tatsächlichen Beziehungen in der relativen Effizienz von Generizität und dynamischem Binden keineswegs so einfach wie hier dargestellt. Durch die Verwendung von Generizität zur Vermeidung von dynamischem Binden ändert sich die Struktur des Programms, wodurch sich die Laufzeiteffizienz wesentlich stärker (eher negativ als positiv) ändern kann als durch die Vermeidung von dynamischem Binden. Wenn beispielsweise eine `switch`-Anweisung zusätzlich ausgeführt werden muss, ist die Effizienz ziemlich sicher schlechter geworden.

> *Faustregel:* Wir sollen Überlegungen zur Laufzeiteffizienz beiseite lassen, wenn es um die Entscheidung zwischen Generizität und Untertypbeziehungen geht.

Solche Optimierungen auf der untersten Ebene erfordern sehr viel Expertenwissen über Details von Compilern und Hardware und sind in der Regel nicht portabel. Viel wichtiger ist es, auf die Einfachheit und Verständlichkeit zu achten. Wenn Effizienz entscheidend ist, müssen wir vor allem die Effizienz der Algorithmen betrachten.

Natürlichkeit. Häufig bekommen wir auf die Frage, ob in einer bestimmten Situation Generizität oder Subtyping einzusetzen sei, die Antwort, dass der *natürlichere* Mechanismus am besten geeignet sei. Für erfahrene Leute ist diese Antwort häufig zutreffend: Mit einem gewissen Erfahrungsschatz kommt es ihnen selbstverständlich vor, den richtigen Mechanismus zu wählen ohne die Entscheidung begründen zu müssen. Hinter der Natürlichkeit verbirgt sich der Erfahrungsschatz. Mit wenig einschlägiger Erfahrung sehen wir kaum, was natürlicher ist. Es zahlt sich in jedem Fall aus, genau zu überlegen, was wir mit Generizität erreichen wollen und erreichen können. Wenn wir uns zwischen Generizität und Subtyping entscheiden sollen, ist es angebracht, auch eine Kombination von Generizität und Subtyping ins Auge zu fassen. Erst wenn diese Überlegungen zu keinem eindeutigen Ziel führen, entscheiden wir uns für die natürlichere Alternative.

4.2.2 Arten der Generizität

Bisher haben wir Generizität als ein einziges Sprachkonzept betrachtet. Tatsächlich gibt es zahlreiche Varianten mit unterschiedlichen Eigenschaften. Wir wollen hier einige Varianten miteinander vergleichen.

Für die Übersetzung generischer Klassen und Methoden in ausführbaren Code gibt es im Großen und Ganzen zwei Möglichkeiten, die homogene und die heterogene Übersetzung. In Java wird die *homogene* Übersetzung verwendet. Dabei wird jede generische Klasse, so wie auch jede nicht-generische Klasse, in genau eine Klasse mit JVM-Code übersetzt. Jeder gebundene Typparameter wird im übersetzten Code durch die erste Schranke des Typparameters ersetzt, jeder ungebundene Typparameter durch `Object`. Wenn eine Methode ein Objekt eines Typparameters als Parameter nimmt oder zurückgibt, wird der Typ des Objekts vor (für Parameter) oder nach dem Methodenaufruf (für Ergebnisse) dynamisch in den Typ, der den Typparameter ersetzt, umgewandelt, wie wir in Abschnitt 4.3 sehen werden. Dies entspricht der Simulation einiger Aspekte von Generizität. Im Unterschied zur simulierten Generizität wird die Typkompatibilität vom Compiler garantiert.

Bei der *heterogenen* Übersetzung wird für jede Verwendung einer generischen Klasse oder Methode mit anderen Typparametern eigener übersetzter Code erzeugt. Die heterogene Übersetzung entspricht also eher der Verwendung von Copy-and-Paste, wie in Abschnitt 4.1.1 argumentiert. Dem Nachteil einer größeren Anzahl übersetzter Klassen und Methoden und der Duplizierung von Klassenvariablen stehen einige Vorteile gegenüber: Da für alle Typen eigener Code erzeugt wird, sind elementare Typen wie `int`, `char` und `boolean` ohne Einbußen an Laufzeiteffizienz als Ersatz für Typparameter geeignet. Zur Laufzeit müssen keine Typumwandlungen und damit zusammenhängende Überprüfungen durchgeführt werden. Außerdem sind auf jede übersetzte Klasse eigene Optimierungen anwendbar, die von den Typen abhängen. Daher bietet heterogene Übersetzung etwas bessere Laufzeiteffizienz. Heterogene Übersetzung wird beispielsweise für *Templates* in C++ verwendet.

Große Unterschiede zwischen Programmiersprachen gibt es im Zusammenhang mit gebundener Generizität. Viele Sprachen (auch Java) verlangen die Vorgabe einer Schranke, wobei nur Untertypen der Schranke den Typparameter ersetzen können. Dafür müssen geeignete Typhierarchien erstellt werden. Vor allem bei Verwendung von Typen aus vorgefertigten Bibliotheken, deren Untertypbeziehungen zueinander nicht mehr änderbar sind, ist das ein bedeutender Nachteil. Um beispielsweise `compareTo` über einen Typparameter verwenden zu können, reicht es nicht, wenn diese Methode implementiert ist; auch `Comparable<...>` muss implementiert sein und als Schranke verwendet werden.

In der Theorie wäre es gar nicht nötig, dass Typparameter mit Schranken nur durch Untertypen der Schranken ersetzbar sind. Untertypbeziehungen höherer Ordnung (siehe Abschnitt 1.4.3) würden reichen; diese Beziehungen weisen starke Ähnlichkeit mit Untertypbeziehungen auf, garantieren jedoch keine Ersetzbarkeit, unterstützen dafür aber binäre Methoden. Solche Beziehungen würden die Flexibilität von Generizität verbessern. Aus praktischen Gründen verzichten

jedoch viele objektorientierte Sprachen darauf, weil eine zweite Typstruktur neben Subtyping die meisten Menschen verwirren würde. Nicht-objektorientierte Sprachen wie Haskell haben jedoch diese zusätzliche Flexibilität (etwa durch „Typklassen“, zwischen denen Untertypbeziehungen höherer Ordnung bestehen, jedoch keine Untertypbeziehungen).

Durch die heterogene Übersetzung von Templates müssen wir in C++ keine Schranken angeben, um Eigenschaften der Typen, die Typparameter ersetzen, verwenden zu können. Es wird einfach für jede übersetzte Klasse getrennt überprüft, ob die Typen alle vorausgesetzten Eigenschaften erfüllen. In dieser Hinsicht ist Generizität mit heterogener Übersetzung sehr flexibel (ähnlich den Typklassen in Haskell, aber ohne explizite Spezifikationen). Unterschiedliche Typen, die einen Typparameter ersetzen, müssen keinen gemeinsamen Obertyp haben. Schranken auf Typparametern wären jedoch hilfreich, indem sie Client-Code vom Server-Code entkoppeln, sodass Clients keine versteckten Details des Servers kennen müssen. In C++ gibt es derzeit noch nichts Entsprechendes. Dadurch haben Programmänderungen manchmal unvorhersehbare Auswirkungen und die Qualität von Fehlermeldungen ist oft schlecht, da sie sich auf Server-Code beziehen, den ein Client gar nicht sehen soll.

Für Interessierte, nicht Prüfungsstoff. Das folgende Beispiel zeigt einen Ausschnitt aus einer generischen Klasse in C++:

```
template<typename T> class Pair {
    public:  Pair(T x, T y) { first = x; second = y; }
             T sum() { return first + second; }
    private: T first, second;
    ...
};
```

Die Klasse `Pair` verwendet `T` als Typparameter. Für `T` kann jeder Typ eingesetzt werden, auf dessen Instanzen der Operator `+` definiert ist, da `+` in der Methode `sum` verwendet wird. Im Kopf der Klasse ist diese Einschränkung nicht angeführt. Hier sind einige Beispiele für die Verwendung von `Pair`:

```
    Pair<int>    anIntPair(2, 3);
    Pair<Person> aPersonPair(Person("Susi"),
                             Person("Strolchi"));
    Pair<char*>  aStringPair("Susi", "Strolchi");
```

Die erste Zeile ersetzt `T` durch `int` und erzeugt eine Variable `anIntPair`, die mit einem neuen Objekt von `Pair` initialisiert ist. Der Konstruktor wird mit den Argumenten 2 und 3 aufgerufen. Ein Aufruf von `anIntPair.sum()` liefert als Ergebnis 5, da für `+` die ganzzahlige Addition verwendet wird. Ob die weiteren Zeilen korrekt sind, hängt davon ab, ob für die Typen `Person` und `char*` (Zeiger auf Zeichen, als String verwendbar) der Operator `+` implementiert wurde. (In C++ können Operatoren überladen und selbst implementiert werden.) Falls dem so ist, werden im Rumpf von `sum` die entsprechenden Implementierungen von `+` aufgerufen. Sonst liefert der Compiler eine Fehlermeldung.

Das nächste Beispiel zeigt eine generische Funktion in C++:

```
template<typename T> T max(T a, T b) {
    if (a > b)
        return a;
    return b;
}
    ...
    int       i, j;
    char      *x, *y;
    Pair<int> p, q;
    ...
    i = max(i, j);  // maximum of integers
    x = max(x, y);  // maximum of pointers to characters
    p = max(p, q);  // maximum of integer pairs
```

Wie in Java erkennt der Compiler anhand der Typen der Argumente, welcher Typ für den Typparameter zu verwenden ist. Im Beispiel wird vorausgesetzt, dass der Operator `>` auf allen Typen, die für `T` verwendet werden, definiert ist, ohne diese Eigenschaft explizit zu machen. **Ende des Einschubs für Interessierte**

Eine weitere Variante für den Umgang mit gebundenen Typparametern bietet Ada: Als Schranke werden Funktionen (oder Prozeduren) angegeben, welche die Typen, die Typparameter ersetzen, bereitstellen müssen. Auf den ersten Blick ähnelt diese Variante der von Templates in C++. Jedoch kann bei jeder Verwendung einer generischen Einheit getrennt angegeben werden, welche konkrete Funktion einen generischen Parameter ersetzt. Funktionen werden als generische Parameter behandelt. Ada hat den Nachteil, dass generische Funktionen nicht einfach aufgerufen werden können, sondern zuvor daraus nicht-generische Funktionen abgeleitet werden müssen. Gründe kommen aus der Philosophie hinter Ada: Alles steht so weit wie möglich explizit im Programmcode.

Für Interessierte, nicht Prüfungsstoff. Eine generische Funktion in Ada [18] soll zeigen, welche Flexibilität Einschränkungen auf Typparametern bieten können:

```
generic
    type T is private;
    with function "<" (X, Y: T) return Boolean is (<>);
function Max (X, Y: T) return T is
begin
    if X < Y
    then
        return Y
    else
        return X
    end if
end Max;
...
function IntMax is new Max (Integer);
function IntMin is new Max (Integer, ">");
```

Die Funktion `Max` hat zwei generische Parameter: den Typparameter `T` und den Funktionsparameter `<`, dessen Parametertypen mit dem Typparameter in Beziehung stehen. Aufgrund von „`is (<>)`“ kann der zweite Parameter weggelassen werden. In diesem Fall wird dafür die Funktion namens `<` mit den entsprechenden Parametertypen gewählt, wie in C++. Die Funktion `IntMax` entspricht `Max`, wobei an Stelle von `T` der Typ `Integer` verwendet wird. Als Vergleichsoperator wird der Kleiner-Vergleich auf ganzen Zahlen verwendet. In der Funktion `IntMin` ist `T` ebenfalls durch `Integer` ersetzt, zum Vergleich wird aber der Größer-Vergleich verwendet, sodass von `IntMin` das kleinere Argument zurückgegeben wird. Anders als in C++ und Java müssen die für Typparameter zu verwendenden Typen explizit angegeben werden.

Ende des Einschubs für Interessierte

In Java ist Ähnliches wie in Ada durch Komparatoren (siehe Abschnitt 4.1.2) auch erzielbar, jedoch nur mit hohem Aufwand.

4.3 Typabfragen und Typumwandlungen

Wir wollen nun den Umgang mit dynamischer Typinformation untersuchen. In Abschnitt 4.3.1 finden sich allgemeine Hinweise dazu. In den nächsten beiden Abschnitten werden spezifische Probleme durch Verwendung dynamischer Typinformation gelöst: Abschnitt 4.3.2 behandelt simulierte Generizität und Abschnitt 4.3.3 kovariante Probleme.

4.3.1 Verwendung dynamischer Typinformation

Jedes Objekt hat eine Methode `getClass`, welche die interne Repräsentation der Klasse des Objekts (vom Typ `Class`) als Ergebnis zurückgibt. Diese Methode bietet die direkteste Möglichkeit des Zugriffs auf den dynamischen Typ. Für jedes Interface, jede Klasse, jeden elementaren Typ sowie davon abgeleiteten Array-Typ gibt es ein eigenes Objekt vom Typ `Class`. Solche Objekte lassen sich einfach mit `==` vergleichen, `equals` ist dafür nicht nötig. Objekte von `Class` sind die besten Ausgangspunkte, wenn es darum geht, zur Laufzeit festzustellen, wie der entsprechende Typ aufgebaut ist. Objekte von `Class` lassen sich auch direkt durch Anhängen von `.class` an einen Typ ansprechen; z. B. `int.class`, `int[].class`, `Person.class` und `Comparable.class`.

Häufig möchten wir nur wissen, ob der dynamische Typ eines Referenzobjekts Untertyp eines gegebenen Typs ist. Dafür bietet Java den `instanceof`-Operator an, wie folgendes Beispiel zeigt:

```
int calculateTicketPrice(Person p) {
    if (p.age() < 15 || p instanceof Student)
        return standardPrice / 2;
    return standardPrice;
}
```

Eine Anwendung des `instanceof`-Operators liefert `true`, wenn das Objekt links vom Operator eine Instanz des Typs rechts vom Operator ist. Im Beispiel liefert die Typabfrage `true` wenn `p` vom dynamischen Typ `Student`, `Tutor` oder `StudTrainee` ist (entsprechend der Typhierarchie aus Abschnitt 3.1.2). Die Abfrage liefert `false` wenn `p` gleich `null` oder von einem anderen dynamischen Typ ist. Solche dynamischen Typabfragen können, wie alle Vergleichsoperationen, überall stehen, wo Boolesche Ausdrücke erlaubt sind.

Mittels Typabfragen lässt sich zwar der Typ eines Objekts zur Laufzeit bestimmen, aber Typabfragen reichen nicht aus, um auf Methoden und Variablen des dynamisch ermittelten Typs zugreifen zu können. Wir wollen auch Nachrichten an das Objekt senden können, die nur Instanzen des dynamisch ermittelten Typs verstehen, oder das Objekt als aktuellen Parameter verwenden, wobei der Typ des formalen Parameters dem dynamisch ermittelten Typ entspricht. Für diese Zwecke gibt es in Java explizite Typumwandlungen als Casts auf Referenzobjekten. Folgendes (auf den ersten Blick überzeugende, tatsächlich aber fehlerhafte) Beispiel[2] modifiziert ein Beispiel aus Abschnitt 3.1.1:

```
class Point3D extends Point2D {
    private int z;
    public boolean equal(Point2D p) {
        // true if points are equal in all coordinates
        if (p instanceof Point3D)
            return super.equal(p) && ((Point3D)p).z == z;
        return false;
    }
}
```

In `Point3D` liefert `equal` als Ergebnis `false` wenn der dynamische Typ des Arguments kein Untertyp von `Point3D` ist. Sonst wird die Methode aus `Point2D` aufgerufen und die zusätzlichen Objektvariablen `z` werden verglichen. Vor dem Zugriff auf `z` von `p` ist eine explizite Typumwandlung nötig, die den deklarierten Typ von `p` von `Point2D` (wie im Kopf der Methode angegeben) nach `Point3D` umwandelt, da `z` in Objekten von `Point2D` nicht zugreifbar ist. Syntaktisch wird die Typumwandlung als `(Point3D)p` geschrieben. Um den Ausdruck sind Klammern nötig, damit der Typ von `p` umgewandelt wird, nicht der Typ des Ergebnisses von `p.z == z` wie in `(Point3D)p.z == z`.

Verbesserungen von `Point2D` und `Point3D` folgen in Abschnitt 4.3.3. Dieses Beispiel ist typisch für in der Praxis häufig vorkommende Lösungen ähnlicher Probleme: Obwohl der Typvergleich und die Typumwandlung hier korrekt eingesetzt sind, ergibt sich aus der Problemstellung selbst zusammen mit dem Lösungsansatz ein unerwünschtes Programmverhalten. Alleine schon die Notwendigkeit für Typabfragen und Typumwandlungen deutet darauf hin, dass

[2] Versuchen Sie, Fehler in dieser „Lösung“ selbst zu finden. Die Verwendung von dynamischer Typinformation sowie die Typumwandlung ist hier korrekt. Aber vielleicht wird eine implizite Zusicherung verletzt, wie die, dass `a.equal(b)` dasselbe Ergebnis liefert wie `b.equal(a)`. Am schnellsten wird der Fehler sichtbar, wenn wir Zusicherungen auf `Point3D` und `Point2D` explizit machen.

wir es mit einer Problemstellung zu tun haben, die mit den üblichen Konzepten der objektorientierten Programmierung nicht einfach in den Griff zu bekommen ist. Deswegen müssen wir sehr vorsichtig sein. Wir können uns nicht auf die Intuition verlassen. Das liegt (neben der Gefährlichkeit des falschen Einsatzes von Typabfragen und Typumwandlungen) auch an der komplexen Natur entsprechender Problemstellungen.

Typumwandlungen sind auf Referenzobjekten nur durchführbar, wenn der Ausdruck, dessen deklarierter Typ in einen anderen Typ umgewandelt werden soll, tatsächlich den gewünschten Typ – oder einen Untertyp davon – als dynamischen Typ hat oder gleich `null` ist. Im Allgemeinen ist das erst zur Laufzeit feststellbar. Zur Laufzeit erfolgt eine entsprechende Überprüfung. Sind die Bedingungen nicht erfüllt, wird eine Ausnahme ausgelöst.

Dynamische Typabfragen und Typumwandlungen sind sehr mächtige Werkzeuge. Wir können damit einiges machen, was sonst nicht oder nur sehr umständlich machbar wäre. Allerdings kann die falsche Verwendung Fehler in einem Programm verdecken und die Wartbarkeit erschweren. Fehler werden oft dadurch verdeckt, dass der deklarierte Typ einer Variablen oder eines formalen Parameters nur mehr wenig mit dem Typ zu tun hat, dessen Objekte wir erwarten. Beispielsweise ist der deklarierte Typ `Object`, obwohl wir erwarten, dass nur Objekte von `Integer` oder `Person` vorkommen. Einen konkreteren gemeinsamen Obertyp dieser beiden Typen als `Object` gibt es im System ja nicht. Um auf Eigenschaften von `Integer` oder `Person` zuzugreifen, werden dynamische Typabfragen und Typumwandlungen eingesetzt. Wenn in Wirklichkeit statt einem Objekt von `Integer` oder `Person` ein Objekt von `Point2D` verwendet wird, liefert der Compiler keine Fehlermeldung. Erst zur Laufzeit kann es im günstigsten Fall zu einer Ausnahmebehandlung kommen. Es ist aber auch möglich, dass einfach nur die Ergebnisse falsch sind, oder – noch schlimmer – falsche Daten in einer Datenbank gespeichert werden. Der Grund für das mangelhafte Erkennen dieses Typfehlers liegt darin, dass mit Hilfe von dynamischen Typabfragen und Typumwandlungen statische Typüberprüfungen durch den Compiler ausgeschaltet wurden, obwohl wir uns vermutlich nach wie vor auf statische Typsicherheit verlassen.

Die schlechte Wartbarkeit ist ein weiterer Grund, um Typabfragen (auch ohne Typumwandlungen) nur sehr sparsam zu nutzen:

```
if (x instanceof T1)
    doSomethingOfTypeT1((T1)x);
else if (x instanceof T2)
    doSomethingOfTypeT2((T2)x);
...
else
    doSomethingOfAnyType(x);
```

Wie schon bekannt, lassen sich `switch`- und `if`-Anweisungen durch dynamisches Binden ersetzen. Das sollen wir tun, da dynamisches Binden wartungsfreundlicher ist. Dasselbe gilt für dynamische Typabfragen, welche die möglichen Typen im Programmcode fix verdrahten und daher bei Änderungen der

Typhierarchie ebenfalls geändert werden müssen. Oft sind dynamische Typabfragen wie im Beispiel durch `x.doSomething()` ersetzbar. Die Auswahl des auszuführenden Programmcodes erfolgt durch dynamisches Binden. Die Klasse des deklarierten Typs von `x` implementiert `doSomething` entsprechend der Methode `doSomethingOfAnyType`, die Unterklassen `T1`, `T2` und so weiter entsprechend `doSomethingOfTypeT1`, `doSomethingOfTypeT2` und so weiter.

Manchmal ist es nicht einfach, dynamische Typabfragen durch dynamisches Binden zu ersetzen. Dies trifft vor allem in diesen Fällen zu:

- Der deklarierte Typ von `x` ist zu allgemein; die einzelnen Alternativen decken nicht alle Möglichkeiten ab. Das ist genau die oben erwähnte gefährliche Situation, in der die statische Typsicherheit von Java umgangen wird. In dieser Situation ist eine Refaktorisierung des Programms angebracht, die Typabfragen vermeidet.

- Die Klassen, die dem deklarierten Typ von `x` und dessen Untertypen entsprechen, können nicht erweitert werden, beispielsweise weil sie als `final` definiert wurden und wir keinen Zugriff auf den Source-Code haben. Als Lösung können wir parallel zur unveränderbaren Klassenhierarchie eine gleich strukturierte Hierarchie aufbauen, deren Klassen (Wrapper-Klassen) die zusätzlichen Methoden beschreiben. Typabfragen ermöglichen in einfachen Fällen eine einfache Lösung. In komplexeren Fällen wird ein Wrapper die bessere Lösung sein.

- Manchmal ist die Verwendung dynamischen Bindens schwierig, weil die einzelnen Alternativen auf private Variablen und Methoden zugreifen. Methoden anderer Klassen haben diese Information nicht. Oft lässt sich die fehlende Information durch Übergabe geeigneter Argumente beim Aufruf der Methode oder durch „Call-Backs“ (also Rückfragen an das aufrufende Objekt – sinnvoll wenn die Information nur selten benötigt wird) verfügbar machen. Diese Techniken werden in Java vor allem durch (anonyme) innere Klassen und (seit Java 8) Lambda-Ausdrücke unterstützt.

- Der deklarierte Typ von `x` kann sehr viele Untertypen haben. Wenn `doSomething` nicht in einer gemeinsamen Oberklasse in der Bedeutung von `doSomethingOfAnyType` implementierbar ist, muss `doSomething` in vielen Klassen auf gleiche Weise implementiert werden. Das bedeutet einen Mehraufwand für die Wartung. Typabfragen scheinen eine einfachere Lösung zu bieten. Ein Grund, warum die Methode nicht in der Oberklasse implementierbar sein könnte, liegt in der fehlenden Mehrfachvererbung. Default-Methoden in Interfaces reduzieren dieses Problem seit Java 8: Bis auf direkte Variablenzugriffe gibt es Mehrfachvererbung.

  Faustregel: Typabfragen und Typumwandlungen sollen nach Möglichkeit vermieden werden.

In wenigen Fällen ist es nötig und durchaus angebracht, diese mächtigen, aber unsicheren Werkzeuge zu verwenden, wie wir noch sehen werden.

Typumwandlungen werden auch auf elementaren Typen wie `int`, `char` und `float` unterstützt. In diesen Fällen haben Typumwandlungen aber eine andere Bedeutung, da für Variablen elementarer Typen die dynamischen Typen immer gleich den statischen und deklarierten Typen sind. Bei elementaren Typen werden tatsächlich die Werte umgewandelt, nicht deklarierte Typen. Aus einer Gleitkommazahl wird beispielsweise durch Runden gegen Null eine ganze Zahl. Dabei kann Information verloren gehen, aber es kommt zu keiner Ausnahmebehandlung. Daher haben Typumwandlungen auf elementaren Typen eine ganz andere Qualität als auf Referenztypen und machen im Normalfall keinerlei Probleme. Typumwandlungen zwischen elementaren Typen und Referenztypen werden in Java nicht unterstützt.

4.3.2 Typumwandlungen und Generizität

Die homogene Übersetzung einer generischen Klasse oder Methode in eine Klasse oder Methode ohne Generizität ist im Prinzip sehr einfach, wie wir bereits in Abschnitt 4.2.2 gesehen haben:

- Spitze Klammern werden samt ihren Inhalten weggelassen.
- Jedes verbliebene Vorkommen eines Typparameters wird durch `Object` oder, falls vorhanden, die erste Schranke ersetzt.
- Ergebnisse und Argumente werden in die nötigen deklarierten Typen umgewandelt, wenn die entsprechenden Ergebnistypen und formalen Parametertypen Typparameter sind, die durch Typen ersetzt wurden.

Aus der Listen-Klasse und den Interfaces aus Abschnitt 4.1.2 entsteht das:

```
public class List implements Collection {
    private Node head = null, tail = null;
    public void add(Object elem) {
        if (head == null) tail = head = new Node((Object)elem);
        else tail.setNext(tail = new Node((Object)elem));
    }
    private class ListIter implements Iterator {
        private Node p = head;
        public boolean hasNext() { return p != null; }
        public Object next() {
            if (p == null)
                throw new java.util.NoSuchElementException();
            Object elem = (Object)p.elem();
            p = p.next();
            return elem;
        }
    }
    public Iterator iterator() { return new ListIter(); }
}
```

```
class Node {
    private Object elem;
    private Node next = null;
    public Node(Object elem) { this.elem = elem; }
    public Object elem() { return elem; }
    public Node next() { return next; }
    public void setNext(Node next) { this.next = next; }
}

public interface Collection {
    void add(Object elem);
    Iterator iterator();
}
public interface Iterator {
    Object next();
    boolean hasNext();
}
```

Mangels expliziter Schranke wurden die Typparameter durch `Object` ersetzt. In den Konstruktoraufrufen von `Node` und auf dem Ergebniswert von `p.elem()` wurden nach dem gegebenen Schema automatisch Casts auf `Object` hinzugefügt – `Object` weil die Typparameter durch `Object` ersetzt sind. Diese Casts sind hier sinnlos und werden vom Compiler gleich wieder wegoptimiert. Sinnvollere Typumwandlungen ergeben sich durch die Übersetzung der Klasse `ListTest`:

```
class ListTest {
    public static void main(String[] args) {
        List xs = new List();
        xs.add((Integer)new Integer(0));
        Integer x = (Integer)xs.iterator().next();

        List ys = new List();
        ys.add((String)"zerro");
        String y = (String)ys.iterator().next();

        List zs = new List();
        zs.add((List)xs);
        List z = (List)zs.iterator().next();
    }
}
```

Die Typumwandlungen in den Aufrufen von `add` haben nur einen Zweck: Falls `add` überladen wäre, würden die Typumwandlungen für die Auswahl der richtigen Methoden sorgen, indem sie die deklarierten Typen entsprechend anpassen. Ein Compiler wird solche Typumwandlungen wohl wegoptimieren.

Die Typumwandlungen der Ergebnisse der Methodenaufrufe sind dagegen von großer Bedeutung. Wenn der Ergebnistyp der Methode vor Übersetzung

der Generizität ein Typparameter war, wird das Ergebnis unmittelbar nach dem Aufruf in den Typ umgewandelt, der den Typparameter ersetzt. So bekommt das Ergebnis des ersten Aufrufs von `next` den erwarteten deklarierten Typ `Integer`, obwohl `next` nur ein Ergebnis vom deklarierten Typ `Object` zurückgibt. Das Ergebnis des zweiten Aufrufs bekommt dagegen wie erwartet den deklarierten Typ `String`, obwohl genau dieselbe Methode `next` aufgerufen wird. Durch die Typumwandlungen auf den Ergebnissen kann ein und dieselbe Methode also für unterschiedliche Typen eingesetzt werden.

Im letzten Teil des Beispiels wird nur der Typ `List` verwendet, obwohl eigentlich der Typ `List<Integer>` korrekt wäre. Die spitzen Klammern samt Inhalt werden jedoch weggelassen. Weiter unten werden wir sehen, dass `List` genau der Typ ist, der durch Übersetzung der Generizität aus `List<A>` erzeugt wurde. Dessen Verwendung macht durchaus Sinn.

Abgesehen von einigen unbedeutenden Details, auf die wir hier nicht näher eingehen, ist die Übersetzung so einfach, dass wir sie auch selbst ohne Unterstützung durch den Compiler durchführen können. Wir können gleich direkt Programmcode ohne Generizität schreiben. Allerdings hat das auch einen schwerwiegenden Nachteil: Statt Fehlermeldungen, die bei Verwendung von Generizität der Compiler generiert, werden ohne Generizität erst zur Laufzeit Ausnahmen ausgelöst. Zum Beispiel liefert der Java-Compiler für

```
List<Integer> xs = new List<Integer>();
xs.add(new Integer(0));
String y = xs.iterator().next();
// Syntaxfehler: String erwartet, Integer gefunden
```

eine Fehlermeldung, nicht aber für den daraus generierten Code:

```
List xs = new List();
xs.add(new Integer(0));
String y = (String)xs.iterator().next();
// Exception bei Typumwandlung von Object auf String
```

Die Vorteile von Generizität liegen also in erster Linie in der besseren Lesbarkeit und höheren Typsicherheit.

Viele ältere, nicht-generische Java-Bibliotheken verwenden Klassen, die so aussehen, als ob sie aus generischen Klassen erzeugt worden wären. Das heißt, Objekte, die in Listen etc. eingefügt werden, müssen in der Regel nur Untertypen von `Object` sein. Vor der Verwendung von aus solchen Datenstrukturen gelesenen Objekten steht meist eine Typumwandlung. Die durchgehende Verwendung von Generizität würde den Bedarf an Typumwandlungen vermeiden oder zumindest erheblich reduzieren.

> *Faustregel:* Wir sollen nur sichere Formen der Typumwandlung (die keine Ausnahmen auslösen) einsetzen.

Sichere Typumwandlungen. Dieser Argumentation folgend ist es leicht, sich auch bei der Programmierung in einer Sprache ohne Generizität anzugewöhnen, nur „sichere“ Typumwandlungen einzusetzen: Typumwandlungen sind sicher (lösen keine Ausnahmebehandlung aus) wenn

- in einen Obertyp des deklarierten Objekttyps umgewandelt wird,
- oder davor eine dynamische Typabfrage erfolgt, die sicherstellt, dass das Objekt einen entsprechenden dynamischen Typ hat,
- oder das Programmstück so geschrieben ist, als ob Generizität verwendet würde und alle Konsistenzprüfungen, die normalerweise der Compiler macht, vor der homogenen Übersetzung der Generizität händisch von uns durchgeführt werden.

Im ersten Fall handelt es sich um eine völlig harmlose Typumwandlung nach oben in der Typhierarchie (genannt *Up-Cast*), die aber kaum gebraucht wird. Nur auf Argumenten ist sie manchmal sinnvoll, um zwischen überladenen Methoden zu wählen – wie die Typumwandlungen, die bei der Übersetzung der Generizität auf Argumenten eingeführt wurden.

Die beiden anderen Fälle sind wichtiger, beziehen sich aber auf weniger harmlose Typumwandlungen nach unten – sogenannte *Down-Casts.*

Der zweite Punkt in obiger Aufzählung impliziert, dass es einen sinnvollen Programmzweig geben muss, der im Falle des Scheiterns des Typvergleichs ausgeführt wird. Würden wir im alternativen Zweig nur eine Ausnahme werfen, könnten wir nicht von einer sicheren Typumwandlung sprechen. Leider erweisen sich gerade falsche Typannahmen in alternativen Zweigen als häufige Fehlerquelle. Fehler zeigen sich oft erst später. Angenommen, ein abstrakter Typ `A` hat die beiden Untertypen `B` und `C` und `x` ist vom deklarierten Typ `A`. Es ist sehr verlockend, im `else`-Zweig von `if(x instanceof B){...}else{...}` anzunehmen, dass `x` vom Typ `C` wäre und einen vermeintlich sicheren Cast `(C)x` zu verwenden. Das Testen des Programmstücks wird keinen Fehler erkennen lassen. Wenn jedoch nachträglich ein neuer Untertyp `D` von `A` hinzugefügt wird, ist die Annahme verletzt und der Fehler zeigt sich.

> *Faustregel:* Bei Zutreffen des zweiten Punkts ist besonders darauf zu achten, dass alle Annahmen im alternativen Zweig (bei Scheitern des Typvergleichs) durch Zusicherungen abgesichert sind.

Außerdem gibt es oft mehrere alternative Zweige, die sich in geschachtelten `if`-Anweisungen zeigen. Aufgrund der damit verbundenen Wartungsprobleme sollten wir auf solche Lösungen verzichten.

Bei Zutreffen des dritten Punkts treten keine solchen Probleme auf. Stattdessen sind aufwendige händische Programmanalysen notwendig. Es muss vor allem sichergestellt werden, dass

- wirklich alle Ersetzungen eines (gedachten) Typparameters durch einen Typ gleichförmig erfolgen – das heißt, jedes Vorkommen des Typparameters tatsächlich durch denselben Typ ersetzt wird

- und keine impliziten Untertypbeziehungen vorkommen.

Vor allem hinsichtlich impliziter Untertypbeziehungen ist die Intuition leicht irreführend, da beispielsweise sowohl `List<Integer>` als auch `List<String>` in der homogenen Übersetzung durch `List` dargestellt werden, obwohl sie nicht gegeneinander ersetzbar sind.

> *Faustregel:* Wenn die Programmiersprache Generizität unterstützt, soll die dritte Möglichkeit nicht verwendet werden.

Generizität ist einer dynamischen Typumwandlung immer vorzuziehen. Wenn Generizität nicht unterstützt wird, ist der dritte Punkt dem zweiten vorzuziehen. Unnötige dynamische Typvergleiche (z. B. zur Absicherung einer Typumwandlung, obwohl die Voraussetzungen dafür gemäß dem dritten Punkt händisch überprüft wurden) sollen vermieden werden, da sie die Sicherheit nicht wirklich erhöhen, aber die Wartung erschweren können.

Umgang mit Einschränkungen der Generizität. Generizität ist, mit einigen Einschränkungen, auch in dynamischen Typabfragen und Typumwandlungen einsetzbar:

```
<A> Collection<A> up(List<A> xs) {
    return (Collection<A>)xs;
}
<A> List<A> down(Collection<A> xs) {
    if (xs instanceof List<A>)
        return (List<A>)xs;
    else { ... }          // Was machen wir hier?
}
List<String> bad(Object o) {
    if (o instanceof List<String>)      // error
        return (List<String>)o;         // error
    else { ... }          // Was machen wir hier?
}
```

In der Methode `bad` werden vom Compiler Fehlermeldungen ausgegeben, da es zur Laufzeit keine Information über den gemeinsamen Typ der Listenelemente gibt. Es ist daher unmöglich, in einer Typabfrage oder Typumwandlung dynamisch zu prüfen, ob `o` den gewünschten Typ hat. Die Methoden `up` und `down` haben dieses Problem nicht, weil der bekannte Unter- beziehungsweise Obertyp den Typ aller Listenelemente bereits statisch festlegt, falls es sich tatsächlich um eine Liste handelt. Der Compiler ist intelligent genug, solche Situationen zu erkennen. Bei der Übersetzung werden einfach alle spitzen Klammern (und deren Inhalte) weggelassen. Im übersetzten Programm sind die strukturellen Unterschiede zwischen `down` und `bad` nicht mehr erkennbar, aber `bad` kann zu einer Ausnahme führen. Bei der händischen Programmüberprüfung ist auf solche Feinheiten besonders zu achten.

Java erlaubt die gemischte Verwendung von generischen Klassen und Klassen, die durch homogene Übersetzung daraus erzeugt wurden:

```
public class List<A> implements Collection<A> {
    ...
    public boolean equals(Object o) {
        if (o == null || o.getClass() != List.class)
            return false;
        Iterator<A> xi = this.iterator();
        Iterator yi = ((List)o).iterator();
        while (xi.hasNext() && yi.hasNext()) {
            A x = xi.next();
            Object y = yi.next();
            if (!(x == null ? y == null : x.equals(y)))
                return false;
        }
        return !(xi.hasNext() || yi.hasNext());
    }
}
```

Im Beispiel sind `List<A>` und `Iterator<A>` generisch, `List` die entsprechende übersetzte Klasse und `Iterator` das übersetzte Interface. Übersetzte Klassen und Interfaces werden *Raw-Types* genannt. Wir sprechen auch von *Type-Erasures* um darauf hinzuweisen, dass durch die Übersetzung Typinformation weggelassen wird, die zur Laufzeit nicht mehr zur Verfügung steht. Sind Ausdrücke in spitzen Klammern angegeben, erfolgt die statische Typüberprüfung für Generizität. Sonst prüft der Compiler Typparameter nicht.

Ein häufiges Anfänger- und Unachtsamkeitsproblem besteht darin, dass wir bei Typdeklarationen die spitzen Klammern anzugeben vergessen. Da solche Typen vom Compiler als Raw-Types verstanden werden, erfolgt keine Typüberprüfung der Generizität. Folglich halten wir das falsche Programm für korrekt. Daher müssen wir stets auf das Vorhandensein der spitzen Klammern achten.

Der Java-Compiler kann in einem lokalen Kontext die Typen in spitzen Klammern, die Typparameter ersetzen, selbst ableiten (inferieren). Wir müssen diese Typen daher gar nicht selbst hinschreiben. Allerdings müssen wir in allen Fällen leere spitze Klammern hinschreiben (z. B. `List<>` in einer Deklaration `List<String> lst = new List<>()`), damit der Compiler Typen mit derart abgeleiteten Typen von Raw-Types unterscheiden kann. Es passiert nichts Schlimmes, wenn wir versuchen, überall Typen in spitzen Klammern wegzulassen, aber die spitzen Klammern hinschreiben. Wenn der Compiler nicht genug Information für die Typprüfung hat, wird uns eine Fehlermeldung darauf aufmerksam machen, aber den Programmablauf beeinflusst das nicht.

4.3.3 Kovariante Probleme

In Abschnitt 3.1 haben wir gesehen, dass Typen von Eingangsparametern nur kontravariant sein können. Kovariante Eingangsparametertypen verletzen das Ersetzbarkeitsprinzip. In der Praxis wünschen wir uns manchmal gerade kovariante Eingangsparametertypen. Entsprechende Aufgabenstellungen heißen

kovariante Probleme. Zur Lösung kovarianter Probleme bieten sich dynamische Typabfragen und Typumwandlungen an, wie folgendes Beispiel zeigt:

```
abstract class Food { ... }
class Grass extends Food { ... }
class Meat extends Food { ... }
abstract class Animal {
    public abstract void eat(Food x);
}
class Cow extends Animal {
    public void eat(Food x) {
        if (x instanceof Grass) { ... }
        else fallIll();
    }
}
class Tiger extends Animal {
    public void eat(Food x) {
        if (x instanceof Meat) { ... }
        else showTeeth();
    }
}
```

Es ist ganz natürlich, `Grass` und `Meat` als Untertypen von `Food` anzusehen. `Grass` und `Meat` sind offensichtlich einander ausschließende Spezialisierungen von `Food`. Ebenso sind `Cow` und `Tiger` Spezialisierungen von `Animal`. Es entspricht dem Alltagswissen, dass Tiere im Allgemeinen Futter fressen, Rinder aber nur Gras und Tiger nur Fleisch. Als Parametertyp der Methode `eat` wünschen wir uns daher in `Animal Food`, in `Cow Grass` und in `Tiger Meat`.

Genau diese Beziehungen aus der realen Welt sind aber nicht typsicher darstellbar. Zur Lösung des Problems bietet sich eine erweiterte Sicht der Beziehungen aus der realen Welt an: Wir können auch einem Rind Fleisch und einem Tiger Gras zum Fressen anbieten. In diesem Fall müssen wir aber mit unerwünschten Reaktionen der Tiere rechnen. Obiges Programmstück beschreibt entsprechendes Verhalten: Wenn dem Tier geeignetes Futter angeboten wird, erledigen die Methoden `eat` die Aufgaben problemlos. Sonst führen sie Aktionen aus, die vermutlich nicht erwünscht sind.

In obigem Programmtext sind immer dynamische Typabfragen nötig, auch wenn Aufrufer die richtige Futterart kennen. Durch Überladen von Methoden können wir in diesem Fall gleich den richtigen Code ausführbar machen:

```
class Cow extends Animal {
    public void eat(Grass x) { ... }
    public void eat(Food x) {
        if (x instanceof Grass) eat((Grass)x);
        else fallIll();
    }
}
```

```
class Tiger extends Animal {
    public void eat(Meat x) { ... }
    public void eat(Food x) {
        if (x instanceof Meat) eat((Meat)x);
        else showTeeth();
    }
}
```

In `Cow` und `Tiger` ist `eat` überladen, es gibt also mehrere Methoden desselben Namens. Die Methoden mit `Food` als Parametertyp überschreiben jene aus `Animal`, während die anderen nicht in `Animal` vorkommen. Es wird die Methode ausgeführt, deren formaler Parametertyp der spezifischste Obertyp des *deklarierten* Argumenttyps ist. Nur wenn in einem Aufruf `a.eat(f)` sowohl die Tierart durch den deklarierten Typ von `a` und die passende Futterart durch den deklarierten Typ von `f` dem Compiler bekannt sind, wird gleich die richtige Methode ausgeführt. In allen anderen Fällen wird die Methode mit `Food` als formaler Parametertyp ausgeführt, sodass eine dynamische Typabfrage nötig ist. Falls die Futterart (über den dynamischen Typ ermittelt) passt, wird zur Laufzeit auf die richtige Methode verzweigt, wobei der deklarierte Typ des Arguments durch einen Cast bestimmt wird.

Durch Umschreiben des Programms können wir zwar Typabfragen und Typumwandlungen vermeiden (eine Technik dafür sehen wir bald), aber die unerwünschten Aktionen bei kovarianten Problemen bleiben erhalten. Die einzige Möglichkeit besteht darin, kovariante Probleme zu vermeiden. Beispielsweise reicht es, `eat` aus `Animal` zu entfernen. Dann können wir zwar `eat` nur mehr mit Futter der richtigen Art in `Cow` und `Tiger` aufrufen, aber wir können Tiere nur mehr füttern, wenn wir die Art der Tiere und des Futters genau kennen.

Faustregel: Kovariante Probleme sind möglichst zu vermeiden.

Für Interessierte, nicht Prüfungsstoff. Einige Programmiersprachen bieten teilweise Lösungen für kovariante Probleme an. Betrachten wir Eiffel: In dieser Sprache sind kovariante Eingangsparametertypen erlaubt. Wenn die Klasse `Animal` die Methode `eat` mit dem Parametertyp `Food` enthält, können die überschriebenen Methoden in den Klassen `Cow` und `Tiger` die Parametertypen `Grass` und `Meat` haben. Dies ermöglicht eine natürliche Modellierung kovarianter Probleme. Weil dadurch das Ersetzbarkeitsprinzip verletzt ist, können an Stelle dieses Parameters keine Argumente von einem Untertyp des Parametertyps verwendet werden. Der Compiler kann jedoch die Art des Tieres oder die Art des Futters nicht immer statisch feststellen. Wird `eat` mit einer falschen Futterart aufgerufen, kommt es zu einer Ausnahme zur Laufzeit. Tatsächlich ergibt sich dadurch derselbe Effekt, als ob wir in Java ohne vorhergehende Überprüfung den Typ des Arguments von `eat` auf die gewünschte Futterart umwandeln würden. Eine echte Lösung des Problems ist das daher nicht.

Einen anderen Ansatz bieten *virtuelle Typen,* die derzeit in keiner gängigen Programmiersprache verwendet werden [20, 17]. Virtuelle Typen ähneln geschachtelten Klassen wie in Java, die jedoch, anders als in Java, in Unterklassen überschreibbar sind. Die beiden Klassenhierarchien mit `Animal` und `Food` als Wurzeln werden eng

verknüpft: `Food` ist in `Animal` enthalten. In `Cow` ist `Food` mit einer neuen Klasse überschrieben, welche die Funktionalität von `Grass` aufweist, `Food` in `Tiger` mit einer Klasse der Funktionalität von `Meat`. Statt `Grass` und `Food` schreiben wir dann `Cow.Food` und `Tiger.Food`. Der Typ `Food` des Parameters von `eat` bezieht sich immer auf den lokal gültigen Namen, in `Cow` also auf `Cow.Food`. Noch immer müssen wir `eat` in `Cow` mit einem Argument vom Typ `Cow.Food` und in `Tiger` mit einem Argument vom Typ `Tiger.Food` aufrufen; die Art des Tieres muss also mit der Art des Futters übereinstimmen und der Compiler muss die Übereinstimmung überprüfen können. Aber wenn `Animal` (und daher auch `Cow` und `Tiger`) eine Methode hat, die ein Objekt vom Typ `Food` als Ergebnis liefert, kann das Ergebnis eines solchen Methodenaufrufs als Argument eines Aufrufs von `eat` in demselben Objekt verwendet werden. Dabei muss die Art des Tieres nicht bekannt sein und es ist trotzdem kein Typfehler möglich. Eine ähnliche Methode können wir durch kovariante Ergebnistypen auch in Java schreiben; sie liefert in `Cow` ein Ergebnis vom Typ `Grass` und in `Tiger` eines vom Typ `Meat`. Aber in Java können wir das Ergebnis eines solchen Methodenaufrufs in einem unbekannten Tier nicht typsicher und ohne dynamische Typprüfung an dieses Tier verfüttern. Mit virtuellen Typen wäre das möglich. Gerade in dieser Möglichkeit liegt der (kleine, aber doch vorhandene) inhaltliche Vorteil virtueller Typen. Ein weiterer Vorteil könnte auf der psychologischen Ebene liegen, da der Umgang damit sehr natürlich wirkt, sodass tief gehende Schwierigkeiten im Umgang mit kovarianten Problemen nicht in den Vordergrund treten.

Ende des Einschubs für Interessierte

Binäre Methoden. Einen häufig vorkommenden Spezialfall kovarianter Probleme stellen binäre Methoden dar. Wie in Abschnitt 3.1 eingeführt, hat eine binäre Methode mindestens einen formalen Parameter, dessen Typ gleich der Klasse ist, welche die Methode enthält. Wir könnten binäre Methoden auf die gleiche Weise behandeln wie alle anderen kovarianten Probleme, also dynamische Typabfragen verwenden. Es gibt bessere Lösungen. Hier ist eine bessere Lösung für die binäre Methode `equal` in `Point2D` und `Point3D`:

```
abstract class Point {
    public final boolean equal(Point that) {
        if (that != null && this.getClass() == that.getClass())
            return uncheckedEqual(that);
        return false;
    }
    protected abstract boolean uncheckedEqual(Point p);
}
class Point2D extends Point {
    private int x, y;
    protected boolean uncheckedEqual(Point p) {
        Point2D that = (Point2D)p;
        return this.x == that.x && this.y == that.y;
    }
}
```

```
class Point3D extends Point {
    private int x, y, z;
    protected boolean uncheckedEqual(Point p) {
       Point3D that = (Point3D)p;
       return this.x==that.x && this.y==that.y && this.z==that.z;
    }
}
```

Anders als in vorangegangenen Lösungsansätzen ist `Point3D` kein Untertyp von `Point2D`, sondern sowohl `Point3D` als auch `Point2D` sind von einer gemeinsamen abstrakten Oberklasse `Point` abgeleitet. Dieser Unterschied hat nichts mit binären Methoden zu tun, sondern verdeutlicht, dass `Point3D` in der Regel keine Spezialisierung von `Point2D` ist. Die Rolle, die bisher `Point2D` hatte, spielt jetzt `Point`. Die Methode `equal` ist in `Point` definiert und kann in Unterklassen nicht überschrieben werden. Wenn die beiden zu vergleichenden Punkte genau den gleichen Typ haben, wird in der betreffenden Unterklasse von `Point` die Methode `uncheckedEqual` aufgerufen, die den eigentlichen Vergleich durchführt. Im Unterschied zur in Abschnitt 4.3.1 angerissenen Lösung vergleicht diese Lösung, ob die Typen wirklich gleich sind, nicht nur, ob der dynamische Typ des Arguments ein Untertyp der Klasse ist, in der die Methode ausgeführt wird. Die Lösung in Abschnitt 4.3.1 ist falsch, da ein Aufruf von `equal` in `Point2D` mit einem Argument vom Typ `Point3D` als Ergebnis `true` liefern kann.

Für Interessierte, nicht Prüfungsstoff. Die Programmiersprache Ada unterstützt binäre Methoden direkt: Alle Parameter, die denselben Typ wie das Äquivalent zu `this` in Java haben, werden beim Überschreiben auf die gleiche Weise kovariant verändert. Wenn mehrere Parameter denselben überschriebenen Typ haben, handelt es sich um binäre Methoden. Eine Regel in Ada besagt, dass alle Argumente, die für diese Parameter eingesetzt werden, genau den gleichen dynamischen Typ haben müssen. Das wird zur Laufzeit überprüft. Schlägt die Überprüfung fehl, wird eine Ausnahme ausgelöst. Methoden wie `equal` in obigem Beispiel sind damit sehr einfach programmierbar. Falls die zu vergleichenden Objekte unterschiedliche Typen haben, tritt eine Ausnahme auf, die an geeigneten Stellen abgefangen werden kann.

Ende des Einschubs für Interessierte

4.4 Überladene Methoden und Multimethoden

Dynamisches Binden erfolgt in Java (wie in vielen anderen objektorientierten Programmiersprachen auch) über den dynamischen Typ eines speziellen Parameters. Beispielsweise wird die auszuführende Methode in `x.equal(y)` durch den dynamischen Typ von `x` festgelegt. Der dynamische Typ von `y` ist für die Methodenauswahl irrelevant. Aber der deklarierte Typ von `y` ist bei der Methodenauswahl relevant, wenn `equal` überladen ist. Bereits der Compiler kann anhand des deklarierten Typs von `y` auswählen, welche der überladenen Methoden auszuführen ist. Der dynamische Typ von `y` ist dafür unerheblich.

Generell, aber nicht in Java, ist es möglich, dass dynamisches Binden auch den dynamischen Typ von `y` in die Methodenauswahl einbezieht. Dann legt nicht bereits der Compiler anhand des deklarierten Typs fest, welche überladene Methode auszuwählen ist, sondern erst zur Laufzeit des Programms wird die auszuführende Methode durch die dynamischen Typen von `x` und `y` bestimmt. In diesem Fall sprechen wir nicht von Überladen, sondern von *Multimethoden* [9]; es wird bei einem Methodenaufruf mehrfach dynamisch gebunden.

Leider wird Überladen viel zu oft mit Multimethoden verwechselt. Das führt zu schweren Fehlern. In Abschnitt 4.4.1 werden wir uns die Unterschiede deshalb deutlich vor Augen führen. In Abschnitt 4.4.2 werden wir sehen, dass Multimethoden auch in Sprachen wie Java recht einfach simulierbar sind.

4.4.1 Deklarierte versus dynamische Argumenttypen

Folgendes Beispiel soll verdeutlichen, dass bei der Auswahl zwischen überladenen Methoden nur der deklarierte Typ eines Arguments entscheidend ist, nicht der dynamische. Wir verwenden das Beispiel zu kovarianten Problemen aus Abschnitt 4.3.3:

```
Cow cow = new Cow();
Food grass = new Grass();
cow.eat(grass);          // Cow.eat(Food x)
cow.eat((Grass)grass);   // Cow.eat(Grass x)
```

Wegen dynamischem Binden wird `eat` auf jeden Fall in der Klasse `Cow` ausgeführt. Der Methodenaufruf in der dritten Zeile führt die überladene Methode mit dem Parameter vom Typ `Food` aus, da `grass` mit dem Typ `Food` deklariert ist. Für die Methodenauswahl ist es unerheblich, dass `grass` tatsächlich ein Objekt von `Grass` enthält; der dynamische Typ von `grass` ist `Grass`, da `grass` direkt vor dem Methodenaufruf mit einem Objekt von `Grass` initialisiert wird. Es zählt aber nur der deklarierte Typ. Der Methodenaufruf in der vierten Zeile führt die überladene Methode mit dem Parameter vom Typ `Grass` aus, weil der deklarierte Typ von `grass` wegen der Typumwandlung an dieser Stelle `Grass` ist. Typumwandlungen ändern ja den deklarierten Typ eines Ausdrucks.

Häufig ist bekannt, dass `grass` ein Objekt von `Grass` enthält und nimmt an, dass sowieso die Methode mit dem Parameter vom Typ `Grass` gewählt wird. Diese Annahme ist falsch! Es wird stets der deklarierte Typ verwendet, unabhängig davon, ob wir den dynamischen Typ kennen. Der Compiler darf nur den deklarierten Typ verwenden, sogar wenn er einen spezifischeren Typ als statischen Typ kennt.

Nehmen wir an, die erste Zeile des Beispiels sehe so aus:

```
Animal cow = new Cow();
```

Wegen dynamischen Bindens würde `eat` weiterhin in `Cow` ausgeführt. Aber zur Auswahl überladener Methoden kann der Compiler nur deklarierte Typen verwenden. Das gilt auch für den Empfänger einer Nachricht. Die überladenen

Methoden werden in `Animal` gesucht, nicht in `Cow`. In `Animal` ist `eat` nicht überladen, sondern es gibt nur eine Methode mit einem Parameter vom Typ `Food`. Daher wird in `Cow` auf jeden Fall die Methode mit dem Parameter vom Typ `Food` ausgeführt, unabhängig davon, ob der deklarierte Typ des Arguments `Food` oder (nach einem Cast) `Grass` ist. Wie das Beispiel zeigt, kann sich die Auswahl zwischen überladenen Methoden stark von der Intuition unterscheiden und ist von vielen Details abhängig. Daher ist besondere Vorsicht geboten.

Die Methoden `eat` in `Cow` und `Tiger` sind so überladen, dass es (außer für die Laufzeiteffizienz) keine Rolle spielt, welche der überladenen Methoden aufgerufen wird. Wenn der dynamische Typ des Arguments `Grass` ist, wird im Endeffekt immer die Methode mit dem Parametertyp `Grass` aufgerufen. Es ist empfehlenswert, Überladen nur so zu verwenden.

> *Faustregel:* Wir sollen Überladen nur so verwenden, dass es keine Rolle spielt, ob bei der Methodenauswahl deklarierte oder dynamische Typen der Argumente verwendet werden.

Unter folgender Bedingung ist die strikte Unterscheidung zwischen deklarierten und dynamischen Typen bei der Methodenauswahl nicht wichtig, das Überladen von Methoden also *sicher:* Für je zwei überladene Methoden gleicher Parameterzahl

- gibt es zumindest eine Parameterposition, an der sich die Parametertypen unterscheiden, wobei diese Typen nicht in Untertyprelation zueinander stehen und auch keinen gemeinsamen Untertyp haben,
- oder alle Parametertypen der einen Methode sind Obertypen der Parametertypen der anderen Methode, und bei Aufruf der einen Methode wird nichts anderes gemacht, als auf die andere Methode zu verzweigen, falls die entsprechenden dynamischen Typen der Argumente dies erlauben.

Das Problem der Verwechslung von dynamischen und deklarierten Typen könnte nachhaltig gelöst werden, indem zur Methodenauswahl generell die dynamischen Typen aller Argumente verwendet würden. Statt überladener Methoden hätten wir dann Multimethoden. Würde Java Multimethoden unterstützen, könnten wir die Klasse `Cow` im Beispiel aus Abschnitt 4.3.3 kürzer und ohne dynamische Typabfragen und -umwandlungen schreiben:

```
class Cow extends Animal {
    public void eat(Grass x) { ... }
    public void eat(Food x) {
        fallIll();
    }
} // Achtung: In Java ist diese Lösung falsch !!
```

Die Abfrage, ob `x` den dynamischen Typ `Grass` hat, hätten wir uns erspart, da `eat` mit dem Parametertyp `Food` bei Multimethoden nur aufgerufen wird, wenn der dynamische Argumenttyp nicht `Grass` ist.

Als Grund für die fehlende Unterstützung von Multimethoden in vielen heute üblichen Programmiersprachen wird häufig die höhere Komplexität der Methodenauswahl genannt. Der dynamische Typ der Argumente muss ja zur Laufzeit in die Methodenauswahl einbezogen werden. Im Beispiel mit der Multimethode `eat` ist jedoch, wie in vielen Fällen, in denen Multimethoden sinnvoll sind, kein zusätzlicher Aufwand nötig; eine dynamische Typabfrage auf dem Argument ist immer nötig, wenn der statische Typ kein Untertyp von `Grass` ist. Die Multimethodenvariante von `eat` kann sogar effizienter sein als die Variante mit Überladen, wenn der statische Typ des Arguments ein Untertyp von `Grass` ist, nicht jedoch der deklarierte Typ. Die Laufzeiteffizienz ist daher kaum ein Grund für fehlende Multimethoden in einer Programmiersprache.

Unter der Komplexität der Methodenauswahl verstehen wir nicht nur die Laufzeiteffizienz: Für Menschen ist nicht gleich erkennbar, unter welchen Bedingungen welche Methode ausgeführt wird. Eine Regel besagt, dass immer jene Methode mit den speziellsten Parametertypen, die mit den dynamischen Typen der Argumente kompatibel sind, auszuführen ist. Wenn wir `eat` mit einem Argument vom Typ `Grass` (oder einem Untertyp davon) aufrufen, sind die Parametertypen beider Methoden mit dem Argumenttyp kompatibel. Da `Grass` spezieller ist als `Food`, wird die Methode mit dem Parametertyp `Grass` ausgeführt. Diese Regel ist für die Methodenauswahl aber nicht hinreichend wenn Multimethoden mehrere Parameter haben, wie folgendes Beispiel zeigt:

```
public void eatTwice(Food x, Grass y) { ... }
public void eatTwice(Grass x, Food y) { ... }
```

Mit einem Aufruf von `eatTwice` mit zwei Argumenten vom Typ `Grass` sind beide Methoden kompatibel. Aber keine Methode ist spezieller als die andere. Es gibt verschiedene Möglichkeiten, mit solchen Mehrdeutigkeiten umzugehen. Eine Möglichkeit besteht darin, die erste passende Methode zu wählen; das wäre die Methode in der ersten Zeile. Es ist auch möglich, die Übereinstimmung zwischen Parametertyp und Argumenttyp für jede Parameterposition getrennt zu prüfen, und dabei von links nach rechts jeweils die Methode mit den spezielleren Parametertypen zu wählen; das wäre die Methode in der zweiten Zeile. CLOS (Common Lisp Object System [19]) bietet zahlreiche weitere Auswahlmöglichkeiten. Keine dieser Möglichkeiten bietet klare Vorteile gegenüber den anderen. Daher scheint eine weitere Variante günstig zu sein: Der Compiler verlangt, dass es immer genau eine eindeutige speziellste Methode gibt. Wir müssen eine zusätzliche Methode

```
public void eatTwice(Grass x, Grass y) { ... }
```

hinzufügen, die das Auswahlproblem beseitigt. Dieses Beispiel soll klar machen, dass Multimethoden für den Compiler und beim Programmieren eine höhere Komplexität haben als überladene Methoden. In üblichen Anwendungsbeispielen haben Multimethoden aber keine höhere Komplexität als überladene Methoden. Die Frage, ob Multimethoden in der Gesamtbetrachtung günstiger sind als überladene Methoden, bleibt offen.

In Java kommt es zu Fehlern, wenn wir unbewusst Überladen statt Überschreiben verwenden, wenn wir also eine Methode überschreiben wollen, die Methode im Untertyp sich aber in den Parametertypen von der Methode im Obertyp unterscheidet. Abhilfe schafft nur spezielle Achtsamkeit und die konsequente Verwendung von `@Override`-Annotationen. Nur Ergebnistypen dürfen ab Java 1.5 kovariant verschieden sein. Es sei darauf hingewiesen, dass das Überladen von Methoden nicht mit Vererbung zusammenhängt; überladene Methoden können auch direkt in nur einer einzigen Klasse eingeführt werden.

4.4.2 Simulation von Multimethoden

Multimethoden verwenden mehrfaches dynamisches Binden: Die auszuführende Methode wird dynamisch durch die Typen mehrerer Argumente bestimmt. In Java gibt es nur einfaches dynamisches Binden. Trotzdem können wir mehrfaches dynamisches Binden durch wiederholtes einfaches Binden simulieren. Wir nutzen mehrfaches dynamisches Binden für das Beispiel aus Abschnitt 4.3.3 und eliminieren damit dynamische Typabfragen und Typumwandlungen:

```
public abstract class Animal {
    public abstract void eat(Food food);
}
public class Cow extends Animal {
    public void eat(Food food) { food.eatenByCow(this); }
}
public class Tiger extends Animal {
    public void eat(Food food) { food.eatenByTiger(this); }
}
public abstract class Food {
    abstract void eatenByCow(Cow cow);
    abstract void eatenByTiger(Tiger tiger);
}
public class Grass extends Food {
    void eatenByCow(Cow cow) { ... }
    void eatenByTiger(Tiger tiger) { tiger.showTeeth(); }
}
public class Meat extends Food {
    void eatenByCow(Cow cow) { cow.fallIll(); }
    void eatenByTiger(Tiger tiger) { ... }
}
```

Die Methoden `eat` in `Cow` und `Tiger` rufen Methoden in `Food` auf, die die eigentlichen Aufgaben durchführen. Hier nehmen wir an, dass alle diese Klassen im selben Paket liegen, sodass Default-Sichtbarkeit für die zusätzlichen Methoden reicht. Scheinbar verlagern wir die Arbeit nur von den Tieren zu den Futterarten. Dabei passiert aber etwas Wesentliches: In `Grass` und `Meat` gibt es nicht nur *eine* entsprechende Methode, sondern je eine für Objekte von `Cow` und `Tiger`. Bei einem Aufruf von `animal.eat(food)` wird zweimal dynamisch

gebunden. Das erste dynamische Binden unterscheidet zwischen Objekten von `Cow` und `Tiger` und spiegelt sich im Aufruf von `eatenByCow` und `eatenByTiger` wider. Ein zweites dynamisches Binden unterscheidet zwischen Objekten von `Grass` und `Meat`. In den Unterklassen von `Food` sind insgesamt vier Methoden implementiert, die alle Kombinationen von Tierarten mit Futterarten abdecken.

Statt `eatenByCow` und `eatenByTiger` hätten wir auch einen gemeinsamen Namen wählen können (= Überladen), da sich die Typen der formalen Parameter eindeutig unterscheiden (keine gemeinsamen Untertypen).

Stellen wir uns vor, diese Lösung sei dadurch zustandegekommen, dass wir eine ursprüngliche Lösung mit Multimethoden in Java implementiert und dabei für den formalen Parameter einen zusätzlichen Schritt dynamischen Bindens eingeführt hätten. Damit wird klar, wie mehrfaches dynamisches Binden durch wiederholtes einfaches dynamisches Binden ersetzbar ist. Bei Multimethoden mit mehreren Parametern muss entsprechend oft dynamisch gebunden werden. Sobald wir den Übersetzungsschritt verstanden haben, können wir ihn ohne intellektuelle Anstrengung, aber mit viel Schreibaufwand, für vielfaches dynamisches Binden durchführen.

Diese Lösung kann auch dadurch erzeugt worden sein, dass in der ursprünglichen Lösung aus Abschnitt 4.3.3 `if`-Anweisungen mit dynamischen Typabfragen durch dynamisches Binden ersetzt wurden. Nebenbei sind auch die Typumwandlungen verschwunden. Auch diese Umformung ist automatisch durchführbar. Wir haben damit die Möglichkeit, dynamische Typabfragen genauso wie Multimethoden aus Programmen zu entfernen und damit die Struktur des Programms zu verbessern.

Mehrfaches dynamisches Binden wird in der Praxis benötigt. Die Lösung wie im Beispiel entspricht dem *Visitor-Pattern*, einem klassischen Entwurfsmuster – siehe Kapitel 6. Klassen wie `Food` nennen wir *Visitorklassen,* die darin enthaltenen Methoden wie `eatenByCow` sind *Visitormethoden.* Klassen wie `Animal` heißen *Elementklassen.* Visitor- und Elementklassen sind oft gegeneinander austauschbar. Z. B. könnten die eigentlichen Implementierungen (Visitormethoden) in den Tier-Klassen stehen, die in den Futter-Klassen aufgerufen werden.

Das Visitor-Pattern hat einen bedeutenden Nachteil: Die Anzahl der benötigten Methoden wird schnell sehr groß. Nehmen wir an, wir hätten m unterschiedliche Tierarten und n Futterarten. Zusätzlich zu den m Methoden in den Elementklassen werden $m{\cdot}n$ Visitormethoden benötigt. Noch rascher steigt die Methodenanzahl mit der Anzahl der dynamischen Bindungen. Bei $k \geq 2$ dynamischen Bindungen mit n_i Möglichkeiten für die i-te Bindung ($i = 1 \dots k$) werden $n_1{\cdot}n_2 \cdots n_k$ inhaltliche Methoden und zusätzlich $n_1{+}n_1{\cdot}n_2{+}\cdots{+}n_1{\cdot}n_2 \cdots n_{k-1}$ Methoden für die Verteilung benötigt, also sehr viele. (Wir vermeiden hier die Begriffe Element- bzw. Visitormethode, da die Klassenhierarchien mit den Indizes $i = 2 \dots k-1$ gleichzeitig Element- und Visitorklassen sind.) Für $k = 4$ und $n_1, \dots, n_4 = 10$ kommen wir auf 11.110 Methoden. Außer für sehr kleine k und kleine n_i ist diese Technik nicht sinnvoll einsetzbar. Vererbung kann die Zahl der nötigen Methoden meist nur unwesentlich verringern.

Die Anzahl der Methoden lässt sich manchmal durch kreative Ansätze deutlich reduzieren. Am erfolgversprechendsten sind Lösungen, welche die Anzahl

der Klassen n_i klein halten. Beispielsweise müssen wir nicht immer zwischen allen möglichen Tier- und Futterarten unterscheiden, sondern kann Klassen für Gruppen mit gemeinsamen Eigenschaften bilden. Das ist vor allem dann leicht möglich, wenn nicht gleichzeitig auf alle Details der Tier- und Futterarten zugegriffen werden muss. So könnten wir statt `eatenByCow` eine Methode `eatenByVegetarian` schreiben, die nicht nur für Rinder, sondern auch für Kaninchen und Elefanten verwendbar ist. Es muss auch nicht nur das Visitor-Pattern alleine sein. Oft hilft eine einfache Beschreibung der *Rolle* eines Objekts: Dabei enthält jedes Tier oder jede Tierart eine Variable mit einer Referenz auf eines von ganz wenigen Objekten zur Beschreibung der erlaubten Futterarten. Wir können das verfügbare Futter an eine Methode in diesem Objekt weiterleiten, wo das Futter mit der erlaubten Futterart (zwar wieder über mehrfaches dynamisches Binden, aber mit kleinem n_i) verglichen wird.

4.5 Annotationen und Reflexion

Die Idee hinter Annotationen ist einfach: Unterschiedliche Programmteile werden mit Markierungen (die wir Annotationen nennen) versehen und das Laufzeitsystem sowie Entwicklungswerkzeuge prüfen das Vorhandensein bestimmter Markierungen und reagieren darauf entsprechend. Ohne explizite Überprüfungen haben Annotationen keinerlei Auswirkungen auf die Programmsemantik; sie werden einfach ignoriert. So einfach dieses Konzept zu sein scheint, so komplex sind Details der Umsetzung. Einerseits sollte das Hinzufügen von Annotationen zu Java die Syntax nicht allzu sehr ändern und trotzdem aus der Syntax klar hervorgehen, dass es sich um möglicherweise ignorierte Programmteile handelt. Andererseits muss es möglich sein, zur Laufzeit das Vorhandensein von Annotationen abzufragen. Dafür wird Reflexion eingesetzt. In Abschnitt 4.5.1 untersuchen wir, wie diese Details in Java gelöst wurden, und in Abschnitt 4.5.2 betrachten wir Anwendungsbeispiele.

4.5.1 Annotationen und Reflexion in Java

Syntax. Zur klaren Unterscheidung von anderen Sprachkonstrukten beginnt jede Annotation in Java mit dem Zeichen „`@`". Sie steht unmittelbar vor dem Programmteil, auf den sie sich bezieht. Beispielsweise können wir die Annotation `@Override` vor eine Methodendefinition schreiben. Der Compiler prüft, ob die Methodendefinition mit dieser Annotation versehen ist und verlangt nur in diesem Fall, dass die Methode eine andere Methode überschreibt. Ein Compiler, der `@Override` nicht versteht, könnte diese Überprüfung theoretisch auch weglassen. Aber praktisch jeder Compiler versteht `@Override`, da es sich dabei um eine vom System vorgegebene Annotation handelt. Wir können auch eigene Annotationen erfinden. Außerdem können Annotationen Argumente enthalten.

Eigene Annotationen müssen deklariert werden, bevor sie verwendbar sind. Dafür wurde die Syntax von Interface-Definitionen übernommen und abgewandelt. Hier ist ein Beispiel für die Definition einer etwas komplexeren Annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
    String who();     // author of bug fix
    String date();    // when was bug fixed
    int    level();   // importance level 1-5
    String bug();     // description of bug
    String fix();     // description of fix
}
```

Solche Annotationen können wir zu Klassen, Interfaces und Enums hinzufügen um auf Korrekturen hinzuweisen. Das könnte so aussehen:

```
@BugFix(who="Kaspar", date="1.10.2023", level=3,
        bug="class unnecessary and maybe harmful",
        fix="content of class body removed")
public class Buggy { }
```

Die Einträge in der Definition der Annotation beschreiben Datenfelder, die in der Annotation gesetzt werden. Auch wenn diese Einträge syntaktisch wie Methodendeklarationen aussehen, gibt es doch deutliche Unterschiede zu normalen Methoden. Die Parameterlisten müssen leer sein und die erlaubten Ergebnistypen sind stark eingeschränkt: Nur elementare Typen (wie `int`), Enum-Typen, `String`, `Class` und andere Annotationen sowie eindimensionale Arrays dieser Typen sind erlaubt. Wie im Beispiel wird häufig `String` verwendet.

Die Definition von `BugFix` ist selbst mit zwei Annotationen versehen. An ihnen fällt auf, dass die Argumente in den runden Klammern nicht die Form `name=wert` haben, sondern nur einfache Werte darstellen. Der Name des Arguments kann weggelassen werden wenn die Annotation nur ein Argument namens `value` hat. Ebenso können die runden Klammern weggelassen werden, wenn die Annotation keine Argumente hat, etwa bei `@Override`. Das Argument von `@Target` ist ein Array, die geschwungenen Klammern stellen also ein simples Aggregat zur Initialisierung eines (in diesem Fall einelementigen) Arrays dar. Bei einelementigen Arrays können wir die geschwungenen Klammern weglassen.

Annotationen auf der Definition von Annotationen haben folgende Bedeutungen: Das Argument von `@Target` legt fest, was annotiert werden kann. Es ist ein Array von Werten des Enums `ElementType` mit Werten wie `METHOD`, `TYPE`, `PARAMETER`, `CONSTRUCTOR` und so weiter; siehe die Schnittstellenbeschreibung von `ElementType`. Die gerade definierte Annotation kann an alle Sprachelemente angeheftet werden, die im Array vorkommen. Ohne `@Target` ist die gerade definierte Annotation überall anheftbar.

`@Retention` legt fest, wie weit die gerade definierte Annotation sichtbar bleiben soll. Mit dem Wert `SOURCE` der Enum `RetentionPolicy` wird die Annotation vom Compiler genau so verworfen wie Kommentare. Solche Annotationen sind nur für Werkzeuge, die auf dem Source-Code operieren, von Interesse. Der Wert `CLASS` sorgt dafür, dass die Annotation in der übersetzten Klasse vorhanden bleibt, aber während der Programmausführung nicht mehr sichtbar ist. Das

ist nützlich für Werkzeuge, die auf dem Byte-Code operieren. Schließlich sorgt der Wert `RUNTIME` dafür, dass die Annotation auch zur Laufzeit zugreifbar ist.

Zusätzlich wären in der Definition einer Annotation die parameterlosen Annotationen `@Documented` und `@Inherited` verwendbar. Erstere sorgt dafür, dass die Annotation in der generierten Dokumentation vorkommt, letztere dafür, dass das annotierte Element auch in einem Untertyp als annotiert gilt.

Wir können Default-Belegungen für Parameter von Annotationen angeben. Beispielsweise wird die Definition von `BugFix` um folgende Zeile (innerhalb der geschwungenen Klammern) erweitert:

```
    String comment() default "";
```

Aufgrund der Default-Belegung müssen wir bei der Verwendung kein Argument für `comment` angeben. Das ist besonders sinnvoll, wenn die Zeile erst später hinzugefügt wird, also `@BugFix` schon vorher verwendet wurde. Schon existierende Verwendungen können wir wegen der Default-Belegung unverändert lassen.

Verwendung zur Laufzeit. Wurde durch `@Retention(RUNTIME)` bestimmt, dass eine Annotation auch zur Laufzeit zugreifbar ist, dann generiert der Compiler ein entsprechendes Interface. Dieses sieht für obiges Beispiel so aus:

```
public interface BugFix extends Annotation {
    String who();
    String date();
    int    level();
    String bug();
    String fix();
}
```

Wir können wie in folgendem Code-Stück auf die Annotation zugreifen:

```
    String s = "";
    BugFix a = Buggy.class.getAnnotation(BugFix.class);
    if (a != null) {  // null if no such annotation
        s += a.who() + " fixed a level " + a.level() + " bug";
    }
```

Die Basis für Zugriffe ist die Klasse `Buggy`. So wie `getClass` die interne Darstellung der Klasse eines Objekts als Objekt von `Class` ermittelt, so enthält die Pseudovariable `class` jeder Klasse das entsprechende Objekt der Klasse `Class`. Dieses Objekt versteht die Nachricht `getAnnotation` mit einem Argument, das den Typ der Annotation beschreibt. Obwohl `BugFix` ein Interface ist, gibt `BugFix.class` ein Objekt von `Class`, genauer `Class<BugFix>` zurück, welches das Interface beschreibt. Die zweite Zeile im Code-Stück liefert das Objekt von `BugFix`, mit dem die Klasse `Buggy` annotiert ist (oder `null` falls es keine solche Annotation gibt). Auf die einzelnen in `BugFix` beschriebenen Methoden wird ganz normal zugegriffen, um die Datenfelder der Annotation auszulesen.

Obiges Code-Stück ist nur sinnvoll verwendbar, wenn wir genau wissen, auf welche Annotation wir zugreifen möchten. Durch `getAnnotations` können wir alle Annotationen der Klasse gleichzeitig auslesen, ohne deren Typen zu kennen:

```
Annotation[] as = Buggy.class.getAnnotations();
for (Annotation a : as) {
    if (a instanceof BugFix)
        String s = ((BugFix)a).who; ...
}
```

Aber mit dem so erzeugten Array von Annotationen können wir meist nur dann weiterarbeiten, wenn wir wissen, welche Annotationen wir haben möchten. Auf die Datenfelder können wir ja nur zugreifen, wenn wir vorher einen Cast auf den richtigen Typ der Annotation machen.

Die Technik, mit der wir zur Laufzeit auf Annotationen zugreifen, nennt sich *Reflexion* bzw. *Reflection* oder *Introspektion.* Reflexion erlaubt uns, zur Laufzeit auf viele Details eines Programms zuzugreifen. Ausgangspunkt ist meist ein Objekt vom Typ `Class`. `Class` implementiert eine Reihe von Methoden, mit denen sich die Details der Klasse ansehen lassen. So wie wir über `getAnnotation` und `getAnnotations` Informationen über Annotationen bekommt, können wir uns über `getMethod` und `getMethods` Informationen über Methoden und über `getField` und `getFields` Informationen über Objekt- und Klassenvariablen holen. Ähnliches gilt für Konstruktoren, Oberklassen, das Paket und so weiter. Weitere Methoden erlauben beispielsweise die Erzeugung neuer Objekte der Klasse oder fragen den Namen der Klasse ab.

Informationen über Methoden sind in Objekten des Typs `Method` enthalten (die wir etwa durch `getMethods` über ein Objekt von `Class` bekommen). Wie `Class` bietet `Method` Methoden an, um Annotationen und viel anderes Nützliches abzufragen. Wir können etwa die Typen der formalen Parameter und des Ergebnisses sowie Annotationen auf den formalen Parametern abfragen. Über `invoke` können wir die Methode auch aufrufen. Dabei müssen wir den Empfänger der Nachricht und passende Argumente bereitstellen. Wenn der Empfänger oder ein Argument einen unpassenden Typ hat, oder wenn der Aufruf der Methode wegen eingeschränkter Sichtbarkeit nicht erlaubt ist, wird eine Ausnahme ausgelöst. Abgesehen davon, dass die Überprüfungen, die üblicherweise der Compiler macht, erst zur Laufzeit passieren, hat ein solcher Aufruf die gleiche Semantik wie ein normaler Methodenaufruf. Es wird auch dynamisch gebunden.

Ähnliches gilt für Informationen über Variablen in Objekten des Typs `Field`. Wir können über Reflexion die Werte sichtbarer Variablen lesen und schreiben und natürlich auch Annotationen abfragen.

Reflexion ist in der Programmierung eine äußerst mächtige Technik. Wir können damit sehr flexibel fast alles zur Laufzeit entscheiden, was üblicherweise schon vor dem Compilieren festgelegt werden muss; nur das Ändern von Klassen zur Laufzeit ist in Java verboten. Genau in dieser Flexibilität liegt jedoch die größte Gefahr der Reflexion. Das Programm wird gänzlich undurchschaubar und kaum wartbar, wenn wir diese Freiheit auf unkontrollierte Weise nutzen.

4.5.2 Anwendungen von Annotationen und Reflexion

Übliche Annotationen. So wie bei vielen anderen komplexeren Sprachkonzepten verhält es sich auch mit Annotationen: Sie liefern einen wesentlichen Beitrag und sind daher heute unverzichtbar, gleichzeitig verwenden wir sie in der Praxis (wenn überhaupt) meist nur in ihren einfachsten Formen. Der unverzichtbare Beitrag besteht darin, dass Annotationen zusätzliche syntaktische Elemente in Programmen erlauben, ohne dafür die Programmiersprache ändern zu müssen. Gerade für vielseitig und in großem Umfang eingesetzte Sprachen wie Java sind Änderungen der Syntax kaum durchsetzbar. Das gilt vor allem für Änderungen, die nur wenige Einsatzgebiete betreffen. Annotationen ermöglichen syntaktische Erweiterungen auch für ganz spezielle Einsatzgebiete, ohne gleichzeitig andere Einsatzgebiete mit unnötiger Syntax zu überladen. Beim Schreiben von Programmen außerhalb dieser speziellen Gebiete werden wir kaum etwas von der Existenz entsprechender Annotationen mitbekommen.

Häufig verwenden wir `@Override`. Statt dieser Annotation wäre auch ein Modifier sinnvoll gewesen, aber aufgrund der geschichtlichen Entwicklung hat sich eine Annotation angeboten. Wir können jede Annotation als Modifier sehen, den ein (Pre-)Compiler oder das Laufzeitsystem versteht.

Manchmal stolpern wir über eine `@Deprecated`-Annotation, mit der Programmelemente gekennzeichnet werden, die wir nicht mehr verwenden sollten. Eigentlich stellt sie nur eine Form von Kommentar dar. Die Annotation ermöglicht jedoch, dass ein Compiler bei Verwendung dieser Programmelemente eine Warnung ausgibt.

Eine gefährliche Rolle spielt `@SuppressWarnings`. Diese Annotation weist den Compiler an, alle Warnungen zu unterdrücken, die im Argument durch Zeichenketten beschrieben sind. Auch wenn manche Warnung lästig ist, sollten wir von der Verwendung solcher Annotationen Abstand nehmen. Warnungen haben ja einen Grund, den wir nicht vernachlässigen sollten. Es kommt gar nicht so selten vor, dass wir eine solche Annotation „vorübergehend“ in den Code schreiben, weil wir uns erst später um ein Problem kümmern möchten und dann darauf vergessen. Auch wenn wir genau wissen, dass sich ein Compiler mit einer Warnung irrt, kann das Abdrehen dieser Warnung die Fehlersuche erschweren, etwa nach einer Programmänderung. Ein weiterer Grund ist fehlende Portabilität: Unterschiedliche Compiler verwenden unterschiedliche Zeichenketten zur Beschreibung von Warnungen, sodass manche Compiler durch die Annotationen eigenartige Warnungen generieren können.

Mit der Annotation `@FunctionalInterface` können seit Java 8 Interfaces gekennzeichnet werden, die genau eine abstrakte Methode enthalten. Solche Interfaces sind als Typen von Lambda-Ausdrücken verwendbar,[3] wobei die abstrakte Methode beschreibt, wie die entsprechende Funktion aufzurufen ist. Beispielsweise haben wir in Abschnitt 2.3.2 den Typ `ToIntFunction<Point>`

[3] Lambda-Ausdrücke können auch Interfaces mit nur einer abstrakten Methode als Typ haben, die nicht mit der Annotation `@FunctionalInterface` versehen sind. Die Annotation macht nur darauf Aufmerksam, dass das Interface für diesen Einsatzzweck konzipiert wurde und auch in Zukunft für solche Verwendungen zur Verfügung stehen wird.

verwendet, wobei das Interface als `@FunctionalInterface` gekennzeichnet ist. Es enthält (nach Übersetzung der Generizität) nur eine abstrakte Methode

```
int applyAsInt(Point value);
```

wodurch Lambda-Ausdrücke dieses Typs durch `applyAsInt(...)` aufrufbar sind. Die Annotation verhindert, dass das Interface später um zusätzliche abstrakte Methoden erweitert wird. Methoden mit Default-Implementierungen dürfen jedoch zusätzlich vorhanden sein oder später hinzugefügt werden, ohne die Aussage der Annotation zu verletzen.

Reflexion. Reflexion ist eine Variante der *Metaprogrammierung* (siehe Abschnitt 1.5.3), einer schon sehr alten Programmiertechnik, mit der es viel Erfahrung gibt. Während durch Metaprogrammierung das gesamte Programm zur Laufzeit sicht- und änderbar ist, kann Reflexion die Programmstruktur nicht ändern. Es ist bekannt, dass mit diesen Techniken in speziellen Fällen sehr viel erreichbar ist. Aber auch die Gefahren sind bekannt. Daher versuchen wir meist, solche Techniken in der tagtäglichen Programmierung zu vermeiden und greifen nur darauf zurück, wenn wir keine einfachere Lösung finden.

Hier ist ein einfaches Beispiel für den Einsatz von Reflexion:

```
static void execAll(String n, Object... objs) {
   for (Object o : objs) {
      try { o.getClass().getMethod(n).invoke(o); }
      catch(Exception ex) {...}
   }
}
```

In den als Parameter übergebenen Objekten wird jeweils eine parameterlose Methode aufgerufen, die einen ebenfalls als Parameter übergebenen Namen hat. Dieser Name kann genauso wie die Objekte zur Laufzeit bestimmt werden. Auf den ersten Blick geht das ganz einfach. Bei genauerem Hinsehen fällt auf, dass bei den Aufrufen einiges passieren kann, mit dem wir vielleicht nicht rechnen – ganz zu schweigen von der Gefahr, dass wir nicht wissen, was die mit `invoke` aufgerufenen Methoden machen (möglicherweise Daten weiterleiten bzw. zerstören oder jemandem Zugang zum System verschaffen). Möglicherweise ist die Methode nicht `public`, verlangt (andere) Argumente, oder existiert gar nicht. In diesen Fällen werden Ausnahmen ausgelöst, mit denen wir umgehen müssen.

Wie das Beispiel zeigt, ist die Verwendung der Reflexion im Grunde sehr einfach. Schwierigkeiten verursacht nur das ganze Rundherum, z. B. der notwendige Umgang mit vielen Sonderfällen, die in der normalen Programmierung vom Compiler ausgeschlossen werden. Die Gefahr kommt hauptsächlich daher, dass wir keinerlei Verhaltensbeschreibungen der mit `invoke` aufgerufenen Methoden haben. Wir haben nicht einmal intuitive Vorstellungen davon. Bei entsprechender Organisation könnten wir alle diese Probleme lösen. Aber die Erfahrung zeigt, dass die reflexive Programmierung dennoch gefährlich ist und zu Wartungsproblemen führt.

Ein Spezialbereich. Wir betrachten nun JavaBeans-Komponenten als Beispiel für den Einsatz von Reflexion und Annotationen. JavaBeans ist ein Werkzeug, mit dem grafische Benutzeroberflächen ganz einfach aus Komponenten aufgebaut werden. Der Großteil der Arbeit wird von Werkzeugen bzw. fertigen Klassen erledigt. JavaBeans-Komponenten sind gewöhnliche Klassen, die bestimmte Namenskonventionen einhalten.

Ein JavaBeans-Konzept sind „Properties“ – Objektvariablen, deren Werte von außen zugreifbar sind. Properties führen wir ein, indem wir Getter und Setter für die Objektvariablen schreiben. Existieren z. B. die Methoden

```
public void setProp(int x) { ... }
public int getProp() { ... }
```

nehmen die Werkzeuge automatisch an, dass `prop` eine Property des Typs `int` ist. Existiert nur eine dieser Methoden, ist die Property nur les- oder schreibbar. Lesbare Properties des Typs `boolean` können statt mit `get` auch mit `is` beginnen. Die grafische Benutzeroberfläche eines Werkzeugs erlaubt über Menüs simple Zugriffe auf Properties. Dabei findet und verwendet es Properties über Reflexion. Fast alle nötige Information steckt in den Namen, Ergebnistypen und Parametertypen der Methoden. Zum Auffinden komplexerer Konzepte wie „Events“ wird ähnlich vorgegangen.

Bezüglich dieser Vorgehensweise von JavaBeans-Komponenten scheiden sich die Geister. Einerseits können solche Werkzeuge die Entwicklung grafischer Benutzeroberflächen deutlich vereinfachen. Andererseits ist ein Programmierstil, der hauptsächlich auf Gettern und Settern aufbaut, vielleicht in der prozeduralen Programmierung akzeptabel, in der objektorientierten Programmierung aber nicht. Entsprechend propagieren Vertreter des Einsatzes von Werkzeugen wie JavaBeans die Verwendung von Gettern und Settern, die sich genau an die vom Werkzeug vorgegebenen Namenskonventionen halten, während Vertreter eines fortgeschritteneren objektorientierten Stils Getter und Setter so gut wie möglich meiden und (sollten sie sich nicht ganz vermeiden lassen) bewusst nicht an solche Namenskonventionen halten.

In seltenen Fällen benötigen JavaBeans Information, die nicht über Reflexion verfügbar ist. Dafür gibt es z. B. die `@ConstructorProperties`-Annotation: In übersetzten Klassen ist kaum feststellbar, welcher Parameter eines Konstruktors welcher Property entspricht. Die Argumente einer solchen Annotation zählen einfach die Properties entsprechend der Parameterreihenfolge auf und machen diese Information dadurch über Reflexion zugänglich. Diese Information ist im Zusammenhang mit JavaBeans sinnvoll. Wird die Klasse nicht als JavaBean verwendet, stört die Annotation nicht; sie wird einfach ignoriert.

JavaBeans-Komponenten sind für den Bereich der Hobby-Programmierung konzipiert. Die für professionelle Anwendungen ausgelegte Java-EE (Enterprise-Edition) verwendet eher fortgeschrittenere Konzepte, etwa EJB (Enterprise-JavaBeans) als Komponentenmodell mit vielen Möglichkeiten zur Darstellung von Geschäftslogiken, hauptsächlich in Web-Anwendungen. Dabei kommen Annotationen in großem Stil zum Einsatz (mehr als 30 verschiedene). Beispielsweise legt `@TransactionAttribute` fest, ob eine Methode innerhalb einer Transak-

tion zu verwenden ist. Mittels `@Stateful` bzw. `@Stateless` wird neben weiteren Eigenschaften spezifiziert, ob eine „Session-Bean“ – ein für die Dauer einer Sitzung existierendes Objekt – zustandsbehaftet sein soll oder nicht. Mit EJB müssen wir uns schon intensiv auseinandersetzen, bevor wir es sinnvoll verwenden können. Über die vielen Annotationen ergibt sich beinahe schon eine eigene Sprache innerhalb von Java.

4.6 Aspektorientierte Programmierung

Um die aspektorientierte Programmierung verstehen zu können, müssen wir sie aus zwei gänzlich unterschiedlichen Sichtweisen betrachten. Aus der konzeptuellen Sicht wird die Denkweise und Einbettung in das objektorientierte Paradigma auf sehr hoher Ebene deutlich, während aus Sicht der Implementierung auf niedriger Ebene in das Gefüge der Objekte eingegriffen wird. Nur wenn wir die hohe und tiefe Ebene gleichzeitig im Auge haben, können wir die aspektorientierte Programmierung gewinnbringend einsetzen. Wir konzentrieren uns zunächst ganz auf die konzeptuelle Sicht und bewegen uns erst später auf die Implementierungsebene.

4.6.1 Konzeptuelle Sichtweise

Ergebnisse von Berechnungen hängen von folgenden drei Faktoren ab:

- dem Programm, das für die Berechnungen verwendet wird,
- der Semantik der Sprache, in der das Programm geschrieben ist,
- den Daten, auf die das Programm angewandt wird.

Wenn wir die Ergebnisse der Berechnungen ändern wollen, können wir das Programm oder die Daten anpassen. Beides haben wir häufig gemacht. Aber, wie obige Aufzählung zeigt, gibt es noch eine dritte Möglichkeit: Wir können das Programm und die Daten unverändert lassen und die Semantik der Sprache an die Änderungswünsche anpassen. Das klingt provokant, weil wir beim Programmieren in der Regel davon ausgehen, dass die Semantik der Sprache stabil bleibt. Dennoch strebt die aspektorientierte Programmierung genau das an: Ergebnisse von Berechnungen sollen auf gewisse Weise abgeändert werden, ohne den Programmtext und die Daten zu ändern. Das läuft auf eine Änderung der Sprachsemantik hinaus. Diese muss jedoch so erfolgen, dass Erwartungen beim Programmieren nicht auf unkontrollierbare Weise verletzt werden.

Die aspektorientierte Programmierung bewegt sich in einem Graubereich, was die genaue Zuordnung zu Programm, Daten oder Sprachsemantik betrifft. Solche Graubereiche finden wir etwa in der Metaprogrammierung (einschließlich Reflexion), in der Programme selbst als ausführbare Daten angesehen werden und in der Precompilation, in der ein Programm umgeformt wird, bevor es zur Ausführung kommt. Metaprogrammierung und Precompilation sind unterschiedliche Konzepte mit gemeinsamen Eigenschaften: Es handelt sich um

äußerst mächtige (also ausdrucksstarke) und gefährliche (schwer kontrollierbare) Werkzeuge. Die aspektorientierte Programmierung verwendet zwar Metaprogrammierung und Precompilation als Hilfsmittel zur Implementierung, zielt aber darauf ab, die damit verbundenen Gefahren (auch durch Reduktion der Mächtigkeit) bestmöglich zu vermeiden. Metaprogrammierung und Precompilation werden hinter einem anderen konzeptuellen Ansatz versteckt. Wenn wir Metaprogrammierung und Precompilation als nicht vorhanden betrachten, im Hintergrund aber dennoch versteckt einsetzen, ergibt sich etwas, was wir uns gut als Modifikation der Sprachsemantik vorstellen können. Tatsächlich bleiben alle Änderungen der Semantik immer innerhalb des durch Metaprogrammierung und Precompilation vorgegebenen Rahmens. Dieser Rahmen wird nur zu einem kleinen Teil ausgeschöpft, nur so weit, dass die Erwartungen beim Programmieren bestmöglich erhalten bleiben.

Betrachten wir ein abstraktes Beispiel auf hoher Ebene, um eine Vorstellung davon zu bekommen, in welcher Form die Semantik der Sprache durch aspektorientierte Programmierung beeinflusst werden soll. Nehmen wir an, wir seien dafür verantwortlich, dass nur autorisierte Personen Zugriff auf die Software einer Bank bekommen. Die umfangreiche Bankensoftware enthält Funktionalität für die Verwaltung von Konten, Geldtransfers, Krediten, Wertpapieren und Ähnlichem. Sie wurde nach den Prinzipien der objektorientierten Programmierung entwickelt, besteht also aus zahlreichen Paketen und Klassen, für jede Funktionalität mindestens eine. Unsere Aufgabe besteht darin, ein Paket von Klassen zu entwickeln, das für die korrekte Authentifizierung beim Einloggen sorgt. Je nach Person werden unterschiedliche Rechte für den Zugriff auf manche Teile der Software vergeben; Kunden bekommen weniger Rechte als Bankangestellte, für Kreditvergaben zuständige Angestellte andere Rechte als Wertpapierverkäufer und so weiter. Es ist keine leichte Aufgabe, die unterschiedlichen Rechte richtig hinzukriegen, aber noch viel schwieriger ist es, dafür zu sorgen, dass alle Teile der Bankensoftware vor jedem Zugriff überprüfen, ob zugreifende Personen von uns die für den Zugriff nötigen Rechte bekommen haben. Für die meisten der Klassen, in denen die Überprüfungen erfolgen sollen, sind wir nicht zuständig, wir dürfen nicht darauf zugreifen und sie schon gar nicht verändern, einige dieser Klassen sind vielleicht noch gar nicht entwickelt oder werden häufig geändert. Das scheint eine unlösbare Aufgabe zu sein. Ein Ausweg könnte darin bestehen, die Semantik der Programmiersprache durch aspektorientierte Programmierung so abzuändern, dass Überprüfungen der Rechte an kritischen Programmstellen automatisch erfolgen, ohne dafür die Klassen für die Verwaltung von Konten, Geldtransfers und so weiter anfassen zu müssen. Kritische Stellen könnten beispielsweise alle Methodenaufrufe sein. Wahrscheinlich würden dadurch aber so viele Überprüfungen der Rechte entstehen, dass die Effizienz massiv darunter leidet. Vielleicht reicht es auch, wenn Überprüfungen nur bei Aufrufen von Methoden erfolgen, die in anderen Paketen liegen als die Aufrufer. Besser auf das Ziel hin ausgerichtet wären Überprüfungen an allen Stellen, an denen auf eine Datenbank zugegriffen wird, weil die zugegriffenen Datenfelder auch Information darüber liefern, für wen die Daten zugreifbar sein sollen. Dabei kann es passieren, dass Daten ohne weitere Datenbankzugriffe an andere

Programmteile übergeben und bei den Überprüfungen übersehen werden. Möglicherweise waren die Architekten der Bankensoftware schlau genug, bestimmte Namenskonventionen und das Mitführen bestimmter Parameter vorzuschreiben, sodass kritische Stellen für die Überprüfungen aus Methodennamen folgen und Parameterwerte beim Ermitteln der nötigen Rechte helfen. Jedenfalls ist es gar nicht notwendig, dass in allen Klassen Programmtexte für das Überprüfen der Rechte vorhanden sind, solange die projektbezogenen Konventionen eingehalten werden. Die eigentlichen Überprüfungen können auch im Nachhinein durch Abändern der Sprachsemantik hinzugefügt werden. Abänderungen betreffen nur zusätzliche Überprüfungen, die, wenn nötig, vielleicht Ausnahmen auslösen, mit deren Auftreten sowieso immer gerechnet werden muss, die Ausführung des Programms sonst aber in keiner Weise beeinträchtigen. Einzelne Klassen können unabhängig von den Überprüfungen der Rechte weiterentwickelt werden. Auch die Art der Prüfungen kann relativ leicht verändert werden, ohne dafür Klassen abändern zu müssen, die nichts mit der Vergabe der Rechte zu tun haben.

Zusammengefasst: In erster Linie geht es darum, bestimmte Stellen in einem bestehenden Programm zu identifizieren. Kriterien dafür können äußerst vielfältig sein, etwa Zugehörigkeit zu bestimmten Klassen oder Paketen, charakteristische Namensbestandteile (etwa alle Methodennamen, die mit `get`, `set`, oder `access` beginnen), bestimmte Arten von Parametern und so weiter. Je genauer und zuverlässiger Programmstellen identifizierbar sind, desto mächtiger ist das Instrument der aspektorientierten Programmierung. Außerdem muss es möglich sein, bestimmte Informationen aus der Umgebung (als *Kontext* bezeichnet) identifizierter Programmstellen zu extrahieren, etwa die Information, welche Arten von Daten an dieser Stelle bearbeitet werden, welche Rechte dafür nötig sind und wer auf diese Daten zugreifen möchte. Schließlich muss an den identifizierten Stellen (über Precompilation zur Übersetzungszeit bzw. Metaprogrammierung zur Laufzeit) zusätzlicher Programmcode ausgeführt werden, der auch auf Informationen aus dem Kontext zugreifen kann.

In Abschnitt 1.3.2 haben wir Aspekte als eine statische Form der Parametrisierung eingeführt. Parametrisierung bedeutet, dass in einem Programm Lücken gelassen werden, die erst später (in diesem Fall noch vor der Ausführung) befüllt werden. Als Lücken können alle identifizierbaren Programmstellen betrachtet werden, an denen der Mechanismus einhaken kann, das sind fast alle Stellen in einem Programm. Befüllt werden die Lücken mit zusätzlichem Code, der an identifizierten Stellen eingeschleust wird. Wie bei jeder Form der Parametrisierung besteht eine Abhängigkeit zwischen den Lücken und dem, womit die Lücken befüllt werden. In der aspektorientierten Programmierung äußert sich diese Abhängigkeit darin, dass bei der Identifizierung der Programmstellen und der Beschreibung des Kontexts bestimmte Annahmen getroffen werden. Wenn beispielsweise Namenskonventionen nicht eingehalten oder vorgeschriebene Parameter nicht mitgeführt werden, kann der Mechanismus eine Programmstelle nicht mehr richtig identifizieren und nicht die notwendigen Daten aus dem Kontext herauslesen. Auch wenn wir es oben so dargestellt haben, als ob beliebige Programmstellen identifizierbar wären, ist es in der Praxis so, dass schon beim Schreiben des Programms auf die Einhaltung der getroffenen Annahmen geach-

tet werden muss. Die Annahmen selbst können fast beliebig sein, aber wenn sie einmal getroffen wurden, ist es im Nachhinein nur sehr schwer möglich, sie abzuändern. Häufig werden die Annahmen in Form projektbezogener Regeln und Konventionen festgeschrieben. Beim Erstellen einer Klasse kommen uns manche vorgegebene Konventionen vielleicht lächerlich vor, weil wir den größeren Zusammenhang nicht sehen. Aber die Einhaltung ist ganz wesentlich, weil andernfalls der spätere Einsatz der aspektorientierten Programmierung verhindert wird oder dabei schwere Fehler auftreten.

In der aspektorientierten Programmierung wird eine eigene Terminologie verwendet. Allgemein verbreitet ist der Begriff *Separation-of-Concerns*. Das ist im Wesentlichen das Gleiche wie der Klassenzusammenhalt. Es wird ausgedrückt, dass unterschiedliche Belange durch unterschiedliche Klassen abgebildet sein sollen, damit nicht eine Klasse für mehrere, unzusammenhängende Belange verantwortlich ist. Im Zusammenhang mit der aspektorientierten Programmierung werden zwei grundsätzliche Arten von Belangen (Concerns) unterschieden: *Kernfunktionalitäten (Core-Concerns)* und *Querschnittsfunktionalitäten (Cross-Cutting-Concerns)*. Kernfunktionalitäten lassen sich bei der Faktorisierung gut und eindeutig bestimmten Klassen zuordnen und führen (wenn wir uns ausreichend darum bemühen) zu hohem Klassenzusammenhalt bei gleichzeitig schwacher Objektkopplung. Bei Querschnittsfunktionalitäten ist es dagegen unmöglich, hohen Klassenzusammenhalt und schwache Objektkopplung zu erzielen, weil solche Belange immer viele Klassen gleichzeitig betreffen. In obigem Beispiel zählt die Verwaltung von Konten, Geldtransfers und so weiter zu den Kernfunktionalitäten. Auch eine zentrale Stelle für die Überprüfung der Zugriffsrechte ist eine Kernfunktionalität. Aber es muss an sehr vielen Stellen eingegriffen werden um zu veranlassen, dass die zentrale Stelle die Überprüfungen der Zugriffsrechte durchführt. Das ist eine Querschnittsfunktionalität. Es ist von Natur aus unmöglich, die Verantwortung für die Veranlassung der Überprüfung der Zugriffsrechte an nur einer Programmstelle zu konzentrieren, es sind fast alle Klassen betroffen. Die übliche objektorientierte Programmierung kann mit Querschnittsfunktionalitäten schlecht umgehen. Aufrufe von Methoden für die Zugriffskontrolle sind über das gesamte Programm verteilt, wahrscheinlich wird an vielen Stellen auf sie vergessen. Die Verständlichkeit und Sicherheit leidet. Querschnittsfunktionalitäten lassen sich mittels aspektorientierter Programmierung besser integrieren. Auch dafür ist die Einhaltung projektspezifischer Regeln und Konventionen nötig, aber die Verständlichkeit und Sicherheit wird verbessert.

Die aspektorientierte Programmierung darf nicht als Alternative zur objektorientierten Programmierung gesehen werden. Vielmehr handelt es sich um eine Ergänzung (vor allem zur objektorientierten Programmierung, aber nicht darauf beschränkt), die in speziellen Situationen (für Querschnittsfunktionalität) einen wertvollen Beitrag liefern kann, in anderen Situationen (für Kernfunktionalität) aber unbrauchbar ist. Kernfunktionalitäten sind der Normalfall, Querschnittsfunktionalitäten eher selten. Neben der Überprüfung von Zugriffsrechten wird häufig das Generieren von Debug-Information oder Log-Dateien als Standardbeispiel für eine Querschnittsfunktionalität genannt.

4.6.2 AspectJ

AspectJ (`www.eclipse.org/aspectj`) ist ein Werkzeug für die aspektorientierte Programmierung in Java. Ausgangspunkt ist ein Java-Programm, das aus beliebig vielen Klassen bestehen kann. Zusätzlich schreiben wir Dateien, üblicherweise mit der Endung `.aj`, in denen alle gewünschten Änderungen der Semantik von Java festgehalten sind. Statt `javac` verwenden wir zur Übersetzung `ajc` (AspectJ-Compiler). Alle Klassen des Java-Programms werden zusammen mit den `.aj`-Dateien und der Bibliothek `aspectjrt.jar` gleichzeitig übersetzt. `ajc` fungiert als Aspect-Weaver, der die Inhalte der `.aj`-Dateien zur Modifikation der Java-Dateien einsetzt (Precompilation), die danach automatisch weiter zu `.class`-Dateien übersetzt werden. Die Bibliothek `aspectjrt.jar` stellt Hilfsfunktionalität dafür bereit, die in manchen Fällen zur Laufzeit auch Konzepte der Metaprogrammierung einsetzt. Das ursprüngliche Java-Programm wird nicht zerstört; um die Änderungen durch den Aspect-Weaver wieder los zu werden, müssen wir das Programm nur wieder wie üblich mittels `javac` übersetzen.

AspectJ baut auf folgende Begriffe auf:

Join-Point: Das ist eine *zur Laufzeit identifizierbare Stelle* in einem Programm, z. B. der Aufruf einer Methode oder der Zugriff auf ein Objekt. Es handelt sich also nicht um eine Stelle, die direkt im Programmtext steht (etwa der Programmtext für den Aufruf einer Methode), sondern um eine einzelne Ausführung einer solchen Programmstelle. Zur Laufzeit ist der Kontext eines Join-Points (etwa die aktuellen Parameterwerte der aufgerufenen Methode oder die Identität des zugegriffenen Objekts) schon bekannt.

Pointcut Das ist ein syntaktisches Element (also Programmtext) in einer `.aj`-Datei, das einen Join-Point (bzw. eine Menge gleichartiger Join-Points) auswählt und kontextabhängige Information dazu sammelt, z. B. die Argumente eines Methodenaufrufs oder eine Referenz auf das Zielobjekt.

Advice Das ist ist ein syntaktisches Element in einer `.aj`-Datei, das den Programmtext spezifiziert, der an einem Join-Point (zusätzlich) auszuführen ist. Je nach Art des Advices wird der Programmtext zusätzlich vor (`before()`), zusätzlich nach (`after()`) oder anstatt (`around()`) dem Join-Point ausgeführt, wobei im Fall von `around()` der Join-Point häufig irgendwo innerhalb dieses Programmtexts explizit ausgeführt wird.

Aspect Das ist ist ein zentrales syntaktisches Element in einer `.aj`-Datei, das alle Teile zu einer Einheit zusammenführt (auf ähnliche Weise, wie eine Klasse in Java alle Teile zusammenführt). Ein Aspekt enthält Deklarationen von Variablen und Definitionen von Methoden (wie eine Java-Klasse) sowie Pointcuts und Advices.

Ein Pointcut definiert eine Menge von Join-Points. Obwohl es auch anonyme Pointcuts gibt, werden meist benannte Pointcuts verwendet. Die Syntax sieht folgendermaßen aus, wobei *Signatur* die syntaktische Darstellung eines Join-Points als Java-Text ist, die explizit gekennzeichnete Lücken enthalten kann:

[*Sichtbarkeit*] `pointcut` *Name* `(`[*Argumente*]`) :` *Pointcuttyp*`(`*Signatur*`);`

Das folgende Beispiel spezifiziert einen Pointcut für einen Methodenaufruf, der alle Methoden umfasst, die mit beliebiger Sichtbarkeit im Paket `javax` oder Unterpaketen davon vorkommen, deren Namen mit `add` beginnen, mit `Listener` enden und deren einzige Argumente Untertypen von EventListener sind:

```
public pointcut AddListener() :
   call(* javax..*.add*Listener(EventListener+));
```

In der Signatur steht

- `*` für eine beliebige Anzahl von Zeichen außer einem `.`,
- `..` für eine beliebige Anzahl jedes beliebigen Zeichens,
- `+` für jeden Untertyp eines Typs,

Im Beispiel steht `call` für den Typ eines Pointcuts für einen Methodenaufruf. Es gibt viele Arten von Pointcuttypen (Aufzählung unvollständig):

`execution(`*MethodSignature***`)`** – Ausführung einer Methode

`call(`*MethodSignature***`)`** – Aufruf einer Methode

`execution(`*ConstructorSignature***`)`** – Ausführung eines Konstruktors

`call(`*ConstructorSignature***`)`** – Aufruf eines Konstruktors

`get(`*FieldSignature***`)`** – lesender Zugriff auf Objekt- oder Klassenvariable

`set(`*FieldSignature***`)`** – schreibender Zugriff auf Objekt- oder Klassenvariable

`staticinitialization(`*TypeSignature***`)`** – Initialisierung einer Klasse

`preinitialization(`*ConstructorSignature***`)`** – erster Schritt der Initialisierung eines Objekts

`initialization(`*ConstructorSignature***`)`** – Initialisierung eines Objekts

`handler(`*TypeSignature***`)`** – Ausführung einer Ausnahmenbehandlung

`this(`*TypeOrId***`)`** – das aktuelle Objekt (innerhalb einer Objekt-Methode oder eines Konstruktors) ist vom gegebenen Typ oder identisch zum angegebenen Wert

`target(`*TypeOrId***`)`** – der Empfänger einer Nachricht oder das Ziel eines Variablenzugriffs ist ein Objekt vom gegebenen Typ oder identisch zum angegebenen Wert

`args(`*TypeOrId, ...***`)`** – die Argumente in der Argumentliste sind von den gegebenen Typen oder identisch zu den angegebenen Werten

Zum Beispiel werden alle schreibenden Zugriffe auf eine private Variable vom Typ `float` der Klasse `Account` mit dem Namen `balance` durch den Pointcut `set(private float Account.balance)` spezifiziert. Das ist ein anonymer Pointcut (enthält nur den Teil nach dem Doppelpunkt). Solche (anonyme) Pointcuts können über folgende Operatoren miteinander verknüpft werden, wodurch nach dem Doppelpunkt in einem benannten Pointcut auch mehrere, über Operatoren verknüpfte Pointcuts stehen können:

- `!` (Negation) für alle Join-Points außer dem spezifizierten,
- `||` für die Vereinigungsmenge von Join-Points und
- `&&` für die Durchschnittsmenge von Join-Points.

Beispielsweise ist `SpecificAddListener` die Durchschnittsmenge zweier Arten von Join-Points:

```
public pointcut SpecificAddListener(EventListener el) :
   call(* javax..*.add*Listener(EventListener)) && args(el);
```

Hier kommt ein Argument ins Spiel. Ausgewählt werden nur jene Join-Points mit entsprechender Syntax, bei denen das übergebene Argument identisch zu `el` ist. Normalerweise ist `el` allerdings nicht spezifiziert; dadurch werden alle Join-Points passender Syntax gewählt und der Wert von `el` im Join-Point wird zurückgegeben. Bei Anwendung der Pointcuts (siehe unten) wird der Wert von `el` weitergeleitet. Auf diese Weise lässt sich Information aus dem Kontext des Join-Points ermitteln. So wie dies `args(...)` für Argumente macht, geschieht das auch bei `target(...)` für Empfänger von Nachrichten oder Objekten, in denen auf Variablen zugegriffen wird, sowie bei `this(...)`[4] für Objekte, in denen Methodenaufrufe oder Variablenzugriffe erfolgen.

Folgende Arten von Pointcuttypen sind kontrollflussbasiert, das heißt, sie betreffen alle ausgeführten Join-Points im angegebenen Bereich:

`cflow(`*Pointcut*`)` alle dem Kontrollfluss entsprechenden Join-Points eines Pointcuts inklusive dem äußersten; bezieht sich *Pointcut* z. B. auf einen Methodenaufruf, dann alle während der Ausführung der Methode ausgeführten Anweisungen inklusive dem Methodenaufruf

`cflowbelow(`*Pointcut*`)` wie `cflow`, aber exklusive dem äußersten Join-Point (z. B. exklusive Methodenaufruf)

Mit einem solchen Pointcut werden sehr viele Join-Points gewählt. Häufig wird diese Menge durch Bildung der Durchschnittsmenge mit einem anderen Pointcut reduziert. Das gilt auch für folgende Arten von Pointcuttypen, die sichtbarkeitsbasiert sind:

`within(`*Typepattern*`)` alle Join-Points innerhalb des lexikalischen Sichtbereichs einer Klasse oder eines Aspekts

[4] `this(...)` hat die gleiche Syntax wie ein Konstruktoraufruf als erste Anweisung in einem Konstruktor. Da Pointcuts nicht in Konstruktoren stehen, ist die Syntax aber eindeutig.

withincode(*Method/ConstructorSignature***)** alle Join-Points im lexikalischen Sichtbereich der Methode oder des Konstruktors

Der folgende anonyme Pointcut schließt alle Pointcuts innerhalb der Klasse oder des Aspekts `TraceAspect` von der spezifizierten Menge der Pointcuts (alle Aufrufe von Methoden aus `PrintStream`, die mit `print` beginnen) aus.

```
call(* java.io.PrintStream.print*(..)) &&
        !within(TraceAspect)
```

In einem *Advice* wird angegeben, welche Anweisungen an den ausgewählten Join-Points ausgeführt werden sollen. Ein Advice hat die Form

[*Sichtbarkeit*] `before(`[*Argumente*]`) :` *Pointcut* {*Programmtext*}

In diesem Fall wird der Programmtext vor jedem durch den Pointcut bestimmten Join-Point ausgeführt. Mit `after()` statt `before()` wird der Programmcode nach dem Pointcut ausgeführt. Es gibt die Variante `after() returning`, die nur zutrifft, wenn der Join-Point ohne Auftreten einer Ausnahme beendet wird, sowie `after() throwing`, die nur im Fall einer ausgelösten Ausnahme zutrifft. Mit `around()` wird Programmcode statt dem Pointcut ausgeführt, wobei der Code des Join-Points komplett umgangen oder (z. B. mit anderen Argumenten) irgendwo zwischen drin ausgeführt wird.

Im folgenden Programmstück wird ein Advice mit einem anonymen Pointcut und einer mit einem Namen versehenen Pointcut gezeigt:

```
before() : call(* Account.*(..)) { checkUser(); }

pointcut connectionOperation(Connection connection) :
    call(* Connection.*(..) throws SQLException)
    && target(connection);
before(Connection connection) :
    connectionOperation(connection) {
        System.out.println("Operation auf " + connection);
    }
```

Der Advice `{ checkUser(); }` wird vor jedem Aufruf einer Methode mit beliebigem Ergebnistyp und beliebiger Signatur der Klasse `Account` ausgeführt. Der zweite Advice zeigt das Weiterreichen von Information über ein Argument eines Pointcuts und Advices: `connection` ist das Objekt, auf dem die Methode ausgeführt wird; `connectionOperation` beschreibt alle Aufrufe von Methoden der Klasse `Connection`, die eine `SQLException` auslösen können. Wie wir sehen, wird Information über Parameter hier auf andere Weise weitergereicht als in Java: Der Join-Point bestimmt den Wert von `connection`, also den Empfänger der (unbekannten) Nachricht. Nach Weiterreichung über den Pointcut und Advice wird der Wert in der `println`-Anweisung verwendet.

Ein *Aspekt* fasst Pointcuts und Advices sowie ganz normale Variablendeklarationen und Methodendefinitionen, wie sie in jeder Java-Klasse vorkommen

könnten, zu einer Einheit zusammen. Die Variablen und Methoden werden vor allem in den Advices verwendet. Die Definition sieht wie die einer Klasse aus, bis auf `aspect` statt `class`. Sie steht meist in einer Datei mit der Endung `.aj`, obwohl eine Datei mit der Endung `.java` auch funktioniert.

```
public aspect JoinPointTraceAspect {
   private int callDepth = -1;

   pointcut tracePoints(): !within(JoinPointTraceAspect);

   before() : tracePoints() {
      callDepth++;
      print("Before", thisJoinPoint);
   }

   after() : tracePoints() {
      callDepth--;
      print("After", thisJoinPoint);
   }

   private void print(String prefix, Object message) {
      for(int i=0, spaces=callDepth*2; i<spaces; i++) {
         System.out.print(" ");
      }
      System.out.println(prefix + ": " + message);
   }
}
```

Der Aspekt `JoinPointTraceAspect` gibt alle Join-Points eines Programms aus. Die Ausgabe wird der Schachtelungstiefe entsprechend eingerückt. In einem `before`-Advice wird die Einrücktiefe erhöht, in einem `after`-Advice wieder erniedrigt und die Art des Join-Points ausgegeben. Dabei ist `thisJoinPoint` ein Zeiger auf das Join-Point-Objekt mit allen Informationen über einen Join-Point; der Wert dieser Pseudovariable wird automatisch gesetzt. Am besten gleich an einem eigenen Java-Programm ausprobieren. Achtung: Es wird viel Output produziert, daher nur an einem kleinen, kurz laufenden Programm ausprobieren.

Hier wurde nur ein Bruchteil der Sprachelemente von AspectJ vorgestellt. Weitergehende Informationen und der Compiler `ajc` finden sich auf der Homepage von AspectJ (`eclipse.org/aspectj`) und in der Literatur (z. B. *AspectJ in Action* von Ramnivas Laddad [21]).

5 Applikative Programmierung und Parallelausführung

Unterstützung für die applikative Programmierung als eine fortgeschrittene Form der funktionalen Programmierung wurde vergleichsweise spät in Java integriert, später als in die meisten anderen objektorientierten Sprachen. Dennoch (oder vielleicht gerade deswegen) hat sich ein entsprechender Programmierstil in sehr kurzer Zeit durchgesetzt. Ein bedeutender Auslöser für den Wunsch nach Integration funktionaler und applikativer Programmiertechniken in objektorientierte Sprachen liegt darin, dass sich die funktionale und applikative Programmierung als recht erfolgversprechende Basis für nebenläufige und parallele Programmierung erwiesen hat. Die rasch zunehmende und breite Verfügbarkeit paralleler Recheneinheiten und der große Bedarf nach Big-Data-Anwendungen lassen eine solche Entwicklung logisch erscheinen, auch wenn dadurch Programmierparadigmen miteinander kombiniert werden, die von Natur aus in Widerspruch zueinander stehen. In diesem Kapitel geht es also nicht nur darum, welche Mechanismen Java für die funktionale und applikative Programmierung zur Verfügung stellt und wie diese zusammen mit speziell für Nebenläufigkeit entwickelten Mechanismen für die nebenläufige und parallele Programmierung einsetzbar sind, sondern auch darum, wie wir mit den Widersprüchen in den Paradigmen umgehen können.

5.1 Lambdas und Java-8-Streams

Ein Beispiel mit Lambdas und Java-8-Streams haben wir in Abschnitt 2.3.2 gesehen. Nun betrachten wir diese Konzepte aus einem anderen Blickwinkel. Zunächst betrachten wir Lambdas zusammen mit anonymen inneren Klassen, damit deren Leistungsfähigkeit und Einbettung in Java besser verständlich wird. Danach beschäftigen wir uns mit den Grundkonzepten der Java-8-Streams und Richtlinien für deren Einsatz in der applikativen Java-Programmierung.

5.1.1 Anonyme innere Klassen und Lambdas

Lambdas ähneln Objekten innerer Klassen, siehe Abschnitt 3.4.1. Eine *anonyme innere Klasse* ist eine innere Klasse ohne einen von uns vorgegebenen Namen. Sie kombiniert die Syntax zur Objekterzeugung mit jener zur Definition einer Klasse: Nach `new`, dem Namen eines Typs (der auch eine abstrakte Klasse oder ein Interface sein kann) sowie runden Klammern steht ein Klassen-Rumpf. In folgendem Beispiel enthält eine `return`-Anweisung eine anonyme innere Klasse:

```
public class List<A> implements Collection<A> {
    private Node<A> head = ...;  // as in Section 4.1.2
    public Iterator<A> iterator() {
         return new Iterator<A>() {
             private Node<A> p = head;
             public boolean hasNext() { return p != null; }
             public boolean next() { ... }
         }
    }
   ...
}
```

Der Klassenrumpf wird wie jede andere innere Klasse (die auch an fast beliebigen Programmstellen definiert werden kann) vom Compiler übersetzt, wobei der Compiler für die Klasse einen internen Namen wählt, der sonst nirgends vorkommt (z. B. `List$1`). Über Reflexion kann dieser Name im Programm sichtbar werden. Der Typ nach `new` ist ein Obertyp der Klasse; falls dieser Obertyp eine (abstrakte) Klasse ist, können die runden Klammern Argumente enthalten, die wie durch eine `super`-Anweisung in einem Konstruktor an den Konstruktor der Oberklasse übergeben werden. Bei Ausführung von `new` wird ein Objekt der anonymen inneren Klasse erstellt. Das Ergebnis ist vom deklarierten Obertyp, im Beispiel also `Iterator<A>`. Anonyme innere Klassen erweitern die Fähigkeiten von Java gegenüber normalen inneren Klassen in keiner Weise. Sie stellen nur eine Syntaxvereinfachung für den häufig vorkommenden Fall dar, dass Objekte einer inneren Klasse nur an genau einer Stelle im Programm erzeugt werden. Obiges Beispiel entspricht dem in Abschnitt 4.1.2, abgesehen vom nicht eingeführten Namen der dortigen Klasse `ListIter`.

Wir machen nicht viel falsch, wenn wir Lambdas als weitere Syntaxvereinfachung anonymer innerer Klassen betrachten, wobei jede soche Klasse nur genau eine Methode definiert, sonst nichts. Da der angegebene Obertyp die Signatur genau einer entsprechenden Methode spezifizieren muss, können wir den Methodennamen, den Ergebnistyp und die Typen der Parameter weglassen; diese müssen mit der Signatur übereinstimmen.[1] Ein solcher Obertyp muss immer existieren, wo ein Lambda verwendet wird, aber er muss nicht immer auf den ersten Blick als solcher erkennbar sein. Häufig werden Lambdas als Argumente an Methoden übergeben; dann entsprechen die Obertypen den Parametertypen. In anderen Fällen werden Lambdas in Variablen abgelegt; dann entsprechen Obertypen den deklarierten Variablentypen. Solche Obertypen müssen immer Interfaces sein, weil es keine Klasse mit nur einer Methode geben kann (da mehrere Methoden von `Object` geerbt werden). Konkret muss der Obertyp ein *funktionales Interface* mit genau einer abstrakten Methode sein, die vom Lambda implementiert wird; daneben kann es beliebig viele Methoden mit Default-

[1] Hinsichtlich Subtyping könnte der Ergebnistyp im Untertyp kovariant verändert, also spezieller sein als in der Signatur. Das würde aber keinen Sinn ergeben. Generell ist die Forderung nach einer übereinstimmenden Signatur ein technischer Kunstgriff, der Typinferenz ermöglicht. Wäre der Obertyp beliebig, würde Subtyping Typinferenz verhindern.

Implementierungen, statische Methoden und Konstantendefinitionen enthalten. Eine weitere Einschränkung bei Lambdas (im Gegensatz zu abstakten inneren Klassen) besteht darin, dass im Rumpf nur unveränderliche Variablen aus der Umgebung zugreifbar sind, das sind solche, die als `final` deklariert sind oder so verwendet werden, als ob sie als `final` deklariert wären. Das soll die gefürchtete unkontrollierte Kommunikation über Variablen im Zaum halten. Parameter der Lambdas oder innerhalb des Rumpfs von Lambdas deklarierte lokale Variablen sind dagegen uneingeschränkt änderbar.

Wenn wir schon beim Vereinfachen der Syntax sind, ist leicht zu verstehen, dass auch die geschwungenen Klammern um einen Methoden-Rumpf mit nur einer Anweisung und das Schlüsselwort `return` weggelassen werden kann, falls der Rumpf nur aus einer `return`-Anweisung besteht. Das Weglassen von runden Klammern um einen einzigen Parameter ist nur mehr eine Kleinigkeit. Schließlich müssen wir noch das Symbol `->` zwischen Parameterliste und Rumpf zur Kennzeichnung von Lambdas einführen, damit die Syntax eindeutig wird. Dieses Symbol bietet sich an, weil es aufgrund der C-Syntax von Java ohnehin reserviert war und in neueren funktionalen Sprachen üblicherweise zur Kennzeichnung von Funktionstypen dient, also eine inhaltliche Nähe besteht.

Um Beispiele zu betrachten, brauchen wir passende Interfaces als Obertypen. Wir könnten dafür eigene Interfaces einführen. Einfacher geht es, wenn wir Interfaces verwenden, die in den Java-Standard-Bibliotheken für diesen Einsatzzweck vordefiniert sind. Viele davon sind im Paket `java.util.function` zusammengefasst. Dort gibt es das generische Interface `Function<T,R>` mit der abstrakten Methode `R apply(T t)`, wobei `T` für den Typ des einzigen Parameters und `R` für den Ergebnistyp steht. Das Interface `BiFunction<T,U,R>` mit der abstrakten Methode `R apply(T t, U u)` ist ähnlich, aber mit einem zweiten Parameter vom Typ `U`. Das Interface `Consumer<T>` hat die Methode `void accept(T t)` ohne Ergebnis. Keine dieser Methoden ist mit Zusicherungen versehen, die besagen, wofür sie stehen. Außer den Signaturen ist nichts über die Methoden bekannt; wir haben es (trotz benannter Interfaces) im Wesentlichen mit struktureller Abstraktion zu tun, nicht mit der üblichen nominalen Abstraktion. Jedes Interface in diesem Paket ist mit der Annotation `@FunctionalInterface` versehen, die den Compiler anweist, eine Fehlermeldung auszugeben, falls es sich nicht um ein Interface mit genau einer abstrakten Methode handelt. Das ist eine reine Vorsichtsmaßnahme, die, abgesehen von einer möglichen Fehlermeldung, keinerlei Konsequenzen hat. Viele dieser Interfaces enthalten auch Methoden mit Default-Implementierungen und statische Methoden, die im entsprechenden Kontext hilfreich sein könnten.

Hier sind einige einfache Beispiele für Lambdas und ihre Ausführungen:

```
Consumer<String> p = s -> System.out.println(s);
p.accept("Hello world.");
Function<Integer,String> value = i -> "value = " + i;
p.accept(value.apply(8));
BiFunction<String,Boolean,String> opt = (s,b) -> b ? s : "";
p.accept(opt.apply("maybe", true));
```

Ein Blick lässt erkennen, dass Lambdas zwar kurz, kompakt und intuitiv sein können und wie Funktionsdefinitionen aussehen, die Kürze aber nur durch umfangreiche deklarierte Typen und die Informationen in Interfaces ermöglicht wird. Die Verwendungen der Lambdas machen deutlich, dass es sich dabei nicht um Funktionen handelt, sondern um Objekte, die Methoden enthalten. In folgender Fortsetzung obiger Programmzeilen wird die Variable `value` geändert:

```
value = i -> { String r = "value = ";
               r += i;
               p.accept(r);
               return r;
             };
value.apply(6);
```

Während Methoden zur Laufzeit unveränderlich sind, lassen sich Variablen, die Lambdas enthalten, sehr einfach ändern. Darin liegt ein Vorteil gegenüber Methoden. Die Anweisung `p.accept(r);` greift auf die Variable `p` aus der Umgebung des Lambdas zu. Das geht nur, weil `p` unveränderlich ist. Der Compiler meldet einen Fehler, wenn wir versuchen, einen neuen Wert an `p` zuzuweisen. Die lokale Variable `r` kann dagegen beliebig geändert werden. Lambdas können, wie alle Methoden in Java, uneingeschränkt Seiteneffekte haben.

Tatsächlich sind Lambdas keine inneren Klassen, sondern eigenständige Konstrukte, für deren Einführung die JVM erstmals in der Geschichte erweitert wurde. Der Hauptgrund dafür liegt in der unzureichenden Effizienz im Umgang mit einer großen Zahl sehr kleiner Klassen. Ein Großteil der Komplexität im Umgang mit Klassen ist für Lambdas unnötig. Die Semantik von Lambdas orientiert sich stark an der von anonymen inneren Klassen, sodass es kein Fehler ist, Lambdas als Spezialfall anonymer innerer Klassen anzusehen, wobei die Einschränkungen die gröbsten Probleme geschachtelter Klassen beseitigen.

In Abschnitt 1.1.2 haben wir gesehen, dass der untypisierte λ-Kalkül die Mächtigkeit einer Turing-Maschine mit sehr einfachen Mitteln erreicht. Viel mehr als λ-Abstraktion ist dazu nicht nötig. Es stellt sich die Frage, ob Lambdas in Java auch so mächtig sind. Eine gute Antwort darauf ist vielschichtig und komplex, da es einen fundamentalen Unterschied gibt: Alle Parameter und Ergebnisse von Lambdas in Java haben einen deklarierten Typ. Eine einfach typisierte Variante des λ-Kalküls, die große Ähnlichkeit zu Java-Lambdas hat, erreicht nicht mehr die Mächtigkeit der Turing-Maschine, macht Programme dafür aber einfacher verständlich. Der Grund liegt darin, dass wir keine unendlich großen Typen aufbauen können, die wir bräuchten, um mit den Mitteln des λ-Kalküls Rekursion darzustellen. Das ist nicht tragisch, weil wir die Ursache des Problems kennen. Einer typisierten Variante des λ-Kalküls können wir wieder die Mächtigkeit der Turing-Maschine verleihen, indem wir eine weitere Regel hinzufügen, die rekursive Aufrufe ermöglicht[2] (natürlich nur auf Kosten der Einfachheit). Bei Lambdas in Java ist es ähnlich: Es spielt keine Rolle, wie

[2] Genau genommen handelt es sich um eine mit einem Typ parametrisierte, also generische Regel, was äquivalent zu einer Familie von Regeln ist, häufig *Y-Kombinator* genannt.

mächtig die Lambdas für sich genommen sind, weil die Mächtigkeit der Sprache von anderen Sprachelementen wie rekursiven Methodenaufrufen oder Schleifen bestimmt wird. Natürlich können wir in Java Interpreter für beliebige Varianten des λ-Kalküls implementieren. Für die Mächtigkeit der Sprache ist es kein Nachteil, dass Lambdas nur Objekte stark eingeschränkter Klassen sind. Die Einschränkungen wurden so gewählt, dass Lambdas in der Praxis viel einfacher handhabbar sind als vollständige Klassen. Erst die Vereinfachung ermöglicht Abstraktionen auf sehr hoher Ebene, wie wir sie im applikativen Programmierstil benötigen.

Es gibt eine weitere syntaktische Variante zur Spezifikation von Lambdas: `Klassenname::Methodenname` steht für eine Methode mit dem genannten Namen in der genannten Klasse (oder für die Erzeugung eines Objekts der Klasse, wenn statt dem Methodennamen `new` verwendet wird). Genau wie bei Lambdas, die mittels `->` spezifiziert werden, muss ein Obertyp (funktionales Interface) gegeben sein, der für die passende Auswahl sorgt, wenn Methoden und Konstruktoren überladen sind oder Typparameter benötigt werden. Es hängt von der Art (Objektmethode, Klassenmethode oder Konstruktor) ab, wofür ein derartiges Lambda genau steht, wie folgende Beispiele zeigen:

```
BiFunction<String,String,Integer> cmp = String::compareTo;
             // entspricht cmp = (s,t) -> s.compareTo(t);
BiFunction<Object,Object,Boolean> eq = Objects::equals;
             // entspricht eq = (a,b) -> Objects.equals(a,b);
Function<StringBuilder,String> mk = String::new;
             // entspricht mk = sb -> new String(sb);
```

Im Fall einer Objektmethode erhöht sich die Anzahl der Parameter um eins, weil als erster Parameter der Empfänger der Nachricht dazukommt. Der einzige Grund für die Existenz dieser Form von Lambdas ist die gute Lesbarkeit.

5.1.2 Java-8-Streams

Der Begriff „Java-8-Stream“ klingt nach einem sehr speziellen Konzept in einer ganz bestimmten Java-Version und ist das auch. Es fehlt uns ein allgemeinerer Begriff. Am ehesten können wir das Konzept als Form eines Iterators betrachten.

In der objekorientierten Programmierung häufig eingesetzte *externe* Iteratoren (Objekte von `Iterator` mit den Methoden `next` und `hasNext`) iterieren mittels einer außerhalb des Iterators gelegenen Schleife über die Elemente und verändern den Zustand des Iterators dabei wiederholt durch Aufrufe von `next`. Iteratoren verlagern die Kontrolle von den Stellen, an denen Iteratoren erzeugt werden, an die Stellen, an denen `next` aufgerufen wird. Diese imperative Vorgehensweise ist aufgrund der Seiteneffekte nicht mit einem funktionalen Programmierstil vereinbar. Funktionale und applikative Programme, insbesondere in neueren funktionalen Sprachen wie Haskell, setzen *interne* Iteratoren ein, bei denen innerhalb des Iterators mittels Rekursion über die Elemente iteriert wird. Ein interner Iterator ist eine Funktion höherer Ordnung (eine Funktion, die eine weitere Funktion als Parameter nimmt), die die übergebene Funktion

erst bei Bedarf auf jedes Element anwendet. In erster Näherung erfüllen externe und interne Iteratoren den gleichen Zweck, aber externe Iteratoren sind besser kontrollierbar, interne semantisch einfacher und nicht auf Seiteneffekte angewiesen. Durch Lazy-Evaluation kann der Unterschied zwischen internen und externen Iteratoren sehr klein sein oder verschwinden. Java-8-Streams betten für Haskell-Programme typische Abläufe einschließlich Lazy-Evaluation zusammen mit Iteratoren (die wie interne Iteratoren ausschauen, aber wie externe Iteratoren ausgeführt werden) in Java ein. Trotz des Namens sind Java-8-Streams also kein Java-typisches Konzept. Allerdings mussten für die Einbettung zahlreiche Erweiterungen in vielen Java-Standard-Klassen vorgenommen und das Konzept an die Gegebenheiten einer objektorientierten Sprache angepasst werden, sodass der Begriff doch nicht falsch gewählt ist.

Wie in Abschnitt 2.3.2 erläutert, sind Java-8-Streams Objekte der Klassen `Stream<T>` (Datenstrom mit Elementen des generischen Typs `T`), `IntStream`, `LongStream` und `DoubleStream` (Datenströme mit Elementen der entsprechenden elementaren Typen), die jeweils als sequentielle oder parallele Datenströme verwendbar sind. Im Mittelpunkt stehen die zahlreichen Methoden, die auf den Datenströmen operieren. Wir unterscheiden drei Arten solcher Methoden:

Stream-erzeugende Operationen: Das sind Methoden, die einen neuen Datenstrom erzeugen und die Elemente, über die iteriert werden soll, in den Datenstrom füttern. Standard-Klassen, die das Interface `Iterable<T>` implementieren (also iterierbar sind) unterstützen auch die Methoden `stream()` und `parallelStream()`, die jeweils einen (sequentiellen oder parallelen) neuen Datenstrom mit den iterierbaren Elementen erzeugen. Das sind vor allem Klassen aus dem Collections-Framework. Die Stream-Klassen selbst bieten statische Methoden zum Erzeugen neuer Streams an, vor allem `to` aufgerufen mit den Elementen, über die iteriert werden soll, oder einem Array dieser Elemente. Die Methoden `iterate` und `generate` zum Erzeugen unendlich vieler Elemente in einem Stream (mit etwas unterschiedlichen Techniken) sind Funktionen höherer Ordnung, übernehmen also Lambdas als Argumente, die die Elemente produzieren. Die Klasse `StreamSupport` stellt statische Methoden bereit, um neue Streams aus *Spliteratoren* zu erzeugen. Ein Objekt vom Typ `Spliterator<T>` ist eine erweiterte Form eines (sowohl internen als auch externen) Iterators, der Unterstützung für das Aufteilen der Elemente auf mehrere Datenblöcke (das sind selbst wieder Spliteratoren) hat, sodass parallele Ströme unabhängig voneinander auf unterschiedlichen Datenblöcken arbeiten können. Das Interface `Iterable<T>` hat die Methode `spliterator()` mit einer Default-Implementierung (die meist zu überschreiben ist), die die Elemente, über die iteriert wird, in ein Objekt vom Typ `Spliterator<T>` füttert, wodurch Spliteratoren leicht verfügbar sind.

Stream-modifizierende Operationen: Das sind von den Stream-Klassen bereitgestellte Objekt-Methoden, die Operationen auf den Elementen des Streams ausführen und Ergebnisse wieder in einen Stream füttern. Ergebnisse dieser Methoden sind von einem Stream-Typ. Viele dieser Methoden

dienen als Funktionen höherer Ordnung, denen Lambdas als Parameter übergeben werden. Beispielsweise gibt es die Methode `map`, die eine Operation (als Lambda frei wählbar) auf jedes Element anwendet und die Ergebnisse weiterreicht, wobei Element-Typ und Ergebnistyp verschieden sein können; der Strom an Daten nach der Anwendung von `map` kann einen anderen Typ haben als davor.[3] Methoden wie `filter` belassen einige Elemente im Strom und filtern andere heraus. Einige Methoden kümmern sich um Spezialfälle, z. B. limitiert `limit` die Anzahl der Elemente, sorgt `sorted` für eine sortierte Reihenfolge und vermeidet `distinct` Duplikate.

Stream-abschließende Operationen: So wie modifizierende Operationen sind auch abschließende Operationen Objekt-Methoden, die von den Stream-Klassen bereitgestellt und auf Elementen des Streams ausgeführt werden. Ergebnisse werden jedoch nicht mehr in einen Stream gefüttert, sondern der Stream wird abgeschlossen und Elemente werden auf andere Weise (außerhalb des Streams) weiterverarbeitet. Ergebnisse sind nicht von einem Stream-Typ, außer in Sonderfällen, z. B. wenn Stream-Elemente wieder Streams sind, wobei es sich um andere Streams handelt. Häufig verwenden wir `reduce` (in verschiedenen Varianten), um die einzelnen Elemente im Stream durch Anwendung von Lambdas zu einem einzigen Wert zusammenzufassen, etwa eine Zahlensumme zu bilden. Ebenso häufig dient `collect` (in verschiedenen Varianten) dazu, Elemente im Stream in irgendeiner Art von Collection abzulegen, was konzeptuell mit `reduce` eng verwandt ist. Die vielleicht allgemeinste Form des Abschlusses ist `forEach`, eine Methode, die irgendeine Aktion auf jedem Element ausführt (z. B. Ausgabe oder Abspeichern in einer Collection bzw. Addieren zu einer Summe). Spezielle abschließende Operationen sind z. B. `allMatch` und `anyMatch`, die ein Boolean zurückgeben, das besagt, ob alle Elemente oder irgendein Element eine bestimmte (über ein Lambda festgelegte) Eigenschaft erfüllt, sowie `count`, das einfach nur die Elemente zählt.

Die Ausführung der Stream-Operationen erfolgt mittels Lazy-Evaluation, wie in Abschnitt 2.3.3 demonstriert. Hinter jeder Operation, die einen Stream erzeugt oder modifiziert, steht ein Iterator und die Ausführung erfolgt in mehreren Schritten: Bei Ausführung der Stream-erzeugenden und Stream-modifizierenden Methoden in einem ersten Schritt passiert noch keine inhaltliche Berechnung, sondern es werden nur die dahinter stehenden Iteratoren erzeugt und miteinander verknüpft. Erst die Ausführung einer Stream-abschließenden Operation stößt die eigentlichen Berechnungen an. Auf fast die gleiche Weise, wie wir bei Anwendung eines externen Iterators durch Aufrufe von `next` wiederholt auf das

[3] Hier ergibt sich ein terminologisches Problem: Sollen wir von nur einem Datenstrom sprechen, in dem die Daten während des Durchfließens transformiert werden, oder ist der Datenstrom vor der Ausführung von `map` ein anderer als danach? Formal betrachtet und aus der Implementierungssicht müssen wir jedenfalls von unterschiedlichen Strömen sprechen, weil nicht einmal die Typen übereinstimmen müssen. Aber der Ablauf wird einfacher verständlich, wenn wir nur einen Strom sich ändernder Daten im Kopf haben.

jeweils nächste Element zugreifen, holt sich eine abschließende Operation wiederholt das nächste Element aus dem davor stehenden Iterator, solange noch weitere Elemente benötigt werden und der Stream (das ist der Iterator) noch weitere Elemente liefern kann. Zunächst erfolgt der Aufruf in dem Iterator, der der Operation direkt vor der abschließenden Operation entspricht, der leitet den Aufruf gegebenenfalls an den Iterator weiter, der der davor stehenden Operation entspricht und so weiter, bis zur erzeugenden Operation. Die Ergebnisse werden jeweils nach Ausführung der modifizierenden Operationen zurückgegeben. Hinter Lazy-Evaluation steckt also keine Hexerei und auch kein undurchschaubar komplizierter Mechanismus, sondern das ist das ganz normale Programmverhalten, das wir bekommen, wenn wir mehrere Iteratoren hintereinanderschalten.

Die Iteratoren, die hinter den Stream-Operationen stecken, sind vom Typ `Spliterator<T>`, den wir oben schon im Zusammenhang mit der Erzeugung von Streams gesehen haben. Wir können die Fähigkeiten von Streams selbst erweitern, indem wir neue Spliteratoren schreiben (das Interface `Spliterator` implementieren). Über die Klasse `StreamSupport` werden Spliteratoren in Streams eingebunden. Wir erhalten einen modifizierenden Operator, wenn unser Spliterator über Elemente iteriert, die zuvor aus einem anderen Spliterator gelesen wurden; andernfalls erhalten wir einen erzeugenden Operator. Jede Methode, die Elemente aus einem Spliterator liest, kann als abschließende Operation verstanden werden. Die Einteilung der Methoden in die drei Kategorien (erzeugen, modifizieren, abschließen) ergibt sich ganz natürlich, ebenso wie Lazy-Evaluation. Wir müssen jedoch bedenken, dass Spliteratoren nur aus der Sicht der Implementierung existieren. Beim Programmieren mit Streams bleiben Spliteratoren meist versteckt; die Abstraktion über Iteratoren ist ja ein wesentlicher Grund, warum wir Java-8-Streams überhaupt verwenden. Das Verhalten bezüglich Lazy-Evaluation wird nur aus der klaren Unterscheidung zwischen erzeugenden, modifizierenden und abschließenden Operationen ablesbar. Beschreibungen der Methoden müssen in dieser Hinsicht deutlich sein.

Es folgt ein einfaches Beispiel für die Faktorielle-Berechnung. Wir gehen bei allen Beispielen implizit davon aus, dass der Programmtext mit `import`-Anweisungen für `java.util.*` und `java.util.stream.*` beginnt.

```
public static long fact(int n) {
    return LongStream.rangeClosed(2, n).reduce(1, (i,j) -> i*j);
}
```

Faktorielle ist nichts anderes als die Reduktion einer fortlaufenden Zahlensequenz über Multiplikation. Das zu erkennen ist die Magie, die hinter Java-8-Streams steckt. Wir müssen eine solche Zahlensequenz erzeugen können. Wie in Abschnitt 2.3.3 könnten wir eine eigene Methode dafür schreiben, aber meist zahlt es sich aus, stattdessen in den Stream-Klassen nach passenden vorgefertigten Methoden zu suchen. Über Iteratoren und andere Implementierungsdetails müssen wir dabei kaum nachdenken – eine abstrakte Form des Programmierens.

Folgende Methode zeigt das Aufspalten und Zusammenfassen von Einträgen in Streams. Der Parameter `sales` stellt eine Ansammlung von Verkäufen dar,

die jeweils aus einer Menge von Produkten (als Strings) bestehen. Das Methodenergebnis bildet jedes Produkt auf eine `Map` ab, die angibt, welche anderen Produkte wie häufig zusammen mit diesem verkauft wurden.

```
public static Map<String, Map<String, Long>>
              toMap(Collection<Set<String>> sales) {
    return sales.stream()
        .flatMap(set -> set.stream()
            .flatMap(p -> set.stream()
                .filter(q -> !p.equals(q))
                .map(q -> new AbstractMap.SimpleEntry<>(p, q))
            )
        )
        .collect(Collectors.groupingBy(e -> e.getKey(),
             Collectors.groupingBy(e -> e.getValue(),
                  Collectors.counting())));
}
```

So wie `map` aus jedem Eintrag genau einen neuen Eintrag macht, macht `flatMap` aus jedem Eintrag beliebig viele (auch keinen) neuen Eintrag. Im Beispiel wird mittels `flatMap` zwei Mal ineinander geschachtelt über die Produkte pro Verkauf iteriert, wobei `set` für die Menge der Produkte eines Verkauf steht, `p` (aus einer Iteration) und `q` (aus einer anderen Iteration) für je ein Produkt innerhalb eines Verkaufs. Über `filter` werden jene `q` entfernt, die gleich den entsprechenden `p` sind, weil nur gezählt werden soll, wie viele *andere* Produkte `q` zusammen mit `p` verkauft wurden. Über `map` werden die `p` und `q` zu Paaren zusammengefasst. Da es in Java keine standardmäßig vordefinierten Klassen für Paare gibt und wir die Einführung eigener Klassen vermeiden wollen, verwenden wir dafür `AbstractMap.SimpleEntry`, eine Klasse, die eigentlich für Key-Value-Paare in Maps vorgesehen ist. Weil wir am Ende Objekte von `Map` erzeugen wollen, ist das nicht ganz unpassend. Aufgrund von `flatMap` entsteht ein einziger Strom mit `p`-`q`-Paaren, unabhängig davon, aus welchem Verkauf ein Paar stammt. Schließlich müssen wir die Ergebnisse nur mehr über `collect` zusammenfassen. Zu diesem Zweck bietet die Klasse `Collectors` umfangreiche Unterstützung. Es hat sich bewährt, sich bei der Auswahl geeigneter Methoden vom gewünschten Typ des Ergebnisses leiten zu lassen. Wir brauchen ein Ergebnis vom Typ `Map`, was die Auswahl erheblich reduziert. Bei genauerer Betrachtung bleibt nur die Methode `groupingBy`, deren beide Parameter beschreiben, wie der Schlüssel und der damit assoziierte Wert berechnet werden. Der Schlüssel soll das `p` in einem `p`-`q`-Paar sein, das wir mittels `getKey` aus dem Paar auslesen. Der zweite Parameter von `groupingBy` iteriert für jedes `p` über das entsprechende `p`-`q`-Paar. Die Werte unserer `Map` sollen wieder Objekte von `Map` sein, was zu einer weiteren Anwendung von `groupingBy` führt. Nun sind die Schlüssel die `q` in jedem `p`-`q`-Paar, die wir mittels `getValue` auslesen. Schließlich müssen wir noch den Wert des `Integer` aus einem Strom an `p`-`q`-Paaren ermitteln, wobei jedes `p` und jedes `q` für die Ermittlung eines Werts gleich ist. Dafür könnten

wir die Methode `reducing` aus `Collectors` mit einem geeigneten Lambda verwenden. Es geht aber einfacher: Die Methode `Collectors.counting` zählt die Paare und ermittelt damit genau das Ergebnis, das wir haben wollen.

Das Programm ist kurz und mit etwas Erklärung leicht nachvollziehbar. Aber viele Leute, die mit der prozeduralen Programmierung vertraut sind, nicht jedoch mit der applikativen, werden das Gefühl haben, dass Wesentliches in der Beschreibung fehlt. Was ist das Argument von `collect`? Es bringt keinen Erkenntnisgewinn, zu wissen, dass es ein `Collector` ist. Das, was dahinter steckt, bleibt abstrakt. An solche Abstraktionen müssen wir uns in der applikativen Programmierung gewöhnen. Wir müssen auch nicht wissen, welche Art von `Map` von `groupingBy` erzeugt wird. Vorteile ergeben sich vor allem dann, wenn wir gar nicht zu verstehen versuchen, wie die abstrakten Programmteile funktionieren, so lange sie das tun, was wir von ihnen erwarten.

Folgendes Beispiel löst zum Vergleich die gleiche Aufgabe auf ähnliche Weise mit Lambdas, aber ohne Streams:

```
public static Map<String, Map<String, Long>>
              toMap2(Collection<Set<String>> sales) {
    Map<String, Map<String, Long>> res = new HashMap<>();
    sales.forEach(set ->
        set.forEach(p -> {
            Map<String, Long> map = res.computeIfAbsent(p,
                                          k -> new HashMap<>());
            set.forEach(q -> {
                if (!p.equals(q))
                    map.compute(q, (k,v) -> v==null ? 1 : v+1);
            });
        })
    );
    return res;
}
```

Statt Schleifen werden `forEach`-Methoden verwendet, die im Wesentlichen das Gleiche machen wie Schleifen, aber nicht auf Seiteneffekte angewiesen sind. Anders als mit Streams, die auf Lazy-Evaluation beruhen, verwenden wir hier *Eager-Evaluation,* also die sofortige Berechnung. Zum schrittweisen Aufbau brauchen wir eine Datenstruktur wie `res` daher von Anfang an, nicht erst am Ende. Diese Lösung ist nicht frei von Seiteneffekten, weil Daten in die Hash-Tabellen gefüllt und bestehende Daten verändert werden. Aber alle Seiteneffekte sind auf abstrakte Datentypen (Hash-Tabellen) beschränkt, von denen wir wissen, dass Kommunikation über Variablen nur über entsprechende Zugriffsmethoden möglich ist. Es gibt keine destruktive Zuweisung in der Methode selbst, auch weil Lambdas das erzwingen, da wir sonst in den Lambdas nicht auf die Variablen zugreifen könnten. Aus diesem Grund verwenden wir Methoden wie `computeIfAbsent` und `compute` für Zugriffe auf die Hash-Tabellen; mit üblichen Methoden wie `put` und `get` könnten wir das Gleiche nur zusammen

mit destruktiven Veränderungen von Variablen erreichen. Lambdas in Methoden wie `compute` ziehen Berechnungen, die bei Verwendung von `put` und `get` außerhalb erfolgen müssten, in die Hash-Tabelle hinein.

Es stellt sich die Frage, was wir mit Streams machen können, was ohne sie nicht geht. Die Antwort ist ähnlich wie bei den Lambdas: Nichts. Es geht nicht ums Erhöhen der Mächtigkeit der Sprache, sondern darum, eine zusätzliche Abstraktionsebene einzuziehen. Mit entsprechender Erfahrung kann das Denken in Streams und den dahinter stehenden Mustern Komplexität aus Aufgaben nehmen, sodass sie effizienter lösbar sind. So wie die strukturierte Programmierung das Programmieren auf das Kombinieren weniger einfacher Denkmuster reduziert, können Streams das Programmieren auf andere, vor allem für algorithmisch komplexe Aufgaben noch einfachere Denkmuster reduzieren.

5.1.3 Applikative Programmierung in der Praxis

Fassen wir einige Erfahrungen bei der Programmierung mit Java-8-Streams und Lambdas zusammen:

- Die Abarbeitung folgt einem fixen Schema: Ausgangspunkt ist eine Ansammlung an Daten, Einträge sind weitgehend unabhängig voneinander. Einträge werden in beliebig vielen Schritten umgeformt, jeder Eintrag für sich (allgemein als *Map* bezeichnet). Die umgeformten Einträge werden am Ende aufgesammelt und in das gewünschte Format gebracht (als *Reduce* bezeichnet). Das gesamte Schema nennt sich daher *Map-Reduce*.

- Viele Programmieraufgaben sind nach dem Map-Reduce-Schema lösbar.

- Es reicht eine (nach einem gewissen Einlernaufwand) überschaubar kleine Menge an vorgefertigten Funktionen höherer Ordnung angewandt auf vergleichsweise einfache Lambdas, um nur damit (ohne komplexe eigene Methoden schreiben zu müssen) eine große Zahl von Map-Reduce-Aufgaben zu lösen – Kombinieren bestehender Funktionen statt Entwickeln eigener Funktionen. Das ergibt eine sehr effiziente Form der Programmierung.

- Map-Reduce basiert auf dem funktionalen Paradigma. Häufig müssen andere Methoden eingesetzt werden, als die sonst in Java üblichen, weil veränderliche Variablen aus der Umgebung in Lambdas nicht verwendbar sind (in `Map` etwa `compute` statt einer Kombination aus `get` und `put`).

- Generizität spielt eine große Rolle. Typparameter und Typen, die Typparameter ersetzen, stehen nur an wenigen Stellen explizit im Programm, die meisten werden über Typinferenz ermittelt und auf Korrektheit geprüft. Erst in einer fertig ausprogrammierten Stream-Anwendung sind die Typen in sich konsistent. In unvollständigen Ausdrücken kann eine IDE oft keinen Sinn erkennen, Methoden nicht den richtigen Typen zuordnen und keine brauchbaren Vorschläge machen.

- Wenn Ausdrücke so weit ausgereift sind, dass alle Typen in sich konsistent sind, dann sind diese Ausdrücke häufig auch inhaltlich fehlerfrei. Typkonsistenz ist in diesem Fall (wegen der komplexen Typabhängigkeiten) ein recht zuverlässiger Hinweis darauf, dass alles zusammenpasst. Fehlende Typkonsistenz kann Hinweise darauf liefern, was noch zu verbessern ist, kann aber auch zu Fehlinterpretationen führen.

- Funktionale Interfaces, die für Lambdas stehen, enthalten keine über Signaturen hinausgehende Informationen über das erwartete Verhalten der Lambdas, also keine Zusicherungen. Stattdessen sind alle der Signatur entsprechenden Lambdas akzeptabel. Nur auf diese Weise ist Typkonsistenz ein guter Indikator für Korrektheit. Aufgrund umfangreicher Typinferenz wäre es praktisch unmöglich, die Konsistenz von Zusicherungen händisch zu prüfen (strukturelle Abstraktion ≠ nominale Abstraktion und daher funktionale Abstraktion ≠ objektorientierte Abstraktion).

- Das Map-Reduce-Schema ist nicht die einzig mögliche Form der applikativen Programmierung, wenn auch eine wichtige. So wie Java-8-Streams auf Map-Reduce zugeschnitten sind, kann zu jedem beliebigen Programmierschema (nicht notwendigerweise auf Iteratoren beruhend) eine Menge von Klassen mit Funktionen höherer Ordnung entwickelt werden, die dieses Schema auf abstrakte Weise unterstützen. Häufig sind das sehr anwendungsspezifische Schemata. Die Entwicklung derartiger Funktionen höherer Ordnung in der nötigen Qualität kann sehr aufwändig und fordernd sein, allerdings wird die Programmierung im entsprechenden Schema damit möglicherweise stark vereinfacht. Es entsteht quasi eine eigene Sprache innerhalb der Programmiersprache. Jedes Schema hat andere Eigenheiten, aber alle oben genannten Punkte, die sich nicht direkt auf Map-Reduce beziehen, werden in jedem Schema zutreffen.

Ein bekanntes Sprichwort besagt: „Wer (nur) einen Hammer hat, sieht in jedem Problem einen Nagel.“ Das lässt sich leicht umformen in: „Wer (nur) mit Java-8-Streams umgehen kann, betrachtet jedes Problem als Map-Reduce-Problem.“ Fast jede Aufgabe lässt sich mehr oder weniger gut nach diesem Schema lösen, weil ja die meisten Programme Input aufsammeln und auf Output abbilden bzw. zu Output reduzieren. Wer es gewohnt ist, mit Streams zu arbeiten, ist in der Regel auch sehr kreativ darin, Wege zu finden, um Aufgaben nur mit vorhandenen Methoden und kleinen Ergänzungen über Lambdas nach dem Map-Reduce-Schema recht effizient zu lösen. Von dieser Kreativität kommt die Mächtigkeit des eigentlich simplen Werkzeugs, gleichzeitig ist das aber auch das größte Manko: Leute, die das Programm lesen, können die kreativen Ideen dahinter nur schwer erkennen und das Programm kaum verstehen.

> *Faustregel:* In nichttrivialen applikativen Programmteilen sollen wir Ideen hinter Vorgehensweisen durch Kommentare skizzieren. Zusicherungen auf dabei verwendeten kleinen Hilfsmethoden (Lambdas) sind dagegen zu vermeiden.

Auch wenn der Zweck der Beschreibung von Ideen nachvollziehbar ist, scheint diese Faustregel auf den ersten Blick in krassem Widerspruch zu den Faustregeln zu stehen, die wir im Zusammenhang mit der objektorientierten Programmierung betrachtet haben. Dort sind Zusicherungen auf Methoden ein wesentlicher Programmteil, während Kommentare zur Beschreibung des Programmablaufs häufig unnötig sind. Auf den Inhalt bezogen ähneln die Faustregeln einander jedoch: Das, was wir in der applikativen Programmierung unter der „Idee“ verstehen, haben wir in der objektorientierten Programmierung als Abstraktion bezeichnet; genau das ist in jedem Paradigma klar zu beschreiben. Während Abstraktionen in der objektorientierten Programmierung deutlich sichtbar als Klassen mit ihren Methoden (und Variablen) dargestellt werden, haben wir in der applikativen Programmierung keine entsprechend eindeutig identifizierbaren Programmstellen, an denen die Abstraktion passiert. Die Umsetzung einer Idee besteht ja nur aus Aufrufen vorgefertigter Funktionen, weswegen die Beschreibung der Idee nur bei diesen Aufrufen stehen kann. In gewisser Weise übernehmen Funktionen höherer Ordnung die Rolle von Kontrollstrukturen in der imperativen Programmierung und Lambdas entsprechen Ausdrücken oder Anweisungen in den Kontrollstrukturen. In der objektorientierten Programmierung vermeiden wir meist Kommentare auf einzelnen Teilen von Kontrollstrukturen (ausgenommen eventuell komplexere Schleifeninvarianten) weil diese Teile in der Regel auch ohne Kommentare gut lesbar sind und Kommentare den Lesefluss stören würden. Die gleiche Argumentation trifft auch auf Lambdas zu. Zusätzlich ergibt sich bei Lambdas das Problem, dass wir nicht im Detail wissen, wie die Funktionen höherer Ordnung diese Lambdas verwenden, sodass ein als Zusicherung zu verstehender Kommentar von uns fast gar nicht überprüfbar wäre. Der andere Umgang mit Kommentaren ist eine direkte Folge einer anderen Form von Abstraktion, nicht nur eine Konvention, die sich im Laufe der Zeit herausgebildet hat.

> *Faustregel:* Im Umfeld applikativer Programmteile sind Variablen so zu verwenden, als ob sie `final` wären.

Destruktive Zuweisungen könnten sich unkontrollierbar negativ auf scheinbar nicht betroffene Programmteile auswirken. Beim Einsatz von Lambdas wird das deutlich, weil sie nicht auf änderbare Variablen aus der Umgebung zugreifen dürfen. Ausgeführt werden Lambdas ja an anderen Stellen und zu anderen Zeitpunkten als sie eingeführt werden. Diese Faustregel impliziert, dass destruktive Zuweisungen auch auf Variablen, die in keinen Lambdas verwendet werden, vermieden werden sollen. Einerseits führen Programmänderungen leicht dazu, dass Variablen später doch in Lambdas benötigt werden, andererseits geht es um die Denkweise. Eine auf Zustandsänderungen durch direkte Zuweisungen beruhende Denkweise verträgt sich kaum mit der Verwendung von Lambdas.

> *Faustregel:* Meist ist es vorteilhaft, entweder ganz in einer funktionalen (nicht auf Zustandsänderungen ausgelegten) oder ganz in einer prozedural-objektorientierten Denkweise zu bleiben.

In Abschnitt 5.1.2 haben wir zwei Lösungen der gleichen Aufgabe gesehen, die Methode `toMap` beruhend auf Java-8-Streams, die Methode `toMap2` ohne Streams. Die Variante mit Streams können wir uneingeschränkt dem funktionalen Paradigma zuordnen. Obwohl Methoden aus Objekten aufgerufen werden, müssen wir bei keinem einzigen Methodenaufruf damit einhergehende Zustandsänderungen bedenken. Dabei spielt es keine Rolle, ob im Hintergrund möglicherweise doch Zustandsänderungen erfolgen. Möglicherweise zählt `equals` die Anzahl der Aufrufe mit, aber solche Zustandsänderungen bleiben uns verborgen. Die Variante `toMap2` ohne Streams entspricht dagegen einer prozeduralen Denkweise (obwohl in der Methode selbst keinerlei destruktive Zuweisungen erfolgen), weil der Algorithmus zur Gänze auf schrittweisen Zustandsänderungen der Datenstruktur beruht. Die Aufgabe ist in beiden Denkweisen gut lösbar. Eine Schwierigkeit der Variante mit Streams besteht darin, Paare von Strings zu bilden, die in der anderen Variante nicht nötig sind. Eine Schwierigkeit der Variante ohne Streams besteht darin, häufig Inhalte von Hash-Tabellen ändern zu müssen, was in der anderen Variante nicht nötig ist. Die Erfahrung zeigt, dass es oft eine schlechte Entscheidung ist, einen Teil eines Algorithmus' mit Streams und einen anderen Teil ohne Streams zu lösen, weil dann die Schwierigkeiten beider Ansätze gleichzeitig zu lösen wären. Häufig entstehen dennoch Methoden, die beide Denkansätze mischen, einfach weil kein besserer Lösungsansatz in den Sinn kommt. Eine nachträgliche Überarbeitung solcher Methoden kann die Qualität manchmal erheblich verbessern.

Um Verwirrungen zu vermeiden, wollen wir die Begriffe etwas klarer abgrenzen: Von einer *applikativen* Denkweise sprechen wir, wenn es darum geht, ganze Programme nur aus vorgefertigten Funktionen zusammenzusetzen. Das kann gut gelingen, wenn wir auf destruktive Zuweisungen verzichten und Lambdas einsetzen. Da dabei Funktionen eingesetzt werden, gibt es natürlich einen Bezug zur funktionalen Programmierung. Aber nicht jede applikative Denkweise ist notwendigerweise funktional, sie kann auch prozedural (oder objektorientiert) sein. Von einer *funktionalen* Denkweise sprechen wir, wenn keinerlei Zustandsänderungen mitbedacht werden müssen. Das impliziert natürlich auch den Verzicht auf destruktive Zuweisungen und den Einsatz von Lambdas. In einer *prozeduralen* Denkweise müssen Zustandsänderungen mitbedacht werden, unabhängig davon, ob auf direkte destruktive Zuweisungen verzichtet wird und Lambdas zum Einsatz kommen. Es stimmt nicht, dass eine funktionale Denkweise immer gut und eine prozedurale immer schlecht ist. Aber es stimmt, dass es in einer applikativen und gleichzeitig funktionalen Denkweise leichter ist, Programme nur aus vorgefertigten Funktionen zusammenzusetzen als in einer applikativen prozeduralen Denkweise; in einer nicht-applikativen Denkweise hätten wir das nicht als Ziel. Wenn das Zusammensetzen für eine Aufgabe in einer prozeduralen Denkweise gut gelingt, kann diese Lösung gegenüber einer funktionalen auch vorteilhaft sein.

> *Faustregel:* Funktionen (höherer Ordnung) sollen so allgemein wie möglich sein und Zustandsänderungen lokal halten.

Funktionen in den Stream-Klassen sind mächtig, weil sie sehr allgemein gehalten

und auf vielfältige Weise parametrisiert sind. Sie sind hochgradig generisch und verwenden Lambdas zur Festlegung einzelner Schritte. Die Funktionen lassen alles offen, was entweder beim Methodenaufruf festgelegt werden kann (Typen für Typparameter sowie Lambdas) oder als Implementierungsdetail dem Aufrufer nicht bekannt sein muss (etwa die Art der zurückgegebenen Collection). Dadurch sind die Funktionen nicht nur vielseitig anwendbar, sondern Typprüfungen können auch einen hohen Grad an Zuverlässigkeit garantieren, weil (fast) keine Annahmen gemacht werden, die über die vom Compiler prüfbaren Informationen in Typen hinausgehen. Damit wird es möglich, sich beim Kombinieren von Funktionen von der Typkonsistenz leiten zu lassen. Wir wissen, dass Streams intern auf Iteratoren beruhen und daher zustandsbehaftet sind. Aber die Zustände werden bei üblichen Stream-Anwendungen nicht von außen sichtbar.

Methoden in üblichen Objekten können nicht ganz so allgemein sein, weil die dahinter stehenden Abstraktionen (etwa über Zusicherungen) festgelegt werden müssen, sodass statisch prüfbare Typen nur einen Teil der für die Typkonsistenz nötigen Informationen enthalten. Sie können auch Zustände (je nach Abstraktion) nicht gänzlich verbergen. Dennoch sollten wir auch diese Methoden durch Parametrisierung so allgemein wie möglich halten, so wie wir das am Beispiel von `compute` in `Map` gesehen haben. Die Existenz dieser Methode zeigt auch, wie Zustandsänderungen lokal gehalten werden können, obwohl Aufrufer von den Zuständen wissen und Zustandsänderungen bewusst herbeiführen: Der Ort der Zustandsänderungen wird über Lambdas vom Aufrufer hin zur aufgerufenen Methode verschoben. Damit werden Seiteneffekte in der Umgebung des Aufrufers vermieden und die aufgerufene Methode bekommt Kontrolle über den Zeitpunkt der Zustandsänderung und die dabei verwendeten Werte (Argumente, mit denen Lambdas aufgerufen werden). Das ist in mehrerlei Hinsicht vorteilhaft, passt sehr gut zur objektorientierten Programmierung und ermöglicht applikative Denkweisen. Dennoch erfordert dies einen Umdenkprozess, weil an die Stelle der einfachen prozeduralen Denkweise eine auf die applikative Programmierung ausgelegte (prozedurale) Denkweise treten muss.

5.2 Funktionen höherer Ordnung

Java-8-Strems haben in der praktischen Java-Programmierung heute einen so hohen Stellenwert, dass daneben die zahlreichen anderen Möglichkeiten von Lambdas beinahe untergehen. Wir wollen uns nun damit beschäftigen, wie wir Lambdas als Funktionen höherer Ordnung auch ohne Streams einsetzen können. Damit entwickeln wir etwas, das Kontrollstrukturen in üblichen Programmiersprachen recht nahe kommt, sich aber nicht darauf beschränkt, was Programmiersprachen uns vorgeben.

5.2.1 Nachbildung typischer Kontrollstrukturen

Bedingte Anweisungen zählen zu den wichtigsten Kontrollstrukturen von Java und fast allen anderen Sprachen, die sehr tief in die Sprache integriert sind.

Zunächst zeigen wir, dass wir in Java (und allen objektorientierten Sprachen) auch ohne vordefinierten Typ `boolean` und ohne vorgegebene `if`-Anweisungen und ähnliche Kontrollstrukturen in der Lage sind, mit Booleschen Ausdrücken (in einem weiteren Sinn) zu arbeiten. Wir werden aber auch sehen, dass dies einen tiefen Einschnitt in die Sprache bedeutet und wir besonders vorsichtig agieren müssen, um semantische Details korrekt darzustellen. Die Basis für Fallunterscheidungen bildet natürlich dynamisches Binden:

```
interface Bool {
    <A> A ifThenElse(A t, A f);
    default Bool negate() {
        return ifThenElse(False.VALUE, True.VALUE);
    }
    default Bool and(Bool b) {
        return ifThenElse(b, False.VALUE);
    }
    default Bool or(Bool b) {
        return ifThenElse(True.VALUE, b);
    }
    default Bool isEqual(Bool b) {
        return ifThenElse(b, b.negate());
    }
}
final class True implements Bool {
    private True() {}
    public static final True VALUE = new True();
    public <A> A ifThenElse(A t, A f) { return t; }
}
final class False implements Bool {
    private False() {}
    public static final False VALUE = new False();
    public <A> A ifThenElse(A t, A f) { return f; }
}
```

Die Konstruktoren von `True` und `False` sind `private`, damit außer `True.VALUE` und `False.VALUE` keine weiteren Objekte dieser Klassen erzeugt werden können. Die Implementierungen des bedingten Ausdrucks `ifThenElse` ist sehr einfach: In `True` wird der eine Parameter zurückgegeben, in `False` der andere. Default-Implementierungen typischer Boolescher Operationen im Interface `Bool` werden durchwegs auf `ifThenElse` zurückgeführt.

Wenn wir `Bool` praktisch einsetzen, erkennen wir ein Problem: Wir müssen in jedem Aufruf von `ifThenElse` zwei Argumente spezifizieren, die beide sofort, noch vor Ausführung von `ifThenElse` ausgewertet werden. Das ist nicht die übliche Semantik einer bedingten Anweisung. Wir erwarten, dass nur eines der beiden Argumente ausgewertet wird, für `True` das erste und für `False` das zweite. Entsprechendes gilt auch für die Methoden `and` und `or`, die in der gegebenen Implementierung das Verhalten der in Java vordefinierten Operatoren `&` und `|` auf

`boolean` haben, nicht das der meist eingesetzten Kurzschlussoperatoren `&&` und `||`. Mit Funktionen höherer Ordnung ist dieses Problem lösbar. Beispielsweise fügen wir die Anweisung „`import java.util.function.Supplier;`“ und folgende Methoden zu `Bool` hinzu:

```
default <T> T getIfThenElse(Supplier<T> t, Supplier<T> f) {
    return ifThenElse(t, f).get();
}
default Bool andThen(Supplier<Bool> b) {
    return getIfThenElse(b, () -> False.VALUE);
}
default Bool orElse(Supplier<Bool> b) {
    return getIfThenElse(() -> True.VALUE, b);
}
```

Das funktionale Interface `Supplier<T>` enthält nur die parameterlose Methode `T get()`. Einem Aufruf von `getIfThenElse` übergeben wir nicht direkt die Werte, zwischen denen gewählt werden soll, sondern zwei parameterlose Lambdas, die die entsprechenden Werte zurückgeben. Der Aufruf von `ifThenElse` wählt eines der Lambdas aus, erst der Aufruf von `get()` bringt das gewählte Lambda zur Ausführung. Damit entspricht `getIfThenElse` viel eher einer `if`-Anweisung in Java und `andThen` sowie `orElse` entsprechen den Kurzschlussoperatoren `&&` und `||`. Beispielsweise gibt ein Aufruf

```
True.VALUE.orElse(() -> False.VALUE)
          .getIfThenElse(() -> "True", () -> "False")
```

als Ergebnis `"True"` zurück, ohne `()->False.VALUE` und `()->"False"` auszuwerten. Dieser Ansatz führt zu vielen Funktionen. Wir kommen unvermeidlich in den Bereich der funktionalen Programmierung.

`Bool` ist nur eine Nachbildung von `boolean`, nicht äquivalent zu `boolean`. Die vielen vordefinierten Operatoren und Methoden, die auf `boolean` beruhen, sind nicht automatisch für `Bool` verfügbar. Für eine vollständige Nachbildung müssten wir alle diese Operatoren und Methoden neu schreiben. Auf diesen Aufwand verzichten wir gerne. Die Beispiele sollen nur ein Gefühl dafür vermitteln, wie solche Nachbildungen aussehen könnten. Simple Konvertierungsfunktionen können eine Brücke zwischen `Bool` und `boolean` schlagen (innerhalb von `Bool`):

```
default boolean toBoolean() {
    return this == True.VALUE;
}
static Bool fromBoolean(boolean b) {
    return b ? True.VALUE : False.VALUE;
}
```

Lambdas spielen in `getIfThenElse` eine ähnliche Rolle wie Iteratoren in Java-8-Streams: Sie sorgen dafür, dass Ergebnisse nicht gleich berechnet werden, sondern erst später auf Anfrage, wenn sich ein Bedarf dafür ergibt. Ergibt sich kein

Bedarf, bleiben Ausdrücke unausgewertet. Der Zeitpunkt der Auswertung ist in `getIfThenElse` und damit auch in `andThen` und `orElse` fix festgelegt, um eine größtmögliche Nähe zur Semantik einer `if`-Anweisung in Java herzustellen. Da wir nun schon einfache Möglichkeiten zur Verschiebung von Ausführungszeitpunkten kennen, lassen sich die Zeitpunkte auch deutlich weiter, beinahe beliebig weit nach hinten schieben. Davon wird in der funktionalen Programmierung häufig Gebrauch gemacht. Das folgende Interface kann als Variante von `Bool` mit Lazy-Evaluation gesehen werden:

```
import java.util.function.*;
@FunctionalInterface
interface LazyBool extends Supplier<Bool> {
    static final LazyBool TRUE = () -> True.VALUE;
    static final LazyBool FALSE = () -> False.VALUE;
    default <T> Supplier<T> ifThenElse(Supplier<T> t,
                                       Supplier<T> f) {
        return () -> get().ifThenElse(t, f).get();
    }
    default LazyBool negate() {
      return () -> get().ifThenElse(False.VALUE, True.VALUE);
    }
    default LazyBool and(LazyBool b) {
        return () -> get().ifThenElse(b, FALSE).get();
    }
    default LazyBool or(LazyBool b) {
        return () -> get().ifThenElse(TRUE, b).get();
    }
    default LazyBool isEqual(LazyBool b) {
        return () -> get().ifThenElse(b, b.negate()).get();
    }
}
```

Eine Konsequenz aus der Verwendung von Lazy-Evaluation ist, dass fast alle Werte, die im Programm vorkommen, Funktionen bzw. Lambdas sind, so wie `TRUE` ein Lambda ist. Wer einen entsprechenden Programmierstil gewohnt ist, findet daran nichts Ungewöhnliches. Ein Großteil aller Berechnungen besteht daraus, ein Netzwerk an Verbindungen zwischen Funktionen aufzubauen, die erst am Ende (wenn für eine Ausgabe konkrete Werte, die keine Funktionen sind, benötigt werden) zur Ausführung kommen.

> *Faustregel:* Es gibt zwei sinnvolle Ausführungszeitpunkte für Funktionen: so früh wie möglich (Eager-Evaluation) oder so spät wie möglich (Lazy-Evaluation). Andere Zeitpunkte sind eher zu meiden.

In dieser Faustregel nehmen wir an, dass zur Erhaltung der Semantik nötige Verschiebungen der Ausführungszeitpunkte wie in `getIfThenElse`, `andThen` und `orElse` noch zu Eager-Evaluation zählen, frühere Ausführungszeitpunkte also

nicht möglich sind. Eager-Evaluation ist damit begründbar, dass wir beim Programmieren die genauen Ausführungszeitpunkte stets im Kopf haben und stets wissen, wann was passiert. So werden Programme verständlich und der Verwaltungsaufwand auf ein Minimum reduziert (was häufig zu guter Laufzeiteffizienz führt). Bei Lazy-Evaluation verzichten wir dagegen auf das Nachverfolgen der genauen Ausführungszeitpunkte, wir haben nur die logischen Zusammenhänge im Kopf. Ein Verzicht auf die Kontrolle der Zeitpunkte macht Programme einfach verständlich, der höhere Verwaltungsaufwand wird oft dadurch kompensiert, dass keine unnötigen Berechnungen ausgeführt werden (was häufig auch zu ausreichend guter Laufzeiteffizienz führt). Andere Ausführungszeitpunkte sind meist schlecht gewählt, da sie Programme schwerer verständlich machen (Ausführungszeitpunkte müssen kontrolliert werden, sind aber nur schwer kontrollierbar) und die Laufzeiteffizienz oft schlecht ist (viel Verwaltungsaufwand, unnötige Berechnungen nicht vollständig eliminiert).

Erkenntnisse aus der Nachbildung bedingter Ausführungen lassen sich auf andere Kontrollstrukturen übertragen. Besonders einfach ist die Nachbildung der Hintereinanderausführung durch Zusammensetzen von zwei Lambdas zu einem (hier als Klassenmethode in irgendeiner Klasse):

```
public static <T,V,R> Function<T,R>
              compose(Function<V,R> f, Function<T,V> g) {
    return t -> f.apply(g.apply(t));
}
```

Das dabei verwendete funktionale Interface `Function<T,R>` aus dem Paket `java.util.function` enthält die Methode `R apply(T t)`. Zurück kommt ein Lambda, das zuerst `g` auf das Argument `t` des Lambdas anwendet, danach `f` auf das Ergebnis davon. Beispielsweise führt

```
compose(String::length, String::trim).apply(" a ")
```

`" a ".trim().length()` aus und gibt 1 zurück. Der Ausführungszeitpunkt des durch `compose` erzeugten Lambdas, das ist der Zeitpunkt, an dem `apply` ausgeführt wird, lässt sich beliebig weit in die Zukunft verschieben.

Neben sequenzieller und bedingter Ausführung ist die wiederholte Ausführung ein wesentliches Element der strukturierten Programmierung. Mangels destruktiver Zuweisung in der funktionalen Programmierung bieten sich dafür wiederholte Iterationen durch Rekursion an:

```
 public static <T> Function<T,T>
              loopWhile(Function<T,Bool> cond,
                        Function<T,T> iter) {
    Function<T,T> doIt = i -> loopWhile(cond, iter)
                              .apply(iter.apply(i));
    Function<T,T> done = i -> i;
    return init -> cond.apply(init)
                        .ifThenElse(doIt, done).apply(init);
}
```

Die Schleifenbedingung `cond` und eine Funktion `iter`, die einen Iterationsschritt festlegt, werden als Parameter an `loopWhile` übergeben. Das Ergebnis ist ein Lambda, das durch einen Aufruf von `apply` mit einem Anfangswert `init` als Parameter zur Ausführung gebracht wird. Erfüllt `init` die in `cond` festgelegte Bedingung, wird `doIt` auf `init` angewandt, sonst `done`. Dabei ist `done` ein Lambda, das einfach nur das Argument unverändert zurückgibt; `doIt` macht dagegen den rekursiven Aufruf von `loopWhile` angewandt auf das Ergebnis einer Anwendung von `iter` auf `init`. Insgesamt wird also wiederholt immer wieder `iter` angewandt, bis eine Anwendung von `cond` als Ergebnis `False` liefert. Hier ist ein Beispiel für eine Anwendung von `loopWhile`:

```
loopWhile((String s) -> Bool.fromBoolean(s.charAt(0)==' '),
          s -> s.substring(1))
.apply("  a")
```

Die Schleifenbedingung ist erfüllt, solange das erste Zeichen der Zeichenkette im Parameter ein Leerzeichen ist; jeder Iterationsschritt entfernt dieses. Angewandt auf `"  a"` wird also `"a"` zurückgegeben. In diesem Beispiel ist es notwendig, in zumindest einem der beiden Lambdas den Typ des Parameters hinzuschreiben, weil Typinferenz über die Typparameter nur feststellen kann, dass beide Parameter vom gleichen Typ sind, aber nicht von welchem. Ohne explizite Deklaration von `String` könnten wir nicht auf die Methoden von `String` zugreifen.

5.2.2 Funktionale Elemente in Java

Wir haben Kontrollstrukturen nachgebildet, um den Zusammenhang mit Funktionen höherer Ordnung zu sehen und einige dabei eingesetzte Programmiertechniken kennenzulernen. Praktisch werden wir kaum bestehende Kontrollstrukturen nachbilden, sondern neue Funktionalität hinzufügen. Wir müssen uns nicht auf die funktionale Programmierung beschränken, da beliebige Methoden Lambdas verwenden können; nur die Lambdas selbst sollten sich an der funktionalen Programmierung orientieren. Hier ist ein Beispiel, das ein Lambda auf jeden Array-Eintrag anwendet und das Array dabei verändert:

```
public static <T> void arrayMap(T[] xs, Function<T,T> f) {
    for (int i = 0; i < xs.length; i++)
        xs[i] = f.apply(xs[i]);
}
```

Es ist zwar nicht schwer, Methoden wie diese zu schreiben, aber häufig ist das gar nicht nötig. Sehr viele sinnvolle Methoden sind schon in Standardbibliotheken vordefiniert. Eine Methode wie `arrayMap` gibt es zwar nicht genau in dieser Form, aber eine vordefinierte Methode lässt sich auf diese Weise verwenden:

```
public static <T> void arrayMap2(T[] xs, Function<T, T> f) {
    Arrays.setAll(xs, i -> f.apply(xs[i]));
}
```

In `Arrays` unterscheidet sich `setAll` von unserem `arrayMap` im Wesentlichen nur dadurch, dass das Lambda als Parameter den Index des Array-Eintrags erwartet, nicht den Wert an diesem Index. Solche kleinen Unterschiede lassen sich beim Aufruf leicht anpassen.

> *Faustregel:* Vor dem Implementieren einer eigenen Funktion höherer Ordnung sollten wir uns vergewissern, dass nicht eine ähnliche Methode schon standardmäßig vordefiniert ist. Die vordefinierte Methode ist zu bevorzugen.

Tatsächlich sind viele solche Methoden vorimplementiert. Der Grund liegt darin, dass Funktionen höherer Ordnung von Natur aus meist allgemein gehalten, also nicht anwendungsspezifisch sind. Da immer wieder gleiche oder ähnliche Funktionen höherer Ordnung gebraucht werden (im Gegensatz zu anwendungsspezifischen Methoden), ist eine überschaubare Menge entsprechender Methoden ausreichend. Vordefinierte Methoden sind von hoher Qualität, weil sie in allen Details durchdacht und sehr ausgiebig, auch im Praxiseinsatz, getestet wurden. Die Schwierigkeit liegt darin, dass eben nur ähnliche Methoden vorimplementiert sind, nicht genau die erwarteten. Wir brauchen etwas Fantasie, um die Ähnlichkeit zu erkennen. Lambdas lassen sich leicht anpassen, wodurch eine schwach ausgeprägte Ähnlichkeit ausreicht und eine einzige Methode ein breites Anwendungsspektrum erschließen kann. Wer die wichtigsten vordefinierten Funktionen höherer Ordnung kennt und in der Lage ist, Ähnlichkeiten richtig zu erkennen, kann sehr effizent programmieren und dabei Programme von hoher Qualität schreiben. Sowohl das Kennen der Methoden (deren Menge ständig erweitert wird) als auch das Erkennen von Ähnlichkeiten hängt von der Erfahrung ab.

`Optional`. Die Verfügbarkeit von Funktionen höherer Ordnung lässt Programmiertechniken, die bislang nur in der funktionalen Programmierung verbreitet waren, auch in die Java-Programmierung und allgemein in die objektorientierten Programmierung einsickern. Java-8-Streams sind ein Beispiel dafür. Nun wollen wir im Zusammenhang mit der vordefinierten Klasse `Optional` eine weitere solche Programmiertechnik betrachten. Ein Objekt von `Optional<T>` enthält einfach nur ein Objekt vom Typ `T` oder ist leer, was im Wesentlichen gleichbedeutend damit ist, dass das enthaltene Objekt `null` ist. Methoden von `Optional` bieten, ohne das direkt zu sagen, eine Reihe von Möglichkeiten für den Umgang mit `null` an. Beispielsweise gibt die Methode `isPresent()` genau dann `true` zurück, wenn das enthaltene Objekt nicht `null` ist. Diese Methode wird meist als Bedingung in einer bedingten Verzweigung eingesetzt, etwa in der gleichen Bedeutung wie `x!=null` auf einem enthaltenen Wert `x`. Interessanter sind Methoden von `Optional`, die eine Weiterverarbeitung ohne (direkt sichtbare) bedingte Programmverzweigung ermöglichen. So liefert die Methode `T orElse(T other)` als Ergebnis das enthaltene Objekt `x` falls es existiert, andernfalls den Wert `other`; für `other!= null` bekommen wir also immer ein

Ergebnis ungleich `null`. Die Variante `orElseGet` nimmt statt `other` ein Lambda und gibt bei `x==null` das Ergebnis einer Ausführung des Lambdas zurück. Die Variante `orElseThrow` nimmt ebenfalls ein Lambda, das bei `x==null` eine Exception wirft. Die Methode `ifPresent` liefert kein Ergebnis, sondern führt bei `x!=null` einfach nur das als Parameter übergebene Lambda aus. Praktisch von größerer Bedeutung ist die Methode `map` mit einem Parameter (Lambda) vom Typ `Function<T,U>` (etwas vereinfacht), die ein neues Objekt von `Optional<U>` zurückgibt; das im Ergebnis-Optional enthaltene Objekt ist das Ergebnis einer Anwendung des Lambdas auf das Objekt im ursprünglichen Optional (falls ein solches Objekt existiert) oder sonst ein leeres Optional. Mit Hilfe von `map` können wir umfangreiche Berechnungen auf Werten durchführen, ohne jemals darauf zu achten, ob diese Werte überhaupt existieren.

`Optional` ist eine einfache Klasse mit recht simpler Funktionalität, die dennoch von großer praktischer Bedeutung ist. Wie wir in Abschnitt 5.2.1 gesehen haben, sollen im Zusammenhang mit der funktionalen Programmierung und Lazy-Evaluation Ausführungszeitpunkte so weit wie möglich auf später verschoben werden. Eine Programmverzweigung verlangt, dass wir die Bedingung, etwa `x==null`, ausführen, um feststellen zu können, welcher Programmzweig zu wählen ist. `Optional` bietet, vor allem zusammen mit `map`, eine einfache Möglichkeit, diese Entscheidung auf später zu verschieben. Es ist eine gängige Praxis, in der funktionalen Programmierung in Java auf den expliziten Umgang mit `null` so weit wie möglich zu verzichten und stattdessen `Optional` einzusetzen. Das erleichtert Lazy-Evaluation. Außerdem kann `Optional` dazu dienen, den Zeitpunkt des Werfens einer Exception auf später zu verschieben oder gänzlich zu vermeiden (weil der Programmteil, in dem die Exception auftreten würde, gar nicht zur Ausführung kommt).

> *Faustregel:* Zusammen mit Lazy-Evaluation soll auf den expliziten Umgang mit `null` verzichtet und stattdessen `Optional` eingesetzt werden. Zusammen mit Eager-Evaluation ist `Optional` wenig sinnvoll und ein expliziter Umgang mit `null` vorteilhaft.

Da wir in einem Programmteil entweder nur Lazy-Evaluation oder nur Eager-Evaluation (nicht gemischt) einsetzen, ist es sinnvoll, entweder nur `Optional` einzusetzen oder nur explizit mit `null` umzugehen (nicht gemischt).

Folgendes Beispiel setzt `Optional` zusammen mit einem Stream ein:

```
public static Optional<FileReader> openFile(String... path) {
    return Stream.of(path)
        .map(String::trim)
        .reduce((s, t) -> s + "/" + t)
        .map(s -> {try{return new FileReader(s);}
                   catch(java.io.IOException e){return null;}});
}
```

Die einzelnen Elemente von `path` werden im Stream über das erste `map` bearbeitet (Leerzeichen am Rand entfernt) und danach über `reduce` zu einer

Zeichenkette mit `"/"` zwischen den Elementen reduziert. Diese Variante von `reduce` verwendet als Anfangswert das erste Stream-Element, wodurch ein leerer Stream nicht bearbeitbar ist. Daher gibt `reduce` ein Ergebnis vom Typ `Optional<String>` zurück. Das zweite `map` wird im Optional-Objekt ausgeführt. Dabei wird die von `reduce` erzeugte Zeichenkette als Datei-Pfad interpretiert und ein `FileReader` geöffnet. Falls keine Datei öffenbar ist, gibt das Lambda in `map` nach Abfangen der Exception `null` zurück, was zu einem leeren `Optional` führt. Das Ergebnis von `map` und damit auch von `openFile` ist ein Optional-Objekt, das entweder einen `FileReader` enthält, oder leer ist, wenn `path` leer ist (Lambda in `map` nicht aufgerufen) oder keine Datei öffenbar ist (Lambda in `map` liefert `null`). Ein Aufrufer von `openFile` kann mit dem Optional-Objekt weiterarbeiten, vielleicht durch einen weiteren Aufruf von `map`.

Currying. Wie schon in Abschnitt 1.1.2 ausgeführt, sind Funktionen mit nur einem Parameter ausreichend, um Funktionen mit beliebig vielen Parametern darzustellen. Die Technik dahinter ist als *Currying* bekannt, benannt nach Haskell Curry, einem Mathematiker, der viel zu den Grundlagen der funktionalen Programmierung beigetragen hat; auch die Programmiersprache Haskell ist nach ihm benannt. Die Technik ist einfach: Statt einer Funktion mit zwei Parametern schreiben wir eine Funktion mit nur einem Parameter, die als Ergebnis eine Funktion zurückgibt, die den zweiten Parameter hat und das eigentliche Ergebnis berechnet. Wiederholt angewandt lässt sich die Zahl der Parameter damit beliebig erhöhen. Die beiden folgenden Funktionen `f` und `g` machen inhaltlich das Gleiche, aber `f` hat zwei Parameter und `g` verwendet Currying:

```
BiFunction<String,String,String> f = (s, t) -> s + t;
Function<String,Function<String,String>> g = s -> t -> s + t;
```

Wir sehen hier, dass `BiFunction` inhaltlich große Ähnlichkeit zu zwei geschachtelten Vorkommen von `Function` hat. Tatsächlich sind die Typen verschieden, da es sich um nominale Typen handelt, die nicht in einer Untertypbeziehung zueinander stehen. Wir können `f` also nicht dort verwenden, wo `g` erwartet wird und `g` nicht dort, wo `f` erwartet wird. Aber wir können frei entscheiden, welche der beiden Varianten wir einsetzen wollen, weil sie inhaltlich das Gleiche machen. Die beiden entsprechenden Lambda-Ausdrücke verursachen etwa den gleichen Schreibaufwand. Hinsichtlich der Auswertungen dieser Lambdas ergeben sich jedoch Unterschiede:

```
String s = f.apply("a", "b");
String t = g.apply("a").apply("b");
```

Mit Currying werden zwei Funktionen aufgerufen, nicht nur eine. Wir können aus dieser Gegenüberstellung eine Reihe von Schlussfolgerungen ableiten:

- In Java sind keine funktionalen Interfaces für Funktionen mit mehr als zwei Parametern vordefiniert, weil wir auch mit Funktionen mit nur einem Parameter alles ausdrücken können. Funktionale Interfaces für zwei

Parameter gibt es, weil sich viele Funktionen so auf gewohnte Weise ausdrücken lassen, nicht weil sie nötig sind. Wenn wir wollen, können wir funktionale Interfaces für beliebig viele Parameter schreiben. Das ist aber kaum sinnvoll. Currying ist die bessere Alternative. Spezielle funktionale Interfaces für parameterlose „Funktionen“ sind dagegen schon sinnvoll und auch vorhanden; allerdings ist der Begriff „Funktion“ dafür nicht passend, Namen wie `Supplier` also sicher besser gewählt.

- Die Syntax von Lambdas ist so ausgelegt, dass Currying keinen zusätzlichen Schreibaufwand verursacht (keine Klammerung nötig) und, wenn man den Umgang damit gewohnt ist, ganz natürlich aussieht. Ausdrücke wie `a -> b -> ... -> ...` sind genau so leicht als Aneinanderreihung mehrerer Parameter vor einem Rumpf lesbar wie als Ineinanderschachtelung so vieler Lambdas wie `->` vorhanden sind. Es besteht kein Unterschied zwischen diesen beiden Lesarten.

- Bei der Auswertung von Lambdas ist die Variante mit Currying etwas aufwändiger, weil für jeden Parameter ein eigener Aufruf nötig ist. Auch der Ressourcenverbrauch hinsichtlich Speicher und Laufzeit ist mit Currying etwas größer. Das ist ein Nachteil dieser Technik, der aber durch optimierende Compiler verkleinert werden kann.

- Currying erhöht die Flexibilität bei der Auswertung. Es wird nicht verlangt, dass alle Argumente, die nötig sind, gleichzeitig an einer bestimmten Stelle im Programm vorliegen, wie das bei einem einzigen Methodenaufruf nötig wäre. Wir können die Aufrufe auch schrittweise an verschiedenen Programmstellen machen. Beispielsweise übergeben wir an einer Programmstelle nur ein Argument und reichen das Ergebnis (ein Lambda) an eine andere Programmstelle weiter, wo das nächste Argument vorliegt und übergeben wird. Eine derartige Vorgehensweise kann die Gesamtzahl der Parameter in einem Programm reduzieren und effizient sein. Es erfordert jedoch viel Programmiererfahrung, um Programme so organisieren zu können, dass solche Effekte zum Tragen kommen.

- Typen von Lambdas können zusammen mit Currying sehr umfangreich und komplex werden. Wenn generische Typen im Wesentlichen nur vom Compiler durch Typinferenz ermittelt werden, ist das kein Problem. Allerdings kann es aufwändig sein, komplexe Typen etwa bei Variablendeklarationen hinzuschreiben. `var`-Deklarationen helfen dabei nicht.[4]

- Komplizierte Typen haben auch Vorteile: Wenn es gelingt, Programme mit nicht-trivialen Lambdas so zu gestalten, dass alle Typen in sich kon-

[4]Seit Java 10 ist es möglich, initialisierte lokale Variablen ohne Typangabe zu deklarieren; statt dem Typ wird das Schlüsselwort `var` verwendet. Meist ist das kein Problem, weil der Typ ohnehin direkt aus der Initialisierung ersichtlich ist. Gerade für komplizierte Typen sollten wir `var` jedoch nicht einsetzen, weil die explizite Typinformation die Lesbarkeit deutlich erhöhen kann. Zusammen mit Lambdas ist `var` nicht einsetzbar, weil Lambdas ihre Typinformation aus den expliziten Typdeklarationen beziehen.

sistent sind, dann sind diese Programme auch inhaltlich ausreichend gut durchdacht, um eine Vielzahl möglicher Fehler zu vermeiden.

Kurz zusammengefasst: Wir brauchen keine Angst vor Currying haben. Wer sich daran gewöhnt hat, wird auf die damit verbundene Flexibilität und gleichzeitig Sicherheit (durch statische Typisierung) nicht mehr verzichten wollen. Wer die Programmierung in einer neueren funktionalen Sprache gewohnt ist, wird sich eine Programmierung ohne Currying kaum vorstellen können. Wer sich aber nicht daran gewöhnen möchte, kommt in Java derzeit auch noch ohne Currying ganz gut zurecht.

Pattern-Matching. Aus neueren funktionalen Sprachen ist Pattern-Matching, vor allem bei Funktionsaufrufen, nicht mehr wegzudenken. Dabei bestimmen Werte in Parametern, welche Funktion auszuführen ist. Es besteht eine Nähe zu Multimethoden (Abschnitt 4.4), aber Parameter sind nicht durch Typen, sondern konkrete Werte festgelegt. Wenn es in Java Pattern-Matching gäbe, könnte ein Beispiel zur Berechnung der Länge einer Zeichenkette so aussehen:

```
int strLength("") {return 0;}
int strLength([char c, String s] c+s) {return 1+strLength(s);}
```

Wir könnten das Programmstück so lesen: Wenn der Parameter gleich dem Literal `""` ist, wird die erste Methode ausgeführt und `0` zurückgegeben. Andernfalls muss der Parameter eine nicht-leere Zeichenkette sein, die als Zusammenfügung eines Zeichens mit einer Zeichenkette verstehbar ist. Wir müssen nur mehr die Länge 1 des Zeichens zur Länge der restlichen Zeichenkette addieren. So funktioniert das in Java natürlich nicht, nicht nur aufgrund der für Pattern-Matching fehlenden Syntax. Wenn wir das Beispiel nach Java übersetzten, würde etwa folgende, recht ineffiziente Methode entstehen:

```
int strLength(String s) {
    return s.equals("") ? 0 : 1 + strLength(s.substring(1));
}
```

Problematisch ist, dass `strLength` kaum zur Abstraktion durch `String` passt. Keine Implementierung außerhalb von `String` kann direkt auf die benötigten Variablen zugreifen. Innerhalb von `String` wäre die Methode einfach und effizient zu implementieren; `length` ist ohnehin vorimplementiert. Der wichtigste Grund, warum es in Java und den meisten objektorientierten Sprachen kein Pattern-Matching gibt, ist ein gänzlich anderer Umgang mit Abstraktionen als in funktionalen Sprachen. Funktionale Sprachen machen die Struktur der Daten öffentlich sichtbar, wodurch es leicht ist, von überall aus direkt (ohne Methodenaufrufe) auf die einzelnen Zeichen einer Zeichenkette zuzugreifen. Weil die bestehenden Daten nicht änderbar sind, ist das in der funktionalen Programmierung ein viel kleineres Problem als in der imperativen Programmierung. In der objektorientierten Programmierung ist Datenabstraktion so wesentlich, dass es unsinnig wäre, für eine schönere Syntax darauf zu verzichten. In einer abgespeckten Variante beruhend auf Literalen (die ohnehin überall sichtbar sind) wäre Pattern-Matching natürlich auch in Java denkbar, etwa in dieser Form:

```
int strLength("") {return 0;}
int strLength(String s) {return 1 + strLength(s.substring(1));}
```

Das ist nur eine andere Syntax für bedingte Verzweigungen, Sichtbarkeit bleibt unberührt, die Ineffizenz nicht beseitigt. Überlegungen zu Multimethoden aus Abschnitt 4.4 kommen zum Tragen. Wenn wir an dynamisches Binden denken, löst sich das Problem von alleine: Angenommen, `""` wäre das einzige Objekt eines Untertyps von `String`; dann könnte die erste Methode in diesem Untertyp und die zweite in `String` implementiert sein. Es bleibt nur das Problem, dass die beiden Methoden an unterschiedlichen Stellen stehen würden.

5.3 Nebenläufige Programmierung in Java

Grundlegende Mechanismen für das Erzeugen von Threads und die Synchronisation in Java haben wir schon in Abschnitt 2.5 betrachtet. Zum besseren Verständnis behandeln wir diese Mechanismen hier noch einmal auf andere Weise. In der Praxis werden für die nebenläufige Programmierung häufig Sprachmechanismen auf einer etwas höheren Ebene eingesetzt, die wir uns im Anschluss daran vor Augen führen.

5.3.1 Thread-Erzeugung und Synchonisation in Java

Folgendes Beispiel soll ein Synchronisationsproblem demonstrieren:

```
public class Counter {
    private int i = 0, j = 0;
    public void flip() { i++; j++; }
}
```

Die Variablen `i` und `j` sollten stets die gleichen Werte enthalten. Wenn wir jedoch in mehreren nebenläufigen Threads `flip` in demselben Objekt von `Counter` wiederholt aufrufen, kann es vorkommen, dass sich `i` und `j` plötzlich voneinander unterscheiden. Den Grund dafür finden wir in der fehlenden Synchronisation: Bei Ausführung des `++`-Operators wird der Wert der Variablen aus dem Speicher gelesen, um eins erhöht und wieder in den Speicher geschrieben. Wird nun `flip` in zwei Threads annähernd gleichzeitig ausgeführt, wird von beiden Threads der gleiche Wert aus der Variablen gelesen, jeweils um eins erhöht, und von beiden Threads derselbe Wert zurückgeschrieben. Das ist nicht das, was wir haben wollen, da sich ein Variablenwert bei zwei Aufrufen nur um eins erhöht hat. Unterschiede zwischen den Werten von `i` und `j` ergeben sich, wenn das nur beim Ändern einer der beiden Variablen passiert.

In einer `synchronized`-Methode kann das nicht passieren:

```
public synchronized void flip() { i++; j++; }
```

In jedem Objekt wird zu jedem Zeitpunkt höchstens eine `synchronized` Methode ausgeführt. Wenn mehrere Threads `flip` auf dem gleichen Objekt annähernd

gleichzeitig aufrufen, werden alle bis auf einen Thread solange blockiert, bis dieser eine aus `flip` zurückkehrt. Dann darf der nächste Thread `flip` ausführen und so weiter. Die oben beschriebenen Synchronisationsprobleme sind damit beseitigt. Die Methode wird *atomar*, also wie eine nicht weiter in Einzelteile zerlegbare Einheit ausgeführt.

> *Faustregel:* In nebenläufigen Programm(teil)en sollen alle Methoden, die auf Objekt- oder Klassenvariablen zugreifen, `synchronized` sein.

Wie in `flip` werden dadurch Inkonsistenzen verhindert. Das gilt vor allem für ändernde Zugriffe wie im Beispiel. Auch bei nur lesenden Zugriffen ist häufig Synchronisation notwendig, um zu verhindern, dass inkonsistente Daten gelesen werden (z. B. `i` schon erhöht, `j` aber noch nicht).

> *Faustregel:* `synchronized` Methoden sollen nur kurz laufen.

Die Einhaltung dieser Faustregel verringert sowohl die Wahrscheinlichkeit für das Blockieren von Threads als auch die durchschnittliche Dauer von Blockaden. Überlegungen zur Synchronisation sind aufwändig. Daher werden manchmal nur wichtige, große Methoden synchronisiert und in kleinen Hilfs-Methoden, die nur von `synchronized` Methoden aus aufgerufen werden, darauf verzichtet. Das widerspricht jedoch der Forderung nach kurz laufenden Methoden und ist kein guter Programmierstil. Richtig ist es, die Granularität der Synchronisation so zu wählen, dass kleine, logisch konsistente Blöcke entstehen, in deren Ausführung man vor Veränderungen durch andere Threads geschützt ist. Oft bilden Methoden solche logischen Blöcke, aber große Methoden sind nicht selten in kleinere logische Blöcke aufzuteilen. Um diese Aufteilung zu erleichtern, gibt es in Java neben synchronisierten Methoden auch synchronisierte Blöcke:

```
public void flip() {
    synchronized(this) { i++; }
    synchronized(this) { j++; }
}
```

Die Ausführungen der Befehle `i++` und `j++` werden getrennt voneinander synchronisiert. Die Methode als ganze braucht nicht synchronisiert zu werden, da in ihr außerhalb von `synchronized`-Blöcken nirgends auf Objekt- oder Klassenvariablen zugegriffen wird. In dieser Variante von `flip` ist es zwar möglich, dass `i` und `j` kurzfristig unterschiedliche Werte enthalten (z. B. weil mehrere Threads, die im nächsten Schritt `i` erhöhen, früher an die Reihe kommen als jene, die `j` erhöhen), aber am Ende des Programms sind `i` und `j` gleich; es wird keine Erhöhung vergessen.

Zur Synchronisation verwendet Java *Locking.* Ein „Lock“ kann in jedem Objekt auf einen bestimmten Thread gesetzt sein um zu verhindern, dass ein anderer als dieser Thread auf das Objekt zugreift. Das Argument des `synchronized`-Blocks bestimmt das Objekt, dessen Lock zu setzen ist. Bei `synchronized` Methoden ist das immer das Objekt, in dem die Methode aufgerufen wird, also

`this`. Dieser Mechanismus erlaubt rekursive Aufrufe: Da Locks bereits auf die richtigen Threads gesetzt sind, müssen sich rekursive Aufrufe nicht mehr um Synchronisation kümmern.

Einzelne Schreib- und Lesezugriffe auf `volatile` Variablen (also solche, die mit diesem Modifier deklariert wurden) sind atomar. Das reicht nicht, wenn wie in `i++` mehrere Variablenzugriffe erfolgen. Aber einige Klassen wie etwa `AtomicInteger` bieten Methoden an, die Werte einzelner Variablen ohne `synchronized` atomar ändern.

Manchmal soll die Ausführung von Threads von weiteren Bedingungen abhängen, die Threads unter Umständen für längere Zeit blockieren. Die Methode `onOff` in folgender Klasse schaltet einen Drucker online bzw. offline und steuert damit, ob Druckaufträge an den Drucker weitergeleitet oder Threads, die den Drucker verwenden wollen, blockiert werden:

```
public class PrinterDriver {
    private boolean online = false;
    public synchronized void print(String s) {
        while (!online) {
            try { wait(); }
            catch(InterruptedException ex) { return; }
        }
        ... // send s to printer
    }
    public synchronized void onOff() {
        online = !online;
        if (online) notifyAll();
    }
    ...
}
```

Die Methode `print` stellt sicher, dass `online` den Wert `true` hat, bevor das Argument an den Drucker weitergeleitet wird. Andernfalls wird `wait` aufgerufen. Diese in `Object` vordefinierte Methode blockiert (bei freigegebenem Lock) den aktuellen Thread so lange, bis er wieder aufgeweckt wird, oder mit einem entsprechenden Argument für eine bestimmte Zeit. Die Überprüfung erfolgt in einer Schleife, da nach Aufwecken des Threads über `notifyAll` in `onOff` durch einen weiteren Aufruf von `onOff` die Bedingung schon wieder verletzt sein kann, bevor der Thread an die Reihe kommt. Es ist immer, auch ohne Grund, damit zu rechnen, dass ein Thread aus dem Wartezustand aufwacht. Daher erfolgen solche Überprüfungen fast immer in Schleifen. Ebenso muss die Ausnahme `InterruptedException` abgefangen werden, die vom System bei vorzeitiger Beendigung des wartenden Threads ausgelöst wird.

Wie wir in Abschnitt 2.5 gesehen haben, laufen nebenläufige Threads in einer Methode namens `run` meist in einer Endlosschleife. Objekte der folgenden Klasse erzeugen nach Aufruf von `run` immer wieder neue Zeichenketten und schicken diese an den im Konstruktor festgelegten Druckertreiber:

```
public class Producer implements Runnable {
    private PrinterDriver t;
    public Producer(PrinterDriver t) { this.t = t; }
    public void run() {
        String s = ....
        for (;;) {
            ...         // produce new value in s
            t.print(s); // send s to the printer server
        }
    }
}
```

Das vordefinierte Interface `Runnable` spezifiziert nur `run`. Objekte von Klassen wie `Producer`, die `Runnable` implementieren, können wie in folgendem Codestück zur Erzeugung neuer Threads verwendet werden:

```
PrinterDriver t = new PrinterDriver(...);
for (int i = 0; i < 10; i++) {
    Producer p = new Producer(t);
    new Thread(p).start();
}
```

Jeder Aufruf von `new Thread(p)` erzeugt einen neuen Thread, der nach Aufruf von `start()` zu Laufen beginnt. Der Parameter `p` ist ein Objekt von `Runnable`; der Aufruf von `start()` bewirkt die Ausführung von `p.run()` im neuen Thread. Im Beispiel produzieren zehn Objekte von `Producer` ständig neue Zeichenketten und schicken sie an denselben Druckertreiber, der nebenläufige Zugriffe auf den Drucker synchronisiert. Objekte von `Thread` bieten viele Möglichkeiten zur Kontrolle der Ausführung des Threads, beispielsweise zum Abbrechen, kurzfristigen Unterbrechen, und so weiter. Beachten Sie, dass einige dieser Methoden veraltet („deprecated“) sind und nicht mehr verwendet werden sollten.

5.3.2 Nebenläufigkeit in der Praxis

Die grundlegenden Sprachkonzepte für die nebenläufige Programmierung werden nur selten verwendet. Gründe sind einerseits prinzipielle Schwierigkeiten im Umgang mit Nebenläufigkeit, andererseits eine Reihe vorgefertigter Lösungen für die häufigsten Aufgaben, die die nebenläufige Programmierung auf eine höhere Ebene verschieben.

Die wichtigsten vorgefertigten Lösungen finden wir in diesen Java-Paketen: `java.util.concurrent` und `java.util.concurrent.atomic`. Zum Teil handelt es sich um gut durchdachte und effiziente Implementierungen von Programmteilen, die wir mit entsprechendem Wissen selbst schreiben könnten, zum Teil (vor allem in `java.util.concurrent.atomic`) aber auch um Lösungen, die heute übliche Hardwareunterstützung für Synchronisation nutzbar machen. Dahinter stecken bekannte Verfahren im Umgang mit Nebenläufigkeit und Parallelität, die in fortgeschritteneren Lehrveranstaltungen thematisiert werden. Wir wollen nur exemplarisch einige wenige Möglichkeiten aufzeigen.

Aufgaben und Threads. Vor allem aus der funktionalen Programmierung mit Nebenläufigkeit stammt ein Konzept namens *Future*. Das ist eine Variable, in der das Ergebnis einer Berechnung abgelegt wird. Das Besondere daran ist, dass die Berechnung, die dieses Ergebnis liefert, im Hintergrund abläuft, während im Vordergrund gleichzeitig andere Berechnungen durchgeführt werden. Nachdem die Berechnung im Hintergrund fertig ist, kann das Ergebnis ganz normal aus der Variablen gelesen werden. Wenn wir von der Variablen lesen bevor das Ergebnis der Hintergrundberechnung vorliegt, wird der lesende Thread so lange blockiert, bis das Ergebnis da ist. Wir haben also eine sehr einfache Möglichkeit, um eine Hintergrundberechnung anzustoßen und mit den Berechnungen im Vordergrund zu synchronisieren. Das geht nur, wenn die Hintergrundberechnung unbeeinflusst von anderen Berechnungen abläuft. In Java gibt es dafür die Klasse `FutureTask` und das Interface `Future` im Paket `java.util.concurrent`.

Generell müssen in der nebenläufigen Programmierung oft verschiedenste Aufgaben (*Tasks*) erledigt werden, die unabhängig voneinander irgendwann (ohne vorgegebene Zeitpunkte), aber möglichst effizient ablaufen sollen. Die Darstellung einzelner Elemente in einem Webbrowser ist ein Beispiel dafür. Aus Effizienzgründen ist es häufig nicht sinnvoll, für jede dieser manchmal kleinen Aufgaben einen eigenen Thread zu erzeugen, aber eine reine Hintereinanderausführung würde die Hardware schlecht auslasten. In solchen Fällen kann ein `Executor` (Interface aus `java.util.concurrent`) sinnvoll sein. Je nach Implementierung des Executors werden die Aufgaben auf verfügbare Threads aufgeteilt. Es gibt mehrere standardmäßige Implementierungen von `Executor`, z. B. `ThreadPoolExecutor`. Über zahlreiche Parameter kann gesteuert werden, wann und wo welche Aufgaben auszuführen sind. Einige Implementierungen erlauben auch das regelmäßig wiederholte Ausführen bestimmter Aufgaben.

Java-8-Streams. Streams bieten eine effizente und einfache Möglichkeit für den Umgang mit großen Datenmengen, auch zusammen mit Nebenläufigkeit:

```
HashSet<String> nums = ...; // "1", "2", ...
int sum = nums.parallelStream()
              .mapToInt(Integer::parseInt)
              .reduce(0, (i, j) -> i + j);
```

Die in `nums` in Form von Zeichenketten dargestellten Zahlen werden zu `int`-Zahlen umgewandelt und mittels `reduce` aufaddiert. Wegen `parallelStream()` werden die Operationen auf dem Stream als Tasks über einen Thread-Pool (also unter Verwendung mehrerer Threads mithilfe von `ThreadPoolExecutor`) abgearbeitet, wobei wir uns nicht um Details kümmern müssen. Der Thread-Pool bestimmt die Anzahl der dabei verwendeten Threads; es wird also nicht für jedes Datenelement ein eigener Thread erstellt. Eine Voraussetzung ist, dass die einzelnen Elemente (wie ganz allgemein bei Verwendung von Strömen) unabhängig voneinander sind, also keine gemeinsamen Variablen haben. Die Aufteilung der Daten erfolgt im Hintergrund über den `Spliterator`. Wir können die Aufteilung beeinflussen, indem wir einen eigenen Spliterator implementieren,

hauptsächlich über die Methode `trySplit()` mit einigen anderen dazu passenden kleinen Methoden. Vordefinierte Klassen wie `HashSet` enthalten schon einen gut angepassten Spliterator. Methoden wie `map` (hier in der Variante `mapToInt`) operieren ohnehin nur auf jeweils einem Element, sodass Nebenläufigkeit keinen Unterschied macht. Methoden wie `sorted()` und `distinct()` erfordern spezielle Algorithmen für den Umgang mit Nebenläufigkeit, vor allem `distinct()` kann mit Nebenläufigkeit ineffizient werden. Auch abschließende Operationen müssen für Nebenläufigkeit ausgelegt sein. Lambdas in `reduce` müssen assoziativ sein. Dadurch kann jeder parallele Datenblock für sich reduziert werden, erst danach werden die Teilergebnisse über das gleiche Lambda zusammengefasst. Ein großer Teil der Komplexität von `collect` hat mit Nebenläufigkeit zu tun. Es reicht nicht, nur eine Datensammlung vorzugeben, in die Elemente eingefügt werden. Daher benötigt die Standardvariante von `collect` (ohne `Collector`) drei Lambdas als Parameter: Ein Lambda erzeugt eine neue Datensammlung, da pro Datenblock eine eigene Datensammlung benötigt wird. Das zweite Lambda fügt ein Element in die Datensammlung ein. Das dritte Lambda fügt zwei Datensammlungen der gleichen Art zu einer zusammen, wodurch bei wiederholter Anwendung am Ende nur eine Datensammlung entsteht. Über die Klasse `Collector` werden verschiedene vorgefertigte Varianten entsprechender Methoden (Lambdas) bereitgestellt, die uns von den Details abschirmen.

Thread-sichere Datenstrukturen. Eine Reihe von Klassen im Java-Paket `java.util.concurrent` stellt synchronisierte Varianten üblicher Datenstrukturen dar. So ähnelt `ConcurrentHashMap` einer normalen `HashMap`, erlaubt jedoch gleichzeitige Zugriffe mehrerer Threads. Tatsächlich ist `ConcurrentHashMap` sehr effizient wenn viele Threads gleichzeitig darauf zugreifen, da diese Implementierung ohne Locks auskommt. Es sind unbeschränkt viele gleichzeitige Lesezugriffe und eine einstellbare Zahl gleichzeitiger Schreibzugriffe erlaubt. Auch parallele Ströme und Spliteratoren sind darauf vergleichsweise effizient. Auf Objekte von `HashMap` darf dagegen, außer über parallele Ströme, nicht gleichzeitig von mehreren Threads aus zugegriffen werden (würde zu Fehlern führen), aber bei nur einem Thread ist `HashMap` natürlich effizienter. Es gibt noch weitere Varianten: Beispielsweise erzeugt

```
Collections.synchronizedMap(new HashMap(...))
```

eine über einen einfachen Lock synchronisierte Variante von `HashMap`, die von mehreren Threads aus sicher verwendbar ist. Solange Threads nur selten gleichzeitig zugreifen wollen, ist diese Variante effizienter als `ConcurrentHashMap`. In frühen Java-Versionen gibt es statt `HashMap` nur `Hashtable`. Diese Klasse wird heute selten verwendet, da `HashMap` einen größeren Funktionsumfang hat und ohne Nebenläufigkeit effizienter ist. Allerdings ist `Hashtable` von Haus aus synchronisiert und bietet bei Nebenläufigkeit ähnliche Effizienz wie `HashMap` synchronisiert über `synchronizedMap`. Aus praktischer Sicht sind oft die kleinen Unterschiede im Funktionsumfang ausschlaggebend dafür, welche Klasse wir einsetzen. Beispielsweise können wir in einem Objekt von `HashMap` auch

den Wert `null` ablegen, in einem Objekt von `Hashtable` aber nicht. Die Klasse `ConcurrentHashMap` ist aus Effizienzgründen hinsichtlich des Funktionsumfangs sehr stark an `Hashtable` angelehnt, nicht an `HashMap`.

Für die meisten Datenstrukturen gilt Ähnliches. Auf in `java.util` definierte Datenstrukturen dürfen meist nicht mehrere Threads gleichzeitig zugreifen, weil nicht synchronisiert wird. Durch Methoden wie `synchronizedMap` und `synchronizedList` in der Klasse `Collections` können diese Datenstrukturen mit Synchronisation ausgestattet werden. Allerdings sind diese Datenstrukturen bei gleichzeitigen Zugriffen durch viele Threads nur wenig effizient. Vor allem sind Iteratoren (im Gegensatz zu Spliteratoren[5]) über diesen Datenstrukturen bei Nebenläufigkeit nicht robust, das heißt, nach Änderungen der Datenstrukturen funktionieren sie nicht mehr vernünftig. In `java.util.concurrent` gibt es Varianten dieser Datenstrukturen, die auch bei gleichzeitigen Zugriffen durch viele Threads noch effizient sind und robustere Iteratoren bieten. Allerdings unterscheiden sich die auf Nebenläufigkeit ausgelegten Datenstrukturen in vielen Details von den Varianten aus `java.util`. Darin spiegelt sich die Tatsache wider, dass für die nebenläufige Programmierung andere Datenstrukturen und Algorithmen zum Einsatz kommen als in der sequentiellen Programmierung.

Vorgehensweise. Sowohl in der nebenläufigen als auch parallelen Programmierung lassen wir uns bei der Suche nach passenden Zerlegungen einer Aufgabe in Teilaufgaben meist davon leiten, ob die Teilaufgaben voneinander unabhängig sind. Wenn die Teilaufgaben nicht voneinander abhängen, sind parallele Ströme, aber auch `Executor` sinnvoll mit guter Effizienz einsetzbar. Wenn Teilaufgaben auf gemeinsame Daten zugreifen, führt das nicht nur zu Ineffizienz wegen der nötigen Synchronisation, sondern es steigt auch die semantische Komplexität und damit die Fehlerwahrscheinlichkeit gewaltig an.

Oft lässt sich das Ziel der Unabhängigkeit nicht ganz erreichen. Dann müssen wir für möglichst wenige gleichzeitige Zugriffe, vor allem Schreibzugriffe auf gemeinsame Daten sorgen. Klassen wie `ConcurrentHashMap` können in diesem Fall einen Ausweg bieten. Allerdings muss sichergestellt sein, dass die gemeinsamen Daten keine Einschränkungen in der Ausführungsreihenfolge der Teilaufgaben bedingen, die Daten also nicht auf zu komplexe Weise voneinander abhängen. Wir wissen ja nicht, wann genau welche Teilaufgabe ausgeführt wird.

Sind Abhängigkeiten in der Ausführungsreihenfolge unvermeidlich, wird es schwierig. Wir können versuchen, Struktur in die Einschränkungen der Ausführungsreihenfolge zu bringen, sodass Klassen wie `Phaser` einsetzbar werden, die es erlauben, Teilaufgaben in mehreren Phasen auszuführen. Sobald wir auf solche Formen der Synchronisation angewiesen sind, können wir die Zuteilung der Teilaufgaben an Threads nicht mehr einem vordefinierten `Executor` überlassen, sondern müssen uns selbst darum kümmern. Jeder Versuch, die Ausführungsreihenfolge zu kontrollieren, lässt den Schwierigkeitsgrad rasant ansteigen. Auch wenn wir die zahlreichen Synchronisationsmöglichkeiten auf höherer Ebene nüt-

[5] Spliteratoren werden meist nur dadurch robust, dass Änderungen der Datenstrukturen verboten sind, während darüber iteriert wird.

zen, ist bald die gleiche oder eine höhere Komplexität erreicht, als wir durch den Einsatz der primitiven Sprachkonstrukte für Nebenläufigkeit hätten. Ganz überflüssig sind die primitiven Sprachkonstrukte daher noch nicht.

Ein Ausweg steht offen: Wenn es zu schwierig wird, eine Aufgabe über Nebenläufigkeit zu lösen, können wir sie sequentiell lösen. Obwohl es stark vereinfachend klingt, liegt im Kern dieser Aussage viel Potential für eine gute Zerlegung einer Aufgabe in Teilaufgaben. Logische Handlungsstränge lassen sich gleichzeitig und trotzdem sequentiell abarbeiten, indem jeder Handlungsstrang in eine Folge von einzelnen Tätigkeiten zerlegt wird. Die Tätigkeiten werden beispielsweise als Daten in je einer Queue pro Handlungsstrang abgelegt. Zur sequentiellen Abarbeitung werden die jeweils ersten Tätigkeiten in den Queues ausgeführt, wobei jene Queues übersprungen werden, deren erste Tätigkeiten noch nicht zur Ausführung bereit sind (etwa noch auf das Eintreffen von Daten warten). Diese Vorgehensweise lässt sich auf vielfältige Weise verbessern, etwa dadurch, dass mehrere Threads die jeweils ersten Tätigkeiten in Queues bearbeiten, sofern sie unabhängig voneinander sind, oder einige Queues gegenüber anderen bevorzugt werden. Im Wesentlichen machen alle Mechanismen für Nebenläufigkeit auf höherer Ebene genau das, jeweils auf etwas unterschiedliche Weise und auf unterschiedlichen Annahmen beruhend (z. B. müssen Daten für viele Mechanismen unabhängig voneinander sein). Wenn wir es nicht (ohne unnatürliche Verrenkungen wie extrem aufwändige Synchronisation) schaffen, die einem bestimmten Mechanismus zugrunde liegenden Annahmen zu erfüllen, müssen wir auf diesen Mechanismus verzichten. Vielleicht können wir aber einen anderen Mechanismus mit anderen Annahmen einsetzen. Wenn wir keinen problemlos einsetzbaren Mechanismus finden, bleibt noch immer der hier skizzierte Ausweg über eine sequentielle Abarbeitung der Tätigkeiten in den Handlungssträngen. Oft reicht es schon, uns diesen letzten Ausweg bewusst vor Augen zu führen, um zu erkennen, wo es bei Einsatz einer bestimmten Technik hakt, wodurch sich manchmal eine nebenläufige Lösung fast von selbst aufdrängt, manchmal aber auch der Verzicht auf Nebenläufigkeit nahegelegt wird.

Faustregel: Synchronisation muss einfach gehalten werden.

Wenn die nötige Synchronisation zu aufwändig wäre, würde Nebenläufigkeit weder hinsichtlich der Nutzung paralleler Hardware noch hinsichtlich der Einfachheit der gesamten Programmstruktur irgendwelche Vorteile bringen. Statt übermäßig aufwändig zu synchronisieren ist es fast immer besser, auf Nebenläufigkeit zu verzichten. Gerade der Verzicht auf Nebenläufigkeit in einem Bereich kann dazu führen, dass Nebenläufigkeit in einem anderen Bereich effektiv nutzbar wird. Schließlich kommt es nur auf Abhängigkeiten zwischen Daten und darauf ausgeführten Tätigkeiten an. Manche Bereiche in einem Programm können frei von Abhängigkeiten bleiben, wenn andere Bereiche sich darum kümmern.

5.3.3 Synchronisation und die objektorientierte Sicht

Umgang mit Synchronisationsproblemen. Java hat zwar schon von Anfang an Nebenläufigkeit unterstützt, aber im Laufe der Zeit hat sich diesbezüglich

viel verändert. Das hat zu kleinen Inkonsistenzen geführt, auf die wir beim praktischen Programmieren besonders achten müssen. Beispielsweise kümmern sich die Klassen `Vector` und `Hashtable` selbst um Synchronisation, die ähnlichen Klassen `LinkedList` und `HashMap` aber nicht. Wir müssen das Synchronisationsverhalten der von uns verwendeten Klassen gut kennen, da eine Fehlannahme in jede Richtung zu schwerwiegenden Fehlern im Programmablauf führen kann.

Zu viel Synchronisation macht sich folgendermaßen negativ bemerkbar: Die gleichzeitige Ausführung von Threads wird verhindert und die Laufzeit des Programms vielleicht (aber nicht bei jeder Ausführung gleich) verlängert. In Extremfällen wird die Ausführung so stark verzögert, dass überhaupt kein Fortschritt mehr möglich ist. Gefürchtet sind Liveness-Probleme wie Deadlock, Livelock und Starvation. Eine übliche Technik zur Vermeidung von Deadlocks besteht in der Verhinderung von Zyklen beruhend auf einer linearen Anordnung aller Objekte im System; Locks dürfen nur in dieser Reihenfolge angefordert werden, das heißt, wenn wir in einer `synchronized`-Methode in einem Objekt y sind, dürfen wir keine `synchronized`-Methode in einem Objekt x aufrufen, wenn entsprechend der linearen Anordnung x vor y steht. Leider ist eine lineare Anordnung in der Praxis recht einschränkend: Dadurch werden alle Arten von zyklischen Strukturen verhindert, bei deren Abarbeitung Synchronisation nötig sein könnte. Nicht selten nehmen wir für solche Strukturen die Gefahr von Deadlocks in Kauf, ebenso wie die Gefahr von Livelocks und Starvation.

Manchmal wird empfohlen, Liveness-Eigenschaften wie nicht-funktionale Eigenschaften eines Programms zu behandeln. Das bedeutet, dass wir uns beim Schreiben des Programms zunächst nicht darum kümmern, sondern erst durch ausgiebiges Testen Verletzungen geforderter Eigenschaften zu finden versuchen. Treten beim Testen keine Probleme auf, wird die Eigenschaft als erfüllt angenommen. Allerdings ist es kaum möglich, Programme mit Nebenläufigkeit ausreichend intensiv zu testen, weil Probleme bei wiederholten Aufrufen unter einander ähnlichen Bedingungen häufig auch einander ähnliche Ergebnisse liefern, aber bestimmte Umgebungsbedingungen, die beim Testen nur schwer herstellbar sind (bestimmte Hardware, anderes Land, andere Betriebssystemversion, Kombination mit anderer Software, etc.), gravierende Fehler zeigen können.

Heute gibt es Werkzeuge (auch für Java, vorwiegend auf Model-Checking beruhend), die in der Lage sind, bestimmte Eigenschaften von Programmen formal zu beweisen. So ist es auch möglich zu beweisen, dass keine Deadlocks auftreten. Leider können diese Werkzeuge nicht alles. Bei größeren Programmen liefern sie manchmal gar kein Ergebnis (nicht genug Speicher) oder sie zeigen sehr viele Fehlersituationen auf, deren genauere Analyse zeigt, dass angenommene Zustände in der Praxis nicht auftreten können. Da Livelocks und Starvation viele unterschiedliche Ursachen und Auswirkungen haben, gibt es dafür auch keine klaren formalen Definitionen, sodass sie formalen Beweisen kaum zugänglich sind. Daher bleibt nur das Testen, trotz aller Schwierigkeiten.

Vorgefertigte Lösungen für die nebenläufige Programmierung beruhen großteils auf bekannten Techniken, die nicht oder kaum anfällig für Verletzungen der Liveness-Properties sind. Genau deswegen sollten wir sie bevorzugen. Sie wurden ausgiebig getestet, auch im praktischen Einsatz in unzähligen Program-

men. Trotzdem können auch bei Verwendung vorgefertigter Klassen Liveness-Properties verletzt sein. Solche Probleme treten ja nicht nur an einzelnen Programmstellen auf, sondern resultieren aus dem Zusammenspiel unterschiedlicher Programmteile. Wir müssen das gesamte Programm betrachten, nicht nur kleine Teile. Am besten folgen wir Empfehlungen, die aus Erfahrungen resultieren. Empfehlungen zur nebenläufigen Programmierung in Java im herkömmlichen Stil sind in [23] zu finden, aktuellere Programmiertechniken in [13].

Objektorientierte Sicht. Das von Java unterstützte Basiskonzept für Nebenläufigkeit, das *Monitor-Konzept,* ist schon recht alt [14, 6] und wurde nur leicht verändert, um es an Java anzupassen. Objektorientierte Programmiertechniken werden kaum unterstützt: Synchronisation wird weder als zu Objektschnittstellen gehörend betrachtet, noch in Untertypbeziehungen berücksichtigt – abgesehen von Zusicherungen, um die wir uns selbst kümmern müssen. Vorgefertigte Lösungen haben zwar eine Verschiebung hin zu eher funktionalen Denkweisen bewirkt, an der Problematik im Grunde aber nichts geändert. Auch heute ist es schwierig, gute objektorientierte Programme mit Nebenläufigkeit zu schreiben.

Um Synchronisation in Untertypbeziehungen einzubeziehen, müssen wir vor allem Client-kontrollierte History-Constraints berücksichtigen. Synchronisation bewirkt ja Einschränkungen auf der Reihenfolge, in der Methoden abgearbeitet werden. Genau solche Einschränkungen werden durch Client-kontrollierte History-Constraints dargestellt. Das impliziert, dass Objekte von Untertypen Nachrichten zumindest in allen Reihenfolgen verarbeiten können müssen, in denen Objekte von Obertypen sie verarbeiten können. Andernfalls wäre das Ersetzbarkeitsprinzip verletzt. Das heißt auch, dass Methoden in Untertypen über `wait`, `notify` und `notifyAll` nicht stärker synchronisiert sein dürfen als entsprechende Methoden in Obertypen. Nur wenn bereits die Methode im Obertyp in einer bestimmten Situation `wait` aufruft, darf das auch die Methode im Untertyp, muss aber nicht. Wenn die Methode im Obertyp in einer bestimmten Situation `notify` bzw. `notifyAll` aufruft, dann muss dies auch die Methode im Untertyp machen; sie darf diese Methoden auch in anderen Situationen aufrufen. Konkrete Situationen für Synchronisation sind nicht in Schnittstellen beschrieben und dort manchmal kaum beschreibbar. Daher kann es schwierig sein, sich an diese Bedingungen zu halten. Dort wo komplexe Synchronisation notwendig ist, sollten wir deswegen unter den heutigen Gegebenheiten auf Ersetzbarkeit verzichten. Die wichtigste Empfehlung bei der Planung der Synchronisation ist daher, diese so lokal und einfach wie möglich zu gestalten.

Abhängigkeiten, die durch die notwendige Synchronisation in die Software eingeführt werden, stehen auch der Vererbung oft im Weg [26]. Dafür gibt es zwar einige Lösungsansätze, die aber allesamt nicht überzeugen können, vor allem weil zumindest einige davon in Widerspruch zur Ersetzbarkeit stehen.

Ein weiteres Problem ergibt sich daraus, dass wir bei der Faktorisierung der Software nach objektorientierten Gesichtspunkten anders vorgehen müssen als bei der Zerlegung von Aufgaben im Hinblick auf Nebenläufigkeit. Wenn wir die objektorientierten Gesichtspunkte in den Mittelpunkt stellen, ergeben sich häu-

fig so starke Abhängigkeiten zwischen den Teilaufgaben, dass Nebenläufigkeit kaum sinnvoll einsetzbar ist. Andererseits führt eine Zerlegung nach Gesichtspunkten der Nebenläufigkeit leicht zu sehr niedrigem Klassenzusammenhalt und längerfristig hohem Wartungsaufwand wegen nötiger Refaktorisierungen.

5.4 Prozesse und Interprozesskommunikation

Prozesse werden vom Betriebssystem verwaltet (siehe Abschnitt 2.4.2). Die Erzeugung von Prozessen sowie die Interprozesskommunikation kann nur abhängig vom Betriebssystem betrachtet werden. Wir beziehen uns hier auf Linux. Viele Betriebssysteme haben wie Linux ihre Wurzeln in Unix, sodass sich die Unterschiede in Grenzen halten. Wir beschäftigen uns zunächst mit dem Erzeugen neuer Prozesse in Linux von einer Shell aus. Danach betrachten wir den Umgang mit Dateien und Pipelines aus Java-Sicht. Wir beschränken uns auf die einfachsten Formen der Interprozesskommunikation.

5.4.1 Erzeugen von Prozessen in einer Shell

Unter Linux arbeiten wir üblicherweise mit einem Terminal-Fenster, in dem „bash“ als Kommandozeileninterpreter (kurz „Shell“ genannt) läuft, um Kommandos in den Computer einzugeben. Das erste Terminal-Fenster mit Shell, das wir beim Einloggen bekommen, wird (in Anlehnung an die Steuerkonsole früher Rechnergenerationen) häufig „Konsole“ genannt. Zum Starten einer Programmausführung tippen wir den Programmnamen ein und schließen die Eingabe mittels „Enter“ (oder „Return“) ab. Die Shell sucht darauf hin in den ihr bekannten Verzeichnissen nach einer ausführbaren Datei dieses Namens und erzeugt einen neuen Prozess, der den Programmtext in der Datei in den Hauptspeicher lädt und ausführt. Statt durch ihren Namen können wir die ausführbare Datei auch durch ihren Dateipfad bestimmen. Beispielsweise führen `ls` und `/usr/bin/ls` das Programm in der gleichen Datei aus (um den Inhalt des Arbeitsverzeichnisses am Bildschirm auszugeben). Auch relative Dateipfade sind erlaubt, beispielsweise `./ls` wenn das Arbeitsverzeichnis `/usr/bin` ist.

Nach dem Programmnamen oder Dateipfad können, durch White-Space (also Leerzeichen oder Tab-Zeichen) getrennt, beliebig viele Kommandozeilenargumente (und Flags bzw. Optionen, die auch nur Argumente sind) folgen, die als Zeichenketten an den neuen Prozess übergeben werden. Es hängt vom ausgeführten Programm ab, ob und wie diese Argumente in die Programmausführung einfließen. Z. B. wird das Kommando `java Test arg1 arg2` den Java-Interpreter (`java`) starten, an den die Kommandozeilenargumente `Test`, `arg1` und `arg2` übergeben werden; der Interpreter wird `Test` als Namen der übersetzten Java-Klasse (also `Test.class`) verstehen, in den dem Interpreter bekannten Verzeichnissen nach einer Datei dieses Namens suchen und die Methode `main` in dieser Klasse ausführen, wobei die restlichen Kommandozeilenargumente `arg1` und `arg2` Einträge eines Arrays sind, das als Argument an `main` übergeben wird. Das Java-Programm bestimmt, ob und wie dieses Array verwendet wird.

Der Java-Interpreter führt übersetzte Java-Programme also auf ähnliche Weise aus, wie die Shell ausführbare Programme zur Ausführung bringt. Es gibt aber wesentliche Unterschiede: Die Shell erzeugt für die Programmausführung einen neuen Prozess, der die Programmausführung bestmöglich von der Ausführung anderer Prozesse auf dem gleichen Computer abschirmt und davon ausgeht, dass die ausführbare Datei Maschinencode enthält, der direkt vom Prozessor verstanden wird. Der Java-Interpreter läuft hingegen in dem Prozess, der beim Start des Interpreters erzeugt wurde und interpretiert das übersetzte Java-Programm selbst, mehrere vom Interpreter geladene Java-Klassen und im Interpreter erzeugte Threads sind nicht voneinander abgeschirmt. Moderne Java-Interpreter enthalten zwar selbst kleine Compiler (JIT- bzw. Just-in-Time-Compiler), die Teile des JVM-Codes in Maschinencode übersetzen, aber am Prinzip ändert sich nichts, alle geladenen Klassen und Threads existieren noch immer (ohne Abschirmung voneinander) im gleichen Prozess.

Für die einfache Kommunikation mit der Außenwelt bekommt jeder Prozess automatisch drei Ein- und Ausgabekanäle zugeordnet: die Standardeingabe, die Standardausgabe und die Fehlerausgabe. Während ein durch eine Shell gestarteter Prozess läuft, wird alles, was wir in die Shell eintippen, an die Standardeingabe des Prozesses weitergeleitet. Alles, was der Prozess in die Standard- und Fehlerausgabe schreibt, wird durch die Shell im Terminal-Fenster angezeigt. In einem Java-Programm wird die Standardeingabe als `System.in` angesprochen, die Standardausgabe als `System.out` und die Fehlerausgabe als `System.err`. Diese Ein- und Ausgabekanäle werden häufig umgeleitet (redirect), also statt mit der Shell mit einer anderen Quelle oder einem anderen Ziel verbunden. Um das zu bewerkstelligen, versteht die Shell eine Reihe von Symbolen bzw. Operatoren. Hier ist eine kleine Auswahl an Beispielen basierend auf dem Programm `cat` (für andere Programme funktioniert es nach dem gleichen Schema), Detailinformation bekommen wir etwa durch `man cat` und `man bash` (bzw. `man sh` oder `man csh`, etc., je nach verwendeter Shell):

cat file `cat` liest Inhalt der über das Kommandozeilenargument festgelegten Datei `file` und gibt ihn in die Standardausgabe aus, keine Umleitung.

cat <file erreicht gleiches Ergebnis wie voriges Beispiel auf andere Weise: `cat` hat kein Kommandozeilenargument und liest daher aus der Standardeingabe, Standardeingabe ist auf den Inhalt der Datei `file` umgeleitet, dieser Inhalt wird in die Standardausgabe ausgegeben.

cat file >file2 Standardausgabe wird zur Datei `file2` umgeleitet. Daher wird keine Ausgabe im Terminal-Fenster angezeigt, sondern die Ausgabe in die neu erzeugte Datei `file2` geschrieben. Danach enthält `file2` eine Kopie von `file`. Fehlermeldungen (etwa wenn `file` nicht existiert) werden im Terminal-Fenster angezeigt (weil nur die Standardausgabe, nicht die Fehlerausgabe umgeleitet wird). `file` und `file2` dürfen nicht gleich sein.

cat file >>file2 wie im vorigen Beispiel, abgesehen davon, dass die Ausgabe hinten an `file2` angehängt wird, falls `file2` schon existiert.

cat file &>file2 so wie >, aber sowohl die Standard- als auch die Fehlerausgabe wird in die neue Datei file2 umgeleitet.

cat file &>>file2 so wie >>, aber sowohl die Standard- als auch die Fehlerausgabe wird an die Datei file2 angehängt.

cat <file >file2 sowohl die Standardeingabe als auch die Standardausgabe ist umgeleitet.

In einer einzigen Kommandozeile können auch mehrere Prozesse gleichzeitig gestartet werden. Eine sehr effiktive Möglichkeit dazu bieten *Pipelines.* Beispielsweise startet cat file | wc einen Prozess für cat file und einen weiteren Prozess für wc, wobei beide Prozesse gleichzeitig (bei ausreichend vielen Recheneinheiten parallel, sonst überlappt bzw. pseudo-parallel) ausgeführt werden. Die Besonderheit bei Pipelines besteht darin, dass die Standardausgabe des ersten Prozesses mit der Standardeingabe des zweiten Prozesses verbunden ist. Im Beispiel gibt cat file den Inhalt von file in seine Standardausgabe aus und wc liest diesen Inhalt aus seiner Standardeingabe und gibt die Anzahl der Zeichen, Wörter und Zeilen in seine Standardausgabe aus, die mit dem Terminal-Fenster verbunden ist; der Inhalt der Datei erscheint nicht im Teminal-Fenster. In cat <file | wc >file2 liest cat den Inhalt von file aus seiner Standardeingabe und der Output von wc landet in der Datei file2.[6] Über mehrere Vorkommen von „|" können wir auch mehr als zwei Prozesse miteinander verknüpfen, wobei jeweil Standardausgaben mit Standardeingaben benachbarter Prozesse verbunden sind. Wenn wir Prozesse mit „|&" statt „|" verknüpfen, wird auch die Fehlerausgabe des links stehenden Prozesses mit der Standardeingabe des rechts stehenden Prozesses verbunden; davon wird eher selten Gebrauch gemacht, weil wir Fehlermeldungen (von allen in einer Pipeline miteinander verknüpften Prozessen) häufig im Terminal-Fenster sehen und nicht als Input weiterleiten wollen.

Während ein normaler von der Shell aus gestarteter Prozess (genannt *Vordergrundprozess*) läuft, ist die Standardeingabe des Prozesses mit dem Terminal-Fenster verbunden, damit wir über die Tastatur Eingaben machen können. Zu dieser Zeit kann die Shell keine weiteren Befehle von der Tastatur entgegennehmen. Weitere Befehle müssen warten, bis der Prozess beendet ist, unabhängig davon, ob der Prozess tatsächlich Eingaben über die Tastatur entgegennimmt. Wenn der Prozess keine Eingaben entgegennimmt (etwa weil die Standardeingabe umgeleitet ist), gibt es oft keinen Grund, auf die Beendigung des Prozesses zu warten. Wenn wir das Zeichen & an das Ende einer Kommandozeile setzen, etwa cat file &, starten wir einen *Hintergrundprozess.* Dabei wartet die Shell nicht auf die Beendigung des Prozesses, sondern ist sofort nach dem Start des Prozesses bereit, weitere Kommandos entgegenzunehmen. Die Standard- und Fehlerausgabe ist (wenn nicht umgeleitet) weiterhin mit dem Terminal-Fenster verbunden, wodurch sich die Ausgaben mehrerer Prozesse im gleichen Fenster

[6] Das ist eine abgekürzte Sprechweise. Ausführlich müssten wir von der Standardausgabe des Prozesses, der cat ausführt und dem Output des Prozesses, der wc ausführt, sprechen.

überlappen können. Programme, die zur Benutzerinteraktion eigene Fenster öffnen, werden üblicherweise als Hintergrundprozesse gestartet.

So wie beliebig viele Prozesse in mehreren Zeilen der Shell hintereinander gestartet werden können, können mehrere Kommandos auch durch „`;`“ voneinander getrennt in nur einer Zeile hingeschrieben werden, um mehrere Prozesse hintereinander zu starten (wobei sich ein eventuell am Ende vorhandenes „`&`“ auf die ganze Zeile bezieht). Es gibt weitere Möglichkeiten der Aneinanderreihung, die den *Return-Status* von Prozessen in die Steuerung einbeziehen. Am Ende der Ausführung gibt jeder Prozess als Return-Status einen ganzzahligen Wert zurück, den Wert 0 um eine fehlerfreie Ausführung anzuzeigen, einen anderen Wert um einen Fehler zu melden. In Java wird die Ausführung des Interpreters beispielsweise durch einen Aufruf von `System.exit(...)` beendet, wobei das Argument den Return-Status festlegt. Wir können in der Shell die Erzeugung von zwei Prozessen mittels „`&&`“ verknüpfen, wenn der zweite Prozess nur dann und erst dann erzeugt werden soll, wenn der erste Prozess mit einem Return-Status von 0 beendet wurde. Beispielsweise führt `cat file >file2 && wc <file2` zunächst den `cat`-Prozess aus und erst nachdem dieser Prozess erfolgreich beendet wurde und `file2` erzeugt hat, wird ein zweiter Prozess gestartet, der mittels `wc` die Wörter in `file2` zählt. Endet der erste Prozess mit einem Return-Status ungleich 0, wird der zweite Prozess gar nicht gestartet. Bei einer Verknüpfung mit „`||`“ statt „`&&`“ wird der zweite Prozess nur dann gestartet, wenn der erste Prozess einen Return-Status ungleich 0 geliefert hat.

Die Mächtigkeit der Shell wird durch *Expansion* von Pfadnamen erhöht. Damit können wir durch kurze Ausdrücke umfangreiche Listen von Pfadnamen als Kommandozeilenargumente erzeugen. Beispielsweise steht `cat *` für eine Kommandozeile, die nach `cat` die Liste aller Datei- und Verzeichnisnamen im aktuellen Arbeitsverzeichnis enthält, die Inhalte aller dieser Dateien werden ausgegeben. Konkret steht „`*`“ für beliebige Zeichenketten, die in üblichen Datei- und Verzeichnisnamen vorkommen, die also keine Trennzeichen wie White-Space, „`/`“, Klammern und Ähnliches enthalten. Das Zeichen „`?`“ steht für ein einziges solches Zeichen und `[abc]` für eines der drei Zeichen „`a`“, „`b`“ und „`c`“ (beliebige aufgezählte Zeichen), wobei auch *Ranges* wie in `[a-c]` vorkommen können (gleichbedeutend mit `[abc]`). So steht `A*[a-zI].java` für jede Datei im aktuellen Verzeichnis, deren Name mit „`A`“ beginnt und mit einem Kleinbuchstaben oder „`I`“ vor der Extension `.java` endet; der Name kann „`.`“ auch mehrfach enthalten. Auch längere Pfadnamen können angegeben werden, etwa `*/*/*.java` für alle Dateien mit der Endung `.java` in Unterverzeichnissen von Unterverzeichnissen vom Arbeitsverzeichnis. Alle diese Namen stehen in beliebiger Reihenfolge durch White-Space getrennt hintereinander, wodurch sie als Kommandozeilenargumente eines Prozesses dienen. Wir haben hier nur Beispiele gegeben. Es gibt zahlreiche weitere Formen der Expansion von Pfadnamen für unterschiedliche Zwecke. Durch Expansionen ergibt sich das Problem, dass Zeichen wie „`*`“ nicht mehr einfach nur für diesen Buchstaben stehen. Wenn wir nur den Buchstaben haben wollen, müssen wir ihn als „`*`“ darstellen. *Escape-Zeichen* wie „`\`“ können problematisch sein; in der Praxis ergeben sich häufig falsche Expansionen durch vergessene oder überzählige Escape-Zeichen.

Wir können einfache Hochkomma-Zeichen verwenden, um Zeichenketten vor ungewollter Expansion zu schützen. Beispielsweise steht `'a * 2'` einfach nur für ein einziges Kommandozeilenargument bestehend aus diesen 5 Zeichen, ganz ohne Expansion. Wie wir sehen, kann damit ein einzelnes Argument auch Leerzeichen enthalten, obwohl Leerzeichen normalerweise mehrere Argumente voneinander trennen. In doppelten Hochkomma-Zeichen wie in `"a * 2"` sind ebenso Leerzeichen einfach darstellbar, aber Zeichen wie „`*`" werden dennoch expandiert. Eine Besonderheit sind Kommandozeilenargumente in verkehrten einfachen Hochkomma-Zeichen, etwa `` `cat file` ``. Dabei wird ein Prozess erzeugt, der `cat file` ausführt und die Standardausgabe dieses Prozesses als Liste von Kommandozeilenargumenten betrachtet; jedes (durch White-Space getrennte) Wort in `file` wird dadurch zu einem Kommandozeilenargument. Daher gibt `` cat `cat file` `` die Inhalte aller Dateien aus, deren Namen in `file` stehen.

Die Shell ist mehr als nur ein einfacher Kommandozeileninterpreter, sie ist ein Interpreter für eine vollständige Programmiersprache. Jedes Kommando ist eine Anweisung in dieser Sprache. Die Shell kennt Variablen, vordefinierte Befehle, Kontrollstrukturen und sogar definierbare Funktionen. Viele Programme sind in der Sprache der Shell geschrieben, sogenannte Shell-Skripte (meist in Dateien mit der Endung `.sh`). Wir können Shell-Skripte als normale Textdateien schreiben, durch Änderung der Zugriffsrechte über das Dateisystem als „ausführbar" deklarieren (etwa durch `chmod u+x script.sh`) und wie ein normales Programm ausführen, das in einer neuen Shell die Kommandos des Skripts abarbeitet. Auch ohne ausführbar zu sein, können wir durch `. script.sh` dafür sorgen, dass die Befehle in `script.sh` in der aktuellen Shell ausgeführt werden. Alles, was in einem Shell-Skript stehen kann, kann auch direkt in die Kommandozeile getippt werden. Wir wollen hier nicht die gesamte Sprache betrachten, sondern nur einige wenige Beispiele.

Shell-Variablen sind beliebige Namen (außer reservierten Namen), denen wir einen Wert zuweisen. Einige Variablen haben spezielle Bedeutungen. Z. B. ist `PATH` eine Variable, die eine Liste von Pfadnamen enthält (getrennt durch „`:`"), in denen die Shell bei Programmaufrufen in der gegebenen Reihenfolge nach ausführbaren Programmen sucht. Wir können uns den Inhalt der Variablen durch `echo $PATH` (entspricht `echo ${PATH}`) anzeigen lassen. Ein „`$`" vor einem Variablennamen steht für den Inhalt der Variablen. `echo` ist ein Befehl der Shell (das bedeutet, zur Ausführung wird kein eigener Prozess erzeugt), der die Argumente in die Standardausgabe schreibt. Es wird also der Inhalt von `$PATH` im Terminal-Fenster angezeigt, etwa `/usr/local/bin:/bin:/usr/bin`. Nach Ausführung von `PATH=/home/me/bin:$PATH` (eine Zuweisung) ist der Inhalt von `$PATH` vorne um einen Dateipfad erweitert.[7] Die Menge aller Variablen und ihrer Werte wird durch `set` (ohne Argumente) angezeigt.

Bedingte Anweisungen `if ...; then ...; else ...; fi` kommen häufig vor und sind etwas flexibler einsetzbar, aber nicht ganz so einfach handhab-

[7]Diese Änderung wirkt sich nur lokal in der Shell aus, nicht in einer eventuell vorhandenen übergeordneten Shell. Um auch die übergeordnete Shell zu beeinflussen, müssten wir `export PATH=/home/me/bin:$PATH` schreiben.

bar wie die Alternativen „&&“ und „||“. Dabei ist die Bedingung nach `if` ein Prozess, dessen Return-Status 0 für Wahr und jeder andere Wert für Falsch steht. Nach `then` und `else` können beliebig viele Kommandos kommen. Statt „;“ ist überall auch ein Zeilenumbruch möglich. Als Bedingung wird häufig `test ...` verwendet, ein Programm, das über Kommandozeilenargumente viele unterschiedliche einfache Vergleiche durchführen kann. Beispielsweise gibt `test x = y` nur dann 0 zurück, wenn die Argumente `x` und `y` gleiche Zeichenketten sind. Weil das häufig vorkommt, gibt es für solche bedingte Anweisungen auch eine spezielle Syntax: `if [ x = y ]; then ...; fi` (der `else`-Zweig kann in jeder `if`-Anweisung dabei stehen oder weggelassen werden).

Wenn viele gleichartige Prozesse zu erzeugen sind, kommt die `for`-Schleife zum Einsatz. Ein Beispiel: `for i in *.java; do javac $i; done` übersetzt alle `.java`-Dateien im Arbeitsverzeichnis jeweils in einem eigenen Prozess, eine Datei nach der anderen. Dabei ist `i` eine Shell-Variable. In jedem Schleifendurchlauf erhält `i` den Wert eines Arguments nach `in`, wodurch es so viele Schleifendurchläufe wie `.java`-Dateien im Arbeitsverzeichnis gibt; in jedem Durchlauf ist der Wert von `i` ein anderer Dateiname. Nach `do` stehen beliebig viele Befehle, die nacheinander ausgeführt werden. Wir können auch Hintergrundprozesse starten: `for i in *.java; do (javac $i &); done` übersetzt Dateien gleichzeitig (und Fehlermeldungen werden unvorhersehbar überlappt). Die runden Klammern sind nötig, weil „&“ nur am Ende eines Kommandos stehen darf, um den Prozess im Hintergrund zu starten. Diese Klammern bewirken aber auch, dass jeder Hintergrundprozess in einer eigenen Shell läuft, sodass wir die Prozesse nicht über die Shell, in der wir `for` aufgerufen haben, kontrollieren können. Für Fälle, in denen Hintergrundprozesse in der gleichen Shell laufen sollen, gibt es das `coproc`-Kommando, das aber deutlich schwieriger handhabbar ist und auf das wir hier daher nicht näher eingehen. Folgende Variante lenkt Standard- und Fehlerausgaben in einzelne Dateien um:

```
for i in *.java; do (javac $i &>`basename $i .java`.out &); done
```

Dabei entfernt `basename` die Endung `.java` aus dem Dateinamen, die danach durch `.out` ersetzt wird; Fehlermeldungen bei der Übersetzung von `A.java` landen also in `A.out`. Schleifen sind nicht auf Dateipfade eingeschränkt. Nach `in` können beliebige Listen von Argumenten stehen, über die die Schleifenvariable laufen soll. Beispielsweise kann die Schleifenvariable über alle Wörter in einer Datei laufen: `` for f in `cat file`; do javac $f.java; done `` lässt `javac` auf jedem Namen in `file` (erweitert um die Endung `.java`) laufen.

5.4.2 Umgang mit Dateien und I/O-Strömen in Java

Der prinzipielle Umgang mit Kommandozeilenargumenten, üblichen Ein- und Ausgabekanälen und Dateien in Java sollte uns schon bekannt sein:

- Die Klassenmethode `main`, die den Startpunkt der Ausführung eines Java-Programms darstellt, nimmt ein Array von Zeichenketten als Parameter. Der erste, vom Java-Interpreter veranlasste Aufruf dieser Methode über-

gibt in diesem Array die Kommandozeilenargumente in der Reihenfolge, in der sie bei der Prozesserzeugung angegeben wurden.

- Aus der Standardeingabe ist über das Objekt vom Typ `InputStream` in der Variablen `System.in` lesbar. Häufig lesen wir nicht direkt daraus, sondern etwa über einen Scanner, der durch `new Scanner(System.in)` erzeugt wurde, oder aus einem durch `new ObjectInputStream(System.in)` erzeugten Strom (siehe unten).

- In die Standardausgabe und die Fehlerausgabe ist über je ein Objekt vom Typ `PrintStream` in den Variablen `System.out` und `System.err` schreibbar. Durch die benutzerfreundliche Schnittstelle wird häufig direkt in diese Ströme geschrieben, z. B. mittels `println`. Als Standardausgabe wird auch häufig in einen durch `new ObjectOutputStream(System.out)` erzeugten Strom geschrieben (siehe unten).

- Alle Arten von Dateien werden nach dem Öffnen auf relativ einheitliche Weise über diverse Arten von Strömen (nicht zu verwechseln mit Java-8-Streams, die auf andere Weise verwendet werden) gelesen und geschrieben und sollten nach der Verwendung auch wieder geschlossen werden. Beim Umgang mit Dateien können stets Ausnahmen ausgelöst werden, die abgefangen werden müssen. Die Ströme unterscheiden sich in ihrer Richtung (nur lesen oder nur schreiben), nach Ihrer Kodierung (Bytes als rohe Daten oder auf diverse Arten codierte Zeichen vom Typ `char`), nach der Pufferung (mit einem Zwischenspeicher ausgestattete gepufferte und direkt mit dem Betriebssystem verbundene ungepufferte Ströme) und nach den von ihnen unterstützten Zugriffsmethoden.

- Ungepufferte Ein- und Ausgabe wird gewählt, wenn es darauf ankommt, dass alle Daten rasch über das Betriebssystem verarbeitet werden, etwa wenn Output zeilenweise ohne Verzögerung für einen Benutzer sichtbar werden soll (weil vielleicht weiterer Input davon abhängt). Bei direkter Benutzerinteraktion trifft das meist zu. Gepufferte Ein- und Ausgabe ist dagegen insgesamt effizienter und wird in Situationen gewählt, in denen nur in Ausnahmefällen eine sofortige Ausgabe nötig ist. Das trifft zu, wenn große Datenmengen verarbeitet werden und Input meist nicht von davor erfolgtem Output abhängt. Ein Aufruf von `flash()` erzwingt die sofortige Ausgabe von Output. Spätestens am Programmende werden alle geschriebenen Daten an das Betriebssystem weitergeleitet. Viele Arten von Strömen sind ungepuffert. Über die Klassen `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader` und `BufferedWriter` werden ungepufferte Ströme zu gepufferten Strömen.

- Ein- und Ausgabekanäle können nur rohe Daten (Bytes) übertragen. Im Allgemeinen ist nicht spezifiziert, in welcher Form diese Daten kodiert sind. Innerhalb von Java sind Zeichen immer im UTF-16-Format dargestellt. Wenn Zeichen übertragen werden sollen, empfiehlt es sich, Ströme

der Typen `Readable` und `Writer` und deren Untertypen zu verwenden, die nötigenfalls automatisch für die Änderung der Kodierung vom bzw. zum extern verwendeten Format sorgen, wobei die externe Kodierung in der Regel bei der Erzeugung des Stroms angegeben werden muss. Aber auch Methoden in `PrintStream` und `Scanner` können für nötige Änderungen der Kodierung sorgen. Wenn keine Zeichenketten, sondern Daten anderer Art übertragen werden sollen, stehen Ströme der Typen `InputStream` und `OutputStream` und deren Untertypen zur Verfügung. Wir müssen selbst für passende Datenformate sorgen oder diese Aufgabe vordefinierten Klassen überlassen.

- Außer bei Zugriffen über `Scanner` und `PrintStream` ist es bei allen Verwendungen von Strömen nötig, die Ausnahme `IOException` (oder Untertypen davon) abzufangen. Da es schwierig ist, gleichzeitig Ausnahmen abzufangen und zu garantieren, dass Ströme am Ende der Verwendung wieder über `close` geschlossen werden, bietet sich die Verwendung der `try`-With-Resources-Anweisung an, die automatisch für das Schließen am Ende sorgt. Alle Klassen für die Ein- und Ausgabe implementieren das dafür nötige Interface `AutoCloseable`. Beispiel:

```
try (FileReader fr = new FileReader(path);
     BufferedReader br = new BufferedReader(fr)) {
    ... // use br, automatically closed at end of try block
}
catch (IOException ex) {
    ...
}
```

- Die Klasse `java.io.File` bietet umfangreiche Funktionalität zur Darstellung von Dateien und zur Verwendung von Verzeichnissen. Als Ergänzung bieten sich die Klassen im Paket `java.nio.file` an, die brauchbare Lösungen für sehr viele in der Praxis auftretende Aufgaben im Zusammenhang mit Dateien und Verzeichnissen bereitstellen.

Für den Umgang mit Dateien ist es unerlässlich zu wissen, in welchem Format eingelesene Daten dargestellt sind oder ausgegebene Daten dargestellt werden sollen. Wenn es sich um Dateien mit Texten (Zeichen) handelt, ist es vergleichsweise einfach, eine Kodierung für das externe Format anzugeben, beispielsweise über den Konstruktor von `InputStreamReader` oder `FileReader`. Es muss ein Standardname für die Kodierung als String angegeben werden, etwa `"UTF-8"` oder `"ISO-8859-1"`. Darauf kann verzichtet werden, wenn es sich um die Default-Codierung auf dem jeweiligen Betriebssystem handelt. Bei einer falsch angegebenen Kodierung sind einzelne Zeichen (vor allem Umlaute) nicht korrekt lesbar. Meist empfiehlt es sich, bei der Default-Kodierung zu bleiben, wenn alle Datenquellen und -ziele am gleichen Rechner liegen.

Wenn Daten nicht als Texte, sondern in einer anderen Form, etwa als beliebige Objekte vorliegen, wird es schwieriger. Es muss für die Umwandlung zwischen

der internen Darstellung und dem externen Format gesorgt werden. Manchmal ist es verlockend, von `toString` erzeugte Zeichenketten als externes Format anzusehen, weil dadurch keine weitere Methode zur Umwandlung des internen in ein externes Format nötig ist. Wenn das Ergebnis von `toString` alle Informationen enthält, die ein Leser dieser Daten zum Neuaufbau einer entsprechenden internen Datenstruktur benötigt, ist das durchaus machbar. Wir müssen in diesem Fall nur eine zusätzliche Methode (oder einen Konstruktor) schreiben, der aus der Zeichenkette wieder ein Objekt des gleichen Typs erstellen kann. Einer solchen Vorgehensweise stehen jedoch häufig zwei Schwierigkeiten gegenüber:

- Das Ergebnis von `toString` ist dafür vorgesehen, für Menschen gut lesbar zu sein und keine für diesen Zweck unnötige Information zu enthalten. Oft sind in dieser Darstellung nicht alle Informationen enthalten, die für einen Wiederaufbau eines Objekts nötig wären. Andererseits ist beschreibender Text vorhanden, der aus Sicht einer Maschine bedeutungslos ist und den Neuaufbau eines Objekts erschwert.

- Umwandlungen vom internen Format ins externe sowie vom externen ins interne müssen effizient erfolgen und die Menge an Daten, die zu übertragen sind, soll klein sein. Daher soll das Binärformat, das in den internen Daten vorliegt, im externen Format möglichst direkt erhalten bleiben. Vollständig erhalten bleiben kann das Format jedoch nicht, weil unterschiedliche Maschinen unterschiedliche Binärformate verwenden und Referenzen durch etwas anderes ersetzt werden müssen.

In der parallelen Programmierung sind Datenformate häufig einfach strukturiert, etwa Listen von Zahlen, die in jeweils 4 Bytes dargestellt sind. Wir können also jeweils vier Bytes in einem Byte-Strom als eine Zahl auffassen. Allerdings müssen wir festlegen, in welcher Reihenfolge die Bytes in einer Zahl liegen. Je nach Maschine gibt es andere natürliche Reihenfolgen (etwa Big-Endian versus Little-Endian). Welche Festlegung wir treffen ist egal, solange alle Maschinen, die sich Daten teilen, von der gleichen Festlegung ausgehen. Wir könnten die Umwandlungen über eigene Methoden selbst realisieren, aber mit vorgefertigten Klassen und Methoden geht es einfacher.

Die Umwandlung von Daten vom internen zum externen Format heißt *Serialisierung*, die Umwandlung vom externen zum internen Format *Deserialisierung*. In Java dient das Interface `Serializable` (das keine Methoden enthält) zum Kennzeichnen von Typen, deren Instanzen eine Form der automatischen Serialisierung und Deserialisierung unterstützen. Die Klassen `ObjectInputStream` und `ObjectOutputStream` erweitern `InputStream` und `OutputStream` um Funktionalität zur automatischen Deserialisierung und Serialisierung von Objekten, sodass wir uns nicht um Details der Darstellung kümmern müssen.[8] Die Methode `writeObject(...)` in `ObjectOutputStream` schreibt ein beliebiges Objekt in einer serialisierten Form als Folge von Bytes in die Ausgabe, vorausgesetzt das Objekt und alle im Objekt referenzierten Objekte sind vom Typ `Serializable`.

[8] Weitere Methoden dienen zum Serialisieren und Deserialisieren elementarer Typen.

Auf diese Weise erzeugte Byte-Folgen werden mittels `readObject()` aus einem Objekt von `ObjectInputStream` gelesen, wobei die Bytes automatisch wieder in Objekte der richtigen Typen umgewandelt werden, vorausgesetzt die richtigen Klassen dieser Objekte sind vorhanden. Das klingt sehr einfach und ist für entsprechend vorbereitete Objekte auch tatsächlich so einfach. In der Praxis funktioniert das aber nur für solche Objekte gut, die eine klar abgegrenzte Menge an Daten enthalten und nicht zu sehr mit dem Gesamtsystem verwoben sind. Wird ein Objekt serialisiert, werden automatisch auch alle vom Objekt referenzierten weiteren Objekte serialisiert; im Extremfall könnten das alle Objekte eines Systems sein. Wir müssen also darauf achten, dass das nicht passiert. Die serialisierbaren Daten müssen vom restlichen System möglichst gut getrennt bleiben. Generell werden als `static` oder `transient` deklarierte Variablen bei der Serialisierung nicht berücksichtigt. Das reicht aber nicht immer. Manchmal möchten wir steuern können, welche Daten in welcher Form serialisiert werden. Dazu ist es möglich, in den Klassen der zu serialisierenden Objekte Methoden mit vorgegebenen Signaturen (benannt `readObject`, `writeObject` und `readObjectNoData`) zu implementieren, um die eigentliche Arbeit zu erledigen.

Automatische Serialisierung und Deserialisierung über `ObjectInputStream` und `ObjectOutputStream` setzt voraus, dass auf beiden Seiten einer Datenverbindung Java eingesetzt wird und die gleichen Java-Klassen vorhanden sind. Ist das nicht gegeben, müssen wir eine eigene Darstellungsform finden, die von beiden Seiten verstanden wird. Wo es auf bestmögliche Effizienz ankommt, werden wir kaum ohne ein passendes Binärformat auskommen, das auf Besonderheiten der zu übertragenden Daten eingeht und die Datenmenge klein hält.

Häufig ist bestmögliche Effizienz bei der Datenübertragung nicht so wichtig, dagegen aber ein mehr oder weniger standardisiertes Format der übertragenen Daten von großer Bedeutung. Es kommen vor allem Formate zum Einsatz, die auf für Menschen lesbare Texte aufbauen und die Struktur der Texte so einschränken, dass auch Maschinen sie gut lesen können. Standardisiert und zum Datenaustausch im Internet weit verbreitet ist XML, ein Datenformat für *semistrukturierte* Daten, also Daten, die in der Lage sind, ihre eigene Struktur zu beschreiben. XML ist auf vielfältige Weise einsetzbar und wird durch eine große Palette an Werkzeugen in vielen unterschiedlichen Programmiersprachen gut unterstützt, die Menge der zu übertragenden Daten kann aber relativ groß sein. In letzter Zeit wird auch JSON (JavaScript-Object-Notation) häufig als Format für den Datenaustausch gewählt, nicht nur zusammen mit JavaScript, weil die Daten weniger umfangreich als in XML, aber trotzdem für Menschen gut lesbar sind. Wir wollen hier nur kurz erwähnen, dass derartige Formate natürlich auch von Java unterstützt werden und Informationen dazu im Internet leicht auffindbar sind, gehen aber nicht näher darauf ein.

Shell-Variablen der Shell, von der aus der Java-Interpreter gestartet wurde, sind von Java aus zugreifbar: Ein Aufruf von `System.getEnv()` gibt eine Datenstruktur vom Typ `Map<String, String>` zurück, die jede Shell-Variable (als Key) in den Wert abbildet, den diese Variable hat. Ebenso gibt es eine Methode `System.getEnv(...)`, der wir den Namen einer Shell-Variablen als String übergeben und die den Wert dieser Variablen zurückgibt (oder `null`

wenn dieser Variablen kein Wert zugeordnet ist). Neben Kommandozeilenargumenten bieten Shell-Variablen daher eine weitere Möglichkeit, Daten von der Shell aus an ein Java-Programm weiterzureichen. Shell-Variablen werden vor allem dann verwendet, wenn die zu übergebenden Daten über viele Aufrufe des Java-Interpreters hinweg gleich bleiben, während sich die Werte von Kommandozeilenargumenten meist von Aufruf zu Aufruf ändern.

Ein Aufruf der Methode `Runtime.getRuntime()` gibt das einzige Objekt von `Runtime` im aktuell ausgeführten Java-Interpreter zurück. Dieses Objekt bietet uns eine Schnittstelle zur Ablaufumgebung an. Beispielsweise lässt sich durch die Methode `availableProcessors()` die Anzahl der vom Java-Interpreter nutzbaren Prozessor-Kerne erfragen, und `freeMemory()` liefert die Größe des verbleibenden freien Speichers. Ein Aufruf von `gc()` hat in älteren Interpretern die Ausführung einer Garbage-Collection veranlasst, um nicht mehr zugreifbare Objekte aus dem Speicher zu entfernen; heute läuft ein Garbage-Collector fast immer im Hintergrund mit und der Aufruf ist nur eine Empfehlung an den Garbage-Collector, verstärkt nach freigebbarem Speicher zu suchen.

Für die parallele Programmierung besonders interessant ist die Methode `exec(...)`, verfügbar in mehreren Varianten, die einen neuen Prozess erzeugt (nicht nur einen Thread). Beispielsweise erzeugt

```
Process p = Runtime.getRuntime().exec("java -cp ~/java Test")
```

einen Prozess, der die übergebene Zeichenkette als Programmaufruf mit Kommandozeilenargumenten interpretiert. Hier wird ein neuer Java-Interpreter gestartet, der die übersetzte Java-Klasse `Test` im Verzeichnis `~/java` ausführt.[9] Über die Variable `p` vom Typ `Process` sind Verbindungen zum neuen Prozess herstellbar. So können wir mittels `p.getOutputStream()` auf einen Strom vom Typ `OutputStream` zugreifen, der über eine Pipeline mit der Standardeingabe des neuen Prozesses verbunden ist, durch den wir dem Prozess Daten übermitteln. Durch `p.getInputStream()` bekommen wir einen Strom, der mit der Standardausgabe des Prozesses verbunden ist, dessen Ausgabe wir lesen. Entsprechend steht `p.getErrorStream()` für den Strom vom Typ `InputStream`, über den wir die Fehlerausgabe lesen. Manchmal brauchen wir `p.waitFor()`, um auf die Beendigung des Prozesses zu warten; zurückgegeben wird der Return-Status. Mittels `p.destroy()` kann der Prozess abgebrochen werden.

5.4.3 Beispiel zu parallelen Prozessen

Hier ist ein Beispiel zur Demonstration der Prozesserzeugung und Interprozesskommunikation in Java. Wie in `Par` in Abschnitt 2.4.3 werden Primzahlen mittels „Sieb des Eratosthenes“ berechnet, diesmal jedoch über Prozesse, die über Pipelines kommunizieren. Ein Prozess, der als `controller` fungiert, startet eine vorgegebene Zahl an Prozessen, die als `worker` die eigentliche Rechenarbeit

[9] `-cp ~/java` ist eine Option, die der Java-Interpreter (als Kommandozeilenargument) versteht und die den *Class-Path* angibt, also das Verzeichnis, in dem nach einer Klasse entsprechenden Namens gesucht wird. Dabei steht `~` für das Home-Verzeichnis des aktuellen Users; `~user` würde für das Home-Verzeichnis des Users `user` stehen.

leisten und mit dem `controller` über die Standardein- und -ausgabe kommunizieren. Der `controller` führt das gleiche Java-Programm aus wie jeder `worker`, jedoch wird der `controller` ohne Kommandozeilenargumente gestartet, während jeder `worker` seine eindeutige Nummer (von 0 bis zur Anzahl der `worker`-Prozesse minus eins) als Kommandozeilenargument bekommt. Hier ist der Programmtext, in dem `main` anhand des Vorhandenseins eines Kommandozeilenarguments auf die Methoden `worker` und `controller` verzweigt:

```
import java.io.*;

public class ParProc {
  public static final long MAX = 1L << 30; // primes up to MAX
  public static final int PROC = 8; // how many worker processes
  private static final long[] prims = new long[60000000];
  private static int top = 2;

  private static void worker(int nr) throws IOException {
    boolean nothing = true; // no prime number found in cycle
    long limit = 9L; // no larger prime numbers checkable
    try (DataInputStream in = new DataInputStream(
                            new BufferedInputStream(System.in));
         DataOutputStream out = new DataOutputStream(System.out)
    ) {
      for (long n = 5L + 2 * nr; n <= MAX; n += PROC * 2) {
        while (n > limit) {
          if (nothing) {
            out.writeLong(limit + 1L);
            out.flush();
          }
          nothing = true;
          limit = prims[top++] = in.readLong();
          limit *= limit;
        }
        int i = 2; // check if n is a prime number
        for (long p = 3L; n % p != 0; p = prims[i++]) {
          if (p * p > n) { // n is a prime number
            out.writeLong(n);
            out.flush();
            nothing = false;
            break;
          }
        }
      }
      out.writeLong(MAX + 1L);
    }
  }
```

```
  private static void controller() throws IOException {
    final DataOutputStream[] outs = new DataOutputStream[PROC];
    final DataInputStream[] ins = new DataInputStream[PROC];
    final long[] delivered = new long[PROC];
    final long sqrtMax = (long) Math.sqrt(MAX);
    for (int i = 0; i < PROC; i++) {
      Process p = Runtime.getRuntime().exec("java ParProc " + i);
      outs[i] = new DataOutputStream(p.getOutputStream());
      ins[i] = new DataInputStream(new BufferedInputStream(
                                   p.getInputStream()));
    }
    for (int i=0; i<PROC; i++) delivered[i] = ins[i].readLong();
    boolean done = false;
    while (true) {
      int proc = 0;
      long min = delivered[0];
      for (int i = 1; i < PROC; i++)
        if (delivered[i] < min) min = delivered[proc = i];
      if (min > MAX) break;
      if ((min & 1) != 0) {
        if (!done) {
          for (DataOutputStream out : outs) {
            out.writeLong(min);
            out.flush();
          }
          if (min > sqrtMax) done = true;
        }
        prims[top++] = min;
      }
      delivered[proc] = ins[proc].readLong();
    }
    System.out.println(top);
  }

  public static void main(String[] args) throws IOException {
    prims[0] = 2L;
    prims[1] = 3L;
    if (args.length == 0) controller();
    else worker(Integer.valueOf(args[0]));
  }
}
```

Das Array `prims` enthält die in aufsteigender Reihenfolge bekannten Primzahlen, wobei 2 und 3 fix vorgegeben sind. Untersuchen wir zunächst, was ein `worker` macht: Wenn `PROC` den Wert 8 hat, dann untersucht der `worker` mit der

Nummer 0 nur die Zahlen 5, 21, 37, ... auf ihre Primzahleigenschaft, der `worker` mit der Nummer 1 die Zahlen 7, 23, 39, ... und so weiter, wodurch mit 8 `worker`-Prozessen alle ungeraden Zahlen ab 5 abgedeckt sind. Wurde eine Primzahl gefunden (durch Divisionsversuche durch alle Zahlen ab 3 in `prims` bis zur Wurzel der untersuchten Zahl), wird sie in die Standardausgabe geschrieben, von wo sie der `controller` liest und weiterverarbeitet. Als Grundlage kann der `worker` nur Zahlen verwenden, die in `prims` stehen. Aus der Standardeingabe können bei Bedarf weitere Primzahlen gelesen und in das Array eingefügt werden; der `controller` versorgt die `worker`-Prozesse mit entsprechendem Input. Die Variable `limit` enthält das Quadrat der größten bisher in `prims` vorhandenen Primzahl; mit dem Sieb sind nur Primzahlen kleiner als `limit` ermittelbar. Bevor untersucht werden kann, ob eine Zahl `n` eine Primzahl ist, muss `n <= limit` sichergestellt sein und bei Bedarf eine weitere Primzahl aus der Standardeingabe gelesen und zu `prims` hinzugefügt werden, wodurch auch `limit` vergrößert wird. Natürlich kann `controller` nur Primzahlen verschicken, die vorher von einem `worker`-Prozess ermittelt wurden. Es könnte passieren, dass `controller` und `worker` gegenseitig auf das Eintreffen von Daten warten. Damit das nicht passiert, schreibt der `worker` eine Information in die Standardausgabe, anhand der der `controller` feststellen kann, bis zu welcher Zahl schon alle Primzahlen gemeldet wurden. Jede ausgegebene Primzahl ist eine solche Information. Wenn der `worker` seit der letzten aus der Standardeingabe gelesenen Primzahl keine Primzahl gemeldet hat, gibt er `limit + 1` aus – eine gerade Zahl größer 2, sicher keine Primzahl. Sobald alle Zahlen bis `MAX` überprüft sind, wird `MAX + 1` in die Standardausgabe geschrieben und der Prozess beendet.

Der `controller` erzeugt `worker`-Prozesse, liest von den `worker`-Prozessen gelieferte Daten, bereitet daraus eine sortierte Folge von Primzahlen, legt diese in `prims` ab und gibt sie an `worker` weiter. Neue Prozesse werden mittels `Runtime.getRuntime().exec(...)` erstellt, wobei das Argument von `exec` den vom Prozess ausgeführten Programmaufruf darstellt. Es wird ein Java-Interpreter gestartet, der `ParProc` mit `i` als Kommandozeilenargument ausführt. `ParProc.class` muss im Arbeitsverzeichnis liegen. Die Arrays `outs` und `ins` enthalten Eingabe- und Ausgabeströme, die jeweils mit der Standardein- und -ausgabe eines `worker`-Prozesses über eine Pipeline verbunden sind. Das Array `delivered` enthält zuletzt aus diesen Eingabeströmen gelesene Werte, also solche, die `worker`-Prozesse in ihre Standardausgaben geschrieben haben. In einer Schleife wird das Array mit den ersten Werten initialisiert. Nun werden Primzahlen in sortierter Reihenfolge verarbeitet. Dazu wird ständig wiederholt

- nach dem kleinsten Wert `min` in `delivered` gesucht,
- die Schleife abgebrochen falls `min > MAX` ist (dann sind wir fertig),
- `min` an alle `worker`-Prozesse geschickt (über Ströme in `outs`) und in `prims` abgelegt falls `min` ungerade ist (`(min & 1) != 0`),
- der gerade bearbeitete Wert in `delivered` durch den nächsten Wert aus dem Strom ersetzt.

Zur Vermeidung unnötiger Kommunikation wird mit Hilfe der Variablen `done` sichergestellt, dass an die `worker`-Prozesse nur eine einzige Primzahl größer als `sqrtMax` (der Wurzel aus `MAX`) verschickt wird. Primzahlen oberhalb von `sqrtMax` werden zur Primzahlberechnung nicht benötigt, aber zumindest eine Zahl größer als `sqrtMax` wird von `worker`-Prozessen benötigt, um zu erkennen, dass schon alle nötigen Primzahlen vorliegen. Zur Vereinfachung gibt `controller` nur die Zahl der gefundenen Primzahlen aus, aber alle Primzahlen im gewünschten Wertebereich liegen vor und können weiterverarbeitet werden.

Aus diesem Beispiel können wir Folgendes lernen:

- Eine Schwierigkeit besteht darin, die Kommunikation zwischen den Akteuren so zu organisieren, dass es zu keinen Programmzuständen kommt, in denen Akteure unendlich lange auf Daten von anderen Akteuren warten. Wir müssen das Programmverhalten genau analysieren, um das sicherzustellen; einfache Argumentationsketten sind meist nicht zielführend. Schon kleine Änderungen können bedeutende Auswirkungen haben.

- Wir verwenden `DataInputStream` und `DataOutputStream` als Typen der I/O-Streams. Über solche Ströme lassen sich elementare Daten, in unserem Fall `long`-Werte, recht effizient in einem Binärformat übertragen, ohne sie zu Zeichenketten zu konvertieren und dann wieder zurückzukonvertieren. Es muss nur sichergestellt werden, dass Daten im gleichen Format gelesen wie geschrieben werden. Da alle Daten mit `writeLong` geschrieben werden, müssen sie mit `readLong` gelesen werden.

- Nach jedem Aufruf von `writeLong` folgt `flush`, um die Daten sofort weiterzuverarbeiten. Es reicht in diesem Fall nicht, sich darauf zu verlassen, dass Schreibbefehle an ungepufferte Ströme gleich an das Betriebssystem weitergereicht werden, da auch Betriebssysteme zur Effizienzsteigerung Pufferung verwenden. Meist werden Daten bei Erreichen eines Zeilenendes tatsächlich übertragen, was für die zeilenweise Kommunikation von Texten sehr gut geeignet ist. Hier werden jedoch rohe Binärdaten übertragen, es gibt also keine Zeilenenden. Daher ist `flush` nötig.

- Es hängt von vielen Faktoren ab, ob gepufferte oder ungepufferte Ströme vorteilhaft sind. Geschrieben wird hier über ungepufferte Ströme, gelesen über gepufferte Ströme. Wenn es auf Effizienz ankommt, empfiehlt es sich, sowohl mit gepufferten als auch ungepufferten Ströme Laufzeitmessungen durchzuführen und die effizientere Variante zu wählen. Mit etwas Erfahrung lässt sich die bessere Variante zwar recht gut prognostizieren, aber manchmal führt die Intuition in die Irre.

- Mittels `Runtime.getRuntime().exec(...)` ist es in Java einfach, neue Prozesse zu erzeugen. Aber der Parameter von `exec` gibt nur den Namen oder den Dateipfad eines Programms sowie Kommandozeilenargumente an. Damit ergibt sich bei Weitem nicht die Mächtigkeit einer Shell wie `bash`. Der Grund liegt in der Systemunabhängigkeit von Java. Um die

Mächtigkeit der Shell zu nutzen, etwa zwecks Umleitung der Ein- oder Ausgabe oder dem Aufbau komplexer Pipelines, wird häufig ein Shell-Skript aufgerufen. Das geht auf Kosten der Portabilität, weil Shell-Skripts stark vom Betriebssystem abhängen.

- Die Fehlersuche ist in parallelen Systemen sehr aufwändig. Beispielsweise werden in `ParProc` nur Fehlermeldungen vom `controller` in der Fehlerausgabe sichtbar, Fehlermeldungen der `worker`-Prozesse gehen verloren. Wenn `p` ein Objekt vom Typ `Process` ist, das einen `worker`-Prozess darstellt, könnten wir über `p.getErrorStream()` die Fehlermeldungen dieses `worker`-Prozesses lesen. Es erfordert jedoch komplexe Koordination, dies so zu bewerkstelligen, dass der Rest des Programms nicht darunter leidet. Viel einfacher wäre es, die Fehlerausgaben der `worker`-Prozesse auf Dateien umzulenken. Über Methoden von `Process` können wir feststellen, ob ein Prozess überhaupt noch läuft. Wir können unnötig gewordene Prozesse auch abbrechen. Jedoch erfordert auch das Koordinationsaufwand, weil abgebrochene Prozesse noch für eine gewisse Zeit weiterlaufen können und beim Abbrechen von Prozessen Pipelines brechen, was zu zusätzlichen Exceptions führen kann.

Obiges Programm wurde entwickelt, um die Interprozesskommunikation auf einfache Weise zu demonstrieren. Zur effizienten Primzahlberechnung ist es noch nicht optimal. Hier sind einige Verbesserungsvorschläge:

- Bei großen Datenmengen ist ein achtsamer Umgang mit dem Speicherverbrauch nötig. Das Array `prims` hat eine fixe Größe, für `controller` gleich wie für `worker`. Aber `worker` müssen nur Primzahlen bis zur Wurzel von `MAX` speichern, könnten also mit weniger Speicher auskommen.

- Durch Berücksichtigung von Eigenschaften der konkreten Aufgabe lässt sich der Aufwand reduzieren. Für `PROC` kann eine beliebige positive Zahl gewählt werden. Bei manchen Werten von `PROC` (z. B. 5) haben manche `worker` (etwa jener mit Nummer 0) sehr wenig zu tun, weil nach einer ersten Primzahl alle weiteren zu untersuchenden Zahlen Vielfache davon sind (5, 15, 25, ...). Das trifft immer zu, wenn der Anfangswert und `PROC` durch die gleiche Primzahl teilbar ist. Dieses Wissen könnten wir für vorzeitige Abbrüche nutzen, oder so integrieren, dass entsprechende Prozesse gar nicht erzeugt werden.

- Ineffiziente Algorithmen können den Aufwand in die Höhe treiben. Der in `controller` verwendete Algorithmus zur Suche des Minimums ist primitiv. Hier wäre mehr Effizienz möglich.

- Die Vermeidung von Engpässen ist eine sehr effiktive Möglichkeit zur Effizienzsteigerung. Bei einer größeren Zahl an `worker`-Prozessen wird `controller` zu einem Engpass. Wir sollten nach einer Möglichkeit suchen, dessen Funktionalität auf mehrere Prozesse aufzuteilen.

- Unnötig viel Kommunikation wirkt sich störend auf die Laufzeit aus. `controller` schickt gleiche Daten an unterschiedliche `worker`-Prozesse. Über Shells (z. B. `bash`) ist es möglich, File-Descriptoren zu duplizieren und damit den gleichen Ausgabestrom mit mehreren Eingabeströmen zu verbinden. Das würde den Umfang gesendeter Daten reduzieren.

- Eine unnötig kleine Granularität verschickter Datenpakete treibt die Belastung des Systems in die Höhe. Die Sortierung der Primzahlen garantiert, dass schon alle Primzahlen bis zu einer bekannten Zahl gefunden wurden. Würden wir ein anderes Kriterium verwenden, könnten wir die Daten in größere Blöcke zusammenfassen und damit den Kommunikationsaufwand (vor allem die häufigen Aufrufe von `flush`) reduzieren.

- Es stellt sich stets die Frage, wie gut der gewählte Ansatz zur Erfüllung der Aufgabe geeignet ist. Vielleicht ist eine andere Aufteilung der pro `worker` zu bearbeitenden Daten vorteilhaft. Das können wir nur sehen, wenn wir mehrere erfolgversprechende Ansätze ausprobieren.

- Grundannahmen sind zu überdenken, auch die, dass die Aufgabe durch Kommunikation über Pipelines gelöst werden soll. Alle `worker`-Prozesse verwenden die gleichen Daten. Shared-Memory sollte dafür besser einsetzbar sein als das Senden vieler Daten über Pipelines.

Das sind nur einige Hinweise darauf, in welche Richtungen Überlegungen zur Verbesserung des Programms gehen könnten. Der Fantasie sind hier keine Grenzen gesetzt. Die parallele Programmierung ist gerade deswegen ein spannendes Gebiet, weil es scheinbar unendlich viele Möglichkeiten gibt, die Laufzeit eines Programms noch weiter zu verringern oder die pro Zeiteinheit verarbeitete Datenmenge weiter zu erhöhen.

6 Entwurfsmuster und Entscheidungshilfen

Software-Entwurfsmuster, kurz Entwurfsmuster (*Design-Patterns*) dienen der Wiederverwendung kollektiver Erfahrung in der Softwareentwicklung. Wir wollen exemplarisch einige häufig verwendete Entwurfsmuster betrachten. Dabei konzentrieren wir uns, den Themen der Lehrveranstaltung entsprechend, auf Implementierungsaspekte und erwähnen andere in der Praxis wichtige Aspekte nur am Rande. Die Idee der Software-Entwurfsmuster gründet sich im Wesentlichen auf ein weithin bekanntes Buch (Gang-of-Four-Buch, kurz GoF-Buch), das allen Interessierten empfohlen wird [12]:

> E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, Reading, Massachusetts, 1994.

Es gibt eine Reihe neuerer Ausgaben und Übersetzungen in andere Sprachen, die ebenso empfehlenswert sind. Wir betrachten nur einen kleinen Teil der im Buch beschriebenen und in der Praxis häufig eingesetzten Entwurfsmuster, ergänzt um neuere Varianten und Entwicklungen.

6.1 Grundsätzliches und Muster für Verhalten

Erfahrung ist eine wertvolle Ressource zur effizienten Erstellung und Wartung von Software. Am effizientesten ist es, gewonnene Erfahrungen in Programmcode auszudrücken und diesen Code direkt wiederzuverwenden. Aber in vielen Fällen funktioniert Code-Wiederverwendung nicht. In diesen Fällen müssen wir zwar den Code neu schreiben, können dabei aber auf bestehende Erfahrungen zurückgreifen und darauf verzichten, immer wieder „das Rad neu zu erfinden“.

In erster Linie betrifft diese Art der Wiederverwendung die persönliche Erfahrung. Aber auch kollektive Erfahrung ist von großer Bedeutung. Gerade für den Austausch kollektiver Erfahrung können Hilfsmittel nützlich sein. Entwurfsmuster sind das bekannteste Hilfsmittel in diesem Bereich. Wir können heute davon ausgehen, dass so gut wie alle Leute in der Softwareentwicklung die wichtigsten Entwurfsmuster kennen und in der täglichen Arbeit immer wieder darauf zurückgreifen.

Entwurfsmuster geben im Softwareentwurf häufig wiederholt auftauchenden Problemstellungen und deren Lösungen Namen, damit einfacher darüber gesprochen werden kann. Außerdem beschreiben Entwurfsmuster, welche Eigenschaften wir uns von bestimmten Lösungen erwarten dürfen. Wer einen ganzen

Katalog möglicher Lösungen für eine Aufgabe entweder in schriftlicher Form oder nur abstrakt vor Augen hat, kann gezielt jene Lösung wählen, deren Eigenschaften der zu entwickelnden Software am ehesten entgegenkommen. Kaum eine Lösung wird nur gute Eigenschaften haben. Häufig wird jene Lösung gewählt, deren Nachteile am ehesten für akzeptabel gehalten werden.

Wir betrachten den üblichen Aufbau von Entwurfsmustern und danach drei Beispiele, die wir in ihren Grundzügen schon kennen und die Ihre Schwerpunkte im Objektverhalten haben: Visitor, Iterator und Template-Method.

6.1.1 Aufbau von Entwurfsmustern

Jedes Entwurfsmuster besteht hauptsächlich aus diesen vier Elementen:

Name: Der Name ist wichtig, damit wir in einem einzigen Begriff ein Problem, dessen Lösung und Konsequenzen daraus ausdrücken können. Damit wird die Kommunikation im Softwareentwurf auf eine höhere Ebene verlagert; wir müssen nicht mehr jedes Detail einzeln ansprechen. Der Name steht für eine Abstraktion, diesmal nicht bezogen auf ein Konzept in einem konkreten Programm, sondern bezogen auf ein ganz allgemeines Konzept im Softwareentwurf unabhängig von konkreten Programmen oder Programmiersprachen.[1] Es ist nicht leicht, dass alle Leute in der Softwareentwicklung sich auf einen gemeinsamen Namen für ein Entwurfsmuster einigen und das gleiche Verständnis dafür entwickeln, was der Name ausdrückt. Die in der Praxis verwendeten Namen müssen sich im Laufe der Zeit gegenüber anderen durchsetzen.

Beispielsweise haben wir in Abschnitt 4.4.2 das *Visitor-Pattern*, kurz den Visitor kennen gelernt. Ohne lange darüber nachdenken zu müssen, sollten wir diesen Begriff gleich mit mehrfachem dynamischem Binden, Visitor- und Element-Klassen, einer grundlegenden Implementierungstechnik dahinter, der Möglichkeit zur Vermeidung dynamischer Typabfragen und Typumwandlungen, aber auch dem Problem der hohen Anzahl an Methoden verbinden können. Erst dadurch, dass uns so viel dazu einfällt, wird der Begriff zu einem geeigneten Namen für ein Entwurfsmuster. Typisch ist auch die Abstraktion über eine konkrete Problemstellung: Es geht nicht nur um fressende Tiere, sondern um einen breiten Anwendungsbereich, der auch kovariante Probleme einschließt. Wegen der Breite und Wichtigkeit des Anwendungsbereichs wurde das Visitor-Pattern (wie alle

[1] Eine gewisse Abhängigkeit von Programmiersprachen ist insofern schon gegeben, als je nach Sprache unterschiedliche Implementierungstechniken zum Einsatz kommen, die zu unterschiedlichen Konsequenzen führen können. Wenn solche Unterschiede entscheidend sind, werden sie als Teil des Entwurfsmusters beschrieben. Eine Abhängigkeit von Programmierparadigmen ist dadurch häufig gegeben. Entwurfsmuster wurden ursprünglich ausschließlich für die objektorientierte Programmierung entwickelt und haben dort noch immer ihr Hauptanwendungsgebiet. Inzwischen gibt es auch Entwurfsmuster, die in anderen Paradigmen von Bedeutung sind. Paradigmen bestimmen, wie wichtig bestimmte Entwurfsmuster dafür sind. Aber die allgemeinen Aussagen sind unabhängig von Paradigmen. Sie sind jedoch in manchen Paradigmen von größerer Relevanz als in anderen.

wichtigen Entwurfsmuster) in der Fachliteratur eingehend analysiert, mit ähnlichen Techniken verglichen und häufig angezweifelt. Heute kennen wir mehrere Varianten, auf die wir hier nicht näher eingehen. Gerade diese umfangreiche Diskussion hat den Begriff erst wirklich etabliert. Genaugenommen verbinden wir mit dem Begriff nicht mehr nur eine einzige, eng umrissene Technik, sondern eine ganze Fülle ähnlicher Lösungsansätze in einem Anwendungsbereich, deren Eigenschaften gut bekannt sind.

Problemstellung: Das ist die Beschreibung des Problems zusammen mit dessen Umfeld. Daraus geht hervor, unter welchen Bedingungen das Entwurfsmuster überhaupt anwendbar ist. Bevor wir ein Entwurfsmuster in Betracht ziehen, müssen wir uns überlegen, ob die zu lösende Aufgabe mit dieser Beschreibung übereinstimmt.

Beispielsweise empfiehlt sich der Visitor dann, wenn „viele unterschiedliche, nicht verwandte Operationen auf einer Objektstruktur realisiert werden sollen, sich die Klassen der Objektstruktur nicht verändern, häufig neue Operationen auf der Objektstruktur integriert werden müssen oder ein Algorithmus über die Klassen einer Objektstruktur verteilt arbeitet, aber zentral verwaltet werden soll." Das klingt nicht nur kompliziert, sondern ist es auch. In der Praxis müssen wir uns schon recht intensiv mit der Problemstellung und den Einsatzmöglichkeiten beschäftigen, bevor wir ein Gefühl dafür bekommen, in welchen Situationen die Verwendung eines Entwurfsmusters angebracht ist. Mit wenig Erfahrung kennen wir oft nur eine oder einige wenige Einsatzmöglichkeiten. Durch den praktischen Einsatz, aber auch durch theoretische Überlegungen und einschlägige Literatur lernen wir im Laufe der Zeit immer weitere Einsatzmöglichkeiten kennen, bis wir das gesamte Einsatzgebiet abschätzen können.

Lösung: Das ist die Beschreibung einer bestimmten Lösung der Problemstellung. Die Beschreibung ist so allgemein wie möglich gehalten, damit sie leicht an unterschiedliche Situationen angepasst werden kann. Sie enthält jene Einzelheiten, die zu den beschriebenen Konsequenzen führen, aber nicht mehr. Manchmal sind mehrere unterschiedliche Lösungsvarianten beschrieben (oft als „Implementierungsdetails" bezeichnet), die zu unterschiedlichen Konsequenzen führen können, wobei die Unterschiede nicht groß genug sind, um die Einführung eigener Namen zu rechtfertigen.

Im Beispiel des Visitor-Patterns enthält die Beschreibung Erklärungen dafür, wie die Klassenstrukturen aussehen, welche Abhängigkeiten zwischen den Klassen bestehen und wie sich bestimmte Methoden darin verhalten. Meist gibt es nicht nur eine einzige „empfohlene" Struktur, sondern mehrere, einander ähnliche Varianten. Wir können jene Variante wählen, die am ehesten zur konkreten Aufgabenstellung passt. Je nach Entwurfsmuster kann aus einer mehr oder weniger breiten Palette an möglichen Implementierungsdetails gewählt werden. Gerade diese Freiheiten machen ein Entwurfsmuster aus. Ohne breite Auswahlmöglichkeiten wäre eine Klasse besser geeignet, die wir vorgefertigt in ein Programm einbinden können.

Konsequenzen: Das ist eine Liste von Eigenschaften der Lösung. Wir können sie als Liste der Vor- und Nachteile betrachten, müssen aber aufpassen, da ein und dieselbe Eigenschaft in manchen Situationen einen Vorteil darstellt, in anderen einen Nachteil und in wieder anderen irrelevant ist.

Eine wichtige Eigenschaften von Visitor in der betrachteten Version besteht darin, unerwünschte dynamische Typabfragen und Typumwandlungen durch dynamisches Binden zu ersetzen. Damit lässt sich die Wartbarkeit verbessern. Wir können flexibel auf Programmänderungen reagieren, die sonst oft größere Wartungsprobleme verursachen. Eine meist negative Eigenschaft ist die große Anzahl an Methoden bei komplexeren Klassenstrukturen. Für vielfaches dynamisches Binden und viele Klassen ist Visitor einfach nicht geeignet. Für zweifaches dynamisches Binden und wenige Klassen ist es dagegen gut geeignet. Die vollständige Liste der bekannten Konsequenzen ist lang. Häufig hängen Konsequenzen von bestimmten Implementierungsdetails ab.

Entwurfsmuster scheinen die Lösung vieler Probleme zu sein, da nur aus einem Katalog von Mustern gewählt werden muss, um eine ideale Lösung für ein Problem zu finden. Tatsächlich lassen sich Entwurfsmuster häufig so miteinander kombinieren, dass alle gewünschten Eigenschaften abgedeckt sind. Leider führt der exzessive Einsatz von Entwurfsmustern oft zu einem unerwünschten Effekt: Das Programm wird sehr komplex und undurchsichtig. Damit ist die Programmerstellung langwierig und die Wartung schwierig, obwohl die über den Einsatz der Entwurfsmuster erzielten Eigenschaften anderes versprechen. Wir sollen also genau abwägen, ob es sich im Einzelfall auszahlt, eine bestimmte Eigenschaft auf Kosten der Programmkomplexität zu erzielen. Die Softwareentwicklung bleibt auch dann eher eine Kunst als ein Handwerk, wenn Entwurfsmuster eingesetzt werden.

> *Faustregel:* Entwurfsmuster sollen zur Abschätzung der Konsequenzen von Designentscheidungen eingesetzt werden, können aber nur in begrenztem Ausmaß und mit Vorsicht als Bausteine zur Erzielung bestimmter Eigenschaften dienen.

6.1.2 Visitor

Visitor ist ein gutes Beispiel zur Vorstellung eines Entwurfsmusters, weil wir es schon kennen und es eher einfach ist. Heute wird meist eine Variante beschrieben, die der Verwendung in Abschnitt 4.4.2 entspricht. Zuerst betrachten wir diese Variante, danach kurz die ursprüngliche Variante im GoF-Buch.

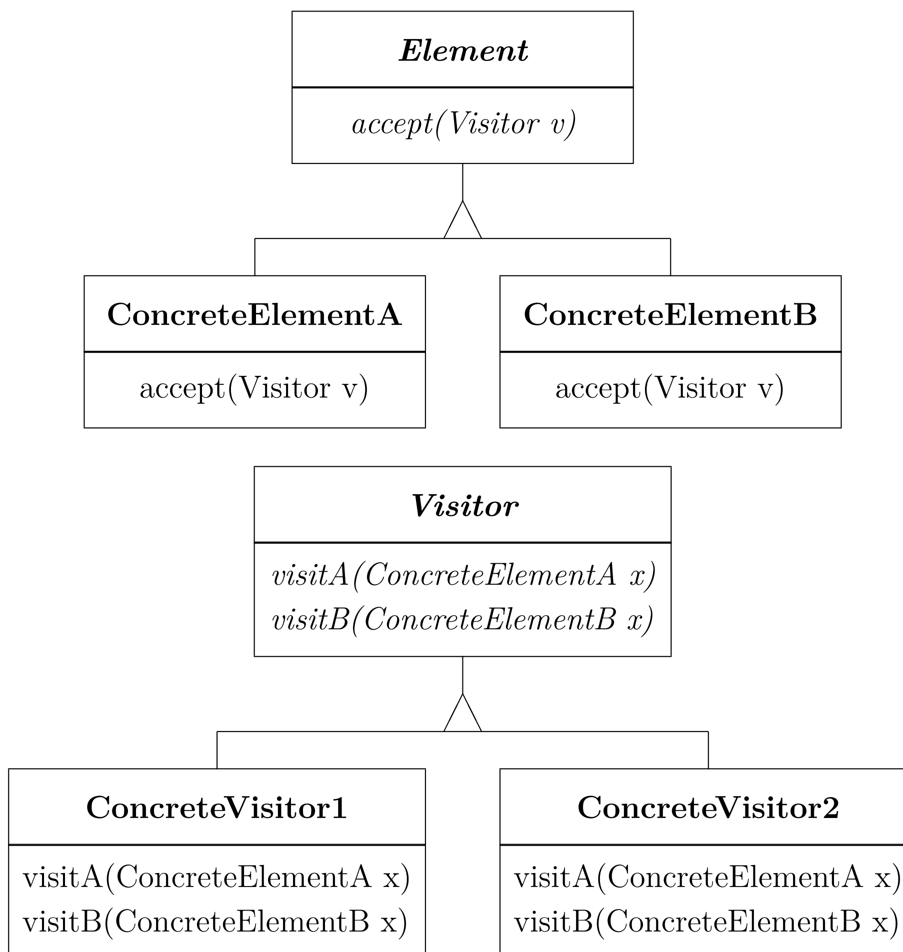
Visitor (deutsch *Besucher*) ist anwendbar, wenn

- viele unterschiedliche, nicht verwandte Operationen auf einer Objektstrutur realisiert werden sollen,
- sich die Klassen der Objektstruktur nicht ändern,

- häufig neue Operationen auf der Objektstruktur integriert werden müssen oder
- ein Algorithmus über die Klassen einer Objektstruktur verteilt arbeitet, aber zentral verwaltet werden soll.

In Abschnitt 4.4.2 war der letzte Grund für die Anwendung von Visitor ausschlaggebend. Zur Beschreibung der meisten Entwurfsmuster folgt hier ein typisches Anwendungsbeispiel. Da wir schon mit einem Beispiel vertraut sind, verzichten wir hier darauf.

Es folgt die Veranschaulichung der Struktur des Entwurfsmusters durch ein Diagramm. Die oben genannte *Objektstruktur* von Visitor entspricht dem Typ `Element` und dessen Untertypen:



Wir stellen Klassen (als allgemeines Konzept, schließen in Java z. B. auch Interfaces ein) als Kästchen dar, die die Namen der Klassen in Fettschrift enthalten.[2] Durch einen waagrechten Strich getrennt können Namen von Methoden (mit

[2]Solche Diagramme ähneln UML-Diagrammen. Viele Diagramme zur Beschreibung von Entwurfsmustern sind älter als der UML-Standard, weshalb es kleine Unterschiede zu heute üblichen UML-Diagrammen gibt. Wir verwenden hier wegen ihrer Kompaktheit die im GoF-Buch verwendete Form.

einer Parameterliste) und Variablen (ohne Parameterliste) in den Klassen in nicht-fetter Schrift angegeben sein. Namen von abstrakten Klassen und Methoden sind kursiv dargestellt, konkrete Klassen und Methoden nicht kursiv. Unterklassen sind mit deren Oberklassen durch Striche und Dreiecke, deren Spitzen zu den Oberklassen zeigen, verbunden. Es wird implizit angenommen, dass jede solche Vererbungsbeziehung gleichzeitig auch eine Untertypbeziehung ist. Eine strichlierte Linie mit einem Pfeil zwischen Klassen bedeutet, dass eine Klasse ein Objekt der Klasse, zu der der Pfeil zeigt, erzeugen kann (wird für Visitor nicht benötigt). Eine durchgezogene Linie mit Pfeil deutet eine Referenz an. Namen im Programmcode können sich von den Namen in der Grafik unterscheiden. Namen in der Grafik helfen dem intuitiven Verständnis der Struktur und ermöglichen Diskussionen darüber. Daher sollten wir diese Namen kennen, auch wenn im Programmcode andere Namen verwendet werden.

Visitor hat unter anderem folgende Eigenschaften:

- Neue Operationen lassen sich leicht durch die Definition neuer Untertypen von *Visitor* hinzufügen.
- Verwandte Operationen werden im Visitor zentral verwaltet und von Visitor-fremden Operationen getrennt.
- Ein Visitor kann mit Objekten aus voneinander unabhängigen Klassenhierarchien arbeiten.
- Die gute Erweiterungsmöglichkeit der Klassen unterhalb von Visitor muss mit einer schlechten Erweiterbarkeit der Klassen der konkreten Elemente erkauft werden. Müssen neue konkrete Elemente hinzugefügt werden, so führt dies dazu, dass viele Methoden implementiert werden müssen.
- Häufig wird angeführt, dass die visit-Methoden nicht einfach auf konkrete Elemente zugreifen können; oft können dies Parameter der visit-Methoden ausgleichen, wodurch es auf Implementierungsdetails ankommt, inwieweit das relevant ist.

Im GoF-Buch war Visitor ursprünglich etwas anders beschrieben: Die accept-Methoden rufen visit-Methoden nicht direkt in den entsprechenden Visitor-Klassen auf. Stattdessen verwalten Element-Klassen dynamische Listen von Visitors und leiten Aufrufe an entsprechende Listeneinträge weiter. Das ist zwar eine flexible Variante, aber eine, die in der Praxis eher selten verwendet wird, unter anderem, weil die Verwaltung der Listen Schwierigkeiten bereiten kann. Hier zeigt sich, dass Entwurfsmuster oft erst nach längeren Diskussionen stabil werden. Es stellt sich die Frage, ob das im GoF-Buch beschriebene Entwurfsmuster vielleicht etwas ganz anderes ist als das hier beschriebene. Bei genauerer Betrachtung ist das nur ein Implementierungsdetail. Wir können Details der Implementierung relativ stark abändern und ausweiten, ohne die Einsatzgebiete und Eigenschaften wesentlich zu ändern. Das macht ein stabiles Entwurfsmuster aus. Wir haben es wirklich mit einem abstrakten Muster zu tun, nicht nur mit

einer stark vereinfachten Beschreibung einer Implementierung. Von allen Implementierungen, auf die die Beschreibung des Entwurfsmusters zutrifft (auch wenn sie sich sehr weit von einer üblichen Implementierung entfernen), können wir erwarten, dass sie die Eigenschaften dieses Musters haben.

6.1.3 Iterator

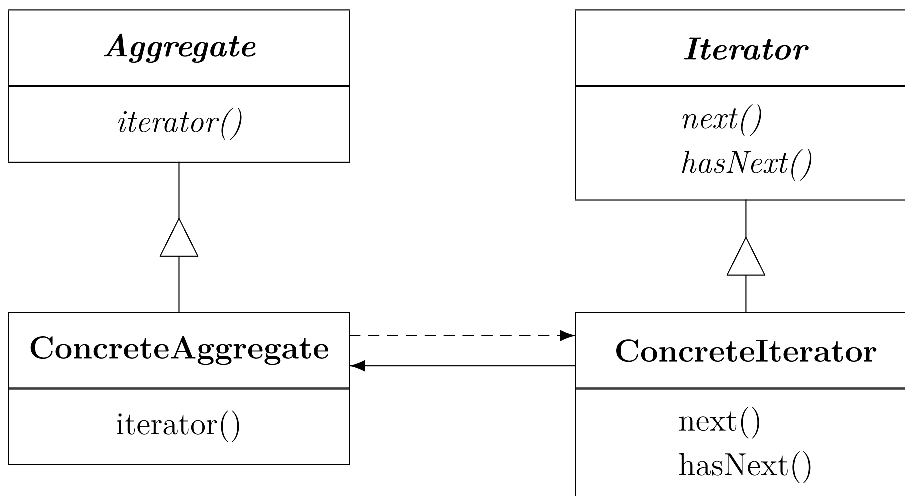
Als weiteres, gut bekanntes Beispiel betrachten wir nun Iteratoren auf der Ebene eines Entwurfsmusters. Name und Problemstellung lassen sich kurz so umreißen: Ein *Iterator*, auch *Cursor* genannt, ermöglicht den sequentiellen Zugriff auf die Elemente eines *Aggregats* (das ist eine Sammlung von Elementen, beispielsweise eine Collection), ohne die innere Darstellung des Aggregats offenzulegen.

Dieses Entwurfsmuster ist verwendbar, um

- auf die Inhalte eines Aggregats zugreifen zu können, ohne die innere Darstellung offen legen zu müssen,
- mehrere (gleichzeitige bzw. überlappende) Abarbeitungen der Elemente in einem Aggregat zu ermöglichen,
- eine einheitliche Schnittstelle für die Abarbeitung verschiedener Aggregatstrukturen zu haben, also um polymorphe Iterationen zu unterstützen.

Wegen der Bekanntheit verzichten wir auf ein Beispiel.

Die Struktur eines Iterators sieht wie in der folgenden Grafik aus:



Die abstrakte Klasse oder das Interface *Iterator* definiert eine Schnittstelle für den Zugriff auf Elemente sowie deren Abarbeitung. Die Klasse „ConcreteIterator“ implementiert diese Schnittstelle und verwaltet die aktuelle Position in der Abarbeitung. Die abstrakte Klasse oder das Interface *Aggregate*[3] definiert eine

[3] *Aggregat* (bzw. englisch *Aggregate*) hat sich als Fachbegriff etabliert. In der Kommunikation sollen wir diese Fachbegriffe verwenden (statt konkreter Namen wie „Iterable“ oder Umschreibungen wie „Collection“), weil dadurch ohne weitere Erläuterungen klar ist, dass wir dabei die Eigenschaften des Entwurfsmusters im Kopf haben. Aus diesem Grund unterscheiden sich Klassen- und Methodennamen oft bewusst von entsprechenden Namen in Entwurfsmustern.

Schnittstelle für die Erzeugung eines neuen Iterators (`Iterable<T>` in Java). Die Klasse „ConcreteAggregate“ implementiert diese Schnittstelle. Ein Aufruf von „iterator“ erzeugt üblicherweise ein neues Objekt von „ConcreteIterator“, was durch den strichlierten Pfeil angedeutet ist. Um die aktuelle Position im Aggregat verwalten zu können, braucht jedes Objekt von „ConcreteIterator“ eine Referenz auf das entsprechende Objekt von „ConcreteAggregate“, angedeutet mittels durchgezogenem Pfeil.

Iteratoren haben drei wichtige Eigenschaften:

- Sie unterstützen unterschiedliche Varianten in der Abarbeitung von Aggregaten. Für komplexe Aggregate wie beispielsweise Bäume gibt es zahlreiche Möglichkeiten, in welcher Reihenfolge die Elemente abgearbeitet werden. Es ist leicht, mehrere Iteratoren für unterschiedliche Reihenfolgen auf demselben Aggregat zu implementieren.

- Iteratoren vereinfachen die Schnittstelle von *Aggregate*, da Zugriffsmöglichkeiten, die über Iteratoren bereitgestellt werden, durch die Schnittstelle von *Aggregate* nicht unterstützt werden müssen. Die in obiger Struktur in der Klasse *Iterator* enthaltenen Methoden stellen nur eine Mindestanforderung dar. Daneben kann es weitere Methoden geben, die *Aggregate* zusätzlich vereinfachen. Beispielsweise enthält das in den Java-Standardbibliotheken vordefinierte Interface `Iterator` auch eine Methode `remove`, um das aktuelle Element zu entfernen.

- Auf ein und demselben Aggregat können gleichzeitig mehrere Abarbeitungen stattfinden, da jeder Iterator selbst den aktuellen Abarbeitungszustand verwaltet. Gelegentlich finden wir Iterator-ähnlichen Code, in dem der Abarbeitungszustand im Aggregat und nicht im Iterator verwaltet wird. Solcher Code widerspricht dem hier vorgestellten Entwurfsmuster, da zwingend gefordert ist, dass auf demselben Aggregat mehrere gleichzeitige Abarbeitungen möglich sein müssen.

Es gibt zahlreiche Möglichkeiten zur Implementierung von Iteratoren. Hier sind einige Anmerkungen zu Implementierungsvarianten:

- Wir können zwischen *internen* und *externen* Iteratoren unterscheiden. Interne Iteratoren kontrollieren selbst, wann die nächste Iteration erfolgt, bei externen Iteratoren bestimmt die Anwendung, wann sie das nächste Element abarbeiten möchte. Bei internen Iteratoren liegt die Schleife (oder Rekursion), mit der das Aggregat durchlaufen wird, innerhalb der Iterator-Implementierung, bei externen Iteratoren außerhalb. Die Methoden „next“ und „hasNext“ sind nur bei externen Iteratoren öffentlich sichtbar, bei internen Iteratoren bleiben sie nach außen verborgen (oder existieren nur indirekt in versteckter Form). In Java ist beispielsweise jeder Iterator, der mittels `iterator()` erzeugt wird, ein externer Iterator. Wenn wir dagegen etwa über eine `Collection` oder in einem Spliterator mittels `forEach(...)` iterieren, verwenden wir einen internen Iterator.

Auch `map(...)` in Java-8-Streams ist ein interner Iterator. Die Methode *iterator* in *Aggregate* kann in einer Implementierung also auch etwa `forEach` oder `map` heißen. Bei der Erzeugung übergeben wir einem internen Iterator eine Operation (z. B. in Form eines Lambdas), die vom Iterator auf die einzelnen Elemente angewandt wird.

Externe Iteratoren sind flexibler als interne Iteratoren. Zum Beispiel ist es mit externen Iteratoren leicht, zwei Aggregate miteinander zu vergleichen. Mit internen Iteratoren ist das schwieriger. Andererseits sind interne Iteratoren oft einfacher zu verwenden, da eine Anwendung die Logik für die Iterationen (also die Schleife) nicht braucht. Interne Iteratoren spielen vor allem in der funktionalen Programmierung eine große Rolle, da es dort gute Unterstützung für die Übergabe von Funktionen bei der Iterator-Erzeugung gibt und externe Schleifen problematisch sind. In der objektorientierten Programmierung wurden bisher hauptsächlich externe Iteratoren eingesetzt. Seit der Einführung von Lambdas und Java-8-Streams gibt es auch in Java interne Iteratoren von hoher Qualität, wodurch sich das bald ändern könnte. Schließlich bieten interne Iteratoren eine einfache Möglichkeit zur parallelen Verarbeitung großer Mengen voneinander unabhängiger Daten, während externe Iteratoren die Parallelverarbeitung eher erschweren.

- Oft ist es schwierig, externe Iteratoren auf Sammlungen von Elementen zu verwenden, wenn diese Elemente in komplexen Beziehungen zueinander stehen. Durch die sequentielle Abarbeitung geht die Struktur dieser Beziehungen verloren. Beispielsweise erkennen wir an einem vom Iterator zurückgegebenen Element nicht mehr, an welcher Stelle in einem Baum das Element steht. Wenn die Beziehungen zwischen den Elementen bei der Abarbeitung benötigt werden, ist es meist einfacher, interne statt externer Iteratoren zu verwenden oder ganz auf Iteratoren zu verzichten.

- Der Algorithmus zum Durchwandern eines Aggregats muss nicht immer im Iterator definiert sein. Häufig wird er vom Aggregat bereitgestellt. Wenn der Iterator den Algorithmus definiert, ist es leichter, mehrere Iteratoren mit unterschiedlichen Algorithmen zu verwenden. In diesem Fall ist es auch leichter, Teile eines Algorithmus in einem anderen Algorithmus wiederzuverwenden. Andererseits müssen die Algorithmen oft private Implementierungsdetails des Aggregats verwenden. Das geht natürlich leichter, wenn die Algorithmen im Aggregat definiert sind. In Java können wir Iteratoren durch innere Klassen in Aggregaten definieren, wie zum Beispiel den Iterator in der Klasse `List` (siehe Abschnitt 4.1.2). Dies ermöglicht dem Iterator, auf private Details des Aggregats zuzugreifen. Allerdings wird dadurch die ohnehin schon starke Abhängigkeit zwischen Aggregat und Iterator noch stärker. Trotzdem sind innere Klassen in diesem Fall meist vorteilhaft.

- Es kann gefährlich sein, ein Aggregat zu verändern, während es von einem Iterator durchwandert wird. Wenn Elemente dazugefügt oder entfernt

werden, passiert es leicht, dass Elemente nicht oder doppelt abgearbeitet werden. Eine scheinbar einfache Lösung besteht darin, die Elemente des Aggregats bei der Iterator-Erzeugung zu kopieren. Aus praktischer Sicht ist diese Lösung meist viel zu aufwändig. Häufig möchten wir, dass Iteratoren Änderungen des Aggregats „sehen", also Änderungen nicht ignorieren. Es ist strittig, ob ein Iterator auf einer Kopie der Daten überhaupt als Iterator auf dem Original angesehen werden kann. Ein *robuster Iterator* erreicht das Ziel ohne Kopieren der Daten. Es ist aufwändig, robuste Iteratoren zu schreiben. Probleme hängen von der Art des Aggregats ab.

- Aus Gründen der Allgemeinheit ist es oft praktisch, Iteratoren auch auf leeren Aggregaten bereitzustellen. In einer Anwendung müssen wir die Schleife nur so lange ausführen, so lange es Elemente gibt – bei leeren Aggregaten daher nie – ohne eine eigene Behandlung für den Spezialfall zu benötigen. Das gilt auch für interne Iteratoren. Operationen auf Java-8-Streams können meist gut mit leeren Strömen umgehen. Der Wegfall von Sonderbehandlungen macht Programme kürzer und weniger fehleranfällig.

6.1.4 Template-Method

Eine *Template-Method* definiert das Grundgerüst eines Algorithmus in einer Operation, überlässt die Implementierung einiger Schritte aber einer Unterklasse. Template-Methods erlauben einer Unterklasse, bestimmte Schritte zu überschreiben, ohne die Struktur des Algorithmus zu ändern.

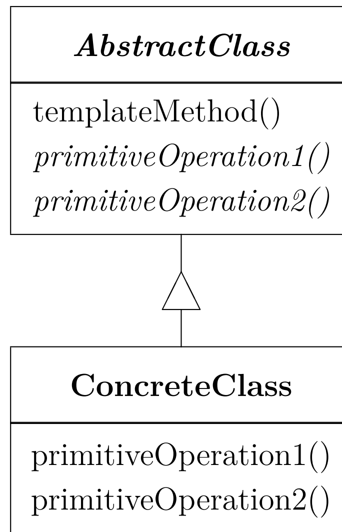
Dieses Entwurfsmuster ist anwendbar

- um den unveränderlichen Teil eines Algorithmus nur einmal zu implementieren und es Unterklassen zu überlassen, den veränderbaren Teil des Verhaltens festzulegen;
- wenn gemeinsames Verhalten mehrerer Unterklassen (zum Beispiel im Zuge einer Refaktorisierung) in einer einzigen Klasse lokal zusammengefasst werden soll, um Duplikate im Code zu vermeiden;
- um mögliche Erweiterungen in Unterklassen zu kontrollieren, beispielsweise durch Template-Methods, die *Hooks* als primitive Operationen (siehe die unten stehende Strukturzeichnung) aufrufen und nur das Überschreiben dieser Hooks (keiner anderer Methoden) in Unterklassen ermöglichen.

Ein Hook ist eine Methode mit einer Default-Implementierung, die dafür vorgesehen ist, in Untertypen überschrieben zu werden. Im Gegensatz zu einer abstrakten Methode muss ein Hook aber nicht überschrieben werden. Die Default-Implementierung hat in der Regel keinen Effekt, macht also nichts und gibt nur einen Wert zurück, der keinen Effekt andeutet (etwa 0 wenn eine Zahlensumme gebildet werden soll oder 1 für ein Zahlenprodukt).

Wir haben Template-Method in Abschnitt 3.3.2 eingesetzt, um zu zeigen, wie wir die direkte Code-Wiederverwendung unterstützen können. Das entspricht dem ersten Punkt in obiger Liste von Einsatzszenarien.

Die Struktur dieses Entwurfsmusters ist recht einfach:



Die (meist abstrakte) Klasse *AbstractClass* definiert (möglicherweise abstrakte) primitive Operationen und implementiert als „templateMethod" das Grundgerüst des Algorithmus, das die primitiven Operationen aufruft. Jede von der „templateMethod" aufgerufende Methode wird als primitive Operation bezeichnet und stellt einen Schritt in der Ausführung der „templateMethod" dar. Die Klasse „ConcreteClass" implementiert (einige) primitive Operationen, erbt die „templateMethod" aber unverändert.

Template-Methods haben unter anderem folgende Eigenschaften:

- Sie stellen eine fundamentale Technik zur direkten Wiederverwendung von Programmcode dar. Sie sind vor allem in Klassenbibliotheken und Frameworks sinnvoll, weil sie gemeinsames Verhalten faktorisieren.
- Sie führen zu einer Umkehrung der üblichen Kontrollstruktur, die manchmal als *Hollywood-Prinzip* bezeichnet wird („Don't call us, we'll call you"). Das bedeutet, die Oberklasse ruft Methoden der Unterklasse auf – nicht wie in den meisten Fällen umgekehrt.
- „templateMethod" ruft folgende Arten von primitiven Operationen auf (wobei die in einer bestimmten „templateMethod" aufgerufenen Methoden häufig alle von der gleichen Art sind):
  - konkrete Operationen in „AbstractClass", also Operationen, die ganz allgemein auch für Unterklassen sinnvoll sind;
  - abstrakte primitive Operationen, die einzelne Schritte im Algorithmus ausführen (und in „ConcreteClass" implementiert sind);
  - Hooks, also Operationen mit in „AbstractClass" definiertem Default-Verhalten, das in Unterklassen überschrieben werden kann,
  - Factory-Methods, also Methoden, die in Unterklassen neue Objekte erzeugen und zurückgeben und damit die Template-Method auch zu einem anderen Entwurfsmuster werden lassen – siehe Abschnitt 6.2.1.

> Es muss genau spezifiziert sein, welche Operationen Hooks sind (dürfen überschrieben werden), welche abstrakt sind (müssen überschrieben werden) und welche nur in „AbstractClass“ implementiert sein sollen. Für die effektive Wiederverwendung ist es wichtig, dass alle Beteiligten wissen, welche Operationen dafür vorgesehen sind, in Unterklassen überschrieben zu werden. Alle Operationen, bei denen dies Sinn macht, sollen Hooks oder abstrakte Methoden sein, da es beim Überschreiben anderer Operationen leicht zu Fehlern kommt.

Die primitiven Operationen, die von der Template-Methode aufgerufen werden, sind (wenn *AbstractClass* kein Interface ist) häufig `protected` Methoden, damit sie nicht in unerwünschten Zusammenhängen aufrufbar sind. Primitive Operationen, die überschrieben werden müssen, sind als `abstract` deklariert. „templateMethod“ selbst, also die Methode, die den Algorithmus implementiert, soll nicht überschrieben werden. Sie kann (wie alle nicht-abstrakten Methoden in *AbstractClass* außer Hooks) `final` sein.

Ein Ziel bei der Entwicklung einer Template-Methode sollte sein, die Anzahl der primitiven Operationen möglichst klein zu halten. Je mehr Operationen überschrieben werden müssen, desto komplizierter wird die direkte Wiederverwendung von *AbstractClass*.

6.2 Erzeugende Entwurfsmuster

Erzeugende Entwurfsmuster beschäftigen sich mit der Erzeugung neuer Objekte auf eine Art und Weise, die weit über die Möglichkeiten der Verwendung von `new` in Java hinausgeht. Entwurfsmuster sind für die Objekterzeugung deswegen besonders interessant, weil die Objekterzeugung eng mit der Parametrisierung verknüpft ist – siehe Abschnitt 1.3.2. Wir betrachten drei recht einfache erzeugende Entwurfsmuster: Factory-Method, Prototype und Singleton. Diese Entwurfsmuster wurden gewählt, da sie zeigen, dass die in Programmiersprachen vorgegebenen Möglichkeiten oft mit relativ einfachen Programmiertechniken erweiterbar sind.

6.2.1 Factory Method

Der Zweck einer *Factory-Method*, auch *Virtual-Constructor* genannt, ist die Definition einer Schnittstelle für die Objekterzeugung, wobei Unterklassen entscheiden, von welcher Klasse die erzeugten Objekte sein sollen. Die tatsächliche Erzeugung der Objekte wird in Unterklassen verschoben.

Als Beispiel für eine Anwendung der Factory-Method können wir uns ein System zur Verwaltung von Dokumenten unterschiedlicher Arten (Texte, Grafiken, Videos, etc.) vorstellen. Dabei gibt es eine (abstrakte) Klasse `DocCreator` mit der Aufgabe, neue Dokumente anzulegen. Nur in einer Unterklasse, der die Art des neuen Dokuments bekannt ist, kann die Erzeugung tatsächlich durchgeführt werden. Wie in `NewDocManager` ist der genaue Typ des zu erzeugenden Objekts zur Übersetzungszeit oft unbekannt, sodass ein `new` nicht ausreicht:

```
public interface Document { ... }
public class Text implements Document { ... }
...   // classes Picture, Video, ...
public interface DocCreator {
    Document create();
}
public class TextCreator implements DocCreator {
    public Document create() { return new Text(); }
}
...   // classes PictureCreator, VideoCreator, ...
public class NewDocManager {
    private DocCreator c = ...;
    public void set(DocCreator c) { this.c = c; }
    public Document newDoc() { return c.create(); }
}
```

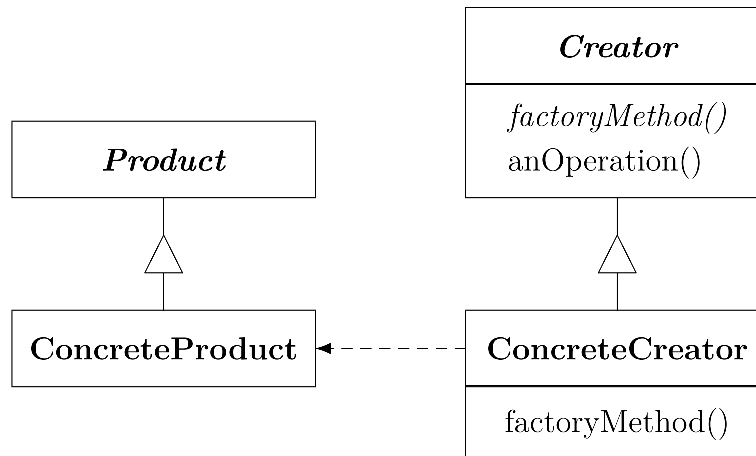
In diesem Beispiel ist der Typ des zu erzeugenden Objekts unter Zuhilfenahme eines Objekts von `DocCreator` in einer Variablen als zentraler Ablage festgelegt. Aus Gründen der Einfachheit haben die Konstruktoren der Dokumente hier keine Parameter. Wenn Sie welche hätten, dann könnten entsprechende Argumente auf zahlreiche Arten festgelegt werden:

- Argumente von allgemeinem Interesse (nicht spezifisch für bestimmte Dokumente) können an `newDoc` übergeben und über `create` an den (nicht näher bekannten) Konstruktor weitergeleitet werden.
- Wir können Argumente an zentraler Stelle ablegen. Eine Variable wie `c` in `NewDocManager` eignet sich dafür nur für Argumente von allgemeinem Interesse, die über `create` an den (nicht näher bekannten) Konstruktor weitergeleitet werden können. Argumente, die nur für bestimmte Dokumente sinnvoll sind, können wir direkt in Objekten der entsprechenden Untertypen von `DocCreator` ablegen.
- Am einfachsten ist es, für bestimmte Dokumente spezifische, aber unveränderliche Argumente fix in die Methode `create` einzucodieren.

Generell ist das Entwurfsmuster anwendbar wenn

- eine Klasse Objekte erzeugen soll, deren Klasse aber nicht kennt;
- eine Klasse möchte, dass ihre Unterklassen die Art der Objekte bestimmen, welche die Klasse erzeugt;
- Klassen Verantwortlichkeiten an eine von mehreren Unterklassen delegieren und das Wissen, an welche Unterklasse delegiert wird, lokal gehalten werden soll;
- die Allokation und Freigabe von Objekten zentral in einer Klasse verwaltet werden soll.

Das Entwurfsmuster hat folgende Struktur:



Die (oft abstrakte) Klasse *Product* ist (wie `Document` im Beispiel) ein gemeinsamer Obertyp aller Objekte, die von der Factory-Method erzeugt werden können. Die Klasse „ConcreteProduct" ist eine bestimmte Unterklasse davon, beispielsweise `Text`. Die abstrakte Klasse *Creator* enthält neben anderen Operationen die *factoryMethod* als (meist abstrakte) Methode. Diese Methode kann von außen, aber auch beispielsweise in „anOperation" von der Klasse selbst verwendet werden. Eine Unterklasse „ConcreteCreator" implementiert die *factoryMethod*. Ausführungen dieser Methode erzeugen neue Objekte von „ConcreteProduct".

Factory-Methods haben unter anderem folgende Eigenschaften:

- Sie bieten (durch abstrakte Methoden oder Hooks) Anknüpfungspunkte für Unterklassen, indem sie genau vorgeben, welche Methoden für das Überschreiben vorgesehen sind – siehe Abschnitt 6.1.4. Das führt zu einer Umkehrung der Abhängigkeiten: Oberklassen hängen von Unterklassen ab, nicht (nur) Unterklassen von Oberklassen (Hollywood-Prinzip). Die Erzeugung eines neuen Objekts mittels Factory-Method ist fast immer flexibler als die direkte Objekterzeugung. Vor allem wird die Entwicklung von Unterklassen vereinfacht.

- Sie verknüpfen *parallele Typhierarchien,* die *Creator*-Hierarchie mit der *Product*-Hierarchie. Z. B. ist die Typstruktur bestehend aus `Document`, `Text` und so weiter äquivalent zu der, die von den Typen `DocCreator`, `TextCreator` und so weiter gebildet wird – für jedes „ConcreteProduct" ein „ConcreteCreator" und umgekehrt. Dies kann unter anderem bei kovarianten Problemen hilfreich sein. Beispielsweise erzeugt eine Methode `generateFood` in der Klasse `Animal` nicht direkt Futter einer bestimmten Art, sondern liefert in der Unterklasse `Cow` ein neues Objekt von `Grass` und in `Tiger` eines von `Meat` zurück. Meist sind parallele Klassenhierarchien (wegen der vielen Klassen) aber unerwünscht.

Zur Implementierung dieses Entwurfsmusters können wir die *factoryMethod* in *Creator* entweder als abstrakte Methode realisieren, oder als Default eine

Implementierung dafür vorgeben (das ist ein Hook). Im ersten Fall muss *Creator* keine Klasse kennen, die als „ConcreteProduct“ verwendbar ist, dafür sind alle konkreten Unterklassen gezwungen, die *factoryMethod* zu implementieren. Im zweiten Fall kann *Creator* selbst zu einer konkreten Klasse werden, gibt aber Unterklassen die Möglichkeit, die *factoryMethod* zu überschreiben.

Es ist oft sinnvoll, der *factoryMethod* Parameter mitzugeben, die bestimmen, welche Art von Produkt zu erzeugen ist. In diesem Fall bietet die Möglichkeit des Überschreibens mehr Flexibilität, ist aber keine unabdingbare Voraussetzung. In Java bietet es sich an, der *factoryMethod* ein Lambda als Parameter zu übergeben, das die eigentliche Objekterzeugung übernimmt. Dadurch ergibt sich eine deutliche Vereinfachung. Im Grunde ändert die Verwendung von Lambdas nicht viel an der Struktur der Factory-Method: Jedes Lambda entspricht einer anonymen inneren Klasse als „ConcreteCreator“ und das für das Lambda verwendete funktionale Interface dem *Creator*, die *factoryMethod* selbst kann irgendwo stehen. Nur der Schreibaufwand wird erheblich verringert.

Hier ist eine Anwendung einer Factory-Method mit *Lazy-Initialization*:

```
public abstract class Creator {
    private Product product = null;
    protected abstract Product createProduct();
    public Product getProduct() {
        if (product == null)
            product = createProduct();
        return product;
    }
}
```

Ein neues Objekt wird nur einmal erzeugt. Die Methode `getProduct` gibt bei jedem Aufruf dasselbe Objekt zurück.

Ein Nachteil des Entwurfsmusters besteht in der Notwendigkeit, viele Unterklassen von *Creator* zu erzeugen, die nur `new` mit einem bestimmten „ConcreteProduct“ aufrufen. Die Erstellung dieser Klassen ist lästig. Mit Lambdas lässt sich zumindest der Schreibaufwand reduzieren.

6.2.2 Prototype

Das Entwurfsmuster *Prototype* dient dazu, die Art eines neu zu erzeugenden Objekts durch ein Prototyp-Objekt zu spezifizieren. Neue Objekte werden durch Kopieren dieses Prototyps erzeugt.

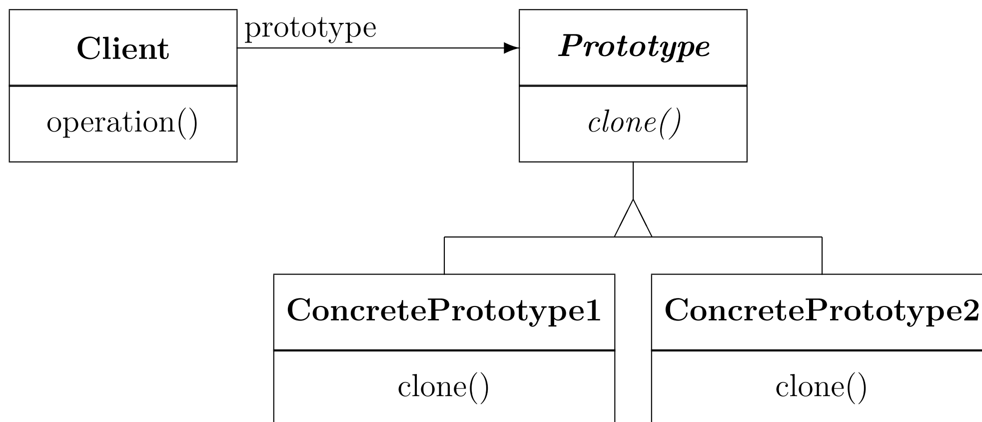
Zum Beispiel können wir in einem System, in dem verschiedene Arten von Polygonen wie Dreiecke und Rechtecke vorkommen, ein neues Polygon durch Kopieren eines bestehenden Polygons erzeugen. Das neue Polygon hat dieselbe Klasse wie das Polygon, von dem die Kopie erstellt wurde. An der Stelle im Programm, an der der Kopiervorgang aufgerufen wird (etwa in einem Zeichenprogramm), muss diese Klasse nicht bekannt sein. Das neue Polygon kann, vielleicht durch Ändern seiner Größe oder Position, einen vom kopierten Polygon verschiedenen Zustand erhalten:

```
public Polygon duplicate(Polygon orig) {
    Polygon copy = orig.clone();
    copy.move(X_OFFSET, Y_OFFSET);
    return copy;
}
```

Generell ist dieses Entwurfsmuster anwendbar, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und dargestellt werden, und wenn

- die Klassen, von denen Objekte erzeugt werden sollen, erst zur Laufzeit bekannt sind, oder
- vermieden werden soll, eine Hierarchie von *Creator*-Klassen zusammen mit einer parallelen Hierarchie von *Product*-Klassen zu erzeugen (also Factory-Method vermieden werden soll), oder
- jedes Objekt einer Klasse nur wenige unterschiedliche Zustände haben kann; es ist oft einfacher, für jeden möglichen Zustand einen Prototyp zu erzeugen und diese Prototypen zu kopieren, als Objekte durch `new` zu erzeugen und dabei passende Zustände anzugeben.

Das Entwurfsmuster hat folgende Struktur. Ein durchgezogener Pfeil bedeutet, dass jedes Objekt der Klasse, von der der Pfeil ausgeht, auf ein Objekt der Klasse, auf die der Pfeil zeigt, verweist. Die entsprechende Variable hat den Namen, mit dem der Pfeil bezeichnet ist.



Die (möglicherweise abstrakte) Klasse *Prototype* spezifiziert (wie `Polygon` im Beispiel) eine (möglicherweise abstrakte) Methode *clone*, um sich selbst zu kopieren. Die konkreten Unterklassen (wie `Dreieck` und `Rechteck`) überschreiben diese Methode. Die Klasse „Client“ entspricht im Beispiel dem Zeichenprogramm mit der Methode `duplicate`. Zur Erzeugung eines neuen Objekts wird *clone* in *Prototype* oder durch dynamisches Binden in einem Untertyp von *Prototype* aufgerufen.

Prototypen haben unter anderem folgende Eigenschaften:

- Sie verstecken die konkreten Produktklassen vor den Anwendern („Client") und reduzieren damit die Anzahl der Klassen, die Anwender kennen müssen. Die Anwender müssen nicht geändert werden, wenn neue Produktklassen dazukommen oder geändert werden.

- Prototypen können auch zur Laufzeit jederzeit dazugegeben und weggenommen werden. Im Gegensatz dazu darf die Klassenstruktur zur Laufzeit in der Regel nicht verändert werden.

- Sie erlauben die Spezifikation neuer Objekte durch änderbare Werte. In hochdynamischen Systemen kann neues Verhalten durch Objektkomposition (das Zusammensetzen neuer Objekte aus mehreren bestehenden Objekten) statt durch die Definition neuer Klassen erzeugt werden, beispielsweise durch die Spezifikation von Werten in Objektvariablen. Verweise auf andere Objekte in Variablen ersetzen dabei Vererbung. Die Erzeugung einer Kopie eines Objekts ähnelt der Erzeugung einer Klasseninstanz. Der Zustand eines Prototyps kann sich (wie der jedes beliebigen Objekts) jederzeit ändern, während Klassen zur Laufzeit unveränderlich sind.

- Sie vermeiden eine übertrieben große Anzahl an Unterklassen. Im Gegensatz zur Factory-Method ist es nicht nötig, parallele Klassenhierarchien zu erzeugen.

- Sie erlauben die dynamische Konfiguration von Programmen. In Programmiersprachen wie C++ ist es nicht möglich, Klassen dynamisch zu laden. Prototypes erlauben ähnliches auch in diesen Sprachen.

Für dieses Entwurfsmuster ist es notwendig, dass jede konkrete Unterklasse von *Prototype* die Methode *clone* implementiert. Gerade das ist aber oft schwierig, vor allem, wenn Klassen aus Klassenbibliotheken Verwendung finden, oder wenn es zyklische Referenzen gibt.

Um die Verwendung dieses Entwurfsmusters zu fördern, haben die Entwickler von Java die Methode `clone` bereits in `Object` vordefiniert. Damit ist `clone` in jeder Java-Klasse vorhanden und kann überschrieben werden. Die Default-Implementierung in `Object` erzeugt *flache* Kopien von Objekten, das heißt, der Wert jeder Variable in der Kopie ist identisch mit dem Wert der entsprechenden Variable im kopierten Objekt. Wenn die Werte von Variablen nicht identisch sondern nur gleich sein sollen, muss `clone` für jede Variable aufgerufen werden. Zur Erzeugung solcher *tiefer* Kopien muss die Default-Implementierung überschrieben werden. Um unerwünschte Kopien von Objekten in Java zu vermeiden, gibt die Default-Implementierung von `clone` nur dann eine Kopie des Objekts zurück, wenn die Klasse des Objekts das Interface `Cloneable` implementiert. Andernfalls löst `clone` eine Ausnahme aus.

Eine Implementierung von `clone` zur Erzeugung tiefer Kopien kann sehr komplex sein. Das Hauptproblem stellen zyklische Referenzen dar. Wenn `clone` einfach nur naiv rekursiv auf zyklische Strukturen angewandt wird, ergibt sich eine Endlosrekursion, die zum Programmabbruch aus Speichermangel führt.

Dieses Problem ist lösbar, indem wir eine Liste bereits kopierter Objekte mitführen und abbrechen, wenn wir auf ein schon kopiertes Objekt treffen. Das Mitführen einer Liste erfordert einen Parameter der Methode, weshalb die vorimplementierte parameterlose Methode `clone()` dafür nicht geeignet ist. Ähnliche Probleme ergeben sich, wenn Objekte ausgegeben und wieder eingelesen werden sollen. Das vordefinierte Interface `Serializable` in Java hilft bei der Erstellung entsprechender Umformungen.

Es ist schwer, den Überblick über ein System zu behalten, das viele Prototypen enthält. Das gilt vor allem für Prototypen, die zur Laufzeit dazukommen. Zur Lösung dieses Problems haben sich *Prototyp-Manager* bewährt, das sind assoziative Datenstrukturen (kleine Datenbanken), in denen nach geeigneten Prototypen gesucht wird.

Oft ist es notwendig, nach Erzeugung einer Kopie den Objektzustand zu verändern. Im Gegensatz zu Konstruktoren kann „clone" auf Grund des Ersetzbarkeitsprinzips meist nicht mit passenden Argumenten aufgerufen werden. In diesen Fällen ist es nötig, dass die Klassen Methoden zur Initialisierung beziehungsweise zum Ändern des Zustands bereitstellen.

Prototype ist als Entwurfsmuster vor allem in eher statisch typisierten Sprachen wie C++ und Java sinnvoll. In dynamisch typisierten Sprachen wie Smalltalk und Python wird ähnliche Funktionalität bereits direkt von der Sprache unterstützt. Es gibt sehr dynamische objektorientierte Sprachen, die kein Klassenkonzept bereitstellen und in denen neue Objekte ausschließlich durch Kopieren bestehender Objekte eingeführt werden, etwa die Sprache *Self* [34]. Diese Sprache wurde und wird zwar praktisch gar nicht verwendet, hatte aber großen Einfluss auf die Entwicklung aktueller Sprachen.

6.2.3 Singleton

Das Entwurfsmuster *Singleton* sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globalen Zugriff auf dieses Objekt.

Wir können uns zahlreiche Anwendungsmöglichkeiten vorstellen. Beispielsweise soll in einem System nur ein Drucker-Spooler existieren. Eine einfache Lösung besteht in der Verwendung einer globalen Variable. Aber globale Variablen verhindern nicht, dass mehrere Objekte der Klasse erzeugt werden. Wir können die Klasse selbst für die Verwaltung ihres einzigen Objekts verantwortlich machen. Das ist die Aufgabe des Singleton-Patterns.

Dieses Entwurfsmuster ist anwendbar wenn

- es genau ein Objekt einer Klasse geben soll und dieses global zugreifbar sein soll;
- die Klasse durch Vererbung erweiterbar sein soll und Anwender die erweiterte Klasse ohne Änderungen verwenden können sollen.

Auf Grund der scheinbaren Einfachheit dieses Entwurfsmusters verzichten wir auf eine grafische Darstellung. Ein Singleton besteht nur aus einer gleichnamigen Klasse mit einer statischen Methode `instance`, die das einzige Objekt der

Klasse zurückgibt. Obwohl die Erklärung so einfach ist, sind einige Probleme bei der Implementation kaum zu lösen, weswegen heute oft von der Verwendung dieses Entwurfsmusters abgeraten wird. Konkret wird in abgewandelten Varianten häufig auf die Unterstützung von Vererbung verzichtet.

Singleton im ursprünglichen Sinn hat unter anderem folgende Eigenschaften:

- Der Zugriff auf das einzige Objekt kann kontrolliert werden.
- Durch Verzicht auf globale Variablen werden unnötige Namen und weitere unangenehme Eigenschaften globaler Variablen vermieden.
- Vererbung wird unterstützt (jedoch nicht in abgewandelten Varianten).
- Es wird verhindert, dass irgendwo Instanzen außerhalb der Kontrolle der Klasse erzeugt werden. Konstruktoren sind in der Regel nicht `public`.
- Prinzipiell sind auch mehrere Instanzen erzeugbar. Wir können die Entscheidung zugunsten nur eines Objekts im System jederzeit ändern und auch die Erzeugung mehrerer Objekte ermöglichen. Die Klasse hat weiterhin vollständige Kontrolle darüber, wie viele Objekte erzeugt werden.
- Auf das von `instance` zurückgegebene (nur einmal existierende) Objekt kann über Objektmethoden durch dynamisches Binden flexibler zugegriffen werden als wenn statt diesem Objekt eine Klasse mit statischen Methoden verwendet worden wäre.

Einfache Implementierungen dieses Entwurfsmusters bereiten keine Schwierigkeiten, wie folgendes Beispiel mit Initialisierung beim ersten Zugriff zeigt:

```
public class Singleton {
    private static Singleton singleton = null;
    private Singleton() {}  // no object creation from outside
    public static Singleton instance() {
        if (singleton == null)
            singleton = new Singleton();
        return singleton;
    }
}
```

Der leere `private` Konstruktor hat ausschließlich den Zweck zu verhindern, dass automatisch ein leerer `public` Konstruktor eingeführt wird.

Häufig ist es sinnvoll, mehrere Implementierungen eines Singletons (nur im ursprünglichen Sinn) zur Verfügung zu stellen. Das heißt, die Klasse `Singleton` hat Unterklassen. Beispielsweise gibt es mehrere Implementierungen für Drucker-Spooler, im System darf trotzdem immer nur ein Drucker-Spooler aktiv sein. Im Programm kann eine der Alternativen gewählt werden.

Überraschenderweise ist die Implementierung eines solchen Singletons recht schwierig. Die folgende Lösung, bei der nur der erste Aufruf von `instance` die zu verwendende Alternative wählt, ist noch am einfachsten, wenn auch wegen der unflexibel vorgegebenen Fallunterscheidung nicht zufriedenstellend:

```
public class Singleton {
    private static Singleton singleton = null;
    private Singleton() { ... }
    public static Singleton instance(int kind) {
        if (singleton == null)
            switch (kind) {
            case 1:  singleton = new SingletonA(); break
            case 2:  singleton = new SingletonB(); break
            default: singleton = new Singleton();
            }
        return singleton;
    }
}
public class SingletonA extends Singleton {
    private SingletonA() { ... }
}
public class SingletonB extends Singleton {
    private SingletonB() { ... }
}
```

Nach Erzeugung des Objekts hat `kind` keinerlei Bedeutung mehr, wodurch ein Aufrufer ein Objekt einer anderen Art zurückbekommen kann, als im Parameter angegeben. Würden wir das Objekt in der Klassenvariable `singleton` durch ein neues Objekt ersetzen, hätte ein früherer Aufrufer von `instance` ein anderes Objekt erhalten; das wäre kein Singleton.

Um die feste Verdrahtung der Alternativen in `Singleton` zu vermeiden, können wir `instance` in den Untertypen implementieren:

```
public class Singleton {
    protected static Singleton singleton = null;
    private Singleton() { ... }
    public static Singleton instance() {
        if(singleton==null) singleton = new Singleton();
        return singleton;
    }
}
public class SingletonA extends Singleton {
    private SingletonA() { ... }
    public static Singleton instance() {
        if(singleton==null) singleton = new SingletonA();
        return singleton;
    }
}
```

Die gewünschte Alternative ist wählbar, indem der erste Aufruf von `instance` in der entsprechenden Klasse durchgeführt wird. Alle weiteren Aufrufe geben stets das im ersten Aufruf erzeugte Objekt zurück. Jetzt ist nicht mehr die Klasse

`Singleton` alleine für die Existenz nur eines Objekts verantwortlich, sondern alle Unterklassen müssen mitspielen und `instance` passend implementieren.

Es gibt einige weitere Lösungen für dieses Problem, die aber alle ihre eigenen Nachteile haben. Daher wird Singleton heute kaum mehr in dieser Form eingesetzt. Stattdessen werden entsprechende Aufgaben meist ohne Untertypen gelöst, also das einzige Objekt einer Klasse über eine globale Variable bereitgestellt, in Java über eine Konstante (das ist eine `static final` Variable). Beispiele dazu haben wir etwa in Abschnitt 5.2.1 gesehen.

6.3 Entwurfsmuster für Struktur

Wir betrachten Decorator und Proxy als zwei einfache Vertreter häufig gebrauchter struktureller Entwurfsmuster, also solche, die die Programmstruktur beeinflussen. Diese Muster können ähnlich aufgebaut sein, unterscheiden sich aber in ihrer Verwendung und ihren Eigenschaften.

6.3.1 Decorator

Das Entwurfsmuster *Decorator*, auch *Wrapper* genannt, gibt Objekten dynamisch zusätzliche Verantwortlichkeiten (siehe Abschnitt 2.2.3). Decorators stellen eine flexible Alternative zur Vererbung bereit.

Manchmal möchten wir einzelnen Objekten zusätzliche Verantwortlichkeiten geben, nicht aber der ganzen Klasse. Zum Beispiel möchten wir einem Fenster am Bildschirm Bestandteile wie einen Scroll-Bar geben, anderen Fenstern aber nicht. Es ist sogar üblich, dass der Scroll-Bar dynamisch während der Verwendung eines Fensters nach Bedarf dazukommt und wieder weggenommen wird:

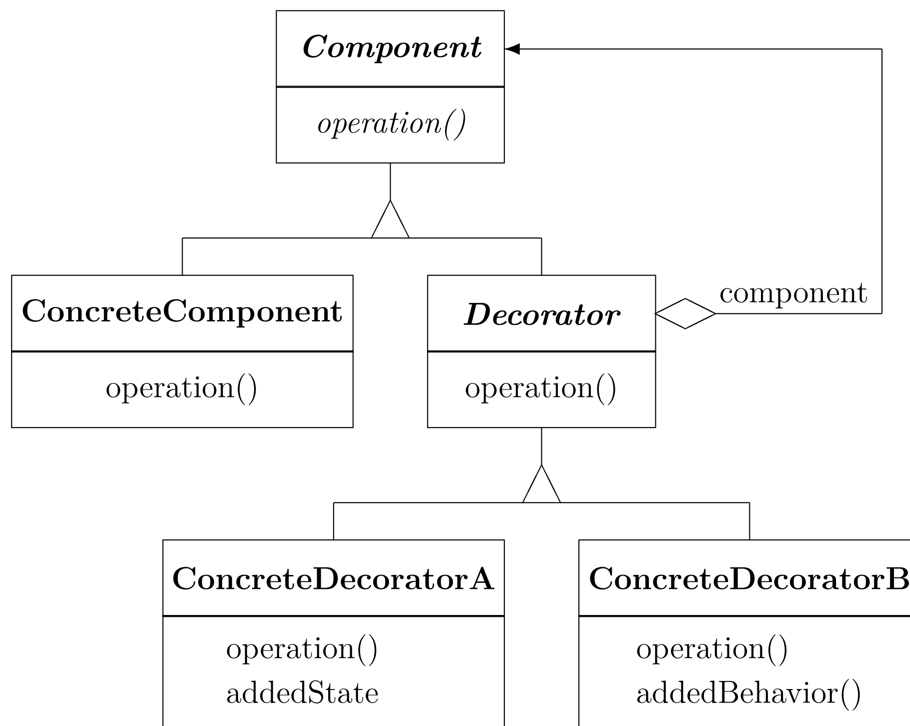
```
public interface Window {
    void show(String text);
}
public class WindowImpl implements Window {
    public void show(String text) { ... }
}
public abstract class WinDecorator implements Window {
    protected Window win;
    public void show(String text) { win.show(text); }
}
public class ScrollBar extends WinDecorator {
    public ScrollBar(Window w) { win = w; }
    public void scroll(int lines) { ... }
    public Window noScrollBar() {
        Window w = win;
        win = null;    // no longer usable
        return w;
    }
}
```

```
Window w = new WindowImpl();    // no scroll bar
ScrollBar s = new ScrollBar(w); // add scroll bar
w = s;             // s aware of scroll bar, w not
w.show("Text");    // no matter if scroll bar or not
s.scroll(3);       // works only with scroll bar
w = s.noScrollBar();            // remove scroll bar
```

Im Allgemeinen ist dieses Entwurfsmuster anwendbar

- um dynamisch Verantwortlichkeiten zu einzelnen Objekten hinzuzufügen, ohne andere Objekte dadurch zu beeinflussen;
- für Verantwortlichkeiten, die wieder entzogen werden können;
- wenn Erweiterungen einer Klasse durch Vererbung unpraktisch sind, beispielsweise um eine sehr große Zahl an Unterklassen zu vermeiden, oder weil die Programmiersprache oder ein Programm in einem speziellen Fall keine Vererbung unterstützt (etwa bei `final` Klassen).

Das Entwurfsmuster hat folgende Struktur, wobei der Pfeil mit einem Kästchen für Aggregation steht (eine Referenz auf ein Objekt, dessen Bestandteil das die Referenz enthaltende Objekt ist).



Die abstrakte Klasse bzw. das Interface *Component* (entspricht `Window`) definiert eine Schnittstelle für Objekte, an die Verantwortlichkeiten dynamisch hinzugefügt werden können. Die Klasse „ConcreteComponent“ ist, wie beispielsweise `WindowImpl`, eine konkrete Unterklasse davon. Die (abstrakte) Klasse *Decorator* (`WinDecorator` im Beispiel) definiert eine Schnittstelle für Verantwortlichkeiten, die dynamisch zu Komponenten hinzugefügt werden können. Jedes

Objekt dieses Typs enthält eine Referenz namens „component“ (bzw. `win` im Beispiel) auf ein Objekt des Typs *Component*, das ist das Objekt, zu dem die Verantwortlichkeit hinzugefügt ist. Unterklassen von *Decorator* sind konkrete Klassen, die bestimmte Funktionalität wie beispielsweise Scroll-Bars bereitstellen. Sie definieren neben den Methoden, die bereits in *Component* definiert sind, weitere Methoden und Variablen, welche die zusätzliche Funktionalität verfügbar machen. Wird eine Methode, die in *Component* definiert ist, aufgerufen, so wird dieser Aufruf einfach an das Objekt, das über „component“ referenziert ist, weitergegeben.

Decorators haben folgende Eigenschaften:

- Sie bieten mehr Flexibilität als statische Vererbung. Wie bei statischer Erweiterung einer Klasse durch Vererbung werden Verantwortlichkeiten hinzugefügt. Anders als bei Vererbung erfolgt das Hinzufügen der Verantwortlichkeiten zur Laufzeit und zu einzelnen Objekten, nicht ganzen Klassen. Die Verantwortlichkeiten können auch jederzeit wieder weggenommen werden.

- Sie vermeiden Klassen, die bereits weit oben in der Typhierarchie mit Methoden und Variablen überladen sind. Es ist nicht notwendig, dass „ConcreteComponent“ die volle gewünschte Funktionalität enthält, da durch das Hinzufügen von Dekoratoren gezielt neue Funktionalität verfügbar gemacht werden kann.

- Objekte von *Decorator* und die dazugehörenden Objekte von „ConcreteComponent“ sind nicht identisch. Beispielsweise hat ein Fenster-Objekt, auf das über einen Dekorator zugegriffen wird, eine andere Identität als das Fenster-Objekt selbst (ohne Dekorator) oder dasselbe Fenster-Objekt, auf das über einen anderen Dekorator zugegriffen wird. Bei Verwendung dieses Entwurfsmusters sollen wir uns nicht auf Objektidentität verlassen.

- Sie führen zu vielen kleinen Objekten. Ein Design, das Dekoratoren häufig verwendet, führt nicht selten zu einem System, in dem es viele kleine Objekte gibt, die einander ähneln. Solche Systeme sind zwar einfach konfigurierbar, aber schwer zu verstehen und zu warten.

Wenn es nur eine Dekorator-Klasse gibt, kann die abstrakte Klasse *Decorator* weglassen und statt dessen die konkrete Klasse verwendet werden. Bei mehreren Dekorator-Klassen zahlt sich die abstrakte Klasse aus: Alle Methoden, die bereits in *Component* definiert sind, müssen in den Dekorator-Klassen auf gleiche Weise überschrieben werden. Sie rufen einfach dieselbe Methode in „component“ auf. Man muss diese Methoden nur einmal in der abstrakten Klasse überschreiben. Von den konkreten Klassen werden sie geerbt.

Die Klasse oder das Interface „Component“ soll so klein wie möglich gehalten werden. Dies kann dadurch erreicht werden, dass „Component“ wirklich nur die notwendigen Operationen, aber keine Daten definiert. Daten und Implementierungsdetails sollen erst in „ConcreteComponent“ vorkommen. Andernfalls werden Dekoratoren umfangreich und ineffizient.

Dekoratoren eignen sich gut dazu, die Oberfläche beziehungsweise das Erscheinungsbild eines Objekts zu erweitern. Sie sind nicht gut für inhaltliche Erweiterungen geeignet. Auch für Objekte, die von Grund auf umfangreich sind, eignen sich Dekoratoren kaum. Für solche Objekte sind andere Entwurfsmuster, beispielsweise *Strategy*, besser geeignet. Auf diese Entwurfsmuster wollen wir hier aber nicht eingehen.

6.3.2 Proxy

Ein *Proxy*, auch *Surrogate* genannt, stellt einen Platzhalter für ein anderes Objekt dar und kontrolliert Zugriffe darauf.

Es gibt zahlreiche, sehr unterschiedliche Anwendungsmöglichkeiten für Platzhalterobjekte. Ein Beispiel ist ein Objekt, dessen Erzeugung teuer ist, weil umfangreiche Daten geladen werden. Wir erzeugen das eigentliche Objekt erst, wenn es wirklich gebraucht wird. Stattdessen verwenden wir in der Zwischenzeit einen Platzhalter, der erst bei Bedarf durch das eigentliche Objekt ersetzt wird. Falls nie auf die Daten zugegriffen wird, ersparen wir uns den Aufwand der Objekterzeugung:

```
public interface Something {
    void doSomething();
}
public class ExpensiveSomething implements Something {
    public void doSomething() { ... }
}
public class VirtualSomething implements Something {
    private ExpensiveSomething real = null;
    public void doSomething() {
        if (real == null)
            real = new ExpensiveSomething();
        real.doSomething();
    }
}
```

Jedes Platzhalterobjekt enthält einen Zeiger auf das eigentliche Objekt (sofern dieses existiert) und leitet in der Regel Nachrichten an das eigentliche Objekt weiter, möglicherweise nachdem weitere Aktionen gesetzt wurden. Einige Nachrichten werden manchmal auch direkt vom Proxy behandelt.

Das Entwurfsmuster ist anwendbar, wenn eine intelligentere Referenz auf ein Objekt als ein simpler Zeiger nötig ist. Hier sind einige übliche Situationen, in denen ein Proxy eingesetzt werden kann (keine vollständige Aufzählung):

Remote-Proxies sind Platzhalter für Objekte, die in anderen Namensräumen (zum Beispiel auf Festplatten oder auf anderen Rechnern) existieren. Nachrichten an die Objekte werden von den Proxies über komplexere Kommunikationskanäle weitergeleitet.

Virtual-Proxies erzeugen Objekte bei Bedarf, wie in obigem Beispiel. Da die Erzeugung eines Objekts aufwendig sein kann, wird sie so lange hinausgezögert, bis es wirklich einen Bedarf dafür gibt.

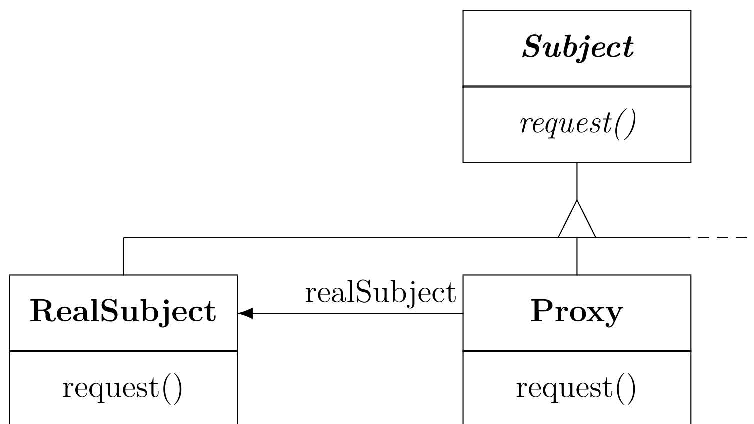
Protection-Proxies kontrollieren Zugriffe auf Objekte. Derartige Proxies sind sinnvoll, wenn Objekte je nach Zugreifer oder Situation unterschiedliche Zugriffsrechte haben sollen.

Smart-References ersetzen einfache Zeiger. Sie können bei Zugriffen zusätzliche Aktionen ausführen. Typische Verwendungen sind

- das Mitzählen der Referenzen auf das eigentliche Objekt, damit das Objekt entfernt werden kann, wenn es keine Referenz mehr darauf gibt (Reference-Counting);
- das Laden von persistenten Objekten in den Speicher, wenn das erste Mal darauf zugegriffen wird (wobei die Unterscheidung zu Virtual-Proxies manchmal unklar ist);
- das Zusichern, dass während des Zugriffs auf das Objekt kein gleichzeitiger Zugriff durch einen anderen Thread erfolgt (beispielsweise durch Setzen eines „Locks“).

Es gibt zahlreiche weitere Einsatzmöglichkeiten. Der Phantasie sind hier kaum Grenzen gesetzt.

Die Struktur dieses Entwurfsmusters ist recht einfach:



Die abstrakte Klasse oder das Interface *Subject* ist eine gemeinsame Schnittstelle für Objekte von „RealSubject“ und „Proxy“. Objekte von „RealSubject“ und „Proxy“ können gleichermaßen verwendet werden, wo ein Objekt von *Subject* erwartet wird. Die Klasse „RealSubject“ definiert die eigentlichen Objekte, die durch die Proxies (Platzhalter) repräsentiert werden. Die Klasse „Proxy“ definiert schließlich die Proxies. Diese Klasse

- verwaltet eine Referenz „realSubject“, über die ein Proxy auf Objekte von „RealSubject“ (oder auch andere Objekte von *Subject*) zugreifen kann;
- stellt eine Schnittstelle bereit, die der von *Subject* entspricht, damit ein Proxy als Ersatz des eigentlichen Objekts verwendet werden kann;

- kontrolliert Zugriffe auf das eigentliche Objekt und kann für dessen Erzeugung oder Entfernung verantwortlich sein;
- hat weitere Verantwortlichkeiten, die von der Art abhängen.

Es kann mehrere unterschiedliche Klassen für Proxies geben. Zugriffe auf Objekte von „RealSubject“ können durch mehrere Proxies (möglicherweise unterschiedlicher Typen) kontrolliert werden, die in Form einer Kette bzw. Liste miteinander verbunden sind.

In obiger Grafik zur Struktur des Entwurfsmusters zeigt ein Pfeil von „Proxy“ auf „RealSubject“. Das bedeutet, „Proxy“ muss „RealSubject“ kennen. Dies ist z. B. notwendig, wenn ein Proxy Objekte von „RealSubject“ erzeugen soll. In anderen Fällen reicht es, wenn „Proxy“ nur *Subject* kennt, der Pfeil also auf *Subject* zeigt.

In der Implementierung müssen wir beachten, wie ein Objekt referenziert wird, das in einem anderen Namensraum liegt oder noch gar nicht existiert. Für nicht existierende Objekte könnten wir zum Beispiel `null` verwenden und für Objekte in einer Datei den Dateinamen.

Ein Proxy kann die gleiche Struktur wie ein Decorator haben. Aber Proxies dienen einem ganz anderen Zweck als Decorators: Ein Decorator erweitert ein Objekt um zusätzliche Verantwortlichkeiten, während ein Proxy den Zugriff auf das Objekt kontrolliert. Damit haben diese Entwurfsmuster auch gänzlich unterschiedliche Eigenschaften.

6.4 Entscheidungshilfen

Schon zu Beginn dieses Kapitels haben wir festgehalten, dass Entwurfsmuster nicht als Regeln missverstanden werden dürfen, an die wir uns halten müssen oder sollten. Sie sind keine Regeln, zumindest nicht in dem Sinn, wie wir in vielen Teilen des Skriptums Faustregeln gesehen haben. Trotzdem helfen sie uns, Entscheidungen zu treffen. Zunächst betrachten wir, wie wir Entscheidungen auf der Basis von Entwurfsmustern treffen können. Danach beschäftigen wir uns ganz allgemein damit, wie die Einhaltung von Richtlinien oder Faustregeln in der Programmierung zu verstehen ist.

6.4.1 Entwurfsmuster als Entscheidungshilfen

Seit der Entwicklung der Software-Entwurfsmuster sind schon einige Jahrzehnte vergangen, wobei die Idee der Entwurfsmuster aus dem Bereich der Architektur (Bauwesen, nicht Computer-Architektur) stammt und wesentlich älter ist. Der Begriff Software-Entwurfsmuster hatte in der Informatik von Anfang an einen positiven Beigeschmack, was dazu führte, dass er für Unterschiedliches eingesetzt wurde. Im Wesentlichen lassen sich zwei Sichtweisen unterscheiden:

Wertend: Als Entwurfsmuster werden nur solche abstrakte Konzepte bezeichnet, die überwiegend positive Eigenschaften in ein Programm bringen.

Dementsprechend wird es als vorteilhaft angesehen, möglichst viele Entwurfsmuster in einem Programm vorzufinden. Natürlich ist nicht jedes Konzept gut. Zur Unterscheidung der schlechten von den guten Konzepten wurde der Begriff *Anti-Pattern* eingeführt. Wir wollen im Programm möglichst wenige Anti-Pattern finden.

Nicht wertend: Entwurfsmuster sind für sich (also außerhalb des Kontexts eines bestimmten Programms) weder gut noch schlecht. Es handelt sich nur um eine benannte Beschreibung eines Problems, eines Lösungsansatzes und von Konsequenzen, die daraus folgen. Fast jedes Entwurfsmuster wird sowohl erwünschte als auch unerwünschte Konsequenzen haben. Ob der Einsatz eines Entwurfsmusters insgesamt als vorteilhaft angesehen wird oder nicht, hängt vor allem von den Gewichtungen ab, die wir den Konsequenzen in einer bestimmten Anwendung beimessen. Begriffe wie Anti-Pattern ergeben kaum Sinn, abgesehen vielleicht für die ganz wenigen Muster, die eindeutig nur negative Eigenschaften haben.

Es wird selten klar gesagt, ob der Begriff Entwurfsmuster wertend oder nicht wertend gebraucht wird. Meist steht eine der beiden Sichtweisen im Vordergrund, aber die andere schwingt auch mit. Wenn wir ein Entwurfsmuster betrachten, besonders wenn es sich um eine Kurzbeschreibung auf einer Webseite handelt, müssen wir darauf achten, welche Sichtweise wahrscheinlich dahinter steckt. Viele Leute bevorzugen kurze Beschreibungen mit möglichst einfachen Aussagen dazu, ob ein Muster verwendet werden soll oder nicht. Deswegen finden Suchmaschinen häufig vereinfachte Beschreibungen, die stark wertend sind. Oft werden nur wenige Konsequenzen als „Vorteile“ und „Nachteile“ angerissen, ohne die Frage zu stellen, welche Kriterien für die beabsichtigte Anwendung relevant sind. Darin liegt eine Gefahr, die wir nicht unterschätzen sollten.

Aus dem Einsatz von Entwurfsmustern in der Achitektur ist bekannt, dass Ergebnisse selten gut sind. Auf der Basis von Entwurfsmustern entwickelte Bauten sind vergleichsweise komplex und wenig ansprechend, sie wirken ideenlos. Letzteres trifft genau den Punkt: Es steckt keine Idee dahinter, sondern nur das mehr oder weniger willkürliche Zusammenwürfeln fertiger Muster. In der Programmierung wurden ähnliche Erfahrungen gemacht: Aus Entwurfsmustern zusammengesetzte Programme sind umfangreich, haben keine klare Struktur und aus den Beschreibungen der Entwurfsmuster ableitbare Eigenschaften sind selten wirklich gegeben. Das Wesentliche (die Idee) scheint zu fehlen.

Hinter ideenlosen Programmen kann ein falscher Ansatz im Umgang mit Entwurfsmustern stecken. Entwurfsmuster können die Idee nicht ersetzen, sie können nur helfen, die Idee zu beurteilen. Erst die Idee lässt die Einzelteile (auch die Entwurfsmuster) so miteinander in Beziehung treten, dass ein in sich konsistentes Gesamtgefüge entsteht. Mit einer guten Idee ergibt es sich nicht selten, dass in einem kurzen, einfachen Programmstück mehrere typische Entwurfsmuster stecken, obwohl das bei der Entwicklung nicht beabsichtigt war. Die Entwurfsmuster ergeben sich wie von selbst und sind komplex ineinander verstrickt; ohne Idee liegen die Entwurfsmuster quasi flach nebeneinander, ohne

sich zu überschneiden. Wenn die Entwurfsmuster sich nicht überschneiden, können wir aus Konsequenzen aus einzelnen Mustern nur wenig Rückschlüsse auf die Qualität des gesamten Programms ziehen. Bei komplex ineinander verstrickten Entwurfsmustern sind auch die Konsequenzen eng ineinander verzahnt, wodurch die Programmqualität selbst (bzw. die Qualität der Idee) sich in den Konsequenzen widerspiegelt. Wir müssen aufpassen, wenn wir von der Eigenschaft „komplex" sprechen: Gerade bei einfachen Ideen und entsprechend einfachen Programmen sind Entwurfsmuster komplex ineinander verstrickt; wenn Entwurfsmuster nicht komplex ineinander verstrickt sind, kann das Programm im Vergleich dazu ziemlich komplex und umfangreich sein, weil viel Code nötig ist, um die unabhängigen Muster doch irgendwie miteinander zu kombinieren. Entwurfsmuster, die komplex ineinander verzahnt sind, müssen wir fast zwangsläufig nicht wertend betrachten; das heißt, wir dürfen nicht die Anwesenheit eines Musters selbst als Qualitätsmerkmal betrachten, sondern wir müssen die gewichteten Konsequenzen als Basis für die Beurteilung heranziehen. Ohne Verzahnung der Muster ist es egal, ob wir sie als wertend oder nicht wertend betrachten, weil ohnehin kaum Rückschlüsse auf die Qualität möglich sind.

Aus obigen Überlegungen folgt, wie eine erfolgreiche Herangehensweise an den Entwurf einer Software aussehen könnte. Wir setzen voraus, dass wir den Bedarf schon genau analysiert haben, also wissen, welche Anforderungen an die Software gestellt werden. Jetzt brauchen wir die entscheidende Idee. Das heißt, mit den Anforderungen im Blick müssen wir eine Organisationsform des gesamten Programms finden, die eine einheitliche gemeinsame Klammer über alle zu entwickelnden Programmteile stülpt. Diese Klammer oder Grobstruktur muss möglichst einfach und ganz auf die Gesamtheit der Anforderungen ausgerichtet sein, darf also nicht einzelne Anforderungen stark überbewerten oder vergessen. Die Gesamtheit der Anforderungen ist mehr als die Summe der Anforderungen. Wir brauchen ein intuitives Verständnis für die gemeinsamen typischen Merkmale aller Anforderungen, die die zu entwickelnde Software von anderer Software abhebt und ihr einen eigenen Charakter verleiht. Wir können nicht allgemein sagen, worin die Gemeinsamkeiten in den Anforderungen bestehen, weil das von Projekt zu Projekt sehr verschieden ist. Es kann ein bestimmtes Bedienkonzept sein, ein bestimmer Umgang mit Daten, ein bestimmter Programmierstil, was auch immer. Hier ist Kreativität und Intuition gefragt, Intuition, die aus der Erfahrung kommt. Damit kommen indirekt Entwurfsmuster ins Spiel, weil Entwurfsmuster uns erlauben, Erfahrungen auszutauschen und auf eine bewusste Ebene zu ziehen. Jetzt dürfen wir aber nicht den Fehler machen, den Entwurf bzw. die Idee auf eine zu kleine Menge an Entwurfsmustern auszurichten und uns zu früh willkürlich auf bestimmte Muster festzulegen. Vielmehr müssen wir uns darauf verlassen, dass wir uns durch die intensive Beschäftigung mit Software-Entwürfen und Entwurfsmustern eine ausreichend gute Intuition angeeignet haben. Indirekt bedeutet das, dass unsere Intuition aus einer Vielzahl von Entwurfsmustern kommt (nicht nur den wenigen, die häufig öffentlich diskutiert werden), ohne uns dessen bewusst zu sein. Mit einer guten Idee wissen wir, wie das Projekt zu einem guten Abschluss geführt werden kann. Ohne Idee tappen wir hilflos im Dunkeln und müssen weiter nach einer Idee suchen.

Eine Idee ist nur ein erster Schritt. Um sicher zu gehen, werden wir mehrere alternative Ideen entwickeln und miteinander vergleichen. Erst hier kommen Entwurfsmuster nicht nur unbewusst ins Spiel, sondern werden ganz gezielt eingesetzt. Mit einer konkreten Idee im Hinterkopf können wir erkennen, welche uns bekannten (nicht wertenden) Entwurfsmuster in einem entsprechenden Programm stecken werden. Ebenso auf der Idee und den Anforderungen beruhend können wir entscheiden, wie relevant bestimmte Kriterien für uns am Ende sein werden, wir können die Konsequenzen aus den Entwurfsmustern also gewichten. Das gibt uns ein Werkzeug für die Beurteilung verschiedener Ideen. Damit alleine ist es noch nicht getan, weil sich, wie oben beschrieben, Konsequenzen aus einzelnen Entwurfsmustern nicht immer problemlos auf die Qualität des gesamten Programms übertragen lassen. Es gibt ein Beurteilungskriterium, das alle anderen Kriterien überwiegt, nämlich die erwartete Gesamtkomplexität des Programms. Eine Idee, die zu einem einfacheren Programm führt, ist fast immer vorteilhaft, egal was eine Beurteilung auf Basis von Entwurfsmustern ergibt. Vergleiche von Ideen über Entwurfsmuster sind also in der Regel nur dann sinnvoll, wenn die Ideen zu einigermaßen gleich geringer Komplexität führen.

Wie oben argumentiert, beruhen unsere Ideen unbewusst meist auch auf Erfahrungen, die von Entwurfsmustern kommen. Es spricht nichts dagegen, diese unbewussten Abhängigkeiten auf eine bewusste Ebene zu ziehen. Wir wissen, welche Anwendungen von Entwurfsmustern mit vergleichsweise hoher Wahrscheinlichkeit zu schlechter Software führen, nämlich solche, die nicht auf natürliche Weise gut in das gesamte Programm eingebunden sind, sondern über speziellen Code aufwendig eingebunden werden müssen. Wenn uns eine solche Situation auffällt, können wir bewusst etwas dagegen tun. Wir müssen unsere Idee weiterentwickeln, sodass dieses Problem nicht mehr auftritt. Das heißt, wir müssen auf das Entwurfsmuster verzichten oder einen Weg (durch Überarbeiten der Idee) finden, wie wir es auf natürliche Weise integrieren können.

Entwurfsmuster können also sehr wohl das sein, was der Name besagt: ein Werkzeug, um uns beim Entwurf eines Systens zu leiten. Der richtige Umgang damit will jedoch geübt sein. Es reicht auf keinen Fall, ein Stück Software nur durch additives Kombinieren einiger uns sinnvoll erscheinender Entwurfsmuster zu entwerfen. Ohne zielführende Idee als gemeinsame Klammer über dem Entwurf könnte ein solches Vorgehen in einer Katastrophe enden. Ein bewusster Umgang mit Entwurfsmustern kann helfen, eine gute Idee zu finden. Allerdings ist es dafür nicht ausreichend, Entwurfsmuster nur oberflächlich zu kennen. Sie müssen tief in unserem Unterbewusstsein verankert sein, bevor wir Sie wirklich produktiv einsetzen können.

6.4.2 Richtlinien in der Entscheidungsfindung

Wir kennen eine große Zahl unterschiedlicher Regeln, Konventionen und Richtlinien, die darauf abzielen, uns beim Entwurf von Programmen zu unterstützen. Hier ist eine (unvollständige) Zusammenstellung, jeweils mit den wichtigsten Zielsetzungen und Konsequenzen bei Verstößen dagegen:

- Syntax- und Semantikregeln einer Programmiersprache sind mehr oder weniger standardisiert und sorgen dafür, dass alle dazugehörigen Werkzeuge Programme auf die gleiche Weise verstehen. Natürlich müssen auch wir uns beim Programmieren an diese Regeln halten; eine Regelverletzung führt meist zu einem nicht lauffähigen Programm oder schweren Fehlern bei der Programmausführung.

- Allgemeine Konventionen wie z. B. Namenskonventionen helfen uns beim Lesen von Programmen, können aber auch dazu dienen, die Vertrauenswürdigkeit von Programmtexten abzuschätzen. Die Nichteinhaltung der Konventionen kann durch verminderte Lesbarkeit die Wartung erschweren und vereinzelt durch Missverständnisse zu Fehlern führen. Schwerwiegendere Folgen kann durch Nichteinhaltung entstehendes mangelndes Vertrauen haben, das zu zusätzlichen Programmtexten für Überprüfungen oder als Alternativen zu vorhandenen Programmtexten führen kann.

- Konventionen, die für Projekte, Firmen, Frameworks oder Bibliotheken spezifisch sind, haben verschiedenartige Hintergründe mit unterschiedlichen Auswirkungen bei Nichteinhaltung. Einerseits sind Ziele und Auswirkungen ähnlich denen allgemeiner Konventionen, wobei neben der Lesbarkeit häufig auch die Teambildung eine Rolle spielt. Es gibt auch Konventionen, deren Einhaltung für die Funktionalität der Software entscheidend ist, deren Nichteinhaltung zu schwerwiegenden Fehlern führen kann. Insbesondere bei Konventionen im Zusammenhang mit Annotationen, Reflexion und aspektorientierter Programmierung ist das der Fall.

- Wir haben zahlreiche Faustregeln angeführt, eine Internet-Recherche wird uns eine schier endlose Liste an Faustregeln bescheren. Faustregeln sollen uns helfen, bei der Wahl zwischen verschiedenen Optionen in bestimmten Situationen jene zu finden, die am erfolgversprechendsten ist. Es gibt keinerlei Garantie, dabei die beste oder auch nur eine gute Wahl zu treffen. Mangels ausreichender Information über die Konsequenzen einer Entscheidung zum Zeitpunkt der Entscheidungsfindung verlassen wir uns dennoch gerne auf Faustregeln. Unter der Voraussetzung, dass Faustregeln gut gewählt sind, laufen wir Gefahr, bei Nichtbefolgung in vielen Entscheidungen deutlich mehr Fehlentscheidungen zu treffen als bei Befolgung.

- Softwareentwurfsmuster geben uns ein Werkzeug in die Hand, das es uns erlaubt, bei der Wahl zwischen verschiedenen Optionen in bestimmten Situationen die Konsequenzen einer Entscheidung zum Zeitpunkt der Entscheidungsfindung mit einer relevanten Wahrscheinlichkeit richtig vorherzusehen. Wie Faustregeln bieten auch Softwareentwurfsmuster keine zuverlässigen Aussagen, sondern nur zufallsbehaftete Entscheidungsgrundlagen. Im Gegensatz zu Faustregeln ermöglichen Softwareentwurfsmuster viel genauere Abschätzungen möglicher Konsequenzen und erlauben uns, die Wichtigkeit einzelner Kriterien für die zu entwickelnde Software in die Entscheidungsfindung einzubeziehen.

- Eine nicht zu unterschätzende Grundlage für eine Entscheidungsfindung ist die persönliche Expertise (bzw. die Summe gesammelter Erfahrungen). Auf den ersten Blick ist Expertise nicht sofort als Menge von Regeln erkennbar, dennoch hängen die meisten von uns getroffenen Entscheidungen davon ab. Im Gegensatz zu Faustregeln und Softwareentwurfsmustern können wir unsere Expertise meist nicht in ausreichender Qualität in Form von Regeln ausformulieren. Dennoch steckt dahinter viel Wissen, das auf Wahrscheinlichkeiten beruht. Mit ausreichend Erfahrung können wir uns häufig auf unsere Expertise verlassen und Entscheidungen rasch und dennoch mit hoher Wahrscheinlichkeit richtig treffen, aber es werden (auch mit sehr viel Erfahrung) immer auch einige Entscheidungen dabei sein, die eindeutig falsch sind.

Es stellt sich die Frage, wie wir mit diesen vielen Regeln, Konventionen und Richtlinien umgehen sollen, insbesondere dann, wenn sie sich teilweise widersprechen. Bei Syntax- und Semantikregeln, aber auch bei Konventionen, egal ob allgemein oder spezifisch, sollte die Antwort klar sein: Wir müssen uns ganz genau daran halten. Diese Regeln und Konventionen sind so gestaltet, dass sie sich nicht widersprechen, sodass die Befolgung immer möglich sein sollte. Auch wenn wir oft nicht wissen, welche Überlegung hinter einer Regel oder Konvention steckt, können wir davon ausgehen, dass sich das jemand genau überlegt hat. Die Nichteinhaltung kann nur negative Konsequenzen nach sich ziehen, keine positiven.

Faustregeln, Entwurfsmuster und persönliche Expertisen sind Syntax- und Semantikregeln sowie Konventionen unterzuordnen, da sie auf Wahrscheinlichkeiten des Eintritts bestimmter Situationen beruhen, meist weit weg von Gewissheit. Dennoch sind sie wertvoll. Faustregeln und Softwareentwurfsmuster spiegeln allgemeine Erfahrungen wider, Expteritsen persönliche Erfahrungen. Faustreglen und Softwareentwurfsmuster müssen von uns internalisiert werden, bevor sie als unsere persönlichen Expertisen effizient als Entscheidungsgrundlagen herangezogen werden. Faustregeln und Softwareentwurfsmuster dienen aber auch dazu, eigene Expertisen ständig zu hinterfragen und darauf hin zu überprüfen, wie gut sie in sich konsistent sind.

Verschiedene Faustregeln, Entwurfsmuster und persönliche Expertisen können sich gegenseitig widersprechen. Entscheidungsprozesse sind daher komplexer als das Befolgen einer großen Menge an Regeln. Letztendlich ist immer die persönliche Expertise ausschlaggebend für eine Entscheidung.

Programmierparadigmen und persönliche Programmierstile spielen eine wesentliche Rolle. Programmierparadigmen können wir als großteils in sich konsistente Mengen an Faustregeln betrachten, persönliche Programmierstile als die von Paradigmen beieinflussten persönlichen Expertisen. Jede Person hat aber nur eine Menge an persönlichen Expertisen, die Summe aller Expertisen in vielen Bereichen. Trotzdem kann eine Person einmal in einem Stil (nach einem Paradigma) programmieren, dann wieder in einem anderen. Das widerspricht sich nicht. Die persönliche Expertise umfasst auch die Fähigkeit, zwischen unterschiedlichen Paradigmen umzuschalten. Wir sind in der Lage, zu erkennen,

wann welches Paradigma den größten Erfolg verspricht. Das ist wesentlich mehr als nur die Summe der Erfahrungen in einzelnen Bereichen. Die Lehrveranstaltung „Programmierparadigmen“ zielt(e) darauf ab, die persönliche Expertise auf dem Gebiet der Programmierung so aufzubauen, dass bestimmte Paradigmen und Stile zur Lösung einer Programmieraufgabe ganz bewusst gewählt werden können, nicht nur nach dem Zufallsprinzip.

Literaturverzeichnis

[1] Martin Abadi and Luca Cardelli. On subtyping and matching. *ACM Transactions on Programming Languages and Systems*, 18(4):401–423, July 1996.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[4] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the 18th Symposium on Principles of Programming Languages*, pages 104–118. ACM, 1991.

[5] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings, Redwood City, California, second edition, 1994.

[6] P. Brinch-Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[7] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89 Proceedings of the fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, New York, USA, 1989. ACM.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[9] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, June 1992. Springer.

[10] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer, Berlin, 2003.

[11] Neal Ford. *Functional Thinking. Paradigm over syntax*. O'Reilly, 2014.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[13] Brian Goetz. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[14] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[15] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[16] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, 1969.

[17] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Information and Computation*, 175(1):34–49, 2002.

[18] ISO/IEC 8652:1995. Annotated ada reference manual. Intermetrics, Inc., 1995.

[19] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA, 1989.

[20] B.B. Kristensen, O.L. Madsen, B. Moller-Pedersen, and K. Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[21] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications, second edition, 2009.

[22] Wilf LaLonde and John Pugh. Subclassing ≠ subtyping ≠ is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, 1991.

[23] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison-Wesley, Reading, MA, 1997.

[24] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. Proceedings OOPSLA'93.

[25] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[26] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, editor, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.

[27] John McCarthy. History of LISP. *ACM SIGPLAN Notices*, 13(9):217–223, 1978.

[28] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[29] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition edition, 1997.

[30] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.

[31] John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.

[32] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 1965.

[33] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for Self. In *ECOOP '95 Conference Proceedings*, Aarhus, Denmark, August 1995.

[34] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227–241, Orlando, FL, October 1987.

[35] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.